15-745 Lecture 10

Instruction Scheduling
Software Pipelining

Copyright © Seth Copen Goldstein 2000-5
© Tim Callahan 2007

(some slides borrowed from M. Voss)

Instruction-level Parallelism

- Most modern processors have the ability to execute several adjacent instructions simultaneously:
  - Pipelined machines.
  - Very-long-instruction-word machines (VLIW).
  - Superscalar machines.
  - Dynamic scheduling/out-of-order machines.
- ILP is limited by several kinds of execution constraints:
  - Data dependence constraints.
  - Resource constraints ("hazards")
  - Control hazards

Execution Constraints

- Data-dependence constraints:
  - If instruction A computes a value that is read by instruction B, then B cannot execute before A is completed.
- Resource hazards:
  - Limited # of functional units:
    - If there are n functional units (e.g., multipliers), then only n instructions of a particular kind can execute at once.
  - Limited instruction issue:
    - If the instruction-issue unit can issue only n instructions at a time, then this limits ILP.
  - Limited register set:
    - Any schedule of instructions must have a valid register allocation.

Instruction Scheduling

- The purpose of instruction scheduling (IS) is to order the instructions for maximum ILP:
  - Keep all resources busy every cycle.
  - If necessary, eliminate data dependences and resource hazards to accomplish this.
- The IS problem is NP-complete (and bad in practice).
  - So heuristic methods are necessary.
Instruction Scheduling

• There are many different techniques for IS.
  - Still an open area of research.
• Most optimizing compilers perform good local IS, and only simple global IS.
• The biggest opportunities are in scheduling the code for loops.

Should the Compiler Do IS?

• Many modern machines perform dynamic reordering of instructions.
  - Also called “out-of-order execution” (OOOE).
  - Not yet clear whether this is a good idea.
  - Pro:
    • OOOE can use additional registers and register renaming to eliminate data dependences that no amount of static IS can accomplish.
    • No need to recompile programs when hardware changes.
  - Con:
    • OOOE means more complex hardware (and thus longer cycle times and more wattage).
    • And can’t be optimal since IS is NP-complete.

What we will cover

• Scheduling basic blocks
  - List scheduling
  - Long-latency operations
  - Delay slots
• Software Pipelining
• What we need to know
  - pipeline structure
  - data dependencies
  - register renaming
  - scalar replacement

Defining Dependencies

• Flow Dependence \( W \rightarrow R \quad \delta^f \) \{ true \}
• Anti-Dependence \( R \rightarrow W \quad \delta^a \) \{ false \}
• Output Dependence \( W \rightarrow W \quad \delta^o \)
• Input Dependence \( R \rightarrow R \quad \delta^i \)

S1) a=0; S2) b=a; S3) c=a+d+e; S4) d=b; S5) b=5+e; Not generally defined
Example Dependencies

S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;

δ f due to a
δ f due to a
δ f due to b
δ f due to d
δ f due to b

Renaming of Variables

- Sometimes constraints are not “real,” in the sense that a simple renaming of variables/registers can eliminate them.
  - Output dependence (WW): A and B write to the same variable.
  - Anti dependence (RW): A reads from a variable to which B writes.
- In such cases, the order of A and B cannot be changed unless variables are renamed.
  - Can sometimes be done by the hardware, to a limited extent.

Renaming of Variables

- Sometimes constraints are not “real,” in the sense that a simple renaming of variables/registers can eliminate them.
  - Output dependence (WW): A and B write to the same variable.
  - Anti dependence (RW): A reads from a variable to which B writes.
- In such cases, the order of A and B cannot be changed unless variables are renamed.
  - Can sometimes be done by the hardware, to a limited extent.

