Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection

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

Sebastian Burckhardt
Microsoft Research
sburckha@microsoft.com

Milo M. K. Martin
University of Pennsylvania
milom@cis.upenn.edu

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Abstract

Testing multithreaded programs is difficult as threads can interleave in a nondeterministic fashion. Untested interleavings can cause failures, but testing all interleavings is infeasible. Many interleaving exploration strategies for bug detection have been proposed, but their relative effectiveness and performance remains unclear as they often lack publicly available implementations and have not been evaluated using common benchmarks. We describe NeedlePoint, an open-source framework that allows selection and comparison of a wide range of interleaving exploration policies for bug detection proposed by prior work.

Our experience with NeedlePoint indicates that priority-based probabilistic concurrency testing (the PCT algorithm) finds bugs quickly, but it runs only one thread at a time, which destroys parallelism by serializing executions. To address this problem we propose a parallel version of the PCT algorithm (PPCT). We show that the new algorithm outperforms the original by a factor of 5× when testing parallel programs on an eight-core machine. We formally prove that parallel PCT provides the same probabilistic coverage guarantees as PCT. Moreover, PPCT is the first algorithm that runs multiple threads while providing coverage guarantees.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algorithms, Reliability, Verification

Keywords Concurrency, priority-based scheduling, multithreading, probabilistic concurrency testing, parallel testing

1. Introduction

Multithreaded programs are difficult to test and debug because their behavior depends on the specific interleaving of shared memory accesses, which in turn depends on how the threads are interleaved by multicore hardware and thread scheduling software. Because thread interleaving is non-deterministic and largely unpredictable, an astronomical number of interleavings is possible even for small programs. However, due to the statistical nature of the interleavings, repeatedly testing the program on the same platform results in redundant exploration of similar interleavings. The untested uncommon interleavings can have *concurrency bugs* that escape the testing process but manifest in deployed systems, especially if differences in the deployed system's hardware or software influence observed interleavings.

One way to address this problem is to increase the coverage achieved during testing by steering executions towards uncommon schedules. We classify prior proposals for such controlled scheduling into two categories. *Best-effort* tools insert explicit thread yields at selected points at runtime [7, 11, 12, 22–25]. The strategies for inserting yield points vary, ranging from random selection [7, 24] to heuristics based on identifying symptomatic bug patterns in the code, such as data races [25], potential atomicity violations [22, 23], and potential deadlocks [12]. On the other hand, *guaranteed-coverage* tools such as CHESS [21] and PCT [5] provide provable guarantees of the schedule coverage achieved during concurrency testing. To provide such coverage guarantees, both these tools require absolute control of the thread scheduling achieved by running the threads serially one at a time. Although coverage guarantees are appealing in practice, the serial execution limits the applicability of these tools, a disadvantage not shared by best-effort approaches.

To better understand the relative effectiveness and performance of prior best-effort and guaranteed-coverage tools, we set out to reproduce and compare prior research proposals by reimplementing several published algorithms. In this process, we identified a common set of mechanisms required by a wide range of bug detection techniques. These mechanisms include instrumenting synchronization accesses, identifying blocked threads, and handling starvation. A systematic approach to providing these mechanisms is useful because manual instrumentation or annotations become quickly impractical when controlling schedules of large applications.

Motivated by this observation, we developed NeedlePoint, a unified framework for implementing a wide range of controlled scheduling approaches — ranging from simple random sleeps to prioritized scheduling. The framework separates the mechanisms required to implement the various scheduling policies from the policies themselves. This separation allowed us to implement several previously proposed approaches as policies that plug into the framework. In particular, we reimplemented the following strategies: (1) a randomized version of preemption bounding [19], (2) AtomFuzzer [22], (3) simple random sleeps, and (4) probabilistic concurrency testing (PCT) [5].

We then used Needlepoint to test a collection of multithreaded programs containing known concurrency bugs and also benchmarks from the SPLASH and Parsec suites. Our experience with NeedlePoint shows empirically that PCT is highly effective at finding bugs. However, it also reveals that PCT's performance suffers significantly from its inherent restriction of scheduling only one thread at a time. Although running one thread is good enough for running small unit tests, it becomes a problem and causes significant slowdowns when testing large programs with a large number of threads or when testing highly parallel benchmarks.

PLDI'12, June 11–16, 2012, Beijing, China.
Copyright © 2012 ACM 978-1-4503-1205-9/12/06. . . $10.00

To alleviate this problem, we propose a new testing algorithm called PPCT (parallel PCT). Like best-effort tools, PPCT can schedule multiple threads at a time, thereby making better use of hardware parallelism. At the same time, PPCT provides the same coverage guarantees as PCT. To the best of our knowledge, PPCT is the first algorithm to provide coverage guarantees without requiring serial execution of threads.

We provide a brief overview of the PCT algorithm [5] before explaining our improvement. PCT assigns thread priorities chosen uniformly randomly at the beginning of the program. At every step of the algorithm, PCT schedules the unblocked thread with the highest priority. The scheduled thread executes instructions up to the next synchronization event before yielding the control back to the scheduler. During this execution, PCT performs a small number of priority changes at steps chosen uniformly randomly at the beginning of the execution. The guarantees provided by PCT depend on the *depth* of a concurrency bug, which is the number of scheduling constraints sufficient to trigger the buggy interleaving. For instance, ordering violations have a depth 1 and atomicity violations have a depth 2. Given a program with n threads that executes a maximum of k instructions, PCT triggers a bug of depth d with probability at least $\frac{1}{nk^{d-1}}$. For instance, PCT can find ordering violations with probability at least $\frac{1}{n}$.

The key intuition behind PPCT is that when searching for bugs of depth d, it is necessary to control the scheduling of only d threads. In particular, PPCT schedules *all* unblocked threads with a priority greater than d at every step of the algorithm. In the rare case when all unblocked threads have a priority smaller than or equal to d, PPCT resorts to serial scheduling and schedules the unblocked thread with the highest priority. We adapt the formal proof of PCT to show that PPCT provides the same coverage guarantees as PCT.

Empirically, our experiments confirm that PPCT detects bugs with either the same or higher rate when compared to PCT. Compared to PCT, however, PPCT provides significant speedups both on multi-core and single-core machines. PPCT provides speedups over PCT even on a single-core machine by providing more flexibility to the operating system scheduler to schedule a thread from a larger pool of enabled threads. On a multi-core machine with eight cores, replacing PCT by PPCT reduces the overhead of executing the program from 34× to just 6× on an average. Unlike a slowdown of 34×, a slowdown of 6× is still tolerable for testing interactive applications, and it enabled us to run priority based exploration with the Chrome web browser and other applications.

