Radius Aware Probabilistic Testing of Deadlocks with Guarantees

Yan Cai†
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
ycai.mail@gmail.com

Zijiang Yang
Department of Computer Science
Western Michigan University
Kalamazoo, MI, USA
zijiang.yang@wmich.edu

ABSTRACT
Concurrency bugs only occur under certain interleaving. Existing randomized techniques are usually ineffective. PCT innovatively generates scheduling, before executing a program, based on priorities and priority change points. Hence, it provides a probabilistic guarantee to trigger concurrency bugs. PCT randomly selects priority change points among all events, which might be effective for non-deadlock concurrency bugs. However, deadlocks usually involve two or more threads and locks, and require more ordering constraints to be triggered. We interestingly observe that, every two events of a deadlock usually occur within a short range. We generally formulate this range as the bug Radius, to denote the max distance of every two events of a concurrency bug. Based on the bug radius, we propose RPro (Radius aware Probabilistic testing) for triggering deadlocks. Unlike PCT, RPro selects priority change points within the radius of the targeted deadlocks but not among all events. Hence, it guarantees larger probabilities to trigger deadlocks. We have implemented RPro and PCT and evaluated them on a set of real-world benchmarks containing 10 unique deadlocks. The experimental results show that RPro triggered all deadlocks with higher probabilities (i.e., >7.7x times larger on average) than that by PCT. We also evaluated RPro with radius varying from 1 to 150 (or 300). The result shows that the radius of a deadlock is much smaller (i.e., from 2 to 114 in our experiment) than the number of all events. This further confirms our observation and makes RPro meaningful in practice.

CCS Concepts
• Software and its engineering→Deadlocks • Software and its engineering→Software testing and debugging.

Keywords
Deadlock, random testing, bug radius, multithreaded program

1. INTRODUCTION
Concurrency bugs widely exist in multithreaded programs [39], including data races [21], atomicity violations [29][57], and deadlocks [8][39]. Their occurrences usually involve multiple memory accesses or synchronizations (known as events in this paper) from different threads. Among these concurrency bugs, deadlocks are a kind of high level concurrency bugs caused by incorrect synchronization orders; whereas others (e.g., atomicity violations) are usually caused by wrong memory access orders. Many techniques differentiate concurrency bugs as deadlock bugs and non-deadlock bugs [35][39][47][58][65] as they require different techniques to detect. For example, ConcBugAssist [35] only focuses on non-deadlock bugs while Sherlock [19] only focuses on deadlocks.

A deadlock occurs when a set of threads are holding some locks and are waiting for other locks held by the threads in the same set [8][31][32]. There are both static [45][56][61] and dynamic approaches [8][11][15][31][46][53] to detect deadlocks. However, static approaches may report false positives as it is difficult to infer whether two events may occur concurrently [31]. Dynamic ones usually predict a set of potential deadlocks from the execution traces by identifying cycles or cyclic lock dependencies. These potential deadlocks also include many false positives and, hence, the real deadlocks should be further isolated [17][31].

Randomized testing does not rely on predicted information from concrete executions to infer potential deadlocks. Traditional randomized testing approaches [9][20] try to introduce additional randomness (e.g., random sleep) on top of OS scheduling. Other kinds of testing (e.g., heuristic directed ones [48] and systematic scheduling [43][63]) may be effective on detecting certain kinds of concurrency bugs. But they do not provide any guarantee.

PCT [12][44] then introduces mathematical randomness to provide a guarantee to trigger concurrency bugs of given bug depths. The bug depth of a concurrency is the minimal number of ordering constraints to trigger this bug [12] (see Section 2.2 for details). The innovation of PCT is to generate a scheduling prior to executing a program. The scheduling consists of a set of initial thread priorities and a set of priority change points. A priority change point is an event such that, if this event is executed, the priority of the involved thread is changed accordingly. As the scheduling of PCT is generated purely based on the mathematical randomness, it probabilistically guarantees to detect a concurrency bug with bug depth of $d$ at a probability of $1/(n \times k^{d-2})$, where $n$ and $k$ are the approximated number of threads and the approximated number of events, respectively, of the given program.

However, PCT assumes that the events of a concurrency bug are uniformly distributed among all events. Therefore, it randomly selects thread priority change points among all events. Hence, if the bug depth increases by one, the guaranteed probability decreases to be $1/k$ times. For real-world programs, the value of $k$ could be very large, which makes PCT ineffective.

We interestingly found that all events involved in a concurrency bug usually fall into a short range, which is also suggested by

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

AASE’16, September 3–7, 2016, Singapore, Singapore
ACM, 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970307
356
existing works (e.g., short depth [17][39] and small scope hypothesis [42]). The number of events in the range is much smaller compared to the total number of events of a program. Besides, existing works also show that deadlocks usually involve more ordering constraints than other concurrency bugs [39]. For example, a data race only involves two events (i.e., memory accesses) and an atomicity violation only involves three; however, a deadlock must involve at least four direct events and other indirect events [17]. Therefore, random selection of events among all events could be ineffective for PCT to trigger a concurrency bug, especially for deadlocks.

In this paper, we propose a new approach RPro (Radius aware Probabilistic testing) to generate execution. Given a bug depth, RPro selects the first priority change point randomly among all. Let's define this event as \( k_i \) (in term of the order of events to be executed). However, for the remaining priority change points, RPro selects them out of a certain range of event \( k_i \). The range is defined to be \([k_i - r, k_i + r]\) excluding the \( k_i \) itself, where the number \( r \) is referred as the Radius of the concurrency bug. In this way, all priority change points are within the range \([k_i - r, k_i + r]\).

Hence, RPro guarantees to trigger a concurrency bug with a probability of \( 1/(n \times k \times r^{d-2}) \) which is \((k/r)^{d-2}\) times the guarantee by PCT, if all the events of the bug fall within the range, where \( k \) and \( n \) are the number of events and the number of threads of the given program, respectively.

In theory, the radius of a bug could be as large as the number of the total events. Even a concurrency bug has very large radius, the same bug can be manifested with a much smaller radius. That is, a concurrency bug that must have very large distance (e.g., \( > 1,000 \)) is rare. After an event is selected, assuming the same concurrency bug can be triggered by selecting another event at either distance \( r_1 \) or \( r_2 \) from all events, where \( r_1 < r_2 < k \) and \( k \) is the number of total events. By randomly selecting an event, the probability to trigger the event is \( 1/k + 1/k = 2/k \) (corresponding to selecting either \( r_1 \) or \( r_2 \)). However, if we restrict the radius to be \( r \), where \( r < r_1 < r_2 \), the probability becomes \( 1/r \) (corresponding to only select \( r_1 \)), which can be significantly larger than \( 2/k \).

