Software Pipelining
- Modulo Scheduling

University of Michigan

October 15, 2014
Announcements + Reading Material

❖ HW 2 Due this Thursday

❖ Today’s class reading

❖ Next next Wednesday’s reading
Machine: 2 issue, 1 memory port, 1 ALU
Memory port = 2 cycles, pipelined
ALU = 1 cycle

1. Calculate height-based priorities
2. Schedule using Operation scheduler
Generalize Beyond a Basic Block

❖ Superblock
  » Single entry
  » Multiple exits (side exits)
  » No side entries

❖ Schedule just like a BB
  » Priority calculations needs change
  » Dealing with control deps
Change Focus to Scheduling Loops

Most of program execution time is spent in loops

Problem: How do we achieve compact schedules for loops

```c
for (j=0; j<100; j++)
b[j] = a[j] * 26
```

```c
r1 = _a
r2 = _b
r9 = r1 * 4
r1 = r1 + 4
r2 = r2 + 4
p1 = cmpp (r1 < r9)
brct p1 Loop
```
Basic Approach – List Schedule the Loop Body

<table>
<thead>
<tr>
<th>Iteration</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
</tr>
</tbody>
</table>

Schedule each iteration
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop

<table>
<thead>
<tr>
<th>time</th>
<th>ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 4</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>3, 5, 7</td>
</tr>
</tbody>
</table>

Total time = 6 * n
Unroll Then Schedule Larger Body

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>0,1</td>
</tr>
<tr>
<td>3,4</td>
<td>1,2</td>
</tr>
<tr>
<td>5,6</td>
<td>3,4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>n-1,n</td>
<td>6,7</td>
</tr>
</tbody>
</table>

Schedule each iteration
resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, cmpp = 1, mpy=3, ld = 2, st = 1, br = 1

1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop

time | ops
-----|-----
0    | 1, 4
1    | 1’, 6, 4’
2    | 2, 6’
3    | 2’
4    | -
5    | 3, 5, 7
6    | 3’,5’,7’

Total time = 7 * n/2
Problems With Unrolling

❖ Code bloat
  » Typical unroll is 4-16x
  » Use profile statistics to only unroll “important” loops
  » But still, code grows fast

❖ Barrier after across unrolled bodies
  » I.e., for unroll 2, can only overlap iterations 1 and 2, 3 and 4, …

❖ Does this mean unrolling is bad?
  » No, in some settings its very useful
    Ṣ Low trip count
    Ṣ Lots of branches in the loop body
  » But, in other settings, there is room for improvement
Overlap Iterations Using Pipelining

With hardware pipelining, while one instruction is in fetch, another is in decode, another in execute. Same thing here, multiple iterations are processed simultaneously, with each instruction in a separate stage. 1 iteration still takes the same time, but time to complete n iterations is reduced!
A Software Pipeline

Loop body with 4 ops

Steady state: 4 iterations executed simultaneously, 1 operation from each iteration. Every cycle, an iteration starts and finishes when the pipe is full.

Prologue - fill the pipe

Kernel – steady state

Epilogue - drain the pipe
Creating Software Pipelines

❖ Lots of software pipelining techniques out there
❖ Modulo scheduling
   » Most widely adopted
   » Practical to implement, yields good results
❖ Conceptual strategy
   » Unroll the loop completely
   » Then, schedule the code completely with 2 constraints
      ✈ All iteration bodies have identical schedules
      ✈ Each iteration is scheduled to start some fixed number of cycles later than the previous iteration
   » \textit{Initiation Interval} (II) = fixed delay between the start of successive iterations
   » Given the 2 constraints, the unrolled schedule is repetitive (kernel) except the portion at the beginning (prologue) and end (epilogue)
      ✈ Kernel can be re-rolled to yield a new loop
Creating Software Pipelines (2)

❖ Create a schedule for 1 iteration of the loop such that when the same schedule is repeated at intervals of II cycles
   » No intra-iteration dependence is violated
   » No inter-iteration dependence is violated
   » No resource conflict arises between operation in same or distinct iterations

❖ We will start out assuming Itanium-style hardware support, then remove it later
   » Rotating registers
   » Predicates
   » Software pipeline loop branch
Terminology

Initiation Interval (II) = fixed delay between the start of successive iterations

Each iteration can be divided into stages consisting of II cycles each

Number of stages in 1 iteration is termed the stage count (SC)

