Variable and Thread Bounding for Systematic Testing of Multithreaded Programs

Sandeep Bindal * Sorav Bansal
Indian Institute of Technology Delhi, India

Akash Lal
Microsoft Research India

ABSTRACT

Previous approaches to systematic state-space exploration for testing multi-threaded programs have proposed context-bounding [20] and depth-bounding [6] to be effective ranking algorithms for testing multithreaded programs. This paper proposes two new metrics to rank thread schedules for systematic state-space exploration. Our metrics are based on characterization of a concurrency bug using v (the minimum number of distinct variables that need to be involved for the bug to manifest) and t (the minimum number of distinct threads among which scheduling constraints are required to manifest the bug). Our algorithm is based on the hypothesis that in practice, most concurrency bugs have low v (typically 1-2) and low t (typically 2-4) characteristics. We iteratively explore the search space of schedules in increasing orders of v and t. We show qualitatively and empirically that our algorithm finds common bugs in fewer number of execution runs, compared with previous approaches. We also show that using v and t improves the lower bounds on the probability of finding bugs through randomized algorithms.

Systematic exploration of schedules requires instrumenting each variable access made by a program, which can be very expensive and severely limits the applicability of this approach. Previous work [6, 20] has avoided this problem by interposing only on synchronization operations (and ignoring other variable accesses). We demonstrate that by using variable bounding (v) and a static imprecise alias analysis, we can interpose on all variable accesses (and not just synchronization operations) at 10-100x less overhead than previous approaches.

Categories and Subject Descriptors

D.2.4 [*Software Engineering*]: Software/Program Verification — formal methods, validation; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — mechanical verification, specification techniques; D.2.5 [*Software Engineering*]: Testing and Debugging — debugging aids, diagnostics, monitors, tracing

* A part of this work was done when the author was interning at Microsoft Research, India.

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
ACM 978-1-4503-2159-4/13/07
http://dx.doi.org/10.1145/2483760.2483764

General Terms

Algorithms, Reliability, Verification

Keywords

Concurrency, context-bounding, variable-bounding, thread-bounding, model checking, multi-threading, concurrency-bug classification, shared-memory programs, software testing

1. INTRODUCTION

Testing concurrent programs is notoriously difficult because of its inherent non-determinism. 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. For a multi-threaded program with n threads, each executing k instructions, the total number of schedules (or thread interleavings) is $\frac{(nk)!}{(k!)^n}$. This space of schedules further explodes if each instruction is not guaranteed to be atomic. For a very small program with $k = 100$ and $n = 2$, the total number of interleavings is around 10^{59}!

As it is practically impossible to exhaustively explore the entire state space of all schedules for any useful program, an alternative is to try and maximize the probability of uncovering a bug rather than trying to ascertain its absence. Many different approaches have been proposed in this direction. Musuvathi and Qadeer proposed using *context-bound* to rank schedules, and show that it is an effective method to uncover most common bugs [20]. 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, i.e., all schedules with context bound $c-1$ are executed before any schedule with context bound c. Musuvathi and Qadeer report experiments on real-world applications, and show that all known bugs in those applications were found at context-bound values of 2 or less.

Iterative context bounding is an effective way of ranking schedules. However, this metric is often too coarse-grained. For a multi-threaded program with n threads, each executing k instructions, the total number of schedules at context-bound c grows with $(nk)^c$. For a small program with $k = 10,000$ instructions and $n = 4$, the number of schedules at context bound 2 is on the order of 10^9! Musuvathi et. al's concurrency-testing tool based on this algorithm, CHESS, reduces this search space by considering only explicit synchronization operations as possible pre-emption points, thus reducing k by at least 2-3 orders of magnitude. This simplification is justified by the assumption that most programs follow a mutual-exclusion locking discipline, and hence all shared-memory accesses will be protected by `lock()` and `unlock()` calls. Violation of this locking discipline can be separately checked using other race-detection tools. This approach, though effective, is not completely general, as many systems deliberately avoid explicit synchronization [28], often for performance reasons.

Another approach to testing multithreaded programs is randomization of scheduling decisions with probabilistic guarantees. Burckhardt et. al. [6] characterize a concurrency bug by its *depth*—the minimum number of scheduling constraints required to find the bug. They provide an algorithm that provides a lower bound on the probability of finding a depth-d bug. Ranking on bug-depth d restricts the search space of a multi-threaded program with n threads and executing k instructions to nk^{d-1}. This, again, may be too large for most programs.

Another recent tool, CTrigger [23], 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 (for example, by inserting a write to location M by some thread between two accesses to the same location M by another thread). CTrigger is primarily interested in atomicity-violation bugs and often overlooks other concurrency bugs.

Our first contribution is to propose the use of number of variables to further classify and reduce the schedule search space. Our algorithm is based on the hypothesis that in practice, most concurrency bugs can be uncovered by restricting our search to only a few variables at a time. We search for bugs involving a small subset of v variables. These variables may include synchronization operations. We consider all such variable subsets in turn. For a given subset of variables, we perform static alias analysis to identify all program locations where these variables may be accessed. We instrumented only these program locations. This selective instrumentation allows us to run our program at near-native speed. Consequently, our approach can interpose on any variable accesses, and not just synchronization variables as reported in previous work. We show that using variable bounding, the search space reduces by a factor of roughly $(\frac{Q}{v})^{c-v}$ when searching for bugs with context-bound c and variable bound v, where Q is the total number of variables in the program. We confirm this result experimentally by showing that variable bounding reduces time to discovery of concurrency bugs.

Our second contribution is characterizing a concurrency bug by the number of distinct threads that need to be order-constrained to uncover the bug. A bug that can be uncovered by constraining the order of t threads is called a t-thread bug. In practice, most bugs have a small t. We provide a randomized algorithm with guarantees on the probability of uncovering a t-thread bug, if any exists. The search space decreases by a factor of $\frac{n!}{(t+1)!log(n)}$ when using thread-bounding to search for bugs with thread-bound t out of a total of n program threads.

Our hypothesis that most bugs can be uncovered at low (v, t) values conform with the observations made in previous work on studying real-world concurrency bug characteristics [17].

The paper is organized as follows. Section 2 presents and analyzes variable bounding for exhaustive model-checking algorithms. Section 3 discusses variable bounding for randomized algorithms and analyzes the resulting probabilistic guarantees of finding a bug, if one exists. Section 4 discusses thread bounding. Sections 5 and 6 discuss our implementation and empirical results. Section 7 discusses related work, and Section 8 concludes.

2. VARIABLE BOUNDING

Recent work on studying characteristics of real-world concurrency bugs [17] concluded that 66% of the non-deadlock concurrency bugs they examined involved only one variable. Perhaps, the most common type of concurrency bug involving one variable access is a data race. i.e., simultaneous access of a shared variable (of which, one is a write) by two or more threads without proper synchronization. Also, 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 that exhaustively check all thread interactions involving a single variable. We then enumerate schedules that exhaustively check thread interactions involving two variables, and so on.

We first discuss variable bounding in the context of a model-checker. For a model-checker like CHESS [21], a custom priority scheduler implements the exhaustive enumeration of schedules, and context-bounding [20] is used to limit the number of schedules executed. To implement variable bounding, we first identify all program variables (or points in the program that generate new variables) by parsing the program. These program variables include globals and heap-allocated variables (allocated using `malloc()` or `new`). A heap variable is identified and named by its allocation statement and the number of times that statement has been invoked. For example, if a particular `new` statement is called multiple times, we will consider each return value as a separate variable. We call this set of program variables ϑ. Iteratively, we take all v-sized subsets of variables in ϑ for $v \in \{1, 2, 3, \ldots\}$. For a subset V of size v, we execute schedules that explore *all* interactions between all variables in V.

