Deadlock Prediction via Generalized Dependency

Jinpeng Zhou
University of Pittsburgh
USA
jinpeng@cs.pitt.edu

Hanmei Yang
University of Massachusetts Amherst
USA
hanmeiyang@umass.edu

John Lange
Oak Ridge National Lab
University of Pittsburgh
USA
jacklange@cs.pitt.edu

Tongping Liu
University of Massachusetts Amherst
USA
tongping@umass.edu

ABSTRACT

Deadlocks are notorious bugs in multithreaded programs, causing serious reliability issues. However, they are difficult to be fully expunged before deployment, as their appearances typically depend on specific inputs and thread schedules, which require the assistance of dynamic tools. However, existing deadlock detection tools mainly focus on locks, but cannot detect deadlocks related to condition variables. This paper presents a novel approach to fill this gap. It extends the classic lock dependency to generalized dependency by abstracting the signal for the condition variable as a special resource so that communication deadlocks can be modeled as hold-and-wait cycles as well. It further designs multiple practical mechanisms to record and analyze generalized dependencies. In the end, this paper presents the implementation of the tool, called `UnHang`. Experimental results on real applications show that `UnHang` is able to find all known deadlocks and uncover two new deadlocks. Overall, `UnHang` only imposes around 3% performance overhead and 8% memory overhead, making it a practical tool for the deployment environment.

CCS CONCEPTS

• **Software and its engineering → Deadlocks**.

KEYWORDS

Deadlock Prediction, Condition Variables, Multithreaded Programs

ACM Reference Format:
Jinpeng Zhou, Hanmei Yang, John Lange, and Tongping Liu. 2022. Deadlock Prediction via Generalized Dependency. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534377

1 INTRODUCTION

Deadlocks are common concurrency bugs of multi-threaded programs, caused by incorrect synchronizations that could make a program hang without any progress. In particular, deadlocks can be divided into two types: *resource deadlocks* and *communication deadlocks*. In resource deadlocks, a task (e.g., process/thread) cannot advance without the acquisition of the necessary resource (e.g., lock), while a communication deadlock occurs when a task is not receiving the required communication (e.g., message or signal). It is very challenging to detect and reproduce deadlocks, as they require certain thread interleavings to occur.

Existing detection tools mainly focus on resource deadlocks. Among them, static detection tools are known to have numerous false positives and scalability issues [16, 17, 19, 30, 37, 41]. In contrast, dynamic tools typically record the execution trace, and then detect hold-and-wait cycles among threads and locks [9, 10, 12, 25, 43]. They could *predict* deadlocks that did not occur in the current execution but may occur in a different execution. In the remainder of this paper, detect and predict are used interchangeably. In particular, iGoodLock [25] introduces lock dependency that describes the relationship between a thread, its requested lock, and its held locks, as further discussed in Section 2.2, in order to identify hold-and-wait relation. Whenever there exists a cycle of hold-and-wait relations, a deadlock is detected. Deadlock dependency has been adopted by multiple detectors [9, 10, 12, 43], but they focus more on reducing the overhead of recording and detecting. For instance, a recent work–AirLock [12]–reduces the search space of the detection, only imposing less than 5% overhead.

However, communication deadlocks related to condition variables do not get much attention that they deserve (possibly due to the difficulty), although condition variables are also widely employed in real applications. The condition variable is a type of synchronization that enables threads to wait for a particular condition, which further includes wait (e.g., `wait()`) and signal (e.g., `signal()` or `broadcast()`) primitives. Based on our investigation, very few tools have been developed in the past, but they either suffer from false positives or require significant manual effort. Among them, FindBugs [22] reports an alarm when a thread invoking a condition wait operation is holding more than one lock. This over-simplified pattern could easily introduce false positives, especially in real applications like MySQL, and cannot guide bug fixes without the detailed information of deadlocks that explains how the deadlocks can occur. CheckMate [24] records the trace of synchronization operations with manual annotations and then utilizes a model checker to explore all possibilities of deadlocks. However, it is too expensive to be employed with an average recording overhead of 10.8× and an even higher model checking overhead (up to 60×).

