Lecture 20
Global Scheduling & Software Pipelining

[ALSU 10.4-10.5]

Review: 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

Scheduling Roadmap

List Scheduling:
• within a basic block (lecture 15)

Global Scheduling:
• across basic blocks

Software Pipelining:
• across loop iterations

Review: The Data Precedence Graph (DPG)
- Two different kinds of edges:
  - Code:
    - E: true “edges”: (read-after-write)
    - E': “anti-edges”: (write-after-read)
  - DPG:
    - Why distinguish them?
    - do they affect scheduling differently?
    - What about output dependences?
List Scheduling with Priorities

\[ \text{priority}(v) = \max (\forall e \in \text{edges}(DAG) \forall p \in \text{paths}(e, v)) \sum_{p_i = z}^1 \text{latency}(p_i) \]

- 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: \( \text{JMP L1} \)

Cycle

0
1
2
3
4
5
6

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

Introduction to Global Scheduling

Assume each clock can execute 2 operations of any kind

if (a==0) goto L

LD R6 <- O(R1)
nop
BEQ R6, L

L:
e = d + d

LD R7 <- O(R2)
nop
ST O(R3) <- R7

LD R8 <- O(R4)
nop
ADD R8 <- R8,R8
ST O(R5) <- R8

LD R6 <- O(R1) ; LD R8 <- O(R4)
LD R7 <- O(R2)
ADD R8 <- R8,R8
BEQZ R6, L

L:
ST O(R5) <- R8
ST O(R5) <- R8 ; ST O(R3) <- R7

Scheduling Roadmap

- List Scheduling: within a basic block
- Global Scheduling: across basic blocks
- Software Pipelining: across loop iterations

Terminology

Control equivalence:
- Two operations \( o_1 \) and \( o_2 \) are control equivalent if \( o_1 \) is executed if and only if \( o_2 \) is executed.

Control dependence:
- An op \( o_2 \) is control dependent on op \( o_1 \) if the execution of \( o_2 \) depends on the outcome of \( o_1 \).

Speculation:
- An operation is speculatively executed if it is executed before all the operations it depends on (control-wise) have been executed.
- Requirements:
  - does not raise an exception
  - satisfies data dependences
Code Motions

Goal: Shorten execution time \textit{probabilistically}

Moving instructions up:
- Move instruction to a cut set (from entry)
- Speculation: even when not anticipated.

Moving instructions down:
- Move instruction to a cut set (from exit)
- May execute extra instruction
- Can duplicate code

Review: Code Motion for Partial Redundancy Elimination

- \textbf{Partial redundancy at} \textit{p}: redundant on some but not all paths
  - Add operations to create a cut set containing \(a+b\)
  - Note: Moving operations up can eliminate redundancy
- \textbf{Constraint on placement: no wasted operation}
  - \(a+b\) is \textit{"anticipated"} at \(B\) if its value computed at \(B\) will be used along all subsequent paths
  - \(a, b\) not redefined, no branches that lead to exit without use
- \textbf{Range where} \(a+b\) \textbf{is anticipated} \rightarrow \textbf{Choice}

General-Purpose Applications

- Lots of data dependences
- Key performance factor: \textit{memory latencies}
- \textbf{Move memory fetches up}
  - Speculative memory fetches can be expensive
- \textbf{Control-intensive: get execution profile}
  - Static estimation
    - Innermost loops are frequently executed
      - \textit{back edges are likely to be taken}
    - Edges that branch to exit and exception routines are not likely to be taken
  - Dynamic profiling
    - Instrument code and \textit{measure} using representative data

A Basic Global Scheduling Algorithm

- Schedule innermost loops first
- Only upward code motion
- No creation of copies
- Move operations speculatively up only one branch
Program Representation

- Recall: A region in a control flow graph is:
  - a set of basic blocks and all the edges connecting these blocks,
  - such that control from outside the region must enter through a single entry block
- A procedure is represented as a hierarchy of regions
  - The whole control flow graph is a region
  - Each natural loop (single entry with back edge to it) in the flow graph is a region
  - Natural loops are hierarchically nested
- Schedule regions from inner to outer
  - treat inner loop as a black box unit
  - can schedule around it but not into it
  - ignore all the loop back edges \( \Rightarrow \) get an acyclic graph

