# A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Pravesh Kothari
Indian Institute of Technology, Kanpur
praveshk@iitk.ac.in

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Santosh Nagarakatte
University of Pennsylvania
santoshn@cis.upenn.edu

## Abstract

This paper presents a randomized scheduler for finding concurrency bugs. Like current stress-testing methods, it repeatedly runs a given test program with supplied inputs. However, it improves on stress-testing by finding buggy schedules more effectively and by quantifying the probability of missing concurrency bugs. Key to its design is the characterization of the depth of a concurrency bug as the minimum number of scheduling constraints required to find it. In a single run of a program with $n$ threads and $k$ steps, our scheduler detects a concurrency bug of depth $d$ with probability at least $1/nk^{d-1}$. We hypothesize that in practice, many concurrency bugs (including well-known types such as ordering errors, atomicity violations, and deadlocks) have small bug-depths, and we confirm the efficiency of our schedule randomization by detecting previously unknown and known concurrency bugs in several production-scale concurrent programs.

***Categories and Subject Descriptors*** D [*2*]: 5

***General Terms*** Algorithms, Reliability, Verification

***Keywords*** Concurrency, Race Conditions, Randomized Algorithms, Testing

## 1. Introduction

Concurrent programming is known to be error prone. Concurrency bugs can be hard to find and are notorious for hiding in rare thread schedules. The goal of concurrency testing is to swiftly identify and exercise these buggy schedules from the astronomically large number of possible schedules. Popular testing methods involve various forms of *stress testing* where the program is run for days or even weeks under heavy loads with the hope of hitting buggy schedules. This is a slow and expensive process. Moreover, any bugs found are hard to reproduce and debug.

In this paper, we present PCT (Probabilistic Concurrency Testing), a randomized algorithm for concurrency testing. Given a concurrent program and an input test harness, PCT randomly schedules the threads of the program during each test run. In contrast to prior randomized testing techniques, PCT uses randomization sparingly and in a disciplined manner. As a result, PCT provides efficient mathematical probability of finding a concurrency bug in each run. Repeated independent runs can increase the probability of finding bugs to any user-desired level of certainty. In this paper, we demonstrate the ability of PCT to find bugs both theoretically, by stating and proving the probabilistic guarantees, and empirically, by applying PCT to several production-scale concurrent programs.

At the outset, it may seem impossible to provide effective probabilistic guarantees without exercising an astronomical number of schedules. Let us take a program with $n$ threads that together execute at most $k$ instructions. This program, to the first-order of approximation, has $n^k$ possible thread schedules. If an adversary picks any one of these schedules to be the only buggy schedule, then no randomized scheduler can find the bug with a probability greater than $1/n^k$. Given that realistic programs create tens of threads ($n$) and can execute millions, if not billions, of instructions ($k$), such a bound is not useful.

PCT relies on the crucial observation that bugs in practice are not adversarial. Concurrency bugs typically involve unexpected interactions among few instructions executed by a small number of threads [19, 20]. If PCT is able to schedule these few instructions correctly, it succeeds in finding the bug irrespective of the numerous ways it can schedule instructions irrelevant to the bug. The following characterization of concurrency bugs captures this intuition precisely.

We define the *depth* of a concurrency bug as the minimum number of scheduling constraints that are sufficient to find the bug. Intuitively, bugs with a higher depth exhibit in fewer schedules and are thus inherently harder to find. Fig. 1 explains this through a series of examples. The bug in Fig. 1(a) manifests whenever Thread 2 accesses `t` before the initialization by Thread 1. We graphically represent this *ordering constraint* as an arrow. Any schedule that satisfies this ordering constraint finds the bug irrespective of the ordering of other instructions in the program. By our definition, this bug is of depth 1. Fig. 1 shows two more examples of common concurrency errors, an atomicity violation in (b) and a deadlock in (c). Both these errors require two ordering constraints and are thus of depth 2.



*ASPLOS'10,* March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

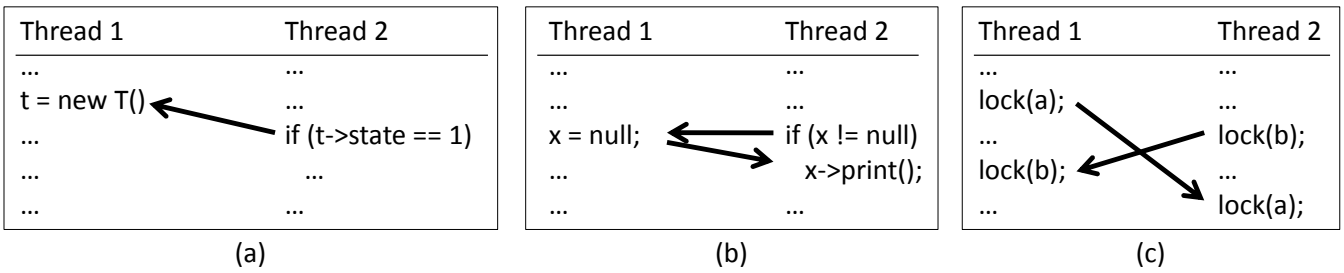


**Figure 1.** Three typical concurrency bugs, and ordering edges sufficient to find each. (A) This ordering bug manifests whenever the test by thread 2 is executed before the initialization by thread 1. (B) This atomicity violation manifests whenever the test by thread 2 executed before the assignment by thread 1, and the latter is executed before the method call by thread 2. (C) This deadlock manifests whenever thread 1 locks a before thread 2, and thread 2 locks b before thread 1.

On each test run, PCT focuses on probabilistically finding bugs of a particular[1] depth $d$. PCT is a priority-based scheduler and schedules the runnable thread with the highest priority at each scheduling step. Priorities are determined as follows. On thread creation, PCT assigns a random priority to the created thread. Additionally, PCT changes thread priorities at $d-1$ randomly chosen steps during the execution.

These few but carefully designed random choices are provably effective for finding concurrency bugs of low depth. Specifically, when run on a program that creates at most $n$ threads and executes at most $k$ instructions, PCT finds a bug of depth $d$ with a probability of at least $1/nk^{d-1}$. For small $d$, this bound is much better than the adversarial bound $1/n^k$. In particular, for the cases $d = 1$ and $d = 2$ (which cover all examples in Fig. 1), the probability for finding the bug in each run is at least $1/n$ and $1/nk$, respectively.[2]

We describe the randomized algorithm informally in Section 2 and the formal treatment with the proof of the bound in Section 3. As described above, the scheduler is simple and can readily be implemented on large systems without knowledge of the proof mechanics. Note that the proof was instrumental to the design of PCT because it provided the insight on how to use randomization sparingly, yet effectively.

The probabilistic bound implies that on average, one can expect to find a bug of depth $d$ within $nk^{d-1}$ independent runs of PCT. As our experiments show (Section 5), PCT finds depth 1 bugs in the first few runs of the program. These bugs are certainly not trivial and were discovered by prior state-of-art research tools [20, 24] in well-tested real-world programs.

Our implementation of PCT, described in Section 4, works on executables compiled from C/C++ programs. In addition to the base algorithm described in Section 2, our implementation employs various optimizations including one that reduces $k$ to the maximum number of synchronization operations (rather than the number of instructions) performed in any run for a given test input.

We evaluate PCT on six benchmarks from prior work [21, 24] that contain known concurrency bugs. This includes the open source program PBZIP2 [24], three SPLASH2 benchmarks [24], an implementation of a work stealing queue [21], and Dryad Channels [21]. PCT finds all of the known bugs much faster than respectively reported in prior work. We also find two *new* bugs that were missed by prior work in these benchmarks. To test our scalability, we ran PCT on unmodified recently-shipped versions of two popular web browsers Mozilla and Internet Explorer. We find one previously *unknown* bug in each of them. Finally, we empirically demonstrate that PCT often detects bugs with probabilities greater than the theoretical worst-case bound.

[1] For exposition, we assume that the bug depth parameter $d$ is provided as an input by the user. In practice, our tool chooses $d$ automatically from an appropriate random distribution.

[2] In theory, $d$ can be as large as $k$. In this case, our bound (as required) is worse than $1/n^k$. However, we consider a depth $k$ bug a practical impossibility, especially for modern software that is built from a large number of loosely-coupled components.

## 2. PCT Overview

In this section we provide necessary background and an informal description of our algorithm.

### 2.1 Concurrency Testing

The general problem of testing a program involves many steps. In this paper, we focus on concurrency testing. We define a *concurrency bug* as one that manifests on a strict subset of possible schedules. Bugs that manifest in all schedules are not concurrency bugs. The problem of concurrency testing is to find schedules that can trigger these bugs among the vast number of potential schedules.

