A Scalable Approach to Thread-Level Speculation

J. Gregory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry

Computer Science Department
Carnegie Mellon University
(Appeared in ISCA 2000.)

Multithreaded Machines Are Everywhere

How can we use them? Parallelism!

Threads

SUN MAJC, IBM Power4
ALPHA 21464 Dual Pentium
SGI Origin

Shared Memory

Processor

while (...){
    x = hash[index1];
    ...
    hash[index2] = y;
    ...
}

Automatic Parallelization

Proving independence of threads is hard:
– complex control flow
– complex data structures
– pointers, pointers, pointers
– run-time inputs

How can we make the compiler’s job feasible?

Thread-Level Speculation (TLS)
A Scalable Approach to Thread-Level Speculation

Example of Thread-Level Speculation

Time

Processor

Epoch 1

= hash[3]

... 

hash[10] = 

Epoch 2

= hash[19] 

... 

hash[21] = 

Epoch 3

= hash[33] 

... 

hash[30] = 

Epoch 4

= hash[10] 

... 

hash[25] = 

Violation!

Epoch 1

= hash[3]

... 

hash[10] = 

commit?

Epoch 2

= hash[19] 

... 

hash[21] = 

commit?

Epoch 3

= hash[33] 

... 

hash[30] = 

commit?

Epoch 4

= hash[10] 

... 

hash[25] = 

commit?

Violation!

Epoch 4

= hash[10] 

... 

hash[25] = 

commit?

epoch 4

Retry
Goals of Our Approach

1) Handle arbitrary memory accesses  
   – i.e. not just array references
2) Preserve performance of non-speculative workloads  
   – keep hardware support minimal and simple
3) Apply to any scale of multithreaded architecture  
   – CMPs, SMT processors, more traditional MPs

effective, simple, and scalable TLS

Overview of Our Approach

System requirements:

1) Detect data dependence violations  
   • extend invalidation-based cache coherence
2) Buffer speculative modifications  
   • use the caches as speculative buffers

coherence already works at a variety of scales

hence our scheme is also scalable

Related Schemes

• Wisconsin (Multiscalar, Trace Processor)
• Stanford (Hydra)
• U.P. Catalunya (Speculative Multithreading)
• Intel/U. Portland (Dynamic Multithreading)
• Illinois at U.C. (I-ACOMA)

our approach seamlessly scales both up and down

Outline

Details of our Approach
   – life cycle of an epoch
   – speculative coherence
   – what happens at commit time
   – forwarding data between epochs

• Performance
• Conclusions
**Life Cycle of an Epoch**

- **Spawning**: Init
- **Becoming Speculative**: Speculative Work
- **Commit?**: Wait to be Homefree?
- **Commit**: Slow Commit, Fast Commit
- **Complete, Pass Homefree**: Complete

**Epoch Numbers**

- **Thread Identifier (TID)**
- **Sequence Number**

**Represent a partial ordering**
- Signed-compare sequence numbers if TIDs match
- Allows for wrap-around
- Otherwise, the epochs are unordered
  - From independent programs
  - From independent chains of speculation within one program

**Speculative Thread Model**

- **Round-robin schedule of epochs to processors**
  - Not a requirement of our scheme, just for convenience
- **Each epoch spawns the next**
  - Through a lightweight fork instruction (10 cycles)
- **Violations detected through polling**
  - Each epoch runs to completion before detecting failed speculation and restarting
- **Violation chaining**
  - If an epoch suffers a violation, we squash all logically-later epochs
  - Many possibilities to be evaluated in future work

**Preserving Correctness**

- Speculation must fail whenever speculative state is lost
  - Eg., replacement of a speculative line, ORB overflow
- Any exceptions are suppressed until epoch is homefree
  - Eg., divide by zero, segfault
- Polling violation detection must avoid infinite looping
  - Requires a poll inside each loop
- No system calls while speculative (for now)

- Ensures original sequential semantics are preserved
Life Cycle of an Epoch

- Time
- Spawning
- Becomes Speculative
- Speculative Coherence
- Commit?
- Complete, Pass Homefree

MESI Coherence Example

Thread A:
- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Thread B:
- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Shared Memory (X=2)

Thread A:
- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Thread B:
- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Load X
Read
Shared Memory (X=2)

