Software Pipelining

I. Introduction
II. Problem Formulation
III. Algorithm

Reading: Chapter 10.5 - 10.6
I. Example of DoAll Loops

• **Machine:**
  - Per clock: 1 read, 1 write, 1 fully pipelined but 2-cycle ALU with hardware loop op and auto-incrementing addressing mode.

• **Source code:**
  
  ```
  For i = 1 to n
  A[i] = B[i];
  ```

• **Code for one iteration:**
  1. LD  R5,0(R1++)
  2. ST  0(R3++),R5

• **No parallelism in basic block**
Unroll

1. LD[i]
2. LD[i+1]    ST[i]
3.          ST[i+1]

• 2 iterations in 3 cycles
  1. LD[i]
  2. LD[i+1]    ST[i]
  3. LD[i+2]    ST[i+1]
  4. LD[i+3]    ST[i+2]
  5.          ST[i+3]

• 4 iterations in 5 cycles (harder to unroll by 3)
• U iterations in 1+U cycles
• As you unroll more
  – Better amortization
  – Bigger code size
  – Might exceed trip count of loop
Better way

1. LD[i]
2. LD[i+1] ST[i]
3. ST[i+1]

• 2 iterations in 3 cycles
  1. LD[i]
  2. LD[i+1] ST[i]
  3. LD[i+2] ST[i+1]
  4. LD[i+3] ST[i+2]
  5. ST[i+3]

Each cycle doing the same thing on a different iteration

• 4 iterations in 5 cycles (harder to unroll by 3)
• U iterations in 1+U cycles
• As you unroll more
  – Better amortization
  – Bigger code size
  – Might exceed trip count of loop
Better Way

LD[1]
loop N-1 times
   LD[i+1]      ST[i]

ST[n]

- N iterations in 2 + N-1 cycles
  - Performance of unrolling N times
  - Code size of unrolling twice
Software Pipelining

Every initiation interval (in this case 1 cycle), Iteration \( i \) enters the pipeline (i.e. the first instruction starts)
Iteration \( i-1 \) (maybe \( i-x \) in general) leaves the pipeline (i.e. the last instruction of iteration \( i-1 \) finishes)
Aside Unrolling

• Let’s say you have a VLIW machine with two multiply units

\[
\text{for } i = 1 \text{ to } n \\
A[i] = B[i] * C[i];
\]

Loop body will contain two loads, one multiply and one store where different instructions come from different iterations

No way to execute two multiples in parallel without unrolling
More Complicated Example

• **Source code:**

  For \( i = 1 \) 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. 
  5. **ADD** R8,R7,R4
  6. 
  7. **ST** 0 (R3++), R8
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
12.                             ST
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 with infinitely many iterations (assuming infinite registers)
Example of DoAcross Loop

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 (partially) 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

Complexity:
- NP-complete in general
Resources Bound Initiation Interval

- **Example:** Resource usage of 1 iteration; Machine can execute 1 LD, 1 ST, 2 ALU per clock
  
  \[ \text{LD, LD, MUL, ADD, ST} \]

- **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: Resource

- **RT**: resource reservation table for single iteration
- **RT_s**: modulo resource reservation table
  \[ RT_s[i] = \sum_{t \mid (t \mod T = i)} RT[t] \]

Schedule like list scheduling but use modulo reservation table
Scheduling Constraints: Precedence

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

- Minimum initiation interval, T?

