Online Reduction of Shared Memory Dependences

Yanyan Jiang, Chang Xu, Xiaoxing Ma, Jian Lu (Nanjing University, China)
Du Li (Carnegie Mellon University, USA)

November 17 @ FSE 2016
Outline

• Background
  • shared memory dependence and its applications

• Motivation and problem formulation
  • online reduction of shared memory dependences

• Solution
  • the “bisectional coordination” protocol
Background
The Programming World is Becoming Concurrent
Concurrent Programs: Hard to Test and to Debug

• Shared memory is the major source of non-determinism
  • atomicity/order violation is the major cause of non-deadlock concurrency bugs [Lu12]

```c
if (thd->proc_info) {
  thd->proc_info = NULL;
  fputs(thd->proc_info, …);
}
```

MySQL `ha_innodb.cc`
Addressing the Challenges: Dynamic Analyses

- **Detection**
  - Predictive Analysis [Flanagan09]

- **Diagnosis**
  - Time-travel Debugging
    - [Dunlap08, Park09, Altker09, Huang13]

- **Avoidance**
  - STM / DMT / Schedule Avoidance
    - [Dice07, Devietti09]

- **Testing Run**
- **Production Run**
How Do We Build Dynamic Analyses?

• “Dynamic analysis operates by executing a program and observing the executions” [Ernst03]

• How to observe a concurrent program’s execution?
  -- observe the order of shared memory accesses!
Shared Memory Dependence: Definition

- **Two shared memory accesses** executed by **different threads** that have **data dependence**: read-after-write (RAW), write-after-write (WAW), and write-after-read (WAR). [Bond13]
Shared Memory Dependence: Example

Initially, $x = y = 0$

Thread 1

Write($x$) = 1

Write($y$) = 1

Read($y$) = 1

Thread 2

Read($x$) = 1

Write($y$) = 2

Read($y$) = 2

$R \leftarrow W$

$W \leftarrow R$
Application: Record and Replay

- Reproduce a past concurrent program execution

Thread 1

1. Write(x) = 1
2. Read(x) = 1
3. Write(y) = 1
4. Read(y) = 1

Thread 2

5. Write(y) = 2
6. Read(y) = 2

Re-execute the program w.r.t the shared memory dependences
Application: Data Race Detection

- Data race: two accesses that can simultaneously happen and at least one is a write

```
Thread 1

Lock(l)
↓
Write(x)
↓
Unlock(l)
```

```
Thread 2

Lock(l)
↓
Read(x)
↓
Unlock(l)
```

Happens-before race detector: check each shared memory dependence (a → b) against →
Application: Transactional Memory

- Ensure serializability of atomic regions

Thread 1

```
TX_BEGIN
Write(y)
Read(z)
TX_END
```

Thread 2

```
TX_BEGIN
Write(y)
```

Abort

Transactional memory: shared memory dependences indicate conflicts
Capturing Shared Memory Dependences: The Basic Idea

• Synchronize shared memory accesses with locks

Optimizations: allow concurrent readers and have an $O(1)$ wait-free fast path [Bond13, Jiang15]
Motivation and Problem Formulation
Motivation

• Shared memory dependences support dynamic analyses
  • record and replay, data race detection, transactional memory, etc.

• Not only the overhead but also the amount of shared memory dependences impact the analyses
Motivation (cont’d)

• Less dependences, more efficient analysis

<table>
<thead>
<tr>
<th>Application</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver-based record and replay</td>
<td>smaller constraint formula to solve</td>
</tr>
<tr>
<td>Data race detection</td>
<td>less checks (clock/epoch comparisons) to performed</td>
</tr>
<tr>
<td>Software transactional memory</td>
<td>less conflict detection</td>
</tr>
</tbody>
</table>

• Can we also reduce shared memory dependences along with the program execution?
Transitive Reduction (TR) of Shared Memory Dependences

- $a \rightarrow b \land b \rightarrow c$ implies $a \rightarrow c$ [Netzer93]
Regular Transitive Reduction (RTR) of Shared Memory Dependences

• Replace parallel dependences by a stricter one \cite{xu06}
The Challenge

• Both TR and RTR require tracking of transitivity
  • only practical with hardware support, $\Omega(T)$ lower bound

• We want to reduce shared memory dependences, but how to make it efficient?
  • online software-only reduction of shared memory dependences
The Bisectional Coordination Protocol
Ingredient 1: Group Variables

• Shared memory dependences can be traced in terms of “variable groups”
  • make variables share a same lock and metadata
  • existing work already does this (group variables in a cache-line, an object, etc.)

• Good grouping yields reduced dependences
Variable Grouping as Transitive Reduction

• Grouping $x$, $y$, and $z$

Thread-local access, no inter-thread data dependence
Ingredient 2: Spatial Locality

• Concurrent programs have spatial locality
  • consecutive variable accesses are usually near in address
    → group nearby variables together!