Thread A:
- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Thread B:
- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Load X
Read
Fill
Shared Memory (X=2)
**MESI Coherence Example**

Thread A: Store X = 3  
Thread B: Load X

Processor
Cache
State  Tag  Data
Invalid  -  -

Read-Exclusive

Thread A: Store X = 3
Thread B: Load X

Processor
Cache
State  Tag  Data
Invalid  -  -

Shared Memory (X = 2)

*read-exclusive invalidates all other copies*

**Speculative Coherence Example**

Epoch 4:
Load X

Epoch 5:
Store X = 3  
Load X

Epoch 6:

Highlights of our scheme:
- detection of a data dependence violation
- speculatively modified and shared cache lines

*the state ‘dirty’ implies exclusiveness*
A Scalable Approach to Thread-Level Speculation
Steffan & Mowry
Carnegie Mellon

Speculative Coherence Example

Epoch 5: Store X=3
Epoch 6: Load X

- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Speculative msgs piggyback epoch number

Sp Read-Ex (epoch 5)
Shared Memory (X=2)

Speculative Coherence Example

Epoch 5: Load X
Epoch 6: Load X

- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Spec. Loaded

track which lines are speculatively loaded

Speculative Coherence Example

Epoch 5: Store X=3
Epoch 6: Load X

- Processor
- Cache
- State: Excl.
- Tag: X
- Data: 2

Spec. Loaded

Sp Read-Ex (epoch 5)
Shared Memory (X=2)

Speculative Coherence Example

Epoch 5: Load X
Epoch 6: Load X

- Processor
- Cache
- State: Invalid
- Tag: -
- Data: -

Spec. Loaded

Sp Inv (epoch 5)
Shared Memory (X=2)

epoch 5 < epoch 6, and speculatively loaded
Speculative Coherence Example

Epoch 5:
- Store X=3
- Speculation failed!

Epoch 6:
- Load X
- Sp Inv (epoch 5)
- Sp Read-Ex (epoch 5)
- Data

Shared Memory (X=2)

Speculation fails for epoch 6

Highlights of our scheme:
- Detection of a data dependence violation
- Speculatively modified and shared cache lines

Speculative Coherence Example

Epoch 5:
- Store X=3

Epoch 6:
- Load X
- speculation failed!

Speculative Coherence Example

Epoch 4:
- Load X

Epoch 5:
- Store X=3

Epoch 6:
- Load X

Speculative Coherence Example

Epoch 4:
- Load X

Epoch 5:
- Store X=3

Epoch 6:
- Load X

Speculative Coherence Example

Epoch 4:
- Load X

Epoch 5:
- Store X=3

Epoch 6:
- Load X

Speculative Coherence Example

Epoch 4:
- Load X

Epoch 5:
- Store X=3
A Scalable Approach to Thread-Level Speculation

Speculative Coherence Example

Epoch 4:
- Load X
- Processor
- Cache
  - State: Invalid
  - Tag: -
  - Data: -
- Read
- Shared Memory (X=2)

Epoch 5:
- Store X=3
- Processor
- Cache
  - State: Excl.
  - Tag: X
  - Data: 3
- Spec. Modified

Summary of New Speculative Line State

- New cache line state:
  - has it been speculatively loaded?
    - detect dependence violations
  - has it been speculatively modified?
    - buffer speculative modifications
  - is it in a speculative shared or exclusive state?
    - important performance optimizations

What if a speculative cache line is replaced?
- speculation fails for that epoch

both speculatively modified and shared!

multiple versions of the same cache line
Implementation of Speculative State

```
<table>
<thead>
<tr>
<th>Processor</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>
```

modest amount of extra space

Life Cycle of an Epoch

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Sp Ex</th>
<th>Sp Sh</th>
<th>Speculative Coherence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Speculative</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Coherence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Commit?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Complete, Pass Homefree</td>
</tr>
</tbody>
</table>
```

When Speculation Fails

```
<table>
<thead>
<tr>
<th>Cache</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Flash Reset
When Speculation Fails

Cache
SL SM State Tag Data
0 0 Excl * *
0 0 Shared * *
0 1 Sp Ex * *
0 1 Sp Sh * *

Processor

If Set then Invalidate; Flash Reset

When Speculation Succeeds

Cache
SL SM State Tag Data
1 0 Sp Ex * *
1 0 Sp Sh * *
0 1 Sp Ex * *
1 1 Sp Sh * *

Processor

Flash Reset

Life Cycle of an Epoch

Time

Spawning

Becomes Speculative

Speculative Coherence

Mechanisms to Squash or Commit

Commit?

