Hardware Support for Reliability

Instructor: Josep Torrellas
CS533
Enhancing Software Reliability with Speculative Threads
Oplinger and Lam, ASPLOS 02

ReEnact: Using Thread-Level Speculation Support to Debug Data Races in Multithreaded Codes
Prvulovic and Torrellas, ISCA03

Detailed Design and Evaluation of Redundant Multithreading Alternatives
Mukherjee et al, ISCA02

ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors
Prvulovic et al, ISCA02
Oplinger et al: Contributions

• Use speculative threads to support:
  – Efficient fine-grain code monitoring
  – Fine-grain transactional programming

• Goal: Enhance software reliability
• Rationale: We have plenty of transistors; no need to devote them all to performance
Main code 1

Main code 2
(speculative)

Monitoring code of
main code 1
(non-speculative)

Main code 2 assumes monitoring code will return “success”

Question: What code speedups are possible?
Fine-Grain Execution Monitoring

• Current state of the art:
  – execution monitoring has large overheads
  – not run on production codes

• Proposed solution:
  – low overhead
  – if error caught, monitoring function communicates the failed check to the original program: return error code to the caller
  – hope to recover without terminating the program
  – typically: the speculative thread will not be rolled back
Fine-Grain Transactional Programming

All threads are speculative

All threads contain checking code at the end, which either aborts or okays the commit

Transactional model of execution
Fine-Grain Transactional Prgm

• Envision that programs in the future be built as a composition of transactions.
• Goal: error containment and recovery
• Current programming construct in Java:

```java
TRY {
    … the original code…
    if (error-detected())
        ABORT;
}

} CATCH{
    return an error code;
}

return OK;
}
```

If exception, go to CATCH, but side effects are not removed
Fine-Grain Transactional Prgm

• What we want:
  TRY <L1>;
  … the original code…
  if (error-detected())
      ABORT;
  COMMIT;
  return OK;
  L1: return an error code;

• If exception, side effects are removed
• architectural support needed to minimize overhead of transactions of finest grain
ReEnact: Using Thread-Level Speculation Support to Debug Data Races in Multithreaded Codes

Prvulovic and Torrellas, ISCA03
SM Provides Support to …

- **Buffer** state for squash or commit (caches)
- Maintain task **ordering** information (Task ID)
- **Monitor** communication across tasks to enforce ordering (cache coherence protocol)
- Cleanly **undo** side-effects of speculative task in a few cycles (flash-invalidate cache, restore regs)
Implementation Issues

• When entering the speculative thread:
  – Fence-type instruction that creates a checkpoint of the register state

• While executing thread speculatively:
  – Buffer all memory updates in the cache -- cannot update memory
  – Mark cache lines read and written
  – Monitor for errors or faults

• If an error or fault occurs:
  – Invalidate updated cache lines, reset marks, restore the register checkpoint

• Successful end of speculation:
  – Allow updated cache lines to be committed (displaced to memory)
ReEnact

Enhance Software Debugging:

Detect + Characterize + Correct software bugs in programs on-the-fly automatically in production codes
Conventional Debugging

Detection

Run

Yes

Seems OK?

No

Repair

Fix it

Figure it out

Figured it out?

No

Tweak and/or Instrument

Yes

Re-run

Analysis
What ReEnact Provides

- Cleanly undo group of tasks (buggy code section, hopefully)
- Re-execute those tasks only
- Re-execution of tasks is deterministic even under multithreading
- Bonus: detect bugs that appear as communication (e.g. Data Races)
Enhancing Debugging

Detection

Figure it out?

No

Re-run

Yes

Re-execution is deterministic

Tweak

Incremental undo and re-execution

Fix it?

No

Final incremental undo, possibly fix, and continue

Yes

Seems OK?