In practice, this radius is usually much smaller than the number of total events (e.g., from 50 to 120 vs 400,000 or more events in our experiments).

We have implemented RPro and evaluated it on a set of real-world benchmarks. We run both PCT and RPro on each benchmark for 10,000 times. In the experiment, we configured two radiuses 10 and 50 to RPro. The experimental result shows that, with either radius, RPro triggered each deadlock with a much higher probability. That is, on average, RPro achieved \( \geq 7x \) probabilities of that by PCT. Particularly, on two large-scale benchmarks (i.e., two versions of MySQL), PCT failed to trigger any deadlock; whereas RPro triggered the two deadlocks with probabilities of 0.0004 and 0.0055 (with radius 10) and 0.0018 and 0.0041 (with radius 50).

We also configured RPro with radius from 1 to 150 (or 300). The experimental results show RPro achieved the highest probability with the radius at most 114, even the events of some benchmarks could be \( \geq 4 \times 10^6 \). This shows that, in practice, a deadlock usually has a smaller radius (e.g., 114 vs \( 4 \times 10^6 \)).

The main contributions of this paper are as follows:

1. It proposes a new approach RPro toward randomized testing of deadlocks with probabilistic guarantees. Compared to PCT, RPro has a larger probabilistic guarantee to trigger a deadlock with depth of three or more.
2. We have implemented RPro as a prototype tool to trigger deadlocks. The experiment results demonstrate the effectiveness of RPro compared to PCT. It also shows that the radius of a concurrency bug is usually much smaller compared to the number of events in a program.

The rest of this paper. Section 2 presents the preliminaries and the motivation of our work. Section 3 presents our RPro approach. Section 4 presents an experiment to evaluate RPro. Section 5 discusses the related works, followed by the conclusion in Section 6.

2. PRELIMINARIES AND MOTIVATIONS

2.1 Events and Deadlocks

An execution of a multithreaded program involves the following kinds of events:

- \( \text{acq}(t, m) \): A thread \( t \) acquires a lock \( m \), \( \text{acq}(m) \) for short.
- \( \text{rel}(t, m) \): A thread \( t \) releases a lock \( m \), \( \text{rel}(m) \) for short.
- others (e.g., memory read and write)

An execution trace (or a trace) is a sequence of events. A deadlock occurs if a set of threads wait mutually for a set of locks that are held by other threads in the same set [15][32]. For example, Figure 1 shows an example program with a deadlock \( D_1 \). The program has two threads \( t_1 \) and \( t_2 \) that totally execute eight events (i.e., lock acquisitions and releases on locks \( m \) and \( n \)). Deadlock \( D_1 \) occurs if: thread \( t_1 \) tries to acquire lock \( n \) after it acquired lock \( m \) but thread \( t_2 \) tries to acquire lock \( m \) after it acquired lock \( n \). The two threads then mutually wait for each other to release a lock.

2.2 The PCT Algorithm

We firstly explain the concept bug depth [12] of a concurrency bug. A concurrency bug occurs if the involved events occurring under a certain interleaving. In other words, there is one or more ordering constraints among these events. If a concurrency bug could occur in a way such that the number of required constraints is minimal, we say the bug depth of this bug is the number of these constraints. For example, deadlock \( D_1 \) in Figure 1 occurs only if both (1) the event \( \text{acq}(m) \) at site \( s_{01} \) occurs before the events \( \text{acq}(m) \) at site \( s_{05} \) and (2) the event \( \text{acq}(n) \) at site \( s_{03} \) occurs before the event \( \text{acq}(n) \) at site \( s_{05} \), as indicated by two arrows. The two conditions are known as two ordering constraints of deadlock \( D_1 \). If only one of two ordering constraints is satisfied, deadlock \( D_1 \) cannot occur. Therefore, the bug depth of deadlock \( D_1 \) is 2.

PCT firstly generates a scheduling prior to executing a program. During runtime, it enforces an execution to follow its generated scheduling. Hence, PCT can mathematically explore the interleaving to trigger a concurrency bug of given bug depths. Therefore, it can guarantee a probability to trigger a concurrency bug.

Algorithm 1 outlines the core idea of PCT. PCT accepts a program \( P \) and the following parameters over \( P \): a parameter \( n \) indicating the approximate number of threads, a parameter \( k \) indicat-
Algorithm 1: PCT(P, k, n, d)
Input P: a given program.
Input n: an approximation of the total number of threads.
Input k: an approximation of the total number of events.
Input d: a bug depth of the targeted concurrency bug.

1. Assign the n priority values d, d + 1, ..., d + n − 1 randomly to the n threads. (A larger number indicates a larger priority.)
2. Pick d − 1 unique random priority change points k₁, ..., kₕ₋₁ in the range [1, k]. These d − 1 priority change points are ordered and each kᵢ has an associated priority value of i.
3. Schedule the k threads by their priorities assigned in step 1 and count the events executed. After executing an event:
   3.1 If the count becomes kᵢ, change the priority of the current thread to be i.
   3.2 If a concurrency bug occurs, report the bug.

PCT is based on priorities to schedule all threads. In Step 1, PCT randomly assigns n priorities d, d + 1, ..., d + n − 1 to the n threads, where a larger number indicates a larger priority. The lowest d − 1 priorities 1, 2, ..., d − 1 are reserved for runtime scheduling. Next (Step 2), from all k events (labeled from 1 to k), PCT randomly selects d − 1 unique events k₁, ..., kₕ₋₁ as the priority change points. Each event kᵢ is associated with a priority value of i. Totally, the d − 1 priorities are 1, 2, ..., d − 1. Finally (Step 3), PCT executes the given program and changes the priorities of all threads as follows:

- PCT only executes events from the thread with the largest priority, and counts all executed events (starting from 1).
- If an event is counted as the number k’ and k’ is equal to kᵢ out of k₁, ..., kₕ₋₁, PCT changes the priority of the current thread to be i. (The priority change brings a chance for other threads to be executed.)

