Lecture 19

Software Pipelining

I. Introduction

II. Problem Formulation

III. Algorithm

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
        D[i] = A[i] * B[i]+ c
```

- **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
```

- **Little or no parallelism within basic block**

Loop Unrolling

Schedule after unrolling by a factor of 4

```
 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 = (7+2(**u**-1)) / **u** = 2 + 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.                                 MUL
12.                 ST      ADD             ...
13.
14.                         ST      ADD
15.
16.                                 ST
```

- 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:

```
Sum = Sum + A[i];
B[i] = A[i] * c;
```

→

```
1. LD
2. MUL
3. ADD
4. ST
```

Software Pipelined Code

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

Doacross loops

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

II. 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

```
S
0 LD
1 MUL        T=2
2 ADD    LD
3 ST     MUL
         ADD
         ST
```

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)

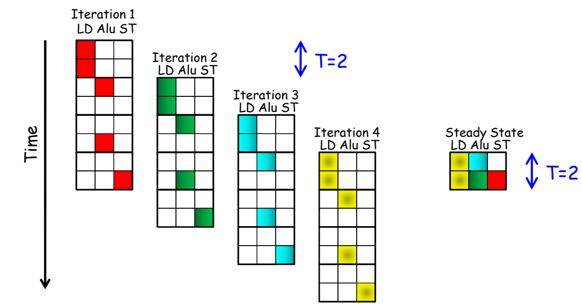
  LD, LD, MUL, ADD, ST

- **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: $\max_i n_i/R_i$

Scheduling Constraints: Resources

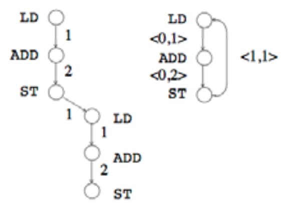


- RT: resource reservation table for single iteration
- RT_s: modulo resource reservation table

$$RT_s[i] = \Sigma_{t|(t \bmod T = i)} RT[t]$$

Scheduling Constraints: Precedence

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

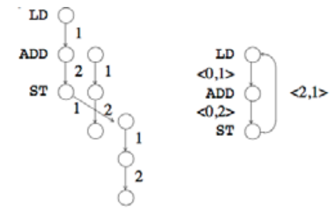


- Minimum initiation interval?
- S(n): schedule for n with respect to the beginning of the schedule
- Label edges with < δ, d >
  - δ = iteration difference, d = delay

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

Scheduling Constraints: Precedence

```
for (i = 2; i < n; i++) {
    A[i] = A[i-2] + 1;
}
```



- Minimum initiation interval?
- S(n): schedule for n with respect to the beginning of the schedule
- Label edges with < δ, d >
  - δ = iteration difference, d = delay

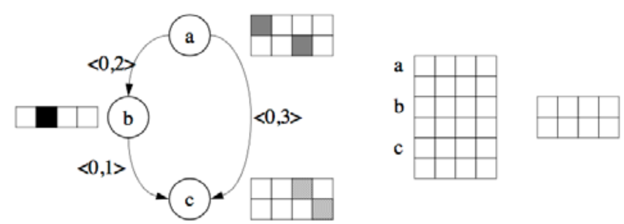
$$\delta \times T + S(n_2) - S(n_1) \geq d$$

Minimum Initiation Interval

For all cycles c,

$\max_c$ CycleLength(c) / IterationDifference (c)

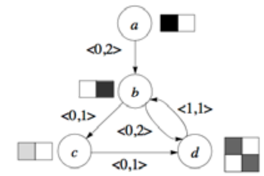
III. Example: An Acyclic Graph



Algorithm for Acyclic Graphs

- **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 = earliest n 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
  - no cyclic dependences in the loop

Cyclic Graphs



- No such thing as "topological order"
- b → c; c → b

$$S(c) - S(b) \geq 1$$
$$T + S(b) - S(c) \geq 2$$

- Scheduling b constrains c, and vice versa

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

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

Algorithm Design

- **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;

Modulo Variable Expansion

- **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     LD
 10.         ST      ADD             LD
 11.                                 MUL
 12.                 ST      ADD
 13.
 14.                         ST      ADD
```

```
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
```


Modulo Variable Expansion

```
    1.   LD R5,0(R1++)
    2.   LD R6,0(R1++)
    3.   LD R5,0(R1++)  MUL R7,R5,R6
    4.   LD R6,0(R1++)
    5.   LD R5,0(R1++)  MUL R17,R5,R6
    6.   LD R6,0(R1++)  ADD R8,R7,R7
L   7.   LD R5,0(R1++)  MUL R7,R5,R6
    8.   LD R6,0(R1++)  ADD R8,R17,R17  ST 0(R3++),R8
    9.   LD R5,0(R1++)  MUL R17,R5,R6
    10.  LD R6,0(R1++)  ADD R8,R7,R7    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 = $\max_r$ (lifetime$_r$ /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