Effective and Inexpensive (Memory) Race Recording

Min Xu
Thesis Defense
05/04/2006

Electrical and Computer Engineering Department, UW-Madison
Advisors: Mark Hill, Rastislav Bodik
Committee: Remzi Arpaci-Dusseau, Mikko Lipasti, Barton Miller, David Wood

Overview

Increasingly useful to replay multithreaded code
  • Race recording: key to dealing with nondeterminism

A Case Study
  • Long recording: 1 byte/kilo-instr
  • Always-on recording: less than 2% overhead
  • Low cost: 24 KB RAM/core
  • Support both SC & TSO (x86-like)
Thesis Contributions

- Small Log Size
- Effective
- SC & TSO Applicability
- Low Runtime Overhead
- Inexpensive
- Low Cost Hardware
- RTR Algorithm
- Coherence Piggyback
- Order-Value Hybrid
- Set/LRU Approximation

Outline

1. Motivation & Problem: 5 slides
2. An Effective and Inexpensive Race Recorder: 21 slides
   - RTR Algorithm
   - Coherence Piggyback
   - Set/LRU Approximation
   - Order-Value Hybrid
3. Evaluation Method & Results: 6 slides
4. Conclusion & My Other Research: 3 slides
## Motivation & Problem

## Multithreaded Debugging

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gcc hash.c</code>&lt;br&gt;<code>a.out</code>&lt;br&gt;segmentation fault&lt;br&gt;%</td>
<td><code>% gdb a.out&lt;br&gt;gdb: run&lt;br&gt;Program received SIGSEGV.&lt;br&gt;In get() at hash.c:43&lt;br&gt;43 a = bucket-&gt;d;</code></td>
</tr>
<tr>
<td><code>gcc para-hash.c</code>&lt;br&gt;<code>a.out</code>&lt;br&gt;segmentation fault&lt;br&gt;%</td>
<td><code>% gdb a.out&lt;br&gt;gdb: run&lt;br&gt;Program exited normally&lt;br&gt;gdb:</code></td>
</tr>
<tr>
<td><code>gcc para-hash.c</code>&lt;br&gt;<code>a.out</code>&lt;br&gt;segmentation fault&lt;br&gt;<code>race recorded in &quot;log&quot;</code>&lt;br&gt;%</td>
<td><code>% gdb a.out&lt;br&gt;gdb: run&lt;br&gt;Program received SIGSEGV.&lt;br&gt;In get() at para-hash.c:67&lt;br&gt;67 a = bucket-&gt;d;</code></td>
</tr>
</tbody>
</table>
Race Recording

Recording for Multithreaded Replay

- Race Recording
  - Not-an-issue for a single thread
  - Create the same general & data races

- Checkpointing
  - Provide a snapshot of the program state
  - Many proposals (e.g., SafetyNet), not focus

- Input Recording
  - Provide repeatable inputs
  - Some proposals (e.g., part of FDR), not focus
A Good Race Recorder

- Low cost
- Low runtime overhead
- Applicability

Desired & Existing Race Recorders

<table>
<thead>
<tr>
<th>Desired Recorder</th>
<th>Recording Length</th>
<th>Applicability</th>
<th>Overhead</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>MP Racey Code SC TSO</td>
<td>Negligible Slowdown</td>
<td>Little Hardware</td>
</tr>
<tr>
<td>EnstRply ’87</td>
<td>×</td>
<td>✓ ✓ ×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>R&amp;C ’90</td>
<td>×</td>
<td>× ✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bacon’91</td>
<td>×</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Netzer’93</td>
<td>✓</td>
<td>✓ ✓ ×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Déjà Vu ’98</td>
<td>✓</td>
<td>× ✓ ×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RecPlay’00</td>
<td>✓</td>
<td>✓ × ×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>JoRec’04</td>
<td>✓</td>
<td>✓ × ×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Our Recorder</strong></td>
<td>✓</td>
<td>✓ ✓ ✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Small Log Size

RTR Algorithm

Order-Value Hybrid

Set/LRU Approximation

Coherence Piggyback

**Problem Formulation**

Thread 1
- Conflicts (red)
  - Id A
  - st B
  - st C
  - Id D
  - sub
  - Id B

Recording

Log

Thread 2
- Dependence (black)
  - Id A
  - st B
  - st C
  - Id D
  - sub
  - Id B

Replay

Reproduce exact same conflicts: no more, no less
Log All Conflicts

Thread I
1. ld A
2. st B
3. st C
4. ld D
5. sub
6. ld B

Thread J
1. add
2. st C
3. ld B
4. st A
5. st C
6. st D

Dependency Log
Log J: 2 → 3
1 → 4
3 → 5
4 → 6
Log I: 2 → 3

Log Size: 5*16=80 bytes
(10 integers)