PCT repeats the above procedure until the execution terminates. In practice, the scheduling generated by PCT is not always feasible. For example, PCT may first schedule a thread that is not forked by the main thread or is disabled by operating system. In this case, PCT schedules a thread with the largest priority out of all enabled threads.

Illustration of PCT. We show an illustration of PCT on our example in Figure 1. The program contains 2 threads and 8 events (i.e., n = 2 and k = 8). Its deadlock Dₙ has a bug depth d = 2 as indicated by two solid arrows. Therefore, PCT randomly assigns two priorities 2 and 3 (i.e., d, ..., d + n − 1) to the two threads. If then randomly selects 1 (i.e., d − 1) priority change point and this priority change point is reversed for runtime priority change. Suppose that the randomly assigned priorities of threads t₁ and t₂ are 3 and 2 (indicated by the symbols ◊ and ⊙ in Figure 1), respectively, and the change points is 1 (i.e., right after executing the 1ˢᵗ event of thread t₁, indicated by the symbol Θ). Then, PCT generates a scheduling for the example program toward triggering deadlock Dₚ, as indicated by the dotted (red) arrow in Figure 1. (Please ignore whether the generated scheduling is feasible or not as discussed in the last paragraph). During execution, PCT follows its schedule to execute the program. It firstly schedules thread t₁ (i.e., the thread with the largest priority) to execute. After executing the event acq(m) at site sₙ₁, PCT counts the executed events, where the count is 1 that is equal to the selected change point 1. Hence, PCT changes the priority of thread t₁ to be the 1ˢᵗ reserved priority that is 1. And the priorities of threads t₁ and t₂ are 1 and 2, respectively. PCT then schedules thread t₂ (that has a larger priority) to execute its events and counts the executed events. As there is only one priority change point which has been consumed by thread t₁, no priority of any thread would be changed.

That is, in order to trigger deadlock Dₙ in the example, PCT should select the first event of the thread with the largest priority that are randomly assigned. The probability of this selection is (1/2) × (1/8) = 1/16, corresponding to select the first event (with the probability of 1/8) of the thread with a largest priority (with the probability of 1/2).

Intuitively from the above illustration, PCT targets to select a set of priority change points that could form a minimal set of ordering constraints to trigger a concurrency bug. Formally, given a program P that contains n threads and executes k events, PCT is able to find a concurrency bug of depth d with a probability at least 1/(n × kᵈ⁻¹) [12].

2.3 Motivations
PCT can avoid exploring similar interleaving mathematically, resulting in relatively effective scheduling with repeatedly executing the same program. And it also has a probabilistic guarantee to find concurrency bugs of given bug depths.

However, PCT utilizes the randomized generation of priority change points. The randomized generation may become ineffective in practice if the bug depth is 3 or even larger. For example, to detect data races or atomicity violations, the required bug depths are usually 1 or 2 [12], respectively. If the bug depth is 1 or 2, the guaranteed probability is 1/n or 1/(n × k), respectively. However, for real-world deadlocks, such bug depths could be 3 or larger. In this case, PCT might become ineffective as its guaranteed probability decreases by a factor of 1/k if the bug depth increases by one. For example, if the bug depth is 3, the guaranteed probability becomes 1/n × k². For real-world programs, the number of events (i.e., k) is usually large; and hence, the guaranteed probability by PCT becomes much smaller. We illustrate this by another real-world example shown in Figure 2.

The program in Figure 2(a) is adapted from a real-world deadlock program (i.e., JDBC Connector 5.0 with bugID=2147). It contains two threads t₁ and t₂ that may form a deadlock Dₚ on two locks p and n (at sites s₁₀, s₁₉ of thread t₁ and sites s₁₅ and s₁₆ of thread t₂) as shown in bold. Figure 2 (b) and (c) shows two different scheduling to trigger deadlock Dₚ. Unlike deadlock Dₚ in Figure 1, deadlock Dₚ requires more ordering constraints (i.e., the two dotted arrows from site s₁₄ to site s₁₁ and from site s₁₁ to site s₁₅) to be triggered, besides the two ordering constraints between the events directly involved in deadlock Dₚ (i.e., the two solid arrows from site s₁₀ to site s₁₉ and from site s₁₅ to site s₁₈) that is, if thread t₁ is firstly scheduled to execute as shown in Figure 2(b), after executing the event rel(k) at site s₁₀, thread t₂ should be scheduled to execute the two events acq(s) and rel(s) at sites s₁₅.
Otherwise, if thread t1 is allowed to execute events from site s01 to site s07 (i.e., all events before site s08), thread t2 cannot reach site s13 to execute event acq(n) as lock s is being held by thread t1. Figure 2(c) also shows the second case if thread t2 is firstly scheduled to execute, where thread t1 should be scheduled to execute right before thread t2 executes event acq(s) at site s14.

For PCT, if we follow deadlock D1 to set up a bug depth of 2 (i.e., by only considering the two solid arrows in Figure 2(a)), it never triggers deadlock D2 theoretically as the minimal number of ordering constraints to trigger deadlock D2 is 3, corresponding to the scheduling shown in Figure 2(b). Even if we configure PCT to work with the bug depth of 3, it has to assign thread t2 to have a larger priority initially. Note that, due to symmetric property, PCT could trigger deadlock D1 with any one of two initial assignments of two priorities to two threads under the bug depth of 2. That is, besides the priority assignments and change point selection shown in Figure 1, PCT can trigger deadlock D1 by assigning priorities 3 and 2 to threads t2 and t1, respectively, and selecting a priority change point at site s05 as shown in Figure 3. The generated scheduling is denoted by a dotted (red) arrow. However, not all concurrency bugs have a symmetric property to be triggered, such as data races.

Besides, PCT treats all events uniformly. However, in practice, a concurrency bug usually has a short depth to be exposed [17][39][42]. In other words, if one priority change point is selected, the remaining priority change points should not be randomly selected in practice. In Section 3, we introduce how our algorithm selects more effective priority change points to trigger concurrency bugs as well as the rationales.

