Lecture 17
List Scheduling

Reading: Chapter 10.3 - 10.4

Review: The Ideal Scheduling Outcome
• What prevents us from achieving this ideal?

Review: Scheduling Constraints
• Hardware Resources
  – finite set of FUs with instruction type, bandwidth, and latency constraints
  – cache hierarchy also has many constraints
• Data Dependences
  – can't consume a result before it is produced
  – ambiguous dependences create many challenges
• Control Dependences
  – impractical to schedule for all possible paths
  – choosing an "expected" path may be difficult
    • recovery costs can be non-trivial if you are wrong

Scheduling Roadmap

List Scheduling:
• within a basic block
Trace Scheduling:
• across basic blocks
Software Pipelining:
• across loop iterations
List Scheduling

- The most common technique for scheduling instructions within a basic block.

We don't need to worry about:
- control flow

We do need to worry about:
- data dependences
- hardware resources

- Even without control flow, the problem is still NP-hard.

List Scheduling: The Basic Idea

- Maintain a list of instructions that are ready to execute:
  - data dependence constraints would be preserved
  - machine resources are available

- Moving cycle-by-cycle through the schedule template:
  - choose instructions from the list & schedule them
  - update the list for the next cycle

List Scheduling Algorithm: Inputs and Outputs

Inputs:
- Data Precedence Graph (DPG)
- Machine Parameters

Outputs:
- Scheduled Code
- Cycle

What Makes Life Interesting: Choice

Easy case:
- all ready instructions can be scheduled this cycle

Interesting case:
- we need to pick a subset of the ready instructions
- List scheduling makes choices based upon priorities
  - assigning priorities correctly is a key challenge
**Intuition Behind Priorities**

- Intuitively, what should the priority correspond to?
- What factors are used to compute it?
  - data dependences?
  - machine parameters?

**Representing Data Dependences:**
The Data Precedence Graph (DPG)

- Two different kinds of edges:
  - true "edges": \( E \)
  - "anti-edges": \( E' \)

**Computing Priorities**

- Let's start with just true dependences (i.e. "edges" in DPG)
- Priority = latency-weighted depth in the DPG

\[
\text{priority}(x) = \max_{\ell \in \text{E}(\text{DPG})} \sum_{p \in \text{paths}(x, \ldots, l)} \text{latency}(p)
\]
List Scheduling Algorithm

cycle = 0;
ready-list = root nodes in DPG; inflight-list = {};

while (((ready-list)∪(inflight-list)) > 0) if an issue slot is available) {
    for op = (all nodes in ready-list in descending priority order) {
        if (an FU exists for op to start at cycle) {
            remove op from ready-list and add to inflight-list;
            add op to schedule at time cycle;
            if (op has an outgoing anti-edge)
                add all targets of op’s anti-edges that are ready to ready-list;
        }
    }
    cycle = cycle + 1;
    for op = (all nodes in inflight-list)
        if (op finishes at time cycle) {
            check nodes waiting for op & add to ready-list if all operands available;
        }
}

Example

I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1

• 2 identical fully-pipelined FUs
• adds take 2 cycles; all other insts take 1 cycle

What if We Break Ties Differently?

I0: a = 1
I1: f = a + x
I2: b = 7
I3: c = 9
I4: g = f + b
I5: d = 13
I6: e = 19
I7: h = f + c
I8: j = d + y
I9: z = -1
I10: JMP L1

• 2 identical fully-pipelined FUs
• adds take 2 cycles; all other insts take 1 cycle
What if We Break Ties Differently?

I0: \( a = 1 \)
I1: \( f = a + x \)
I2: \( b = 7 \)
I3: \( c = 9 \)
I4: \( d = 13 \)
I5: \( e = 19 \)

Cycle
0: I0
1: I1

Cycle
0: I0
1: I1
2: I2
3: I3
4: I4
5: I5

Cycle
0: I0
1: I1
2: I2
3: I3
4: I4
5: I5
6: I6

• 2 identical fully-pipelined FUs
• adds take 2 cycles; all other insts take 1 cycle

Contrasting the Two Schedules

• Breaking ties \textit{arbitrarily} may not be the best approach

Backward List Scheduling

Modify the algorithm as follows:
– reverse the direction of all edges in the DPG
– schedule the \textit{finish times} of each operation
  • start times must still be used to ensure FU availability

Impact of scheduling backwards:
– clusters operations near the end (vs. the beginning)
– may be either better or worse than forward scheduling
Now Let's Try Scheduling Backward

Evaluate of List Scheduling

Cooper et al. propose "RBF" scheduling:
- schedule each block \( M \) times forward & backward
- break any priority ties randomly

For real programs:
- regular list scheduling works very well

For synthetic blocks:
- RBF wins when "available parallelism" (AP) is \( \approx 2.5 \)
- for smaller AP, scheduling is too constrained
- for larger AP, any decision tends to work well

Contrasting Forward vs. Backward List Scheduling

Hardware parameters:
- 2 INT units: ADDs take 2 cycles; others take 1 cycle
- 1 MEM unit: stores (ST) take 4 cycles

- backward scheduling clusters work near the end
- backward is better in this case, but this is not always true

List Scheduling Wrap-Up

- The priority function can be arbitrarily sophisticated
  - e.g., filling branch delay slots in early RISC processors

- List scheduling is widely used, and it works fairly well

- It is limited, however, by basic block boundaries
Scheduling Roadmap

List Scheduling:
- within a basic block

Trace Scheduling:
- across basic blocks

Software Pipelining:
- across loop iterations

Carnegie Mellon
15745: List Scheduling
Todd C. Mowry