CSC2/458 Parallel and Distributed Systems
Automated Parallelization in Software

Sreepathi Pai
January 30, 2018

URCS
Out-of-order Superscalars and their Limitations

Static Instruction Scheduling
Out-of-order Superscalars and their Limitations

Static Instruction Scheduling
How will a processor parallelize this?

```c
for(i = 0; i < A; i++) {
    sum1 = sum1 + i;
}

for(j = 0; j < A; j++) {
    sum2 = sum2 + j;
}
```
i = 0
i < A (true)
sum1 = sum1 + 0
i++
i < A (true)
sum1 = sum1 + 1
i++
...
i < A (false)

j = 0
j < A (true)
sum2 = sum2 + 0
j++
j < A (true)
sum2 = sum2 + 1
j++
j < A (true)
...

An Intel Processor Pipeline

Source: Intel
Instruction Pipeline

- Instructions flow into “issue window”
  - from dynamic instruction stream
- Dependences are calculated and resources allocated
- Independent instructions are dispatched to backend *out-of-order*
- Instructions are *retired in-order* using a “reorder buffer”
Out-of-order Superscalars and their Limitations

Static Instruction Scheduling
VLIW Processors

- Very Long Instruction Word Processors
- Can execute multiple instructions at the same time
  - So superscalar
- But leaves independence checking to the compiler
  - Compiler packs instructions into "long words"

Example:

<table>
<thead>
<tr>
<th></th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLIW1:</td>
<td>ins1</td>
<td>ins2</td>
</tr>
<tr>
<td>VLIW2:</td>
<td>ins3</td>
<td>[empty]</td>
</tr>
</tbody>
</table>
Consider *static code* below:

```c
for(i = 0; i < A; i++) {
    sum1 = sum1 + i;
}
for(j = 0; j < A; j++) {
    sum2 = sum2 + j;
}
```

For a 2-wide VLIW, one packing could be:

<table>
<thead>
<tr>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>i = 0</td>
<td>j = 0</td>
</tr>
<tr>
<td>i &lt; A</td>
<td>j &lt; A</td>
</tr>
<tr>
<td>sum1 = sum1 + i</td>
<td>sum2 = sum2 + j</td>
</tr>
<tr>
<td>i++</td>
<td>j++</td>
</tr>
</tbody>
</table>
• When processors commit in-order, they preserve appearance of executing in program order
  • Not always true when multiple processors are involved
• But when compilers emit code, they change order from what is in program
• Which orders in the original program must be preserved?
• Which orders do not need to be preserved?
Our Ordering Principles

- Preserve Data Dependences
- Preserve Control Dependences

What about:

```c
printf("hello");
printf("world");
```
Basic Block Scheduling

- Basic block is a single-entry, single-exit code block
- Instructions in basic block have the same control dependence
  - All can execute together if they have no dependence
- Is there an advantage in reordering instructions within a basic block?
Consider:

\[
\begin{align*}
A &= 1 \quad /\!/\text{ takes } 1 \text{ cycle} \\
B &= A + 1 \quad /\!/\text{ takes } 1 \text{ cycle} \\
C &= A \times 3 \quad /\!/\text{ takes } 2 \text{ cycles and } 2 \text{ ALUs} \\
D &= A + 5 \quad /\!/\text{ takes } 1 \text{ cycle}
\end{align*}
\]

Assume you have 2 ALUs. How should you schedule these instructions to lower total time?
Basic blocks are usually small
  - Not many opportunities to schedule instructions
How can we increase size of basic blocks?
  - Remember out-of-order processors do speculation ...