Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors

Marcelo Cintra and Josep Torrellas

University of Edinburgh
http://www.dcs.ed.ac.uk/home/mc

University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu
Speculative Parallelization

- Assume no dependences and execute threads in parallel
- Track data accesses at run-time
- Detect violations
- Squash offending threads and restart them

```c
for(i=0; i<100; i++) {
    ... = A[L[i]] + ...  
    A[K[i]] = ...
}
```

\[ \text{Iteration } J \]
\[ \text{Iteration } J+1 \]
\[ \text{Iteration } J+2 \]
Squashing in Speculative Parallelization

- Speculative Parallelization: Threads may get squashed

- Dependence violations are statically unpredictable

- False sharing may cause further squashes
Squashing is very costly
Contribution: Eliminate Squashes

- Framework of hardware mechanisms to eliminate squashes

- Based on learning and prediction of violations

- Improvement: average of 4.3 speedup over plain squash-and-retry for 16 processors
Outline

- Motivation and Background
- Overview of Framework
- Mechanisms
- Implementation
- Evaluation
- Related Work
- Conclusions
# Types of Dependence Violations

<table>
<thead>
<tr>
<th>Type</th>
<th>Avoiding Squashes</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>Disambiguate line addresses</td>
<td>Plain Speculation</td>
</tr>
</tbody>
</table>
## Types of Dependence Violations

<table>
<thead>
<tr>
<th>Type</th>
<th>Avoiding Squashes</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>Disambiguate line addresses</td>
<td>Plain Speculation</td>
</tr>
<tr>
<td>False</td>
<td>Disambiguate word addresses</td>
<td>Delay &amp; Disambiguate</td>
</tr>
</tbody>
</table>
# Types of Dependence Violations

<table>
<thead>
<tr>
<th>Type</th>
<th>Avoiding Squashes</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>Disambiguate line addresses</td>
<td>Plain Speculation</td>
</tr>
<tr>
<td>False</td>
<td>Disambiguate word addresses</td>
<td><em>Delay &amp; Disambiguate</em></td>
</tr>
<tr>
<td>Same-word, predictable value</td>
<td>Compare values produced and consumed</td>
<td><em>Value Predict</em></td>
</tr>
</tbody>
</table>
## Types of Dependence Violations

<table>
<thead>
<tr>
<th>Type</th>
<th>Avoiding Squashes</th>
<th>Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>Disambiguate line addresses</td>
<td>Plain Speculation</td>
</tr>
<tr>
<td>False</td>
<td>Disambiguate word addresses</td>
<td>Delay &amp; Disambiguate</td>
</tr>
<tr>
<td>Same-word, predictable value</td>
<td>Compare values produced and consumed</td>
<td>Value Predict</td>
</tr>
<tr>
<td>Same-word, unpredictable value</td>
<td>Stall thread, release when safe</td>
<td>Stall &amp; Release</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stall &amp; Wait</td>
</tr>
</tbody>
</table>
Learning and Predicting Violations

- Monitor violations at directory
- Remember data lines causing violations
- Count violations and choose mechanism

Diagram:

- Plain Speculation
- Delay & Disambiguate
- Stall & Release
- Stall & Wait
- Value Predict

Age and potential violation connections:

- Violations and squashes
- Violation and squash
Outline

- Motivation and Background
- Overview of Framework
- Mechanisms
- Implementation
- Evaluation
- Related Work
- Conclusions
Delay&Disambiguate

- Assume potential violation is false
- Let speculative read proceed
- Remember unresolved potential violation
- Perform a *late* or *delayed* per-word disambiguation when consumer thread becomes non-speculative
  - No per-word access information at directory
  - No word addresses on memory operations
- Squash if same-word violation
Delay & Disambiguate (Successful)

Producer

i

i+j

Consumer

i+j+1

i+j+2

Useful work

Delayed disambiguation overhead

Time
Delay & Disambiguate (Unsuccessful)

Producer

Consumer

Time

Useful work
Wasted correct work
Possibly correct work
Squash overhead
Delayed disambiguation overhead
ValuePredict

- Predict value based on past observed values
  - Assume value is the same as last value written (*last-value prediction, value reuse, or silent store*)
  - More complex predictors are possible
- Provide predicted value to consumer thread
- Remember predicted value
- Compare predicted value against correct value when consumer thread becomes non-speculative
- Squash if mispredicted value
Stall&Release