Register Renaming Example

\[
\begin{align*}
\text{r1} & \leftarrow r2 + 1 \\
[fp+8] & \leftarrow r1 \\
r1 & \leftarrow r3 + 2 \\
[fp+12] & \leftarrow r1 \\
\end{align*}
\]

\[
\begin{align*}
\text{r7} & \leftarrow r2 + 1 \\
[fp+8] & \leftarrow r7 \\
r1 & \leftarrow r3 + 2 \\
[fp+12] & \leftarrow r1 \\
\end{align*}
\]

\[
\begin{align*}
\text{r7} & \leftarrow r2 + 1 \\
r1 & \leftarrow r3 + 2 \\
[fp+8] & \leftarrow r7 \\
[fp+12] & \leftarrow r1 \\
\end{align*}
\]

• Can perform register renaming after register allocation
  - Constrained by available registers
  - Constrained by live on entry/exit
• Instead, do scheduling before register allocation

The Scheduler

- Given:
  - Code to schedule
  - Resources available (FU and # of Reg)
  - Latencies of instructions
- Goal:
  - Correct code
  - Better code [fewer cycles, less power, fewer registers, …]
  - Do it quickly
More Abstractly

- Given a graph $G = (V,E)$ where
  - nodes are operations
  - Each operation has an associated delay and type
  - edges between nodes represent dependencies
  - The number of resources of type $t$, $R(t)$
- A schedule assigns to each node a cycle number:
  - $\sigma(n) \geq 0$
  - If $(n,m) \in G$, $\sigma(m) \geq \sigma(n) + \text{delay}(n)$
  - $|\{ n | \sigma(n) = x \text{ and type}(n) = t\}| \leq R(t)$
- Goal is shortest length schedule, where length
  - $L(S) = \max \text{ over } n, \sigma(n) + \text{delay}(n)$

List Scheduling

- Keep a list of ready instructions, I.e.,
  - If we are at cycle $k$, then all predecessors, $p$, in graph have all been scheduled so that $\sigma(p) + \text{delay}(p) \leq k$
  - Alternate?
- Pick some instruction, $n$, from ready queue such that there are available resources for type($n$)
  - move $n$ from "ready" to "scheduled"
- Update lists and continue
  - maybe move some from "not ready" to "ready"

Lots of Heuristics

- forward or backward
- choose instructions on critical path
- ASAP or ALAP
- Balanced paths
- depth in schedule graph

Slack
Software Pipelining

- Software pipelining is an IS technique that reorders the instructions in a loop.
  - Possibly moving instructions from one iteration to the previous or the next iteration.
  - Very large improvements in running time are possible.

- The first serious approach to software pipelining was presented by Aiken & Nicolau.
  - Impractical as it ignores resource hazards (focusing only on data-dependence constraints).
    - But sparked a large amount of follow-on research.
Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration

Assume all have latency of 2

A: a ← ld [d]
B: b ← a * a
C: st [d], b
D: d ← d + 4

Can we decrease the latency?

- Lets unroll

A: a ← ld [d]
B: b ← a * a
C: st [d], b
D: d ← d + 4
A1: a ← ld [d]
B1: b ← a * a
C1: st [d], b1
D1: d ← d + 4

Rename variables

A: a ← ld [d]
B: b ← a * a
C: st [d], b
D: d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1: st [d1], b1
D1: d1 ← d1 + 4

Schedule
Unroll Some More

A: \( a \leftarrow \text{ld} \{d\} \)
B: \( b \leftarrow a \cdot a \)
C: \( \text{st} \{d\}, b \)
D: \( d_1 \leftarrow d + 4 \)
A1: \( a_1 \leftarrow \text{ld} \{d_1\} \)
B1: \( b_1 \leftarrow a_1 \cdot a_1 \)
C1: \( \text{st} \{d_1\}, b_1 \)
D1: \( d_2 \leftarrow d_1 + 4 \)
A2: \( a_2 \leftarrow \text{ld} \{d_2\} \)
B2: \( b_2 \leftarrow a_2 \cdot a_2 \)
C2: \( \text{st} \{d_2\}, b_2 \)
D2: \( d \leftarrow d_2 + 4 \)

One More Time

A B C D
D A1 B1 C1
D1 A2 B2 C2
D2 A3 B3 C3
D3 A4 B4 C4

Can Rearrange

A B C D4
D A1 B1 C1
D1 A2 B2 C2
D2 A3 B3 C3
D3 A4 B4 C4
Rearrange

A: a ← ld [d]
B: b ← a * a
C: st [d], b
D: d1 ← d + 4
A1: a1 ← ld [d1]
B1: b1 ← a1 * a1
C1: st [d1], b1
D1: d2 ← d1 + 4
A2: a2 ← ld [d2]
B2: b2 ← a2 * a2
C2: st [d2], b2
D2: d ← d2 + 4