2. Background on Controlled Scheduling

This section provides background on the problem of schedule selection when testing multithreaded programs and the use of controlled scheduling to address the problem.

2.1 Schedule Selection

Of the many aspects of testing multithreaded programs, this paper focuses on the *schedule selection* problem, which involves effectively finding, among an astronomical number of possible thread schedules, those schedules that drive a program to an error. To this effect, we assume the existence of a test harness that provides necessary inputs to a multithreaded program and a test mechanism that determines a test failure, such as a program crash, an assertion violation, or an incorrect output.

Data-race detection is related to and is different from the schedule selection problem. A data race occurs when two threads concurrently access the same shared variable without appropriate synchronization, such as a missing lock. Although they are indications of erroneous programming, a data race is neither necessary or sufficient for a concurrency error [8, 14].[1] Schedule selection may also help data race detectors to uncover additional races.

2.2 Controlled Scheduling

Controlled scheduling techniques attempt to steer the program towards less common interleavings. This is typically accomplished by controlling the scheduling of threads and by allowing exactly one thread to execute at a given time. These techniques can systematically explore all possible interleavings (for smaller codes) or use various heuristics to guide which interleavings are explored (for larger codes). Further, they can provide repeatability by enforcing the same thread scheduling decisions, which aids debugging.

To control the order in which operations from various threads are executed, such systems typically use OS-level threads but control the execution in user mode by preventing all but one thread from making progress at a given time. A *scheduling policy* determines the thread that makes progress. A running thread invokes the user-level scheduler at each *scheduling point*. Scheduling points occur at thread creation, synchronization operations, and thread termination. To achieve full coverage of sequentially consistent behaviors, one also needs to insert scheduling points at certain shared-memory accesses, such as accesses to `volatile` variables (in Java), `atomic` variables (in C++), and variables participating in a data race. Instrumenting such shared-memory accesses, however, incurs additional overhead, requiring the tool designer to make an explicit overhead versus coverage tradeoff.

Controlled scheduling allows the user-level scheduler to determine the running/not-running status of each thread at each scheduling point. The status of each thread is maintained in shared memory. When the main thread spawns a thread, the newly created thread's status defaults to *not-running*. Whenever the running thread encounters a scheduling point, it invokes the scheduling policy which can then either (1) decide to execute the currently running thread or (2) choose another thread to execute, in which case the scheduling policy sets the other thread's status to *running* and sets the current thread's status to *not-running*. Inactive threads (ones that are marked as *not-running* by the user-level scheduler) check their status whenever scheduled by the OS-level scheduler, either discovering they have become the new active thread or yielding otherwise. To ensure forward progress for busy-waiting and other non-trivial synchronization idioms, the inactive threads also periodically invoke the scheduling policy

Lock-based synchronization can potentially block, *e.g.*, if the lock is held by some other thread. A controlled scheduling system must identify the threads that are blocked and should not schedule blocked threads, as doing so can cause the system to livelock. For simple locks, a controlled scheduling system can maintain a hashtable of all acquired locks with information about threads acquiring them. On a lock acquire operation, if the lock is already held by some other thread, then (1) the thread's status is changed to *blocked*, (2) the thread is added to the list of threads waiting for the lock, and (3) the scheduling policy is invoked as described above, which chooses one of the non-blocked threads. On a successful lock acquire, the hashtable is also updated with the information about the acquired lock and the acquiring thread. On a lock release, the scheduler changes the state of all blocked threads on the released lock from *blocked* to *not-running*.

2.3 Challenges with Controlled Scheduling

Implementing controlled scheduling is conceptually simple, but challenging in practice due to the following two reasons. First, real world programs use a plethora of synchronization primitives. Many programs define custom synchronization using the atomic primi-

[1] Here, we explicitly ignore weak-memory-model issues issues [3, 18]

tives provided by the hardware or use adhoc synchronization [26]. Accurately interpreting and handling all these synchronization operations in a controlled scheduling system is difficult and infeasible in practice. Further, even minor mistakes in interpreting synchronization operations can lead to erroneous livelocks in the controlled scheduling system and/or missed bugs. Second, the scheduling decisions made by the underlying scheduling policy to steer the multithreaded program towards unexplored and buggy interleavings can have pathological interactions with the synchronization operations in the program. For example, scheduling a thread that is executing busy-wait synchronization operations or arbitrary spin loops to completion with preemption bounded exploration or priority based scheduling can cause starvation. These problems make interleaving exploration of real world programs with controlled scheduling challenging.

3. NeedlePoint Scheduling Framework

Although a large number of concurrency bug detection techniques have been proposed to address the schedule selection problem [8, 13, 15, 16, 23], publicly available concurrency bug detectors are primarily data race detectors such as Helgrind [1] and Intel Thread Checker [2]. Thus, the efficacy of the various previously proposed concurrency bug detectors with respect to each other is unknown.

To address this problem, we present NeedlePoint, a unified framework for implementing a wide range of controlled scheduling approaches — ranging from simple sleep insertions to randomized prioritization — with support for repeatable execution. NeedlePoint has two overall goals. First, the NeedlePoint framework is designed to handle the real-world complexities of instrumentation for inserting scheduling points and interpreting full-fledged synchronization libraries, resulting in a tool robust enough for use in testing real-world concurrent software with state-of-the-art controlled scheduling policies. Second, NeedlePoint aims to provide researchers in concurrency bug detection a framework to build upon when designing and evaluating new scheduling policies.

3.1 Mechanisms

NeedlePoint's key contribution is the separation of mechanisms required to implement the various scheduling policies from the scheduling policies themselves. This separation of concerns between the mechanisms required to implement controlled scheduling and the scheduling policies enabled us to implement many previously proposed controlled scheduling techniques and test real-world programs with them. We identified three key mechanisms for building a wide range of concurrency bug detectors: (1) instrumenting synchronization operations, (2) identifying blocked threads, and (3) ensuring starvation freedom.

3.1.1 Instrumenting Synchronization Operations

NeedlePoint invokes the underlying scheduling policy at every dynamic program point where threads can interleave. We name these dynamic program points as schedule points. NeedlePoint uses binary instrumentation to identify such schedule points, namely all synchronization operations, atomic operations, and user specified synchronization operations. Furthermore, to detect bugs with data races, NeedlePoint can be configured to instrument every memory access and invoke the scheduling policy on such accesses. We have built NeedlePoint using the Pin [17] dynamic binary instrumentation framework running on a x86 Linux machine. Pin enables NeedlePoint to identify synchronization operations (POSIX threads API by default), atomic operations (x86 instructions with a lock prefix), and memory operations. NeedlePoint uses Pin to insert a call to the scheduling framework at these scheduling points.