To identify variables, we instrumented heap allocation statements to generate a new variable name for each invocation of the statement. As we explain later, we also prioritized the variables which are generated in the first few loop iterations. To identify interactions between a subset of variables, we instrumented accesses to these variables. We used a lightweight and imprecise static alias analysis [1, 16, 27] to identify program points at which each variable in ϑ *may* be accessed. Our static analysis assumes that the program is *memory-safe*. i.e., locations outside allocation boundaries will not be accessed. Memory-safety can be separately checked using other available tools.

Without variable bounding, all accesses to all variables must be instrumented with a call to the scheduler which implements exhaustive schedule enumeration. With variable bounding, this instrumentation can be significantly reduced. For a variable $x_i \in \vartheta$, we call the set of program locations at which it may be accessed a_{x_i}. With variable bounding, we only checked interactions within a variable subset $V = \{x_0, x_1, \ldots, x_v\}$, and instrumented all locations in the set $(a_{x_0} \cup a_{x_1} \cup \cdots \cup a_{x_v})$. The instrumentation code includes a call to a scheduler function, `varaccess()` that yields to the scheduler which implements priority scheduling and systematic pre-emption. `varaccess()` is inserted after the program has accessed and possibly updated the variable. To ensure that pre-emption occurs only on accesses to the set of tracked variables, the instrumentation code dynamically checks that the accessed memory address is one of the tracked variables before calling `varaccess()`. The `varaccess()` call serves as a potential yield point (or context-switch point), i.e., at this point, the scheduler can choose to run another thread. To allow a thread to be pre-empted before its first access to a variable, we also inserted a *fake* `varaccess()` before the first instruction of each thread. Our enumeration algorithm is similar to that used in CHESS [21] and we discuss it in Section 5.

2.1 Bug Characterization

We call a concurrency bug a c context bug if at least c pre-emptive context switches are required for the bug to manifest. c is also called the bug's context-bound. This definition of context bound is taken from previous work [20].

We call a concurrency bug a v-variable bug if the minimal set of constraints required to manifest the bug involve preemption points at accesses to v distinct variables. v is also called the bug's variable bound. By definition, $v \leq c$ for any c, v bug.

Figures 1, 2, 3, 4 show short programs with $(c = 0, v = 0)$, $(c = 1, v = 1)$, $(c = 2, v = 1)$, $(c = 2, v = 2)$ bugs respectively for exposition. The numbers in comments give the order of execution for an assertion failure. In these short programs, we count a pre-emption against the shared variable that was last accessed. Also, we assume that a bug exists if the ASSERT statement can fail.

```
                 a = 0
Thread 1:                  | Thread 2:
  ASSERT(a == 0); // 2     |   a++; // 1
```

Figure 1. A short program with a $\mathbf{c = 0, v = 0}$ bug

```
                 a = 0
Thread 1:                  | Thread 2:
  t1 = a; // 1             |   a++; // 2
  t2 = a; // 3             |
  ASSERT(t1==t2); // 4     |
```

Figure 2. A short program with a $\mathbf{c = 1, v = 1}$ bug

```
                 a = 0
Thread 1:                  | Thread 2:
  t1 = a; // 2             |   a = 1; // 1
  t2 = a; // 4             |   a = 0; // 3
  ASSERT(t1==t2); // 5     |
```

Figure 3. A short program with a $\mathbf{c = 2, v = 1}$ bug

```
                    a = 0, b = 0
Thread 1:                            | Thread 2:
  t1 = a; // 1                       |   a = 1; // 2
  t2 = a; // 3                       |   b = 1; // 4
  t3 = b; // 5                       |   b = 0;
  ASSERT(t1==t2 or t3 != 1); // 6    |
```

Figure 4. A short program with a $\mathbf{c = 2, v = 2}$ bug

2.2 Schedule Characterization

A schedule is characterized by c—the number of pre-emptive context switches in it, and v—the number of distinct variables at which a pre-emptive context switch was performed.

2.3 Search Space Reduction

We now discuss how variable bounding helps reduce the search space. Let us assume that a multi-threaded program with t threads has Q distinct shared variables, represented as a set ϑ of variables, i.e., $|\vartheta| = Q$. For simplicity, let us also assume that each thread in the program accesses each variable in ϑ exactly d times. Hence, the total number of variable accesses by a thread are dQ. Assuming that only accesses to these shared variables are interesting context-switch points, and assuming n threads, $k = ndQ$ (k is the number of steps in a program). Therefore, the number of schedules that need to be explored at context bound c are $O((ndQ)^c)$. Let us call this expression A.

If we focus on a subset $V \subset \vartheta$ of v variables, the number of schedules that need to be explored at context bound c are $\binom{Q}{v}(ndv)^c$ (first choose a subset $V \subset \vartheta$, then explore all schedules with preemptions at accesses to variables in V). Assuming $v, c \ll Q$, this expression is $O(Q^v(ndv)^c)$. Comparing with A, we see that this expression is less than A if $v < c$. This reduction in the search space (number of execution runs) is significant for programs with a large number of variables (large Q). Apart from this reduction in the number of execution runs, the time taken by each execution run also decreases dramatically with variable bounding, as only the accesses to variables being tracked need to be instrumented. We study both these improvements in detail in our experiments in Section 6.

At $v = c$, variable bounding provides no improvement in the size of the search space, but still provides a significant reduction in runtime because of much lower instrumentation overhead (only the tracked variables need to be instrumented). Effectively, by slicing the program into accesses to a small subset of variables, we reduce the number of program steps k. This is because only accesses to the variable being tracked are considered valid context switch points. As we discuss in our experiments (Section 6), this reduction is significant for most programs. This method of reducing k is more general than the approach used in previous tools (e.g., CHESS [21]) where all accesses to non-synchronization variables are ignored.

While we have used a simplified assumption of constant number of accesses d to each variable by each thread, the result does not change (although the analysis gets more involved) if we assume varying number of accesses by each thread to different variables. The same result can be obtained by replacing d with the average number of accesses by a thread to a randomly-chosen variable, and we skip this discussion for brevity. We analyze a more general scenario in our discussion on probability bounds for randomized bug-finding algorithms (Section 3).

2.4 Heap Allocated Variables and Arrays

Our set of tracked variables include heap-allocated variables. Heap-allocated variables are named using the heap-allocation statement and the number of times that statement was executed before this variable was generated. A large number of heap allocations by one statement can generate a large number of variables causing our variable-bounding algorithm to get stuck at low v values.

In our experience, if the program contains a bug involving a certain *type* of heap variable, the bug usually manifests while tracking the first few variables of that type. For example, if the program constructs and accesses a heap data structure (e.g., linked list), it is very likely that a bug, if it exists, will be exposed by exploring all interactions among the first few elements of that data structure.

The challenge is to identify and group variables of a certain type, so that only the first few variables of that type are considered. We use a simple heuristic that we found to work well in practice. The type of a variable is defined by the callstack at the time of allocation of that variable. We expect that largely, variables allocated with identical callstacks are of the same type. This heuristic is neither sound nor complete. For example, it is possible that variables of the same type are allocated at different points in the program, hence having different callstacks. This can cause our algorithm to execute more than the required number of schedules. A more serious problem is that two identical callstacks could generate completely different types of variables. This can cause our algorithm to overlook certain bugs. Fortunately, in practice, such code is rare.

The algorithm works as follows. For each heap allocation, we generate a new variable ID labeled by the location of the allocation statement and the number of times that statement was executed. With each variable ID, we also associate the number of times this allocation statement has previously been executed with an identical callstack. We call this latter number, the loop iteration number (because the allocations with identical callstacks must be happening through a loop) of that variable. We first search for bugs involving variables with lower loop iteration numbers before searching for bugs involving variables with higher loop iteration numbers. We call this algorithm loop-iteration bounding and denote the current loop-iteration number being searched with letter l. Figure 5 shows our logic for implementing loop iteration numbers. Note that, by design, variables allocated by recursive calls with different recursion depth will be named differently (because they will have different call stacks).

```
<instrumentation code for new()>
callstack := get_current_callstack();
v := <heap-allocation-statement, alloc#>;
lin := loop_iteration_number(callstack);
increment_loop_iteration_number(callstack);
if (lin <= l) {
 add v to the set Q of the variables to be tracked;
}
```

Figure 5. Instrumentation code for heap-allocation statements that considers only variables with loop-iteration number $\leq$ l.

