The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization

J. Gregory Steffan and Todd C. Mowry
Department of Computer Science
Carnegie Mellon University

http://www.cs.cmu.edu/~{steffan,tcm}

{steffan,tcm}@cs.cmu.edu
State-of-the-Art vs. Future Processors

**Challenge:** translating these resources into higher performance

**One possibility:** multiple processors on a single chip
Performance Benefits of Single-Chip Multiprocessing

**Multiprogramming Workload:**
- improved *throughput*

**Single Application:**
- how to reduce *execution time* => must contain parallel threads

**The Big Question:**
- *how do we automatically parallelize all applications?*
State-of-the-Art in Automatic Parallelization

**Numeric Applications:**

- dominated by regular array references inside loops:
  
  ```
  FOR i = 1 to N
    FOR j = 1 to N
      FOR k = 1 to N
        C[i][j] += A[i][k] * B[k][j];
  ```

- significant progress has been made
  - e.g., fastest SPECfp95 number (Bugnion et al. 1996)

**Non-Numeric Applications:**

- access patterns & control flow may be highly irregular
  - pointer dereferences, recursion, unstructured loops, etc.
- little (if any) success in automatic parallelization
- but these applications are important!

we must expand the scope of automatic parallelization
Why Is Automatic Parallelization So Difficult?

**Current Approach:**

- parallelize only if we can statically prove independence

```
FOR i = 1 to N
  A[i] += i;
FOR i = 1 to N
  A[i] += f(A[i-1]);
```

- Parallel
- Sequential

- transformations can help to eliminate dependences

**For Non-Numeric Codes:**

- understanding memory addresses is extremely difficult

```
while (foo()) {
  x = *q;
  ...
  *p = bar();
  ...
}
```

**Major Limitation:**

- when the compiler is uncertain, it must be conservative
Expanding the Scope of Automatic Parallelization

The Problem:

- statically proving independence is hopelessly restrictive
  - a full understanding of memory addresses is unrealistic
- instead, we should be focusing on performance issues

Our Solution:

Thread-Level Data Speculation (TLDS)
Overview

- Thread-Level Data Speculation (TLDS)
- An Example: Compress
- Experimental Results
- Architectural Support
- Conclusions
**Data Speculation**

**Basic Idea:**
- optimistically perform access assuming no dependence
- if speculation was unsafe, invoke recovery action

**Example:**

```
a = f()
STORE *p = a
LOAD b = *q
b = a
```

Original Execution

```
LOAD b = *q
a = f()
STORE *p = a
```

Instruction-Level Data Speculation
Thread-Level Data Speculation

- Analogous to instruction-level data speculation
  - except that it involves separate, parallel threads of control

Original Execution

Thread-Level Data Speculation
Related Work

Prior to this Study:

- **Wisconsin Multiscalar Architecture** (Sohi et al, 1995)
  - tightly-coupled ring architecture with register forwarding
  - “ARB” detects memory dependences, hardware rollback
  + requires relatively little software support
  - large, centralized ARB may increase load latency
  - ring architecture limits flexibility (multiprogramming, locality)

Other Recent Work:

- **Stanford Hydra** (Oplinger et al, 1997)
- **Wisconsin Speculative Versioning Cache** (Gopal et al, 1997)
- **Illinois Speculative Run-Time Parallelization** (Zhang et al, 1998)
Objectives of This Study

- Does TLDS offer compelling performance improvements?
  - study of the SPEC92 and SPEC95 integer benchmarks

- Can we provide cost-effective hardware support for TLDS?
  - detecting dependence violations
  - recovering from failed speculation

  goal: performance of non-TLDS code is not sacrificed

- What compiler support is necessary to exploit TLDS?
  - optimizations, scheduling, etc.
Overview

✓ Thread-Level Data Speculation (TLDS)

☞ An Example: Compress

• Experimental Results
• Architectural Support
• Conclusions
An Example: Compress

```
while ((c = getchar()) != EOF) {
    /* perform data compression */
    ...
    ... = hash[hash_function(c)];
    ...
    hash[hash_function(...)] = ...;
    ...
}
```

**Epoch i**

**Potential Source of Parallelism:**

- data parallelism across the input stream?

