ASN: A Dynamic Barrier-based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs†

Yan Cai, Changjiang Jia, Shangru Wu, Ke Zhai, and W.K. Chan†

Abstract—Many large-scale multithreaded programs incur deadlock bugs. Existing deadlock warning detection techniques only report warning scenarios, which may or may not be real deadlocks. Each warning should be further verified on whether it may manifest into a real deadlock. For this purpose, a number of active randomized testing schedulers have been developed to trigger them, and yet pervious experiments show that their deadlock confirmation probability can be low. This paper presents ASN, a novel barrier-based randomized scheduler that triggers real deadlocks with high probabilities. We exploit the insights that in a confirmation run, the threads involved in a real deadlock should properly acquire one or more sets of locks prior to deadlocking. ASN automatically identifies three interesting sets of such positions. It guides the threads participating in a given warning to stay at these position sets in turn. When all the threads are staying at the last position set, ASN checks whether any deadlock that matches with the given warning has been triggered. We have evaluated ASN on 15 deadlock bugs in a suite of real-world multithreaded programs. The results show that ASN either confirms more deadlocks from the benchmark suite or triggers the same deadlocks with significantly higher probabilities than existing schedulers.

Index Terms—Debugging, deadlock triggering, randomized testing, large-scale multithreaded programs

1 INTRODUCTION

Many real-world multithreaded programs incur concurrency bugs [18], [32], [36]. These bugs (e.g., data races [11], [18], atomicity violations [29], [35], and deadlocks [6], [9], [13]) should be detected and rectified. Deadlocks are severe concurrency bugs. They prevent program executions from terminating correctly. Generic techniques can expose different kinds of concurrency bugs but only with a very low bug triggering ability [10]. On the other hand, techniques that precisely detect specific kinds of deadlocks are still unable to handle large-scale programs [25]. In this paper, we study the confirmation problem of resource deadlock, where locks are resources [13], [23].

A deadlock is triggered if all threads involved in a deadlock circularly wait for one another to release certain locks. Many static techniques [6], [16], [37], [38] and dynamic techniques [9], [15], [21], [23] infer such circular wait conditions in one way or another. They identify cycles in lock order graphs [21] or from sets of lock dependencies [13], [23]. But, such cycles may merely be deadlock warnings instead of real deadlocks [13], [38].

A promising approach to isolating deadlocks from the pool of such warnings is randomized active testing techniques [13], [23], which we refer to as deadlock confirmation techniques. Suppose that a dynamic deadlock warning detection technique has ran a program and generated a pool of deadlock warnings from it. A typical, existing deadlock confirmation technique [13], [38] re-runs the program attempting to reveal real deadlocks hidden in the pool. To ease our presentation, we refer to such a re-run of the program as a confirmation run.

Such a technique [13], [23] only suspends each thread involving in a deadlock warning whenever the thread is located at a deadlock site [14] without allowing the thread to acquire the lock associated with the site. However, we are going to show in Section 3 that this strategy may systematically miss to confirm a real deadlock. Moreover, our experiment to be presented in Section 5 shows that in some scenarios, this strategy works effectively, but in some other scenarios, it often causes the confirmation runs deadlock with the testing tool, even though the corresponding native runs can proceed naturally (where the problem is referred to as thrashing [23]). These phenomena are counter-intuitive if this class of strategy is adequate to confirm deadlocks.

We observe that all the threads involving in a deadlock should synchronize their execution steps not only at their deadlocking sites, where the deadlock occurs, but also at some other sites prior to these deadlocking sites. Suspending all threads at their deadlocking sites is only a necessary condition to trigger the deadlock. A good scheduler should create at least one sufficient condition of deadlock triggering, which, to our best knowledge, is never mentioned in related work (e.g., [13], [23]).
This paper proposes **ASN**, a barrier based dynamic deadlock triggering scheduler, where each barrier is a set of sites, one for each thread involved in a given cycle. ASN formulates with a sequence of three barriers in mind: **Admission Barrier** (the sites that these threads start using the locks associated with the deadlock), ** Sufficiency Barrier** (the sites related to the sufficient condition discussed above), and **Necessary Barrier** (deadlocking sites). In the course of a confirmation run, ASN schedules a program to traverse each barrier in cohort and one after another. The intuition behind this design is to reduce thrashing potentials across different segments of the same execution trace separated by consecutive barriers.

We also prove that there exists a set of sites such that, if a scheduler can suspend all the threads in the given cycle (which is a real deadlock) at this set of sites, it is sufficient to trigger an occurrence of the deadlock.

We have evaluated ASN on a suite of widely-used large-scale Java/C/C++ programs that contains 15 real-world deadlock bugs. The experimental results show that, compared to existing techniques (DeadlockFuzzer [23], MagicScheduler [13], and PCT [10]), ASN can either confirm more real deadlocks on the same benchmark or trigger the same deadlock bugs with significantly higher probability and incur significantly less thrashing. In the experiment, ASN incurs high performance overheads.

The contribution of this paper is threefold: (1) this paper presents ASN, a new class of active randomized testing scheduler to confirm real deadlocks from warnings. (2) The paper presents a theoretical guarantee of ASN, which shows that if a given warning is a real deadlock, ASN guarantees to manifest the warning into a real deadlock under certain conditions. (3) We show the feasibility of ASN by implementing it as two tool prototypes and reported a validation experiment on a suite of large-scale multithreaded programs. The experimental results show that ASN can be significantly more effective than peer techniques in confirming real deadlocks.

The rest of this paper is organized as follows: Section 2 gives the preliminaries. We motivate our work by an example in Section 3. Section 4 presents ASN followed by its evaluation in Section 5. Section 6 reviews the closely related work. Section 7 concludes the paper.

## 2 PRELIMINARIES

### 2.1 Lock Trace and Dependency

There are two types of event related to deadlock confirmation: *acquire*(t, m) and *release*(t, m), meaning that a thread t acquires a lock m and releases a lock m, respectively. Other events can be similarly handled [11], [18].

A site is an abstraction of an execution context [13], [14], [23], [26]. We denote by c@es the event e occurring at the site s, and denote the site s of the event e by *site*(e).