Takes SC-1 cycles to fill/drain the pipe
Resource Usage Legality

❖ Need to guarantee that
  » No resource is used at 2 points in time that are separated by an interval which is a multiple of II
  » I.E., within a single iteration, the same resource is never used more than 1x at the same time modulo II
  » Known as modulo constraint, where the name modulo scheduling comes from
  » Modulo reservation table solves this problem
    Ŷ To schedule an op at time T needing resource R
      ♦ The entry for R at T mod II must be free
    Ŷ Mark busy at T mod II if schedule

<table>
<thead>
<tr>
<th>II = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
Dependences in a Loop

❖ Need worry about 2 kinds
  » Intra-iteration
  » Inter-iteration

❖ Delay
  » Minimum time interval between the start of operations
  » Operation read/write times

❖ Distance
  » Number of iterations separating the 2 operations involved
  » Distance of 0 means intra-iteration

❖ Recurrence manifests itself as a circuit in the dependence graph

Edges annotated with tuple <delay, distance>
Dynamic Single Assignment (DSA) Form

Impossible to overlap iterations because each iteration writes to the same register. So, we’ll have to remove the anti and output dependences.

Virtual rotating registers
* Each register is an infinite push down array (Expanded virtual reg or EVR)
* Write to top element, but can reference any element
* Remap operation slides everything down \(\rightarrow\) \(r[n]\) changes to \(r[n+1]\)

A program is in DSA form if the same virtual register (EVR element) is never assigned to more than 1x on any dynamic execution path

```
1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop
```

```
1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
6: p1[-1] = cmpp (r1[-1] < r9)
7: brct p1[-1] Loop
```
Physical Realization of EVRs

- EVR may contain an unlimited number values
  - But, only a finite contiguous set of elements of an EVR are ever live at any point in time
  - These must be given physical registers

- Conventional register file
  - Remaps are essentially copies, so each EVR is realized by a set of physical registers and copies are inserted

- Rotating registers
  - Direct support for EVRs
  - No copies needed
  - File “rotated” after each loop iteration is completed
Loop Dependence Example

1: r3[-1] = load(r1[0])
2: r4[-1] = r3[-1] * 26
3: store (r2[0], r4[-1])
4: r1[-1] = r1[0] + 4
5: r2[-1] = r2[0] + 4
6: p1[-1] = cmpp (r1[-1] < r9)
remap r1, r2, r3, r4, p1
7: brct p1[-1] Loop

In DSA form, there are no inter-iteration anti or output dependences!
Class Problem

Latencies: ld = 2, st = 1, add = 1, cmpp = 1, br = 1

1: \texttt{r1[-1] = load(r2[0])}
3: \texttt{store (r3[-1], r2[0])}
4: \texttt{r2[-1] = r2[0] + 4}
5: \texttt{p1[-1] = cmpp (r2[-1] < 100)}
remap r1, r2, r3
6: \texttt{brct p1[-1] Loop}

Draw the dependence graph showing both intra and inter iteration dependences
Minimum Initiation Interval (MII)

- Remember, $II = \text{number of cycles between the start of successive iterations}$
- Modulo scheduling requires a candidate $II$ be selected before scheduling is attempted
  - Try candidate $II$, see if it works
  - If not, increase by 1, try again repeating until successful
- MII is a lower bound on the $II$
  - $MII = \text{Max}(\text{ResMII}, \text{RecMII})$
  - $\text{ResMII} = \text{resource constrained MII}$
    - Resource usage requirements of 1 iteration
  - $\text{RecMII} = \text{recurrence constrained MII}$
    - Latency of the circuits in the dependence graph
ResMII

Concept: If there were no dependences between the operations, what is the shortest possible schedule?

Simple resource model

A processor has a set of resources $R$. For each resource $r$ in $R$ there is $\text{count}(r)$ specifying the number of identical copies

$$\text{ResMII} = \max \quad \left( \frac{\text{uses}(r)}{\text{count}(r)} \right)$$

for all $r$ in $R$

$$\text{uses}(r) = \text{number of times the resource is used in 1 iteration}$$

In reality its more complex than this because operations can have multiple alternatives (different choices for resources it could be assigned to), but we will ignore this for now.
ResMII Example

resources: 4 issue, 2 alu, 1 mem, 1 br
latencies: add=1, mpy=3, ld = 2, st = 1, br = 1