- Stall consumer thread when attempting to read data
- Release consumer thread when a predecessor commits modified copy of the data to memory
  - Provided no intervening thread has a modified version
- Squash released thread if a later violation is detected
Stall&Release (Successful)

Producer

\[ i \]

\[ i+j \]

Consumer

\[ i+j+1 \]

\[ i+j+2 \]

Wr

Rd

Useful work

Stall overhead

Time
Stall&Wait

- Stall consumer thread when attempting to read data
- Release consumer thread only when it becomes non-speculative
Outline

- Motivation and Background
- Overview of Framework
- Mechanisms
- Implementation
- Evaluation
- Related Work
- Conclusions
Baseline Architecture

Processor + Caches

Memory

Directory Controller

Conventional support for speculative parallelization:

Per-word access bits in caches

Speculation module with per-line access bits

Network
Speculation Module

- One entry per memory line speculatively touched
- Load and Store bits per line per thread
- Mapping of threads to processors and ordering of threads

<table>
<thead>
<tr>
<th>Line Tag</th>
<th>Valid Bit</th>
<th>Load Bits</th>
<th>Store Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Global Memory Disambiguation Table (GMDT)
(Cintra, Martínez, and Torrellas – ISCA’00)
Enhanced Architecture

New modules:

VPT: Violation Prediction Table
Monitor and learn violations + enforce our mechanisms

LDE: Late Disambiguation Engine
Use local per-word info to support the Delay & Disambiguate and ValuePredict
Violation Prediction Table (VPT)

- Entries for lines recently causing potential violations
- Appended to every row in the GMDT

<table>
<thead>
<tr>
<th>Line Tag</th>
<th>Valid Bit</th>
<th>Load Bits</th>
<th>Store Bits</th>
<th>Valid Bit</th>
<th>State Bits</th>
<th>Squash Counter</th>
<th>SavedSquash Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intl. Symp. on High Performance Computer Architecture - February 2002
VPT Pending Transactions Buffer

- Entries for pending transactions on VPT lines

<table>
<thead>
<tr>
<th>Line Tag</th>
<th>Valid Bit</th>
<th>Thread ID</th>
<th>State Bits</th>
<th>Modified Mask</th>
<th>Predicted Line Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2K-entry VPT: < 2Kbytes
128-entry Pend. Trans. Buffer: < 10Kbytes
Operation under Delay & Disambiguate

1. Load word 1
2. Store word 3
3. Commit
4. Commit
5. Delayed Disambiguation request
6. No squash

<table>
<thead>
<tr>
<th>State</th>
<th>Thread ID</th>
<th>Mask</th>
<th>Load Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>D&amp;D</td>
<td>i+j+k</td>
<td>1000</td>
<td>0010</td>
</tr>
</tbody>
</table>

thread i

thread i+j

thread i+j+k

VPT Pend. Trans. Buffer
Operation under Delay & Disambiguate

- Thread i
  2. Store word 3
  4. Commit

- Thread i+j
  3. Store word 1
  5. Commit

- Thread i+j+k
  1. Load word 1
  6. Delayed Disambiguation request
  7. Squash

State
- D&D
- VPT

Thread ID
- i+j+k

Mask
- 1010

Load Bits
- 0010

VPT Pend. Trans. Buffer
Outline

- Motivation and Background
- Overview of Framework
- Mechanisms
- Implementation
- Evaluation
- Related Work
- Conclusions
Simulation Environment

- Execution-driven simulation
- Detailed superscalar processor model
- Coherent+speculative memory back-end
- Directory-based multiprocessor: 4 nodes

Node: spec CMP + 1M L2 + 2K-entry GMDT + Mem.
CMPl: 4 x (processor + 32K L1)
Processor: 4-issue, dynamic
Applications

- Applications dominated by non-analyzable loops with cross-iteration dependences → average of 60% of sequential time
- TRACK (PERFECT)
  EQUAKE, WUPWISE (SPECfp2000)
  DSMC3D, EULER (HPF2)
- Non-analyzable loops and accesses identified by the Polaris parallelizing compiler
- Results shown for the non-analyzable loops only
Delay & Disambiguate Performance

![Chart showing normalized execution time for different applications and delays.]
Delay & Disambiguate Performance