We assume that inputs to our program are already provided, and the only challenge is to find buggy schedules for that input. Determining bug-triggering inputs for concurrent programs is a challenging open problem beyond the scope of this paper. Our assumption is validated by the fact that there already exists large suites of stress tests carefully constructed by programmers over the years.

### 2.2 State of the Art

We identify the following basic strategies for flushing out concurrency bugs. We describe them in detail in Section 6.

**Stress Testing** relies on repetition and heavy load to find bug-triggering schedules by chance. The schedules explored are not uniformly distributed and are determined by the chaotic noise in the system.

**Heuristic-Directed Testing** improves upon stress testing by using runtime monitors and heuristics to (1) detect suspicious activity in a program (such as variable access patterns that indicate potential atomicity violations [24], or lock acquisition orderings that indicate potential deadlocks [15]), and (2) direct the schedule towards suspected bugs.

**Systematic Scheduling** controls the scheduler to systematically enumerates possible schedules either exhaustively or within some bound (such as a bound on the number of preemptions) [21].

**Randomized Scheduling** is similar to stress testing, but attempts to amplify the 'randomness' of the OS scheduler[1]. It can

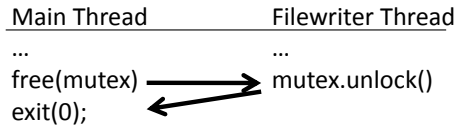


**Figure 2.** An example of a bug of depth 2 we found in PBZIP2. The bug surfaces if (1) the mutex is unlocked after it is freed, and (2) the mutex is unlocked before the main thread terminates the process by calling exit.

do so by inserting random delays, context switches, or thread priority changes.

PCT falls in the last category. But unlike all the methods above, PCT provides a guaranteed probability of finding bugs in every run of the program. Our experiments validate this guarantee. Note that PCT is orthogonal to heuristic-directed testing methods above, in the sense that the analysis used in these methods can be used to further improve PCT.

### 2.3 Bug Depth

We classify concurrency bugs according to a *depth* metric. Intuitively, deeper bugs are inherently harder to find. PCT is designed to provide better guarantees for bugs with smaller depth.

Concurrency bugs happen when instructions are scheduled in an order not envisioned by the programmer. We identify a set of these *ordering constraints* between instructions from different threads that are sufficient to trigger the bug. It is possible for different sets of ordering constraints to trigger the same bug. In such a case, we focus on the set with lesser number of constraints. We define the depth of a concurrency bug as the minimum number of ordering constraints sufficient to find the bug.

For example, Fig. 1 shows examples of common concurrency errors with ordering constraints, represented by arrows, that are sufficient to find the bug. Any schedule that satisfies these ordering constraints is guaranteed to find the bug irrespective of how it schedules instructions not relevant to the bug. For the examples in Fig. 1 the depth respectively is 1, 2, and 2. We expect many concurrency bugs to have small depths. This is further validated by our experimental results.

#### 2.3.1 Relationship with Prior Classification

Fig. 1 also demonstrates how previous classifications of concurrency bugs correspond to bugs of low depth. For example, ordering bugs [19] have depth 1, atomicity violations and non-serializable interleavings [24], in general, have depth 2, and deadlocks caused by circular lock acquisition [15] have depth 2, or more generally $n$ if $n$ threads are involved. However, this classification is not strict. For instance, not all atomicity violations have a depth 2, and in fact, three of the bugs reported by prior work as atomicity violations [24] have a depth 1.

However, our notion of bug depth is more general and can capture concurrency bugs not classified before. Fig. 2 shows an example of a bug of depth 2 that does not directly fall into any of the mentioned categories. In particular, the ordering constraints do not have to be between instructions that access the same variable.

Another characterization of a concurrency bug is its *preemption bound* [20]. A preemption bound is the smallest number of preemptions sufficient to find a concurrency bug. For the examples shown in Fig. 1, $d-1$ preemptions are sufficient to find the bugs. However, that is not always the case, as shown in the example in Fig. 3 where $d = 1$, yet at least one preemption is required to find the bug.

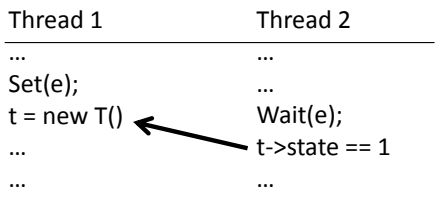


**Figure 3.** A variation of the example in Fig. 1(a), with the same bug depth of $d = 1$. Unlike in the other example, however, this bug requires Thread 1 to be preempted right after the instruction that sets the event `e`, and thus has a preemption bound of 1.

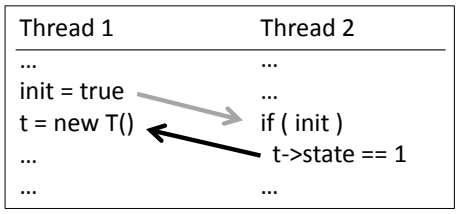


**Figure 4.** Although it may seem like one constraint (black arrow) is sufficient to find this bug, an extra constraint (gray arrow) is needed to ensure that thread 2 really executes the access of `t`. Thus, the depth of this bug is 2.

#### 2.3.2 Interaction with Control Flow

Fig. 4 shows a slight modification to Fig. 1(a). In this example, the program (incorrectly) maintains a Boolean variable `init` to indicate whether `t` is initialized or not. Now, the single ordering constraint (black arrow) between the initialization and access of `t` is not sufficient to find the bug. The scheduler should also ensure the right ordering constraint between `init` accesses (grey arrow). Thus, the presence of control flow increases the bug depth to 2.

This example brings out two interesting points. First, the notion of bug depth is inherently tied to the difficulty of the concurrency bug. Fig. 4 is arguably a more subtle bug than Fig. 1(a). Second, in a program with complex control flow, the depth of a bug might not be readily apparent to the programmer. However, our technique does not require the programmer or a prior program analysis to identify these constraints explicitly. It relies on the mere *existence* of the right number of ordering constraints.

### 2.4 Naive Randomization

Using a randomized scheduler may appear like an obvious choice. However, it is not a priori clear how to design such a scheduler with a good detection probability for concurrency bugs. For illustration purposes, let us consider the simple case of a program shown in Fig. 5 with two threads containing a bug of depth 1, shown by the black arrow. (Neglect the grey arrow for now.) Even this simple bug can frustrate a naive randomization technique.

Consider a naive randomized scheduler that flips a coin in each step to decide which thread to schedule next. This scheduler is unlikely to detect the bug in Fig. 5 even though its depth is only 1. To force the black constraint, this scheduler has to consistently schedule Thread 1 for $m + 2$ steps, resulting in a probability that is inverse exponential in $m$ — a small quantity even for moderate $m$. One could then try to improve this scheduler by biasing the coin towards Thread 1 to increase the likelihood of hitting this bug. This still contains an exponential in $m$. But more importantly, any

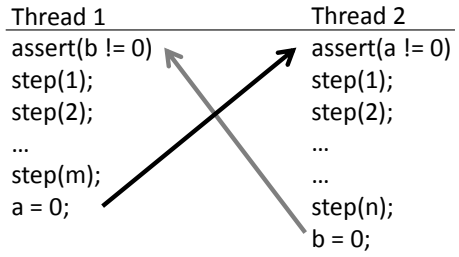


**Figure 5.** A program with two bugs of depth 1 that are hard to find with naive randomized schedulers that flip a coin in each step. PCT finds both these bugs with a probability $1/2$.

bias towards the black constraint, will be equally biased *against* the second bug represented by the grey constraint.

In contrast, our PCT scheduler will find both bugs (and all other bugs with depth 1) with probability 1/2 for this program.

### 2.5 The PCT Algorithm

We now describe our key contribution, a randomized scheduler that detects bugs of depth $d$ with a guaranteed probability in every run of the program. Our scheduler is priority-based. The scheduler maintains a priority for every thread, where lower numbers indicate lower priorities. During execution, the scheduler schedules a low priority thread only when all higher priority threads are blocked. Only one thread is scheduled to execute in each step. A thread can get blocked if it is waiting for a resource, such as a lock that is currently held by another thread.

Threads can change priorities during execution when they pass a *priority change point*. Each such point is a step in the dynamic execution and has a predetermined priority value associated with it. When the execution reaches a change point, the scheduler changes the priority of the current thread to the priority value associated with the change point.

Given inputs $n$, $k$, and $d$, PCT works as follows.