We also need special handling for array variables. Whenever possible, we treat each location in the array as a separate variable. If the search space size becomes unmanageably large (for high values of v), we use a less precise but sound approach of considering the whole array as a single variable.

3. VARIABLE BOUNDING ON RANDOMIZED ALGORITHMS

Apart from exhaustive state space exploration to ascertain the absence of certain bugs, randomized schedulers that provide probabilistic guarantees of finding certain types of bugs have also been proposed. Depth-bounding [6] (also called Probabilistic Concurrency Testing in the paper) is one such approach. The primary advantage of randomized approaches over exhaustive search is that the former can cover a large part of the program in relatively fewer runs. Exhaustive search, on the other hand, can get stuck in local regions of the program for long periods of time causing bugs in other regions to go undetected. In this section, we discuss variable bounding in the context of randomized search.

In particular, we study Probabilistic Context Bounding (PCT) [6] that proposed the *bug-depth* metric. While we analyze only PCT, similar arguments will hold for other randomized algorithms. For a program spawning at most n threads and executing at most k total instructions, PCT algorithm works as follows (for an input parameter d, denoting the depth of the bug being searched):

1. Assign n priority values d, $d+1$,..., $d+n$ randomly to the n threads.
2. Pick $d-1$ priority change points k_1,..., k_{d-1} randomly in the range $[1, k]$. Each k_i has an associated priority value of i.
3. Schedule the threads by honoring their priorities, i.e., always execute an enabled thread with the highest priority. When a thread reaches the i-th change point (i.e., executes the k_i-th instruction), change the priority of that thread to i.

Burckhardt et. al. [6] proved that this algorithm finds a bug of depth d with probability at least $1/nk^{d-1}$.

We implemented variable bounding on PCT by first randomly choosing a set of v variables, and then randomly choosing $d-1$ priority change points at one of the accesses to the chosen variables (other instructions in the program are not considered as potential priority change points). For heap-allocated variables, we simply choose a heap allocation statement (`new` and `malloc`) in lieu of a variable. Accesses to any of the variables allocated at the chosen heap-allocation statement are considered potential priority change points.

Notice that using a heap-allocation statement as one "variable" in the randomized algorithm is a departure from the strategy used in the exhaustive-search strategy, where each heap allocation is considered a separate variable. This is done to ensure that we know the number of these variables at compile time, and hence can appropriately choose a variable set to provide probabilistic guarantees. Under this new definition of a variable, a v-variable bug is a bug that involves memory locations allocated at at most v distinct heap-allocation statements (or globals). This new definition performs a coarser classification of program's memory locations. This could potentially cause higher number of required executions for effective state space search for the same v value. However, this is still a significant improvement over not using variable bounding at all. Also, this definition of variable bounding does not make our argument on most bugs having low variable bounds any weaker.

Assume that the total number of global variables and heap allocation statements in a program is Q. We changed the PCT algorithm to implement variable bounding as follows:

0. Choose a set of v variables q_1,...,q_v representing the minimal set of variables involved in the bug being searched ($v < d$).
1. Assign n priority values d, $d+1$,..., $d+n$ randomly to the n threads.
2. Let k_{q_1},...,k_{q_v} denote upper-bounds on the number of instructions accessing q_1,... q_v respectively in any run of the program. Hence, variable q_r is accessed at most k_{q_r} times in any execution of the program. Construct a set S of elements of the form (q_r, j), where r is in the range $[1, v]$ and j is in the range $[1, k_{q_r}]$. The set S will have $k = \sum_{r=1...v} k_{q_r}$ elements. Pick $d-1$ random elements from S to represent the priority change points.
3. Schedule the threads by honoring their priorities. For i-th chosen element (q_{r_i}, j_i) in the previous step, force a priority change point at the j_ith access of the q_{r_i}th variable. i.e., change the priority of the thread at this point to i.

We call this modified algorithm PCTVB (PCT with Variable Bounding). Unlike PCT, where the priority change points k_is are chosen randomly from $1, \ldots, k$, PCTVB first picks a set of variables (or heap-allocation statements), and then chooses priority change points among the accesses to this set. PCTVB has two advantages over PCT:

1. As we show below, PCTVB improves the probability bound on finding a bug with depth d and variable-bound v. Because most bugs have low v, this results in overall improvement in the bug-finding probability.
2. Choosing a set of variables apriori allows us to instrument only the program points that can potentially access these variables. These program points are identified using static (imprecise) alias analysis. This is a significant improvement over PCT where all variable accesses need to be instrumented.

3.1 Probabilistic Guarantees of PCTVB

The analysis of the probabilistic guarantees of PCTVB is identical to that of PCT, as presented in the original paper [6] and we omit it for brevity. We simply revisit Theorem 9 (without proof) of the original paper with our new variable bounding enhancement.

THEOREM 3.1. *Let P be a program with a bug B of depth d and q_1,...,q_v be the minimal set of unique variables, accesses to which need to be preempted to trigger B. For a variable q_i, let $k_{q_i} \geq maxaccesses(P, q_i)$. Assuming $n \geq maxthreads(P)$,*

$$Pr[\mathit{PCTVB}(n, k, d, q_1, \ldots, q_v) \in B] \geq \frac{1}{n(\sum_{i=1..v} k_{q_i})^{d-1}}$$

Here, B is the set of schedules that expose the d-depth bug in the program. $maxaccesses(P, q_i)$ returns the maximum number of accesses made by P to variable q_i in any single run. $maxthreads(P)$ is the maximum number of threads spawned in P. The proof is identical to that of the original theorem, and is obtained by simply replacing k with $\sum_{i=1..v} k_{q_i}$.

For a program with Q total global variables and heap allocation statements, the probability that we pick the correct v variables (q_1,...,q_v) to trigger the v-variable bug (if it exists) is $\frac{1}{\binom{Q}{v}}$. Hence, the probability of finding the v-variable bug is

$$Pr[\text{PCTVB}(n, k, d, v) \in B] \geq \frac{1}{\binom{Q}{v}} \frac{1}{n(\sum_{i=1..v} k_{q_i})^{d-1}}$$

This expression depends on the sum of the access frequencies k_{q_1},...,k_{q_v} of the variables q_1,...,q_v. Given that the total number of variables is Q, and the total number of variable accesses in any single run is at most $k = \sum_{q_i} k_{q_i}$, we expect this sum to be less than $\frac{vk}{Q}$ on average (averaged over all v-sized sets of global variables and heap allocation statements). Let us assume that the sum is $f\frac{vk}{Q}$ where $f \leq 1$ on average but could be higher depending on the set of chosen variables. Upper-bounding $\binom{Q}{v}$ with $O(Q^v)$ for small values of v, this expression evaluates to

$$Pr[\text{PCTVB}(n, k, d, v) \in B] \geq \frac{Q^{d-v-1}}{n(kvf)^{d-1}}$$

Comparing this with PCT's original bound of $\frac{1}{nk^{d-1}}$, variable bounding helps if

$$Q^{d-v-1} \geq (vf)^{d-1}$$

Assuming $v \ll Q$, variable bounding significantly improves the lower bound on probability if $v < d - 1$ and f is small. In other words, variable bounding helps if the bug being searched involves fewer variables than its bug depth, *and* these variables are accessed less than average access frequencies.