To ease readers to follow, each example in this paper only uses a program line number to illustrate a site. For instance, in Fig. 1(a), the lock acquisition event e₁ = acquire(t₁, k) occurred at site s₀₁ is referred to as e₁@s₀₁. Similarly, we refer to c@es as m@es if e = acquire(t, m).

A **lock dependency** [13], [23] τ = (t, m@es, L) is a triple consisting of a thread t, a lock m acquired at site s, and a *lockset* L. It represents that the thread t acquires a lock m at a site s while holding all locks in the lockset L.

For presentation clarity, we may hide the site associated with an event. Also, if a thread is involving in a cycle, we may refer to it without mentioning the cycle.

### 2.2 Cycle, Direct locks and Indirect Locks

A deadlock or a deadlock warning is a cycle [13] c = ⟨⟨t₁, m₁, L₁⟩, ⟨t₂, m₂, L₂⟩, ..., ⟨tᵥ, mᵥ, Lᵥ⟩⟩ such that m₁ ∈ L₂, m₂ ∈ L₃, ..., mᵥ ∈ L₁, and t₁ ≠ tᵥ. mᵢ ≠ mᵢ [mᵢ] ∩ Lᵢ = ∅ and Lᵢ ∩ Lᵢ = ∅, for 1 ≤ i, j ≤ k (i ≠ j).

For instance, the cycle c₀ = ⟨⟨t₁, n₁, (s, p, m), ⟨t₂, p, ⟨n⟩⟩⟩⟩ can be used to represent the deadlock shown in Fig. 1(a).

Each lock dependency τᵣ = ⟨⟨mᵣ, nᵣ@es, Lᵣ⟩⟩ in a cycle c can be reorganized into the following format: τᵣ’ = ⟨⟨mᵣ, nᵣ@es, nᵣ@es, Lᵣ⟩⟩ such that Lᵣ = Lᵣ \ [nᵣ@es] and τᵣ = ⟨⟨mᵣ, nᵣ@es, Lᵣ⟩⟩ is another lock dependency in c. We refer to this format as a **cyclic lock dependency**. As such, an alternative form of the above cycle c is ⟨⟨t₁, m₁, L₁⟩’, ⟨t₂, m₂, L₂⟩’, ..., ⟨tᵥ, mᵥ, Lᵥ⟩’⟩, where L’ = L \ [mᵢ@es], for 1 ≤ i ≤ k, and m₀ = mᵣ. In the running example, the cycle c₀ can be rewritten as c₁ = ⟨⟨t₁, n₁, (s, p, m), ⟨t₂, p, ⟨n⟩⟩⟩⟩.

Given a cyclic lock dependency ⟨⟨t, m, L⟩’⟩, both m and n are called the **direct locks**, and every lock in L’ is called an **indirect lock**. We denote the set of all direct locks in a cycle c by *directLocks*(c), and the set of all indirect locks by *indirectLocks*(c).

For instance, with respect to the cycle c₁ in Fig. 1(a), *directLocks*(c₁) is {n, p}, and *indirectLocks*(c₁) is {s, m}.

We also denote the set of all threads in the cycle c by *threads*(c). In the running example, *threads*(c₁) = {t₁, t₂}.

## 3 MOTIVATING EXAMPLE

Fig. 1(a) shows an example program with two threads (t₁ and t₂) and five locks (k, n, s, p, and m). There is a deadlock as highlighted on the lock acquisitions of p and n. The thread t₁ firstly acquires the lock k at site s₀₁ and then releases it at site s₀₂. Then, t₁ acquires the lock s at site s₀₃. Before releasing s, t₁ holds the lock n for a brief period (at sites s₀₄ and s₀₅) followed by acquiring three more locks p, m, and n at sites s₀₆, s₀₇, and s₀₈, respectively. Finally, t₁ releases all its locks from site s₀₉ to site s₁₂.

The thread t₂ acquires the lock s at site s₁₃ and then releases it at site s₁₄. Then, t₂ acquires the locks n and p at sites s₁₅ and s₁₆ and releases them at sites s₁₇ and s₁₈.

Fig. 1(b) and Fig. 1(c) show two possible threads schedules of the program: *Schedule 1* and *Schedule 2*.

**Schedule 1** triggers the deadlock: Suppose that thread t₂ is located at site s₁₅ before acquiring the lock n, and t₁ executes all the operations from site s₀₅ to site s₁₀. Thus, t₁ is holding the lockset {s, p, m}. Now, t₂ acquires the lock n at site s₁₅. However, t₂ cannot further acquire the lock p at site s₁₆ because t₁ is holding p. **Schedule 1** then switches to guide t₁ to acquire n at site s₀₉, but it fails as t₂ is holding the lock n. As such, a real deadlock occurs.

The deadlock in Fig. 1(a) is not easy to trigger. As shown in Fig. 1(c), in Schedule 2, right after t₂ has released the lock s at site s₁₄, t₁ acquires the lock k and then releases it. Before t₁ proceeds further, t₂ completes its execution. Following **Schedule 2** does not trigger the deadlock.
By analyzing Schedule 2, existing deadlock warning detection techniques [13], [21], [23] can report a deadlock warning on the four sites $s_{06}$, $s_{06}$, $s_{15}$, and $s_{16}$, where the warning is denoted by the cycle $c_0 = \langle t_1, n, \{ s, p, m \} \rangle, \langle t_2, p, \{ n \} \rangle$. To ease our discussion, we refer to these four sites as **deadlocking sites**.

Existing approaches schedule confirmation runs to trigger the reported deadlock warning $c_0$ as follows:

**Randomized Scheduler** (RS, see Algorithm 3 in [23]): Based on the given warning $c_0$, RS (e.g., DeadlockFuzzer [23], MagicScheduler [13]) targets to suspend the thread $t_1$ at site $s_{06}$ and suspend the thread $t_2$ at site $s_{16}$.

If the confirmation run is scheduled as Schedule 1, then RS successfully triggers the deadlock.

Otherwise, RS may miss to trigger the deadlock. For instance, with the attempt to follow Schedule 2, RS firstly suspends $t_2$ once $t_2$ is locating at site $s_{15}$ (i.e., after $t_2$ has acquired the lock $n$ at site $s_{15}$). However, when $t_1$ is locating at $s_{06}$, the thread $t_1$ has to wait for $t_2$ to release the lock $n$. Now, RS is suspending $t_2$ and $t_2$ is blocking $t_1$: thrashing [23] occurs. To resolve thrashing, RS has to resume $t_2$. After $t_2$ has acquired the lock $p$, there is no way to trigger the deadlock indicated by $c_0$ anymore.

