Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection

Santosh Nagarakatte, Sebastian Burckhardt, Milo Martin, Madan Musuvathi

University of Pennsylvania
Microsoft Research
This work licensed under the Creative Commons Attribution-Share Alike 3.0 United States License

• You are free:
  • to Share — to copy, distribute, display, and perform the work
  • to Remix — to make derivative works

• Under the following conditions:
  • Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
  • Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

• For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to: http://creativecommons.org/licenses/by-sa/3.0/us/

• Any of the above conditions can be waived if you get permission from the copyright holder.
• Apart from the remix rights granted under this license, nothing in this license impairs or restricts the author's moral rights.
Overview

• **Context: Testing multithreaded programs**
  • Many policies proposed to explore interleavings

• **NeedlePoint: Interleaving Exploration Tool**
  • Test and evaluate real world code with policies
  • Separate mechanisms from policy, infers blocking

• **Probabilistic Concurrency Testing (PCT) works well**
  • Detected bugs at least once in 1000 executions
  • Unfortunately, it runs one thread at a time
    • Serious bottleneck with real code

• **Parallel Probabilistic Concurrency Testing (PPCT)**
  • Runs multiple threads
  • Effective as PCT in theory and practice
Testing Multithreaded Programs

• Two aspects of the testing problem
  • Detection of bugs
    • Use a test harness
    • Use pattern detectors like data race detectors
  • Driving & exploring the interleaving
    • Explore all scheduling choices at each point

*Focus of this talk*
Space of Interleavings

(p = 0, bal = 10, q = 0)

T1: \( w(y) \)
- \( q = 7 \)
- \( t = \text{bal}; \)
- \( \text{bal} = t - y \)

T2: \( d(x) \)
- \( \text{bal} += x \)
- \( p = 5 \)

(p = 5, bal = 1000, q = 7)
Space of Interleavings

\[(p = 0, \text{bal} = 10, q = 0)\]

T1: \(w(y)\)
- \(q = 7\)
- \(t = \text{bal};\)
- \(\text{bal} = t - y\)

T2: \(d(x)\)
- \(\text{bal} += x\)
- \(p = 5\)

\[(p = 5, \text{bal} = 1000, q = 7)\]
Space of Interleavings

\((p = 0, \text{bal} = 10, q = 0)\)

\[T1: \text{w(y)}\]
\[
q = 7 \\
t = \text{bal}; \\
\text{bal} = t - y
\]

\[T2: \text{d(x)}\]
\[
\text{bal} += x \\
p = 5
\]

\((p = 5, \text{bal} = 1000, q = 7)\)
Space of Interleavings

(p = 0, bal = 10, q = 0)

Numerous interleavings in the same program

(p = 5, bal = 1000, q = 7)

T1: w(y)
- q = 7
- t = bal;
- bal = t - y

T2: d(x)
- bal += x
- p = 5
Interleaving Exploration Policies

- Best-effort exploration
  - Random sleep: actively introduce delays
  - AtomFuzzer: delays to trigger atomicity violations
  - PreemptionAlways: a preemption at every point

- Exploration with coverage guarantees
  - Preemption bounding: introduce bounded number of preemptions
  - PCT: schedule threads according to priorities
Test and understand the interleaving exploration schemes with real world code
NEEDLEPOINT
NeedlePoint Framework

• Unified framework for controlled scheduling
  • Separation of concerns between mechanisms & policy

  Modeling each synchronization is error-prone & real programs define ad-hoc synchronization

• Three mechanisms for controlled scheduling
  1. Introduction of callbacks to the scheduler
     • Real programs use a wide range of synchronization
  2. Support for identifying blocked threads
     • To provide coverage, need to explore the interleaving when the synchronization blocks
  3. Support for starvation freedom
     • Spin loops & busy wait synchronization $\rightarrow$ livelocks
Calls to the NeedlePoint Framework at Schedule Points

<table>
<thead>
<tr>
<th>TID</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>2</td>
<td>Ready</td>
</tr>
</tbody>
</table>

Synchronization library calls, atomic operations, shared memory accesses

```
Lock (l);
bal += x;
Unlock(l);

Lock (l);
t = bal;
Unlock(l);

while(true){
schedulingPolicy();
if(state[tid] == Running)
break;
else
yield();
}
```
Inferring Blocked Threads

How to identify which threads block

Infers whether the thread is blocked

<table>
<thead>
<tr>
<th>TID</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>2</td>
<td>Ready</td>
</tr>
</tbody>
</table>

```java
while(true){
    schedulingPolicy();
    if(state[tid] == Running)
        break;
    else{
        yield(); timeout[tid] ++;
        if(timeout[tid] > T)
            state[rid] = BLOCKED
    }
    reset_timeout_counters();

    Lock (l);
    bal += x;
    Unlock(l);

    Lock (l);
    t = bal;
    Unlock(l);
    ....
```
Inferring Blocked Threads