A case of particular interest are bugs with variable bound $v = 1$, as they are by-far the most common. The inequality shows that the probability of finding a 1-variable bug of depth 2 or higher improves significantly if $f < 1$. In other words, the probability of finding bugs involving "corner variables" (variables used rarely compared to others) improves with variable bounding. Intuitively, variable bounding gives all variables an equal chance, while plain depth-bounding (or context-bounding) gives higher chance to more frequently-accessed variables. We confirmed this experimentally by writing a small program with two variables and varied the relative access frequencies of the variables. One of the two variables was involved in a $c = 1, v = 1, t = 2$ concurrency bug. Figure 6 shows that as the frequency of access to the variable containing the bug is decreased, PCTVB requires fewer executions to find the bug compared to PCT.

We profiled the access frequencies of variables listed in Table 1. Some of the detailed graphs can be found in Figure 7 and other can be found in our technical report [4]. The number of accesses varies widely across different variables for almost all benchmarks. Typically, we expect variables with fewer accesses to undergo relatively less testing and thus have higher likelihood of having bugs. Even if we assume that all variables are equally likely to contain bugs, we see that variable bounding improves the overall probability of finding a bug (if one exists). We present a simple example. Consider a program with a $v = 1, d = 2$ bug that manifests if a certain priority sequence is followed and priority change point occurs on a certain access a_{q_b} to variable q_b. Assume there are Q different variables in the program, and each variable q_i is accessed at most k_{q_i} number of times in any one run of the program. Hence the probability of uncovering the bug is the probability that we pick the correct priority sequence, and the probability that we choose a_{q_b} as the lone priority change point. The former is independent of variable bounding. Below, we compare the latter, with and without variable bounding.

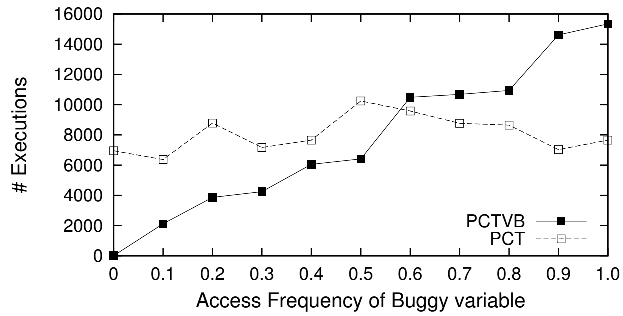

Figure 6. Figure represents the number of executions required (on average) to trigger the bug for PCT and PCTVB as the access frequency of the buggy variable is changed.

Without variable bounding, the probability of picking a_{q_b} as a priority change point is at least $\frac{1}{\sum_{q_i} k_{q_i}}$ (let's call this expression $E1$). This expression is simply the probability of choosing a_{q_b} among $\sum_{q_i} k_{q_i}$ potential priority change points. Notice that $E1$ is independent of k_{q_b}.

With variable bounding, we first choose a variable and then choose an access point of that variable. Hence, the probability that we pick a_{q_b} as a priority change point is $\geq \frac{1}{Q} \cdot \frac{1}{k_{q_b}}$ (the probability that we pick q_b multiplied by the probability that we pick a_{q_b}). This expression depends on q_b and k_{q_b}. Assuming each variable is equally likely to contain a bug, further computing the expected value of this expression over all q_b, we get $\frac{1}{Q} \sum_{q_b} \frac{1}{Qk_{q_b}}$ (let's call this expression $E2$).

Comparing $E1$ and $E2$, and using Jensen's inequality, we get

$$\frac{1}{\sum_{q_i} k_{q_i}} \leq \frac{1}{Q} \sum_{q_i} \frac{1}{Qk_{q_i}}$$

or $E1 \leq E2$ with the equality happening only at $k_{q_0} = k_{q_1} = \ldots = k_{q_Q}$. For typical access patterns to variables in common programs (see Figure 7), $E2$ is expected to be significantly higher than $E1$. Hence, assuming all variables are equally likely to have a bug, variable bounding provides a tighter bound ($E2$) on the probability of finding the bug at $v = 1, d = 2$. A similar argument holds for higher values of v and d, and we omit the discussion for brevity.

4. THREAD BOUNDING

Previous work on studying concurrency bugs found that most concurrency bugs can be discovered by enforcing ordering constraints between a small number (typically two) of threads [17]. This is our inspiration for using thread-bounding while searching for concurrency bugs. We call a bug that requires ordering constraints between at-least t distinct threads to be uncovered, a t-thread bug. t is also called the *thread-bound* of the bug. By definition, the thread-bound of a concurrency bug is always 2 or higher. Notice that our definition of thread-bound 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. Also, t is independent of c and v. i.e., a c context-bound bug and a v variable-bound bug, can have any thread bound $t \geq 2$. Figures 8, 9, 10 show short programs with ($c = 0, v = 0, t = 3$), ($c = 1, v = 1, t = 3$), and ($c = 2, v = 2, t = 3$) bugs, respectively.

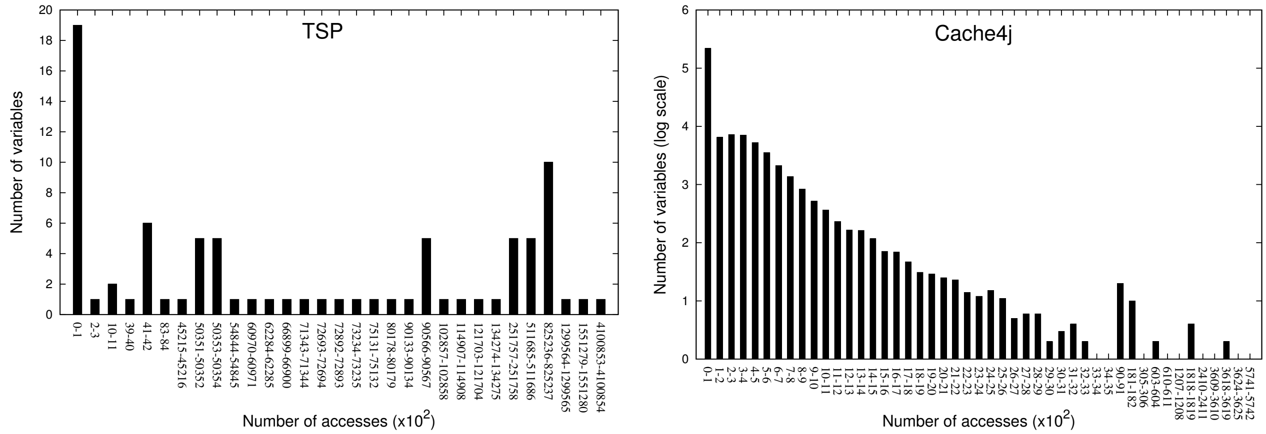

Figure 7. This figure plots the variable access frequency profile for six of our benchmarks. The values on the x-axis represent the frequency of access of a variable, and the y-axis plots the number of variables that are accessed at that frequency. For example, in tsp, 19 variables are accessed between 0 to 100 times (first vertical bar), and only 1 variable is accessed between 200 to 300 times. These access frequencies were determined after running our benchmarks multiple times on different inputs and averaging the results.

```
a = 0
Thread 1:    | Thread 2:    | Thread 3:
  a++;       |   a++;       |   ASSERT(a!=2);
```

Figure 8. A short program with a $\mathbf{c = 0, v = 0, t = 3}$ bug

```
a = 0
Thread 1:               | Thread 2:  | Thread 3:
  t1 = a;               |   a++;     |   a++;
  t2 = a;               |            |
  ASSERT(t1 ≤ t2+1);    |            |
```

Figure 9. A short program with a $\mathbf{c = 1, v = 1, t = 3}$ bug

```
a = 0
Thread 1:                      | Thread 2:  | Thread 3:
  t1 = a;                      |   a++;     |   b++;
  t2 = a;                      |            |
  t3 = b;                      |            |
  t4 = b;                      |            |
  ASSERT(t1==t2 or t3==t4);    |            |
```

Figure 10. A short program with a $\mathbf{c = 2, v = 2, t = 3}$ bug

We posit that the number of schedules required to uncover a t-thread bug increases with t. For example, for a program with n threads $T_1, \cdots, T_n$, 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 subset of 2 threads $\{T_i, T_j\}$, both orders between T_i and T_j (i.e., $\{T_i, T_j\}$ and $\{T_j, T_i\}$) are covered by these two schedules. In other words, if we arrange the threads in an arbitrary permutation, enumerating two orders (increasing and decreasing) are enough to uncover all 2-thread bugs at context bound 0.