Complete, Pass Homefree
When Speculation Succeeds

When Speculation Succeeds

When Speculation Succeeds

When Speculation Succeeds

SM & Exclusive: Become Dirty

SM & Shared: Need Exclusive Access

want to avoid searching entire cache

ownership required buffer (ORB)

Upgrade-Request (X)
When Speculation Succeeds

Cache

<table>
<thead>
<tr>
<th>SL</th>
<th>SM</th>
<th>State</th>
<th>Tag Data</th>
<th>ORB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Excl</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Shared</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Sp Ex</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Sp Sh</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

If SM, Become Dirty; Flash Reset

Ack (X) Upgrade-Request (X)

Flush the ORB, then quick bit operations

Forwarding Data Between Epochs

Store X Load X

Wait

Signal

• predictable dependences cause frequent violations
• compiler inserts wait-signal synchronization

Speculation in a Shared Cache

Why?
1) Shared-cache multithreaded architectures
   • eg. simultaneous multithreading
2) Context switch to another chain of speculation
3) Start new epoch while current epoch waits to commit

How?

Replicate the speculative context
Outline

- Details of our Approach
- Performance
  - simulation infrastructure
  - single-chip multiprocessor performance
  - scaling beyond chip boundaries
- Conclusions

Simulation Infrastructure

Compiler system and tools based on SUIF
- help analyze dependences, insert synchronization
- produce MIPS binaries containing TLS primitives

Benchmarks (all run to completion)
- buk, compress95, jpeg, equake

Simulator
- superscalar, similar to MIPS R10K
- models all bandwidth and contention

replicate the speculative context

A Scalable Approach to Thread-Level Speculation
Carnegie Mellon

Support for Speculation in a Shared Cache

replicate the speculative context

A Scalable Approach to Thread-Level Speculation
Carnegie Mellon
### Pipeline Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue Width</td>
<td>4</td>
</tr>
<tr>
<td>Functional Units</td>
<td>2Int, 2FP, 1Mem, 1Bra</td>
</tr>
<tr>
<td>Reorder Buffer Size</td>
<td>32</td>
</tr>
<tr>
<td>Integer Multiply</td>
<td>12 cycles</td>
</tr>
<tr>
<td>Integer Divide</td>
<td>76 cycles</td>
</tr>
<tr>
<td>All Other Integer</td>
<td>1 cycle</td>
</tr>
<tr>
<td>FP Divide</td>
<td>15 cycles</td>
</tr>
<tr>
<td>FP Square Root</td>
<td>20 cycles</td>
</tr>
<tr>
<td>All Other FP</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>GShare (16KB, 8 history bits)</td>
</tr>
</tbody>
</table>

### Memory Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Line Size</td>
<td>32B</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>32KB, 4-way set-assoc</td>
</tr>
<tr>
<td>Data Cache</td>
<td>32KB, 2-way set-assoc, 2 banks</td>
</tr>
<tr>
<td>Unified Secondary Cache</td>
<td>2MB, 4-way set-assoc, 4 banks</td>
</tr>
<tr>
<td>Miss Handlers</td>
<td>8 for data, 2 for insts</td>
</tr>
<tr>
<td>Crossbar Interconnect</td>
<td>8B per cycle per bank</td>
</tr>
<tr>
<td>Minimum Miss Latency to Secondary Cache</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Minimum Miss Latency to Local Memory</td>
<td>75 cycles</td>
</tr>
<tr>
<td>Main Memory Bandwidth</td>
<td>1 access per 20 cycles</td>
</tr>
<tr>
<td>Intra-Chip Communication Latency</td>
<td>10 cycles</td>
</tr>
<tr>
<td>Inter-Chip Communication Latency</td>
<td>200 cycles</td>
</tr>
</tbody>
</table>

