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

slow code

target description

scheduled

fast code

Its task
• produce correct code
  (preserve flow of data)
• minimize waited cycles
  (interlocked stalls)
• avoid spilling registers
  (adding stores & loads)
• operate efficiently
  (reasonable compile time)

Instruction scheduling: The abstract view

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

Definitions
A correct schedule \( S \) maps each \( n \in V \) into a non-negative integer that represents its cycle number, and
1. \( S(n) \geq 0 \) for all \( n \in V \)
2. if \( (n_1, n_2) \in E \), \( S(n_1) + \text{delay}(n_1) \leq S(n_2) \)
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
\[ L(S) = \max_{n \in V} \{ S(n) + \text{delay}(n) \} \]
The goal is to find the shortest possible correct schedule. \( S \) is optimal if \( L(S) \leq L(S') \), \( \forall \) schedules \( S' \).

Instruction scheduling: What’s so difficult?

Critical points
1. operands must be available (correctness)
2. multiple ops can be ready (clean)
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

Example
\[ w = v * 2 * x * y * z \]

Cycles per op

Assume:
• non-blocking load
  \( \text{fetch} \) 5
• arguments can be reused next cycle
  \( \text{fetch} \) 1
  \( \text{fetch} \) 1
  \( \text{finish} \) 2

simple schedule

load r1 ← sp@w

load r2 ← 2

mult r1 ← r2, r1

store sp+@2

loadi r5 ← f5

mult r6 ← r5, r1

load r7 ← 2

mult r8 ← sp@w

store sp+@2

load r9 ← f8

mult r10 ← r9, r1

store sp+@2

\( \ldots \)

19 r1 available again

5 registers, 23 cycles

load aggressively

load r1 ← sp@w

load r2 ← 2

mult r1 ← r2, r1

store sp+@2

loadi r5 ← f5

mult r6 ← r5, r1

load r7 ← 2

mult r8 ← sp@w

store sp+@2

load r9 ← f8

mult r10 ← r9, r1

store sp+@2

\( \ldots \)

19 r1 available again

5 registers, 19 cycles

Heuristics
• finishing latency of \( k \) requires \( k + 1 \) registers
• load aggressively, fill with operations
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
   • 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 (2a).

Input:
• DAG \( (N, E) \) for basic block
• \( \text{ExecTime}(n) \) for each node
   (Can add specific latencies between pairs of instruction types to model more complex resource conflicts.)

Output:
• \( \text{Start}(n) \) = cycle in which instruction at node \( n \) begins execution

See example on slide 12.

---

List scheduling algorithm: Engineering

Use clever data structures:
1. \( \text{ReadyL} \), single ready list
2. \( W[c] \), \( 0 \leq c < \text{MaxExecTime} \):
   worklists by cycle
3. For each edge \( e : n \rightarrow s \):
   \( \text{Avail}(s,e) \) = cycle in which value from node \( n \) is available to
   node \( s \) (at the start of the cycle)

Heuristic function ChooseInstr([cycle, ReadyL, DAG]):
• choose instruction from \( \text{ReadyL} \) to schedule in cycle \( c \)
• delete the instruction from \( \text{ReadyL} \)
• heuristics used here are key to good performance (later)

---

List scheduling algorithm: details

I. Initialization:
1. for each instruction \( i \) in block
   a. initialize \( \text{Avail}(s,e) \) appropriately
   b. if \( i \) is ready, add it to \( \text{ReadyL} \)
      Which \( i \)?
2. \( 0 \leq c < \text{MaxExecTime}, W[c] \leftarrow \emptyset \)
3. cycle \( \leftarrow 1 \)

II. Repeat until all instructions are scheduled:
1. \( i \leftarrow \text{ChooseInstr(cycle, ReadyL, DAG)} \)
   \( \text{What if no such } i? \)
2. \( \text{Start}(i) \leftarrow c \)
3. for each outgoing edge \( e : i \rightarrow s \):
   a. \( \text{Avail}(s,e) \leftarrow \text{cycle} + \text{ExecTime}(i) \)
   b. if \( \text{Avail}(s,e) \) has been initialized for all edges coming in to \( s \):
      i. \( c \leftarrow \text{MAX} \), \( \text{Avail}(s,e) \)
      ii. \( c \leftarrow c + \text{MaxExecTime} \)
      iii. \( W[c] \leftarrow W[c] \cup s \)
   c. \( \text{cycle} \leftarrow \text{cycle} + 1 \)
5. \( \text{ReadyL} \leftarrow \text{ReadyL} \cup W[cycle \text{mod \text{MaxExecTime}}] \)
6. \( W[cycle \text{mod \text{MaxExecTime}}] \leftarrow \emptyset \)
### Heuristic choices in list scheduling

**ChooseInstr(c, ReadyL, DAG)**

**Most common priority scheme:** Give priority to instructions on the critical path.
- Critical paths are the longest paths through the scheduling graph
- Use depth-first traversal to compute path lengths
- Replace worklist with priority queue

**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
- Go depth first in schedule graph
  - Minimize register lifetimes
- Schedule last use as soon as possible
  - Free up a register quickly

### 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 use

### 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

---

### Gibbons & Muchnick


**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 \(O(n^3)\) for competition

This paper became one of the classics of scheduling literature.
**List scheduling for extended basic blocks**

- Single basic blocks are usually too small for wide-issue processors.
- An extended basic block (EBB) is a sequence 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.
- Need to keep many functional units busy.
- A trace is 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.

Software pipelining

- Critical for scheduling small loops on wide-issue processors.
- Begins by folding loop to create longer loop bodies.
- Proceeds once to execute 2 iterations concurrently.
- Body of loop (the kernel) executes second half of iteration \( i \) and first half of iteration \( i + 1 \).
- Allows compiler to overlap long operations of iteration \( i + 1 \) with use 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).