SP Loop

A: a ← ld [d]
B: b ← a * a
D: d1 ← d + 4
A1: a1 ← ld [d1]
D1: d2 ← d1 + 4
B1: b1 ← a1 * a1
A2: a2 ← ld [d2]
D2: d ← d2 + 4
B2: b2 ← a2 * a2
C1: st [d1], b1
D3: d2 ← d1 + 4
C2: st [d2], b2

Goal of SP

- Increase distance between dependent operations by moving destination operation to a later iteration
**Goal of SP**

- Increase distance between dependent operations by moving destination operation to a later iteration
- But also, to uncover ILP across iteration boundaries!

### Example

Assume operating on a infinite wide machine

A0
A1
B0
A2
B1
C0
A3
B2
C1
B3
C2
C3

---

**Prolog**

\[
\text{for (i=0; i<N; i++) \{ }
\]

\[
\text{A_i B_{i-1} C_{i-2}}
\]

\[
\text{loop body}
\]

\[
\text{epilog}
\]

**Dealing with exit conditions**

\[
\text{for (i=0; i<N; i++) \{ }
\]

\[
\text{A_i B_{i-1} C_{i-2}}
\]

\[
\text{loop: A_i}
\]

\[
\text{if (i >= N) goto done A_0}
\]

\[
\text{B_1}
\]

\[
\text{if (i+1 == N) goto last i=1 A_1}
\]

\[
\text{C_{i-2}}
\]

\[
\text{if (i+2 == N) goto epilog i=2 A_2}
\]

\[
\text{B_{i-1}}
\]

\[
\text{if (i < N) goto loop}
\]

\[
\text{epilog: B_i}
\]

\[
\text{if (i < N) goto loop}
\]

\[
\text{last: C_{i-1}}
\]

\[
\text{done: C_i}
\]
Loop Unrolling V. SP

For SuperScalar or VLIW
- Loop Unrolling reduces loop overhead
- Software Pipelining reduces fill/drain
- Best is if you combine them

Aiken/Nicolau Scheduling

Step 1

Perform scalar replacement to eliminate memory references where possible.

for i:=1 to N do
  a := j @ V[i-1]
  b := a @ f
  c := e @ j
  d := f @ c
  e := b @ d
  f := U[i]
  g: V[i] := b
  h: W[i] := d
  j := X[i]

Step 2

Unroll the loop and compute the data-dependence graph (DDG).

DDG for rolled loop:

for i:=1 to N do
  a := j @ b
  b := a @ f
  c := e @ j
  d := f @ c
  e := b @ d
  f := U[i]
  g: V[i] := b
  h: W[i] := d
  j := X[i]

Aiken/Nicolau Scheduling

Step 2, cont’d

DDG for unrolled loop:

for i:=1 to N do
  a := j @ b
  b := a @ f
  c := e @ j
  d := f @ c
  e := b @ d
  f := U[i]
  g: V[i] := b
  h: W[i] := d
  j := X[i]
Aiken/Nicolau Scheduling

Step 3

Build a tableau of iteration number vs cycle time.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h_3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

basically, you're emulating a superscalar with infinite resources, infinite register renaming, always predicting the loop-back branch: thus, just pure data dependency.

Aiken/Nicolau Scheduling

Step 4

Find repeating patterns of instructions.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

basically, you're emulating a superscalar with infinite resources, infinite register renaming, always predicting the loop-back branch: thus, just pure data dependency.