1. Assign the $n$ priority values $d, d+1, \ldots, d+n$ randomly to the $n$ threads (we reserve the lower priority values $1, \ldots, (d-1)$ for change points).
2. Pick $d-1$ random priority change points $k_1, \ldots, k_{d-1}$ in the range $[1, k]$. Each $k_i$ has an associated priority value of $i$.
3. Schedule the threads by honoring their priorities. When a thread reaches the $i$-th change point (that is, when it executes the $k_i$-th step of the execution), change the priority of that thread to $i$.

This randomized scheduler provides the following guarantee.

> Given a program that creates at most $n$ threads and executes at most $k$ instructions, PCT finds a bug of depth $d$ with probability at least $1/nk^{d-1}$.

### 2.6 Intuition Behind the Algorithm

See Fig. 6 for an illustration of how our algorithm finds the errors in Fig. 1. This figure shows the initial thread priorities in white circles and the priority change points in black circles. To understand the working of the scheduler, observe that a high priority thread runs faster than a low priority thread. So, barring priority inversion issues, an ordering constraint $a \rightarrow b$ is satisfied if $a$ is executed by a higher priority thread. In Fig. 6(a), the bug is found if PCT chooses a lower priority for Thread 1 than Thread 2. The probability of this is $1/2$ and thus PCT is expected to find this bug within the first two runs.

If there are more than two threads in the program in Fig. 6(a), then the algorithm has to work harder because of priority inversion issues. Even if Thread 1 has a lower priority than Thread 2, the latter can be blocked on a resource held by another thread, say Thread 3. If Thread 3 has a priority lower than Thread 1, then this priority inversion can allow Thread 1 to execute the initialization before Thread 2 reads `t`. However, such a priority inversion cannot happen if Thread 1 has the *lowest* priority of all threads in the program. The probability of this happening is $1/n$ which is our guarantee.

For bugs with depth greater than 1, we need to understand the effects of priority change points. (Our algorithm does not introduce priority change points when $d = 1$.) In Fig. 6(b), the atomicity violation is induced if PCT inserts a priority change point after the null check but before executing the branch. The probability of this is $1/k$ as PCT will pick the change point uniformly over all dynamic instructions. In addition, PCT needs to ensure the first constraint by running Thread 1 with lowest priority till Thread 2 does the null check. Together, the probability of finding this atomicity violation is at least $1/nk$.

The same argument holds for the deadlock in Fig. 6(c). PCT has to insert a priority change point after Thread 1 acquires the first lock before acquiring the second. The probabilistic guarantee of our algorithm with multiple priority change points in the presence of arbitrary synchronizations and control flow in the program, and issues of priority inversion is not readily apparent from the discussion above. Section 3 provides a formal proof that accounts for all these complications.

#### 2.6.1 Worst-case vs. actual probability

The probabilistic guarantee provided by PCT is a *worst-case* bound. In other words, for any program that an adversary might pick, and for any bug of depth $d$ in that program, PCT is guaranteed to find the bug with a probability not less than $1/nk^{d-1}$. This bound is also tight. There exists programs, such as the one in Fig. 4, for which PCT can do no better than this bound.

In practice, our experiments (discussed in Section 5) show that PCT often detects bugs with much better probability than the worst-case guarantee. The reason is that there are often multiple independent ways to find the same bug, and the probability of these adds up. More specifically, consider the following 3 example scenarios.

1. Sometimes it is good enough for priority change points to fall within some range of instructions. For example, Thread 1 in Fig. 6(c) may perform lots of instructions between the two acquires. PCT will find the deadlock if it picks any one of them to be a priority change point.
2. Sometimes a bug can be found in different ways. For instance, in Fig. 6(c), there exists a symmetric case in which PCT inserts a priority change point in Thread 2.
3. Sometimes a buggy code fragment is repeated many times in a test, by the same thread or by different threads, and thus offers multiple opportunities to trigger the bug.

## 3. Algorithm

In this section, we build a formal foundation for describing our scheduler and prove its probabilistic guarantees.

### 3.1 Definitions

We briefly recount some standard notation for operations on sequences. Let $T$ be any set. Define $T^*$ to be the set of finite sequences of elements from $T$. For a sequence $S \in T^*$, define $length(S)$ to be the length of the sequence. We let $\epsilon$ denote the sequence of length 0. For a sequence $S \in T^*$ and a number $n$

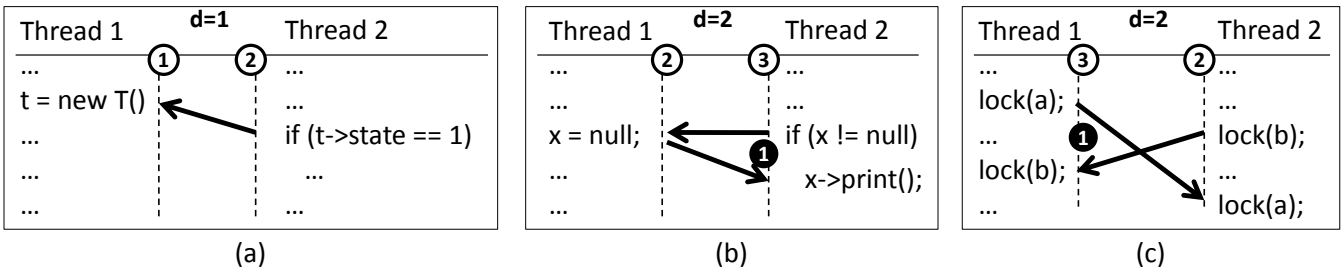


**Figure 6.** Illustration on how our randomized scheduler finds bugs of depth $d$, using the examples from Fig. 1. The scheduler assigns random initial thread priorities $\{d, \ldots, d+n-1\}$ (white circles) and randomly places $d-1$ priority change points of values $\{1, \ldots, d-1\}$ (black circles) into the execution. The bug is found if the scheduler happens to make the random choices shown above.

**Require:** program $P$, $d \geq 0$
**Require:** $n \geq maxthreads(P)$, $k \geq maxsteps(P)$
**Require:** **random variables** $k_1, \ldots, k_{d-1} \in \{1, \ldots, k\}$
**Require:** **random variable** $\pi \in Permutations(n)$

```
1:  procedure RandS(n, k, d) begin
2:      var S : schedule
3:      var p : array[n] of ℕ
4:      S ← ϵ
        // set initial priorities
5:      for all t ∈ {1, ..., n} do
6:          p[t] ← d + π(t) − 1
7:      end for
8:      while en_P(S) ≠ ∅ do
9:          /* schedule thread of maximal priority */
10:         t ← element of en_P(S) such that p[t] maximal
11:         S ← S t
            /* are we at priority change point? */
12:         for all i ∈ {1, ..., d − 1} do
13:             if length(S) = k_i then
14:                 p[t] = d − i
15:             end if
16:         end for
17:     end while
18:     return S
19: end
```

**Figure 7.** The randomized scheduler.

such that $0 \leq n < length(S)$, let $S[n]$ be the $n$-th element of $S$ (where counting starts with 0). For $t \in T$ and $S \in T^*$, we write $t \in S$ as a shorthand for $\exists m : S[m] = t$. For any $S \subset T^*$ and for any $n, m$ such that $0 \leq n \leq m \leq length(S)$, let $S[n, m]$ be the contiguous subsequence of $S$ starting at position $n$ and ending at (and including) position $m$. For two sequences $S_1, S_2 \in T^*$, we let $S_1 S_2$ denote the concatenation as usual. We do not distinguish between sequences of length one and the respective element. We call a sequence $S_1 \in T^*$ a *prefix* of a sequence $S \in T^*$ if there exists a sequence $S_2 \in T^*$ such that $S = S_1 S_2$. A set of sequences $P \subseteq T^*$ is called *prefix-closed* if for any $S \in P$, all prefixes of $P$ are also in $P$.

DEFINITION 1. *Define $T = \mathbb{N}$ to be the set of thread identifiers. Define $Sched = T^*$ to be the set of all schedules. Define a* program *to be a prefix-closed subset of $Sched$. For a given program $P \subseteq Sched$, we say a schedule $S \in P$ is* complete *if it is not the prefix of any schedule in $P$ beside itself, and* partial *otherwise.*

Thus, we represent a program abstractly by its schedules, and each schedule is simply a sequence of thread identifiers. For example, the sequence 1 2 2 1 represents the schedule where thread 1 takes one step, followed by two steps by thread 2, followed by another step of thread 1. We think of schedules as an abstract representation of the program state. Not all threads can be scheduled from all states, as they may be blocked. We say a thread is enabled in a state if it can be scheduled from that state.