3.1.2 Blocking Information

Accurately interpreting all synchronization operations becomes impractical in the presence of wide range of synchronization primitives and adhoc synchronizations [26] used by real world programs. Rather than interpreting the blocking semantics of each synchronization operation, we decided to infer the blocked status. NeedlePoint lets the synchronization operation execute and infers whether it is blocked based on the number of yields performed by other threads spin-waiting to be scheduled. This scheme is based on the following intuition: a thread executing a synchronization operation that did not block will subsequently encounter another scheduling point. In contrast, a thread trying to acquire a lock that is already held will block and thus not encounter another scheduling point. NeedlePoint counts the number of yields performed by the threads waiting to be scheduled. When this count exceeds a threshold, NeedlePoint infers that the running thread is actually blocked on a synchronization operation. This threshold is parameterizable. Setting a smaller threshold can result in an unblocked thread being erroneously considered as blocked by NeedlePoint, which can hurt repeatability. In contrast, larger thresholds cause slowdowns in the execution of the program. In our experiments, a threshold of ten yields was sufficient to ensure repeatability (for serial scheduling policies).

3.1.3 Fairness and Starvation Freedom

We found that many programs use busy wait synchronization (*e.g.* waiting for a queue to become empty and sleeping, spinning, barriers, spin loops, and others) and adhoc synchronization [26]. Scheduling a thread to completion (as in preemption bounding or as in priority based scheduling) without inducing preemptions or thread switches may cause starvation resulting in program livelocks. To avoid such starvation, NeedlePoint must choose some other thread to be scheduled. However, making a large number of such choices arbitrarily in the presence of busy wait synchronization can cause the tool to miss bugs losing the benefits of controlled scheduling. To address this problem of starvation, we make the following observation inspired from prior work on a fair scheduler [20]: a program performing busy wait synchronization will encounter numerous scheduling points, which inturn invoke the scheduling policy. Hence, if we override the default scheduling policy with a small probability, starvation will be avoided. Thus, the NeedlePoint framework overrides the scheduling policy with a small probability for a single scheduling point thereby ensuring starvation freedom. As these starvation freedom mechanisms are invoked uniformly for each schedule point rather than invoking them based on the amount of time spent waiting, NeedlePoint ensures repeatability using the same random seed.

3.2 Policies

We implemented five previously proposed concurrency bug detectors using the NeedlePoint mechanisms.

- ***Random sleep (RS)***: The random sleep policy runs more than one unblocked thread at any point in time. At every schedule point, this policy introduces a small delay with an OS sleep call with a small probability. We used a probability of 1/30 to peform the sleep as it performed well in our experiments.
- ***Preemption always (PA)***: This policy runs one thread at a time and attempts to introduce a preemption at every schedule point. At every schedule point, the scheduler performs a preemption switching the current running thread to a randomly chosen non-blocked thread.
- ***Preemption bounding (PB) [19]***: This policy performs an exploration with a predetermined number of preemptions by run-

Program	Lines of code	Schedule points	Threads	Bug type	Bug depth
Pbzip2	15,188	1210	3	Ordering violation	2
Memc	11,182	845	4	Atomicity violation	2
t+15		2300			
t+25		3400			
t+35		3900			
am		79132			
WSQ-1	541	1916	4	Atomicity violation	3
WSQ-2		1086			2
WSQ-3		1717			2
Trans	33,622	38118	2	Ordering violation	1
NSPR	1,100	5361	3	Deadlock	2

Table 1. Concurrency bugs used for NeedlePoint's evaluation that we obtained from prior research [5, 10, 16, 27, 28]. WSQ is an implementation of the work stealing queue with lock free data structures. WSQ-1, WSQ-2, and WSQ-3 are the three distinct concurrency bugs in the WSQ implementation. Memc is the memcached daemon. Trans is the Transmission BitTorrent client.

ning one thread at a time. At the beginning of the program, few scheduling points are chosen as the preemption points using a distribution. When a pre-determined preemption point is encountered, a forced preemption is induced and a non-blocked thread is chosen to become the running thread. Unlike prior research [19] that used a round robin scheme to select the next running thread when a thread is blocked, this policy chooses a random thread because we found it to be more effective at detecting bugs.

- ***AtomFuzzer (AF) [22]***: This policy directs the search to find atomicity violations by running one thread at a time. At every schedule point, if the thread attempts to acquire a lock that was previously acquired by the same thread, then it pauses the thread and schedules another thread in an effort to trigger an atomicity violation.
- ***PCT [5]***: Our prior work uses priorities to make a few random choices at the beginning of the program and direct the search. The policy assigns priorities to the thread before execution. It runs the highest priority thread that is non-blocked at every step. At predetermined schedule points, the priority of the executing thread is changed to a predetermined priority.

The NeedlePoint framework is around 6K lines of C++ code. The individual policies that implement random sleeps, preemption always, randomized version of preemption bounding [19], AtomFuzzer [24], and PCT [5] are 75, 125, 221, 208 and 258 lines of C++ code respectively, signifying the ease of writing a custom scheduler with NeedlePoint. As a proof of concept, we have tested large multi-threaded programs including the Chrome web browser with various scheduling policies in the NeedlePoint framework.

4. Evaluation of Previous Techniques

This section provides an evaluation of previously proposed techniques using the NeedlePoint framework on a common set of concurrency bugs.

4.1 Concurrency Bugs

We evaluate the effectiveness of various scheduling policies with the previously known bugs listed in Table 1. These bugs have been widely used in prior research in this area [5, 10, 16, 27, 28]. Although many bug reports provided test cases or patches that introduce sleeps at appropriate places to trigger the bug, we did not patch or modify the application to increase the likelihood of finding bugs. For the memcached bug, we designed a test harness and created various instances of the memcached bug by varying the number of memcached operations performed before performing the increment operation that has the atomicity violation. These instances are listed as $t+x$ in Table 1. We also created an instance of memcached bug where NeedlePoint introduces a schedule point before every memory access and it is listed `am` in Table 1.

4.2 Comparison of Various Scheduling Policies

To compare the bug detection abilities of various scheduling policies in Section 3.2, we ran each application listed in Table 1 with each scheduling policy 100,000 times. We checked each run to see if the bug was triggered. Figure 1 reports the number of executions in which the bug was triggered. Similarly, Figure 2 reports the number of executions that triggered the bug with the different instances of the memcached configurations created by our test harness with the various scheduling policies. Both the graphs have five bars for each bug where each individual bar represents the number of buggy executions.