Lastly, the generated scheduling of PCT is meaningful in theory. During real execution, PCT has to resolve various thrashing. For example, Figure 4 shows that a scheduling generated by PCT indicated by the dotted (red) arrow, where the priority change point (marked as 1) is right after evt2. The actual scheduling (denoted by the solid (green) arrow) differentiates the generate scheduling as there is a pair of wait(m) and notify(m). This pair of events requires that thread t1 has to wait after executing evt2 until thread t2 sends a notification to thread t1. Such kinds of synchronizations are very common [21][39]. But it is difficult to be considered by PCT, which may further reduce the probability of PCT to trigger concurrency bugs.

Figure 2. An example program p and with a deadlock D2, adapted from the deadlock of JDBC Connector 3.0 with bugID=2147.

Figure 3. The second way for PCT to trigger deadlock D2.

Figure 4. Comparison between a generated scheduling by PCT and an actual scheduling.

3. RPRO ALGORITHM

This section presents our algorithm RPro. We firstly present the design rationales of RPro, followed by the RPro algorithm and its probabilistic guarantee.

3.1 Rationales of RPro

PCT is designed not to consider any program information, except the basic program statistics (i.e., the approximations on the number of threads and the number of events). It treats each thread and each event uniformly. Therefore, for any given bug depth d, it randomly selects d – 1 priority change points (Section 3.3 discusses why PCT selects d – 1 but not d priority change points). This selection is fair to all events. We illustrate this selection strategy in Figure 5(a). In Figure 5, the arrows under different threads indicate the events of each corresponding thread. The solid black circles indicate the events selected as priority change points and the (red) dotted arrows indicate the communications among all threads. The dotted area depicts the range among which PCT or our RPro selects priority change points.

By following PCT, the selected priority change points are uniformly distributed among all events, as shown in Figure 5(a).
However, our observation is that a multithreaded program is not arbitrarily designed and developed. For example, there are many synchronization primitives to coordinate different threads of a program, such as `wait()/notify()/barrier()` operations [21] as well as various customized conditionals [31]. That is, a concurrency bug is unlikely to involve two or more events that have an execution distance (in term of the number of events) as long as the total number of events in the program. Contrastly, for a concurrency bug, the execution distance of its events may be centralized such that if one of its event occurs, the other events of the same bug may also occur after or before several other events [17][39]. Figure 5(b) illustrates this kind of scenarios: if an event occurs, all other events of the same concurrency bug fall into the $r$ events of the first event.

In this paper, we define the **Radius** of a concurrency bug to be: the largest distance of every two events involved in the bug during execution, where the distance of two events is $1 + x$ and $x$ is the number of events executed or to be executed immediately (i.e., the next event of a thread) between the two events. From this definition, the radius of a concurrency bug is interleaving sensitive. In Figure 2(b), by following the scheduling indicated by green arrows, the radius of deadlock $D_2$ is 9 as the trace is $(\ldots, s14, s15, s04, s05, s06, s07, s08, s16, \ldots)$. Note that, although the event $acq(n)$ at site $s15$ is executed as the second last executed event of deadlock $D_2$, it is the next event to be executed after the execution of $rel(s)$ at site $s14$. Therefore, the event $acq(n)$ at site $s15$ (underlined in the trace) is the first event to trigger deadlock $D_2$ in the execution in Figure 2(b). For the scheduling in Figure 2(c), the radius of deadlock $D_2$ is 11 where the corresponding trace is $(\ldots, s14, s01, s02, s03, s04, s05, s06, s07, s08, s16, \ldots)$.

Our observation is that, during an execution, threads are actually synchronized to execute their events with similar pace. Let's consider the example in Figure 4 again. Suppose that the two events $evt2$ and $evt3$ form a concurrency bug. (This kind of bugs widely exist in real-world programs [31], e.g., shared conditionals to control whether the next `wait(m)` or `notify(m)` should be executed.) Then, the two events are actually very close: one is right after the `wait(m)` event and the other is right before the `notify(m)` event; and the two events `wait(m)` and `notify(m)` are expected to execute almost at the same time (as the execution of `wait(m)` has to be suspended until the `notify(m)` is executed).

Now, let's re-consider the real-world deadlock $D_2$ in Figure 2 again. Both schedulings in Figure 2 (b) and (c) can successfully trigger deadlock $D_2$. When the deadlock occurs, the distance of the involved events is 9 or 11. However, the program containing this bug actually involves more than 5,000 additional events (see the experiment in Section 4) not related to this bug. Compared to the total number of events, the radius of 9 or 11 is much smaller. Therefore, if the priority change points of a scheduling could be selected within the radius of a concurrency bug, the probability to trigger this bug could be significantly improved.

Let's consider the scheduling in Figure 2(c). There are two priority change points as indicated by $\bullet$ and $\circ$. The probability for PCT to select the two points is actually: $(1/k_2) \times (1/k_1) = 1/k^2$ as PCT selects them independently, where $k$ is the number of total events. However, if considering the radius (denoted as $r$) of deadlock $D_2$, the probability would be $(1/k_2) \times (1/r) = 1/(k \times r)$ as: once the first priority change point is selected to be one of events of deadlock $D_2$ (corresponding to a probability of $1/k$), the remaining one is selected within the $r$ events from the first one (corresponding to a probability of $1/r$). Therefore, the probability to select the right priority change points could be improved to be $k/r$ times of what PCT guarantees. For deadlock $D_2$, as the radius $r = 11$ and there are actually more than 5,000 events, the above probability could be improved to be $5000/11 \approx 455$ times larger.

### 3.2 RPro Algorithm

To improve the probability of PCT, we present a new approach based on our bug radius concept, known as **RPro**, namely **Radius aware Probabilistic testing**.

RPro algorithm is straightforward, as shown in Algorithm 2. The difference between PCT and RPro is highlighted (i.e., steps 2 and 3 in Algorithm 2). RPro takes a program $P$ with the approximated number of threads and events (i.e., $n$ and $k$), as well as the bug depth $d$ and the radius $r$ of the targeted concurrency bug. Prior to executing the program $P$, RPro selects a random priority change point $k_i$ among all $k$ events (step 2) like PCT. Then, it selects the remaining $d – 2$ priority change points $k_2, \ldots, k_{d-1}$ within the $r$ events starting from $k_i$ (i.e., the range $[k_i – r, k_i) \cup (k_i, k_i + r]$) (step 3). During runtime scheduling, it adopts the same scheduling strategy as that of PCT to trigger the targeted bug.