DEFINITION 2. *Let $P \subseteq Sched$ be a program. For a schedule $S \in P$, define $en_P(S)$ to be the set $\{t \in T \mid S\,t \in P\}$. Define $maxsteps(P) = \max\{length(S) \mid S \in P\}$ and $maxthreads(P) = \max\{S[i] \mid S \in P\}$ (or $\infty$ if unbounded).*

Finally, we represent a concurrency bug abstractly as the set of schedules that find it:

DEFINITION 3. *Let $P \subseteq Sched$ be a program. Define a* bug *$B$ of $P$ to be a subset $B \subset P$.*

### 3.2 The Algorithm

We now introduce the randomized scheduler (Fig. 7). It operates as described informally in Section 2.5. We expect $RandS(n, k, d)$ to be called with a conservative estimate for $n$ (number of threads) and $k$ (number of steps). During the progress of the algorithm, we store the current schedule in the variable $S$, and the current thread priorities in an array $p$ of size $n$. The thread priorities are initially assigned random values (chosen by the random permutation $\pi$). In each iteration, we pick an enabled thread of maximal priority $t$ and schedule it for one step. Then we check if we have reached a priority change point (determined by the random values $k_i$), and if so, we change the priority of $t$ accordingly. This process repeats until no more threads are enabled (that is, we have reached a deadlock or the program has terminated).

### 3.3 Probabilistic Coverage Guarantee

In this section, we precisely state and then prove the probabilistic coverage guarantees for our randomized scheduler, in three steps. First, we introduce a general mechanism for identifying dynamic events in threads, which is a necessary prerequisite for defining ordering constraints on such events. Next, we build on that basis to define the depth of a bug as the minimum number of ordering constraints on thread events that will reliably reveal the bug. Finally, we state and prove the core theorem.

#### 3.3.1 Event Labeling

The first problem is to clarify how we define the events that participate in the ordering constraints. For this purpose, we introduce

a general definition of event labeling. Event labels must be unique within each execution, but may vary across executions. Essentially, an event labeling $E$ defines a set of labels $L_E$ (where each label $a \in L_E$ belongs to a particular thread $thread_E(a)$) and a function $next_E(S,t)$ that tells us what label (if any) the thread $t$ is going to emit if scheduled next after schedule $S$. More formally, we define:

DEFINITION 4. *Let $P$ be a program. An* event labeling *$E$ is a triple $(L_E, thread_E, next_E)$ where $L_E$ is a set of labels, $thread_E$ is a function $L_E \to T$, and $next_E$ is a function $P \times T \to (L_E \cup \{\bot\})$, such that the following conditions are satisfied:*

1. *(Affinity) If $next_E(S,t) = a$ for some $a \in L_E$, then $thread_E(a) = t$.*
2. *(Stability) If $next_E(S,t) = a$ for some $a \in L_E$, and if $t \neq t'$, then $next_E(S\,t', t) = a$.*
3. *(Uniqueness) If $next_E(S_1,t) = next_E(S_1 S_2, t) = a$ for some $a \in L_E$, then $t \notin S_2$.*
4. *(NotFirst) $next_E(\epsilon, t) = \bot$ for all $t \in T$.*

Sometimes, we would like to talk about labels that have already been emitted in a schedule. For this purpose we define the auxiliary functions $label_E$ and $labels_E$ as follows. For $S \in P$ and $0 \leq m < length(S)$, we define $label_E(S,m) = a$ if the label $a$ is being emitted at position $m$, and we define $labels_E(S)$ to be the set of all labels emitted in $S$ (more formally, $label_E(S,m) = a$ if there exists $k < m$ and an $a \in L_E$ such that $next_E(S[0,k], S[m]) = a$ and $S[m] \notin S[k+1, m-1]$, and $label_E(S,m) = \bot$ otherwise; and $labels_E(S) = \{label_E(S,m) \mid 0 \leq m < length(S)\}$).

### 3.3.2 Bug Depth

We now formalize the notion of 'ordering constraints' and 'bug depth' that we motivated earlier. Compared to our informal introduction from Section 2.3, there are two variations worth mentioning. First, we generalize each edge constraint $(a,b)$ (where $a$ and $b$ are event labels) to allow multiple sources $(A,b)$, where $A$ is a set of labels all of which have to be scheduled before $b$ to satisfy the constraint. Second, because we are using dynamically generated labels as our events, we require that the ordering constraints are sufficient to guide the scheduler to the bug without needing to know about additional constraints implied by the program structure (as motivated by the example in Fig. 4).

We formulate the notion of a *directive* $D$ of size $d$, which consists of a labeling and $d$ constraints. The idea is that a directive can guide a schedule towards a bug, and that the depth of a bug is defined as the minimal size of a directive that is guaranteed to find it.

DEFINITION 5. *For some $d \geq 1$, a directive $D$ for a program $P$ is a tuple $(E, A_1, b_1, A_2, b_2, \ldots, A_d, b_d)$ where $E$ is an event labeling for $P$, where $A_1, \ldots, A_d \subseteq L_E$ are sets of labels, and where $b_1, \ldots b_d \in L_E$ are labels that are pairwise distinct ($b_i \neq b_j$ for $i \neq j$). The* size *of $D$ is $d$ and is denoted by $size(D)$.*

DEFINITION 6. *Let $P$ be a program and let $D$ be a directive for $P$. We say a schedule $S \in P$* violates *the directive $D$ if either (1) there exists an $i \in \{1, \ldots, d\}$ and an $a \in A_i$ such that $b_i \in labels_E(S)$, but $a \notin labels_E(S)$, or (2) there exist $1 \leq i < j \leq d$ such that $b_j \in labels_E(S)$, but $b_i \notin labels_E(S)$. We say a schedule $S \in P$* satisfies *$D$ if it does not violate $D$, and if $b_i \in labels_E(S)$ for all $1 \leq i \leq d$.*

DEFINITION 7. *Let $P$ be a program, $B$ be a bug of $P$, and $D$ be a directive for $P$. We say $D$* guarantees *$B$ if and only if the following conditions are satisfied:*

1. *For any partial schedule $S \in P$ that does not violate $D$, there exists a thread $t \in en_P(S)$ such that $S\,t$ does not violate $D$.*
2. *Any complete schedule $S$ that does not violate $D$ does satisfy $D$ and is in $B$.*

**Require:** program $P$, $d \geq 0$
**Require:** $n >= maxthreads(P)$
**Require:** $k_1, \ldots, k_{d-1} \geq 1$
**Require:** $\pi \in Permutations(n)$
**Require:** **random variables** $k_1, \ldots, k_{d-1} \in \{1, \ldots, k\}$
**Require:** **random variable** $\pi \in Permutations(n)$
**Require:** bug $B$
**Require:** directive $D = (E, A_1, b_1, \ldots, A_d, b_d)$ for $B$

```
1:  procedure DirS(n, k, d, D) begin
2:      var S : schedule
3:      var p : array[n] of ℕ
4:      S ← ϵ
        // set initial priorities
5:      for all t ∈ {1, ..., n} do
6:          p[t] ← d + π(t) − 1
7:      end for
8:      [ assert: p[thread_E(b_1)] = d ]
9:      while en_P(S) ≠ ∅ do
            /* schedule thread of maximal priority */
10:         t ← element of en_P(S) such that p[t] maximal
11:         S ← S t
            /* change priority first time we peek a b-label */
12:         for all i ∈ {1, ..., d − 1} do
13:             if next_E(S, t) = b_{i+1} and p[t] ≠ d − i then
14:                 p[t] = d − i
15:                 [ assert: length(S) = k_i ]
16:             end if
17:         end for
18:     end while
19:     return S
20: end
```

**Figure 8.** The directed scheduler.

DEFINITION 8. *Let $P$ be a program, and let $B$ be a bug of $P$. Then we define the* depth *of $B$ to be*

$$depth(B) = \min\{size(D) \mid D \text{ guarantees } B\}$$

### 3.3.3 Coverage Theorem

The following theorem states the key guarantee: the probability that one invocation $RandS(n,k,d)$ of our randomized scheduler (Fig. 7) detects a bug of depth $d$ is at least $\frac{1}{nk^{d-1}}$.

