Module: Software Instruction Scheduling – Part I

Reading for this Module

- Loop Unrolling and Instruction Scheduling
  - Section 2.2

- Dependence Analysis
  - Section 2.1

- Optimizations
  - Appendix G.2
Compiler Hardware Interface

Hardware/Software Interface: The ISA

Instruction Scheduling

- A compiler can enhance a processor’s ability to exploit instruction level parallelism

- Program dependencies can be preserved via different sequence of instructions
  - Some sequences are more efficient than others, e.g., less stall cycles
  - Schedule to avoid hazards

- Instruction scheduling is a complex optimization problem
Opportunity in Superscalars

- High degree of *Instruction Level Parallelism (ILP)* via multiple (possibly) *pipelined functional units (FUs)*.
  
  Essential to harness promised performance.

- Clean simple model and Instruction Set makes *compile-time* optimizations feasible.

- Therefore, performance advantages can be harnessed automatically

---

Example of ILP

Processor components

- 5 functional units: 2 fixed point units, 2 floating point units and 1 branch unit.

- Pipeline depth: floating point unit is 2 deep, and the others are 1 deep.

Peak rates: 7 instructions being processed simultaneously in each cycle

- *Given a source program P, schedule the instructions so as to minimize the overall execution time on the functional units in the target machine.*
Cost Functions

- **Effectiveness** of the Optimizations: *How well can we optimize our objective function?*
  Impact on running time of the *compiled* code determined by the completion time.

- **Efficiency** of the optimization: *How fast can we optimize?*
  Impact on the time it takes to compile or cost for gaining the benefit of code with fast running time.

---

**Example**

**Original Loop**

```
Loop: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, Loop
```

**Scheduled Loop**

```
Loop: L.D F0, 0(R1)
stall
ADD.D F4, F0, F2
stall
S.D F4, 0(R1)
stall
DADDUI R1, R1, #-8
stall
BNE R1, R2, Loop
```

<table>
<thead>
<tr>
<th>Source</th>
<th>Consumer</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU</td>
<td>FP ALU</td>
<td>3 (clocks)</td>
</tr>
<tr>
<td>FP ALU</td>
<td>Store</td>
<td>2</td>
</tr>
<tr>
<td>Load</td>
<td>FP ALU</td>
<td>1</td>
</tr>
<tr>
<td>Load</td>
<td>Store</td>
<td>0</td>
</tr>
</tbody>
</table>

#intervening clock cycles for dependent instructions
Scheduled Loop Body

Loop:  
- L.D  F0, 0(R1)
- DADDUI  R1, R1, #-8
- ADD.D  F4, F0, F2
- stall
- stall
- S.D  F4, 0(R1)
- BNE  R1, R2, Loop

- Schedule instructions to fill stall cycles and honor flow dependencies
- Apply to any basic block in the program
- How can we increase the available instructions to schedule?

Enhancing Parallelism

- Where does concurrency exist?  
  - across instructions
  - across loops
  - How do we discover it?  
  - Program analysis
    - Dependence analysis
    - Aliasing analysis

- How do we enhance it?  
  - Program transformations:
    - loop unrolling
    - trace generation (later)
- How do we exploit it?  
  - instruction scheduling
    - hardware
    - software pipelining (later)
**Loop Unrolling**

For (i = 0; i < 4; i++) {
    \(c[2i] = a[2i] \times X + Y;\)
    \(c[2i+1] = a[2i+1] \times X + Y;\)
}

- Loops are syntactic constructs for specifying repetitive code sequences
  - We would prefer longer code sequences
  - Create longer code sequences via unrolling

- Replicate the loop body multiple times

- Example

---

**The Unrolled Loop: The Simple Case**

Loop:

```
L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, Loop
L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, Loop
L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, Loop
L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, Loop
...```

What about loop termination? (need to transform)

Output dependencies (need more registers)
The Unscheduled and Unrolled Loop

Loop:  
L.D F0, 0(R1)  
ADD.D F4, F0, F2  
S.D F4, 0(R1)  
L.D F6, -8(R1)  
ADD.D F8, F6, F2  
S.D F8, -8(R1)  
L.D F10, -16(R1)  
ADD.D F12, F10, F2  
S.D F12, -16(R1)  
L.D F14, -24(R1)  
ADD.D F16, F14, F2  
S.D F16, -24(R1)  
DADDUI R1, R1, #-32  
BNE R1, R2, Loop

Modified loop termination condition via merging (DADDUI) and removal (BNE)
Use of additional registers → register pressure
Address compensation

With stalls → 6.75 cycles/element

Unrolled and Scheduled Loop

Loop:  
L.D F0, 0(R1)  
L.D F6, -8(R1)  
L.D F10, -16(R1)  
L.D F14, -24(R1)  
ADD.D F4, F0, F2  
ADD.D F8, F6, F2  
ADD.D F12, F10, F2  
ADD.D F16, F14, F2  
S.D F4, 0(R1)  
S.D F8, -8(R1)  
DADDUI R1, R1, #-32  
BNE R1, R2, Loop

Use additional registers to recover concurrency and create ILP

With scheduling → no stalls and 3.5 cycles/element
Key Elements of Loop Unrolling

- The overhead of loop termination checks and branching is amortized
  - More useful work is performed per branch

- There is greater demand for registers
  - Register pressure → more concurrency means more resources!

- Symbolic optimizations
  - Need to recognize memory address dependencies
  - Symbolic optimizations to adjust addresses

- Instruction scheduling to hide hazards

---

Key Elements of Loop Unrolling

- Using the loop to optimize its dynamic behavior is a challenging problem.

- Hard to optimize well without detailed knowledge of the range of the iteration.
  - Program transformations

- In practice, profiling can offer limited help in estimating loop bounds.
  - Use of pragmas to communicate information to the optimizer
Some Pragmatics of Loop Unrolling

- Do we always know the loop iteration count?

- **Key idea**: transform the loop into two loops
  - Assume the loop iteration count is \( n \), and is determined at run time
  - The first loop is unrolled \( k \) times and executes \( n/k \) times
  - The second loop contains a single copy of the loop body and is executed \((n \mod k)\) times

Example

Loops with Data Dependent Iteration Count

