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!

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)

Example

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

Processor

= hash[10]
= hash[19]
= hash[21]
= hash[33]
= hash[30]
= hash[10]
= hash[25]
Example of Thread-Level Speculation

Time

Epoch 1
= hash[3]
... hash[10] = ...

Epoch 2
= hash[19]
... hash[21] = ...

Epoch 3
= hash[33]
... hash[30] = ...

Epoch 4
= hash[10]
... hash[25] = ...

Processor

Processor

Processor

Processor

Carnegie Mellon

Steffan & Mowry

Example of Thread-Level Speculation
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
A Scalable Approach to Thread-Level Speculation

Steffan & Mowry

Carnegie Mellon

Life Cycle of an Epoch

Spawning

Becomes Speculative

Commit?

Complete, Pass Homefree

Fast Commit:

Slow Commit:

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

Ensures 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
A Scalable Approach to Thread-Level Speculation

Steffan & Mowry

Carnegie Mellon

Life Cycle of an Epoch

Time

Spawned

Becomes Speculative

Speculative Coherence

Commit?

Complete, Pass Homefree

MESI Coherence Example

Thread A: Thread B:

Processor

Cache

State  Tag  Data
Invalid  -  -

Thread A:

Thread B:

Processor

Cache

State  Tag  Data
Invalid  -  -

Load X

Read

Shared Memory (X=2)

Thread A: Thread B:

Processor

Cache

State  Tag  Data
Invalid  -  -

Load X

Read

Shared Memory (X=2)

Thread A: Thread B:

Processor

Cache

State  Tag  Data
Invalid  -  -

Load X

Read

Fill

Shared Memory (X=2)

Thread A: Thread B:

Processor

Cache

State  Tag  Data
Invalid  -  -

Load X

Read

Excl.  X  2

Shared Memory (X=2)
A Scalable Approach to Thread-Level Speculation
Steffan & Mowry
Carnegie Mellon

**MESI Coherence Example**

**Thread A:**
- **Store X=3**
- Data: Invalid
- State: Invalid
- Tag: -

**Thread B:**
- **Load X**
- Data: Excl.
- State: X
- Tag: 2

Read-Exclusive

**Shared Memory (X=2)**

**Highlights:**
- Read-exclusive invalidates all other copies

**Speculative Coherence Example**

**Epoch 4:**
- **Load X**

**Epoch 5:**
- **Store X=3**

**Epoch 6:**
- **Load X**

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

**the state ‘dirty’ implies exclusiveness**

**Invalidation**
Speculative Coherence Example

Epoch 5:
- Processor
- Cache
  - State: Invalid
  - Tag: -
  - Data: -

Read
- Shared Memory (X=2)

Epoch 6:
- Load X
- Processor
- Cache
  - State: Excl.
  - Tag: X
  - Data: 2

Speculative Coherence Example

Epoch 5:
- Store X=3
- Processor
- Cache
  - State: Invalid
  - Tag: -
  - Data: -

Speculative Coherence Example

Epoch 6:
- Load X
- Processor
- Cache
  - State: Excl.
  - Tag: X
  - Data: 2

track which lines are speculatively loaded

Speculative Coherence Example

Epoch 5:
- Store X=3
- Processor
- Cache
  - Spec. Loaded

Speculative Coherence Example

Epoch 6:
- Load X
- Processor
- Cache
  - Spec. Loaded

speculative msgs piggyback epoch number

epoch5 < epoch6, and speculatively loaded

Sp Read-Ex (epoch5)

Sp Inv (epoch5)
Speculative Coherence Example

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

speculation failed!

Processor
Cache
State Tag Data
Invalid - -

Spec Read-Ex (epoch 5)
Spec Inv (epoch 5)

Shared Memory (X=2)

f speculation fails for epoch 6

Epoch 4:
Load X

Epoch 5:
Store X=3

Epoch 6:
Load X

Speculative Coherence Example

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!

Processor
Cache
State Tag Data
Excl. X 3

Specified Modified

Fill
Spec Read-Ex (epoch 5)
Spec Inv (epoch 5)

Shared Memory (X=2)

track which lines are speculatively modified
A Scalable Approach to Thread-Level Speculation

Speculative Coherence Example

Epoch 4: Load X

Cache
State Invalid
Tag -
Data -

Processor

Spec. Modified

Read

Shared Memory (X=2)

Epoch 5: Store X=3

Cache
State Excl.
Tag X
Data 3

Processor

Spec. Modified

notify shared

Shared Memory (X=2)

both speculatively modified and shared!