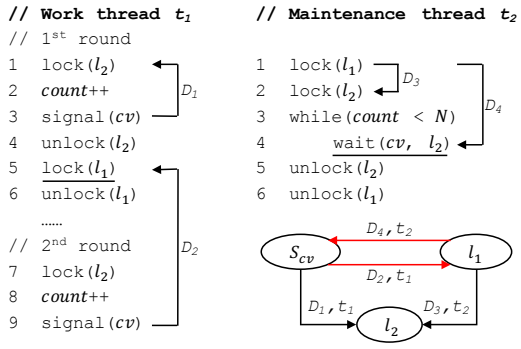

Figure 1: New deadlock in Memcached-1.5.19. The hold-and-wait relations D_{1-4} are shown in the bottom-right corner, where deadlock-related edges are highlighted with red colors.

This paper presents UnHang, a unified tool that predicts both resource and communication deadlocks. UnHang's major contribution is to extend the concept of lock dependency to **generalized dependency**, which includes both locks and condition variables. *Its basic idea is to treat a signal (related to a condition variable) as a special type of resource*. It further extends the "wait" and "hold" concepts to cover condition variables additionally. In particular, "wait" indicates a state that the thread is waiting for the signal on a condition variable, while "hold" is a state that the thread has not issued a signal (or released the resource) via `signal()` or `broadcast()`. Based on such extensions, UnHang constructs generalized dependencies by tracking synchronization events (e.g., all locks and condition variables) in an execution, and then detects hold-and-wait cycles of generalized dependencies to predict both types of deadlocks.

Figure 1 shows a communication deadlock in Memcached (**newly-detected** by UnHang). This example involves N worker threads and one maintenance thread, with multiple rounds of maintenance. For each round, the maintenance thread will wait (at line 4) for the signal issued from N workers threads (at line 3). A communication deadlock can occur if the maintenance thread waits on condition variable *cv* before a worker thread acquires the lock l_1 (at line 5, to exit from a previous maintenance round). Because the maintenance thread is holding the lock l_1, all later worker threads cannot acquire l_1 and then cannot send the signal required by the maintenance thread, causing a deadlock. This example's hold-and-wait relations are shown as a graph in the bottom-right corner of Figure 1. In this graph, every resource (either a lock or a signal) is represented as a node in the graph, and an edge (related to a thread) indicates a hold-and-wait relation. When there is a cycle of hold-and-wait relation, a deadlock is detected. We explicitly put D_i there just for the explanation purpose. Based on these definitions, the worker thread t_1 is waiting for the lock l_1 while holding the resource (or signal) s_{cv} for the second round, which is related to the D_2, t_1 edge. Instead, the maintenance thread t_2 is waiting for the signal s_{cv} (related to *cv*) while holding the lock l_1, which is related to the D_4, t_2 edge. Therefore, D_2 and D_4 are consisting of a hold-and-wait cycle as shown in highlighted edges of Figure 1. Clearly, such a deadlock cannot be detected by lock dependency, as there is no loop between locks (with the edge D_3, t_2 only).

Although the basic idea of generalized dependency is very intuitive, there are several technical challenges. *Challenge 1: how to reduce false positives introduced by condition variables?* Condition variables are fundamentally different from locks, as (1) they are not mutually exclusive, (2) they have some predicates/conditions that can control the wait operation. Figure 2 shows an example with a cycle: the producer thread waits for signal s_{cv_1} while holding the signal s_{cv_2}, and the consumer thread waits for s_{cv_2} while holding the signal s_{cv_1}. However, this cycle is not a real deadlock, because the predicates of cv_1 and cv_2 conflict with each other, i.e., either `full()` or `empty()` is true at one time. *Challenge 2: how to construct generalized dependencies from an execution, especially with a low recording overhead?* During the execution, we have no idea initially whether a thread is holding a special resource (i.e., signal) until the `signal()` or `broadcast()` is invoked. This signal may consist of hold-and-wait relations with all past locks, even for situations holding one lock. However, traditional lock dependency focuses only on situations holding more than one lock. That is, generalized dependency requires recording orders of magnitude more events. *Challenge 3: how to reduce the prediction overhead?* The huge number of generalized dependencies require efficient detection, especially for situations with up to hundreds of threads, typical in real applications.