A similar argument holds for t-thread bugs where $t > 2$. At $c = 0$, it suffices to enumerate enough schedules to explore all $t!$ relative orderings of all t-sized subsets of the n threads, to uncover a t-thread bug. To do this, we require an algorithm that generates enough permutations of n numbers, such that all $t!$ permutations of all t-sized subsets of the n numbers are exhaustively covered.

Based on the following lemma (lemma 4.1), we present a randomized algorithm to enumerate all $t!$ permutations of *all* t-sized subsets of n numbers using less than $O((t+1)!log(n))$ permutations of n numbers with a high probability. Notice that the algorithm has only logarithmic growth with n, as opposed to $n!$ growth without thread bounding.

LEMMA 4.1. *The number of independent random permutations of n numbers that need to be generated to observe* all $t!$ *relative orderings of* all $\binom{n}{t}$ *subsets of size t with probability at least* $(1-\epsilon)$, *is* $(t+1)!(log(nt) + log(\frac{1}{\epsilon}))$.

Proof Let N be a set of n distinct elements. Consider a fixed subset $S \subset N$ of t elements and let π be some arbitrary permutation of S. For any random permutation σ of n elements, the probability that π is a subsequence of σ is $\frac{1}{t!}$ (by argument of symmetry). Hence, the probability of π *not* appearing in σ is $(1 - \frac{1}{t!})$. If we enumerate P independent random permutations of n numbers, the probability of π not appearing in any of the P permutations is $(1 - \frac{1}{t!})^P$. For a fixed permutation π, let us denote this probability of π not appearing in any of the P permutations by F_π.

There are $\binom{n}{t}$ subsets of N of size t, each having $t!$ permutations. Let us denote this set of $t!\binom{n}{t}$ permutations by Θ. The probability that *any* one of the permutations in Θ is not observed in P random permutations of n numbers is at most the sum of individual probabilities $\sum_{\pi \in \Theta} F_\pi = t!\binom{n}{t} F_\pi$. We require this quantity to be less than ϵ.

$$t!\binom{n}{t}(1 - \frac{1}{t!})^P \leq \epsilon$$

Writing P as $(t!M)$, and approximating $(1 - \frac{1}{t!})^{t!}$ by $\frac{1}{e}$,

$$t!\binom{n}{t}(\frac{1}{e})^M \leq \epsilon$$

Approximating $t!$ by t^t, and $\binom{n}{t}$ by n^t,

$$M \geq tlog(nt) + log(\frac{1}{\epsilon})$$

Replacing M with P,

$$P \geq (t+1)!(log(nt) + log(\frac{1}{\epsilon}))$$

Even if ϵ is inverse-exponential in n, P is still linear in n. ∎

As an example, given a maximum of n threads, at $t = 3$, it suffices to enumerate $(24log(n))$ random permutations of the n numbers to observe all 3! relative orderings of *all* $\binom{n}{3}$ subsets with high probability. For $n = 600$, we found using simulations that 70, 360 and 2000 random permutations were enough to generate all relative orders of all $\binom{n}{3}$ $(t = 3)$, $\binom{n}{4}$ $(t = 4)$ and $\binom{n}{5}$ $(t = 5)$ subsets respectively, with more than 99% probability.

To generalize to higher context-bounds, we consider a pre-empted thread as two distinct threads (thread fragments) in this algorithm. Hence, for context-bound c bugs on a program with at most n threads, we consider $n + c$ distinct thread fragments. To cover all t-thread bugs at c context-bound, it suffices if we enumerate all $(t+c)!$ permutations of all $(t+c)$-sized subsets of the $n + c$ thread fragments. (This is more than what is strictly required because here we are also enumerating orderings between thread fragments belonging to the same thread). Hence, using Lemma 4.1, the number of schedules that need to be executed before all t-thread bugs have been tested at context bound c with high probability is $O((t + c + 1)!log(n + c))$.

To summarize, the exploration algorithm works as follows. A random permutation of $1, \ldots, (n + c)$ numbers is generated at the start of each execution run. Let us label the generated permutation $P_1, \ldots, P_{n+c}$. The scheduler uses strict priority scheduling using $P_1, \ldots, P_n$ as the priorities of threads $1, \ldots, n$ respectively. On the ith pre-emptive context switch, the priority of the running thread is changed to P_{n+i}. If $(t + c + 1)!log(n + c)$ such executions are performed, each time with a new random permutation, we expect all t thread bugs at context bound c to be covered with a high probability. (If variable bounding is also being used, then this is repeated for each set of variables). Notice that the algorithm is independent of t; we only provide probabilistic guarantees on the absence of bugs with thread-bound less than t after a certain number of schedules have been executed.

5. IMPLEMENTATION

We implemented variable and thread bounding in a concurrency testing tool for Java, called RankChecker. RankChecker instruments the binary class code of a Java program and associated libraries to insert appropriate schedule points. It does not require any source-level annotations. We instrumented Java bytecode using the Javassist library [7]. The instrumented test program is linked with a RankChecker library that implements a scheduler to dictate the thread interleavings. We implemented static alias analysis using BDDs, similar to that used in [22, 27]. Like previous approaches on systematic and probabilistic testing [6, 21], the program under test is required to be terminating, so that it can be run repeatedly to explore different schedules. It is usually straightforward to convert a non-terminating program to a terminating program.

We implemented two different algorithms: exhaustive and randomized. The exhaustive algorithm searches the state space of all schedules systematically. The randomized algorithm searches the state space randomly, with probabilistic guarantees on the probability of finding a bug of certain type (e.g., depth).

We first discuss the implementation of the exhaustive search strategy. The pseudo-code is shown in Algorithm 1. The algorithm is invoked for each set of variables (determined using variable bounding). For each set of variables, a set of thread priority orders $threadPrios$ are generated and executed. Strict priority scheduling is followed (line 28) and priorities are changed at variable accesses using the thread bounding algorithm (line 37).

A program state s is identified by the partial thread schedule that was executed. We implemented a simple record-replay mechanism, whereby a thread schedule is recorded and later replayed to reconstruct the same state. As noted in [21], replays may not result in identical states due to other sources of non-determinism (e.g., environment, non-deterministic calls, etc.). Our current implementation deals with these issues by enforcing a deterministic input at all of these non-deterministic points through bytecode instrumentation.

Algorithm 1 Iterative context bounding algorithm for t-thread bugs

Input: initial state $s_0 \in$ State.