But too many conflicts

Netzer's Transitive Reduction

Thread I
1. ld A
2. st B
3. st C
4. ld D
5. sub
6. ld B

Thread J
1. TR reduced
2. st C
3. ld B
4. st A
5. st C
6. st D

TR Reduced Log
Log I: 2 → 3
3 → 5
4 → 6
Log J: 2 → 3

Log Size: 64 bytes
(8 integers)
The Intuition of the New RTR Algorithm

After Reduction

From I to J

Vectors

Regulate Replay (RTR)

Vectors

Stricter Dependencies to Aid Vectorization

New Reduced Log

Log I: 2→3

Log J: 2→3

Log Size: 48 bytes
(6 integers)
Compress Vectorized Dependencies

Vectorized Log
- Log J: x=3,5, Δ=1
- Log I: x=3, Δ=1

Log Size: 40 bytes (5 integers)

Reduce log size to KB/core/second

Low Runtime Overhead
Detect Conflicts

A.readers.add(I, 1)
B.writer = (I, 2)

if (C.writer != I)
log(WAW)

foreach C.readers
if (reader != I)
log(WAR)
C.readers.clear()
C.writer = (I, 3)

B.writer = (I, 2)
C.writer = (J, 2)

Recording

Expensive in software

Use Cache and Cache Coherence

Proc I
Tag State Data Timestamp
A S ... 1
B M ... 4

Proc J
Tag State Data Timestamp
A S ... 3
B I ... 2

A.readers B.reader
A.writer B.writer

Get/S Request

RAW Detected & Logged

Detect conflict in hardware with little runtime cost
Cache Evictions and Writebacks

$$
\begin{array}{c|c|c|c|c}
\text{Tag} & \text{State} & \text{Data} & \text{Timestamp} \\
C & M & ... & 3 \\
B & M & ... & 4 \\
\end{array}
$$

$$
\begin{array}{c|c|c|c|c}
\text{Tag} & \text{State} & \text{Data} & \text{Timestamp} \\
A & M & ... & 4 \\
B & I & ... & 2 \\
\end{array}
$$

Directory of A: Shared(I, J) Owner()

OK with nonsilent eviction & directory eviction

Implement TR and RTR in Hardware

Ideal TR requires vector timestamps
- Too expensive
- New idea: Pairwise-TR (use scalar timestamp)
- Enable pairwise transitive reduction

Optimal RTR algorithm is likely expensive
- Implement a greedy RTR algorithm
- One-pass, online algorithm
- Keep a sliding window of vectorizable dependencies
## Hardware Implementation

<table>
<thead>
<tr>
<th>Cache</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Eviction/writeback</td>
<td>Solved, more details later</td>
</tr>
<tr>
<td>Directory protocols</td>
<td>Solved</td>
</tr>
<tr>
<td>Snooping protocols</td>
<td>Partly solved</td>
</tr>
<tr>
<td>Two-level coherence</td>
<td>Not yet solved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-order/Prefetching</td>
<td>Solved</td>
</tr>
<tr>
<td>Unordered message</td>
<td>Solved</td>
</tr>
<tr>
<td>Counter overflow</td>
<td>Solved</td>
</tr>
<tr>
<td>Thread Migration</td>
<td>Not yet solved</td>
</tr>
</tbody>
</table>

![Diagram](image.png)
## Timestamp Approximation

### One Set of I’s $^*$

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>M</td>
<td>...</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>M</td>
<td>...</td>
<td>2</td>
</tr>
</tbody>
</table>

Use current IC of thread I

Directory of A: **Shared(I)**

**Correct, but more evictions $\rightarrow$ more logged conflicts**
Set/LRU Approximation

One Set of I's $\xi$

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>M</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>M</td>
<td>...</td>
<td>2</td>
</tr>
</tbody>
</table>

LRU guarantee B's TS > A's TS

Set/LRU better preserve reducibility
Small $\xi$ → more misses → but still small log

Hardware Cost of Timestamps

Coupled Timestamp Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>M</td>
<td>...</td>
<td>2</td>
</tr>
</tbody>
</table>

Coupled timestamp memory: overhead $\propto$ cache size
- Not flexible
- 64B line + 64b (24b) timestamp $\rightarrow$ 12.5% (4.7%) overhead
- 192 KB for a 4MB L2

Need to modify cache
Decoupled Timestamp Memory

Coupled Timestamp Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>...</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>M</td>
<td>...</td>
<td>2</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>State</th>
<th>Data</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>M</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Timestamp Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
</tbody>
</table>

Decoupling $\Rightarrow$ Small timestamp memory (Set/LRU)

- e.g., 32-set, 64-way $\rightarrow$ 99% transitive reduction
- Timestamps Memory $\rightarrow$ 24 KB

From 192 KB to 24 KB: 8x reduction

SC & TSO Applicability
**Recording with Total Store Order (TSO)**

Majority of existing MP are non-SC

TSO is well defined, x86-like

<table>
<thead>
<tr>
<th>SC</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**TSO Execution**
Order-Value-Hybrid Recording

Thread 1
1. st A,1
2. ld B

Recording
1. st A,1
2. ld B

Value Used
A=0

Value Logged
A=1

WAR
Omitted
B=0

Replay
1. st A,1
2. ld B

Story Syntax
A=0
B=0

A Changed!