### Benchmark Details: Regions and Epochs

<table>
<thead>
<tr>
<th>Application</th>
<th>Unrolling Factor</th>
<th>Avg. Insts. per Epoch</th>
<th>Parallel Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>buk</td>
<td>8</td>
<td>81.0</td>
<td>22.8%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>135.0</td>
<td>33.8%</td>
</tr>
<tr>
<td>compress95</td>
<td>1</td>
<td>196.7</td>
<td>24.6%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>240.4</td>
<td>22.7%</td>
</tr>
<tr>
<td>jpeg</td>
<td>32</td>
<td>1467.9</td>
<td>8.2%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>80.8</td>
<td>2.2%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>84.0</td>
<td>5.0%</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>100.3</td>
<td>6.7%</td>
</tr>
<tr>
<td>equake</td>
<td>1</td>
<td>2925.5</td>
<td>39.3%</td>
</tr>
</tbody>
</table>

### Performance on a 4-Processor CMP

<table>
<thead>
<tr>
<th>Application</th>
<th>Overall Region Speedup</th>
<th>Parallel Coverage</th>
<th>Program Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>buk</td>
<td>2.26</td>
<td>56.6%</td>
<td>1.46</td>
</tr>
<tr>
<td>compress95</td>
<td>1.27</td>
<td>47.3%</td>
<td>1.12</td>
</tr>
<tr>
<td>equake</td>
<td>1.77</td>
<td>39.3%</td>
<td>1.21</td>
</tr>
<tr>
<td>jpeg</td>
<td>1.94</td>
<td>22.1%</td>
<td>1.08</td>
</tr>
</tbody>
</table>

Program speedups are limited by coverage.
A Scalable Approach to Thread-Level Speculation

Performance on a 4-Processor CMP

<table>
<thead>
<tr>
<th>Program Speedup</th>
<th>Buk</th>
<th>Compress95</th>
<th>Equake</th>
<th>Ijpeg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buk</td>
<td>2.26</td>
<td>1.46</td>
<td>1.27</td>
<td>1.12</td>
</tr>
<tr>
<td>Compress95</td>
<td>1.77</td>
<td>1.77</td>
<td>1.12</td>
<td>1.21</td>
</tr>
<tr>
<td>Equake</td>
<td>1.94</td>
<td>1.94</td>
<td>1.77</td>
<td>1.08</td>
</tr>
<tr>
<td>Ijpeg</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Varying the Number of Processors

- Buk and equake are memory-bound.
- Compress95 and ijpeg are computation-intensive.
- Buk and equake scale well.
- Passing the homefree token is not a bottleneck.

Program speedups are limited by coverage.
Performance of the ORB (on a 4-CMP)

<table>
<thead>
<tr>
<th>Application</th>
<th>Average Flush Latency (cycles)</th>
<th>ORB Size (entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average</td>
<td>Maximum</td>
</tr>
<tr>
<td>buk</td>
<td>13.95</td>
<td>2.38</td>
</tr>
<tr>
<td>compress95</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>equake</td>
<td>0.13</td>
<td>0.04</td>
</tr>
<tr>
<td>jpeg</td>
<td>1.06</td>
<td>0.17</td>
</tr>
</tbody>
</table>

A small ORB is sufficient

Tracking Dependences Per Cache Line

Problem:
- analogous to false sharing: false violations
- write-after-write dependences also cause violations
  - but not a true dependence!

Solution:
- track dependences at a word granularity
- have an SM and SL bit per word in each cache line

✿ is per-word state worth the extra overhead?

Scaling Beyond Chip Boundaries

Does it do any good?
- not for our 4 benchmarks
- adding this support showed no improvement

Why not?
- buk and equake have random access patterns
- compress95 is heavily synchronized
- jpeg is unrolled to avoid false sharing

✿ existing techniques for avoiding false sharing can address this problem

simulate architectures with 1, 2 and 4 nodes
Conclusions

The overheads of our scheme are low:
- mechanisms to squash or commit are not a bottleneck
- per-word speculative state is not always necessary

It offers compelling performance improvements:
- program speedups from 8% to 46% on a 4-processor CMP
- program speedups up to 75% on multi-chip architectures

It is scalable:
- coherence provides elegant data dependence tracking

Seamless TLS on a wide range of architectures