Plain line-based protocol is slow

Normalized Execution Time (%)
Delay & Disambiguate Performance

Most squashes eliminated with Delay & Disambiguate
Delay & Disambiguate Performance

Our scheme gets very close to oracle.
Complete Framework Performance

![Normalized Execution Time (%)](chart.png)

- TRACK
- DSMC3D
- EULER
- EQUAKE
- WUPWISE

- **Baseline**
- **Word**
- **Combined**
- **Oracle**

**Normalized Execution Time (%)**

- `2.6` for TRACK
- `0.8` for DSMC3D
- `1.1` for EULER
- `0.8` for EQUAKE
- `2.6` for WUPWISE

**Legend**
- Ovhd+Other
- Squash
- Stall
- Busy+Mem
Complete Framework Performance

Complete framework performs as well as or better than each mechanism alone.
Complete Framework Performance

Framework gets very close to oracle
Outline

- Motivation and Background
- Overview of Framework
- Mechanisms
- Implementation
- Evaluation
- Related Work
- Conclusions
Related Work

Other dynamic learning, prediction, and specialized handling of dependences for speculative parallelization:

– Synchronization: Multiscalar; TLDS
  ▪ Mostly tailored to CMP
  ▪ Learning based on instruction addresses (Multiscalar)

– Value Prediction: Clustered Speculative; TLDS
  ▪ We use compiler to eliminate trivial dependences (like TLDS)
  ▪ We use floating-point applications

– Mixed word/line speculation: I-ACOMA
  ▪ We never require word addresses on memory operations
Outline

- Motivation and Background
- Overview of Framework
- Mechanisms
- Implementation
- Evaluation
- Related Work
- Conclusions
Conclusions

- Framework of hardware mechanisms eliminates most squashes
- Very good performance of line-based speculative machine with framework:
  - 4.3 times faster than a line-based speculative machine without framework
  - 1.2 times faster than expensive word-based speculative machine
- Delay&Disambiguate has largest impact but combination with Stall&Wait is best
- ValuePredict did not work well in our environment
Eliminating Squashes Through Learning Cross-Thread Violations in Speculative Parallelization for Multiprocessors

Marcelo Cintra and Josep Torrellas

University of Edinburgh
http://www.dcs.ed.ac.uk/home/mc

University of Illinois
at Urbana-Champaign
http://iacoma.cs.uiuc.edu
VPT Fields

- **State Bits**: the mechanism currently in use for the line
- **Squash Counter**: number of squashes to the line incurred with current mechanism
- **SavedSquashBit**: squash saved by the current mechanism
- **ThrsStallR**: number of squashes to trigger Stall&Release mechanism
- **ThrsStallW**: number of squashes to trigger Stall&Wait mechanism
- **AgePeriod**: number of commits to age the state of line
Pending Transactions Buffer Fields

- **Line Tag**: address of line with unresolved violation
- **Thread ID**: consumer thread with a unresolved violation
- **State Bits**: mechanism used
- **Modified Mask**: bitmap of all modifications to words of the line by predecessor threads
- **Predicted Line Value**: actual data values provided to consumer thread
Operation under ValuePredict

- 1. Load word 1 (value = 25)
- 2. Store word 1
- 3. Commit
- 4. Commit
- 5. Delayed Disambiguation request
- 6. No squash

<table>
<thead>
<tr>
<th>State</th>
<th>Thread ID</th>
<th>Mask</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP</td>
<td>i+j+k</td>
<td>0000</td>
<td>25</td>
</tr>
<tr>
<td>VPT</td>
<td></td>
<td></td>
<td>VPT Pend. Trans. Buffer</td>
</tr>
</tbody>
</table>

Load Bits: 0010
Operation under ValuePredict

2. Store word 1 (value = 26)
3. Commit

4. Commit

1. Load word 1
5. Delayed Disambiguation request
6. Squash

<table>
<thead>
<tr>
<th>Thread ID</th>
<th>Mask</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+j+k</td>
<td>0010</td>
<td>25</td>
</tr>
</tbody>
</table>

State
- VP
- VPT

Thread ID
- i+j+k

Load Bits
- 0010
Summary

+ Delay&Disambiguate offers word-level disambiguation without word addresses in most memory transactions
+ Simple implementation combines all different mechanisms seamlessly
+ Learning/Prediction policy effective when subset of speculative data tends to be accessed with dependences
- Learning/Prediction policy not so effective when subset of speculative instructions tends to cause dependences
- Learning/Prediction policy reacts to previous behavior but cannot extrapolate/anticipate behavior
# Simulation Environment