The bug detection efficacy of the scheduling policies varies with the bug. Simple choices such as random sleep and preemption always are effective for some bugs. From Figure 1, we observe that the random sleep policy performs reasonably well with the memcached bug but does not detect other bugs. We found that random sleep policy works best when the test case is small and the bugs are localized. Further Figure 2 shows that the random sleep policy's bug detection ability decreases with the increase in the amount of work done before the buggy access for the memcached bug.

Figure 1 and Figure 2 show that preemption bounding is reasonably effective in triggering most of the bugs except the Pbzip bug. Preemption bounding requires more executions to trigger some bugs as they execute a large number of schedule points. Figure 1 also shows that AtomFuzzer, which is directed to find atomicity violations, performs better than preemption bounding for many of the atomicity violations.

Our prior work PCT triggers all these bugs with similar or better efficacy than other scheduling policies. Only PCT detected the Pbzip bug. Pbzip uses adhoc synchronization with spin loops to signal when the compression is over. The program crashes with a segmentation fault when the main thread frees a synchronization variable that is later used by one of the other threads. In the presence of such adhoc synchronization, the threads need to be scheduled with a few random choices to trigger the bug. A combination of priority based scheduling with a robust starvation freedom mechanisms enable PCT to trigger the Pbzip bug effectively. Table 2 summarizes the ability of the scheduling policies to trigger the bug at least once in 1000 executions on average. We observe that our prior work PCT triggers all these bugs at least once within the 1000 runs.

4.3 Deficiencies of PCT

As a result of running one thread at a time, PCT suffers from two major problems in testing long running parallel applications. First, running one thread at at time cannot leverage multicores to speedup each execution. Further even on a single core, pathological interactions with PCT scheduling decisions and the OS scheduling decisions can cause significant performance slowdowns. Second, many multithreaded applications that use adhoc synchronization and busy-wait synchronization incur large slowdowns. PCT, which uses priority based scheduling with only one thread executing at any time, causes starvation in the presence of such synchronization idioms. PCT relies NeedlePoint's starvation freedom mechanisms to make progress thereby incurring large slowdowns that effectively

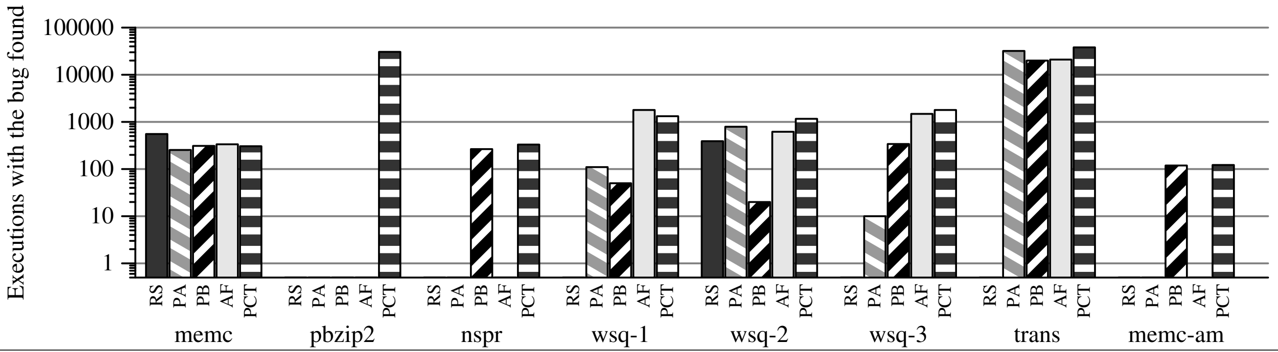

Figure 1. Bug detection abilities on common concurrency bugs for five different scheduling policies described in Section 3.2: Random Sleep (RS), Preemption Always (PA), Preemption Bounding (PB), AtomFuzzer (AF), and Probabilistic Concurrency Testing (PCT).

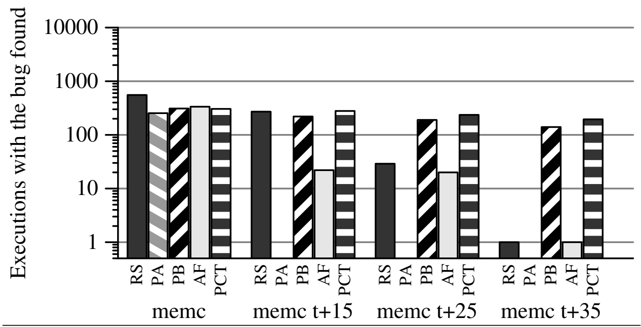

Figure 2. Bug detection ability of various scheduling policies with the variants of the memcached bug generated by the test harness.

Program	Random Sleep	Preemption		Atom Fuzzer	PCT
		Always	Bound		
Memc	Yes	Yes	Yes	Yes	Yes
Pbzip2	No	No	No	No	Yes
NSPR	No	No	Yes	No	Yes
WSQ-1	No	Yes	No	Yes	Yes
WSQ-2	Yes	Yes	No	Yes	Yes
WSQ-3	No	No	Yes	Yes	Yes
Trans	No	Yes	Yes	Yes	Yes
Memc-am	No	No	Yes	No	Yes

Table 2. Do they trigger the bugs in 1000 runs?

prohibit the usage of PCT to test such applications. To enable testing of large parallel programs with large inputs rather than unit tests, we pursued parallelization of PCT. The key contribution of our parallelization effort is that our parallel PCT (PPCT) algorithm runs multiple threads while retaining the same probabilistic guarantee of PCT running one thread at at time.

5. Parallel PCT (PPCT)

In this section we provide background on the *bug depth* metric defined by PCT and used by our PPCT algorithm to classify concurrency bugs. We subsequently provide the PPCT algorithm for a particular bug depth.

5.1 Background on PCT's Bug Depth

Concurrency bugs in multithreaded software occur when instructions are scheduled in an order not envisioned by the programmer. *Bug depth* is defined as the minimum 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 the fewest constraints. For bugs of greater depth, more orderings need to enforced by the scheduler to trigger the bug, increasing the hardness of finding it. Figure 3 shows examples of common concurrency errors with ordering constraints, represented by arrows, that are sufficient to trigger the bug. Any schedule that satisfies these ordering constraints is guaranteed to trigger the bug irrespective of how it schedules instructions not relevant to the bug. For the examples in Figure 3 the depth respectively is 1, 2, and 2. In practice, we have found that many concurrency bugs to have small depths [5].