![Figure 5. Two distribution models of bug events.](image-url)
3.3 Guarantee and Limitations of RPro

In this subsection, we present an analysis on the probabilistic guarantee of RPro on triggering concurrency bugs. We firstly present Lemma 1 to show the formal guarantee of PCT.

Lemma 1. Given a concurrency bug of depth \( d \) from a program that produces at most \( n \) threads that totally execute at most \( k \) events, PCT guarantees to trigger this bug with a probability of \( 1/(n \times k^{d-1}) \). And this probability is \( 1/n \) times of the probability by selecting \( d - 1 \) ordered events among all \( k \) events (i.e., \( 1/k^{d-1} \)).

Proof. See the proof of PCT [12]. □

It is interesting that the guaranteed probability of PCT is \( 1/(n \times k^{d-1}) \) but not \( 1/k^d \). From the PCT algorithm (i.e., Algorithm 1), we know that PCT only selects \( d - 1 \) but not \( d \) priority change points, where \( d \) is also the number of minimal ordering constraints to trigger the same bug. Actually, the first ordering constraint is enforced by the initial priorities randomly assigned to all threads.

Let’s consider the example in Figure 4 again and suppose that a concurrency bug only requires 1 ordering constraint from \( e_{vt1} \) to \( e_{vt2} \). Then, PCT only needs to assign a larger priority to thread \( t1 \) without selecting any priority change point (i.e., \( 1 - 1 = 0 \)). This probability is \( 1/(n \times k^{1-1}) = 1/n \) where \( n = 2 \) is the number of threads of the program. One may refer the detailed proof in [12] to find more about the proof of Lemma 1.

Theorem 1. Given a concurrency bug of depth \( d \) (\( d \geq 2 \)) from a program that produces at most \( n \) threads that totally execute at most \( k \) events, if the radius of this bug is less than or equal to \( r \) (\( d - 2 \leq r < k \)), RPro guarantees to trigger this bug with a probability of \( 1/(n \times k \times r^{d-2}) \).

Proof sketch. We prove Theorem 1 based on PCT algorithm and Lemma 1. Like PCT, RPro selects the first priority change point randomly from all \( k \) events with a probability of \( 1/k \). However, for the remaining \( d - 2 \) priority change points, RPro selects them only within the \( r \) events of the first priority change point, with a probability of \( 1/r^{d-2} \). As the all events of the given bug are within the \( r \) events \( \text{of any event from this bug} \), the probability of RPro to exactly select all the priority changes is \( x \) times of the probability of PCT, where \( x \) is:

\[
x = \left(\left(\frac{1}{k}\right) \times \left(\frac{1}{r^{d-2}}\right)\right) \div \left(\frac{1}{k^{d-1}}\right) = \left(\frac{1}{k}\right)^{d-2}.
\]

By Lemma 1, PCT guarantees to trigger the given bug with a probability of \( 1/(n \times k^{d-1}) \), which is \( 1/n \) of that probability by selecting \( d - 1 \) ordered events out of \( k \) events. Therefore, RPro guarantees to trigger the given bug with a probability of:

\[
(1/n) \times \left(\left(\frac{1}{k^{d-1}}\right) \times \left(\frac{k}{r}\right)^{d-2}\right) = 1/(n \times k \times r^{d-2}).
\]

Theorem 1 is proved. □

Theorem 1 shows that RPro is more effective than PCT if the bug radius \( r \) is smaller than the total number of events \( k \). Of course, PCT can be viewed as a special case of RPro where the radius is the number of all events (i.e., \( r = k \)). Another point that should be mentioned is that, with increasing depth value \( d \), the guaranteed probability by RPro decreases much slower than that by PCT.

Discussion on the bug radius of RPro. PCT relies on the approximated program execution information to generate scheduling but does not consider the practical features of concurrency bugs, especially the deadlocks. Whereas, RPro takes this into consideration. It is based on the radius of a concurrency bug to generate scheduling. However, like the bug depth, this radius is also unknown until a concurrency bug is detected. Therefore, if RPro takes a smaller radius than the actual radius of the bug, it may fail to select the right set of priority change points; hence, it may fail to trigger any concurrency bug. And its guaranteed probability becomes zero. If RPro takes a too large radius value than the actual radius of the targeted concurrency bug, its guaranteed probability may also decrease. For example, in the worst case, given that \( r = k \), RPro would have the same guaranteed probability as that of PCT. Figure 6 shows such a comparison, where \( x \)-axis is the value of radius \( r \) and the \( y \)-axis is the guaranteed probability. We use \( r_{bug} \) to denote the actual radius of a concurrency bug. As PCT is unaware of bug radius, it always has the same probabilistic guarantee. For RPro, it guarantees either a probability of zero if the radius is less than \( r_{bug} \) or a larger probability if the radius is from \( r_{bug} \) to the number of events (i.e., \( k \)). In practice, even if the given radius is less than \( r_{bug} \), RPro may still trigger occurrences of a concurrency bug. This also applies to PCT because their guaranteed probabilities are only the low bounds [12] and has been verified in our experiments (see the Observation 1 in Section 4.3.2). In Figure 6, we also plot the practical probability of RPro.

3.4 Optimization for Deadlock Triggering

RPro is designed for triggering concurrency bugs with larger bug depth. However, in this paper, we focus on deadlocks. And a deadlock occurs only after the involved threads try to acquire some locks but these locks are already acquired by the same set of threads. That is, an acquisition may result in a deadlock occurrence but a release event cannot directly result in a deadlock occurrence. Therefore, RPro considers lock acquisition events but discards lock release events. Hence, RPro only needs to select priority change point among the half of events of a program (as a release event is paired with an acquisition event).

3.5 Limitations

RPro considers the bug radius to select priority change points. However, it still suffers from several limitations. Firstly, it might be difficult to find a proper radius value in practice, although this value is usually much smaller compared to the total number of events. Secondly, compared to PCT, RPro restricts its priority change points into a smaller range; hence, it may not be able to expose those bugs with a large radius, although the probability for PCT to find such a bug is small. Lastly, like PCT, RPro also requires a larger number of executions to exhibit its effectiveness as...
both of them have probabilistic guarantees. From the last two points, there is a tradeoff between the probability and the number of executions (i.e., cost) to find a concurrency bug.

