Outline

• The Problem
• Processing basics and wish lists
• Options for high performance
• The multiscalar model
• Performance results
• Concluding remarks
The Problem

Software: create static image of dynamic computation

Hardware: recreate dynamic computation from static representation and carry out computation

Processing Hardware: Big Picture

- Start with a static representation of a program
- **Sequence** through the program to generate the dynamic stream of operations
  - Use single PC to walk through static representation?
- **Execute** operations in dynamic stream
  - **Schedule** operations for execution
  - **Execute** operations
  - **Communicate** values
Why Multiscalar?

Basic Issues

- Sequencing
- Scheduling
- Operation execution
- Operand communication

The Big Question

- How do we sequence, schedule, execute, and communicate in a more powerful manner?

- Powerful =
  - a large variety of applications
  - time efficient
  - space efficient
  - power efficient?
  - etc.
Target: 10 IPC

- Sequence through static program and establish a large instruction window (100s of instructions)
- Maintain a large window
- Sequence through program and initiate at least 10 operations into this window per cycle
- Schedule for execution at least 10 operations per cycle
- Provide lots of storage for inter-operation communications

Options

- **Sequencing**: single (wide) vs. multiple
- **Scheduling**: central vs. distributed
- **Operation execution**: not much choice; provide requisite bandwidth
- **Operand communication**: central vs. distributed storage
Sequencing/Scheduling Options

<table>
<thead>
<tr>
<th>Static</th>
<th>Dynamic</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Single Sequencer" /></td>
<td><img src="#" alt="Multiple Sequencers" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Engineering Considerations**

- Now throw in engineering into the big picture
- What is desirable from the engineering viewpoint?
  - Hardware wish list
  - Software wish list
**Why Multiscalar?**

- Simplify engineering (design, verification, testing)
  - Use of simple, regular hardware structures
    - clock speeds comparable to single-issue processors
  - “Locality” of interconnect
  - Easy growth path from one generation to next
    - reuse existing processing cores
  - No centralized bottlenecks

**Hardware Wish List**

- Take current generation processor
- Replicate some parts, share others
- Have next generation processor
- Different units can sequence, schedule, etc. in parallel

**The “Hardware-Influenced” Solution**

BUT, the software problem .......
The Software Problem

- Can’t always break up program into “independent” chunks (i.e., multiple sequencers) statically
  - control dependences
  - data dependences (especially ambiguous ones)
  - also load balance

- Can’t map program onto rigid hardware model

Software Wish List

- Simplify engineering
  - Don’t force “rigid model”
  - Don’t ask for guarantees
  - Don’t expect software to track hardware
  - Others ......
Why Multiscalar?

- Take “mostly sequential” static program
- Use speculation to overcome dependence limitations
  - When in doubt, speculate
- Break up program into “potentially independent” chunks dynamically

Sequencing

- Unraveling the operations to be executed dynamically
- Use 2-level sequencing
  - sequence high level in task-sized steps
  - sequence within task
  - vectors?
- Use control flow speculation to increase sequencing power
  - overcome “stalls”
**Scheduling**

- Use *multiple schedulers* to improve scheduling power
- Use *data dependence speculation* to overcome scheduling limitations
  - ambiguous dependences
- Use *value speculation* to overcome scheduling limitations
  - true dependences
- Use *memoization* to avoid re-doing work
  - true dependences

---

**Operand Communication**

- Values bound to registers and memory
- Values created speculatively
- Storage
  - where should values be buffered?
- Synchronization
  - operation uses value of latest producer
- Communication
  - forwarding created value to (future) consumers
- Create and exploit localities to reduce/simplify interconnect!
Multiscalar Paradigm

- Break sequencing process into two steps
  - Sequence through static representation in *task-sized* steps
  - Sequence through each task in conventional manner
- Split large instruction window into ordered tasks
- Assign a task to a simple execution engine; exploit ILP by overlapping execution of multiple tasks
- Use separate PCs to sequence through separate tasks
- Maintain the appearance of a single-PC sequencing through the static representation
- Use control and data dependence speculation

What is a Task?

- A portion of the static representation resulting in a contiguous portion of the dynamic instruction stream
  - part of a basic block
  - basic block
  - multiple basic blocks
  - loop iteration
  - entire loop
  - procedure call, etc.
Multiscalar Big Picture: Basics

Why Multiscalar?