1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop

ALU: used by 2, 4, 5, 6  
  → 4 ops / 2 units = 2

Mem: used by 1, 3  
  → 2 ops / 1 unit = 2

Br: used by 7  
  → 1 op / 1 unit = 1

ResMII = MAX(2,2,1) = 2
RecMII

Approach: Enumerate all irredundant elementary circuits in the dependence graph

\[
\text{RecMII} = \max \left( \frac{\text{delay}(c)}{\text{distance}(c)} \right) 
\]

for all \( c \) in \( C \)

delay(\( c \)) = total latency in dependence cycle \( c \) (sum of delays)
distance(\( c \)) = total iteration distance of cycle \( c \) (sum of distances)

delay(\( c \)) = 1 + 3 = 4
distance(\( c \)) = 0 + 1 = 1
RecMII = 4/1 = 4
RecMII Example

1: r3 = load(r1)
2: r4 = r3 * 26
3: store (r2, r4)
4: r1 = r1 + 4
5: r2 = r2 + 4
6: p1 = cmpp (r1 < r9)
7: brct p1 Loop

4 \rightarrow 4: 1 / 1 = 1
5 \rightarrow 5: 1 / 1 = 1
4 \rightarrow 1 \rightarrow 4: 1 / 1 = 1
5 \rightarrow 3 \rightarrow 5: 1 / 1 = 1

RecMII = MAX(1,1,1,1) = 1

Then,

MII = MAX(ResMII, RecMII)
MII = MAX(2,1) = 2

<delay, distance>
Class Problem

Latencies: \( \text{ld} = 2, \text{st} = 1, \text{add} = 1, \text{cmpp} = 1, \text{br} = 1 \)
Resources: 1 ALU, 1 MEM, 1 BR

1: \( r1[-1] = \text{load}(r2[0]) \)
3: \( \text{store}(r3[-1], r2[0]) \)
4: \( r2[-1] = r2[0] + 4 \)
5: \( p1[-1] = \text{cmpp}(r2[-1] < 100) \)
remap \( r1, r2, r3 \)
6: \( \text{brct}\ p1[-1] \text{ Loop} \)

Calculate RecMII, ResMII, and MII
Modulo Scheduling Process

- Use list scheduling but we need a few twists
  - II is predetermined – starts at MII, then is incremented
  - Cyclic dependences complicate matters
    - Estart/Priority/etc.
    - Consumer scheduled before producer is considered
      - There is a window where something can be scheduled!
  - Guarantee the repeating pattern

- 2 constraints enforced on the schedule
  - Each iteration begin exactly II cycles after the previous one
  - Each time an operation is scheduled in 1 iteration, it is tentatively scheduled in subsequent iterations at intervals of II
    - MRT used for this
Priority Function

Height-based priority worked well for acyclic scheduling, makes sense that it will work for loops as well

**Acyclic:**

\[
\text{Height}(X) = \begin{cases} 
0, & \text{if } X \text{ has no successors} \\
\max & ((\text{Height}(Y) + \text{Delay}(X,Y)), \text{otherwise}) \\
\text{for all } Y = \text{succ}(X) 
\end{cases}
\]

**Cyclic:**

\[
\text{Height}_R(X) = \begin{cases} 
0, & \text{if } X \text{ has no successors} \\
\max & ((\text{Height}_R(Y) + \text{EffDelay}(X,Y)), \text{otherwise}) \\
\text{for all } Y = \text{succ}(X) 
\end{cases}
\]

\[
\text{EffDelay}(X,Y) = \text{Delay}(X,Y) - \text{II} \times \text{Distance}(X,Y)
\]
Calculating Height

1. Insert pseudo edges from all nodes to branch with latency = 0, distance = 0 (dotted edges)
2. Compute II, For this example assume II = 2
3. $\text{HeightR}(4) =$
4. $\text{HeightR}(3) =$
5. $\text{HeightR}(2) =$
6. $\text{HeightR}(1)$
The Scheduling Window

With cyclic scheduling, not all the predecessors may be scheduled, so a more flexible earliest schedule time is:

\[
E(Y) = \begin{cases} 
\max \text{ for all } X = \text{pred}(Y) & \text{0, if } X \text{ is not scheduled} \\
\max(0, \text{SchedTime}(X) + \text{EffDelay}(X,Y)) & \text{otherwise}
\end{cases}
\]

where \(\text{EffDelay}(X,Y) = \text{Delay}(X,Y) - II \times \text{Distance}(X,Y)\)

Every II cycles a new loop iteration will be initialized, thus every II cycles the pattern will repeat. Thus, you only have to look in a window of size II, if the operation cannot be scheduled there, then it cannot be scheduled.

\[
\text{Latest schedule time}(Y) = L(Y) = E(Y) + II - 1
\]
To be continued…