UnHang is implemented as a library that can be simply preloaded in order to collect synchronizations of multithreaded applications, without the modification and recompilation of applications. UnHang does not require a custom OS or hardware. UnHang has been evaluated on PARSEC benchmarks and several real applications. Based on our evaluation, UnHang imposes a geomean performance overhead of less than 3%, including the recording and predicting overhead. UnHang successfully detects known and **new** deadlocks in real applications, such as MySQL and Memcached. Note that UnHang may report deadlocks that cannot occur in reality, which share the same shortcomings with existing dynamic detection tools [7, 12, 15, 43]. Overall, this paper has the following contributions:

- It proposes generalized dependency that includes both locks and condition variables, enabling the detection of both resource and communication deadlocks in a unified tool.
- It proposes practical mechanisms to detect deadlocks by overcoming the correctness and performance issues.
- It presents the implementation of UnHang. Experimental results show that UnHang only imposes a geomean overhead of 3%, but detects all known deadlocks and two new deadlocks.

The remainder of this paper is organized as follows. Section 2 introduces the proposed generalized dependency and new rules defined for correctness. Section 3 further describes the design and implementation of UnHang, our prototype using generalized dependencies to predict deadlocks. Section 4 describes the experimental evaluation of UnHang. Section 5 discusses UnHang's weaknesses. After

```
  // Producer              // Consumer
1 lock(l)               1 lock(l)
2 while (full())        2 while (empty())
3     wait(cv1, l)      3     wait(cv2, l)
4 produce()             4 consume()
5 signal(cv2)           5 signal(cv1)
6 unlock(l)             6 unlock(l)
```

Figure 2: A false cycle of two condition variables.

that, Section 6 discusses some related work. In the end, Section 7 concludes this paper.

2 METHODOLOGY

In this section, we first revisit the concept of condition variables and lock dependency, then introduce the proposed generalized dependency and its cycle detection, as well as some terms used in the remainder of the paper.

2.1 Background

The condition variable is a synchronization mechanism that allows a thread to wait for a particular condition (i.e., predicate) to be true. There are two basic operations on condition variables, including `wait()` and `signal()`. A waiting thread typically acquires the "associated lock" (i.e., the lock used for the parameter in `wait()`) to check the predicate and invokes `wait()` when the corresponding predicate is false. It will release the associated lock before it starts waiting so that a signaling thread can acquire the lock to update the predicate. The waiting thread can be woken up by the signaling thread which typically sets the predicate to true before invoking `signal()` or `broadcast()`. Condition variables are widely used in multithreaded real applications, such as MySQL and Memcached. Condition variables may cause deadlocks, if the signaling thread cannot advance to issue the signal or the corresponding signal is lost (called as "lost signal").

2.2 Lock Dependency

A *lock dependency* is defined as a triple $D = (t, l, L)$ [25], indicating thread t tries to acquire the lock l while holding all locks in the *lockset* L. Multiple lock dependencies may form a *dependency chain* $C = [D_1, D_2, ..., D_n]$ if (1) the lock l_i in D_i is in the lockset L_{i+1} of D_{i+1}, and (2) there's no *conflict* among them. If two dependencies have a conflict, they cannot cause a deadlock. In one of the following situations, two dependencies D_i and D_j are considered to have a conflict : (1) $t_i == t_j$ (when two threads are the same thread); (2) $l_i == l_j$ (when two threads acquire the same lock); (3) $L_i \cap L_j \neq \emptyset$ (when two threads hold a common lock).

A dependency chain indicates a hold-and-wait relation in multiple threads. It becomes a ***cyclic chain*** *(or* ***cycle*** for simplicity) when lock l_n from the last dependency D_n in the chain is in the lockset L_1 of the first dependency D_1. A cyclic chain represents a hold-and-wait cycle, indicating a deadlock. The prediction is to identify cyclic chains of all lock dependencies. Clearly, the number of lock dependencies could significantly affect the prediction time.