Multiscalar Big Picture: Hardware

Multiscalar Big Picture: Basics

Why Multiscalar?
Register Values

• Each core works out of its “local” register file
• Multiple register files act like separate “renamed” files
• Each register file contains register state at a particular time in the (speculative) execution of a program

Memory Values

• Storage
• Synchronization
• Communication
• Versions
**Why Multiscalar?**

- **Traditional Memory Interface: Load Store Queue**
  
<table>
<thead>
<tr>
<th>Instr</th>
<th>Addr</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>store 200 66</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>load 100 --</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>store ? ?</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>load 200 --</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>store 200 ?</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  Arranged by PROGRAM ORDER
  Searched using ADDRESS

  - Memory Dependence Speculation
  - Multiple Versions

**Memory System I: Address Resolution Buffer**

<table>
<thead>
<tr>
<th>PU 0</th>
<th>PU 1</th>
<th>PU 2</th>
<th>PU 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>store 200 55</code></td>
<td><code>store ? ?</code></td>
<td><code>load 200 --</code></td>
<td><code>load 100 --</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
<th>Data</th>
<th>L</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>55</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Data | L | S |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>66</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>121</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

One ARB line

- Arranged by ADDRESS; Searched using PROGRAM ORDER
- Each line buffers multiple versions
- Committed versions are written back immediately
Memory System II: Speculative Versioning Cache

Program Order

PU 0 | PU 1 | PU 2 | PU 3
---|---|---|---
store 200 | store ? | load 200 | load 100
store 200 | store ? | ? | ?

Addr | Data | State | Next
---|---|---|---
200 | 55 | S | 3

Cache 0

Cache 2

200 121 L -

One SVC line

Cache 1

Cache 3

200 66 S -

- Maintains a linked list of versions; PU #s used as pointers
- Each line buffers only one version
- Committed versions written back only when necessary

Scheduling Memory Operations

- Data dependence speculation is the default
  - predict no dependences
- Improving accuracy of data dependence prediction
  - akin to branch prediction for control dependences
- Track history of dependence mis-speculations
  - small number of static dependence pairs
  - exhibit temporal locality
- Use history for future data dependence speculation/synchronization decisions
Example: Problem

- Process stream of tokens
- Create entry in list for new token
- Use information in list to process token

Example: C Code

```c
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, add it to the tail */
    if (! list) {
        addlist(symbol);
    }
}
```
Why Multiscalar?

- Each task is a complete list search
- Searches are usually independent and parallel
  - Multiscalar can assume they are always independent
- Branches that separate tasks are predictable
- Branches within a task unlikely to be 100% predictable
  - Superscalar/VLIW unlikely to be able to overlap processing of different tokens

Example

Example: Executable

Going from one generation to another could leave binary untouched!
Why Multiscalar?

- Multiscalar-specific information (task successors, create masks, forward bits, stop bits) is available in a binary
- Recover information at run time
  - “Low” performance but run ordinary binaries
- Binary to binary translation
  - Better performance by including some optimizations
- Compiler
  - Best performance, but needs recompilation

Regardless, binary from one multiscalar generation to another can remain the same

Performance: SPECint95
### Why Multiscalar?

#### Performance: SPECfp95

<table>
<thead>
<tr>
<th>Speedup</th>
<th>swim</th>
<th>hydro2d</th>
<th>mgrid</th>
<th>applu</th>
<th>turb3d</th>
<th>apsi</th>
<th>fpppp</th>
</tr>
</thead>
<tbody>
<tr>
<td>4/2/1</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>0.97</td>
<td>0.99</td>
</tr>
<tr>
<td>8/2/1</td>
<td>1.00</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
<td>0.97</td>
<td>0.99</td>
</tr>
</tbody>
</table>

### Attributes Multiprocessor Multiscalar

<table>
<thead>
<tr>
<th>Speculative task initiation</th>
<th>No/Difficult</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple flows of control</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Task determination</td>
<td>Static</td>
<td>Static (possibly dynamic)</td>
</tr>
<tr>
<td>Software guarantee of inter-task control independence</td>
<td>Required?</td>
<td>Not required</td>
</tr>
<tr>
<td>Software knowledge of inter-task data dependencies</td>
<td>Required?</td>
<td>Not required</td>
</tr>
<tr>
<td>Inter-task sync.</td>
<td>Explicit</td>
<td>Implicit/Explicit</td>
</tr>
<tr>
<td>Inter-task communication</td>
<td>Through memory</td>
<td>Through registers and memory</td>
</tr>
<tr>
<td>Register space</td>
<td>Distinct for PEs</td>
<td>Common for PEs</td>
</tr>
<tr>
<td>Memory space</td>
<td>Common</td>
<td>Common for PEs</td>
</tr>
</tbody>
</table>
Why Not Multiscalar?

- Programs can be (statically) parallelized easily
- Hardware replication not desirable
  - interconnect not an issue (copper?)
  - centralized designs easier to design/validate
  - centralized designs easier to test

Concluding Remarks

- Future microarchitectures will be decentralized (operation execution, operand communication, scheduling, sequencing)
- Multiscalar model enables distributed execution of a sequential (or parallel) program
- Beginning of a new generation of microarchitectures
  - much works remains to be done
Related Projects

- Stanford HYDRA
- CMU STAMPede
- Minnesota Superthreaded
- Waikato WARP engine
- Washington SMT
- MIT M-machine
- MIT RAW
- Michigan HPS
- CMU Superflow
- Illinois IMPACT

Acknowledgments

Thanks to:
Faculty: Jim Smith, Chuck Kime

for helping to conceive, develop, realize, and refine the multiscalar vision

Thanks to: DARPA, NSF, Intel for funding the efforts