Module-Level Speculative Execution Techniques on Chip Multiprocessors

Fredrik Warg
Motivation

• Expose more parallelism in existing programs for improved performance
  – Instruction-level parallelism – diminishing returns
  – Multithreaded programs – depend on programmer
  – Speculative threads – automatic parallelization

• Make use of the processing resources in emerging chip multiprocessor architectures
Outline

• Background
  – Thread-level speculative execution and CMPs
  – Module-level speculative execution

• Possibilities and problems

• Improving module-level speculation
  – Run-length prediction
  – Misspeculation prediction

• Wrap-up
Thread-Level Speculation

- Split up existing applications into threads that run in parallel
- Data dependences are detected and handled
- Speculative data is buffered until it is known to be correct

❗️Handled in run-time by a *speculation system*
**Data Dependences**

- **Name dependences**:
  - Thread 1 (head)
  - Thread 2 (speculative)
  - Output: W, W
  - Input: R, anti

- **Execution time**:
  - Thread 1 (head)
  - Thread 2 (speculative)

- **Flow dependences**:
  - Thread 1 (head)
  - Thread 2 (speculative)
  - Forwarding: W → R
  - Roll-back: W → R

**Solutions**

- Renaming
- Forwarding
- Roll-back

☞ **Roll-back means a loss of work – we want to avoid this**
Chip Multiprocessors

Low latency communication between processor cores
Thread-Level Speculation on CMPs

On-chip memory enhanced to:
- Detect dependence violations
- Buffer speculative values
- Commit or squash spec. values

Speculation system handles creation/scheduling of threads.
Module-Level Speculation

Thread view

Code view
Why Module-Level Speculation?

- Well-defined thread boundaries
- No control speculation
- No misspeculations on local variables

Suitable for run-time parallelization
Inherent Parallelism

Machine with no overheads

MLP suitable for 4 to 8-way CMPs
Problems

• Speculation overheads
  – Takes time to start new threads, handle violations (roll-back) etc.
  – Short threads mean relatively more overhead

• Misspeculations
  – Extra (overhead) work due to reexecution

➤ Overheads limit achievable speedup
➤ Reexecution waste processing resources
Performance With Overheads

![Graph showing speedup with and without overheads for different programs]

- **Gcc**
- **Compress**
- **Go**
- **Neuralnet**

- **No thread-management overhead**
- **200-cycle thread-start, roll-back, commit overhead**

☞ Overheads limit speedup and even cause slowdowns
Solutions

**Observation:** Creating new threads for some modules makes performance worse, not better.

So why not *selectively* start new threads only when we expect it to be beneficial?

**Two methods:**

- Run-length prediction – aims at improving speedup
- Misspeculation prediction – aims at lowering total overhead
Run-Length Prediction

**Problem:** Too many small or non-overlapping threads

**Solution:** Do not start thread if expected parallel overlap is low

- If module being called has short *run-length*, it is likely not beneficial to create a thread
  - Dynamically measure run-length
  - Set lower threshold for run-length
  - Use history-based prediction to determine if called module will exceed threshold
Measuring Run-Length

- Run-length: dynamically measured execution time of a module excluding:
  - Speculation overheads.
  - Parallel child modules.
- Run-length threshold:
  - Only start speculative thread if called module has a run-length over threshold.
Run-Length Prediction Example

Prediction Table

Above threshold

New thread started again
Results: Run-Length Threshold

8-way machine with 200-cycle thread-management overheads.

