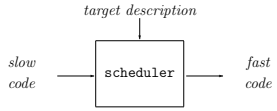
Instruction scheduling: The engineer's view

The problem

Given a code fragment for some target machine and the latencies for each individual instruction, reorder the instructions to minimize execution time

Conceptually, a scheduler looks like



Its task

- produce correct code (*preserve flow of data*)
- minimize wasted cycles (*interlocks & stalls*)
- avoid spilling registers (*adding* `stores` *&* `loads`)
- operate efficiently (*reasonable compile time*)

Example

```
w ← w * 2 * x * y * z
```

Assume:

- non-blocking load
- arguments can be reused next cycle

Cycles per op

*f*load	5
*f*store	5
*f*loadi	1
*f*shift	1
*f*add	1
*f*mult	2

simple schedule

```
 1 fload  r1 ← sp+@w
 2 floadi r2 ← 2
 6 fmult  r1 ← r1, r2
 7 fload  r2 ← sp+@x
12 fmult  r1 ← r1, r2
13 fload  r2 ← sp+@y
18 fmult  r1 ← r1, r2
19 fload  r2 ← sp+@z
24 fmult  r1 ← r1, r2
26 fstore sp+@2 ← r1
33 r1 available again
```

2 registers, 33 cycles

load aggressively

```
 1 fload  r1 ← sp+@w
 2 fload  r2 ← sp+@x
 3 fload  r3 ← sp+@y
 4 fload  r4 ← sp+@z
 5 floadi r5 ← 2
 6 fmult  r1 ← r1, r5
 8 fmult  r1 ← r1, r2
10 fmult  r1 ← r1, r3
12 fmult  r1 ← r1, r4
14 fstore sp+@2 ← r1
15 ...
19 r1 available again
```

5 registers, 19 cycles

Heuristics (Proebsting)

- hiding latency of k requires $k+1$ registers
- load aggressively, fill with operations

Instruction scheduling: The abstract view

Scheduling graph

To capture the important properties of the code, we build a scheduling graph, $G = (N, E, \mathit{type}, \mathit{delay})$.
Each $n \in N$ is an instruction of $\mathit{type}(n)$ with $\mathit{delay}(n)$.
An edge $e = (n_1, n_2) \in E$ *iff* n_2 uses n_1.

Definitions

A correct schedule S maps each $n \in N$ into a non-negative integer that represents its cycle number, and

1. $S(n) \geq 0$, for all $n \in N$
2. if $(n_1, n_2) \in E, S(n_1) + \mathit{delay}(n_1) \leq S(n_2)$
3. for each type t, there are no more instructions of *type* t in any cycle than the machine can issue

The *length* of a schedule S, denoted $L(S)$, is

$$L(S) = \max_{n \in N}(S(n) + \mathit{delay}(n))$$

The goal is to find the shortest possible correct schedule.
S is optimal if $L(S) \leq L(S_1), \forall$ schedules S_1.

Instruction scheduling: What's so difficult?

Critical points

1. operands must be available (*correctness*)
2. multiple ops can be *ready* (*choice*)
3. moving ops can lengthen register lifetimes
4. uses near definitions can shorten register lifetimes
5. ops have multiple predecessors (*start of block*)

Together, these issues make scheduling hard (*NP-complete*)

Simple case

- restricted to straight-line code
- single instruction per cycle
- consistent and predictable latencies

Even the simple case is NP-complete

Evolution of Instruction Scheduling Algorithms

1. "Labelling" algorithm [Sethi and Ullman]:
   Optimal code assuming single-cycle loads
   - ignores load latency
   - assumes *expression tree* of entire basic-block (*not dag*)
   - combines instruction scheduling and register allocation

2. DLS algorithm [Proebsting and Fischer]:
   Optimal code for a delayed-load architecture
   - fixed multi-cycle load latency
   - still assumes expression tree
   - Direct extension of Sethi-Ullman

3. List-scheduling algorithm [Gibbons and Muchnick]:
   Optimal scheduling for pipelined multiple-issue processors
   - fixed multi-cycle load latency
   - takes linear IL as input (e.g., 3-address code)
   - linear IL ⇒prior optimizations possible
   - linear IL ⇒some register allocation may be done already

List scheduling

Simple idea

1. retain a ready list of instructions by cycle
2. repeat cycle-by-cycle until all instructions scheduled:
   (a) choose an instruction and schedule it
   (b) add successors to appropriate ready list

But "list scheduling" is really a class of algorithms that use different heuristics for step 2(a).

Input

- $DAG(N, E)$ for basic block
- ExecTime(n) ≡ latency for each node (instruction) $n \in DAG$
  (*Can add specific latencies between pairs of instruction types to model more complex resource conflicts.*)

Output

- Start(n) ≡ cycle in which instruction at node n begins execution

See example on slide 12.

List scheduling algorithm: Engineering

Use clever data structures:

1. ReadyL: single ready list
2. W[c], $0 \leq c < MaxExecTime$: worklists by cycle,
   MaxExecTime ≡ $\max_{n \in DAG}$ ExecTime(n)
3. For each edge $e : n \rightarrow s$:
   Avail(s,e) ≡ cycle in which value from node n is available to node s (at the *start* of the cycle)

Heuristic function ChooseInstr(c, ReadyL, DAG):

- choose instruction from ReadyL to schedule in cycle c
- delete the instruction from ReadyL
- heuristics used here are key to good performance (later)

List scheduling algorithm: details

I. Initialization:

1. for each instruction i in block
   a. initialize Avail(i,e) appropriately *How?*
   b. if i is ready, add it to ReadyL *Which i?*
2. $\forall\ 0 \leq c <$ MaxExecTime, W[c] ← ∅
3. cycle ← 1

