1. Problem and Motivation

Concurrency is used pervasively in the development of large programs. However, testing concurrent programs is notoriously hard due to the inherent non-determinism. Recently, many different techniques have been proposed to identify concurrency-related bugs early. An effective but expensive approach is model-checking, where all possible schedules of a program are executed to ascertain the absence of a bug. Unfortunately, the space of all schedules is huge, and exhaustively enumerating it is usually infeasible. Therefore, an alternative is to try and maximize the probability of uncovering a bug rather than trying to ascertain it’s absence.

2. Background and Previous Work

Many different approaches have been proposed in this direction. Musuvathi and Qadeer recently proposed using context-bound to rank schedules, and show that it is an effective method to uncover most common bugs[1]. A context-bound is the number of pre-emptive context-switches required to execute a schedule. The schedules are enumerated in increasing order of their context-bound. They had shown that all known bugs in real-world applications can be found with context-bound values of 2 or less. The space of schedules is reduced significantly at low context-bound values, but it is still too large to exhaustively enumerate. Tools like CHESS based on this algorithm reduces this search space by considering only explicit synchronization operations as possible pre-emption points. This approach is based on the assumption that all shared-memory accesses will be protected by a lock but many systems deliberately avoid explicit synchronization[2], often for performance reasons.

Another recent tool, CTrigger, focuses on atomicity-violation bugs and preferentially searches the space of schedules that are likely to trigger these bugs. CTrigger first profiles executions of the program to determine the shared variables and their unprotected accesses. It then attempts to generate schedules that are likely to violate assumptions of atomicity. CTrigger is primarily interested in atomicity-violation bugs and often overlooks other concurrency bugs.

3. Approach and Uniqueness

3.1 Ranking

In this paper, we propose a ranking scheme of test schedules, which is applicable to all programs irrespective of the synchronization disciplines being followed, and targets all types of bugs. We rank the thread schedules to be tested on four dimensions, namely, context-bound, memory locations (variables), threads, and loop iterations. We discuss all these ranking criteria as follow:

Ranking on Context Bound :- As suggested by Musuvathi et al.[1], a context-bound search is an effective method for exploring the schedule space of multithreaded programs. For a program with $n$ threads, each executing $k$ instructions, the number of schedules at context-bound $c$ grows with $n^{n+c}k^c$. This is much better than an exponential dependence on $k$ for a search strategy that is not context bounded. Therefore, we choose context-bound as one of the dimensions to order the search space.

Ranking on Number of Memory Locations :- Recent work on studying characteristics of real-world concurrency bugs[3] concluded that 66% of the non-deadlock concurrency bugs they examined involved only one variable like data race. Among the remaining fraction of non-deadlock concurrency bugs, most bugs involve only a few variables (typically 2 to 3). This observation motivates our ranking on the number of memory locations involved. We first enumerate schedules which exhaustively check all thread interactions involving a single variable. We then enumerate schedules which exhaustively check thread interactions between a subset of two variables. And so on.

Ranking on Number of Threads :- Most concurrency bugs can be discovered by observing all interactions between a small number (typically two) of threads[3]. Hence, we first check for all bugs involving two threads, before checking for any bug involving three threads, and so on. We call a bug that requires interactions between at-least $t$ threads to be uncovered, a $t$-thread bug. This definition of a $t$-thread bug also counts the threads that should not be executed for a bug to manifest. For example, a bug that manifests only if thread A is executed after thread B and thread C is not executed in between, will be called a 3-thread bug, and not a 2-thread bug. In a system with $n$ threads $T_1, T_2, \ldots, T_n$, at context-bound $c = 0$, all 2-thread bugs can be uncovered by only two schedules, namely $\{T_1, T_2, T_3, \ldots, T_{n-1}, T_n\}$ and $\{T_n, T_{n-1}, T_{n-2}, \ldots, T_2, T_1\}$. This is because for any 2-subset $\{T_i, T_j\}$ of threads, both orders between $T_i$ and $T_j$ are covered by these two schedules. We have used priority order on the $n$ distinct threads in the program. We enumerate enough priority orders to make sure that all $t$-thread bugs are captured.