- \( S(n) \): 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 \( T = (\Sigma \text{delays})/\text{iteration difference} \)

Min \( T = 4/1 \)
Scheduling Constraints: Precedence

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

- Minimum initiation interval? \((1+2+1)/2 = 2\)
- \(S(n)\): 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 $c$,
\[
\max_c \frac{\text{CycleLength}(c)}{\text{IterationDifference}(c)}
\]
III. Example: An Acyclic Graph

resource bound = 2
III. Example: An Acyclic Graph

resource bound = 2
III. Example: An Acyclic Graph: Instructions

resource bound = 2

HW will stall every other cycle
III. Example: An Acyclic Graph

resource bound = 2
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

Resource Bound = 3
Recurrence Bound = 3
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
  \[
  S(b) + 1 \leq S(c) \leq S(b) - 2 + T \\
  S(c) - T + 2 \leq S(b) \leq S(c) - 1
  \]
  \[
  S(d) - S(c) \geq 1 \\
  3 + S(c) - S(d) \leq 2, \ S(d) - S(c) \leq 1
  \]
Cyclic Graphs: T=4

- No such thing as “topological order”
- \( b \rightarrow c; c \rightarrow b \)

\[
S(c) - S(b) \geq 1 \\
T = 4 + S(b) - S(c) \geq 2
\]

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

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

\[
S(d) \geq S(c) + 1 \\
4 + S(c) - S(d) \leq 2, S(d) \leq S(c) + 2
\]
Cyclic Graphs: T=4

- No such thing as “topological order”
- \( b \to c; c \to b \)
  \[ S(c) - S(b) \geq 1 \]
  \[ T = 4 + S(b) - S(c) \geq 2 \]
- Scheduling \( b \) constrains \( c \) and vice versa
  \[ S(b) + 1 \leq S(c) \leq S(b) + 2 \]

\[ S(d) \geq S(c) + 1 \]
\[ 4 + S(c) - S(d) \leq 2, \quad S(d) \leq S(c) + 2 \]
**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 $T = T_0$, $T_0+1$, ..., until all nodes are scheduled
   For each node/SCC $c$ in topological order
      $s_0$ = Earliest $c$ can be scheduled
      For each $s = s_0$, $s_0 +1$, ..., $s_0 +T-1$
         If SCCScheduled($c$, $s$) break;
      If $c$ cannot be scheduled return false;
   Return true;
Scheduling a Strongly Connected Component (SCC)

\textbf{SCCScheduled}(c, s)

E* = longest path between each pair (min and max constraints)
Schedule first node at s, return false if fails
For each remaining node n in c
\begin{align*}
  s_l &= \text{lower bound on n based on E*} \\
  s_u &= \text{upper bound on n based on E*} \\
  \text{For each } s = s_l, s_l + 1, \min (s_l + T - 1, s_u) \\
  &\quad \text{if } \text{NodeScheduled}(n, s) \text{ break;}
\end{align*}
if n cannot be scheduled return false;
Return true;
Anti-dependences on Registers

• Traditional algorithm ignores them because can post-unroll

\begin{align*}
a1 &= ld[i] \\
a2 &= a1 + a1; \\
\text{Store}[i] &= a2; \\
\end{align*}

\begin{align*}
a1 &= ld[i+1]; \\
a2 &= a1+a1; \\
\text{Store}[i+1] &= a2; \\
\end{align*}
Anti-dependences on Registers

- Traditional algorithm ignores them because can post-unroll (or hw support)

\[
a1 = \text{ld}[i] \\
a2 = a1 + a1; \\
\text{Store}[i] = a2;
\]

\[
a3 = \text{ld}[i+1]; \\
a4 = a3 + a3; \\
\text{Store}[i+1] = a4;
\]

Modulo variable expansion

\[
u = \max_r \left( \text{lifetime}_r / T \right)
\]
**Anti-dependences on Registers**

- The code in every unrolled iteration is identical
  - Not ideal
- We unroll in two parts of the algorithm
- Instead, we can run the SWP algorithm for different unrolling factors. For each unroll, we pre-rename the registers but don’t ignore anti-dependences
  - Better potential results
  - But we might not find them
Register Allocation and SWP

• SWP schedules use lots of registers
  – Different schedules may use different amount of registers
• Use more back-tracking than described algorithm
  – If allocation fails, try to schedule again using different heuristic
• Schedule Spills
SWP versus No SWP

Speedup

-50% 0% 50% 100% 150% 200% 250% 300% 350% 400%

1 1073 2145

Speedup
Conclusions

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