<table>
<thead>
<tr>
<th>TID</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running</td>
</tr>
<tr>
<td>2</td>
<td>Blocked</td>
</tr>
</tbody>
</table>

Allows the lock to execute, acquire operation blocks

```
while(true){
    schedulingPolicy();
    if(state[tid] == Running)
        break;
    else{
        yield(); timeout[tid] ++;
        if(timeout[tid] > T)
            state[rtid] = BLOCKED
    }
    reset_timeout_counters();
}
```

Lock (l);
bal = x;
Unlock(l);

---

```
while(true){
    schedulingPolicy();
    if(state[tid] == Running)
        break;
    else{
        yield(); timeout[tid] ++;
        if(timeout[tid] > T)
            state[rtid] = BLOCKED
    }
    reset_timeout_counters();
}
```

```
Lock (l);
t = bal;
Unlock(l);
....
```
Evaluation of Techniques

- Random Sleep
- Preemption Always
- Preemption Bounding
- AtomFuzzer
- Probabilistic Concurrency Testing

PCT detects the bug at least once in 1000 executions
Challenges with PCT

- PCT was effective in detecting all bugs
- However, one thread at a time is a bottleneck
  - Can’t exploit multi-core
  - Interacts poorly with NeedlePoint’s blocking inference
  - Result: Unable to test highly parallel applications
    - More than 64X slowdown with Chrome browser

Can we run multiple threads with PCT and maintain its effectiveness in theory & practice?
PCT BACKGROUND
Probabilistic Concurrency Testing (PCT) [ASPLOS 2010]

• Key Idea: Make a few choices to assign priorities
  • Schedule the highest priority thread that is not blocked

• Bug depth
  • Number of ordering constraints an interleaving need to exhibit to trigger the bug
  • More constraints means more things have to go “just right” to trigger the bug.

PCT explores the program to trigger bugs of particular depth
Bug Depth Illustration

Bug of depth 1: Ordering Violation

...  
p = &a;  
...  

...  
p->f ++;  
...  

Bug of depth 2: Atomicity Violation

Lock(I)  
t = bal  
Unlock(I)  
Lock(I)  
bal += x;  
Unlock(I)  

Lock(I)  
bal = t - y  
Unlock(I);  

Lock(I)  
Unlock(I)
PCT Algorithm for Bug Depth $d$

- At the beginning of the execution
  - Assign unique priorities to the threads
  - Reserve $d$ lower priorities
  - Choose $d-1$ priority change points in the execution

- Schedule the threads according to priorities
  - Run the highest priority non-blocked thread
  - At a change point, lower the priorities to the reserved priorities
PCT for a Bug of Depth 2

Change Point: Step#2
(p = 0, bal = 10, q = 0)

Pri = 3
T1: w(y)
q = 7
t = bal;
bal = t - y

Pri = 2
T2: d (x)
bal += x
p = 5

Change Point: Priority of deposit thread changed to 1

(p = 5, bal = 1000, q = 7)
The PCT Probabilistic Bounds

- Given a program with
  - $n$ threads ($\sim$tens)
  - $k$ steps ($\sim$millions)
  - a bug of depth $d$ (1,2)

- Each run PCT triggers a bug with a probability ($p$) of at least

$$p \geq \frac{1}{n \ k^{d-1}}$$

(this is a worst-case guarantee)
PARALLEL PCT
Parallel PCT (PPCT)

- Key Idea: Serialize only $d$ threads
  - The $d$ lower priority threads are important to trigger the bug
  - Maintain two sets: higher priority set (HS) & lower priority set (LS)
  - Insert priority $d$ into LS and others into HS

Key difference between PCT and PPCT is the number of choices it provides the adversary

- PPCT operates in 2 phases
  - PCT mode: run the highest priority non-blocked thread
  - Parallel mode: run any unblocked thread with priority $> d$

PPCT provides same bounds as PCT
PARALLEL PCT EVALUATION
Bug Detection with PPCT using NeedlePoint

PPCT detects bug as effectively as PCT in practice
Slowdown with PCT & PPCT

- PPCT reduces the overhead even on a single core
- Avoids poor interaction with blocking inference
- PPCT reduces the overhead from 34X to 6X on a 8-core machine
Conclusions

• NeedlePoint framework
  • Separation of mechanisms and exploration policy
  • Inferring blocked threads

• Proposed Parallel Probabilistic Concurrency Testing
  • For bugs of depth d, runs all but “d” threads
  • Provides same coverage guarantees as PCT
  • Significant speedups both on single & multiple cores
http://www.cis.upenn.edu/~santoshn/needlepoint

Subscribe to
needlepoint@lists.seas.upenn.edu