5.2 Intuition Behind the PPCT Algorithm

Both the original PCT algorithm and the PPCT algorithm use thread priorities to probabilistically enforce ordering constraints that drive the program to an error. The key difference between the two algorithms is the number of choices they provide to the adversary, which in our case is the underlying operating system scheduler, to schedule threads at each step.

The PCT algorithm allows the adversary exactly one thread, the highest priority thread, to schedule at each step. In contrast, the PPCT algorithm maintains two sets of threads, a higher priority set and a lower priority set. At each step, the adversary is allowed to pick *any* thread in the higher priority set. If this set is empty, then the adversary is required to pick the highest priority thread in the lower priority set.

In other words, while PCT serializes the execution of all threads, PPCT serializes the execution only of the threads in the lower-priority set. The threads in the higher-priority set can be executed in parallel. Importantly, the PPCT algorithm guarantees that the number of threads in the lower priority set is bounded by the parameter d, the depth of the bug the algorithm is attempting to trigger. Thus, any implementation of the algorithm is required to control the scheduling of only at most d threads, while the operating system is freely allowed to schedule the remaining threads on multiple cores as it deems fit.

Apart from the crucial difference above, the PPCT algorithm functions exactly like the PCT algorithm, assigns random priorities to the threads and changes priorities at randomly chosen points in the execution. We start with an informal description of the PPCT algorithm in the next subsection, before giving precise pseudocode and a proof of the guarantees in Section 6.

5.3 Informal Description of the PPCT Algorithm

Given inputs: number of threads n, total number of dynamic instructions k and the depth of the bug being explored d, PPCT works as follows.

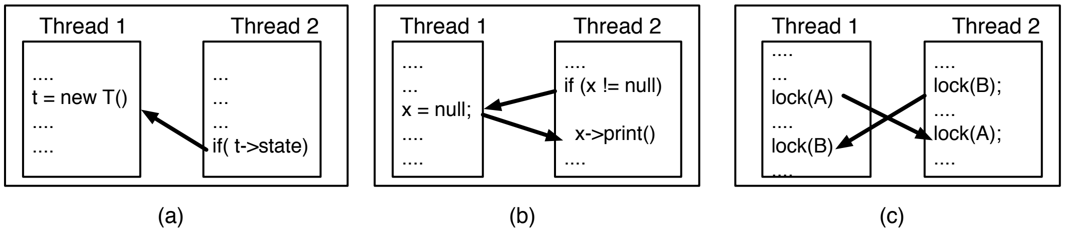

Figure 3. Three typical concurrency bugs and ordering edges sufficient to find each. (A) This ordering bug manifests whenever the test by thread 2 is executed before the initialization by thread 1. (B) This atomicity violation manifests whenever the test by thread 2 executed before the assignment by 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.

- **Pick a random low priority thread:** At the beginning of the program, pick a thread uniformly at random and assign it a priority d. In addition, insert this thread into the lower priority set L. Insert all other threads into the higher priority set H.
- **Pick random priority change points:** At the beginning of the program, pick $d-1$ priority change points $k_1,...,k_{d-1}$ in the range $[1,k]$. Each k_i has an associated priority value of i.
- **Scheduler choice:** At each step, the scheduler picks any non-blocked thread in H to schedule. If H is empty or if all threads in H are blocked, the scheduler picks the highest priority thread in L.
- **Priority change:** After a step, increment a step counter. If the step counter matches k_i for some i, change the priority of the just executed thread to i and insert it into L.

5.4 Coverage Guarantees of PPCT

Given a program with n threads that executes at most k steps, the PPCT algorithm described above finds every bug of depth d with probability at least $\frac{1}{nk^{d-1}}$ in every run of the algorithm. This probabilistic guarantee exactly matches the original PCT algorithm, despite allowing more parallelism. We explain and prove this probabilistic bound in Section 6 below.

5.5 Starvation and Priority Based Scheduling

Not allowing a lower priority thread to execute can cause starvation. In the presence of starvation, to make progress, a lower priority thread must be scheduled for a single step. We randomly choose a thread with priority lower than d to run for a single step (execute one schedule point) by bumping its priority to the highest priority in the system and reverting back the priority of the thread to the original priority when it completes executing the step. We use the mechanisms provided by NeedlePoint to identify when we should make the policy decision to avoid starvation.

6. Proof of PPCT's Probabilistic Guarantees

In this section, we give a detailed description of PPCT and present a proof of its probabilistic guarantee. We start with a high-level informal description of the proof strategy and the differences between PPCT and PCT. We then walk through the proof in detail and fully formalize all concepts (bugs, programs, schedules) in the process.

6.1 Overview

The basic idea is to find bugs by guessing a *directive* that leads to the bug. A directive is essentially a sequence of program points. We say a directive finds a bug if all schedules that follow the directive's sequencing points trigger the bug. Directives exist for all bugs (because if we restrict the scheduler enough, we can guarantee that it finds the bug). The size of the directive (i.e. the number of sequencing points it contains) reflects how hard it is to trigger the bug. We call the size of the smallest directive that can find a given concurrency bug the *depth* of that bug. As explained in Section 5.1, our intuition about typical concurrency bugs suggests that many bugs have a relatively small depth and can thus benefit from a scheduler that is tuned to do well at covering small depths.

Although we know that a directive of size d exists for all bugs of depth d (by definition), in the concurrency testing scenario we do of course not know the actual bugs and which directives will find them. It turns out, however, that PPCT has a reasonable probability of just *guessing* the directive.

In the remainder of this section, we formalize and prove this claim. The basic proof strategy is to construct and compare two schedulers:

- A randomized scheduler that schedules the program under test based on a few random choices. This scheduler has no knowledge about the structure of the program under test or the bug we are targeting.
- A directed scheduler that schedules the program under test based on an explictly supplied directive. This directive identifies points of interest in the program under test, and it prescribes an order in which they should be executed. Note that we did not actually implement such a directed scheduler: its only purpose is to construct a proof.

We then show that given a directive of size d, the randomized scheduler and the directed scheduler produce the same schedule with probability at least $\frac{1}{nk^{d-1}}$ (where n is the number of threads and k is the number of schedule points in the program). This implies that the randomized scheduler finds any bug of depth d with probability at least $\frac{1}{nk^{d-1}}$, without any knowledge about the program or the bug.

6.2 PPCT vs. PCT

Once we understood in what way PCT does in fact require serial execution (it needs to execute the sequencing points of the directive in order) and in what way it does not (threads that are not at a sequencing point can be executed in any order and in parallel), we were able to leave the original algorithm and proof almost completely intact.

