Variable and Thread Bounding for Systematic Testing of Multithreaded Programs

Sandeep Bindal, Sorav Bansal
Indian Institute of Technology Delhi

Akash Lal
Microsoft Research India
Motivation

• Concurrency bugs are hard to find

• Difficult to reproduce

• Model Checking too expensive. Search space size for a program with \( n \) threads and each thread executing \( k \) instructions \( \approx \frac{(nk)!}{(k!)^n} \)
Previous Work (1)

• Musuvathi and Qadeer – Chess\(^1\):
  • Context Bound ‘c’ – minimum number of pre-emptive context switches required to uncover the bug.
  • Real world bugs have small c
  • Search Space – \((nk)^c\)
  • For \(n = 4, k = 10,000\) and \(c = 2\) evaluates to \(10^9\) - too large
  • Alleviated the search space size problem by considering only synchronization variables.

\(^1\)M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI ’07.*
Previous Work (2)

• Probabilistic Concurrency Testing (PCT):¹
  • Bug Depth ‘d’ – minimum number of scheduling constraints required to uncover the bug.
  • Real world bugs have small ‘d’.
  • Search Space – $nk^{d-1}$
  • Too large for most programs.

Introduction

- Variable Bounding

- Thread Bounding
Variable Bounding
Variable Bounding: Motivation

• Previous studies concluded\(^1\) – 66% of the non-deadlock bugs involve only one variable.

• Remaining fraction of non-deadlock bugs involves only 2-3 variables.

• Data-race involves only one variable.

\(^1\)S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In ASPLOS ’08.
Variable Bounding: Definition

A ‘v’ **variable bug** involve accesses to ‘v’ distinct variables at the minimal set of pre-emptive context switches required to manifest the bug.

Note: Pre-emptive Context Switches, \( c \geq v \)
Variable Bounding: Examples

\[
a = 0
\]

Thread 1:  Thread 2:
\[
t1 = a; \\
t2 = a; \\
assert( t1 == t2); \\
a = 0;
\]

\[v = 1 \text{ and } c = 2 \text{ bug}\]
Algorithm for v-variable bugs

1) Do lightweight static alias analysis of the test program.

2) For each possible subset of size v variables, repeat steps 3 & 4

3) Instrument the program for the set of v variables.

4) Explore the search space up to context bound ‘c’ of the instrumented program.
Exploration Algorithm

• Explore the search space in order of increasing values of v.

• Real world bugs have small values of v.

• Most bugs will be uncovered at $v = 1, 2$ and $3$. 
Search Space Reduction

Assume each thread accesses each variable exactly ‘d’ times and ‘Q’ is the total number of variables.

Search Space at Context Bound ‘c’ = \((ndQ)^c\) .... A

Search Space at ‘v’ and ‘c’ = \(\binom{Q}{v} (ndv)^c\) .... B

Assuming \(v,c \ll Q\), if \(v < c\) then B \(\ll\) A
Runtime Improvement

- Instrumentation of tracked variables only.
- Dynamic checking if accessed variable belongs to tracked variables.
- Runtime overhead decreased by 10-100x.
- Enables us to interpose on all variables unlike previous work.
Variable Bounding
Randomized Algorithms

• Applied variable bounding technique in Randomized Algorithms.

• Particularly showed the extension in Probabilistic Concurrency Testing (PCT).

• In general it can be extended to any randomized algorithm.
Program with $n$ threads and $k$ instructions:

• Assign $n$ priority values $d, d+1,\ldots,d+n-1$ randomly to the $n$ threads.

• Pick $d-1$ priority change points $k_1,\ldots,k_{d-1}$ randomly in the range $[1,k]$. Each $k_i$ has an associated priority of $i$.

• Schedule the threads by honouring their priorities. When a thread executes $k_i$ change point, change the priority of that thread to $i$.

---

Program with n threads and k instructions:
- Choose a set of ‘v’ variables randomly.
- Assign n priority values d, d+1, ..., d+n-1 randomly to the n threads.
- Pick d-1 priority change points k₁, ..., kₙ₋₁ randomly in the range [1,k], such that the variables accessed at these priority change points belong to set of randomly selected variables.
- Schedule the threads by honouring their priorities. When a thread reaches the i-th change point, change the priority of that thread to i.
Probabilistic Guarantees

Probability with PCT = \frac{1}{n^k d^{-1}}

Probability with PCT-VB = \frac{1}{(\frac{Q}{v})^n k'^d^{-1}}

(k' - maximum number of accesses of selected variables)

Assuming v << Q
if v < d – 1 \rightarrow Pr-PCTVB >> Pr-PCT
if v == d-1 \rightarrow Pr-PCTVB \approx Pr-PCT
Improvement in Probability for corner variables
Thread Bounding
Thread Bounding

- Most concurrency bugs involve interaction between small number of threads (mostly 2).
- Data-race involve interactions between only two threads.
A t-thread bug requires ordering constraints between at least ‘t’ distinct threads to manifest the bug.

By definition: $t \geq 2$
Thread Bounding: Examples

Thread 1: 
Thread 2: 
Thread 3: 

\[ a = 0; \]
\[ a++; \]
\[ assert(a \neq 2); \]
\[ a++; \]

\[ c = 0, v = 0 \text{ and } t = 3 \text{ bug} \]
Thread Bounding

A program is t-thread-bound bug free if all relative orders of t-thread-bound bugs are explored.
Thread Bounding: Search Space Size

- $c = 0$ and $t = 2$
- $c = 0$ and any general ‘t’
- any general ‘c’ and ‘t’
• Only two executions are required
  • One executing threads $T_1, T_2, \ldots, T_n$ in ascending order \{$T_1, T_2, \ldots, T_n$\}
  • Other executing threads in descending order \{$T_n, T_{n-1}, \ldots, T_1$\}

• Cover both orderings of all 2-sized subsets of all threads.