**Systematic Scheduler** (including PCT [10]). These schedulers aim to detect concurrency bugs in general, and are unaware of specific bug information provided to them. Their ability to expose deadlocks is very low [10].

Given a multithreaded program, PCT firstly chooses a set of **priority change points** [10] (e.g., right before site $s_{15}$ for $t_2$ and right after site $s_{06}$ for $t_1$), and lowers the priority of a thread when the thread is about to execute any statement that is a change point [10]. If PCT happens to work like Schedule 1, it can trigger the deadlock. PCT has a probabilistic guarantee to trigger the deadlock. To expose the deadlock in the running example in Fig. 1(a), according to the formula in Section 2.4 in [10], the guaranteed probability is $1 \div (2 \times 18^{13}) \approx 0.0015$ (for 2 threads, 18 statements, and 3 change points), which is quite low.

To trigger the deadlock in Fig. 1(a), $t_2$ must have **precisely** acquired the lock $s$ at site $s_{13}$ before $t_1$ acquires the same lock $s$ at site $s_{06}$. We have illustrated via Fig. 1(c) that if a thread acquires one more lock (e.g., acquiring the lock $n$ at site $s_{15}$ by $t_2$) before another thread acquires a specific lock (e.g., acquiring the lock $n$ at site $s_{06}$ by $t_1$), then the latter thread may miss to locate at the deadlocking site $s_{06}$ before the former thread has passed through its deadlocking site $s_{16}$. Contrarily, if a thread acquires one less lock (e.g., not acquiring the lock $s$ at site $s_{13}$ by $t_2$), another thread may have already passed through its deadlocking site ($s_{06}$ for $t_1$) before the former thread acquires its next lock (e.g., the lock $s$ at site $s_{13}$ by $t_1$).

The above analysis indicates that setting random time delays [17] or user-chosen time delays [36] for individual threads, selecting a set of change points randomly [10], or suspending individual threads permanently by ignoring other threads [23], [13] are all inadequate to precisely control the execution sequence among the threads needed to trigger a real deadlock from a deadlock warning.

Consider Schedule 2 again. Suppose that a deadlock warning detection technique uses Schedule 2 to generate $c_0$. Deterministically replying [7] Schedule 2 up to the first deadlocking site (i.e., $s_{06}$) does not help to trigger the deadlock because $t_2$ is blocking $t_1$ to acquire the lock $n$ at site $s_{15}$. On the other hand, if a testing tool deterministically replays Schedule 2 up to site $s_{15}$, then the tool should decide which particular thread to be executed next.

In the next section, we present ASN and illustrate how ASN addresses the illustrated challenges.

### 4.1 Overview

For each thread involving in a given cycle $c$, **ASN** infers one site per barrier: **Admission Barrier** (ABr), ** Sufficiency Barrier** (SBr), and **Necessity Barrier** (NBr). In the course of a confirmation run, ASN schedules these threads to pass through the first two barriers in cohort and one by one, and checks for deadlock occurrences at the third barrier.

Fig. 1(c) depicts the three barriers for two threads $t_1$ and $t_2$ where the cycle is $c_0$. ASN firstly targets to suspend $t_1$ and $t_2$ right before the first barrier (ABr): the site $s_{06}$ for $t_1$ and the site $s_{15}$ for $t_2$, which is feasible according to Schedule 2. Then, ASN targets to suspend the two threads at the second barrier (SBr): the site $s_{06}$ for $t_1$ and the site $s_{15}$ for $t_2$. Suppose that $t_2$ has acquired the lock $s$ at site $s_{13}$ prior to $t_1$ acquiring the lock $s$ at site $s_{06}$. In this case, the two threads can locate at the second barrier. Finally, ASN targets to suspend the two threads at the third barrier (NBr): the site $s_{06}$ for $t_1$ and the site $s_{16}$ for $t_2$. Now, the cycle $c_0$ is confirmed by ASN as a real deadlock.

ASN uses an interesting approach to inferring the three barriers, which will be presented in Section 4.2. In Section 4.3, we present the algorithm of ASN. Finally, we present a theorem that shows the theoretical guarantee, optimization and variants of ASN in the rest of Section 4.
4.2 Barriers of ASN

In this section, we present the definitions of the three barriers and their design rationales.

When a deadlock occurs, each thread involving in the deadlock must wait to acquire a specific lock at a specific site, which we refer to as the deadlock triggering site. The Necessary Barrier for each thread involving in the cycle intends to specify that the thread should wait at its corresponding deadlock triggering site.

Definition 1 (Necessity Barrier): The necessity barrier $NBr(c)$ with respect to a given deadlock warning $c$ is the set \( \{ \text{site}(e) \mid e = \text{acquire}(t, m) \text{ and } \exists \langle t', n, m, L' \rangle, \langle t, m@s_2, l, L \rangle \in c \text{ such that } \text{site}(e) = s_3 \} \).

The $NBr$ site of each thread can be directly extracted from the given warning $c$. For instance, in Fig. 1(c), the site $s_{03}$ is the $NBr$ site for the thread $t_1$. It is the site where $t_1$ wants to acquire the lock $n$, but should be held by the thread $t_2$ so that these two threads sets up a circular wait condition necessary to trigger a deadlock.

However, suspending threads involving in a cycle at this barrier only represents a necessary condition after the confirmation run has manifested into a real deadlock at the corresponding sites. The goal of an active deadlock triggering scheduler should be to create certain sufficient conditions that manifest deadlocks and guide confirmation runs to satisfy such sufficient conditions so that the scheduler can effectively confirm real deadlocks.

The Sufficiency Barrier ($SBr$) models such a sufficient condition. The $SBr$ site of a thread $t$ involving in a cycle $c$ is where $t$ acquires a direct lock of another thread in $c$.

Definition 2 (Sufficiency Barrier): The sufficiency barrier $SBr(c)$ with respect to a given deadlock warning $c$ is the set \( \{ \text{site}(e) \mid e = \text{acquire}(t, m) \text{ and } \exists \langle t', n, m, L' \rangle, \langle t, m@s_2, l, L \rangle \text{ such that } \text{site}(e) = s_3 \} \).