Aiken/Nicolau Scheduling

Step 4

Find repeating patterns of instructions.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

basically, you're emulating a superscalar with infinite resources, infinite register renaming, always predicting the loop-back branch: thus, just pure data dependency.
### Aiken/Nicolau Scheduling

#### Step 4

Find repeating patterns of instructions.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

#### Step 5

"Coalesce" the slopes.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

Go back and relate slopes to DDG

#### Step 6

Find the loop body and "reroll" the loop.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
</tbody>
</table>

Prologue/entry code

Loop body

Epilogue/exit code
Aiken/Nicolau Scheduling

Step 7

Generate code.
(Assume VLIW-like machine for this example. The instructions on each line should be issued in parallel.)

\[
\begin{align*}
    &a_1 := b_0 \oplus c_1 := a_0 \oplus j_0 \oplus f_1 := U[1] \oplus j_1 := X[1] \\
    &b_1 := a_1 \oplus e_0 \oplus d_1 := f_0 \oplus c_1 := U[2] \oplus j_2 := X[2] \\
    &c_1 := b_0 \oplus d_0 := V[1] := b_0 \oplus W[1] := d_1 \oplus a_2 := j_1 \oplus b_1 \\
    &d_0 := c_0 \oplus e_1 := b_1 \oplus f_1 := U[2] \oplus j_3 := X[3] \\
    &e_0 := b_0 \oplus d_0 := W[2] := d_0 \oplus a_3 := j_2 \oplus b_2 \\
    &f_1 := b_1 \oplus c_1 := X[3] \\
\end{align*}
\]

Step 8

• Since several versions of a variable (e.g., \( j_i \) and \( j_{i+1} \)) might be live simultaneously, we need to add new temps and moves.

\[
\begin{align*}
    &a_1 := b_0 \oplus c_1 := e_0 \oplus f_0 := U[1] \oplus j_1 := X[1] \\
    &b_1 := a_1 \oplus e_0 \oplus d_1 := f_0 \oplus c_1 := U[2] \oplus j_2 := X[2] \\
    &c_1 := b_0 \oplus d_0 := V[1] := b_0 \oplus W[1] := d_1 \oplus a_2 := j_1 \oplus b_1 \\
    &d_0 := c_0 \oplus e_1 := b_1 \oplus f_1 := U[2] \oplus j_3 := X[3] \\
    &e_0 := b_0 \oplus d_0 := W[2] := d_0 \oplus a_3 := j_2 \oplus b_2 \\
    &f_1 := b_1 \oplus c_1 := X[3] \\
\end{align*}
\]

Next Step in SP

• AN88 did not deal with resource constraints.
• Modulo Scheduling is a SP algorithm that does.
• It schedules the loop based on
  - resource constraints
  - precedence constraints
• Basically, it’s list scheduling that takes into account resource conflicts from overlapping iterations.
Resource Constraints

- Minimally indivisible sequences, \( i \) and \( j \), can execute together if combined resources in a step do not exceed available resources.
- \( R(i) \) is a resource configuration vector
  - \( R(i) \) is the number of units of resource \( i \)
- \( r(i) \) is a resource usage vector s.t.
  - \( 0 \leq r(i) \leq R(i) \)
- Each node in \( G \) has an associated \( r(i) \)

Software Pipelining Goal

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, \( s \)
- Goal: minimize \( s \)
**Software Pipelining Goal**

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, $s$
- Goal: minimize $s$.

resources must be within constraints

**Software Pipelining Goal**

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, $s$
- Goal: minimize $s$.

resources must be within constraints

**Software Pipelining Goal**

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, $s$
- Goal: minimize $s$.

resources must be within constraints

**Software Pipelining Goal**

- Find the same schedule for each iteration.
- Stagger by iteration initiation interval, $s$
- Goal: minimize $s$.

resources must be within constraints

modulo resource table
**Precedence Constraints**

- Review: for acyclic scheduling, constraint is just the required delay between two ops $u, v$: $\langle d(u,v) \rangle$

- For an edge, $u \rightarrow v$, we must have $\sigma(v) - \sigma(u) \geq d(u,v)$

**Precedence Constraints**

- Cyclic: constraint becomes a tuple: $\langle p,d \rangle$
  - $p$ is the minimum iteration delay (or the loop carried dependence distance)
  - $d$ is the delay

- For an edge, $u \rightarrow v$, we must have $\sigma(v) - \sigma(u) \geq d(u,v) - sp(u,v)$

  - $p \geq 0$

  - If data dependence is
    - within an iteration, $p=0$
    - loop-carried across $p$ iter boundaries, $p>0$

**Iterative Approach**

- Finding minimum $S$ that satisfies the constraints is NP-Complete.