Thus the presented algorithms for the randomized and the directed scheduler are exactly the same as in [5] except for one line, where the scheduler is now allowed to pick from more threads. The proof of the probabilistic guarantees is also exactly the same, except for the proof of Lemma 12, which shows that the scheduling restrictions we removed (by providing more choices to the scheduler) do not compromise the invariants of the original algorithm.

Although these changes are few in number and localized, they are difficult to understand out of context. Thus we reproduce the complete proof here.

```
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 $\mathbb{N}$
4:    $S \leftarrow \varepsilon$
      // set initial priorities
5:    for all $t \in \{1, \ldots, n\}$ do
6:      $p[t] \leftarrow d + \pi(t) - 1$
7:    end for
8:    while $en_P(S) \neq \emptyset$ do
9:      /* schedule thread of admissible priority */
10:     $t \leftarrow$ element of $en_P(S)$ such that $p[t] > d$ or $p[t]$ maximal
11:     $S \leftarrow S\,t$
        /* are we at priority change point? */
12:     for all $i \in \{1, \ldots, d-1\}$ 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 4. The PPCT randomized scheduler.

6.3 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 ε denote the sequence of length 0. For a sequence $S \in T^*$ and a number n 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 S 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 other than 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 ∞ 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 $B \subset P$.*

6.4 The Algorithm

We now introduce the PPCT randomized scheduler (Fig. 4), which is identical to the original PCT scheduler [5] except for line 10, in which the scheduler now has more choice when picking the next thread to schedule.

As in the original algorithm, 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 π).

In each iteration, we pick an admissible thread for scheduling. As in the original PCT algorithm, we can always pick the enabled thread of maximal priority to execute next. In addition, however, we also allow any thread to be picked whose priority is larger than d. [2] Once we have (nondeterministically) picked a thread t, we execute 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).

6.5 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.

6.5.1 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 $next_E(S_1,t) \neq next_E(S_1 S_2,t)$, then $t \in S_2$.*
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) $next_E(\varepsilon,t) = \bot$ 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

[2] This scheduling policy corresponds precisely to the informal description in Section 5.3, where (1) all threads with priority larger than d are in the high priority set and (2) all other threads are in the low priority set.

functions $label_E$ and $labels_E$ as follows. For $S \in P$ and $0 \le m < length(S)$, 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 $next_E(S[0,m-1],S[m]) = a$, and $label_E(S,m) = \perp$ otherwise; and $labels_E(S) = \{label_E(S,m) \mid 0 \le m < length(S)\}$).

6.5.2 Bug Depth

We now formalize the notion of ordering constraints and bug depth that we motivated earlier. Compared to our informal introduction from Section 5.1, 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. 3).

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 $d \ge 1$, 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 $1 \le i < j \le d$ 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 $1 \le i \le d$.*

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 St does not violate D.*
2. *Any complete schedule S that does not violate D does satisfy D and is in B.*

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

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

6.5.3 Coverage Theorem

The following theorem states the key guarantee: the probability that one invocation $RandS(n,k,d)$ of our randomized scheduler (Fig. 4) 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 $n \ge maxthreads(P)$, and let $k \ge maxsteps(P)$. Then*

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

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. 5) that uses the same random variables as $RandS$, but has knowledge of D and constructs its schedule accordingly.

```
Require: program $P$, $d \ge 0$
Require: $n >= maxthreads(P)$
Require: $k_1, \ldots, k_{d-1} \ge 1$
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 $\mathbb{N}$
4:      $S \leftarrow \varepsilon$
        // set initial priorities
5:      for all $t \in \{1, \ldots, n\}$ do
6:          $p[t] \leftarrow d + \pi(t) - 1$
7:      end for
8:      [ assert: $p[thread_E(b_1)] = d$ ]
9:      while $en_P(S) \neq \emptyset$ do
            /* schedule thread of admissible priority */
10:         $t \leftarrow$ element of $en_P(S)$ such that $p[t] > d$ or $p[t]$ maximal
11:         $S \leftarrow S\,t$
            /* change priority first time we peek a $b$-label */
12:         for all $i \in \{1, \ldots, d-1\}$ do
13:             if $next_E(S,t) = b_{i+1}$ and $p[t] \neq d - i$ 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 5. The directed scheduler.

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 are key to our proof construction. The proofs of these lemmas are unchanged from the original proofs [5] except for Lemma 12, and the proofs of these lemmas are included in the subsections 6.5.4 through 6.5.7 below.

LEMMA 10. *The probability that $DirS(n,k,d,D)$ succeeds is at least $\frac{1}{nk^{d-1}}$.*

LEMMA 11. *If $DirS(n,k,d,D)$ succeeds, then*

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

LEMMA 12. *If $DirS(n,k,d,D)$ succeeds, it returns a schedule that finds the bug.*

We can formally assemble these lemmas into a proof as follows. Our sample space consists of all valuations of the random variables π 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.

$$
\begin{aligned}
&\mathbf{Pr}[RandS(n,k,d) \in B] \\
&= \mathbf{Pr}[RandS(n,k,d) \in B \mid \mathcal{S}] \cdot \mathbf{Pr}[\mathcal{S}] \\
&\quad + \mathbf{Pr}[RandS(n,k,d) \in B \mid \overline{\mathcal{S}}] \cdot \mathbf{Pr}[\overline{\mathcal{S}}] \\
&\geq \mathbf{Pr}[RandS(n,k,d) \in B \mid \mathcal{S}] \cdot \mathbf{Pr}[\mathcal{S}] \\
&= \mathbf{Pr}[DirS(n,k,d,D) \in B \mid \mathcal{S}] \cdot \mathbf{Pr}[\mathcal{S}] && \text{(by Lemma 11)} \\
&= 1 \cdot \mathbf{Pr}[\mathcal{S}] && \text{(by Lemma 12)} \\
&\geq \frac{1}{nk^{d-1}} && \text{(by Lemma 10)}
\end{aligned}
$$

□

6.5.4 Auxiliary Lemmas

To prepare for proving the three core lemmas used in the proof above, we state and prove the following four auxiliary lemmas.

LEMMA 13. *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 14. *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 15. *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 16. *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 $i \in \{1,\ldots,d-1\}$. □

6.5.5 Proof of Lemma 11

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 16); 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.

6.5.6 Proof of Lemma 10

$DirS(n,k,d,D)$ 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 15, 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 individual success probabilities to obtain a total success probability for $DirS$ of at least $(1/n) \cdot (1/k)^{d-1}$.

6.5.7 Proof of Lemma 12

To prove the lemma, we will first prove that the following invariant holds during the execution of $DirS(n,k,d,D)$:

the variable S is a schedule that does not violate D.

Clearly, this invariant implies the claim of the lemma (the schedule returned in the end finds the bug), 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.