II. Repeat until all instructions are scheduled:

1. i ← ChooseInstr(cycle, ReadyL, DAG) *What if no such i?*
2. Start(i) = cycle
3. for each outgoing edge $e : i \rightarrow s$
   a. Avail(s,e) ← $cycle + ExecTime(i)$
   b. if Avail(s,e) has been initialized for all edges coming in to s:
      i. c ← $\max_e$ Avail(s,e)
      ii. c ← c MOD MaxExecTime *Why?*
      iii. W[c] ←W[c] ∪ s *Why?*
4. cycle ← cycle + 1
5. ReadyL ← ReadyL ∪ W[cycle `mod` MaxExecTime]
6. W[cycle `mod` MaxExecTime] ← ∅

Heuristic choices in list scheduling

ChooseInstr(c, ReadyL, DAG)

Most common priority scheme:
Give priority to instructions on the critical path.

- Critical paths ≡ the longest path through the scheduling graph
- use depth-first traversal to compute path lengths
- replace worklist with priority queue $(+\log_2(n))$

Good tie-breakers are important for robustness

- rank by longest path containing each node
  → *priority to longest paths of original DAG*
- rank by number of successors in the DAG
  → *priority to nodes used by many other nodes*
  → *exposes most #candidates*
- go depth first in schedule graph
  → *minimize register lifetimes*
- schedule last use as soon as possible
  → *frees up a register quickly*

Gibbons & Muchnick

See "Efficient instruction scheduling for a pipelined architecture" by P.B. Gibbons and S.S. Muchnick, *Proceedings of the SIGPLAN 86 Syposium on Compiler Construction*, July 1986 (SIGPLAN Notices 21(7))

Goal: avoiding pipeline hazards

- `load` followed by use of *that* register
- `store` followed by any `load`

Choice heuristics

1. instruction interlocks with successors in the *dag*
   → *interlock early ⇒ more candidates to cover it*
2. largest number of successors
3. longest path to roots of the *dag*

Results

- eliminates "most" pipeline hazards
- in practice, reasonably good schedules
- $O(n^2)$ complexity versus $O(n^4)$ for competition

This paper became one of the classics of scheduling literature.

Forward and Backward List Scheduling

There are many variations of list scheduling algorithms, but they break down into two classes:

Forward list scheduling

- start with available ops
- work forward
- ready ⇒ all ops available

Backward list scheduling

- start with no successors
- work backward
- ready ⇒ latency covers uses

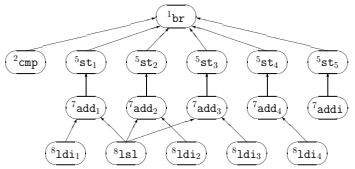
Which one is better?

- No clear choice: depends on dependence patterns, latencies
- A few critical operations may determine the overall schedule
- Critical operations appear near leaves: forward usually better
- Critical operations appear near roots: backward usually better

An idea: Why not try both and pick the best one?

- Building DAG and preprocessing is the biggest cost
- Scheduling algorithms themselves are relatively cheap
- Can try both forward and backward scheduling, and even multiple alternatives for each

List scheduling example: `go` *from SPEC*



opcode	ldi	lsl	add	addi	cmp	st
latency	1	1	2	1	1	4

Forward Schedule

	Int.	Int.	Mem.
1.	ldi_1	lsl	---
2.	ldi_2	ldi_3	---
3.	ldi_4	add_1	---
4.	add_2	add_3	---
5.	add_4	addi	st_1
6.	cmp	---	st_2
7.	---	---	st_3
8.	---	---	st_4
9.	---	---	st_5
10.	---	---	---
11.	---	---	---
12.	---	---	---
13.	br	---	---

Backward Schedule

	Int.	Int.	Mem.
1.	ldi_4	---	---
2.	addi	lsl	---
3.	add_4	ldi_3	---
4.	add_3	ldi_2	st_5
5.	add_2	ldi_1	st_4
6.	add_1	---	st_3
7.	---	---	st_2
8.	---	---	st_1
9.	---	---	---
10.	---	---	---
11.	cmp	---	---
12.	br	---	---

Going beyond single basic blocks (Overview)

List scheduling for extended basic blocks

- Single basic blocks are usually too small for wide-issue processors
- An extended basic block (EBB) $\equiv$ a series of blocks, $b_1 \ldots b_n$, where b_1 has multiple predecessors in the CFG, but $b_2 \ldots b_n$ each has only one predecessor
- Apply list scheduling algorithm to each EBB at a time in the graph
- Be careful when moving code "before" a branch:
  Move SSA register ops that cause no exceptions
  Use speculative loads before branch

Trace scheduling

- Important for VLIW and very wide issue machines
  $\Leftarrow$ need to keep many functional units busy
- Trace $\equiv$ an arbitrary sequence of basic blocks executed consecutively at runtime
- Use runtime profiles to choose most frequently executed traces
- Use list scheduling to schedule instructions on a trace, then eliminate its blocks from the graph, and move to the next trace
- More complicated rules for introducing copies

Beyond single basic blocks (continued)

Software pipelining

- Critical for scheduling small loops on wide-issue processors
- Begins by folding loop to create longer loop bodies
  $\rightarrow$ e.g., fold once to execute 2 iterations concurrently
- Body of loop (the *kernel*) executes second half of iteration i and first half of iteration $i+1$
  $\Rightarrow$ allows compiler to overlap long operations of iteration $i+1$ with uses of iteration i
- Prolog executes first half of iteration 1;
  epilog executes last half of iteration n
- Use list scheduling to schedule the kernel
- Generalizes to 2 or more iterations in kernel
- Important supporting transformations:
  – Loop unrolling
  – Register renaming
  – Variable renaming when unrolling loops (if not SSA)