As shown in Fig. 1(c), the $SBr$ sites for the two threads $t_1$ and $t_2$ are sites $s_{06}$ and $s_{15}$, respectively. If $t_1$ and $t_2$ are concurrently suspended at sites $s_{06}$ and $s_{15}$ followed by resuming their executions, the deadlock indicated by the cycle $c_0$ will be triggered at the necessity barrier $NBr(c_0)$.

Before a thread reaches its $SBr$ site, thrashing may have occurred. It blocks this thread from acquiring an indirect lock being held by a suspending thread. For instance, in Fig. 1(c), if $t_1$ has suspended at its $SBr$ site ($s_{06}$) before $t_2$ acquires the lock $s$ at site $s_{13}$, then $t_1$ cannot reach its $SBr$ site ($s_{13}$) until $t_1$ releases the lock $s$ at site $s_{13}$.

ASN aims to divide the traces of the threads involved in the given cycle $c$ into segments separated by barriers. As such, thrashing will be contained within each segment instead of across multiple segments, thereby reducing the potential of thrashing occurrences.

Definition 3 (Admission barrier): The admission barrier $ABr(c)$ with respect to a given deadlock warning $c$ is the set \( \{ \text{site}(e) \mid \exists t \in \text{Threads}(c) \text{ and } e = \text{acquire}(t, m) \text{ such that } (1) \ m \in \text{directLocks}(c) \cup \text{indirectLocks}(c), \text{ and } (2) \forall e' = \text{acquire} \ (t, n) \text{ such that } (i) \ e' \neq e, \text{ (ii) } n \in \text{directLocks}(c) \cup \text{indirectLocks}(c) \text{ and (iii) } e \mapsto e', \text{ where } \mapsto \text{ is the happened-before relation}) \} )

Intuitively, the admission barrier $ABr$ for a thread represents a site where the thread acquires its very first direct lock or its very first indirect lock along the run. As such, there are 1 segment before the admission barrier and 1 segment between any two consecutive barriers.

In Fig. 1(c), as depicted, the admission barrier for the thread $t_1$ is site $s_{03}$ and for the thread $t_2$ is site $s_{13}$. In the course of execution, if the two threads are suspended at these two sites, the probability for two threads reaching their sufficiency barriers will be at least 50%. Otherwise, it depends on whether the thread $t_2$ acquire the lock $s$ at site $s_{13}$ before the thread $t_1$ acquires the lock $s$ at site $s_{03}$.

4.3 Algorithm

In this section, we present the ASN algorithm. The algorithm firstly monitors the confirmation run against the admission barrier $ABr(c)$ followed by the sufficiency barrier $SBr(c)$ and finally the necessity barrier $NBr(c)$. To ease our presentation, we refer to the site corresponding to a thread in three barriers $ABr(c)$, $SBr(c)$, and $NBr(c)$ as $ABr(c, t)$, $SBr(c, t)$, and $NBr(c, t)$, respectively.

Algorithm 1 summarizes the main ASN algorithm. It takes a program $p$ and a deadlock warning $c$ as inputs. At lines 1–3, it initializes the execution state of the confirmation run. For each thread $t$ in the warning $c$, it assigns $ABr(c, t)$ to the variable $CurBr$, and initializes two maps $\text{Request}$ and $\text{Lockset}$ as empty sets. The set $\text{Enable}$ (lines 4 and 22) models the set of active threads in the confirmation run. If $\text{Enable}$ is non-empty (line 5), the algorithm fetches the next statement (denoted by $\text{stmt}$). It handles $\text{stmt}$ by distinguishing three cases:

Case 1: If $\text{stmt}$ is never a lock acquisition/release event nor a statement executed by any thread involving in $c$, Algorithm 1 simply executes $\text{stmt}$ (lines 7–8). For instance, all memory accesses fall into this case.

Case 2: If $\text{stmt}$ is an $\text{acquire}(t, m)$ event, where $t$ is a thread involving in $c$, the algorithm updates its execution state by associating $t$ with $m$, and keeps the association relation in $\text{Request}$ (lines 9–10). It then checks whether $\text{ stmt }$ is at the barrier under monitoring for the thread $t$ via the function $\text{checkBarrier}$ (line 11). If this is the case, Algorithm 1 pushes $\text{stmt}$ back to the statement execution queue, and suspends $t$ by removing it from the set $\text{Enable}$ (lines 12–13). For instance, in the running example, if $\text{stmt}$ is $\text{acquire}(t, s)$ occurring at site $s_{03}$ which is the $ABr$ site of $t_1$, ASN sets $\text{Request}(t_1)$ to $s_{03}$ and invokes $\text{checkBarrier}(\text{acquire}(t_1, s@s_{03}))$, which returns true. Thus, ASN removes $t_1$ from $\text{Enable}$ because the function $\text{checkBarrier}$ has suspended $t_1$ without executing $\text{acquire}(t_1, s)$. Otherwise, Algorithm 1 executes $\text{stmt}$ and updates the execution state accordingly (lines 15–16). For instance, if $\text{stmt}$ is $\text{acquire}(t_1, k)$ at site $s_{09}$, the statement is directly executed and $\text{Lockset}(t_1)$ is updated to include the lock $k$.

Case 3: If $\text{stmt}$ is a $\text{release}(t, m)$ event, the algorithm removes the lock $m$ from the set $\text{Lockset}$ for the thread $t$, and executes $\text{stmt}$ (lines 18–20). For instance, if $\text{stmt}$ is $\text{release}(t, k)$ occurring at site $s_{09}$, ASN executes it and removes the lock $k$ from $\text{Lockset}(t_1)$.

