Lecture 8
Software Pipelining

I. Introduction
II. DoAll Loops Scheduling
III. DoAcross Loops Scheduling
IV. Register Allocation

Reading: Chapter 10.5 - 10.6
I. 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:**
  
  For \( i = 1 \) to \( n \)
  
  \[
  \]

- **Code for one iteration:**
  1. LD \( R5,0(R1++) \)
  2. LD \( R6,0(R2++) \)
  3. MUL \( R7,R5,R6 \)
  4. 
  5. ADD \( R8,R7,R4 \)
  6. 
  7. ST \( 0(R3++),R8 \)

- **No parallelism in basic block**
Unrolling

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

• Let \( u \) be the degree of unrolling:
  – Length of \( u \) iterations = \( 7+2(u-1) \)
  – Execution time per source iteration = \( \frac{7+2(u-1)}{u} = 2 + \frac{5}{u} \)
Software Pipelined Code

1. LD
2. LD
3. MUL    LD
4.       LD
5.       MUL    LD
6. ADD           LD
7.               MUL    LD
8. ST     ADD           LD
9.                      MUL    LD
10.        ST     ADD           LD
11.        ST     ADD        MUL
12.        ST     ADD
13.                             ST
14.                             ST
15.                             ST
16.                             ST

- Unlike unrolling, software pipelining can give optimal result.
- Locally compacted code may not be globally optimal
- DOALL: Can fill arbitrarily long pipelines
Example of DoAcross Loops

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

Software Pipelined Code
1. LD
2. MUL
3. ADD
4. 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
II. Resources on Bound on Initiation Interval

• Example: Resource usage of 1 iteration; Machine can execute 1 LD, 1 ST, 2 ALU per clock

LD, LD, MUL, ADD, ST

• Lower bound on initiation interval?

for all resource \( i \),

- number of units required by one iteration: \( n_i \)
- number of units in system: \( R_i \)

Lower bound due to resource constraints: \( \max_i \frac{n_i}{R_i} \)
Scheduling Constraints: Resources

- RT: resource reservation table for a single iteration
- RT_m: modulo resource reservation table for initiation interval T

\[ RT_m[i] = \sum_{t \in \{t \mod T = i\}} RT[t] \]
Example: DoAll Loops

[Diagram showing a graph with nodes a, b, and c, and edges labeled 1, 2, and 3.]
Algorithm for DoAll Loops

Find lower bound of initiation interval: $T_0$
   based on resource constraints

For $T = T_0, T_0+1, \ldots$ until all nodes are scheduled
   For each node $n$ in topological order
      $s_0 = \text{earliest } n \text{ can be scheduled}$
      for each $s = s_0, s_0+1, \ldots, s_0+T-1$
         if NodeScheduled($n, s$) break;
         if $n$ cannot be scheduled break;

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
  - the loop is a DoAll loop (no recurrences)
III. Do-Across Loops

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

- Minimum initiation interval?
- Label edges with <<δ, d>>
  - δ = iteration difference, d = delay
- S(n): Schedule for n with respect to the beginning of the iteration it is in
  \[ δ \times T + S(n_2) - S(n_1) ≥ d \]
III. Do-Across Loops

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

- Minimum initiation interval?
- Label edges with $<\delta, d>$
  - $\delta$ = iteration difference, $d$ = delay
- $S(n)$: Schedule for $n$ with respect to the beginning of the iteration it is in
  $$\delta \times T + S(n_2) - S(n_1) \geq d$$
Scheduling Constraints: Precedence

for (i = 2; i < n; i++) {
}

• Minimum initiation interval?
• Label edges with $<\delta, d>$
  • $\delta$ = iteration difference, $d$ = delay
• $S(n)$: Schedule for $n$ with respect to the beginning of the iteration it is in

$$\delta \times T + S(n_2) - S(n_1) \geq d$$
Minimum Initiation Interval

For all cycles $c$,

$$\max_c \frac{\text{CycleLength}(c)}{\text{IterationDifference}(c)}$$
Cyclic Graphs

\[ \begin{array}{c}
\text{A} \\
<0,2> \\
\text{B} \\
<0,1> \\
\text{C} \\
<0,1> \\
\text{D} \\
<1,2> \\
\end{array} \]

\[ \begin{array}{cccccccc}
0 & & & & & & & \\
1 & & & & & & & \\
2 & & & & & & & \\
3 & & & & & & & \\
4 & & & & & & & \\
5 & & & & & & & \\
6 & & & & & & & \\
7 & & & & & & & \\
\end{array} \]
Strongly Connected Components

- A strongly connected component (SCC)
  - Set of nodes such that every node can reach every other node

- Every node constrains all others from above and below
  - Finds longest paths between every pair of nodes
  - As each node scheduled, find lower and upper bounds of all other nodes in SCC

- SCCs are hard to schedule
  - Critical cycle: no slack
    - Backtrack starting with the first node in SCC
    - Increases T, increases slack

- Edges between SCCs are acyclic
  - Acyclic graph: every node is a separate SCC
**Full Algorithm**

Find lower bound of initiation interval: $T_0$
   based on resource constraints and precedence constraints

For $T = T_0, T_0+1, \ldots$, until all nodes are scheduled
   
   $E^*$ = longest path between each pair
   
   For each SCC $c$ in topological order
      $s_0$ = Earliest $c$ can be scheduled
      For each $s = s_0, s_0+1, \ldots, s_0+T-1$
         If $SCCScheduled(c, s)$ break;
      
      If $c$ cannot be scheduled return false;

Return true;
Scheduling a Strongly Connected Component (SCC)

**SCCScheduled(c, s)**

Schedule first node at \( s \), return false if fails

For each remaining node \( n \) in \( c \)

- \( s_l \) = lower bound on \( n \) based on \( E^* \)
- \( s_u \) = upper bound on \( n \) based on \( E^* \)

For each \( s = s_l, s_l + 1, \min (s_l + T-1, s_u) \)

- if NodeScheduled\((n, s)\) break;
- if \( n \) cannot be scheduled return false;

Return true;
IV. Register Allocation

- Software-pipelined code

1. LD
2. LD
3. MUL LD
4. LD
5. MUL LD
6. ADD LD

L: 7. MUL LD
8. ST ADD LD BL L
9. MUL
10. ST ADD
11.
12. ST ADD
13.
14. ST

1. LD R5,0(R1++)
2. LD R6,0(R2++)
3. MUL R7,R5,R6
4.
5.
6. ADD R8,R7,R4
7.
8. ST 0(R3++),R8
Modulo Variable Expansion

1. LD R5,0 (R1++)
2. LD R6,0 (R1++)
3. MUL R7,R5,R6
4. 
5. 
6. ADD R8,R7,R4
7. 
8. ST 0 (R3++),R8
9. MUL R7,R5,R6
10. ADD R8,R17,R17
    ST 0 (R3++),R8
    BL L
11. MUL R7,R5,R6
12. ADD R8,R17,R17
    ST 0 (R3++),R8
13. 
14. ADD R8,R7,R7
    ST 0 (R3++),R8
15. 
16. ST 0 (R3++),R8
Algorithm

- **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 = $\max_r \left( \text{lifetime}_r / T \right)$
  - 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

• **General Lessons**
  – Problem formulation: Important to identify
    • the need (parallel hardware),
    • the opportunity (many numerical codes have independent operations)
  – Designing the right abstraction to address the key constraint
    • modulo scheduling