Algorithm

compute data dependences;
for each region from inner to outer {
  for each basic block \( B \) in prioritized topological order {
    \( \text{CandBlocks} = \text{ControlEquiv}(B) \cup \text{Dominated-Successors}(\text{ControlEquiv}(B)) \);
    \( \text{CandInsts} = \text{ready operations in CandBlocks} \);
    for \( i = 0, 1, \ldots \) until all operations from \( B \) are scheduled {
      if \( i \) has no resource conflicts at time \( t \) {
        \( \text{S}(i) = \langle B, t \rangle \)
        update resource commitments
        update data dependences
      }
    }
    update \( \text{CandInsts} \);
  }
}

Priority functions: non-speculative before speculative

Extensions

- Prepass before scheduling: loop unrolling
- Especially important to move operation up loop back edges

Global Scheduling Summary

- Global scheduling
  - Legal code motions
  - Heuristics
### Scheduling Roadmap

**List Scheduling:**
- within a basic block

**Global Scheduling:**
- across basic blocks

**Software Pipelining:**
- across loop iterations

### List Scheduling:
- \( x = a + b \)  
- \( y = c + d \)

### Global Scheduling:
- \( x = a + b \)  
- \( y = c + d \)

### Software Pipelining:
- \( x = a + b \)  
- \( y = c + d \)

### Example of DoAll Loops

**Machine:**
- Per clock: 1 read, 1 write, 1 (2-stage) arithmetic op, with hardware loop op and auto-incrementing addressing mode.

**Source code:**
\[
\text{For } i = 1 \text{ to } n \\
D[i] = A[i] \times B[i] + c
\]

**Code for one iteration:**
1. LD R5,0(R1++)
2. LD R6,0(R2++)
3. MUL R7,R5,R6
4. ADD R8,R7,R4
5. ST 0(R3++),R8

**Little or no parallelism within basic block**

### Loop Unrolling

1. L: LD
2. LD
3. MUL LD
4. MUL LD
5. ADD LD
6. ADD LD
7. MUL LD
8. MUL LD
9. ADD
10. ST ADD
11. MUL
12. MUL
13. MUL

**Schedule after unrolling by a factor of 4**

**Software Pipelined Code**

1. LD
2. LD
3. MUL LD
4. MUL LD
5. MUL LD
6. ADD LD
7. L: MUL LD
8. ST ADD LD BL (L)
9. MUL
10. ST ADD
11. ST ADD
12. ST
13. ST
14. ST

**Let \( u \) be the degree of unrolling:**
- Length of \( u \) iterations = \( 7 + 2(u - 1) \)
- Execution time per source iteration = \( (7 + 2(u - 1)) / u = 2 + 5/u \)

**Unlike unrolling, software pipelining can give optimal result**
- Locally compacted code may not be globally optimal
- **DOALL**: Can fill arbitrarily long pipelines with infinitely many iterations
**Example of DoAcross Loop**

Loop:
- \( \text{Sum} = \text{Sum} + A[i] \)
- \( B[i] = A[i] \times c \)

Software Pipelined Code
1. \( \text{LD} \)
2. \( \text{MUL} \)
3. \( \text{ADD} \)
4. \( \text{ST} \)
5. \( \text{LD} \)
6. \( \text{MUL} \)
7. \( \text{ADD} \)
8. \( \text{ST} \)

Doacross loops
- Recurrences can be parallelized
- Harder to fully utilize hardware with large degrees of parallelism

**Problem Formulation**

Goals:
- maximize throughput
- small code size

Find:
- an identical relative schedule \( S(n) \) for every iteration
- a constant initiation interval (\( T \)) such that
- the initiation interval is minimized

Complexity:
- NP-complete in general

**Impact of Resources on Bound on Initiation Interval**

Example: Resource usage of 1 iteration
- (assume machine can execute 1 LD, 1 ST, 2 ALU per clock)

\( \text{LD}, \text{LD}, \text{MUL}, \text{ADD}, \text{ST} \)

Lower bound on initiation interval?
- for all resource \( i \)
  - number of units required by one iteration: \( r_i \)
  - number of units in system: \( R_i \)

\[ \text{Lower bound due to resource constraints: } \max_i \left( \frac{r_i}{R_i} \right) \]

**Scheduling Constraints: Resources**

1. \( \text{LD} \)
2. \( \text{LD} \)
3. \( \text{MUL} \)
4. \( \text{ADD} \)
5. \( \text{ADD} \)
6. \( \text{ST} \)
7. \( \text{ST} \)
8. \( \text{ST} \)