4. EXPERIMENT

4.1 Benchmarks

We collected a set of widely-used real-world benchmarks, including: one Java program (i.e., JDBC Connector 5.0 [3]) and five C/C++ programs (i.e., Hawknl [1], SQLite [7], and three versions of MySQL Database Server [4]). There are totally 10 test cases and 1 unique deadlock, covering most of deadlocks cases [39]. All these benchmarks and their test cases have been used in previous works multiple times [15][22][33][59] and are available either online [1][4][5][7] or from the previous works [33][59].

Table 1 shows the statistics of all benchmarks, including benchmark names with version numbers (if available), Bug IDs (if available), program size (SLOC [6]), the numbers of threads of each benchmark ("#prog") and its threads in each unique deadlock ("#dlsk"), the number of deadlocks ("#dlsks", as a deadlock may have two or more variants) in each benchmark. The next column shows the total number of events ("# events") produced in the execution. The last column shows the descriptions on how each deadlock occurs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bug ID</th>
<th>SLOC</th>
<th># threads (prog/dlsk)</th>
<th># dlsk</th>
<th># events</th>
<th>Deadlock descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC-1 (5.0)</td>
<td>2147</td>
<td>36.3K</td>
<td>3/2</td>
<td>4</td>
<td>5,090</td>
<td>Statement.executeQuery() and Connection.prepareStatement()</td>
</tr>
<tr>
<td>JDBC-2 (5.0)</td>
<td>14927</td>
<td>36.3K</td>
<td>3/2</td>
<td>1</td>
<td>5,088</td>
<td>PreparedStatement.getWarnings() and Connection.close()</td>
</tr>
<tr>
<td>JDBC-3 (5.0)</td>
<td>31136</td>
<td>36.3K</td>
<td>3/2</td>
<td>1</td>
<td>5,050</td>
<td>Connection.prepareStatement() and Statement.close()</td>
</tr>
<tr>
<td>JDBC-4 (5.0)</td>
<td>17709</td>
<td>36.3K</td>
<td>3/2</td>
<td>2</td>
<td>5,080</td>
<td>PreparedStatement.executeQuery() and Connection.close()</td>
</tr>
<tr>
<td>Hawknl (1.6b3)</td>
<td>n/a</td>
<td>9.3K</td>
<td>3/2</td>
<td>1</td>
<td>33</td>
<td>NShutdown() and Nclose()</td>
</tr>
<tr>
<td>SQLite (3.3.3)</td>
<td>1767</td>
<td>74.0K</td>
<td>3/2</td>
<td>2</td>
<td>16</td>
<td>sqlite3UnixEnterMutex() and sqlite3UnixLeaveMutex()</td>
</tr>
<tr>
<td>MySQL-1 (6.0.4a)</td>
<td>34567</td>
<td>1,093K</td>
<td>16/2</td>
<td>4</td>
<td>19,300</td>
<td>Alter on a temporary table and a non-temporary table</td>
</tr>
<tr>
<td>MySQL-2 (6.0.4a)</td>
<td>37800</td>
<td>1,093K</td>
<td>17/2</td>
<td>1</td>
<td>15,066</td>
<td>Insert and Truncate on a same table using falcon engine</td>
</tr>
<tr>
<td>MySQL-3 (5.5.17)</td>
<td>62614</td>
<td>1,282.7K</td>
<td>22/2</td>
<td>2</td>
<td>406,117</td>
<td>FUGE BINARY LOG acquires locks in wrong order</td>
</tr>
<tr>
<td>MySQL-4 (5.1.57)</td>
<td>60682</td>
<td>1,146.7K</td>
<td>19/3</td>
<td>6</td>
<td>444,621</td>
<td>SHOW INNODB STATUS deadlocks if LOCK_thd_data_points to LOCK_open</td>
</tr>
</tbody>
</table>

RPro requires a radius value. We selected two values 10 and 50 as two radiauses for RPro and refer them as $RPro_{10}$ and $RPro_{50}$, respectively. We firstly run each benchmark for 10,000 times under each algorithm to collect the probabilities of each algorithm to trigger the corresponding deadlocks from each benchmark. And the probability is computed to be the ratio of the number of the runs triggering deadlocks out of all 10,000 runs. Note that some benchmarks may contain one or more variants of the same deadlock; and we treated these variants as the same deadlock.

To evaluate whether a short value of the radius is required to trigger deadlocks in practice, we further configured RPro to run additional 10,000 times under each of the radiauses from 1 to 150 to calculate the corresponding probabilities except on MySQL-1 and MySQL-3. On these two benchmarks, we run them with radiauses from 1 to 300 in order to show a clearer trend of the probability changes by RPro.

4.3 Result Analysis

This section presents our analysis on the experimental result by comparing $RPro_{10}$, $RPro_{50}$, and $PCT$ on their probabilities to trigger each deadlock. Finally, we analyze the effectiveness of RPro with different radius values (i.e., 1 to 150 or 300).

4.3.1 Effectiveness Comparisons

Table 2 shows the probabilities of three algorithms on each benchmark. The first column shows the benchmark name. The second major column ("Program parameters") lists the program information (i.e., the number of events $k$, the number of threads $n$, and the bug depth $d$). These three parameters are collected from executions and are used as the approximations of three. The sub-column ("Guaranteed probabilities") shows the guaranteed probabilities of three. The third major column lists the collected probabilities of $PCT$, $RPro_{10}$, and $RPro_{50}$, as well as the ratio of probability increases ($\Delta$) of $RPro_{10}$ and $RPro_{50}$ to the probabilities of $PCT$. The probability increase is calculated by the following formula: $\Delta = \frac{(RPro_{10} - PCT) + \frac{PCT \times 100}{x}}{\text{where } x = 10 \text{ or } 50}$. The last row also shows the average improvements on the probabilities of $RPro_{10}$ and $RPro_{50}$ to trigger all deadlocks. We also use the symbol $\mathbf{J}$, $\mathbf{L}$, and $\mathbf{V}$ to indicate whether the probability increased, decreased, and did not changed, respectively, in the two delta columns (i.e., $\Delta$).

From the program parameters shown in Table 2 and the program description shown in Table 1, these programs are representative as they include large-scale ones. For example, MySQL-3 has more than 1,000,000 lines of code and its executions in our experiment produce more than 20 threads and more than 400,000 events.
From the second major column in Table 2, we observe that these real-world deadlocks require three or more priority change points to be triggered. Besides, the events of these deadlocks are hidden within 5,000 to >400,000 events except on Hawknl and SQLite. As a result, the guaranteed probabilities of all three approaches are small (i.e., at the level of 10^{-3} to 10^{-5}) except on Hawknl and SQLite. Therefore, it is challenging to triggering these deadlocks in practice.

