Goodman/Hsu Integrated Code Scheduler

Prepass Schedulers:

Schedule code prior to register allocation.

Can overuse registers—Always using a "fresh" register maximizes freedom to rearrange Instructions.

Postpass Schedulers:

Schedule code after register allocation.

Can be limited by "false dependencies" induced by register reuse.

Example is Gibbons/Muchnick heuristic.

Integrated Schedulers

Capture best of both approaches.

When registers are plentiful, use additional registers to avoid stalls.

Goodman & Hsu call this *CSP*:
Code Scheduling for Pipelines.

When registers are scarce, switch to a policy that frees registers.

Goodman & Hsu call this *CSR*:
Code Scheduling to free Registers.

Assume code is generated in single assignment form, with a unique pseudo-register for each computed value.

We schedule from a DAG where nodes are operations (to be mapped to instructions), and arcs represent data dependencies.

Each node will have an associated Cost, that measures the execution and stall time of the instruction that the node represents.

Nodes are labeled with a critical path cost, used to select the "most critical" instructions to schedule.

Definitions

Leader Set:

Set of DAG nodes ready to be scheduled, possibly with an interlock.

Ready Set:

Subset of Leader Set; Nodes ready to be scheduled without an interlock.

AvailReg:

A count of currently unused registers.

MinThreshold:

Threshold at which heuristic will switch from avoiding interlocks to reducing registers in use.

Goodman/Hsu Heuristic:

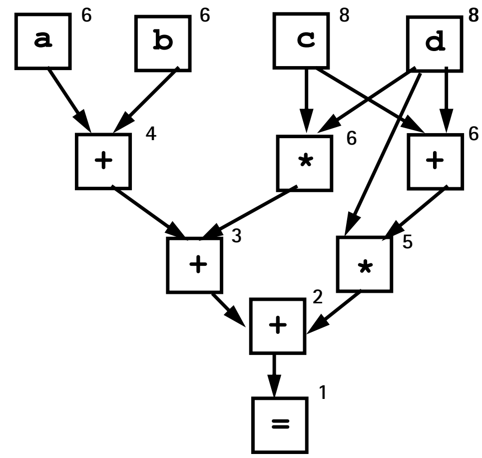
```
while (DAG ≠ ϕ) {
  if ( AvailReg > MinThreshold)
     if (ReadySet ≠ ϕ)
         Select Ready node with Max cost
     else Select Leader node with Max cost
  else  //  Reduce Registers in Use
     if (∃ node ∈ ReadySet that frees registers){
       Select node that frees most registers
       If (selected node isn't unique)
          Select node with Max cost   }
    elsif (∃ node ∈ LeaderSet that frees regs){
           Select node that frees most registers
          If (selected node isn't unique)
             Select node with fewest interlocks}
    else find a partially evaluated path and
          select a leader from this path
    else Select any node in ReadySet
    else Select any node in LeaderSet
Schedule Selected node
Update AvailReg count  }//end while
```


Example

We'll again consider

`a = ((a+b) + (c*d)) + ((c+d) * d);`

Again, assume a 1 cycle stall between a load and use of its value and a 2 cycle stall between a multiplication and first use of the product.



We'll try 4 registers (the minimum), then 5 registers.

Should MinThreshold be 0 or 1?

At MinThreshold = 1, we always have a register to hold a result, but we may force a register to be spilled too soon!

At MinThreshold = 0, we may be forced to spill a register to free a result register.

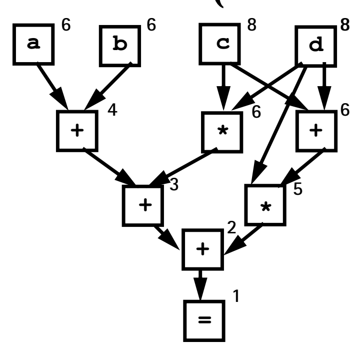
But, we'll also be able to schedule more aggressively.

Is a spill or stall worse?

Note that we may be able to "hide" a spill in a delay slot!

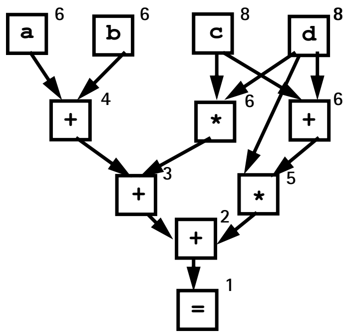
We'll be aggressive and use MinThreshold = 0.

