Course on Advanced Computer Architectures

VLIW Code Scheduling
Code Scheduling

- Main goal: Statically arranging the order of instructions in object code so that they are executed in an optimum and semantically correct order.
  - Execute time-critical operations efficiently.
  - Try to increase the number of independent instructions fetched.

⇒ Minimize Execution Time
Scheduling Basics

- Decompose the function in basic blocks.
- A basic block is a code sequence that does not contain a branch or a branch target within the sequence.
A dependence graph captures true, anti and output dependencies. Anti and output dependencies are name dependences due to variables/registers reuse.

\[
\begin{align*}
i) & \quad A = B + 1 \\
& \quad \ldots \\
i+k) & \quad X = A + C \\
& \quad \ldots \\
i+n) & \quad A = E + C
\end{align*}
\]
Critical Path

- Is the longest path in the dependence graph
- Determines the minimum execution time of a basic block

\[ LP(i) = \text{Max} (LP(Pred(i))) + \text{Latency}(i) \]

\[ LCP = \text{Max} (LP(i)) \]
Scheduling Basics

- Scheduling selects the cycle of execution of each operation so that it executes in the minimum amount of time.
- On a processor with infinite resources, we could schedule all the ops of the CP and then the remaining ones by exploiting available slots.
- With finite resources, the execution time does not depend only on CP but also on how we schedule remaining instructions.
An optimum scheduler must exhaustively search the space of optimal schedule.

Space and time complexity is very big (NP)!

We must use heuristics.
List-based Scheduling for a Basic Block

- For each cycle, choose instructions from the ready set that can fill actual slots.
- An instruction is in the ready set if all of its predecessors have been scheduled and if the operands are ready.
- Before scheduling begins, operations on top of the graph are inserted in the ready set.
- Starting from the first cycle, for each cycle try to schedule instructions/nodes in the ready set.
- When more nodes are in the ready set, select the op. with highest priority (longest path to the bottom of the graph).

C. Silvano, S. Xydis, G. Agosta - Politecnico di Milano - Nov. 2013
Resource Reservation Table

- Keeps track of busy resources
- If the selected instruction has a busy unit, try with another operation in the ready set.
Example

Priority/Latency

VLIW CODE

<table>
<thead>
<tr>
<th>Ready List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 a,b,c</td>
</tr>
<tr>
<td>2 c,e</td>
</tr>
<tr>
<td>3 e</td>
</tr>
<tr>
<td>4 d,f</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7 g</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RRT</th>
<th>ALU1</th>
<th>L/S</th>
<th>MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>c</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>e</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>f</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>f</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>g</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VLIW</th>
<th>ALU</th>
<th>L/S</th>
<th>MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td>n</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>c</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>e</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>f</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>5</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>
Exploiting ILP: Local and Global Scheduling

- To exploit all the possible parallelism, the compiler must expand basic block or schedule instructions across basic blocks.
- Local Scheduling Techniques operate within a single basic block.
  - Example: Loop Unrolling, Software Pipelining
- Global Scheduling Techniques operate across basic blocks.
  - Example: Trace Scheduling, Superblocks Scheduling
The compiler can increase the amount of available ILP by unrolling loops: the loop body is replicated multiple times (depending on the unrolling factor), adjusting the loop termination code.

The compiler must test if loop iterations are independent.

Loop overhead (number of loop counter increments and branches) is minimized.

Loop unrolling extends the length of the basic block ⇒ the loop exposes more computation that can be effectively scheduled to minimize stalls.

Loop unrolling increases the number of required registers (increases register pressure).

Loop unrolling increases the code size.
Loop Unrolling: Example

for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s

Loop iterations are independent

First, we consider a single iteration

Assembly code
(Not Scheduled)
Considering data and control dependences (branch solved in ID stage) and following ops latency:

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Execution in cycles</th>
<th>Latency in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Loop Unrolling: Example

Scheduled Assembly code
No unrolling

Performance:
10 clock cycles per iteration
Loop Unrolling: Example

4 times loop unrolling:

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0 (R1)</td>
</tr>
<tr>
<td>ADD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>F4, 0 (R1)</td>
</tr>
<tr>
<td>LD</td>
<td>F0, -8 (R1)</td>
</tr>
<tr>
<td>ADD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>F4, -8 (R1)</td>
</tr>
<tr>
<td>LD</td>
<td>F0, -16 (R1)</td>
</tr>
<tr>
<td>ADD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>F4, -16 (R1)</td>
</tr>
<tr>
<td>LD</td>
<td>F0, -24 (R1)</td>
</tr>
<tr>
<td>ADD</td>
<td>F4, F0, F2</td>
</tr>
<tr>
<td>SD</td>
<td>F4, -24 (R1)</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1, R1, #32</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, LOOP</td>
</tr>
</tbody>
</table>