Proving this invariant for PPCT is slightly harder than for PCT, since the scheduler can pick any thread with priority larger than d, not just the highest priority thread. However, it turns out that this has no ill effect since only threads of priority d or lower are capable of breaking the invariant!

More formally, we can prove the invariant as follows, proceeding 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, but St does, so by Def. 5, there exists an i such that $next_E(S,t) = b_i$. By Lemma 13 this implies that $p[t] = d - i + 1$. Now, by Def. 6 there must exist an alternate choice $t' \in en_P(S)$ such that St' does not violate D. However, the scheduler did not pick t', but t, and t has priority at most d, so t' must have priority less than d. Therefore $p[t'] = d - j + 1$ for some $j > i$. By Lemma 14, 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.

7. Experimental Evaluation of PPCT

The goal of the experimental evaluation of PPCT is to (1) evaluate the effectiveness of PPCT with respect to bug finding and (2) understand and evaluate the runtime speedups with PPCT algorithm compared to PCT. We implemented PPCT using the NeedlePoint framework. The PPCT scheduling policy obtains the set of non-blocked threads from NeedlePoint and chooses the thread to be scheduled at any given time. PPCT ensures starvation freedom by scheduling one of the lower priority threads for a single schedule point when indicated by the NeedlePoint starvation freedom policy. Using NeedlePoint, implementing a complete PPCT scheduling policy is just 230 lines of C++ code.

7.1 Effectiveness in Finding Bugs

Figure 7.2 shows the effectiveness of PPCT in comparison to PCT in terms of finding concurrency bugs in Table 1. The left and right bars show the number of executions in which the bug was detected by PPCT and PCT respectively. The graph shows us that PPCT finds concurrency bugs either as effectively as PCT and in many cases better than PCT. This empirically validates the fact that parallel bug exploration with PPCT still retains the detection guarantees formally proved in Section 6.

We found two new bugs when we were testing multithreaded applications with PPCT. During our experimentation with Parsec benchmarks, we found a concurrency bug in the Parsec 2.0 benchmark Streamcluster. The root cause of the bug was a missing barrier that caused non-deterministic outputs. This bug was simultaneously discovered by others resulting in a patch that is distributed with the latest Parsec versions. We also found a new bug in the Transmission BitTorrent client in which a platform specific robustness assert was triggered.

7.2 Performance Evaluation

To evaluate the execution time performance overhead of testing parallel applications, we use benchmarks from the Parsec benchmark suite [4]. We also added the Pbzip2 that performs parallel compression as it is included in our bug benchmarks. We use the native inputs to run the Parsec benchmarks. We performed all the experiments on a quad-core dual-socket Intel Core 2 machine (eight cores total). In our performance experiments with multiple cores, we restrict the number of threads to be equal to the number of cores in the system.

There are three sources of runtime overhead incurred when running any scheduling policy with NeedlePoint: (1) binary instrumentation overhead, (2) overhead due to NeedlePoint's mechanism to identify blocked threads, and (3) overhead due to serializations caused by the underlying scheduling policy. Introducing a schedule point using Pin before synchronization operations and atomic operations without doing anything at these schedule points slows down the program by 2× compared to native execution on eight cores. Further, introducing a schedule point at every memory access along with the synchronization accesses introduces another 5× slowdown. These represent the overhead of performing binary instrumentation.

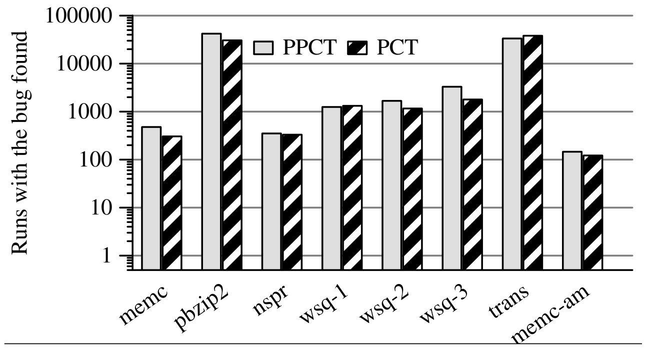

Figure 6. Bug detection of PPCT compared to PCT.

The overheads due to NeedlePoint's mechanisms for identifying blocked threads and scheduling policy serializations are dependent on the individual scheduling policies. Further, NeedlePoint mechanism's overhead for identifying blocked threads can be reduced in the presence of more information about synchronization operations. Reducing the serializations in the scheduling policy will reduce the third source of overhead. PPCT runs multiple threads and reduces the overhead due to serializations with PCT.

Figure 7 presents the execution time overhead of running PCT and PPCT over native execution on eight cores (smaller bars are better as they represent lower runtime overheads). The graph contains three bars for each benchmark. The height of the leftmost bar represents the overhead of running PCT on eight cores. On an average, the slowdown with PCT is 34× compared to the native execution on eight cores.

The height of the middle bar represents the overhead of running PPCT on a single core (which we performed by pinning all the threads in the process to a single core). It shows that running PPCT even on a single core reduces a significant portion of the overhead with PCT. This reduction is attributed to two reasons: (1) real programs have busy wait synchronization and in the presence of such constructs, the programs makes slow progress while running one thread at a time and (2) the underlying operating system scheduler has more flexibility to choose among the enabled threads to be run with PPCT when compared to PCT. On an average, PPCT on a single core incurs an overhead of 11× on a average compared to native execution on eight cores. The height of the rightmost bar in Figure 7 presents the overhead of PPCT running on eight cores. On an average, the slowdown with PPCT is 6× compared to native execution on eight cores.

PPCT serializes at most d threads when performing an exploration to trigger a bug of depth d. In contrast, PCT serializes all the threads. Figure 8 reports the speedup obtained with PPCT when compared to PCT for executions with various bug depths running on eight cores. There are four bars for each benchmark representing speedups over PCT exploration for bug depths from 1 to 4. On an average, PPCT provides 439%, 353%, 278% and 168% speedups on eight cores when compared to PCT for executions with bug depths 1, 2, 3 and 4 respectively. These results show that PPCT can provide significant speedups over PCT even for higher bug depth explorations. This speedup likely encourages the testing of long running parallel programs with PPCT even in the late stages of testing.