Hybrid Recording with TR and RTR

Hybrid recording
- All loads get correct values
- Hardware similar to OoO SC [Gharachorloo et al.’91]

Hybrid + TR & RTR
- TR will not use the omitted WAR in reduction
- RTR vectorize dependencies more conservatively
Evaluation Method & Results

Put-it-together: Determinizer/CMP
Simulation Method

Commercial server hardware
- GEMS: http://www.cs.wisc.edu/gems
- Full-system (OS + application) executions
- 4-core CMP (Sequential Consistent)
  - 1-way in-order issue, 2 GHz,
  - 64KB I/D L1, 4MB L2, 64byte lines, MOSI directory

Commercial server software
- Apache - static web serving
- SpecJBB - middleware
- OLTP - TPC-C like
- Zeus - static web serving

Log Size: 1 byte/kilo-instr

Well within in the capability of current machines
- Long recording (days – months) need improvement
Runtime Overhead

Execution Time

Interconnection Msg. B/W

Apache JBB OLTP Zeus

Baseline

With race recorder

Our recorder can be “always-on”

Benefits of RTR and Set/LRU (Log Size)

Imprvoement by RTR

Effectiveness of Set/LRU

Apache JBB OLTP Zeus AVG

Pairwise-TR

Our RTR

Perfect TSM

24KB Set/LRU TSM
Why RTR and Set/LRU Work Well?

**RTR**
- Processors execute instructions at similar speed
- Therefore, we can find “vectorizable” dependencies

**Set/LRU**
- Temporal locality makes the LRU timestamps old
- We only need to know if a timestamp is “old-enough”

Sensitivity and Scalability

**A design space of the timestamp memory (TSM)**
- **Size**: smaller TSM -> larger log
- **Read/write timestamp**: should be used when TSM is large
- **Partial timestamp**: 24-bit enough
- **Associativity**: higher better for RTR

**Scalability of the recorder**
- Studied with modest processors (2p - 16p)
- Commercial workloads, not scientific workloads
- Log size increase slowly with number of cores
Conclusion & My Other Research

Race Recording

Race recording → Key to combat nondeterminism

My thesis → An effective & inexpensive Recorder
  • RTR algorithm → small log size
  • Coherence piggyback → Negligible slowdown
  • Timestamp approximation → Low hardware cost
  • Order-value hybrid → support SC & TSO

Future work
  • Improve race recording algorithm
  • Improve race recorder implementation
  • Study race replay
Serializability Violation Detector [PLDI'05]

Like a race detector

No *a priori* annotation requirement

- "critical sections" are inferred

Intend to detect bugs “actually” happen

- Check for a 2-Phase-Locking condition

A “Critical Section”

Publications

**FDR (ISCA’03)**

- Adopted by UCSD BugNet (ISCA’05)

**SVD (PLDI’05)**

- Cited by Vaziri *et al.* (POPL’06)
  - Influenced new data race definition

**RTR, Set/LRU & Hybrid**

- Submitted for publication
Thank you!

Acknowledgements

Joint work with my advisors

- Mark Hill, Ras Bodik

Ph.D. Committee

- David Wood, Mikko Lipasti, Remzi Arpaci-Dusseau, Barton Miller

Multifacet Group

- Milo Martin, Dan Sorin, Carl Mauer, Brad Beckmann, Kevin Moore, Alaa Alameldeen, Mike Marty, Luke Yen

Affiliates & Companies

- Joe Emer, CJ Newburn, Peter Hsu, Bob Zak, Eric Bach, Gang Luo, Alex Chow, IBM, Intel, Microsoft, Sun
Deterministic Replay is Useful

Deterministic Replay is logically recreating a program execution

Present applications
- Cyclic Debugging ([Pancake & Netzer ‘93])
- Fault Tolerance (ExtraVirt [Lucchetti et al. ’05])
- Intrusion Analysis (ReVirt [Dunlap et al. ’02])

Future applications
- Data Recovery
- Replay-based Synchronization

Multicore and Multithreading

Multicore is common
- AMD X2
- IBM Power 5/6, Cell
- Intel Pentium D, Core Duo
- Sun SPARC T1

Multithreading is common
- Server: high throughput
- Scientific: high performance
- Desktop/embedded: low response time
Race Recording: Key to Determinism

Races: general race & data race [Netzer & Miller]
- Both cause nondeterminism
- Race recording can help, but