THEOREM 9. *Let $P$ be a program with a bug $B$ of depth $d$, let $n \geq maxthreads(P)$, and let $k \geq maxsteps(P)$. Then*

$$\mathbf{Pr}[RandS(n,k,d) \in B] \;\geq\; \frac{1}{nk^{d-1}}$$

**Proof** Because $B$ has depth $d$, we know there exists a directive $D$ for $B$ of size $d$. Of course, in any real situation, we do not know $D$, but by Def. 8 we know that it exists, so we can use it for the purposes of this proof. Essentially, we show that even without knowing $D$, here is a relatively high probability that $RandS(n,k,d)$ follows the directive $D$ by pure chance. To prove that, we first construct an auxiliary algorithm $DirS(n,k,d,D)$ (Fig. 8) that uses the same random variables as $RandS$, but has knowledge of $D$ and constructs its schedule accordingly.

Comparing the two programs, we see two differences. First, Line 13 uses a condition based on $D$ to decide when to change priorities. In fact, this is where we make sure the call to $DirS(n,k,d,D)$

is following the directive $D$: whenever we catch a glimpse of thread $t$ executing one of the labels $b_i$ (for $i > 1$), we change the priority of $t$ accordingly. Second, $DirS$ has assertions which are not present in $RandS$. We use these assertions for this proof to reason about the probability that $DirS$ guesses the right random choices. The intended behavior is that $DirS$ fails (terminating immediately) if it executes a failing assertion.

The following three lemmas (the proofs are provided in the appendix) are key for our proof construction:

- $DirS$ succeeds with probability $\geq \frac{1}{nk^{d-1}}$ (Lemma 16).
- If $DirS$ succeeds, it returns a schedule that finds the bug (Lemma 12).
- If $DirS$ succeeds, it returns the same schedule as $RandS$ (Lemma 15).

We can formally assemble these lemmas into a proof as follows. Our sample space consists of all valuations of the random variables $\pi$ and $k_1, \ldots, k_{d-1}$. By construction, each variable is distributed uniformly and independently (thus, the probability of each valuation is equal to $n!k^{d-1}$). Define $\mathcal{S}$ to be the event (that is, set of all valuations) such that $DirS(n, k, d, D)$ succeeds, and let $\overline{\mathcal{S}}$ be its complement. Then

$$
\begin{aligned}
&\mathbf{Pr}[RandS(n,k,d) \in B] \\
&= \mathbf{Pr}[RandS(n,k,d) \in B \mid \mathcal{S}] \cdot \mathbf{Pr}[\mathcal{S}] \\
&\quad + \mathbf{Pr}[RandS(n,k,d) \in B \mid \overline{\mathcal{S}}] \cdot \mathbf{Pr}[\overline{\mathcal{S}}] \\
&\geq \mathbf{Pr}[RandS(n,k,d) \in B \mid \mathcal{S}] \cdot \mathbf{Pr}[\mathcal{S}] \\
&= \mathbf{Pr}[DirS(n,k,d,D) \in B \mid \mathcal{S}] \cdot \mathbf{Pr}[\mathcal{S}] && \text{(by Lemma 15)} \\
&= 1 \cdot \mathbf{Pr}[\mathcal{S}] && \text{(by Lemma 12)} \\
&\geq \frac{1}{nk^{d-1}} && \text{(by Lemma 16).}
\end{aligned}
$$

# 4. Implementation

This section describes our implementation of the PCT scheduler for x86 binaries.

## 4.1 Design Choices

The PCT scheduler, as informally described in Section 2, is based on thread priorities. The obvious way to implement PCT would be to reuse the priority mechanisms already supported by modern operating systems. We chose not to for the following reason. The guarantees provided by PCT crucially rely on a low priority thread proceeding strictly slower than a high priority thread. OS priorities do not provide this guarantee. In particular, priority boosting [13] techniques can arbitrarily change user-intended priorities. Similarly, our scheduler would not be able to control the relative speeds of two threads with different priorities running concurrently on different processors.

For fine-grained priority control, we implemented PCT as a user-mode scheduler. PCT works on unmodified x86 binaries. It employs binary instrumentation to insert calls to the scheduler after every instruction that accesses shared memory or makes a system call. The scheduler gains control of a thread the first time the thread calls into the scheduler. From there on, the scheduler ensures that the thread makes progress only when all threads with higher priorities are blocked. Thread priorities are determined by the algorithm as described in Section 2.

Our scheduler is able to reliably scale to large programs. We are successfully able to run unmodified versions of Mozilla Firefox and Internet Explorer, two popular web browsers, and find bugs in them. Our initial prototype slows down the execution of the program by 2 to 3 times. This is well within the expected slowdowns for any binary instrumentation tool.

One challenge we identified during our implementation is the need for our scheduler to be starvation free. It is common for concurrent programs to use spin loops. If, under PCT, a high priority thread spins waiting for a low priority thread, the program will livelock — PCT does not schedule the low priority thread required for the high priority thread to make progress. To avoid such starvation issues, PCT uses heuristics, such as repeated yields performed by a thread, to identify threads that are not making progress and lowers their priorities with a small probability.

## 4.2 Optimizations

The base algorithm described in Section 2 requires that the scheduler have the capability of inserting a priority change point at randomly selected instructions. This has two disadvantages. First, the need to insert a change point at an arbitrary instruction requires PCT to insert a callback after every instruction, slowing down the performance. Second, by counting the number of instructions executed the large value for the parameter $k$ can reduce the effectiveness especially for bugs with depth $\geq 2$. We introduced the optimizations below to address this problem.

**Identifying Synchronization Operations:** The first optimization relies on identifying synchronization operations and inserting priority change points only at these operations. We first classify thread operations into two classes: synchronization operations and local operations. A synchronization operation can be used to communicate between threads, while a local operation is used for local computation within a thread. Synchronization operations include system calls, calls to synchronization libraries (such as `pthreads`), and hardware synchronization instructions (such as interlocked instructions). In addition, we also treat accesses to flag variables, volatile accesses, and data races (both programmer intended and unintended) as "shared-memory synchronization." Our classification reflects the fact that these memory accesses result in communication between the threads. Local operations include instructions that do not access memory and memory accesses that do not participate in a data race (such as accessing the stack or accessing consistently protected shared memory). Any of the existing data-race detection tools [8, 10] or hardware mechanisms [22] can be used to classify memory accesses into local or synchronization operations. Other forms of synchronization are straightforward to identify from the program binary.

This optimization relies on the following observation. For every execution in which a priority change point occurs before a local operation, there exists a behaviorally-equivalent execution in which the priority change point occurs before the synchronization operation that immediately follows the local operation. This is because the two executions differ only in the order of local operations. This means that we only need to insert priority change points before synchronization operations. This effectively reduces $k$ in the probabilistic bound by several orders of magnitude, from the maximum number of instructions executed by the program to the maximum number of synchronization operations. In the rest of the paper, we only report the number of synchronization operations as $k$.

**Identifying Sequential Execution:** We observed for some benchmarks that a significant portion of a concurrent execution is actually sequential where there is only one enabled thread. Inserting priority change points during this sequential execution is not necessary. The same effect can be achieved by reducing the priority at the point the sequential thread enables/creates a second thread.

**Identifying Join Points:** Programs written with a fork-join paradigm typically have multiple phases where a single thread waits for a flurry of concurrent activity belonging to one phase to

| Programs | Bug | |
|---|---|---|
| | Symptom | Known? |
| Splash-FFT | Platform dependent macro missing a wait leading to order violations | YES |
| Splash-LU | | YES |
| Splash-Barnes | | YES/**NO** |
| Pbzip2 | Crash during decompression | YES |
| Work Steal Queue | Internal assertion fails due to a race condition | YES/**NO** |
| Dryad | Use after free failing an internal assertion | YES |
| IE | Javascript parse error | **NO** |
| Mozilla | Crash during restoration | **NO** |

**Table 1.** Concurrency benchmarks and bugs. YES/NO indicates both known and unknown bugs.

finish before starting the next phase. This is also a typical behavior of long running stress tests that perform multiple iterations of concurrency scenarios. Our implementation of PCT identifies these phases whenever the program enters a state with one thread enabled. The effective $k$ is the maximum number of synchronization operations performed per phase.

**Final Wait:** Some concurrency bugs might manifest much later than when they occur. We found that PCT missed some of the manifestations as the main thread exits prematurely at the end of the program. Thus, we artificially insert a priority change point for the main thread before it exits.

## 5. Experiments

In this section, we describe the evaluation of our PCT scheduler on several real-world programs of varying complexity. All experiments were conducted on an quad-core Intel Xeon L5420 running at 2.50GHz, with 16GB of RAM running 64-bit Windows Server Enterprise operating system.