Ranking on Loop Iteration Bound :- For repeated accesses to a variable in a loop, it often suffices to track just one, or a few iterations of the loop to capture most bugs involving that variable. Therefore, we rank schedules based on loop iteration numbers. Each memory access is assigned a loop iteration number which captures the number of repetitions due to a loop. Loop iteration numbers are based on the context in which the memory accesses are performed. We define context of a variable access by it’s call stack. We rank schedules in increasing order of loop iteration numbers ($l$).

In general, the unified ranking of schedules may not be the same for every program. The best ranking order will depend on the size of the program, number of threads, number of shared variables, variable access patterns, and loop iterations. We roughly rank the search space tuples $(c, v, t, l)$ as $(0, 0, 2, 1), (1, 0, 2, 1)$,
Algorithms 1 Iterative context bounding algorithm for $t$-thread bugs

**Input:** initial state $s_0 \in \text{State}$.
1. 1 struct WorkItem { State state; Priorities prio; }
2. 2 Queue<WorkItem> WorkQueue;
3. 3 Queue<WorkItem> nextWorkQueue;
4. 4 WorkItem w;
5. 5 Queue<Priorities> threadPriorities;
6. 6 threadPriorities.init();
7. 7 int currBound = 0;
8. 8 for prio in threadPriorities do
9. 9 workQueue.Add(WorkItem(s0, prio));
10. 10 end for
11. 11 while true do
12. 12 while !workQueue.Empty() do
13. 13 w := workQueue.PopFront();
14. 14 Search(w);
15. 15 end while
16. 16 if nextWorkQueue.Empty() || currBound == c then
17. 17 Exit();
18. 18 end if
19. 19 currBound := currBound + 1;
20. 20 workQueue := nextWorkQueue;
21. 21 nextWorkQueue.Clear();
22. 22 end while
23. 23 function Search(WorkItem w) begin
24. 24 WorkItem x;
25. 25 State s;
26. 26 bool tidenabled, varaccess;
27. 27 Thread tid := highestPriorityEnabledThread(w.prio);
28. 28 s := w.state.Execute(tid);
29. 29 tidenabled := (tid $\in$ enabled(s));
30. 30 varaccess := (tid returned due to memaccess,w());
31. 31 x := WorkItem(s, w.prio);
32. 32 Search(x);
33. 33 if (tidenabled && varaccess) then
34. 34 // pre-emptive caswitch, gen more schedules
35. 35 for $v$ in threadPriorities do
36. 36 newprio := make_tid_lowest_priority(prio, tid);
37. 37 x := WorkItem(s, newprio);
38. 38 nextWorkQueue.Push(x);
39. 39 end for
40. 40 end if
41. 41 end

3.2 Algorithm

Figure 1 shows our enumeration algorithm. Our algorithm is very similar to that presented in [1], with the following differences:

1. The instrumented program points include memory accesses to the variables being tracked, and not just explicit synchronization points. Scheduler regains control from the program only at the variables being tracked, and not just explicit synchronization points. Scheduler regains control from the program only at the variables being tracked, and not just explicit synchronization points. Scheduler regains control from the program only at
2. When a thread yields control to the scheduler (Line 29), new schedules are generated only if it was because of an access to a tracked variable. In all other cases, a pre-determined priority-order is used to decide the next thread
3. Calls to memaccess,w() with callstacks that have been previously seen more than $l$ times do not yield control to the scheduler. Hence, only memory accesses in the first $l$ loop iterations are considered interesting context-switch points.

We have implemented this algorithm in a concurrency testing tool for Java, called RankChecker. Our tool does not require source-level annotations and instruments the binary class code of a Java program and associated libraries to insert appropriate schedule points. The instrumented test program is linked with the RankChecker library which spawns a scheduler thread to dictate the thread interleavings using semaphores. We have implemented an alias analysis using BDDs similar to that used in [5, 6]. Our static alias-analysis implementation is derived from the Chord tool[6] with many changes. We instrument the Java bytecode using the Javassist library[7]. We are also building a happens-before graph similar the way as CHESS[4] but only for variables that are being tracked.