Existing race recorders are inadequate
- Some generate large logs
- Some have high runtime overhead
- Some have high hardware cost (space overhead)
- Support only sequential consistency

Need a better race recorder

Recording/Replay & Debugging

**Online Recorder**
- P1, P2, P3, P4
- Store log A, Store log B, Store log C
- Checkpoint A, Checkpoint B, Checkpoint C
- Crash
- Dump “Core”

**Deterministic Replayer**
- Replaying from log B, C
- Crash
- Read Checkpoint B
Deterministic Replay & Fault Tolerance

Fault Recovery
  • Replay after a failure

Fault Detection
  • Replay then compare

(Courtesy of VMware)

Future: Record/Replay & Undo/Redo

VM as a software platform
  • Ease software development
  • Fine granularity in Undo and Redo
Future: Replay-based Synchronization

Three steps
- Coarse-grain sync. → fine-grain sync. → hardware sync.

Results: higher performance
- Works only if static control flow & fixed data addr
  - DSP kernels

Race Recording Related Work

<table>
<thead>
<tr>
<th>Total-order recorders</th>
<th>Partial-order recorders</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bacon ’91</strong> (Hardware)</td>
<td><strong>Bacon ’91</strong> (Hardware)</td>
</tr>
<tr>
<td>RecPlay ‘00</td>
<td>Instant Replay ’87</td>
</tr>
<tr>
<td>JaRec ‘04</td>
<td>Netzer ’93</td>
</tr>
<tr>
<td>RAC ‘90</td>
<td>Bus transaction groups</td>
</tr>
<tr>
<td>Déjà Vu ’98</td>
<td>Variable version</td>
</tr>
<tr>
<td>Lamport Clocks</td>
<td>Vector clocks</td>
</tr>
<tr>
<td>Scheduling</td>
<td></td>
</tr>
<tr>
<td>Bus transactions</td>
<td></td>
</tr>
<tr>
<td>Large log</td>
<td>Large log</td>
</tr>
<tr>
<td>Small log</td>
<td>Small log</td>
</tr>
<tr>
<td>Low overhead</td>
<td>Low overhead</td>
</tr>
<tr>
<td>Low overhead (sync only)</td>
<td>Low overhead</td>
</tr>
<tr>
<td>Low overhead (non-MP)</td>
<td>High overhead</td>
</tr>
<tr>
<td>Low replay parallelism</td>
<td>High replay parallelism</td>
</tr>
</tbody>
</table>
Correctness of Order-Value-Hybrid

Removing WAR dependencies
• Say thread I read, thread J write
• Removing the WAR affects I’s read, not J’s write
• But, for every dependence removed, thread I reads correct value from the value log
• Therefore, all reads get the correct value

TR and TSO

TR affects dependencies reduced by a WAR
• The WAR itself may later be removed during replay
• Solution: Not use WAR in TR if the WAR can be removed
• Respond with a special flag when a loaded cache line is stolen

Must not be reduced
RTR and TSO

The sliding window may expose the ordered loads
  • Shrink the sliding window to avoid it

Deadlock Avoidance of RTR

Avoid deadlock by adhere to a SC total order
Recording Race-free Executions

No data races

Only need to record synchronization race

Deterministic replay up until the first data race

Replay Parallelism

Replay performance depends on

(1) Number of synchronizations
(2) Extra wait incurred by the synchronizations
Directory Protocols

Add sticky states in the directory
- Retain states after writebacks
- Need extra acknowledgements

Or, add extra timestamp memory in the directory
- Helps to avoid extra acknowledgements

A tradeoff
- Sticky states can be cheaper
- But extra timestamp memory can be faster

Snooping Protocols

Key problem is combined/implicit response
- Not a problem for AMD Hammer

![Diagram showing Proc 1 and Proc J with tags, states, data, and timestamps]
Nonsilent Evictions

Directory eviction: more false conflict, like snooping

Out-of-Order & Hardware Prefetching

Speculative execution
- No IC assigned yet

Hardware prefetching
- No IC assigned

Key idea: receive observation
- Can associate a ld/st with current commit instruction
Unordered Messages in Interconnect

Message arrive out-of-order

Can affect reduction

But better add a sequence number
  • Reconstruct the message order
  • Enable IC compression by sending deltas

Integer Overflow

IC and timestamps may overflow

IC: make it 64bit, will not overflow for a long time

Timestamps: use approximation techniques
  • MSB of IC + LSB of Timestamps
Varying TSM Size

Varying Associativity
Varying Partial Timestamp Width

Log Size Scaling
In Retrospect ...

What are you most proud of?
- RTR improves TR after 13 years

What would you do differently if doing it again?
- “replaying me is deterministic” (just kidding)
- I wish I focused on race recording earlier

What the industry should do?
- Implement the recorder as a VMM extension