Next, if the set $\text{Enable}$ becomes empty (line 22), the
**Algorithm 1: ASN Scheduler (Program p, Cycle c)**

```plaintext
1. for each thread t in Threads(c) do
   2. CurBr(t) := ABrc(t, t); Request(t) := false, Lockset(t) := Ø;
   3. end for

4. Enable := Threads(p) // all threads in the program p
5. while Enable ≠ Ø do
   6. t, stmt := the next statement stmt from a random thread t
   7. if t ≠ Threads(c) ∨ (stmt ≠ acquire ∧ stmt ≠ release) then
     8. execute(stmt)
   9. else if stmt = acquire(t, m) then
     10. Request(t) := m
   11. if checkBarrier(stmt) = true then
     12. push back stmt
     13. Enable := Enable ∪ {t}
   14. else // execute the statement and update the execution state
     15. execute(stmt)
     16. Lockset(t) := Lockset(t) ∪ {m}
   17. end if
   18. else if stmt = release(t, m) then
     19. Lockset(t) := Lockset(t) \ {m}
   20. execute(stmt)
   21. end if
   22. End if
   23. if Enable = Ø then
   24. if some threads are suspended // thrashing is detected
   25. end if
   26. resume a suspended thread randomly
   27. else // Print "A real deadlock is triggered!"
   28. end if
   29. End while

30. function checkBarrier(Event c) // where c = acquire(t, m) or
   31. bar := Br(t)
   32. if c = bar then // the thread t is at its forthcoming barrier
   33. suspend(t)
   34. if each thread x in Threads(c) at site CurBr(x) then
   35. if the monitoring barrier is the necessity barrier then
   36. call checkforDeadlock(x)
   37. end if
   38. for each t' in Threads(c) do
   39. Br(t') := NExt(CurBr(t')) // advance to the next barrier
   40. if c ≠ CurBr(t) then
   41. resume(t) // may still be the barrier under monitoring
   42. Enable := Enable ∪ {t'}
   43. end if
   44. end if
   45. return false
   46. end if
   47. return true
   48. end if
   49. return false
50. end function

51. function checkforDeadlock(Cycle c)
52. if ∃ c = (d₁, d₂, ..., dₙ) where dᵢ = (tᵢ, Request(tᵢ), Lockset(tᵢ)), such that c is a cycle then
53. if c = c then
54. Print "The given warning is confirmed as a real deadlock!" halt!
55. else
56. Print "A real deadlock is triggered!" halt!
57. end if
58. end if
59. end function
```

The function `checkforDeadlock` (lines 51-59) checks real deadlock occurrence and, if any, reports the deadlock, which may be different from the given warning `c` (lines 53-57), and halts the execution.

Compared to existing work, ASN only checks for deadlock occurrences once instead of checking right before each lock acquisition event. It consumes less time on deadlock checking. At the same time, it consumes more time to handle two more barriers.

### 4.4 Theoretical Guarantee of ASN

We firstly recall that a cycle `c` is defined as a sequence of cyclic lock dependencies `c = ⟨(t₁, l₁, lᵢ), ..., (tᵢ, lᵢ, lᵢ₊₁), ..., (tₙ, lₙ, l₁)⟩`. Theorem 1. If a cycle `c` is a real deadlock of a multi-threaded program, ASN guarantees to trigger this deadlock if the following three conditions are satisfied:

(a) Each thread `tᵢ` in `Threads(c)` is at the `SBr(c, t)` site, and is going to acquire the lock `lᵢ`.

(b) There is no deadlock or livelock ever occurred before the deadlock `c` is triggered in the execution.

(c) For each thread `tᵢ`, `SBr(c, t)` dominates `NBr(c, t)` by the program order.

**Proof.** We prove Theorem 1 by mathematical induction. The basic idea is: we firstly prove the base case (i.e., `|Threads(c)| = 2`) and then suppose that, for `|Threads(c)| = n`, the theorem is true. Next, for a program `p` with a cycle `c₊₁` such that `|Threads(c₊₁)| = n₊₁`, we create a new program `p'` containing a deadlock `c₀` with `n` threads by mapping the execution of `(n₊₁)` threads to `n` threads, where the theorem is true as supposed. Finally, we show
that the program \( p \) can be schedules by ASN in the same way as ASN schedules the program \( p' \).

**Base case:** |\( \text{Threads}(c) \)| = 2 as depicted as Fig. 2(a):

Subcase (1): Suppose both \( t_1 \) and \( t_2 \) cannot locate at \( NBr(c, t_1) \) and \( NBr(c, t_2) \), respectively. Then, a deadlock or a livelock must have occurred (as no thread is suspended by ASN), which contradicts to the condition (b).

Subcase (2): Suppose that in a run, only one thread, say the thread \( t_i \), is unable to locate at \( NBr(c, t_i) \) but the thread \( t_l \) is locating at \( NBr(c, t_l) \), as shown in Fig. 2(b). In this case, \( t_i \) must wait for a different thread \( t_l \) to release a lock \( l_i \). A scenario is depicted as Fig. 2(c). If the thread \( t_i = t_l \) then we have \( l_i \neq l_1 \) as the lock \( l_1 \) has not been acquired by \( t_1 \) at \( NBr(c, t_1) \). Then, a real deadlock \( c = \{(t_1, l_1, l_1', l_1''), (t_2, t_2, l_2, l_2'')\} \) must have occurred, which also contradicts to the condition (b). Next, let us consider the case where \( t_i \neq t_l \). Because only the thread \( t_l \) is suspended by ASN, the thread \( t_i \) must be waiting for some other thread to release a lock. Since the total number of threads in the program \( p \) is limited, the run must have encountered a deadlock (which is different from the cycle c) or a livelock. It contradicts to the condition (b).

Based on subcases (1) and (2), threads \( t_1 \) and \( t_2 \) should be able to locate at \( NBr(c, t_i) \) and \( NBr(c, t_l) \), respectively, and trigger the deadlock. c We prove the base case.