There are:
- Data dependences
- Name dependences

The compiler applies register renaming to avoid name dependences
Loop Unrolling: Example

Loop:

LD    F0, 0(R1)
ADD   F4, F0, F2
SD    F4, 0(R1)
LD    F6, -8(R1)
ADD   F8, F6, F2
SD    F8, -8(R1)
LD    F10, -16(R1)
ADD   F12, F10, F2
SD    F12, -16(R1)
LD    F14, -24(R1)
ADD   F16, F14, F2
SD    F16, -24(R1)
SUBI  R1, R1, #32
BNE   R1, R2, LOOP

Only true data dependences remain in each body and between the last 2 instructions

Next step: we can apply scheduling
Loop Unrolling: Example

Execution time:
14 cycles per 4 iterations
⇒ 3.5 cycles per iteration
2 cycles of loop overhead per 4 iterations

Next step: scheduling for a 5-issue VLIW

C. Silvano, S. Xydis, G. Agosta - Politecnico di Milano

Nov. 2013
## Loop Unrolling: Example on VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP operation 2</th>
<th>Integer op/Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F23</td>
<td>NOP</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td>NOP</td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td>NOP</td>
</tr>
<tr>
<td>NOP</td>
<td>NOP</td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td>NOP</td>
</tr>
<tr>
<td>SD F4, 0(R1)</td>
<td>SD F8,-8(R1)</td>
<td>ADDD F28,F26,F2</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>SD F12,-16(R1)</td>
<td>SD F16,-24(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>SD F20,-32(R1)</td>
<td>SD F24,-40(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>SUBI R1,R1,#5</td>
</tr>
<tr>
<td>SD F28,8(R1)</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>BNEZ R1,LOOP</td>
</tr>
</tbody>
</table>

**Execution time:**

9 cycles per 7 iterations ⇒ 1.28 cycles per iteration

2 cycles of loop overhead per 7 iterations

**Average:** 2.5 ops per clock,

**Efficiency i.e. Percentage of available slots that contained an operation is 60%**

---

C. Silvano, S. Xydis, G. Agosta - Politecnico di Milano

Nov. 2013
Loop-carried dependences

- Loop-level analysis involves determining what data dependences exist among the operands in a loop across the iterations.
- Loop-carried dependence: Whether data accesses in later iterations are dependent on data values produced in earlier iterations.
Loop-carried dependences

for (i=1; i<=100; i=i+1)
{
    A[i]=B[i]+C[i];
}

In this example an iteration of the loop does not have dependences with previous iteration (works on different array elements)
⇒ NO LOOP CARRIED DEPENDENCIES

for (i=1; i<=97; i=i+4)
{
    A[i]=B[i]+C[i];
    A[i+1]=B[i+1]+C[i+1];
    A[i+2]=B[i+1]+C[i+2];
    A[i+3]=B[i+3]+C[i+3];
}

Larger Basic Block with extended parallelism (the unrolling factor could also be greater than 4)
Loop-carried dependences

for(i=6; i<=100; i=i+1)
{
    Y[i]=Y[i-5]+Y[i]
}

- Each iteration i depends on the value of the iteration i-5. Iterations i, i+1, i+2, i+3, i+4 are independent! (i+5 is dependent on i) so we can unroll the loop up to 5:

for(i=6; i<=96; i=i+5)
{
    Y[i]=Y[i-5]+Y[i]
    Y[i+1]=Y[i-4]+Y[i+1]
    Y[i+2]=Y[i-3]+Y[i+2]
    Y[i+3]=Y[i-2]+Y[i+3]
}

Larger Basic Block with extended parallelism (the unrolling factor could not be greater than 5)
for (i=1; i<=100; i=i+1) 
{
    A[i+1]=A[i]+C[i];  /* S1 */
}

- Two different dependences:
  - S1 uses a value computed by S1 in an earlier iteration for array A;
    the same for S2 and array B
    ⇒ LOOP CARRIED DEPENDENCES FOR A[
  - S2 uses the value a[i+1] computed by S1 in the same iteration
    ⇒ NO LOOP CARRIED DEPENDENCES FOR A[

C. Silvano, S. Xydis, G. Agosta - Politecnico di Milano - 23 - Nov. 2013
for (i = 0; i < 102; i++) b[i] = b[i-2] + c; // Loop A
for (j = 0; j < 100; j++) a[j] = a[j] * 2; // Loop B

We “peel” loop A of the last two iterations (making it a 100 iteration loop. Then we fuse the two 100 iteration loop to make a larger basic block. Then other techniques can be applied.

for (i = 0; i < 100; i++) {
    b[i] = b[i-2] + c; // fused loops
    a[i] = a[i] * 2; }
    b[100] = b[98] + c; // peeled from loop A
    b[101] = b[99] + c;
Software Pipelining

- Suppose that the following loop presents independent instructions in different iterations (evidenced in red).