Speculative Coherence Example

Epoch 4: Load X

Cache
State Invalid
Tag -
Data -

Processor

Spec. Modified

Read

Shared Memory (X=2)

Epoch 5: Store X=3

Cache
State Shared
Tag X
Data 3

Processor

Spec. Modified

notify shared

Shared Memory (X=2)

multiple versions of the same cache line

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
A Scalable Approach to Thread-Level Speculation
Steffan & Mowry
Carnegie Mellon

Implementation of Speculative State

Processor

Cache

State  Tag  Data
-  -  -
-  -  -
-  -  -
-  -  -
-  -  -

Speculatively Loaded
Speculatively Modified

modest amount of extra space

Life Cycle of an Epoch

Time

Spawned

Speculative Coherence

Becomes Speculative

Commit?

Complete, Pass Homefree

When Speculation Fails

Processor

Cache

SL  SM  State  Tag  Data
1  0  Sp Ex  *  *
1  0  Sp Sh  *  *
...  ...
0  1  Sp Ex  *  *
1  1  Sp Sh  *  *
When Speculation Fails

<table>
<thead>
<tr>
<th>Cache</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL</td>
<td>SM</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

If Set then
Invalidate; Flash Reset

quick bit operation

Life Cycle of an Epoch

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spawned</td>
</tr>
<tr>
<td>Becomes Speculative</td>
</tr>
<tr>
<td>Speculative Coherence</td>
</tr>
<tr>
<td>Commit?</td>
</tr>
<tr>
<td>Complete, Pass Homefree</td>
</tr>
</tbody>
</table>

When Speculation Succeeds

<table>
<thead>
<tr>
<th>Cache</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL</td>
<td>SM</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Flash Reset
A Scalable Approach to Thread-Level Speculation

When Speculation Succeeds

When Speculation Succeeds

Ownership required buffer (ORB)

Upgrade-Request (X)
When Speculation Succeeds

<table>
<thead>
<tr>
<th>Cache</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL SM</td>
<td>State</td>
</tr>
<tr>
<td>0 0</td>
<td>Excl</td>
</tr>
<tr>
<td>0 0</td>
<td>Shared</td>
</tr>
<tr>
<td>0 1</td>
<td>Sp Ex</td>
</tr>
<tr>
<td>0 1</td>
<td>Sp Sh</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

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

Synchronize to avoid violations

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
Support for Speculation in a Shared Cache

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

detailed simulation!
Pipeline Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>
A Scalable Approach to Thread-Level Speculation

Steffan & Mowry
Carnegie Mellon

Performance on a 4-Processor CMP

<table>
<thead>
<tr>
<th>Region</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>buk</td>
<td>2.26</td>
</tr>
<tr>
<td>compress95</td>
<td>1.77</td>
</tr>
<tr>
<td>equake</td>
<td>1.77</td>
</tr>
<tr>
<td>ijpeg</td>
<td>1.94</td>
</tr>
</tbody>
</table>

Parallel Coverage: 56.6% 47.3% 39.3% 22.1%

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
### Performance of the ORB (on a 4-CMP)

<table>
<thead>
<tr>
<th>Application</th>
<th>Average Flush Latency (cycles)</th>
<th>Average ORB Size (entries)</th>
<th>Maximum ORB Size (entries)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buk</td>
<td>13.95</td>
<td>2.38</td>
<td>9</td>
</tr>
<tr>
<td>compress95</td>
<td>0.04</td>
<td>0.01</td>
<td>8</td>
</tr>
<tr>
<td>equake</td>
<td>0.13</td>
<td>0.04</td>
<td>12</td>
</tr>
<tr>
<td>jpeg</td>
<td>1.06</td>
<td>0.17</td>
<td>5</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

$$\text{is per-word state worth the extra overhead?}$$

---

### Tracking Dependences Per Cache Line

**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

$$\text{existing techniques for avoiding false sharing can address this problem}$$

---

### Scaling Beyond Chip Boundaries

$$\text{simulate architectures with 1, 2 and 4 nodes}$$
A Scalable Approach to Thread-Level Speculation
Steffan & Mowry
Carnegie Mellon

Scaling Beyond Chip Boundaries

multi-chip systems benefit from TLS

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

our scheme scales well