6-1-2012

Multicore Acceleration for Priority Based Schedulers for Concurrency Bug Detection

Santosh Nagarakatte
University of Pennsylvania

Sebastian Burckhardt
Microsoft Research

Milo Martin
University of Pennsylvania, milom@cis.upenn.edu

Madan Musuvathi
Micrsoft Research

Nagarakatte, S., Burckhardt, S., Martin, M., & Musuvathi, M., Multicore Acceleration for Priority Based Schedulers for Concurrency Bug Detection, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2012, doi: 10.1145/2254064.2254128

Multicore Acceleration for Priority Based Schedulers for Concurrency Bug Detection

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 5x 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.

Disciplines

Computer Sciences

Comments

Nagarakatte, S., Burckhardt, S., Martin, M., & Musuvathi, M., Multicore Acceleration for Priority Based Schedulers for Concurrency Bug Detection, *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2012, doi: 10.1145/2254064.2254128

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.

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 dif cult 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 multi-threaded 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 ef cacy of the various previously proposed concurrency bug detectors with respect to each other is unknown.

To address this problem, we present NeedlePoint, a uni ed 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- edged 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 identi ed 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 speci ed synchronization operations. Furthermore, to detect bugs with data races, NeedlePoint can be con gured 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 pre x), 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 suf cient 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 bene ts 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 ve 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 as 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	Preemption		Atom	PCT
	Sleep	Always	Bound	Fuzzer	
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.

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.

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 dif cult 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) nds 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 rst 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, Veri cation

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

1. Introduction

Multithreaded programs are dif cult to test and debug because their behavior depends on the speci c 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 in uence 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 identi ed 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 uni ed 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 nding bugs. However, it also reveals that PCT's performance suffers signi cantly 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 signi cant 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.