1 struct WorkItem { State $state$; Priorities $prio$; }
2 Queue<WorkItem> $WorkQueue$;
3 Queue<WorkItem> $nextWorkQueue$;
4 WorkItem w;
5 Queue<Priorities> $threadPrios$;
6 $threadPrios$.init(t);
7 int $currBound$:= 0;

8 **for** $prio \in threadPrios$ **do**
9 $\quad workQueue$.Add(WorkItem (s_0, $prio$));
10 **end for**
11 **while** $true$ **do**
12 $\quad$ **while** $\neg workQueue$.Empty() **do**
13 $\quad\quad w := workQueue$.PopFront();
14 $\quad\quad$ Search(w);
15 $\quad$ **end while**
16 $\quad$ **if** $nextWorkQueue$.Empty() || $currBound == c$ **then**
17 $\quad\quad$ Exit();
18 $\quad$ **end if**
19 $\quad currBound := currBound + 1$;
20 $\quad workQueue := nextWorkQueue$;
21 $\quad nextWorkQueue$.Clear();
22 **end while**

23 **function** Search(WorkItem w) **begin**
24 WorkItem x; State s;
25 TID *effTid*;
26 bool $tidenabled$, $varaccess$;
27 **if** w has no successors **then return**;
28 Thread tid := highestPriorityEnabledThread($w.prio$);
29 $s := w.state$.Execute(tid);
30 $tidenabled$:= ($tid \in enabled(s)$);
31 $varaccess$:= (tid `returned due to varaccess()`);
32 x := WorkItem(s, $w.prio$);
33 Search(x);
34 **if** ($tidenabled$ && $varaccess$) **then**
35 $\quad$ `// pre-emptive cswitch. gen a schedule`
36 $\quad$ *effTid* := effTidOfCurrentThread();
37 $\quad$ changeEffTidOfCurrentThread(*effTid*+$MaxThreads$);
38 $\quad x$:= WorkItem(s, $prio$);
39 $\quad nextWorkQueue$.Push(x);
40 **end if**
41 **end**

We instrument the target program separately for each subset of variables being tracked. For a fixed (v, t) value, the enumeration algorithm iteratively explores the schedules with context bound $0, 1, \ldots, c$ (c is the maximum desired context-bound value). While enumerating schedules for context bound $currBound$, schedules are generated for context-bound $currBound + 1$. Our algorithm is very similar to that presented in [20], with the following differences:

1. The instrumented program points include memory accesses to the variables being tracked, and not just explicit synchronization points. As we show later, variable-bounding allows us to

do this without significant increase in running times. Each instrumented program point yields control to our scheduler.

2. When a thread yields control to the scheduler (`line 29`), the address of the currently accessed variable is compared with the set of variables being tracked (variable bounding). Recall that it is possible that even though the variable access is instrumented, the accessed variable does not belong to the set of variables being tracked. This can happen either due to the imprecision of the static alias analysis or in cases where multiple variables are allocated by the same heap-allocation statement. If the accessed variable belongs to the set of variables being tracked, the priority of the executed thread is re-assigned, as discussed in Section 4.

We also instrumented all entries and exits from `synchronized` blocks, calls to `wait` and `notify`, and other thread library functions like `Thread.create`, `Thread.join`, `Thread.yield`, `Thread.suspend` and `Thread.resume`. We replace all synchronization function calls with calls to the appropriate scheduler functions, through instrumentation. The scheduler function emulates the requested operation and returns to the enumeration algorithm (at `line 29`). The enumeration algorithm then selects the highest-priority active thread (which could have changed due to the synchronization operation) and executes it. For illustration, Figure 11 shows the scheduler's emulation functions for `wait()` and `notify()`. All calls to `wait()` and `notify()` in the target program are replaced with calls to `wait_s()` and `notify_s()` respectively.

```
void wait_s(cond, mutex) {
  curthread.waitingOn = cond;
  curthread.status = BLOCKED;
  add_to_blocked_threads(curthread);
  wakeup_threads_blocked_on(mutex);
  return to scheduler
}

void signal_s(cond, mutex) {
  wakeup_threads_blocked_on(cond);
  return to scheduler
}
```

Figure 11. The scheduler's wait() and notify() functions

All program instructions, where one of the variables being tracked is accessed, are also instrumented with a call to scheduler function `varaccess()`. The `varaccess()` function simply returns to the enumeration algorithm (at `line 29`). The instructions that could potentially access a tracked variable are identified using static alias analysis.

Here, we also point out that our definition of *context-bound* differs from previous work [20] in a subtle way. While the previous work counts all pre-emptive context switches towards the context-bound, we only count the pre-emptive context switches that violate the current priority order. For example, in our scheme, it is possible for a low-priority thread to be pre-empted in favor of a high-priority thread after thread creation, even at $c = 0$. We do not count such pre-emptions towards the context-bound.

Usually, priority-based schemes suffer from issues like priority inversion and starvation. Because we require all threads to be terminating, this is not an issue in our implementation. A priority-based scheme also violates any assumptions of *strong fairness* [2] which says that every thread will eventually be run. As also noted in [21], many programs implicitly make this assumption. For example, while-flags (or spin-loops) are a common synchronization construct that assume strong fairness. These loops will never terminate if the thread that sets the condition of the loop starves. CHESS avoids this situation by assuming that a thread yields when it is not able to make progress, and assigning lower priority to threads calling `thread_yield()`. In our enumeration scheme, lowering the priority of a thread on a call to `yield()` may cause certain schedules to never get enumerated, because unlike CHESS, we enumerate only a small set of priority orders among threads (thread-bounding). To guard against the possibility of infinite loops, we lower the priority of a thread if we observe that thread to `yield()` more than a 100 times. This threshold avoids infinite loops, and yet is reasonably large to not cause interference with our thread-bounding algorithm.

Similar to CHESS [21], we use happens-before relations to construct a happens-before graph to prune the schedules. The happens-before graph characterizes the partial order of related operations in a program execution. The nodes of the happens before graph are the executed instructions. A happens-before directed edge is drawn between two instructions iff the two instructions execute in different threads, the first instruction executes before the second instruction in the given schedule, and the two instructions access the same variable of which at-least one access is a write. The pruning is based on the observation that two schedules with identical happens-before graphs result in the same program state. For a given variable set, if one schedule has an identical happens-before graph to another previously enumerated schedule, this schedule (and all its derivative schedules) need not be enumerated. Pruning is not performed across distinct variable sets. Because our thread-bounding algorithm is randomized, our exhaustive search algorithm is not strictly exhaustive. But as stated in Lemma 4.1, the probability that we have not exhausted the search space can be made arbitrarily small by executing a sufficiently large number of random priority orders.

We also implemented a randomized testing algorithm in RankChecker to test variable and thread bounding. The randomized algorithm simply picks a set of v variables (globals and heap-allocation statements) randomly, and then picks priority change points at accesses to these variables. The values of the maximum number of accesses, $k_{q_1},\ldots,k_{q_Q}$, to variables, $q_1,\ldots,q_Q$ respectively, are estimated by running the program without priority scheduling multiple times and counting the average number of accesses to each variable in these runs. The priority change points are picked uniformly over the interval $[1, k_{q_i}]$. Our randomized algorithm is modeled after PCT's depth-bounding. The only difference between our algorithm and PCT is in the assignment of priorities. PCT generates a set of random priority orders, such that each thread gets to be the lowest priority thread in at least one of the priority orders. Also, on a priority change point, PCT decreases the priority of the current thread to become lower than the priority of all currently-executing threads. Our priority orders are instead chosen using the thread-bounding algorithm given in Section 4.

6. EXPERIMENTAL RESULTS

We perform experiments to answer the following questions:

- What are the typical values of variable-bound and thread-bound in common concurrency bugs?
- What is the runtime improvement due to variable bounding?
- For exhaustive search strategy, do variable and thread bounding improve the number of executions required to expose a bug?
- For randomized search strategy, do variable and thread bounding improve the number of executions required to expose a bug?

We picked a variety of small and large Java programs and one C# program as test programs to evaluate our algorithms. The details of these programs are given in Table 1. The first 13 programs are from the ConTest Concurrency Benchmark Suite [10]. All these programs contain a concurrency bug. The next 8 benchmarks are multi-threaded Java programs commonly used to evaluate concurrency testing and verification tools. Some of these programs contain bugs. The last program (`RegionOwnership`) is a C# program containing a reasonably complex concurrency bug. This program has been previously analyzed using CHESS [9]. As we discuss later, we have also implemented variable and thread bounding in the CHESS tool to test C# programs. We report our experiences with variable bounding on the `RegionOwnership` benchmark. Within a variable and thread bound, we further rank our schedules based on the loop iteration number (recall Section 2). For exhaustive search experiments, while choosing our variable set, we give priority to shared variables. i.e., variables known to be shared are chosen before other variables. A variable is known to be shared if in one of the preparatory runs, we found a variable being accessed by at least two threads.