**Induction step:** Suppose that the theorem is true when |\( \text{Threads}(c) \)| = \( n \).

Now consider the case where |\( \text{Threads}(c) \)| = \( n + 1 \). As depicted in Fig. 2(d), by the condition (a), each thread \( t_i \) of these \( (n + 1) \) threads is able to locate at \( SBr(c, t_i) \). Because the cycle \( c \) is a real deadlock and there is no deadlock occurs prior to the occurrence of \( c \), by condition (c), the executions of two threads \( t_i \) and \( t_{i+1} \) (or any two threads in \( c \) that form a wait-for relation) from \( SBr(c, t_i) \) and \( SBr(c, t_{i+1}) \) must reach \( NBr(c, t_i) \) and \( NBr(c, t_{i+1}) \), respectively, and encounter no any other deadlock. Thus, we can merge the executions of these two threads \( t_i \) and \( t_{i+1} \) into single execution and denote the combined thread as \( t_{i}' \) (depicted in Fig. 2(e)). The merging rule is:

1. The events from two threads happened before the events at \( SBr(c, t_i) \) and \( SBr(c, t_{i+1}) \) are merged into the new execution (denoted by \( \alpha \)) by following their execution order in the original execution trace.
2. The events from \( SBr(c, t_i) \) to \( NBr(c, t_i) \) of \( t_l \) and from \( SBr(c, t_{i+1}) \) to \( NBr(c, t_{i+1}) \) of \( t_{i+1} \) (including the events at \( SBr(c, t_i) \), \( SBr(c, t_{i+1}) \), and \( NBr(c, t_{i+1}) \), but excluding the events at \( NBr(c, t_i) \)) will be appended to the execution \( \alpha \) of the thread \( t_{i}' \).

(Note we need not to consider the events after \( NBr(c, t_i) \) and \( NBr(c, t_{i+1}) \), which can be merged in any feasible order.) We denote the new program that consists of the threads \( t_i \), \..., \( t_{i+1} \), \( t_i' \) as the program \( p' \). Now, on the program \( p' \), we have a deadlock \( c' = \{(t_1, l_1, l_1', l_1''), \ldots, (t_{i+1}, l_{i+1}, l_{i+1}', l_{i+1}''), (t_1, l_1, l_1', l_1'') \} \). Because |\( \text{Threads}(c') \)| = \( n \), ASN can schedule the execution of the program \( p' \) to trigger the deadlock \( c' \) as supposed. Next, we map back the execution of the thread \( t_{i}' \) to the executions of \( t_i \) and \( t_{i+1} \) according to the order that they have been merged into the execution \( \alpha \) of the thread \( t_i' \). Note that when the deadlock \( c' \) occurs, ASN has suspended the thread \( t_{i}' \) at \( NBr(c', t_{i}') \) in \( p' \), which is the same site as \( NBr(c, t_{i+1}) \) in \( p \). Now, we consider the execution of the program \( p \). After mapping back the events from \( t_i \), each thread \( t_i \) in \( \text{Threads}(c) \) except the thread \( t_0 \) should have been suspended at the corresponding \( NBr(c, t_i) \) and should be about to acquire the lock \( l_i \). For the thread \( t_0 \), all its statements right before \( NBr(c, t_0) \) have been executed. But, the thread \( t_0 \) cannot further be scheduled to acquire the lock \( l_{i+1} \) at \( NBr(c, t_0) \) because \( l_{i+1} \) is being held by the thread \( t_{i+1} \). As a result, the deadlock \( c \) is triggered.

By mathematical induction, Theorem 1 is proved. \( \blacksquare \)

### 4.5 Optimization

Algorithm 1 suspends each thread at a barrier right before acquiring the corresponding lock so that other threads may utilize the lock (if necessary) before locating at the same barrier. However, if other threads do not utilize this lock before locating at their deadlocklocking sites, there is no need to continue to suspend the former thread at the barrier in question. In this way, the total number of thread suspensions in a run can be reduced, which further reduces the scheduling overhead of ASN.

For instance, suppose that, in Fig. 1(c), the thread \( t_1 \) is locating at site \( s_{00} \) and is about to acquire the lock \( p \) and the thread \( t_2 \) is locating in between the sites \( s_{41} \) and \( s_{45} \). According to Algorithm 1, ASN suspends \( t_1 \) at site \( s_{30} \) until \( t_2 \) locates at site \( s_{50} \). However, \( t_2 \) does not acquire the lock \( p \) before reaching its \( NBr \) site \( (s_{50}) \). In this case, even if ASN does not suspend thread \( t_1 \) at its \( SBr \) site \( (s_{00}) \), the thread \( t_2 \) is still able to locate at its \( SBr \) and \( NBr \) sites, and the deadlock can still be successfully triggered. On the other hand, suppose that the thread \( t_1 \) is locating at site \( s_{30} \) (its \( ABr \) site) before the thread \( t_2 \) locates at site \( s_{31} \). ASN must suspend the thread \( t_1 \). Otherwise, thrashing would occur if thread \( t_2 \) is locating at site \( s_{51} \) to acquire the lock \( s \). This is because there is a lock acquisition (on the lock \( s \) associated with \( ABr(s_0, t_1) \)) by the thread \( t_2 \) prior to the same thread \( t_1 \) locating at its \( NBr \) site.

General speaking, if a lock \( m \) is associated with a barrier site \( XBr(c, t) \) (where \( XBr \) is either \( ABr \) or \( SBr \) and all other threads involving in the same cycle \( c \) do not acquire the lock \( m \) before locating at their necessity barrier sites (which is determined based on the predictive run), ASN does not suspend this thread \( t \) at this site \( XBr(c, t) \) but just marks that \( t \) has located at this site \( XBr(c, t) \).
4.6 Variants of ASN

In the algorithmic design, ASN is built on three barriers ($\langle AB(r), SB(r), NB(r)\rangle$. Among these three barriers, the necessity barrier $NB(r)$ is precisely the set of sites where the given deadlock occurs, hence, cannot be removed.

To help us to validate ASN, we make two variants of ASN, and each variant uses one less barrier than ASN:

- **AN** is a technique that uses the admission barrier followed by the necessity barrier, that is $\langle AB(r), NB(r)\rangle$.
- **SN** is a technique that uses the sufficiency barrier followed by the necessity barrier, that is $\langle SB(r), NB(r)\rangle$.

Both AN and SN can be straightforwardly implemented by modifying Algorithm 1.

5 EXPERIMENT

5.1 Benchmarks and Implementation

We selected a suite of real-world, large-scale Java and C/C++ programs, including JDBC connector [1], SQLite [4], HawkNL [5], and MySQL [1]. They contained in total 15 real deadlocks. All these benchmarks were available online [1], [26], and had been used in deadlock related experiments (e.g., [13], [26]). We have implemented ASN for Java and C/C++ using ASM 3.2 [2] and Pin 2.10 (probe mode) [31] with Pthreads, respectively.