### 5.1 Experimental Setup

#### 5.1.1 Benchmarks and Bugs

In our evaluation, we used open source applications such as Mozilla Firefox code-named Shiretoko, a commercial web browser-Internet Explorer, a parallel decompression utility - Pbzip2, three Splash benchmarks (FFT, LU, Barnes), a work stealing queue implementation [11] and a component of Dryad [14]. We used these applications as most of these were used in prior work on discovering concurrency bugs [20, 24]. Table 1 lists the manifestation of the bug in these applications and also reports whether the bug was previously known. PCT discovered all previously known bugs faster than reported in respective prior work [20, 24]. We also find new bugs in work stealing queue and Barnes benchmark that were missed by prior work. Finally, we find previously unknown bugs in Firefox and Internet Explorer.

Table 2 also lists the various properties of the benchmarks. The table lists the number of threads ($n$), the total number of synchronization operations executed ($k$), and the depth of the bug ($d$) in the application. It also shows the effective number of operations after optimization ($k_{eff}$) described in Section 4. It is interesting to note that our prototype detected the bugs in Mozilla and Internet Explorer even though, it has a large value of $k$. Moreover, Mozilla Firefox and Internet Explorer are large applications and the ability to detect bugs in these large applications demonstrates the scalability of the tool.

Our prototype counts $k$ (the number of steps) in each execution, and then uses that value as an estimate for $k$ in the next execution. It may thus somewhat over- or underestimate the actual number of steps if $k$ varies between executions. This may somewhat degrade

| Programs | LOC | $d$ | $n$ | $k$ | $k_{eff}$ |
|---|---|---|---|---|---|
| Splash-FFT | 1200 | 1 | 2 | 791 | 139 |
| Splash-LU | 1130 | 1 | 2 | 1517 | 996 |
| Splash-Barnes | 3465 | 1 | 2 | 7917 | 318 |
| Pbzip2 | 1978 | 2 | 4 | 1981 | 1207 |
| Work Steal Queue | 495 | 2 | 2 | 1488 | 75 |
| Dryad | 16036 | 2 | 5 | 9631 | 1990 |
| IE | - | $1^*$ | 25 | 1.4M | 0.13M |
| Mozilla | 245172 | $1^*$ | 12 | 38.4M | 3M |

**Table 2.** Characteristics of various benchmarks. Here 1M means one million operations. $1^*$ indicates that the previously unknown bug was found while running with a bug depth of 1.

| Programs | Stress | Random Sleep | PCT | |
|---|---|---|---|---|
| | | | Measured | Guaranteed |
| Splash-FFT | 0.06 | 0.27 | 0.50 | 0.5 |
| Splash-LU | 0.07 | 0.39 | 0.50 | 0.5 |
| Splash-Barnes | 0.0074 | 0.0101 | 0.4916 | 0.5 |
| Pbzip2 | 0 | 0 | 0.701 | 0.0001 |
| Work Steal Queue | 0 | 0.001 | 0.002 | 0.0003 |
| Dryad | 0 | 0 | 0.164 | $2 \times 10^{-5}$ |

**Table 3.** Measured or guaranteed probability of finding the bug with various methods such as Stress, Random delay insertion methods and PCT and the worst-case bound (based on n, k and d in Table 2).

the guaranteed worst-case probability but is not much of a concern for practical efficiency.

#### 5.1.2 Comparing Other Techniques

As a point of comparison, we also ran the benchmarks in Table 3 with our stress testing infrastructure. Our stress infrastructure ran all these benchmarks under heavy load a million times. We made a *honest, good-faith* effort to tune our stress infrastructure to increase its likelihood of finding bugs. Our effort is reflected by the fact that our stress infrastructure detected the known bugs in the benchmarks with a higher probability and a lot quicker than prior stress capabilities reported in literature [24].

As another interesting comparison, we implemented a scheme that introduces random sleeps at synchronization operations with a certain probability [1]. The experiments are sensitive to the particular probability of sleep. Again, we made a good-faith effort to find the configuration that works best for our benchmarks. We experimentally discovered a probability of $1/50$ performed reasonable well in detecting bugs.

These two reflect the state of the art concurrency testing techniques that we are able to recreate in our setting. Heuristic-directed testing [15, 24] and systematic scheduling [21] require sophisticated analysis and we are currently unable to perform quantitative experiments with these techniques. We compare with these techniques qualitatively.

### 5.2 Effectiveness

#### 5.2.1 Comparison with worst-case guarantee

Apart from discovering known and unknown bugs, to confirm whether our prototype attains the worst-case guaranteed probability of $1/nk^{d-1}$ as discussed in Section 3, we measured empirical probabilities of detecting the bug with our prototype. In this experiment, our prototype ran each application one million times with each run having a different random seed. The relative frequency of occurrence of the bug in these runs represents the empirical probability. Table 3 reports the empirical and the guaranteed worst-case

probability of finding the bug with PCT. We report the measured probabilities only for applications where it was feasible to do one million runs.

prototype attains the worst-case guaranteed probability of $1/nk^{d-1}$ as discussed in Section 3, we measured empirical probabilities of detecting the bug with our prototype. In this experiment, our prototype ran each application one million times with each run having a different random seed. The relative frequency of occurrence of the bug in these runs represents the empirical probability. Table 3 reports the empirical and the guaranteed worst-case probability of finding the bug with PCT. We report the measured probabilities only for applications where it was feasible to do one million runs.

Table 3 reveals that the measured bug detection probability is as low as the worst-case guaranteed probability in some cases, but far exceeds it in others. For Barnes, LU and FFT that have 2 threads, our implementation finds the bug with a probability approximately half. For the Barnes benchmark, PCT slightly misses the worst-case bound. This is the effect of priority perturbations introduced to guarantee starvation-freedom, as discussed in Section 4.

For benchmarks with a bug of depth 2, namely Pbzip2, Work Steal Queue, and Dryad, our implementation is orders of magnitude better than the worst-case bound. The Pbzip2 bug, shown in Fig. 2 requires an extra constraint that the main thread does not prematurely die before the error manifests. Since PCT guarantees this by default (Section 4), PCT finds the bug as if it was a bug of depth 1. Both Work Steal Queue and Dryad demonstrate the effect of optimizations that reduce $k$. We study our results with the Work Steal Queue in detail in Section 5.3.

#### 5.2.2 Comparison to other techniques

Table 3 summarizes our experiments comparing PCT with stress, synchronization based random sleeps. Our sophisticated stress infrastructure has trouble finding bugs of depth one which are trivially caught by PCT. Note, that our stress infrastructure detects bugs of in FFT, LU and Barnes much more successfully than reported in prior literature [24].

Our random sleep scheme detects bugs in FFT, LU, Barnes quicker than stress. It also detected the 2-edge bug in work stealing queue albeit with a low probability. However it is not able to find the bugs in Pbzip and Dryad. To summarize, PCT scheduler discovers bugs with a higher probability and more quickly than other schemes we investigated.

CHESS [21] finds the Work Steal Queue and the Dryad bug after (approx.) 200 and 1000 runs of the program respectively. The PCT scheduler detects the same bug in the $6^{th}$ and $35^{th}$ run of the program respectively. We were unable to run CHESS on other benchmarks. As the next section shows, PCT scales much better if one increases the number of threads, while we expect CHESS to perform exponentially worse.

In comparison to CTrigger [24], PCT finds the bug in the benchmarks common to both well within the first three iterations. This is more efficient as we do not require a separate profiling run required by CTrigger.

### 5.3 Work Steal Queue Case Study

As a case study, we discuss the impact of increasing the number of synchronization operations and the number of threads with the PCT scheduler for the applications in Table 3. Due to space constraints, we report the behavior only with the work stealing queue program. Other benchmarks show similar behavior.

Work Steal Queue program implements a queue of work items and was originally designed for the Cilk multithreaded programming system [11]. In this queue, there are two kinds of threads namely victims and stealers. Victim pushes and pops from the tail of the queue and a stealer steals from the head of the queue. The application uses sophisticated lock-free synchronization to protect the queue. To study the impact of increased threads, we increased the number of stealers and the number of items being pushed and popped.

