MultiScalar
The quest for parallelism

• Single threads have a little bit of ILP
• We want MORE!
• Multithreaded programming is hard
  • Locks are tricky
  • Often, statically available parallelism is just scarce
  • Dynamic data dependences are unpredictable
• OOO super scalars are hard to build
  • The instruction scheduler is not scalable
  • Constructing a big instruction window is hard, because branch predictors are good enough.
MultiScalar idea

• Use SW + HW to divide the program into pieces
  • What should the HW look like?
  • How should the SW express the pieces?
• Speculatively run consecutive pieces in parallel
  • Which pieces?
• Clean up the mess, when we misspeculate
  • This is the tough part.
Main idea: Execute multiple tasks in parallel, Forward values between tasks to maintain sequential semantics

Value forwarding will happen dynamically, at execution time.
MultiScalar tasks

- Any dynamic sequence of instructions can be a task
- Arbitrarily large!
- Arbitrarily small!
- Any of number of exits!
- Can include function calls!
Register value forwarding

- What are the outputs? -- mask from compiler
- What are the inputs? -- mask computed from running tasks
- Which version to use (a task might produce more than one)? -- extra bits on the instruction.
Memory Value Forwarding

- We can’t compute a mask for memory.
- Instead, use the address resolution buffer (ARB)
  - Tasks read *values* speculatively from the ARB, assuming that no younger tasks will write to that location
- If a write occurs, squash the task that read the value and squash *all future tasks.*
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);
    /* do a linear search for the symbol in the list */
    for (list = listhd; list; list = LNEXT(list)) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
    }
    /* if symbol not found in the list, add to the tail */
    if (!list) {
        addlist(symbol);
    }
}

**Figure 3:** An Example Code Segment.

**Figure 4:** An Example of a Multiscalar Program.
Figure 1: A Possible Microarchitecture of a Multiscalar Processor.
Speedup - SPEC95 INT

SPEC95 INT Benchmark

- vortex
- perl
- ijpeg
- li
- compress
- gcc
- m88ksim
- go

Speedup - 16 Issue

From Scott Breach’s MS Slides
Compiler Window Factors

- If Tasks Too Small
  - ✗ Register Dependences
    - ◆ Wait Overhead
    - ◆ Aggravate Critical Paths

- If Tasks Too Big
  - ✗ Memory Dependences
    - ◆ Squash Overhead
    - ◆ Buffer Overflow
Hardware Window Factors

- Communication Delay
- Load Imbalance
- Pipeline Fill/Drain
- Misspeculation Penalty
Bottom Line on Window...

- If Same as Superscalar - SPEC95 INT
  - Difficult to Sustain Same Raw IPC
  - With Clock Advantage Better Speedup

- If Better than Superscalar - SPEC95 FP
  - Possible to Sustain Better Raw IPC
  - With Clock Advantage Even Better Speedup

From Scott Breach’s MS Slides
Comparison to OOO SuperScalars

<table>
<thead>
<tr>
<th></th>
<th>SuperScalar</th>
<th>MultiScalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating the instruction window</td>
<td>Predict many, many branches correctly</td>
<td>Predict fewer, better-behaved(?) branches correctly</td>
</tr>
<tr>
<td>Scheduling instructions</td>
<td>Analyze 100s of instruction dependencies</td>
<td>Track on instruction per task; manage inter-task dependencies</td>
</tr>
<tr>
<td>Memory dependence tracking</td>
<td>Store buffer</td>
<td>Address resolution buffer</td>
</tr>
<tr>
<td>Register dependence tracking</td>
<td>Renaming and OOO scheduling</td>
<td>Generate/Accum masks; Compiler assistance</td>
</tr>
</tbody>
</table>
In Context

• MultiScalar is very influential and extremely ambitious.

• Spawned much work in speculative threading on more reasonable architectures.

• If we evaluate it in terms of modern technology, how does it hold up?