- We can reorganize the loop in a new loop so that each new iteration ("cycle") executes instructions ("stages") chosen from different iteration of the original loop.
Software Pipelining

- Technique for reorganizing loops such that each iteration of the software-pipelined code is made from instructions chosen from different iterations of the original loop.
- Software pipelining interleaves instructions from different iterations without unrolling the loop.
- Software pipelining can be thought of as Symbolic Loop Unrolling.
Software Pipelining: Example

for(i=0; i<100; i++)
{
    A[i]=B[i];   // stage X
    A[i]=A[i]+1; // stage Y
    C[i]=A[i];   // stage Z
}

- No loop carried dep.
- Intra-Body depen. present

Iteration 0                 Iteration 1                          Iteration 2


Startup-code

C. Silvano, S. Xydis, G. Agosta - Politecnico di Milano

Nov. 2013
Software Pipelining

A[0]=B[0];
A[0]=A[0]+1;
for (i=0; i<98; i++)
{
    C[i]=A[i];
    A[i+2]=B[i+2];
}
C[i]=A[i];
C[i+1]=A[i+1];

Independent instructions
Advantages of Software Pipelining

- Consumes less space (no need to duplicate body-code) than loop unrolling.
- Fill and drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling.

- Can be associated with loop unrolling to provide better performance.
Global Code Scheduling

- Software pipelining works well when the loop body is a single basic block.
- When the loop body contains internal control flow, effective scheduling requires moving instructions across branches: Global Code Scheduling
- Global Code Scheduling aims at compacting a code fragment with internal control structures into the shortest possible sequence preserving data and control dependences
Trace Scheduling

- Tries to find parallelism across conditional branches (global code scheduling).
- Composed of two steps:
  - **Trace Selection**
    - Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - **Trace Compaction**
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping (compensation) code in case prediction is wrong
Trace Scheduling

- This is a form of compiler-generated speculation
  - Compiler must generate “fixup” code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding
Superblocks Scheduling

- Extension/Optimization of Trace Scheduling
- A Superblock is a group of basic blocks with a single entrance and multiple control exits.
- Superblocks are constructed by profiling the application and by duplicating tails (blocks after an entrance in the trace).
- Advantages:
  - Optimization simpler because there are no side entrances.
  - We need to create compensation code only for exits and not for entrance.
Superblock Formation

C. Silvano, S. Xydis, G. Agosta - Politecnico di Milano - 34 - Nov. 2013
Techniques such as loop unrolling, software pipelining, and trace scheduling can be used to increase ILP when the branch behavior is fairly predictable at compile time.

Otherwise control dependences may limit the parallelism that can be exploited.

To overcome such limitation we can:

- Extend the instruction set to include Conditional or Predicated Instructions
- Use Compiler Speculation with hardware support to enable the compiler to speculatively move code over branches, while preserving exception behavior.
Conditional (or Predicated) Execution

- **Predicated Instruction:**
  
  \[ (p) \quad \text{op} \quad Rd, R1, R2 \]

  - \( p \) is a boolean predicate register.
  - \( \text{op} \quad Rd, R1, R2 \) is a normal triadic operation.
  - \( \text{op} \) is committed only if \( p \) is **True**

- The execution of the operation is controlled by the predicate: When the predicate is false \( \Rightarrow \) the operation becomes a **nop**.
If-conversion

- If-conversion is the process that converts a conditional branch into a sequence of predicated instructions:

```assembly
bnez r4
add r1=r2,r3
lw r6,0(r5)
```

\[ p = (\text{cmp } r4, 0) \]

\[ (p)\text{add } r1=r2,r3 \]

\[ (p)\text{lw } r6,0(r5) \]
Predicated Execution

- Control dependencies are transformed in data-dependencies \(\Rightarrow\) Branches are eliminated.

- Advantages:
  - Data-dependency based movements can then be applied.
  - Branch misprediction is eliminated. We do not need to flush the pipeline!
  - Enlarge basic block to improve scheduling.
Effective if:

- Misprediction rate and penalty are considerable. Otherwise a branch to the most likely code would result in better performance.

- Branches are unbalanced. The longest path is executed more frequently. Otherwise we must avoid to lengthen the execution of the little and more frequent path.