7.3 Testing with Real World Applications

Our goal is to enable controlled scheduling to be applied to real-world applications. However, we found PCT suffers debilitating slowdowns as a result of running one thread at a time, which prevented us from testing large real-world applications such as the Chrome web browser, the Intel Thread Building Blocks (TBB) test

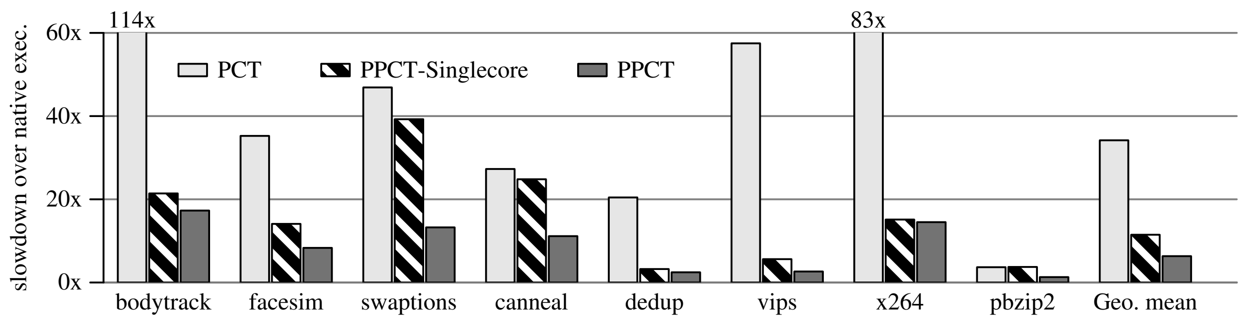

Figure 7. Runtime slowdown of PCT, PPCT on a single core and PPCT when compared to native multithreaded execution.

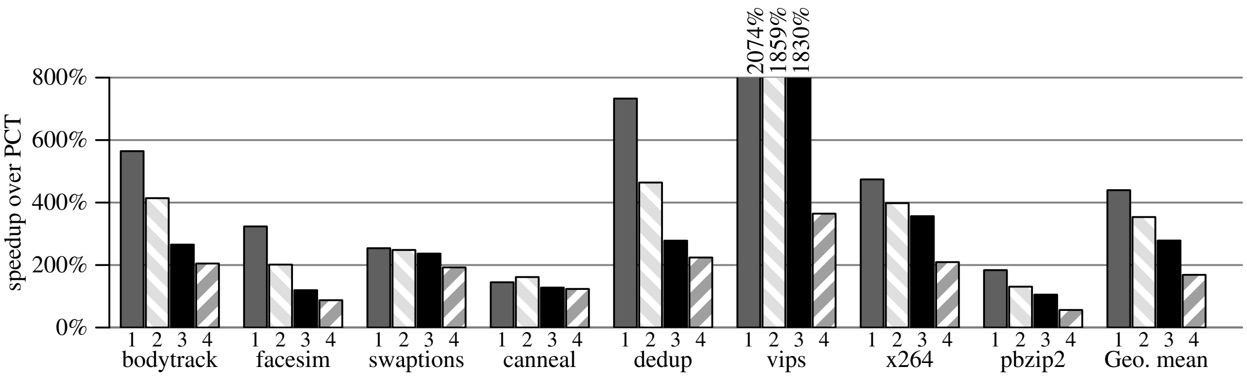

Figure 8. PPCT speedup over PCT that on a multicore machine for explorations with various bug depths. Bug depth of each PPCT exploration is indicated at the bottom of the bar.

suite, and applications using the LevelDB key value store. While experimenting with the Chrome browser, we found that Chrome creates approximately 30 threads in the single process mode, and PCT was executing the operations extremely slowly running one thread at a time. Further inspection revealed that starvation with PCT in the presence of busy-waiting synchronization also added additional slowdowns. These slowdowns resulted in browser warnings and prevented us from successfully applying PCT to Chrome. In contrast, PPCT's ability to execute multiple threads reduces the performance overhead significantly, enabling us to successfully test the Chrome web browser with PPCT without triggering browser warnings.

8. Related Work

Existing work on concurrency bug detection widely falls into best effort detection and detection with coverage guarantees. A commonly used best effort technique is stress testing where a program is run multiple times under heavy loads to trigger crashes. To introduce scheduling variety, researchers have proposed various heuristics to (1) detect suspicious activity in a program (such as variable access patterns that indicate potential atomicity violations [15, 16, 23], typestate errors [8], crashes [28] or lock acquisition orderings that indicate potential deadlocks [12]) and (2) direct the schedule towards suspected bugs [11, 22].

Other approaches such model checking [6] systematically enumerate the possible schedules either exhaustively or within some bound, such as the bound on the length of the execution [9] or the number of preemptions [21] and provide mathematical guarantees. Burckhardt et. al [5] proposed a randomized algorithm for schedule selection providing probabilistic guarantees of finding a bug in every run of the algorithm. All these prior techniques that provide guarantees run one thread at a time making them impractical for testing highly parallel applications and require mechanisms to carefully prevent starvation. PPCT is primarily motivated by the need to achieve the scalability of the stress and heuristic-directed testing without losing the ability to provide coverage guarantees.

9. Conclusion

This paper compared several previously proposed techniques for concurrency bug detection using a common set of concurrency bugs on a common framework. We found that simple policies do work well for concurrency bugs that are localized but not so well for hidden bugs. For hidden bugs, we propose PPCT, that not only detects these bugs but also detects them while running programs more than fives times faster than the serial PCT while retaining the mathematical coverage guarantees. Together, NeedlePoint and PPCT provide a framework and a scheduling policy for testing large real-world programs.

Acknowledgments

The authors thank Nicholas Jalbert and Jie Yu for providing information about RADBench and sharing the concurrency bugs respectively. We also thank the reviewers for their feedback on the paper. This paper is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-0644197 and CCF-0905464. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Helgrind: A Thread Error Detector. `http://valgrind.org/docs/manual/hg-manual.html`.

[2] Intel Thread Checker. `http://software.intel.com/en-us/articles/intel-thread-checker-documentation/`.

[3] S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66–76, 1996.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[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] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndStrike: Toward Manifesting Hidden Concurrency Typestate Bugs. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[9] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, 1997.

[10] N. Jalbert, C. Pereira, G. Pokam, and K. Sen. RADBench: A Concurrency Bug Benchmark Suite. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism (HotPar)*, 2011.

[11] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, 2009.

[12] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[13] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[15] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[16] B. Lucia and L. Ceze. Finding Concurrency Bugs with Context-Aware Communication Graphs. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[18] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *Proceedings of The 32nd ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005.

[19] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[20] M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[21] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[22] C.-S. Park and K. Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2008.

[23] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[24] K. Sen. Effective Random Testing of Concurrent Programs. In *Proceedings of 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007.

[25] K. Sen. Race Directed Random Testing of Concurrent Programs. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[26] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad Hoc Synchronization Considered Harmful. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[27] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[28] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.