- \( R_T \): resource reservation table for single iteration
- \( R_T^s \): modulo resource reservation table

\[ R_T[i] = \Sigma_{i j} \mod T = i R_T[j] \]
Scheduling Constraints: Precedence

```c
for (i = 0; i < n; i++) {
    *(p++) = *(q++) + c
}
```

- Minimum initiation interval $T$: $1+2+1 = 4$
- $S(n)$: schedule for $n$ with respect to the beginning of the schedule
- Label edges with $<\delta, d>$
  - $\delta$: iteration difference, $d$: delay
  - $\delta \times T + S(n_2) - S(n_1) \geq d$

Minimum Initiation Interval

For all cycles $c$, $T = \frac{\text{CycleLength}(c)}{\text{IterationDifference}(c)}$

Example: An Acyclic Graph inside a loop

```
Example Graph
```

Algorithm: Software Pipelining Acyclic Dependence Graphs

- Find lower bound of initiation interval: $T_0$
  - based on resource constraints
- For $T = T_0, T_0 + 1, ...$ until all nodes are scheduled
  - For each node $n$ in topological order
    - $s_n = \text{earliest} n$ can be scheduled
    - for each $s = s_0, s_0 + 1, ..., s_0 + T - 1$
      - if $\text{NodeScheduled}(n, s)$ break;
      - if $n$ cannot be scheduled break;
- $\text{NodeScheduled}(n, s)$
  - Check resources of $n$ at $s$ in modulo resource reservation table
- Can always meet the lower bound if:
  - every operation uses only 1 resource, and
  - no cyclic dependence in the loop
Cyclic Graphs

- No such thing as "topological order"
- \( b \rightarrow c; c \rightarrow b \)
  
\[ S(c) - S(b) \geq 1 \]
\[ T + S(b) - S(c) \geq 2 \]

- Scheduling \( b \) constrains \( c \), and vice versa

\[ S(b) + 1 \leq S(c) \leq S(b) - 2 + T \]
\[ S(c) - T + 2 \leq S(b) \leq S(c) - 1 \]

A Closer Look at Register Allocation for Software Pipelining

Software-pipelined code:

1. LD R5,0(R1++)
2. LD R6,0(R2++)
3. MUL LD
4. LD
5. MUL LD
6. ADD LD
8. ST ADD MUL LD BL L
9. ST ADD MUL LD
10. ST ADD LD
11. ST ADD MUL
12. ST ADD
13. ST ADD
14.
15.
16.

What is the problem w.r.t. \( R7 \)?

Solution: Modulo Variable Expansion

1. LD R5,0(R1++)
2. LD R6,0(R2++)
3. LD R5,0(R1++) MUL R7, R5, R6
4. LD R6,0(R2++) MUL R9, R5, R6
5. LD R5,0(R1++) MUL R9, R5, R6
6. LD R6,0(R2++) ADD R8, R7, R4
8. LD R5,0(R1++) MUL R7, R5, R6
9. LD R6,0(R2++) ADD R8, R7, R4
10. LD R5,0(R1++) MUL R9, R5, R6
11. MUL R7, R5, R6
12. ADD R8, R9, R4
13. ADD R8, R7, R4
14. ADD R8, R7, R4
15.
16.

Algorithm: Software Pipelining with Modulo Variable Expansion

- Normally, every iteration uses the same set of registers
  - introduces artificial anti-dependences for software pipelining
- Modulo variable expansion algorithm
  - schedule each iteration ignoring artificial constraints on registers
  - calculate life times of registers
  - degree of unrolling = max,(lifetime, T)
  - unroll the steady state of software pipelined loop to use different registers
- Code generation
  - generate one pipelined loop with only one exit (at beginning of steady state)
  - generate one unpipelined loop to handle the rest
  - code generation is the messiest part of the algorithm!
Conclusions

- **Numerical Code**
  - Software pipelining is useful for machines with a lot of pipelining and instruction level parallelism
  - Compact code
  - **Limits to parallelism**: dependences, critical resource

List Scheduling:
- within a basic block

Global Scheduling:
- across basic blocks

Software Pipelining:
- across loop iterations

Next Week: Prefetching
- Monday: Prefetching Arrays
- Wednesday: Prefetching Pointer-based Structures