4 Registers Used (1 Stall)



Instruction	Comment	Regs Used
`ld [c], %r1`	Choose ready, cost=8	1
`ld [d], %r2`	Choose ready, cost=8	2
`ld [a], %r3`	Choose ready, cost=6	3
`smul %r1,%r2,%r4`	Choose ready, cost=6	4
`add %r1,%r2,%r1`	Free a register	4
`smul %r1,%r2,%r1`	Free a register	3
`ld [b], %r2`	Choose ready, cost=6	4
`add %r3,%r2,%r3` ←	Choose a leader	3
`add %r3,%r4,%r3`	No choice	2
`add %r3,%r1,%r3`	No choice	1
`st %r3,[a]`	No choice	0

5 Registers Used (No Stalls)



Instruction	Comment	Regs Used
`ld [c], %r1`	Choose ready, cost=8	1
`ld [d], %r2`	Choose ready, cost=8	2
`ld [a], %r3`	Choose ready, cost=6	3
`smul %r1,%r2,%r4`	Choose ready, cost=6	4
`add %r1,%r2,%r1`	Choose ready, cost=6	4
`ld [b], %r5`	Choose ready, cost=6	5
`smul %r1,%r2,%r1`	Free a register	4
`add %r3,%r5,%r3`	Choose ready, cost=4	3
`add %r3,%r4,%r3`	No choice	2
`add %r3,%r1,%r3`	No choice	1
`st %r3,[a]`	No choice	0

Scheduling for Superscalar & Multiple Issue Machines

A number of computers have the ability to issue more than one instruction per cycle *if* the instructions are independent and satisfy constraints on available functional units.

Thus the instructions

```
add %r1,1,%r2
sub %r1,2,%r3
```

can issue and execute in parallel, but

```
add %r1,1,%r2
sub %r2,2,%r3
```

must execute sequentially.

Instructions that are linked by true or output dependencies must execute sequentially, but instructions that are linked by an anti dependence may execute concurrently.

For example,

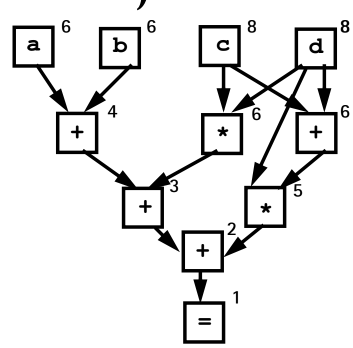
```
add %r1,1,%r2
sub %r3,2,%r1
```

can issue and execute in parallel.

The code scheduling techniques we've studied can be used to schedule machines that can issue 2 or more independent instructions simultaneously.

We select pairs (or triples or n-tuples), verifying (with the Dependence Dag) that they are independent or anti dependent.

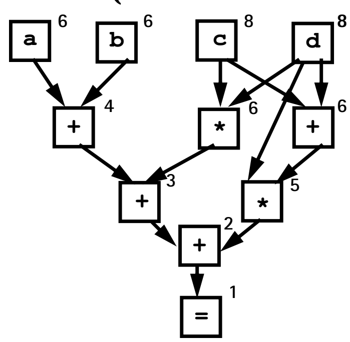
Example: 5 Registers (2 Wide Issue)



1	`ld [c], %r1`	`ld [d], %r2`
2	`ld [a], %r3`	`ld [b], %r4`
3	`smul %r1,%r2,%r5`	`add %r1,%r2,%r1`
4	`add %r3,%r4,%r3`	`smul %r1,%r2,%r1`
5	`nop`	`nop`
6	`add %r3,%r5,%r3`	`nop`
7	`add %r3,%r1,%r3`	`nop`
8	`st %r3,[a]`	`nop`

We need only 8 cycles rather than 11.

5 Registers (3 Wide Issue)



1	`ld [c], %r1`	`ld [d], %r2`	`ld [a],%r3`
2	`ld [b], %r4`	`nop`	`nop`
3	`smul %r1,%r2,%r5`	`add %r1,%r2,%r1`	`nop`
4	`add %r3,%r4,%r3`	`smul %r1,%r2,%r1`	`nop`
5	`nop`	`nop`	`nop`
6	`add %r3,%r5,%r3`	`nop`	`nop`
7	`add %r3,%r1,%r3`	`nop`	`nop`
8	`st %r3,[a]`	`nop`	`nop`

We still need 8 cycles!