- Heuristic:
  - Find lower and upper bounds for $S$
  - foreach $s$ from lower to upper bound?
    - Schedule graph.
      - If succeed, done
      - Otherwise try again (with next higher $s$)

- Thus: “Iterative Modulo Scheduling” Rau, et.al.

**Iterative Approach**

- Heuristic:
  - Find lower and upper bounds for $S$
  - foreach $s$ from lower to upper bound
    - Schedule graph.
      - If succeed, done
      - Otherwise try again (with next higher $s$)

- So the key difference:
  - AN88 does not assume $S$ when scheduling
  - IMS must assume an $S$ for each scheduling attempt to understand resource conflicts
Lower Bounds

- Resource Constraints: $S_R$ (also called $\Pi_{res}$)
  maximum over all resources of $\#$ of uses divided by $\#$ available... rounded up or down?

- Precedence Constraints: $S_E$ (also called $\Pi_{rec}$)
  max over all cycles: $d(c)/p(c)$

In practice, one is easy, other is hard.
Tim's secret approach: just use $S_R$ as lower bound, then do binary search for best $S$

Acyclic Example

Lower Bound: $S_R=2$
Upper Bound: 5

Lower Bound on $s$

- Assume 1 ALU and 1 MU
- Assume latency Op or load is 1 cycle

Scheduling data structures

To schedule for initiation interval $s$:
- Create a resource table with $s$ rows and $R$ columns
- Create a vector, $\sigma$, of length $N$ for $n$ instructions in the loop
  - $\sigma[n] =$ the time at which $n$ is scheduled, or NONE
- Prioritize instructions by some heuristic
  - critical path (or cycle)
  - resource critical
Scheduling algorithm

- Pick an instruction, n
- Calculate earliest time due to dependence constraints
  
  For all x=pred(n),
  
  \[ \text{earliest} = \max(\text{earliest}, \sigma(n)+d(n,x)-s_p(x,n)) \]
- try and schedule n from earliest to (earliest+s-1)
  s.t. resource constraints are obeyed.
  - possible twist: deschedule a conflicting node to make way for n, maybe randomly, like sim anneal
- If we fail, then this schedule is faulty
  (i.e. give up on this s)

Scheduling algorithm - cont.

- We now schedule n at earliest, i.e., \( \sigma(n) = \text{earliest} \)
- Fix up schedule
  - Successors, x, of n must be scheduled s.t.
    \[ \sigma(x) := \sigma(n)+d(n,x)-s_p(n,x) \]
    otherwise they are removed (descheduled) and put back on worklist.
- repeat this some number of times until either
  - succeed, then register allocate
  - fail, then increase s

Simplest Example

```plaintext
for () {
  a = b + c
  b = a * a
  c = a * 194
}
```

What is IIres?
What is IIrec?

Simplest Example

```plaintext
for () {
  a = b + c
  b = a * a
  c = a * 194
}
```

Try II = 2

Modulo Resource Table:

```
0 1 1
1
```
**Simplest Example**

```
for () {
    a = b+c
    b = a*a
    c = a*194
}
```

Try II = 2

Modulo Resource Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**earliest a: sigma(c) + delay(c) - 2 = 2+1-2 = 1**

```
for () {
    a = b+c
    b = a*a
    c = a*194
}
```

Try II = 2

Modulo Resource Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

earliest b?  
scheduled b?  
what next?
Simplest Example

```c
for () {
    a = b + c
    b = a * a
    c = a * 194
}
```

Try II = 2

Modulo Resource Table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Lesson: lower bound may not be achievable

Example

```c
for i := 1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
    g := V[i] := b
    h := W[i] := d
    j := X[i]
```

Priorities: c,d,e,a,b,f,j,g,h

Example

```c
for i := 1 to N do
    a := j ⊕ b
    b := a ⊕ f
    c := e ⊕ j
    d := f ⊕ c
    e := b ⊕ d
    f := U[i]
    g := V[i] := b
    h := W[i] := d
    j := X[i]
```