• An empirical study supports this claim
The Basic Idea:
Grouping is not Need to be Static!

• Maintain a **dynamic** address space’s **interval partition**
  • starting from a coarse (optimistic) grouping that assumes the memory is not shared at all

\[
[l_1 = 0, r_1) \quad [l_2, r_2) \quad [l_3, r_3 = M)
\]

• **Adaptively refine** a partition if does not reflect the locality of shared memory accesses
When an Optimistic Grouping Goes Wrong?

- There can be unnecessary “false dependences”
  - when false dependences accumulate, the group should be refined

```
Thread 1
Write({x,y,z})
Write({x,y,z})
Write({x,y,z})

Thread 2
Read({x,y,z})

False dependence
T1: {x,y} → T2: {z}
```
How to Refine a Partition?

- Bisect a group into two equal halves – bisectional coordination

\[ m = \frac{l + r}{2} \]
Technical Issues

• Why bisection?
  • simple and straightforward

• How to detect false dependences?
  • approximate detection by bloom filters

• How to deal with fragmented groups?
  • reset to the initial partition
Evaluation Results
Evaluation Setup

• **12 benchmarks from three categories**
  - desktop: aget, pfscan, pbzip2
  - scientific: ocean, water, fft, radix, fluid, qsort, x264
  - server: knot, apache

• **Workloads and settings**
  - 16 worker threads, large workloads
  - evaluated on a 24-core Xeon server (Ubuntu Linux)
# Evaluation: Reduction Effectiveness

<table>
<thead>
<tr>
<th>Category</th>
<th>Benchmark</th>
<th># Dep.</th>
<th>RWTrace (64B)</th>
<th>LEAP (64B)</th>
<th># Bisect</th>
<th># Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Desktop</strong></td>
<td>aget</td>
<td>7.40K</td>
<td>-9%</td>
<td>-19%</td>
<td>0</td>
<td>39.9K</td>
</tr>
<tr>
<td></td>
<td>pfscan</td>
<td>116K</td>
<td>-34%</td>
<td>-99%</td>
<td>12</td>
<td>9.82G</td>
</tr>
<tr>
<td></td>
<td>pbzip2</td>
<td>0.30K</td>
<td>-55%</td>
<td>-76%</td>
<td>28</td>
<td>5.21K</td>
</tr>
<tr>
<td></td>
<td>ocean</td>
<td>27.1K</td>
<td>-5.2%</td>
<td>-99%</td>
<td>60</td>
<td>138M</td>
</tr>
<tr>
<td></td>
<td>water</td>
<td>53.8K</td>
<td>-97%</td>
<td>-99%</td>
<td>52</td>
<td>112M</td>
</tr>
<tr>
<td><strong>Scientific</strong></td>
<td>fft</td>
<td>0.23K</td>
<td>-90%</td>
<td>-99%</td>
<td>4</td>
<td>40.0M</td>
</tr>
<tr>
<td></td>
<td>radix</td>
<td>0.16K</td>
<td>-93%</td>
<td>-99%</td>
<td>34</td>
<td>112M</td>
</tr>
<tr>
<td></td>
<td>fluid</td>
<td>9.52K</td>
<td>-39%</td>
<td>-99%</td>
<td>16</td>
<td>463M</td>
</tr>
<tr>
<td></td>
<td>qsort</td>
<td>319K</td>
<td>-55%</td>
<td>-79%</td>
<td>72</td>
<td>15.3M</td>
</tr>
<tr>
<td></td>
<td>x264</td>
<td>1.63M</td>
<td>-91%</td>
<td>-98%</td>
<td>954</td>
<td>6.80G</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td>knot</td>
<td>37.6K</td>
<td>-0.5%</td>
<td>-45%</td>
<td>18</td>
<td>159K</td>
</tr>
<tr>
<td></td>
<td>apache</td>
<td>44.2K</td>
<td>-79%</td>
<td>-98%</td>
<td>89</td>
<td>6.64M</td>
</tr>
</tbody>
</table>
Evaluation: Reduction Overhead

• Bisectional coordination: paying 0—54.7% (median 21%) overhead over RWTrace to achieve up to 97% shared memory dependence reduction
Summary
Online Shared Memory Dependence Reduction via Bisectional Coordination

• The first adaptive variable grouping algorithm of capturing shared memory dependences

\[ m = \frac{l + r}{2} \]
Bisectional Coordination

- Demonstrates a possibility to build more efficient analyses
  - deterministic replay, data race detection, and false sharing detection are discussed in the paper

- Opens a new direction: dynamic variable grouping
  - how to efficiently implement non-consecutive grouping?
  - can we merge split groups?
  - …
Thank You!