4. Results and Contributions

<table>
<thead>
<tr>
<th>Program Name</th>
<th>SLOC</th>
<th># of Threads</th>
<th># of Variables Shared</th>
<th>Schedules Explored</th>
<th>$(c, v, t, l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergeSort</td>
<td>376</td>
<td>100</td>
<td>30</td>
<td>6</td>
<td>651 $(1, vz, t)$</td>
</tr>
<tr>
<td>Producer</td>
<td>279</td>
<td>61</td>
<td>49</td>
<td>1</td>
<td>$(0, v, 2, l)$</td>
</tr>
<tr>
<td>LinkedList</td>
<td>421</td>
<td>30</td>
<td>60</td>
<td>4</td>
<td>$(1, vz, t)$</td>
</tr>
<tr>
<td>MergeSort2</td>
<td>258</td>
<td>39</td>
<td>69</td>
<td>2</td>
<td>$(0, v, 2, l)$</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>129</td>
<td>100</td>
<td>13</td>
<td>2</td>
<td>$(0, v, 2, l)$</td>
</tr>
<tr>
<td>Piped</td>
<td>210</td>
<td>33</td>
<td>69</td>
<td>2</td>
<td>$(2, 1, 2, l)$</td>
</tr>
<tr>
<td>Allocation</td>
<td>288</td>
<td>24</td>
<td>73</td>
<td>3</td>
<td>$(2, 1, 2, l)$</td>
</tr>
<tr>
<td>BufferedWriter</td>
<td>259</td>
<td>27</td>
<td>73</td>
<td>1</td>
<td>$(0, 2, 2, l)$</td>
</tr>
<tr>
<td>PingPong</td>
<td>274</td>
<td>25</td>
<td>23</td>
<td>2</td>
<td>$(1, vz, t)$</td>
</tr>
<tr>
<td>Manager</td>
<td>190</td>
<td>6</td>
<td>25</td>
<td>2</td>
<td>$(1, vz, t)$</td>
</tr>
<tr>
<td>MergeSorting</td>
<td>258</td>
<td>29</td>
<td>69</td>
<td>2</td>
<td>$(1, vz, 2, l)$</td>
</tr>
<tr>
<td>Account</td>
<td>169</td>
<td>26</td>
<td>19</td>
<td>1</td>
<td>$(1, vz, 2, l)$</td>
</tr>
<tr>
<td>AirLineTickets</td>
<td>99</td>
<td>11</td>
<td>18</td>
<td>2</td>
<td>$(0, 2, 2, l)$</td>
</tr>
<tr>
<td>HashSet</td>
<td>7080</td>
<td>100</td>
<td>125</td>
<td>16</td>
<td>813 $(1, vz, 2, l)$</td>
</tr>
<tr>
<td>TreeSet</td>
<td>7532</td>
<td>200</td>
<td>144</td>
<td>20</td>
<td>813 $(1, vz, 2, l)$</td>
</tr>
<tr>
<td>Cache4j</td>
<td>8097</td>
<td>12</td>
<td>338</td>
<td>18</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1. The different columns in this table represents the name of the program, the number of source code lines, number of threads, total number of variables, number of shared variables, the number of schedules explored till we found the first bug and tuple $(c, v, t, l)$ at which we found the bug in the program.

Table 1. shows the results of our experiments. The programs from MergeSort to AirLineTickets are taken from ConTest Benchmark. We are able to find out all the known bugs in the ConTest Benchmark. We have also tested our tool on cache4j, a fast thread-safe implementation of a cache for Java Objects, up to $(c = 2, v = vz, t = 2, l = 1)$ but we are not able to find out any bug. Previously, RaceFuzzer[8] showed a data-race on variable _sleep in cache4j and said that it can throw InterruptedException but according to Java Documentation in this case InterruptedException cannot occur. Our tool also confirmed this datarace to be a benign data race. We have also tested HashSet and TreeSet in JDK 1.4.2. We have used the same multi-threaded test driver as used by RaceFuzzer[8] and we have found the same bugs in HashSet and TreeSet as reported by RaceFuzzer.

Finally, this paper makes the following contributions:

- This paper provides a systematic ranking of test schedules while testing concurrent-programms.
- This paper presents new classifications of bugs based on the number of variables involved, the number of threads required to uncover it, and the number of loop iterations.
- This paper presents a concurrency-testing tool for Java based on our ranking scheme. Our tool tests for many different types of concurrency bugs at once.
References