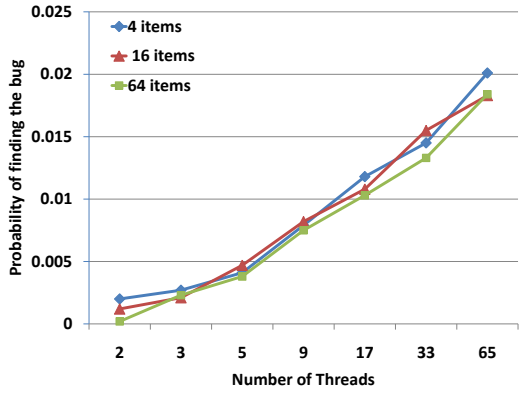


**Figure 9.** Measured probability of finding bugs with an increase in the number of threads and number of items.

#### 5.3.1 Effect of Execution Size on PCT

Fig. 9 shows the probability of finding the bug with our prototype implementing the PCT scheduler with an increase in the number of threads and the number of operations for the work stealing queue program. In this experiment, we increase $n$ by increasing the number of stealers. (The program requires that there is exactly one victim.) Each stealer does two steal attempts, while the victim pushes and pops a specified number of items. We increase $k$ by increasing the number of items.

Fig. 9 reveals that the probability of finding the bug actually increases with the increase in the number of threads. Moreover, the probability of finding the bug is same irrespective of the number of operations. This result may seem surprising as the worst-case guaranteed probability is $1/nk$, which decreases with growing parameters of $n$ and $k$.

However, in this case, an increase of the different opportunities to find the bug means that the probability adds up (as discussed in Section 2.6.1). As the number of threads and the operations increase, there are many opportunities to find the bug. PCT needs to find the race condition when *any* one of the stealer is interfering while attempting to steal *any* of the items.

#### 5.3.2 PCT vs Stress

Fig. 10 shows the probability of finding the bug with the PCT scheduler and stress testing with an increase in the number of threads for the work stealing queue program. In this experiment, the number of stealers was varied from 2 to 64 with total number of items being pushed and popped by another thread at a constant four items. As discussed earlier, the probability of PCT increases with the number of threads. However, the interesting thing to note, even with the sophisticated stress testing framework, the probability of detecting the bug with stress is low and is non-deterministic. Irrespective of the system load, PCT scheduler has the same probability of detecting the bug when given the same random seed.

#### 5.3.3 Interleaving Coverage

To evaluate the coverage of thread events with PCT, we instrumented the work stealing queue program with twenty events, fourteen events in the main thread which does the pushes and the pops and six events in the stealer thread. There were a total of 168 unique

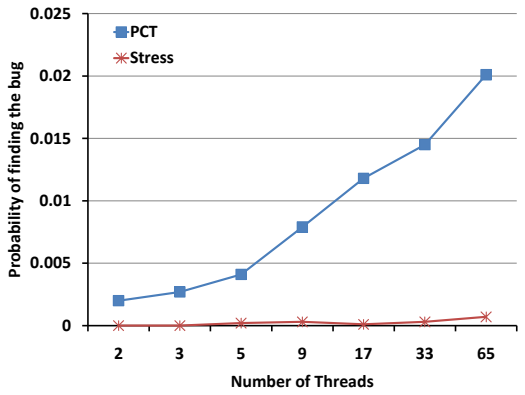


**Figure 10.** Measured probability of finding bugs with PCT and Stress with increase in threads and number of threads

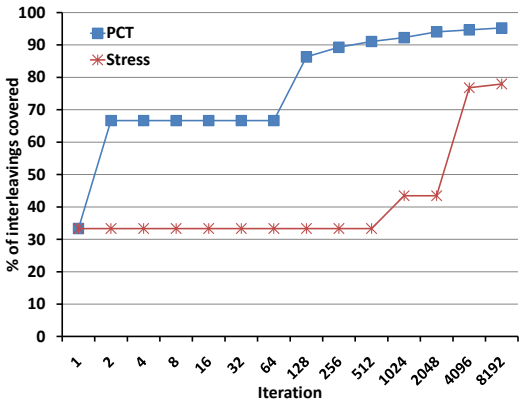


**Figure 11.** Coverage of various thread events with the specified number of program runs with PCT and Stress for the work stealing queue program.

event pairs possible in this setting. Fig. 11 plots the cumulative percentage of the events covered as the program is run specified number of times. The horizontal axis represents the number of times the program was run (in logarithmic scale). We restrict the horizontal axis to the 8192 runs as stress did not explore any new event pair beyond those already explored in the new runs after that and PCT eventually explored all the event pairs. Fig. 10 showed that stress had a low probability of catching the bug. Fig. 11 shows that stress does not cover more than 20% of the event pairs, few of which result in a bug. Thus, stress's inability/ineffectiveness to detect the bug is highly correlated with the event pairs not covered. The ability to cover almost all the event pairs enables PCT to detect the bug.

## 6. Related Work

Our work is closely related with concurrency verification and testing techniques that aim to find bugs in programs. We classify prior work and compare our work below.

**Dynamic Scheduling:** PCT is related to techniques that control the scheduler to force the program along buggy schedules. Stress testing is a commonly used method where the program is subjected to heavy load in the presence of artificial noise created by both running multiple unrelated tests simultaneously and by inserting random sleeps, thread suspensions, and thread priority changes. In contrast, PCT uses a disciplined, mathematically-random priority mechanism for finding concurrency bugs. This paper shows that PCT outperforms stress theoretically and empirically.

Prior approaches for generating randomized schedules [7, 25] insert random sleep at synchronization points and use heuristics based on various coverage criteria, concurrency bug patterns and commutativity of thread actions to guide the scheduler [9]. These approaches involve a random decision at every scheduling point. As shown in Section 2, even simple concurrency bugs can frustrate these techniques. In stark contrast, PCT uses a total of $n + d - 1$ random decisions, together for the initial thread priorities and $d-1$ priority change points. Our key insight is that these small but calculated number random decisions are sufficient for effectively finding bugs.

Researchers have also proposed techniques that actively look for concurrency errors [15, 24]. They use sophisticated analysis, either by running profiling runs that detect suspicious non-serializable access patterns [24], or use static or dynamic analysis to find potential deadlocks [15]. In a subsequent phase these techniques heuristically perturb the OS scheduler guided by the prior phase. Our technique does not require prior analysis but it still is comparable in bug-finding power of these techniques. For instance, for the SPLASH2 benchmarks used in the former technique [24], PCT finds the bug in the first few runs far less time than required for the profiling runs in the previous approach. However, our technique is orthogonal to these approaches and identifying potential buggy locations can improve PCT as well.

**Systematic Exploration:** Model checking techniques [12, 21] systematically explore the space of thread schedules of a given program in order to find bugs. These techniques can prove the absence of errors only after exploring the state space completely. In contrast, PCT provides a probabilistic guarantee after every run of the program. With respect to bug-finding capability, we have compared PCT with the CHESS tool [21] on two benchmarks. PCT finds bugs much faster than CHESS in both cases.

CHESS uses a heuristic of exploring schedules with fewer number of preemptions. By default, CHESS explores executions non-preemptively except at few chosen steps where it inserts preemptions. In contrast, PCT is a priority based scheduler and can introduce arbitrary number of preemptions in the presence of blocking operations even if the bug depth is small. For instance, when a low-priority thread wakes up a higher-priority thread, a priority-based scheduler preempts the former to scheduler the latter.

**Concurrency Bug Detection:** Our approach is also related to numerous hardware and software techniques that find concurrency errors dynamically but without exercising any scheduler control [8, 10, 18, 27]. PCT's goal is to *direct* the scheduler towards buggy schedules and requires that these bugs are detected by other means, either by a program assertion or with the use of these concurrency bug detection engines.

**Scheduling Control:** There are other good reasons for exercising scheduling control. These techniques either attempt to enforce a subset of interleavings [4, 23, 28] or prevent potential bugs [16, 18] inferred during the current (or prior) run of the program. PCT is doing the opposite — forcing the program towards bugs. A controlled scheduler can also be useful for reproducing buggy executions [2, 6, 17, 26] once the bugs have manifested. The PCT scheduler is deterministic given a random seed, and thus can also be used to reproduce execution with an appropriate replay mechanism to regenerate inputs [5]. Finally, researchers have proposed scheduling variations for debugging concurrent programs [3].

## 7. Conclusion

This paper describes PCT, a randomized algorithm for concurrency testing. PCT uses a disciplined schedule-randomization technique to provide efficient probabilistic guarantees of finding bugs during

testing. We evaluate an implementation of PCT for x86 executables on demonstrate its effectiveness in finding several known and unknown bugs in large-scale software.

**Acknowledgements:** We would like to thank Milo Martin, George Candea, Ras Bodik, Tom Ball, and numerous anonymous reviewers for feedback on the initial versions of this paper.

## References

[1] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing scheduling that causes concurrent programs to fail. In *PADTAD*, pages 37–40, 2006.

[2] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8(2):66–74, 1991. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.73751.