We ran RankChecker on the programs containing known bugs with variable bounding to check the bug characteristics. Table 1 lists the (c, v, t) values at which these bugs were uncovered using the exhaustive algorithm. We found that all of these bugs were $c \leq 2, v \leq 2, t = 2$ bugs. We surveyed past papers on studying concurrency bugs and also inspected many bugs reported in bug databases of popular applications. We found that all of these bugs were also of type $c \leq 2, v \leq 2, t = 2$.

We provide pseudo-code of the $c = 2, v = 1, t = 2$ bug found in `AllocationVector` in Figure 12.

```
                               | Block b = FindFreeBlock();
Block b = FindFreeBlock();     | first context switch
ASSERT(IsBlockFree(b));        |
MarkBlockAllocated(b);         |
second context switch          | ASSERT(IsBlockFree(b)); !FAILS!
                               | MarkBlockAllocated(b);
                               | FreeAllBlocks();
FreeAllBlocks();               |
```

Figure 12. Pseudo-code showing the $\mathbf{c = 2, v = 1, t = 2}$ bug in AllocationVector. The routines FindFreeBlock(), MarkBlockAllocated(), and IsBlockFree() are all synchronized (i.e., protected by a monitor lock). FindFreeBlock() searches a global vector to find an unallocated block. MarkBlockAllocated() sets a flag in block b and IsBlockFree() checks that flag.

We next discuss the improvements in running time due to variable bounding. Table 2 shows our results on some of our Java programs. The other Java programs were too small to show any meaningful improvements. The runtime statistics have been averaged over several runs of the programs. With variable bounding, there is up to 100x improvement in the runtime cost of instrumentation. The runtime improvement depends on the proportion of computation and I/O in the test program. Variable bounding results in improvement because only program statements identified by alias analysis as potential accesses to our set of tracked variables need to be instrumented. The performance of an instrumented run is now comparable to that of a native run, which makes it practical to implement systematic testing algorithms where all variables are considered as potential pre-emptions points. (The native run is sometimes slower than the instrumented run; this happens due to the overhead of process creation in the native run which does not exist in our instrumented run.). This is a significant improvement over previous work, where only synchronization operations have been considered as potential pre-emption points [6, 21].

Previously, a tool called RaceFuzzer [25] reported a $c = 1, v = 1, t = 2$ concurrency bug (data race) in `cache4j`. Our tool could not find this bug even after exhaustively enumerating all schedules up to $c \leq 2, v \leq 2, t = 2$. On deeper inspection, we found that the bug did not exist. It turned out that RaceFuzzer had generated a false bug report due to an error in the modelling of the semantics of the Java interrupt exception in the tool. We reported this to the author of RaceFuzzer [25], and he did not object to our findings. Because RankChecker actually runs a schedule to try and trigger assertion failures, a bug report and the associated schedule reported by it also serve as a proof of the bug's existence.

6.1 Variable and Thread Bounding in CHESS

We further validated the effectiveness of variable and thread bounding in practice by implementing it inside CHESS[21] and testing it on C# benchmarks that were previously used with CHESS [9]. However, we did not have an alias analysis readily available for C#, thus, we only implemented a simple form of variable bounding that works as follows. Let VT-CHESS refer to our extension of CHESS with variable bounding. Suppose VT-CHESS is executed on program P with variable bound v and pre-emption bound c. If $v \geq c$ then VT-CHESS behaves exactly like CHESS. When $v < c$, then during an execution of P, VT-CHESS records the shared variables accessed just before the first v pre-emptions in the execution. Subsequent pre-emptions ($v + 1^{\text{th}}$ to c^{th}) are constrained to occur only after an access of one of these v variables. In other words, the v variables for variable bounding are chosen dynamically.

The deepest reported bug found using CHESS is in a program called `RegionOwnership`. It is a C# library that manages concurrency and coordination for objects communicating via asynchronous procedure calls. The library is accompanied by a single test case comprising of a one-producer one-consumer system. The library is 1500 lines of code, and an execution access a synchronization variable at most 280 times. The test reveals a bug that requires at least 3 pre-emptions.

Table 3 shows the number of executions and time taken before VT-CHESS either reported a bug or finished exploring all behaviors under the given bound. We used $c = 3, t = 2$ in all invocations of VT-CHESS. VT-CHESS was able to find the bug about 6 times faster then CHESS while using a variable bound of 2. Using a variable bound of 1 does not expose the bug, but Table 3 shows a further reduction in search space when this bound is imposed.

Table 3. Experiments with the RegionOwnership benchmark.

	Bug found?	# Executions	Time (sec)
No VB, $t = 2$	Yes	132507	6897.3
$v = 2, t = 2$	Yes	47248	1224.4
$v = 1, t = 2$	No	30437	581.0

6.2 Variable and Thread Bounding in Randomized Algorithms

All the bugs found in our test programs, except `RegionOwnership`, were of type $v \leq 1$. As seen in Tables 2 and 3, variable bounding improves both runtime and the number of schedules explored while systematically testing concurrent programs. To further study the effect on bugs with higher v, t values, we modified one of our test programs such that it had a bug of the required type and ran RankChecker on it. Table 4 presents our results.

As expected, the time required to find the bug decreases dramatically with variable bounding. The number of executions required to find a $v = 0$ bug is roughly the same with and without variable bounding, but increases with the thread-bound of the bug. The number of executions required to find the bug improves with variable bounding at $v \geq 1$, for the reasons discussed in Section 3.

7. RELATED WORK

There is a large body of work on static [3, 22] and dynamic [8, 14, 18, 23, 24] techniques to uncover concurrency bugs. While

Table 1. Test programs and their details. The last two columns list, for each buggy program, the number of schedules explored until we found the first bug and tuple $(\mathbf{c}, \mathbf{v}, \mathbf{t})$ at which the bug occurs.

Benchmark	SLOC	# Threads	# Variables	Bug?	Description	Schedules Explored	(c, v, t)
ConTest Benchmarks							
MergeSort	376	100	564	Yes	Sorts a set of integers using mergesort	651	$(1, 1, 2)$
Producer Consumer	279	7	61	Yes	Simulates producer-consumer behavior	1	$(0, 0, 2)$
LinkedList	420	3	60	Yes	LinkedList's implementation with test-harness	23	$(1, 1, 2)$
BubbleSort	365	9	54	Yes	Sorts a set of integers using bubblesort	1	$(0, 0, 2)$
BubbleSort2	129	101	105	Yes	Sorts a set of integers using bubblesort	2	$(0, 0, 2)$
Piper	210	9	33	Yes	Manages airline reservations	64	$(1, 1, 2)$
Allocation Vector	288	3	4010	Yes	Manages free and allocated blocks in a vector	113	$(2, 1, 2)$
BufWriter	259	5	27	Yes	Reads and writes to a buffer concurrently	12	$(0, 0, 2)$
PingPong	276	18	25	Yes	Simulates the behavior of ping-pong game	234	$(1, 1, 2)$
Manager	190	6	25	Yes	Manages free and allocated blocks	33	$(1, 1, 2)$
MergeSortBug	258	29	52	Yes	Sorts a set of integers using mergesort	89	$(1, 1, 2)$
Account	169	3	26	Yes	Manages a bank account	19	$(1, 1, 2)$
AirLineTickets	99	11	5	Yes	Simulates selling of airline tickets	2	$(0, 0, 2)$
Java's Library in JDK 1.4.2							
HashSet	7086	200	4777	Yes	Thread-safe implementation of HashSet	813	$(1, 1, 2)$
TreeSet	7532	200	6140	Yes	Thread-safe implementation of TreeSet	813	$(1, 1, 2)$
Other Java Benchmarks							
Cache4j	3897	12	251,469	No	Cache implementation for Java objects	-	-
Molydn	1410	8	121,371	No	Benchmark from Java Grande Forum	-	-
Montecarlo	3630	8	452,700	No	Benchmark from Java Grande Forum	-	-
TSP	719	18	84	No	Travelling Sales Problem's implementation	-	-
Blocking Queue	57	3	38,828	No	Tests BlockingQueue library implementation	-	-
Sor	17,738	6	53	No	Successive Order Relaxation method's implementation	-	-
C# Benchmark							
RegionOwnership	1500	5	41	Yes	Manages coordination for objects communicating using async calls	47248	$(2, 2, 2)$

Table 2. The different columns in this table represents the name of the program, the (average) number of byte code instructions executed by the program, total number of different instrumentation sites, which includes heap-allocation statements and global variables, total number of accesses, native execution time, the average amount of time taken for one execution when we are tracking 0, 1, 2, and all variables, respectively, and the last column represents the ratio of the v-all and v1 columns.