**From the Compiler’s Perspective:**

- hash accesses cannot be statically disambiguated

**In Reality:**

- consecutive characters rarely hash to the same entry
TLDS Execution of Compress

while (...) {
    ...
    x = hash[idx1];
    ...
    hash[idx2] = y;
    ...
}

Epoch 1

Processor 1

= hash[3]
...
hash[10] =
...
attempt_commit() ✓

Epoch 2

Processor 2

= hash[19]
...
hash[21] =
...
attempt_commit() ✓

Epoch 3

Processor 3

= hash[33]
...
hash[30] =
...
attempt_commit() ✓

Epoch 4

Processor 4

= hash[10]
...
hash[25] =
...
attempt_commit() ✗

Epoch 5

Processor 1

= hash[31]
...

Epoch 6

Processor 2

= hash[9]
...

Epoch 7

Processor 3

= hash[27]
...

Epoch 4

Processor 4

= hash[10]
...
hash[25] =
...

Violation!

Retry

Time

✓ ✓ ✓ ✗
Other Data Dependences in Compress

```c
while ((c = getchar()) != EOF) {
    /* perform data compression */
    in_count++;
    ...
    if (...) {out_count++; putchar();...}
    if (free_entries < ...)
        free_entries = ...
    ...
}
```

- **in_count**:
  - induction variable ➔ *implicit in the epoch number*

- **out_count**:
  - reduction operation ➔ *compute partial sums*

- **getchar(), putchar()**:
  - *use parallel library routines (also, malloc(), etc.)*

- **free_entries**:
  - cannot eliminate dependence ➔ *forward between epochs*
Overview

- Thread-Level Data Speculation (TLDS)
- An Example: Compress

- Experimental Results
  - Relaxing Memory Dependences
  - Forwarding Data Between Epochs
  - Speedups

- Architectural Support
- Conclusions
## Benchmarks

<table>
<thead>
<tr>
<th>Suite</th>
<th>Name</th>
<th>Region</th>
<th>Average Dynamic Instrs per Epoch</th>
<th>% of Total Dynamic Instrs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPEC92</strong></td>
<td><strong>compress</strong></td>
<td>r1</td>
<td>89</td>
<td>99.9</td>
</tr>
<tr>
<td></td>
<td><strong>gcc</strong></td>
<td>r1</td>
<td>1092</td>
<td>8.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2</td>
<td>1593</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td><strong>espresso</strong></td>
<td>r1</td>
<td>32</td>
<td>19.4</td>
</tr>
<tr>
<td></td>
<td><strong>li</strong></td>
<td>r1</td>
<td>19</td>
<td>21.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2</td>
<td>286</td>
<td>51.2</td>
</tr>
<tr>
<td></td>
<td><strong>sc</strong></td>
<td>r1</td>
<td>36</td>
<td>69.3</td>
</tr>
<tr>
<td><strong>SPEC95</strong></td>
<td><strong>m88ksim</strong></td>
<td>r1</td>
<td>1232</td>
<td>99.3</td>
</tr>
<tr>
<td></td>
<td><strong>ijpeg</strong></td>
<td>r1</td>
<td>9406</td>
<td>15.3</td>
</tr>
<tr>
<td></td>
<td><strong>perl</strong></td>
<td>r1</td>
<td>67</td>
<td>35.8</td>
</tr>
<tr>
<td></td>
<td><strong>go</strong></td>
<td>r1</td>
<td>80</td>
<td>6.8</td>
</tr>
<tr>
<td><strong>NAS Parallel</strong></td>
<td><strong>buk</strong></td>
<td>r1</td>
<td>26</td>
<td>16.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2</td>
<td>18</td>
<td>11.4</td>
</tr>
</tbody>
</table>
Measuring Memory Dependences: Run Lengths

**Run Length:**

- # of epochs between Read-After-Write (RAW) dependences

![Diagram showing epochs and run lengths](image)

**Average Run Length = 3**

**average run length = rough limit of potential parallelism**
Relaxing Memory Data Dependences

- eliminating dependences and forwarding scalars are important
  - significant parallelism is available in many cases
Forwarding Data Between Epochs

Scalar Memory Values:

- forward if dependences occur frequently
  - synchronization is faster than speculation recovery
  
  helpful for performance, not necessary for correctness

Register Values:

- must be forwarded to maintain correctness

- some register dependences may be eliminated:
  - induction variables
  - through simple loop rescheduling

- all other register dependences forwarded through memory

what is the impact on performance?
Critical Path Lengths

Course-Grain Synchronization

- $E_i$
- WAIT
- LOAD A
- STORE A
- LOAD B
- STORE B
- SIGNAL

Fine-Grain Synchronization

- $E_i$
- WAIT A
- LOAD A
- STORE A
- SIGNAL A
- WAIT B
- LOAD B
- STORE B
- SIGNAL B
• fine-grain synchronization is helpful
• with aggressive instruction scheduling, forwarding is not a bottleneck
Potential Region Speedup on 4 Processors