We compared ASN to PCT [10], MagicScheduler (MS) [13], and DeadlockFuzzer (DF) [23], AN, and SN on the same framework. Although DF for Java is available from the current release of Calfuzzer [24], yet Calfuzzer only instrumented the test harness but did not instrument the JDBC Connector library that contains the deadlocks. Besides, the released DF is different the algorithm reported [23]. Finally, we faithfully implemented DF based on both [23] and Calfuzzer [24] (including the optimization [23]). The original tool of PCT was not publicly available. We faithfully implemented PCT according to [10].

5.2 Experimental Setup

We performed our experiment on a 3.16GHz Duo processor with Ubuntu 10.04. We used the object abstraction algorithm in [14] to identify sites for each event and used Magilock [12], [13] to generate all cycles. Because no technique in the experiment was able to confirm false positives, therefore, we only applied each cycle that is a real deadlock to each technique for 100 runs [13], [23].

TABLE 1 shows the descriptive statistics of benchmarks, including the benchmark name, the cycles (numbered from c1 to c15) in each benchmark, the bug ID if available, and the size of each benchmark (SLOC) [3]. The fifth column shows a brief deadlock description for each benchmark. The last two columns show the number of threads/locks in each benchmark and the numbers of (direct and indirect) locks in each cycle, respectively.

5.3 Effectiveness

Fig. 3 summarizes the probability of each technique on each cycle listed in TABLE 1. We note that PCT does not use any information on the provided cycles to trigger real deadlocks. Hence, it is not totally fair to compare PCT to the other three techniques. Interpreting the data in the rest of Section 5 must consider this difference.

From Fig. 3, we observe that ASN can confirm each

![Fig. 3. Comparisons on the triggering probability of real deadlocks among PCT, MS, DF, AN, SN, and ASN. In each subfigure, y-axis means the probability.](image-url)
cycle in a higher probability (≥75%), which is consistent across all cycles, than other techniques.

The confirmation probabilities of the other techniques are not consistent. For instance, they may not be able to confirm some cycles in any confirmation run (e.g., c9, c11, c13, and c14 for MS and DF) or may only confirm a cycle with a significantly lower probability than ASN (e.g., on c1-c8, and c12 for MS and DF) by 44–63%.

On the largest benchmark MySQL used in the evaluation, ASN triggers all cycles as real deadlocks that were missed by both DF and MS in all 100 confirmation runs.

TABLE 2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cycle ID</th>
<th>PCT</th>
<th>MS</th>
<th>DF</th>
<th>ASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>c1</td>
<td>49</td>
<td>58</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Connector 5.0</td>
<td>c2</td>
<td>57</td>
<td>55</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c3</td>
<td>48</td>
<td>52</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SQLite 3.3.3</td>
<td>c4</td>
<td>45</td>
<td>48</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>HawkNL 1.6b3</td>
<td>c5</td>
<td>52</td>
<td>47</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MySQL Server 6.0.4</td>
<td>c6</td>
<td>57</td>
<td>57</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c7</td>
<td>44</td>
<td>45</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c8</td>
<td>49</td>
<td>51</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 3

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cycle ID</th>
<th>Native</th>
<th>PCT</th>
<th>MS</th>
<th>DF</th>
<th>ASN</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDBC</td>
<td>c1</td>
<td>2.18</td>
<td>1.82</td>
<td>1.85</td>
<td>1.89</td>
<td>1.95</td>
</tr>
<tr>
<td>Connector 5.0</td>
<td>c2</td>
<td>2.03</td>
<td>1.80</td>
<td>1.87</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c3</td>
<td>1.68</td>
<td>1.90</td>
<td>2.09</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c4</td>
<td>1.49</td>
<td>1.90</td>
<td>2.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>c5</td>
<td>0.97</td>
<td>1.35</td>
<td>1.55</td>
<td>1.86</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c6</td>
<td>0.92</td>
<td>1.43</td>
<td>1.70</td>
<td>1.98</td>
<td>1.75</td>
</tr>
<tr>
<td></td>
<td>c7</td>
<td>1.44</td>
<td>1.57</td>
<td>1.61</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4 Comparison among ASN Variants

Fig. 3 also shows the confirmation probabilities of SN and AN on each cycle.

AN is basically incapable of confirming cycles c1-c6, c9, c13-c14 as real deadlocks. On the remaining six cycles, AN, MS and DF perform similarly. Except on c10 and c15, ASN is significantly more effective than AN.

SN can confirm all 15 cycles. Compared to MS and DF, SN was more effective on cycles c6, c7, c9, and c11-c14. Both ASN and SN achieved similar effectiveness on cycles c6, c7, c9, c10, and c15, but ASN is significantly more effective than SN on the remaining 10 cycles.

The result from AN and SN shows that using one less barrier in ASN (i.e., simply enhancing MS and DF with the admission barrier or sufficiency barrier only), the probability of confirming deadlocks are likely reduced.

5.5 Performance

TABLE 3 shows the time cost that each technique spent, providing that it can successfully trigger the deadlock indicated by each cycle in a run. The MySQL benchmark is a server program and the time cost is not compute-bound. On this benchmark, if a technique cannot make any successful confirmations, we denote the corresponding cell by a dash (“-”). Note that the time included all from the very beginning to the time when the deadlock occurs, and include the instrumentation time.

From TABLE 3, on the Java benchmarks, we observe that PCT, MS, and DF incur similar time overheads. On C/C++ programs, ASN incurs more time. We find that this is mainly attributed to our tool implementation. In the Java-based tool, we directly implement the three barriers using Java library utilities. However, Pin [31] disables the C++ multithreading utilities. So we use the sleep() function in Pin to implement our barriers, which caused more delays than using the wait() calls, because in the latter case, a thread can be woken up by a notify() call at any time. Still, using the current prototype, the time cost is in a matter of a few seconds. We believe that the time cost of ASN in the experiment is practical.

5.6 Scalability

Following [10], we configured all benchmarks (c1-c10) except MySQL to be run with 2 to 64 threads. In each configuration, we repeat the experiment described in Section 5.2. For MySQL, the number of threads in a run is self-governed by MySQL which cannot be changed by us.

Fig. 4 shows the deadlock confirmation probability on cycles c1-c10. With increasing number of threads, the confirmation probability of ASN keeps at 100% on six cycles (c1, c6, c7, c9, and c10). On c2 and c8, the probability is close to 100%. On the remaining cycles (c3-c5), the probabilities are all above 80%. Fig. 4 shows that ASN is able to scale up to confirm deadlocks in programs with many threads.