However, we still observe that both $RPro_{op}$ and $RPro_{op}$ has larger guarantee probabilities except on SQLite (where only 16 events were produced which are less than the radius 50 of $RPro_{op}$). For example, on JDBC-1, the guaranteed probability of $RPro$ is at the level of 10^{-4}, which is about 10^{-2} larger than that of PCT (i.e., at the level of 10^{-6}). With increasing number of events, we could observe that, the guarantee probabilities of $RPro$ decrease much slower than that by PCT. For example, on four versions of MySQL, the guarantee probabilities of $RPro$ is about 10^{2} to 10^{19} larger than that by PCT.

From the last major column in Table 2, we observe that all three techniques triggered all deadlocks with larger probabilities than what they guaranteed, except for PCT on MySQL-3 and MySQL-4. This result is consistent with the previous results evaluating PCT [12][44] as a deadlock may occur in several ways. However, on the two deadlocks form MySQL-3 and MySQL-4, PCT failed to trigger any deadlock in 10,000 runs (i.e., producing a probability of zero, highlighted in Table 2); whereas $RPro_{op}$ and $RPro_{op}$ both triggered two deadlocks in 4 to 55 runs out of 10,000 runs.

With our strategy on selecting priority change points, $RPro$ achieved higher probabilities on triggering deadlocks. Table 2 shows that both $RPro_{op}$ and $RPro_{op}$ achieved an increases on the probabilities from 5.44% to more than 5,400% except on MySQL-1 on which, $RPro_{op}$ achieved the same probability as PCT. On average, the probability improvement of $RPro$ was more than 770% times. This improvement is significant.

In summary from Table 2, we observe that the strategy of $RPro$ is much more effective than that of PCT over all benchmarks.

### 4.3.2 Effectiveness of $RPro$ with Different Radiiuses

Figure 7 shows the probabilities of our $RPro$ on each benchmark with different radiiuses varying from 1 to 150 (or 300). In each sub-figure of Figure 7, the x-axis shows the 150 (or 300) radiiuses and the y-axis shows the corresponding probability on triggering each deadlock. On each subfigure, we show the point (i.e., peak value) where $RPro$ achieved the largest probability in the form of \( r = x, p = y \), indicating that the achieved largest probability was \( y \) when the radius was \( x \). For comparison purpose, we also show the probability of PCT in each sub-figure.

In Table 3, we further list the best radius ($r_{best}$) of $RPro$ that produced the largest probability on each benchmark, as well as the ratio of $r_{best}$ to the number of events (i.e., $r_{best}/#events$) except for Hawknl and SQLite that produced two few events. Table 3 also includes the number of events, the number of threads, the bug depth, and the largest probability for each benchmark. In Table 3, we sort all rows according to the column $r_{best}$.

![Figure 7](image_url)

Table 3. The best radiiuses ($r_{best}$) of each benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># events</th>
<th># threads</th>
<th>bug depth</th>
<th>$r_{best}$</th>
<th>$r_{best}/#events$</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hawknl</td>
<td>28</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>0.4530</td>
</tr>
<tr>
<td>SQLite</td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>0.6863</td>
</tr>
<tr>
<td>JDBC-2</td>
<td>5,050</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0.059%</td>
<td>0.0632</td>
</tr>
<tr>
<td>JDBC-4</td>
<td>5,090</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>0.098%</td>
<td>0.1123</td>
</tr>
<tr>
<td>JDBC-3</td>
<td>5,080</td>
<td>3</td>
<td>3</td>
<td>11</td>
<td>0.217%</td>
<td>0.0229</td>
</tr>
<tr>
<td>JDBC-1</td>
<td>5,088</td>
<td>3</td>
<td>3</td>
<td>17</td>
<td>0.334%</td>
<td>0.0439</td>
</tr>
<tr>
<td>MySQL-4</td>
<td>444,621</td>
<td>19</td>
<td>3</td>
<td>20</td>
<td>0.005%</td>
<td>0.0062</td>
</tr>
<tr>
<td>MySQL-2</td>
<td>15,066</td>
<td>17</td>
<td>3</td>
<td>27</td>
<td>0.179%</td>
<td>0.0256</td>
</tr>
<tr>
<td>MySQL-1</td>
<td>19,300</td>
<td>16</td>
<td>3</td>
<td>47</td>
<td>0.204%</td>
<td>0.0022</td>
</tr>
<tr>
<td>MySQL-3</td>
<td>406,117</td>
<td>22</td>
<td>6</td>
<td>114</td>
<td>0.028%</td>
<td>0.0039</td>
</tr>
</tbody>
</table>

(* All rows are sorted on the data in this column.)

From Figure 7 and Table 3, we have the following four observations:

**Observation 1.** When the radius varied from 1 to 150 (or 300), the probability of $RPro$ on each benchmark firstly increased. After the probability reached a certain value (i.e., the marked peak point of each sub-figure), it began to decrease. And the increase speed before the peak value was usually faster than the decrease speed from the peak value. For example, on JDBC-1, the probability was around 0.0024 when the radius is from 1 to 9; however, when the radius increased from 10 to 17, the probabilities jumped from about 0.0024 to around 0.0439. When the radius increased from 12 to 150, the probability gradually decreased from 0.0439 to around 0.0072. This is roughly consistent with our theoretical analysis on the guaranteed probability of $RPro$ (see Figure 6).
Note that, in practice, if PCT or RPro fails to select a set of right priority change points, it is still possible to trigger the deadlock due the randomness of program execution; however, the probability should be smaller than what they guarantee. We have also shown the trend of this practical probability changes in Figure 6, which is roughly consistent with the curve in Figure 7. Besides, when the radius increased to a larger value (e.g., more than 100), the probabilities by RPro on some benchmarks were still larger than that of PCT. There are two reasons. For some benchmarks (e.g., JDBC-1, JDBC-2, JDBC-3, MySQL-1, MySQL-2, MySQL-3, and MySQL-4), the radius is still much smaller than the number of events; on other benchmarks (e.g., Hawknl and SQLite), our optimization of RPro on deadlock triggering also played an important role.