- aggressive instruction scheduling is a major performance win
- potential speedups of twofold or more in 11 of 13 regions
Program Speedups

Coverage: 99.9% 12.1% 19.4% 73.1% 69.3% 99.3% 15.3% 35.8% 6.8% 27.9%

Region Speedups:
Overview

✓ Thread-Level Data Speculation (TLDS)
✓ An Example: Compress
✓ Experimental Results

 Architectural Support

• Communication Latency
• Key Architectural Issues
• Detecting Data Dependence Violations
• Buffering Speculative State

• Conclusions
Base Architecture

- Each processor has its own L1 data cache
  - maintain single-cycle load latency
- L1 caches are kept coherent
  - shared-memory programming model
How Important Is Communication Latency?

Some Options:

- direct L1-to-L1 communication ➔ ~2 cycles
- communicate through the L2 cache ➔ ~10 cycles
Impact of Communication Latency

- instruction scheduling reduces the sensitivity to communication latency
- communicating through the L2 cache is a viable option
Key Architectural Issues

• Thread Management
  • Thread creation and epoch scheduling
  • Epoch numbers must be visible to the hardware
  • Distinguishing speculative vs. non-speculative memory accesses
  • Recovering from data dependence violations:
    • hardware notifies software of violation
    • software performs the bulk of the recovery

• Detecting Data Dependence Violations
  extend invalidation-based cache coherence

• Buffering Speculative State
  extend the functionality of the primary data caches
Invalidation-Based Cache Coherence

Processor 1

\[
\begin{align*}
\text{\ldots} \\
\text{\ldots} \\
\text{\ldots} \\
\text{\textcolor{red}{\textbf{2}}} \text{ STORE } X = 2; \\
\end{align*}
\]

L1 Cache

\[
\begin{array}{c}
X = 1 \rightarrow 2 \\
\end{array}
\]

Processor 2

\[
\begin{align*}
\text{\textcolor{red}{\textbf{1}}} \text{ LOAD } a = X; \\
\text{\ldots} \\
\text{\ldots} \\
\end{align*}
\]

L1 Cache

\[
\begin{array}{c}
X = 1 \\
\end{array}
\]

\begin{align*}
\text{Read Request} \\
\text{Invalidation} \\
\text{\textcolor{red}{\textbf{2}}} \\
\end{align*}
Detecting Data Dependence Violations

Processor 1

- Epoch 5
  - ...
  - ...
  - STORE *q = 2;
  - ...

- Epoch # = 5
  - Violation? = False

  \[
  x = 1 \rightarrow 2 \quad \text{F F}
  \]
  - SL
  - SM

  Invalidation (Epoch #5)

Processor 2

- Epoch 6
  - become_speculative()
  - LOAD a = *p;
  - ...
  - ...
  - attempt_commit()

- Epoch # = 6
  - Violation? = TRUE

  \[
  x = 1 \rightarrow T \quad \text{F}
  \]
  - SL
  - SM

Speculatively Loaded?
Speculatively Modified?

FAILS!
Buffering Speculative State

Speculative Stores:
• software cannot realistically roll back memory side effects
  ➡️ our solution: buffer in L1 cache until safe to commit to memory

Speculative Loads:
• if displaced, then we can no longer track dependence violations
• set violation flag upon eviction of a speculatively accessed line
  ➡️ correctness is preserved, but performance may suffer

➡️ a 16KB 2-way set-associative cache with 4 victim entries suffices
Overview

✓ Thread-Level Data Speculation (TLDS)
✓ An Example: Compress
✓ Experimental Results
✓ Architectural Support
☞ Conclusions
Conclusions

- **TLDS potentially offers compelling performance improvements**
  - 12 of 13 regions: speedups of 1.78 - 3.97 on 4 processors
  - 7 of 10 programs: speedups of 1.15 - 3.87 on 4 processors

- **Only modest hardware modifications are required**
  - cache coherence protocol augmented to detect violations
  - primary data cache is used to buffer speculative state

- **Compiler support is crucial yet feasible**
  - eliminating data dependences
  - aggressive scheduling to minimize critical path (forwarding)

- **Ongoing and future work**
  - refining the architecture (described in technical report)
  - building the compiler