5.7 Case Study

The program MySQL Server is the largest among our benchmarks. In this section, we analyze the deadlock in

![Fig. 4. Scalability of ASN with increasing number of threads on cycles c1 - c10.](image-url)
high probability. It is because the two threads are likely to reach their sufficiency barrier sites (E7 and E1, respectively), and the algorithms should then make each thread to acquire the corresponding direct lock. This leads the two thread to be later suspended at their necessity barrier sites at E8 and E2, triggering the deadlock. Note that for ASN, the admission barrier and the sufficiency barrier for the thread t2 are the same (E1).

For PCT, its guaranteed probability on MySQL Server is roughly 1/ (17 * 15000^2) ≈ 7.8 × 10^{-6} (for 17 threads, more than 15,000 lock acquisition and release events, and 2 changing points), which is close to 0%.

TABLE 4 summarizes our above qualitative analysis on confirming c11 by MS, DF, AN, SN, and ASN, and the probabilities observed from the experiment (taken from Fig. 3). From TABLE 4, we observe the effectiveness of these techniques in the experiment is in line with the above qualitative analysis.

On c15, there is an interesting thread (Gopher thread) from a group of threads where each thread acquires a lock at beginning and does not release it until the thread dies. Suppose that this Gopher thread is suspended on its acquisition of such a lock (which is the site for the admission barrier of the Gopher thread), MySQL selects another thread from the same group to complete the intended task. Hence, after its resumption, the Gopher thread has nothing to do. However, as such a lock is only acquired by the Gopher thread itself, ASN does not suspend the Gopher thread on its acquisition of above lock at its ABr site. Thus, ASN can confirm c15 with almost a probability of 100%. Note that other techniques except PCT can also confirm this deadlock.

6 Related Work

Predictive deadlock detection. Static techniques analyze the code list to infer potential deadlocks [6], [37], [38]. Naik et al. [33] propose a combination approach to reducing the false positive rate. And yet, real deadlocks could not be isolated. Deshmukh et al. [16] design symbol execution technique to alleviate this problem. Dynamic techniques [13], [15], [23] analyze execution traces to infer potential deadlocks. Joshi et al. [25] propose a model checking approach, which requires manual annotations, to detect generalized deadlocks. JPF [8] is a possible approach to detect general concurrency bugs, which however suffers from severe scalability problems.

The scheduling approach of BTrigger [36] is similar to that of DeadlockFuzzer, except that BTrigger postpones a thread at each concurrent breakpoint "for a while" to eliminate thrashing. Besides, BTrigger requires developers to manually insert concurrent breakpoints.

Dimmunix [26] aims to prevent the second occurrence of any previously occurred deadlocks. It records the patterns of occurred deadlocks and postpones lock acquisi-
tion at runtime if the locking scenario matches the recorded deadlock patterns. Gadara [39] inserts gate lock acquisition code at each deadlock site detected statically and, at runtime, serializes executions whenever a statically detected deadlock is likely to occur. Grechanik et al. [20] also use static analysis and runtime monitoring approach to prevent deadlock in database applications. Nir-Buchbinder et al. [34] use an execution serialization strategy for deadlock healing. Compared to ASN, these techniques develop and utilize no admission or sufficiency barriers. Sammati [27], [28] is a deadlock recovery technique that selects a victim thread and rolls back the execution to resolve deadlock occurrence.

ESD [40] synthesizes the execution that goes into a deadlock state by analyzing the core dump of a previous failed execution. ASN can take a potential deadlock or a real deadlock (specified as a cycle) as an input. Both ConTest [17] and CTrigger [35] inject random noises to the execution being manipulated with the aim of improving its probability of triggering concurrency bugs. ASN can be viewed as an approach to injecting systematic noises to a program execution via each barrier. Huang et al. [22] propose to avoid deadlocks through automatic generation of synchronization logics in design programs.

Replay techniques [7], [19] for concurrency bugs can help developers to locate and understand how these concurrency bugs can happen. Compared to them, ASN does not rely on any execution with real deadlocks.

7 Conclusion

Many real-world large-scale multithreaded programs incur deadlock bugs. This paper has proposed ASN, a novel multi-bariers deadlock triggering scheduler. ASN is currently designed with a sequence of three barriers. We have proven that the second barrier is a sufficient condition to trigger real deadlocks at its last barrier under certain conditions. We have evaluated that ASN can be promising to confirm deadlock bugs in real-world multithreaded programs. Future work includes the deadlock removal confirmation after program changes.

REFERENCES

Yan Cai is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BEng degree in Computer Science and Technology from Shandong University, China in 2009. His current research interest is concurrency bug detection and reproduction in large-scale multithreaded and concurrent systems. His research results have been reported in venues such as ICSE, ISSRE, ICWS, SPE, JWSR, and TPDS.

Changjiang Jia is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BSc and MSc degrees from National University of Defense Technology, China. His research interests are program analysis, test sampling, and failure diagnosis.

Shangru Wu is a PhD student at Department of Computer Science, City University of Hong Kong. He received his BEng degree in Software Engineering from International School of Software, Wuhan University, China, in 2011. His current research interest is concurrency bugs fixing for multithreaded programs.

Ke Zhai is currently a PhD student at The University of Hong Kong and a technology analyst at Goldman Sachs. He received his bachelor and master degrees from the Software College of Northeastern University, Shenyang, China, in 2007 and 2009, respectively. His research interests include software testing and debugging, concurrent system analysis and verification, and engineering issues in services computing.

W.K. Chan is an assistant professor at Department of Computer Science, City University of Hong Kong. He is an editorial board member of Journal of Systems and Software, and was guest co-editors of a few international software engineering journals, program co-chairs of AST 2010 and QSC 2010, and innovative showcase chairs of ICWS and SCC for both 2009 and 2010. He is a program or review committee member of ICSE’15, ICSE’13 DS, and FSE’14. His research results have been reported in many venues including TOSEM, TSE, TPDS, TSC, CACM, COMPUTER, ICSE, FSE, ISTA, ASE, ICDCS, and WWW. His current research interest includes program analysis and testing of large-scale software systems.