• For every $T_i, T_j$ pair both $T_i \rightarrow T_j$ and $T_j \rightarrow T_i$ enumerated
• Need to cover all orderings of t-sized subsets of n threads.

• Showed that \((t+1)! \left( \log(nt) + \log\left(\frac{1}{\epsilon}\right) \right)\) ordering required to explore all t-thread bugs with probability at least \((1-\epsilon)\).

• Therefore, \(O((t+1)!\log(n))\) schedules are sufficient to cover all t-thread bugs at \(c = 0\) with a high probability.
Thread Bounding: Search Space Size

\( c = 0 \) and any general ‘\( t \)’

- Example: \( n = 600 \)
  - \( t = 3 \) → 70 orderings enough to uncover bug with probability > 99%
  - \( t = 4 \) → 360
  - \( t = 5 \) → 2000
• For context bound c, consider a pre-empted thread as two distinct threads. Total number of threads = n + c.

• Explore all ordering of (t+c)-sized subsets of n+c threads.

• \( O((t+c+1)! \log(n+c)) \) schedules are sufficient to capture all c-context & t-thread bugs.
Thread Bounding: Algorithm

1) Generate a random ordering of \((n+c)\) threads.

2) Schedule the threads according to the random priority ordering.

3) On the \(i^{th}\) pre-emptive context switch change current thread id to \(n+i\) and update its priority corresponding to \(n+i\).

4) Repeat 1-3 for \((t+c+1)! \log(n+c)\) times.
Variable & Thread Bounding

• Independent techniques.

• Can be used together to provide cumulative improvements.
Implementation

• Implemented Variable and Thread bounding in Java tool – Rankchecker.

• Static Alias Analysis using BDDs.

• Rankchecker implements scheduler which dictates thread interleavings.
Experimental Results
Experimental Results

• Typical values of ‘v’ and ‘t’ in common concurrency bugs?

• Runtime Improvement due to variable bounding?

• Improvements in number of executions using systematic search strategy and randomized search strategy, due to v-t bounding?
## Experimental Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Num of Threads</th>
<th>Schedules Explored</th>
<th>((c,v,t))</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConTest Benchmarks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MergeSort</td>
<td>100</td>
<td>651</td>
<td>(1,1,2)</td>
</tr>
<tr>
<td>PingPong</td>
<td>18</td>
<td>234</td>
<td>(1,1,2)</td>
</tr>
<tr>
<td>AirLine Tickets</td>
<td>99</td>
<td>2</td>
<td>(0,0,2)</td>
</tr>
<tr>
<td>AllocationVector</td>
<td>3</td>
<td>113</td>
<td>(2,1,2)</td>
</tr>
<tr>
<td>Manager</td>
<td>6</td>
<td>33</td>
<td>(1,1,2)</td>
</tr>
<tr>
<td>Java library JDK 1.4.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td>200</td>
<td>813</td>
<td>(1,1,2)</td>
</tr>
<tr>
<td>TreeSet</td>
<td>200</td>
<td>813</td>
<td>(1,1,2)</td>
</tr>
</tbody>
</table>
Experiments: Runtime Improvements

- **Cache4J**:
  - v0/Native: 1.38x
  - v1/v0: 2.62x
  - v-all/v1: 21.45x

- **TSP**:
  - v0/Native: 1.01x
  - v1/v0: 7.72x
  - v-all/v1: 36.15x

- **Montecarlo**:
  - v0/Native: 2.44x
  - v1/v0: 3.68x
  - v-all/v1: 68.17x

- **Molydn**:
  - v0/Native: 8.08x
  - v1/v0: 3.68x
  - v-all/v1: 106.81x
Experiments: Variable Bounding in Delay Bounded-CHESS\(^1\)

<table>
<thead>
<tr>
<th>RegionOwnership</th>
<th>(C#)</th>
<th>BugFound?</th>
<th>#Executions</th>
<th>Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No VB</td>
<td></td>
<td>Yes</td>
<td>132507</td>
<td>6900</td>
</tr>
<tr>
<td>v=2</td>
<td></td>
<td>Yes</td>
<td>47248</td>
<td>1225</td>
</tr>
<tr>
<td>v=1</td>
<td></td>
<td>No</td>
<td>30437</td>
<td>581.0</td>
</tr>
</tbody>
</table>

## Experiments

### Randomized Search Strategy

MonteCarlo Benchmark

<table>
<thead>
<tr>
<th>Bug Type (c ,ν ,t)</th>
<th>With v-t bounding</th>
<th>Without v-t bounding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg # Executions</td>
<td>Time(sec)</td>
</tr>
<tr>
<td>(0,0,2)</td>
<td>3.1</td>
<td>10.9</td>
</tr>
<tr>
<td>(0,0,3)</td>
<td>3.7</td>
<td>12.1</td>
</tr>
<tr>
<td>(1,1,2)</td>
<td>1636.2</td>
<td>6409.9</td>
</tr>
<tr>
<td>(1,1,3)</td>
<td>4889</td>
<td>20371.2</td>
</tr>
<tr>
<td>(2,1,2)</td>
<td>28121</td>
<td>112950</td>
</tr>
</tbody>
</table>
Conclusion

- Most bugs can be manifest at small values of v and t.

- Variable and thread bounding significantly reduces the search space.

- Due to variable bounding, can interpose on all variable accesses not just synchronization variables.

- Showed improvements over existing tools (CHESS and PCT).
Thank You!
Questions?