Priorities: c,d,e,a,b,f,j,g,h

```
 instr | \sigma \\
--------|
 a      |   \\
 b      |   \\
 c      |   \\
 d      |   \\
 e      |   \\
 f      |   \\
 g      |   \\
 h      |   \\
 j      |   \\
```

```
 ALU | MU \\
-------|------|
 a    |   \\
 b    |   \\
 c    |   \\
 d    |   \\
 e    |   \\
 f    |   \\
 g    |   \\
 h    |   \\
 j    |   \\
```

s = 5
for i:=1 to N do
a := j ⊕ b
b := a ⊕ f
c := e ⊕ j
d := f ⊕ c
e := b ⊕ d
f := U[i]
g: V[i] := b
h: W[i] := d
j := X[i]

Priorities: a,b,f,j,g,h

s=5

for i:=1 to N do
a := j ⊕ b
b := a ⊕ f
c := e ⊕ j
d := f ⊕ c
e := b ⊕ d
f := U[i]
g: V[i] := b
h: W[i] := d
j := X[i]

Priorities: b,f,j,g,h

s=5

for i:=1 to N do
a := j ⊕ b
b := a ⊕ f
c := e ⊕ j
d := f ⊕ c
e := b ⊕ d
f := U[i]
g: V[i] := b
h: W[i] := d
j := X[i]

Priorities: b,f,j,g,h

s=5

for i:=1 to N do
a := j ⊕ b
b := a ⊕ f
c := e ⊕ j
d := f ⊕ c
e := b ⊕ d
f := U[i]
g: V[i] := b
h: W[i] := d
j := X[i]

Priorities: e,f,j,g,h

s=5

b causes b→e edge violation
for i:=1 to N do
  a := j \oplus b
  b := a \oplus f
  c := e \oplus j
  d := f \oplus c
  e := b \oplus d
  f := U[i]
  g := V[i] := b
  h := W[i] := d
  j := X[i]

Priorities: e,f,j,g,h

\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow j \]

s=5

for i:=1 to N do
  a := j \oplus b
  b := a \oplus f
  c := e \oplus j
  d := f \oplus c
  e := b \oplus d
  f := U[i]
  g := V[i] := b
  h := W[i] := d
  j := X[i]

Priorities: f,j,g,h

\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow j \]

s=5

for i:=1 to N do
  a := j \oplus b
  b := a \oplus f
  c := e \oplus j
  d := f \oplus c
  e := b \oplus d
  f := U[i]
  g := V[i] := b
  h := W[i] := d
  j := X[i]

Priorities: g,h

\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow j \]

s=5

for i:=1 to N do
  a := j \oplus b
  b := a \oplus f
  c := e \oplus j
  d := f \oplus c
  e := b \oplus d
  f := U[i]
  g := V[i] := b
  h := W[i] := d
  j := X[i]

Priorities: g,h

\[ a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow j \]

s=5
Creating the Loop

- Create the body from the schedule.
- Determine which iteration an instruction falls into
  - Mark its sources and dest as belonging to that iteration.
  - Add Moves to update registers
- Prolog fills in gaps at beginning
  - For each move we will have an instruction in prolog, and we fill in dependent instructions
- Epilog fills in gaps at end

<table>
<thead>
<tr>
<th>instr</th>
<th>arg</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
</tr>
<tr>
<td>d</td>
<td>6</td>
</tr>
<tr>
<td>e</td>
<td>7</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
</tr>
<tr>
<td>g</td>
<td>7</td>
</tr>
<tr>
<td>h</td>
<td>8</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
</tr>
</tbody>
</table>

Conditionals

- What about internal control structure, i.e., conditionals
- Three approaches
  - Schedule both sides and use conditional moves
  - Schedule each side, then make the body of the conditional a macro op with appropriate resource vector
  - Trace schedule the loop

```
f0 = U[0];
j0 = X[0];

FOR i = 0 to N
  f1 := U[i+1]
  j1 := X[i+1]
  nop
  a := j0 ? b
  b := a ? f0
  c := e ? j0
  d := f0 ? c
  e := b ? d
  g: V[i] := b
  h: W[i] := d
  f0 = f1
  j0 = j1
```

What to take away

- Dependence analysis is very important
- Software pipelining is cool
- Registers are a key resource