- Go (perfect)
- Neuralnet (perfect)
- Go (predicted)
- Neuralnet (predicted)

❖ Best threshold varies, typically ~500
❖ Prediction accuracy typically >90%
Misspeculation Prediction

**Problem:** Misspeculations causes reexecution overhead

**Solution:** Do not start threads expected to cause misspeculation

- If creating a thread for a module call caused a misspeculation, it is likely to happen again the next time the same module is called
  - Record misspeculations in a prediction table
  - Prevent thread creation for previously misspeculating threads
Misspeculation Prediction Example

First call

Second call

Violation!
Load a
Store a

Misspeculation

Prediction Table
Results: Misspeculation Prediction

8-way machine

- Gcc: 462% (100-cycle overheads, speculate on all modules), 62% (100-cycle overheads, misspeculation prediction)
- Compress: 509% (100-cycle overheads, speculate on all modules), 27% (100-cycle overheads, misspeculation prediction)
- Go: 71% (100-cycle overheads, misspeculation prediction)
- Neuralnet: 333% (100-cycle overheads, speculate on all modules), 79% (100-cycle overheads, misspeculation prediction)

Overhead reduced from avg. 336% to avg. 54%
Thesis Contributions

• Characterizing module-level speculative execution
  – Limits
  – Problems

• Methods for selecting when to start new threads
  – Run-length prediction
  – Misspeculation prediction
Conclusions

• Module-level parallelism suitable for small CMPs, but does not scale to >8 cores
• Selective use of module-level speculation necessary to avoid excessive overhead

Future Work

• Tradeoff between ILP and TLP
• Memory-system effects
• …and hopefully other interesting stuff!
The End
Extra

Results: Return Value Prediction

8-way machine

<table>
<thead>
<tr>
<th></th>
<th>Gcc</th>
<th>Compress</th>
<th>Go</th>
<th>Neuralnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>2.0</td>
<td>1.5</td>
<td>5.0</td>
<td>2.5</td>
</tr>
</tbody>
</table>

- **Perfect return value prediction**
- **Stride return value prediction**
- **No return value prediction**

Module-Level Speculative Execution Techniques on Chip Multiprocessors
Extra

Results: Number of processors

8-way machine

<table>
<thead>
<tr>
<th></th>
<th>Gcc</th>
<th>Compress</th>
<th>Go</th>
<th>Neuralnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>Inf</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Gcc, Compress, Go, Neuralnet
Extra
Results: Roll-back policy

8-way machine

Speedup

Violation

Thread roll-back

Perfect roll-back

Violation

Perfect roll-back

With 100-cycle overheads

<table>
<thead>
<tr>
<th></th>
<th>Gcc</th>
<th>Compress</th>
<th>Go</th>
<th>Neuralnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect roll-back</td>
<td>1</td>
<td>0.5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Thread roll-back</td>
<td>1</td>
<td>0.5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>
Extra
Results: Parallel Overlap

8-way machine

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gcc</td>
<td>462%</td>
</tr>
<tr>
<td>Profiling</td>
<td>308%</td>
</tr>
<tr>
<td>Predictor</td>
<td>190%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compress</td>
<td>508%</td>
</tr>
<tr>
<td>Profiling</td>
<td>152%</td>
</tr>
<tr>
<td>Predictor</td>
<td>27%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go</td>
<td>71%</td>
</tr>
<tr>
<td>Profiling</td>
<td>60%</td>
</tr>
<tr>
<td>Predictor</td>
<td>50%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neuralnet</td>
<td>333%</td>
</tr>
<tr>
<td>Profiling</td>
<td>292%</td>
</tr>
<tr>
<td>Predictor</td>
<td>144%</td>
</tr>
</tbody>
</table>

With 100-cycle overheads
Extra Results: Misspec. Prediction with Threshold

8-way machine

<table>
<thead>
<tr>
<th>Application</th>
<th>100-cycle overheads, speculate on all modules</th>
<th>100-cycle overheads, misspeculation prediction</th>
<th>Misspeculation prediction with 0.6 enable threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gcc</td>
<td>462%</td>
<td>62%</td>
<td>146%</td>
</tr>
<tr>
<td>Compress</td>
<td>509%</td>
<td>27%</td>
<td>40%</td>
</tr>
<tr>
<td>Go</td>
<td>71%</td>
<td>27%</td>
<td>70%</td>
</tr>
<tr>
<td>Neuralnet</td>
<td>333%</td>
<td>79%</td>
<td>79%</td>
</tr>
</tbody>
</table>

☞ Overhead reduced from avg. 336% to avg. 54% or 89%