[3] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. In *ISSTA*, pages 210–220, 2002.

[4] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS*, 2009.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[6] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE 08: Virtual Execution Environments*, pages 121–130. ACM, 2008.

[7] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[8] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI*, 2007.

[9] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, page 286, 2003.

[10] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, 2009.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223. ACM Press, 1998.

[12] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97*, pages 174–186. ACM Press, 1997.

[13] J. L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Trans. Softw. Eng.*, 19(8):813–825, 1993. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.238584.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. Technical Report MSR-TR-2006-140, Microsoft Research, 2006.

[15] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.

[16] H. Jula, D. M. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, pages 295–308, 2008.

[17] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.1987.1676929.

[18] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access-interleaving invariants. *IEEE Micro*, 27(1):26–35, 2007.

[19] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[20] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[21] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[22] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas. Sigrace: signature-based data race detection. In *ISCA*, 2009.

[23] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS*, 2009.

[24] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[25] K. Sen. Effective random testing of concurrent programs. In *ASE*, 2007.

[26] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA*, 2003.

[27] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI*, 2005.

[28] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, 2009.

## A. Proofs

The lemmas in the section below are part of the proof of Thm. 9 in Section 3, moved to the appendix as to not disrupt the flow of the paper.

LEMMA 10. *If $next_E(S,t) = b_j$ right before executing line 11 of $DirS(n,k,d,D)$, and if $S$ does not violate $D$, then $p[t] = d-j+1$.*

**Proof** Distinguish cases $j > 1$ and $j = 1$.

**Case** $j > 1$. Def. 4 implies that the first action by any thread is not labeled, so our assumption $next_E(S,t) = b_j$ implies that there is a $m < length(S)$ such that $S[m] = t$. Choose a maximal such $m$. Then we know $next_E(S[0,m],t) = b_j$ (because Def. 4 implies that the next label does not change if other threads are scheduled). Thus, in the iteration that added $S[m]$ to the schedule, the test $next_E(S[0,m],t) = b_{i+1}$ on line 13 must have evaluated to true for $i = j-1$ (and for no other $i$, because the $b_i$ are pairwise distinct by Def. 4). So we must have assigned $p[t] \leftarrow d-j+1$ on line 14. Because we chose $m$ maximal, $t$ was never scheduled after that, so its priority did not change and must thus still be $p[t] = d-j+1$.

**Case** $j = 1$. If we are about to execute line 11, then the assertion on line 8 must have succeeded, so at that time it was the case that $p[t] = d-j+1$. After that, $p[t]$ could not have changed: suppose line 14 was executed at some point to change $p[t]$. Say the value of variable $S$ at that point was $S[0,m]$. Then the condition $next_E(S[0,m],t) = b_{i+1}$ on line 13 must have evaluated to true for some $i$. Now, because we know $next_E(S,t) = b_j$ and because $b_j \neq b_{i+1}$ (by Def. 4), there must exist a $m' > m$ such that $S[m'] = t$ (by Def. 4). But that implies $label_E(S,m') = b_{i+1}$, and since $b_j \notin labels_E(S)$, $S$ violates $D$ which contradicts the assumption.

LEMMA 11. *Let $1 \leq t' \leq n$, and let $p[t'] = d-j+1$ for some $j \geq 2$ at the time line 10 is executed. Then either $b_j \in labels_E(S)$, or $next_E(S,t') = b_j$.*

**Proof** The only way to assign priorities less than $d$ is through the assignment $p[t] = d-i$ on line 14. So this line must have executed with $t = t'$ and $i = j-1$. Thus, the condition $next_E(S,t) = b_{i+1}$ was true at that point, which is identical to $next_E(S,t') = b_j$. If $t'$ is not scheduled after that point, this condition is still true; conversely, if $t'$ is scheduled, then it must execute the label $b_j$, implying $b_j \in labels_E(S)$.

LEMMA 12. *When executing $DirS(n,k,d,D)$, the following conditions are satisfied.*

1. *any schedule returned at the end is in $B$.*
2. *at all times, the variable $S$ is a schedule that does not violate $D$.*

**Proof** Clearly, the second claim follows from the first because if the while-loop terminates, $S$ is a complete schedule, and by Def. 6, any complete schedule that does not violate $D$ is in $B$.

To prove the first claim, we proceed indirectly. If $S$ violates $D$, the first moment it does so must be right after executing $S \leftarrow St$ on line 11. Now, consider the state right before that. $S$ does not violate $D$. Therefore, by Def. 6 there must exist an alternate choice $t' \in en_P(S)$ such that $St'$ does not violate $D$. Because the algorithm is choosing $t$ over $t'$, it must be the case that $p[t] \geq p[t']$. Now, we know $St$ violates $D$, but $S$ does not, therefore (by Def. 5), there is an $i$ such that $next_E(S,t) = b_i$. By Lemma 10, this implies that $p[t] = d-i+1$. Thus we know $p[t'] < d-i+1$, thus $p[t'] = d-j+1$ for some $j > i$. By Lemma 11, that implies that either $b_j \in labels_E(S)$ or $b_j \in next_E(S,t')$. But both of these lead to a contradiction: $b_j \in labels_E(S)$ means that $S$ violates $D$ (because $b_i \notin labels_E(S)$), and $b_j \in next_E(S,t')$ means that $St'$ violates $D$.

LEMMA 13. *During the execution of $DirS(n,k,d,D)$ the assertion on line 15 is executed at most once for each $i \in \{1,\ldots,d-1\}$.*

**Proof** Consider the first time the assertion $length(S) = k_i$ on line 15 is executed for a given $i$. Then it must the case that $next_E(S,t) = b_{i+1}$. Because labels don't repeat, the only chance for this condition to be true again is for the same $i$ (because the $b_i$ are pairwise distinct) during the immediately following iterations of the while loop, and only if threads other than $t$ are scheduled. But in that scenario, the priority $p[t]$ does not change, so the second part $p[t] \neq d-i$ of the condition on line 13 can not be satisfied.

LEMMA 14. *If $DirS(n,k,d,D)$ succeeds, it executes the assertion $length(S) = k_i$ on line 15 at least once for each $i \in \{1,\ldots,d-1\}$.*

**Proof** Because $DirS(n,k,d,D)$ succeeds, it produces a complete schedule $S$ which does not violate $D$. Thus, $b_i \in labels_E(S)$ for all $i \in \{1,\ldots,d\}$. Thus, for each $i \in \{2,\ldots,d\}$, there must be an $m$ such that $S[0,m], thread_E(b_i)) = b_i$. Thus, the condition on line 13 must be satisfied at least once for each $i \in \{1,\ldots,d-1\}$, so we know the assertion $length(S) = k_i$ on line 15 gets executed for each $i \in \{1,\ldots,d-1\}$.

LEMMA 15. *If $DirS(n,k,d,D)$ succeeds, then*

$$RandS(P,n,d) = DirS(n,k,d,D).$$

**Proof** To prove the claim, we now show that the two respective conditions on lines 13

$$length(S) = k_i \tag{1}$$

$$next_E(S,t) = b_{i+1} \text{ and } p[t] \neq d-i \tag{2}$$

evaluate the same way, for any given iteration of the while and for loops (identified by current values of $S$ and $i$, respectively). Clearly, if (2) evaluates to true for some $S$ and $i$, $DirS$ executes the assertion on line 15, thus guaranteeing that (1) also evaluates to true. Conversely, if (1) evaluates to true for some $S$ and $i$, consider that $DirS$ must execute an assertion of the form $length(S') = k_i$ at some point (by Lemma 14); but it turns out that this must happen in the very same iteration because the length of $S$ uniquely identifies the iteration of the while loop, so the condition (2) must be satisfied.

LEMMA 16. *The probability that $DirS(n,k,d,D)$ succeeds is at least $\frac{1}{nk^{d-1}}$.*

**Proof** $DirS(n,k,d,D)$ succeeds if and only if if (1) the assertion $p[thread_E(b_1)] = d$ on line 8 passes, and (2) the assertion $length(S) = k_i$ on line 15 passes every time it is executed. The probability of the former passing is $1/n$ (because a random permutation assigns the lowest priority to any given thread with probability $1/n$), while the probability of each latter passing is $1/k$ (because the random variables $k_i$ range over $\{1,\ldots,k\}$). By Lemma 13, the assertions $length(S) = k_i$ are executed at most once for each $i$. Thus, all of the assertions involve independent random variables, so we can multiply the invidual success probabilities to obtain a total success probability for $DirS$ of at least $(1/n)\cdot(1/k)^{d-1}$.