Run

Repair

Re-run

Fix it

Detect unsynchronized communication

Copyright Josep Torrellas 2003
Breaking Code into Chunks

Dynamic Instructions

ST X
ST Y
ADD
...
ST A
ST B
ST A
...
ST A
ST C

CPU

Cache

X Y
A B
A C

Memory
Breaking Code into Chunks

• New chunk begins ⇒ save CPU state
• Undo (squash) recent chunks if needed
  – Invalidate cache lines, restore saved CPU state
  – Enables re-execution for analysis and repair
• Commit old chunks
  – Allow displacement from cache, free saved CPU state
  – Makes room for buffering more recent chunks
  – Can not undo committed chunks
Undo Chunks

Dynamic Instructions

\[
\begin{align*}
\text{ST} & \ X \\
\text{ST} & \ Y \\
\text{ADD} & \\
\vdots & \\
\text{ST} & \ A \\
\text{ST} & \ B \\
\text{ST} & \ A \\
\vdots & \\
\text{ST} & \ A \\
\text{ST} & \ C \\
\end{align*}
\]

Analysis requires a rerun of these chunks
Undo Chunks

Dynamic Instructions

ST X
ST Y
ADD
...
ST A
ST B
ST A
...
ST A
ST C

CPU

Cache
X Y
A B
A C

Memory
Chunk Commit

Dynamic Instructions

ST X
ST Y
ADD
...
ST A
ST B
ST A
...
ST A
ST C

Need to displace X from this chunk
**Chunk Commit**

**Dynamic Instructions**

- `ST X`
- `ST Y`
- `ADD`
- `...`
- `ST A`
- `ST B`
- `ST A`
- `...`
- `ST A`
- `ST C`
Ordering Chunks

Chunk ■ “happens before” chunk ■
Deterministic Re-execution

• Cross-thread communication $\Rightarrow$ chunk order
  – Order chunks on first communication
  – Enforce order on subsequent communication (may squash)
• Entire chunk ordered before or after another
• Deterministic re-execution $\Leftrightarrow$ repeat chunk order
Chunk Ordering by Synchronization
Data Race Detection

• Detect communication between…
  - Ordered chunks: synchronized access
  - Unordered chunks: data race detected!
Example: Missing Critical Section

<table>
<thead>
<tr>
<th>Thread X</th>
<th>Thread Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(L)</td>
<td></td>
</tr>
<tr>
<td>LD A</td>
<td>LD A</td>
</tr>
<tr>
<td>ST A</td>
<td>ST A</td>
</tr>
<tr>
<td>unlock(L)</td>
<td></td>
</tr>
</tbody>
</table>
Detection: Data Race

Thread X

...  
lock(L)

LD A

ST A

unlock(L)

Thread Y

CPU

Cache

LD A

CPU

Cache

Memory

No order between ■ and ■

Copyright Josep Torrellas 2003
Analysis: Refine Race Signature

Thread X  Thread Y

1. Rollback

2. Put a watchpoint on accesses to data address A

3. Re-execute assuming order: ■ after ■
Found Race Signature

Thread X

LD A
ST A

Thread Y

LD A
ST A
Repair: Pattern Matching

• Analysis resulted in a detailed signature
  – Instruction & data addresses, data values, timing, etc.

• Pattern-match it with a library of common races:
  – Suggest repair to programmer, or
  – Download bug-specific patch, or
  – Try to automatically re-introduce missing ordering

• Squash chunks, re-execute with corrections
ReEnact Pros

• On the fly – debug each problem as it is found
  – Low-latency re-execution of surrounding code
  – Low-latency detection of bug symptoms
• Always on – even in production code
  – Low overhead in bug-free execution
• Debug multi-threaded code
  – Must address non-determinism of parallel execution
ReEnact Evaluation

• Low overhead in error-free execution: 6% avg
• Highly effective: Detect, Analyze & Correct(?) SW bugs
  – Existing bugs
    • Synchronization through plain variables
    • Other existing data races
  – Induced bugs
    • Remove critical section
    • Remove barrier
## How Good ReEnact is to:

<table>
<thead>
<tr>
<th></th>
<th>Detect</th>
<th>Rollback</th>
<th>Analyze</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync through plain variables</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Other Existing Data Races</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>No</td>
</tr>
<tr>
<td>Induced Bugs: Removed Lock</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Induced Bugs: Removed Barrier</td>
<td>✓</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>

- Splash-2 benchmarks on 4-proc CMP
Conclusions

• Speculative multithreading boosts software productivity
  – Enhances Software Debugging (detection, analysis, correction)
• Exciting area: cost effective use of transistors for reliability or debuggability