```c
for (i=0; i<n; i++)
    c[i] = a[i] * X + Y;

for (i=0; i<m; i++)
    c[k*i] = a[k*i] * X + Y;
    ..
    ..
    ..

for (i=0; i<(n mod k); i++)
    c[m*k+i] = a[m*k+i] * X + Y;
    ..
    ..
    ..
```

\[ m = \left\lfloor \frac{n}{k} \right\rfloor \]
Loop Carried Dependencies

/* prologue code */
for (i=6; i<n; i++) {
    a[i] = a[i-5] \times X + Y;
}
/* epilogue code */

- Iteration level parallelism refers to concurrent execution of different iterations of a loop
- Iteration level parallelism is determined by loop carried dependencies
  - Consider data dependencies:
    - Data produced in one iteration is used in a later iteration
    - Concept of dependence distance

Loop Level Parallelism

- The goal is to determine those iterations that can be executed in parallel
- Special cases: loop carried dependencies may be in the form of recurrences
  for (i=6; i<100; i++) {
    a[i] = a[i-5] + b[i];
  }
  - Dependence distance can be used to determine the degree of unrolling
    - Unrolling can produce independent instructions in the unrolled loop body
Detecting Loop Carried Data Dependencies

- Distinguish between dependencies that *may* be to the same location vs. those that *are* to the same location
  - How do we detect such dependences?

  ```c
  for (i=0; i<N; i++) {
  }
  ```

Importance of Detecting Loop Carried Dependencies

- Necessary to detect and exploit parallelism at the level of loop iterations
- Necessary for generating good instruction schedules
- Pointers significantly complicate the analysis
A Simple Test

- A large class of dependence analysis algorithms assume that the array indices are computed as affine functions
  
  \[ X[a*i + b], \text{ where } a \text{ and } b \text{ are constants and } i \text{ is the loop index variable} \]

- For such references, a dependence exists if for two reference to indices \([a*i + b]\) and \([c*k + d]\)
  \[(b-d)/\text{gcd}(a,c) \text{ is an integer}\]

Increasing Instruction Concurrency

- Reducing sequences of dependent instructions \(\rightarrow\) copy propagation

  \[
  \begin{align*}
  &X = A \\
  &Y = Z + X \\
  \end{align*}
  \]

  \[
  Y = Z + A
  \]

- Constant propagation is a special case
Increasing Instruction Concurrency (cont.)

- Tree height reduction: exploiting associativity
  - Use parentheses

\[
\begin{align*}
\text{ADD.D F0, F2, F4} & \quad \text{ADD.D F0, F2, F4} \\
\text{ADD.D F6, F8, F0} & \quad \text{ADD.D F6, F8, F10} \quad \text{Increase ILP} \\
\text{ADD.D F10, F12, F6} & \quad \text{ADD.D F12, F0 F6}
\end{align*}
\]

Increasing Instruction Concurrency (cont.)

- Opportunities for increasing ILP via tree height reduction can increase with loop unrolling
  - For example consider recurrence \( \text{sum} = \text{sum} + x; \)

\[
\begin{align*}
\text{Initial unrolled loop} & & \quad \text{..} \\
\text{sum} &= \text{sum} + x1 \\
\text{sum} &= \text{sum} + x2 \\
\text{sum} &= \text{sum} + x3 \\
\text{sum} &= \text{sum} + x4 \\
\text{sum} &= \text{sum} + x5 \\
\text{..} \\
\text{sum} &= ((\text{sum} + x1) + (x2 + x3)) + (x4 + x5)
\end{align*}
\]
Study Guide: Glossary

- Constant propagation
- Dependence distance
- Instruction scheduling
- Loop unrolling
  - Symbolic optimization
- Loop carried dependency
- Loop level parallelism
- Register pressure
- Tree height reduction

Study Guide

- Schedule simple code sequences on the basic 5 stage, multifunction pipeline
  - Be able to detect dependences and schedule to minimize stalls
- Given a simple loop unroll and schedule the loop body to minimize stalls
- Unroll by a factor of $k$, a loop with unknown loop bounds
- Given a code block, determine if loop carried dependencies exist
  - Be able to state whether one cannot make such a determination
• Can you provide a code sequence where hardware scheduling will always perform better than a compiler scheduled solution?

• Can you provide a short code sequence where a compiler scheduled solution will never perform poorer than a hardware scheduled (Tomasulo) implementation?