Program Name	BCI	var sites	# of accesses	Native time(sec)	v0(sec)	v1(sec)	v2(sec)	v-all(sec)	v-all/v1
Cache4j	231.1m	101	21.4m	0.34	0.47	1.23	2.76	26.38	21.3
Molydn	2.33b	209	1.4b	0.39	3.15	11.59	19.86	1239.76	106.3
Montecarlo	577.7m	235	446.96m	0.48	1.94	4.74	5.21	323.12	68.08
TSP	8.76b	65	2.55b	4.2	4.23	32.64	109.72	1180.28	36.15
Blocking Queue	3.4m	13	0.65m	0.17	0.18	0.194	0.202	1.386	7.14
Sor	0.2m	46	0.68m	0.07	0.25	0.249	0.348	0.392	1.57
HashSet	157.4k	137	16889	0.07	0.0775	0.0901	0.0976	0.2687	2.98
TreeSet	113k	146	16273	0.69	0.078	0.089	0.09	0.259	2.91

Table 4. This table represents the average number of executions and time required to capture a bug of type $(\mathbf{c}, \mathbf{v}, \mathbf{t})$ introduced in the Montecarlo benchmark. Without variable bounding, our tool timed out after executing for more than 3 days for $\mathbf{c} \geq \mathbf{1}, \mathbf{v} \geq \mathbf{1}$, without finding the bug.

Bug Type	With v,t bounding		Without v,t bounding	
(c, v, t)	# Executions	Time (sec)	# Executions	Time (sec)
$(0, 0, 2)$	3.1	10.9	2.7	768.9
$(0, 0, 3)$	3.7	12.1	3.5	1010.7
$(0, 0, 4)$	4.9	14.9	3.8	1101.7
$(1, 1, 2)$	1636.2	6409.9	-	TimedOut
$(1, 1, 3)$	4889	20371.2	-	TimedOut
$(2, 1, 2)$	28121	112950.3	-	TimedOut

many of these techniques are very effective and have uncovered a variety of previously-unknown bugs in well-tested software, the current dominant practice in the software industry still remains *stress testing*. There are a few important likely reasons for this (apart from plain inertia):

1. Plain testing is more natural.
2. Most tools target a small class of bugs. For example, some tools target only dataraces, others target only atomicity-violation bugs, and yet others target only deadlocks. It is confusing for a developer to understand the function of each tool and apply them separately.
3. Many tools have false positives. Spending time and energy on a false-positive bug report is annoying and counter-productive.
4. Many tools require source-level annotations. Many tools rely on certain programming disciplines. e.g., all shared memory accesses must be protected by a lock. At places where the programmer deliberately violates this discipline, source code annotations are required. Most developers are usually reluctant to annotate their source code for better testing.
5. High-Runtime, Low Coverage: Many tools have a high runtime cost, and provide low coverage.

Model checking approaches [5, 11, 12, 15, 21, 26] are closer to the familiar idiom of testing. Our model checker targets all types of concurrency bugs, has no false positives, and requires no source-level annotations. We address the state explosion problem by ranking the schedules using v, t to maximize coverage in the first few schedule executions. Previous approaches have reduced this search space by either limiting context switches only at synchronization operations [21] (resulting in potential false negatives), or using an offline memory trace of the program to identify and rank unserializable interleavings [23] (primarily to identify atomicity-violation bugs). We believe that variable and thread bounding are more general methods of ranking (or reducing) the search space.

CHESS [21] uses iterative context bounding to rank schedules. We borrow many ideas from CHESS, including iterative context bounding [20], using a happens-before graph for stateless model checking, and fair scheduling. We provide further ranking of schedules to uncover most bugs with a smaller number of schedules. While we consider all shared memory accesses as potential context-switch points, CHESS only allows pre-emptible context switches at explicit synchronization primitives. This restriction (first used in ExitBlock [5]) is justified if all shared memory accesses are protected by explicit synchronization (e.g., `lock/unlock`). CHESS relies on a data-race detector to separately check this property. Even if we assume a precise and efficient data-race detector, this approach still overlooks "adhoc" synchronization that do not involve known synchronization primitives [28].

VeriSoft [11–13] also uses an exploration strategy to model-check "distributed" systems using a *state-less search* (i.e., storage of previously-visited states are not required). Verisoft uses partial-order methods to reduce redundancy, similar to happens-before graph pruning used in CHESS and in our tool. S. Stoller [26] uses a similar approach to model-check multi-threaded distributed Java programs. We believe that variable and thread bounding ideas are equally relevant to these model checking approaches as well.

Our work is complementary to race-detection tools [19, 22, 24, 25, 29], deadlock-detection tools [14], and atomicity-violation detection tools [18, 23]. We do not focus on a particular class of bugs, but rather drive a model checker into exploring interesting schedules that are likely to uncover all of these bugs early. In practice, small v, t values uncover most data races, deadlocks, and atomicity-violation bugs.

8. CONCLUSION

We present variable and thread bounding to rank thread schedules for systematic testing of concurrent programs. Through experiments on a variety of Java and C# programs, we find that the ranking significantly aids early discovery of common bugs.

9. ACKNOWLEDGEMENTS

We thank Sandeep Sen (I.I.T. Delhi) for graciously giving us Lemma 4.1 and its proof. The first two authors acknowledge partial support from NetApp's research grant. The first author acknowledges Microsoft Research India for travel grant.

10. REFERENCES

[1] L. O. Andersen. Program analysis and specialization for the C programming language. Technical report, 1994.

[2] K. R. Apt, N. Francez, and S. Katz. Appraising fairness in distributed languages. In *POPL '87*.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

[4] S. Bindal, S. Bansal, and A. Lal. Variable and thread bounding for systematic testing of multithreaded programs. Technical report, IIT Delhi, 2012. `http://arxiv.org/abs/1207.2544`.

[5] D. Bruening and J. Chapin. Systematic testing of multithreaded programs. Technical report, LCS-TM-607, MIT/LCS, 2000.

[6] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS '10*.

[7] S. Chiba. Javassist — a reflection-based programming wizard for java. In *OOPSLA '98*.

[8] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07*.

[9] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL*, pages 411–422, 2011.

[10] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Toward a framework and benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice & Experience*, 19(3):267–279, 2007.

[11] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97*.

[12] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[13] P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model checking without a model: an analysis of the heart-beat monitor of a telephone switch using VeriSoft. In *ISSTA '98*.

[14] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09*.

[15] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03*.

[16] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18:3:1–3:53, October 2008.

[17] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS '08*.

[18] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS '06*.

[19] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *PLDI '09*.

[20] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07*.

[21] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. In *OSDI '08*.

[22] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06*.

[23] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS '09*.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[25] K. Sen. Race directed random testing of concurrent programs. In *PLDI '08*.

[26] S. D. Stoller. Model-checking multi-threaded distributed java programs. In *SPIN Workshop on Model Checking and Software Verification*, 2000.

[27] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04*.

[28] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI '10*.

[29] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05*.