Finding Additional Independent Instructions for Parallel Issue

We can extend the capabilities of processors:

- Out of order execution allows a processor to “search ahead” for independent instructions to launch.
- *But*, since basic blocks are often quite small, the processor may need to accurately predict branches, issuing instructions before the execution path is fully resolved.
- *But*, since branch predictions may be wrong, it will be necessary to “undo” instructions executed speculatively.

Compiler Support for Extended Scheduling

- Trace Scheduling

  Gather sequences of basic blocks together and schedule them as a unit.
- Global Scheduling

  Analyze the control flow graph and move instructions across basic block boundaries to improve scheduling.
- Software Pipelining

  Select instructions from several loop iterations and schedule them together.

Trace Scheduling

Reference:

J. Fisher, “Trace Scheduling: A Technique for Global Microcode Compaction,” IEEE Transactions on Computers, July 1981.

Idea:

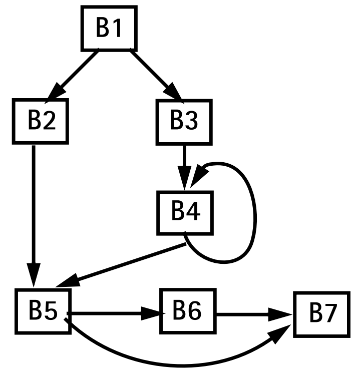
Since basic blocks are often too small to allow effective code scheduling, we will *profile* a program's execution and identify the most frequently executed paths in a program.

Sequences of contiguous basic blocks on frequently executed paths will be gathered together into *traces*.

Trace

- A sequence of basic blocks (excluding loops) executed together can form a trace.
- A trace will be scheduled as a unit, allowing a larger span of instructions for scheduling.
- A loop can be unrolled or scheduled individually.
- *Compensation code* may need to be added when a branch into, or out of, a trace occurs.

Example



Assume profiling shows that
$B1 \rightarrow B3 \rightarrow B4^{+} \rightarrow B5 \rightarrow B7$
is the most common execution path.

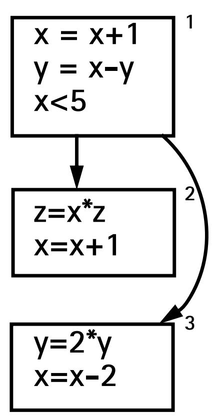
The traces extracted from this path are
$B1 \rightarrow B3$, B4, and $B5 \rightarrow B7$.

Compensation Code

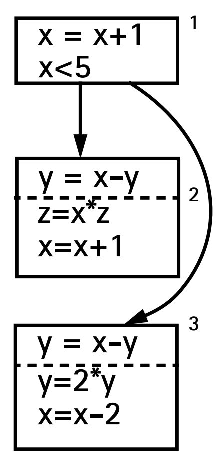
When we move instructions across basic block boundaries within a trace, we may need to add extra instructions that preserve program semantics on paths that enter or leave the trace.

Example

In the previous example, basic block B1 had B2 and B3 as successors, and $B1 \rightarrow B3$ formed a trace.



Before Scheduling



After Scheduling

Advantages & Disadvantages

- Trace scheduling allows scheduling to span multiple basic blocks. This can significantly increase the effectiveness of scheduling, especially in the context of superscalar processors (which need ILP to be effective).
- Trace Scheduling can also increase code size (because of compensation code). It is also sensitive to the accuracy of trace estimates.

Global Code Scheduling

- Bernstein and Rodeh approach.
- A *prepass scheduler* (does scheduling before register allocation).
- Can move instructions across basic block boundaries.
- Prefers to move instructions that *must* eventually be executed.
- Can move Instructions *speculatively*, possibly executing instructions unnecessarily.

Data & Control Dependencies

When moving instructions across basic block boundaries, we must respect both data dependencies and control dependencies.

Data dependencies specify necessary orderings among instructions that produce a value and instructions that use that value.

Control dependencies determine when (and if) various instructions are executed. Thus an instruction is control dependent on expressions that affect flow of control to that instruction.