Observation 2. Our benchmarks include programs of different sizes and they also produced different number of events from less than 100 to more than 400,000. From Figure 7 and Table 3, one could observe that, roughly, a large-scale program have a larger radius than a smaller one. However, one may also observe that a program containing more threads have a larger radius. This also indicates the complexity of deadlock triggering. For example, MySQL-3 has the largest number of threads, the largest bug
depth, and the second largest number of events. It also required the largest radius (i.e., 114) to trigger its deadlock. Besides, on this benchmark, the largest probability produced by RPro is the smallest one among all the largest probabilities on other benchmarks. Actually, PCT also produced the smallest (i.e., 0.0000) probability on this benchmark (as well as on MySQL-4). We will further investigate the correlation between radius and program parameters including the program size and the number of threads.

Observation 3. The radius of RPro to effectively trigger each deadlock is usually much smaller, compared with the number of events from the same program. For example, except on Hawknl and SQLite that produced too few events, the best radius ($r_{best}$) of each program is less than 0.35% of the corresponding number of events, as shown in the column "$r_{best}$/#events" in Table 3. Therefore, for deadlocks with larger bug depths, RPro is more effective than PCT. For example, on MySQL-3 where the bug depth is 6, PCT never triggered any occurrence of the deadlock out of 10,000 runs; whereas, RPro successfully triggered it in 4 to 18 runs as shown in Table 2. Besides, RPro triggered 39 occurrences of the deadlock when the radius is 114 (see Figure 7 (i)).

Observation 4. Figure 7 also shows that on all benchmark, the probabilities of RPro exhibited different vibrations with increasing radius values, although the overall trends are clear. This becomes particularly obvious on four version of MySQL benchmarks. We are interested in this phenomenon but we have not find the cause.

In summary, this experiment further validated the effectiveness of RPro that selects priority change points based on the radiiuses of deadlocks. It also shows that such radius is much smaller in practice than the total number of events of the same program.

5. RELATED WORK
In this section, we review related work on (1) deadlock detection, and (2) fixing and recovery of concurrency bugs.

5.1 Deadlock Detection
Detection of deadlocks is mainly based on detection of cycles in lock order graphs [8][10][11][18][28][40][45][56][61] or cyclic lock dependencies on lock dependency relation [15][16][32]. Both static and dynamic approaches could detect them [8][18][45][52] [56][61]. Static approaches may report false positives [61] compared as they cannot precisely infer the runtime information, even with various filters [45]. Although dynamic approaches are relatively precise, they also report false positives. Hence, many works target on detecting real ones through reachability analysis or active testing [17][19][31][34]. Other works, recently, focus on how to actually trigger occurrences of real-world deadlocks by searching for possible scheduling [13][16][17][32][53].

ESD [64] synthesizes an execution from a core dump file of an execution with a deadlock occurrence. PENELlope [57] also synthesizes part of execution to replay an observed atomicity violations or deadlocks. These techniques may fail due to the lack of thread interleaving and test cases.

ConTeGe [50] targets to generate concurrent test cases so as to trigger an expected concurrency bug. OMEN [54] further synthesizes executions for deadlock triggering based on ConTeGe. Sherlock [19] actively infers test cases based on interleaving constraints of threads involved in a targeted deadlock via concolic executions [55]. Deadlocks easily exist in database applications (e.g., MySQL Database Servers). These deadlocks could also be detected and prevented by analyzing hold-and-wait relations among threads and locks [26][27].

On the other hand, PCT and RPro focus on random testing. PCT has a probabilistic guarantee to find concurrency bugs including deadlocks of given bug depths. RPro further takes bug radius into consideration to improve the probabilities of PCT both theoretically and experimentally.

5.2 Concurrency Bug Fixing and Recovery
Manual bug fixing not only takes a long time [29] but is also error prone [62]. Recently, automated bug fixing become popular [14][23][24][25][35][49][60][66]. Most of these techniques on fixing concurrency bugs insert new locks (known as gate locks) statically or dynamically to serialize all executions of threads involved in a concurrency bug, including AFix [29][30], Axis [37], Grail [38], Gadara [59], and [46]. One of challenges on fixing concurrency bugs is whether new concurrency bugs could be introduced. For example, by introducing new locks to fix atomicity violations or deadlocks, new deadlocks may also be introduced [37][38][46]. Even manual fixing may also introduce deadlocks (e.g., 16.4% incorrect fixing indeed introduced new deadlocks [62]). Axis [37] further iteratively fixes introduced deadlocks by adding more new gate locks. Grail [38] adopts Petri-net analysis to eliminate such introduced deadlocks [59] which, however, is only applicable to deadlocks with two threads [38]. DFixer [14] is designed to fix deadlocks without introducing new deadlocks. Our RPro could be easily adapted to test for deadlocks in fixed program by selecting priority change points near to the events of the original deadlocks.

Recovery techniques could be integrated with deadlock detection and fixing. Sammati [51] aims to provide deadlock recovery by rolling back the executed operations, once a deadlock is detected. ConAir [65] tries to recover most concurrency bugs including deadlock. Lin et al. [36] propose to change lock acquisition primitives to the corresponding primitives with trials (e.g., from pthread_mutex_lock to pthread_mutex_trylock) to partially fix a deadlock. They further propose to recover program executions once a deadlock occurs. However, there are still challenges for recovery from deadlock occurrence as discussed in [36].

6. CONCLUSION
Existing randomized scheduling might be ineffective to trigger concurrency bugs. PCT randomly schedules a program based on priorities generated before executing a program and probabilistically guarantees to trigger concurrency bugs. However, PCT may also become ineffective for concurrency bugs with larger bug depths such as complex deadlocks. We proposed the bug radius concept and RPro approach to generate priorities based on bug radius. RPro has a larger probabilistic guarantee to trigger concurrency bugs with bug depth of three or more. The experiment on a set of real-world program also shows that RPro was much effective on 10 unique deadlocks than PCT. In future, we will apply RPro to trigger other kinds of concurrency bugs.

7. ACKNOWLEDGEMENT
We thank anonymous reviewers for their invaluable comments and suggestions on improving this work. This work is supported in part by National 973 program of China (2014CB340702), and National Natural Science Foundation of China (NSFC) (grant No. 61502465, 91418206, 61472318), and National Science Foundation (DGE-1522883, CCF-1500365).
8. REFERENCES