<table>
<thead>
<tr>
<th>Processor Param.</th>
<th>Value</th>
<th>Memory Param.</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Issue width</td>
<td>4</td>
<td>L1,L2,VC size</td>
<td>32KB,1MB,64KB</td>
</tr>
<tr>
<td>Instruction window size</td>
<td>64</td>
<td>L1,L2,VC assoc.</td>
<td>2-way,4-way,8-way</td>
</tr>
<tr>
<td>No. functional units (Int,FP,Ld/St)</td>
<td>3,2,2</td>
<td>L1,L2,VC,line size</td>
<td>64B,64B,64B</td>
</tr>
<tr>
<td>No. renaming registers (Int,FP)</td>
<td>32,32</td>
<td>L1,L2,VC,latency</td>
<td>1,12,12 cycles</td>
</tr>
<tr>
<td>No. pending memory ops. (Ld,St)</td>
<td>8,16</td>
<td>L1,L2,VC banks</td>
<td>2,3,2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Local memory latency</td>
<td>75 cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2-hop memory latency</td>
<td>290 cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3-hop memory latency</td>
<td>360 cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GMDT size</td>
<td>2K entries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GMDT assoc.</td>
<td>8-way</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GMDT/VPT lookup</td>
<td>20 cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pend. Trans. Buffer size</td>
<td>128 entries</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pend. Trans. Buffer scan</td>
<td>3 cycles/entry</td>
</tr>
</tbody>
</table>
## Application Characteristics

<table>
<thead>
<tr>
<th>Application</th>
<th>Loops</th>
<th>% of Seq. Time</th>
<th>RAW Dependences</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACK</td>
<td>nlfilt_300</td>
<td>58</td>
<td>Same-word and False</td>
</tr>
<tr>
<td>DSMC3D</td>
<td>move3_100</td>
<td>41</td>
<td>Same-word and False</td>
</tr>
<tr>
<td>EULER</td>
<td>dflux_[100,200]</td>
<td>90</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>psmoo_20</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>eflux_[100,200,300]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQUAKE</td>
<td>smvp_1195</td>
<td>45</td>
<td>Same-word</td>
</tr>
<tr>
<td>WUPWISE</td>
<td>muldeo_200’</td>
<td>67</td>
<td>Same-word and False</td>
</tr>
<tr>
<td></td>
<td>muldoe_200’</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Squash Behavior

![Bar chart showing the fraction of squashes per line for different programs such as TRACK, DSMC3D, EULER, EQUAKE, and WUPWISE. The chart is divided into categories based on the number of squashes per line, with different colors representing different types of squashes.]

Int. Symp. on High Performance Computer Architecture - February 2002
Stall&Wait Performance

![Graph showing normalized execution time (%)]
Stall&Wait Performance

Some speedup, but significantly limited by stall time

<table>
<thead>
<tr>
<th></th>
<th>Euler</th>
<th>Equake</th>
<th>Wupwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>2.6</td>
<td>0.8</td>
<td>2.6</td>
</tr>
<tr>
<td>Word</td>
<td>8.7</td>
<td>2.2</td>
<td>8.7</td>
</tr>
<tr>
<td>S&amp;Wait_14</td>
<td>2.8</td>
<td>2.2</td>
<td>2.8</td>
</tr>
<tr>
<td>Oracle_Stall</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>S&amp;Wait_14</td>
<td>1.8</td>
<td>2.1</td>
<td>2.1</td>
</tr>
<tr>
<td>Stall</td>
<td>2.7</td>
<td>2.1</td>
<td>2.6</td>
</tr>
<tr>
<td>Stall</td>
<td>1.1</td>
<td>2.1</td>
<td>1.5</td>
</tr>
</tbody>
</table>

Normalized Execution Time (%)
Stall&Wait Performance

Our scheme gets close to oracle most of the times.
Related Work

- Other hardware-based speculative parallelization:
  - Multiscalar (Wisconsin); Hydra (Stanford);Clustered Speculative (UPC); TLDS (CMU); MAJC (Sun);
    Superthreaded (Minnesota); Illinois
  - Mostly tailored for CMP
  - Mostly word-based speculation
  - Mostly squash-and-retry