2.3 Generalized Dependency

The generalized dependency extends lock dependency to include signals for condition variables (called ***condition signals***, or ***signals*** for simplicity) as a type of resource. A **generalized dependency** is also defined as a triple (t, r, R), where t is the thread, r is the ***requested resource*** (a lock or a condition signal) of the thread t, and R is a set of ***held resources*** (e.g., locks, condition signals) of the thread t. Based on the type of requested resource, a generalized dependency can be further divided into two types:

- *Mutex Dependency (MD)*: $MD = (t, l, R)$ represents that a thread t requests a mutex lock l while holding a set of locks and/or signals. A mutex dependency may not have condition signals in the set R.
- *Signal Dependency (SD)*: $SD = (t, s_{cv}, R)$ represents that a thread t waits for the signal s_{cv} (related to the condition variable cv) while holding a set of resources R, i.e., locks and/or signals.

We propose two basic rules for building a generalized dependency: (1) the *associated lock* of cv (i.e., the lock parameter in the invocation of the corresponding `wait()`) is not considered as a held resource when a waiting thread waits for s_{cv}, such as l_2 of the maintenance thread t_2 in Figure 1, as the lock is always released before the waiting begins; (2) a signal is considered as a held resource before a signaling thread invokes one corresponding `signal()` or `broadcast()`. One example is D_2 in Figure 1. Based on these definitions and rules, Figure 1's dependencies are listed as follows: the worker thread t_1 has two mutex dependencies $D_1 = (t_1, l_2, \{s_{cv}\})$ and $D_2 = (t_1, l_1, \{s_{cv}\})$, while the maintenance thread t_2 has a mutex dependency $D_3 = (t_2, l_2, \{l_1\})$ and a signal dependency $D_4 = (t_2, s_{cv}, \{l_1\})$ (without l_2). However, l_2 and $\{s_{cv}\}$ do not form a dependency, as l_2 is the associated lock of condition variable cv.

2.4 Prediction Using Generalized Dependencies

Based on generalized dependencies, the deadlock prediction is to identify a cyclic chain among all generalized dependencies. Similarly, multiple generalized dependencies may form a *dependency chain* $C = [D_1, D_2, ..., D_n]$ if a requested resource r_i (e.g., lock or condition signal) in D_i is in the held resource set R_{i+1} of D_{i+1}. Further, the following definitions on conflicts from lock dependency can be applied to generalized dependency: (1) $t_i == t_j$ (when two threads are the same thread); (2) $r_i == r_j$ (when two threads are requesting the same resource, e.g., lock or condition signal); (3) when two threads are holding a common lock in R_i and R_{i+1}. That is, only when two threads do not have a conflict, and also satisfy the above chaining condition, they will form a dependency chain.

However, generalized dependency should also handle the differences caused by condition signals (e.g., they are not mutually exclusive and have associated predicates), as discussed in Section 1. We further propose the following additional rules to reduce false positives. Note that these rules may lead to false negatives, but will never generate false positives.

2.4.1 Non-Conflicting Condition Signals. Because condition signals are not mutually exclusive, two generalized dependencies holding a common condition signal do not indicate a conflict. Figure 3 shows a real deadlock in MySQL [4] with three generalized dependencies: $SD_1 = (t_1, s_{cv_x}, \{l_1, s_{cv_y}\})$, $MD_1 = (t_2, l_2, \{s_{cv_x}\})$, $MD_2 = (t_3, l_1, \{l_2, s_{cv_y}\})$. Because the s_{cv_y} is not considered as a conflict between SD_1 and MD_2, these dependencies can form a cyclic chain $[SD_1, MD_1, MD_2]$, indicating a deadlock (as underlined lines in the figure).

2.4.2 Conflicting Condition Signals. As discussed in Section 1, two waiting threads cannot wait concurrently (thus cannot deadlock with each other) if only one of the corresponding predicates can be true at one time. That is, generalized dependencies waiting for different condition signals may conflict with each other, which is different

```
  // Thread t1          // Thread t2          // Thread t3
1 lock(l1)           1 lock(l2)           1 lock(l2)
2 lock(lx)           2 unlock(l2)         2 lock(l1)
3 wait(cvx, lx)      3 lock(lx)           3 signal(cvy)
4 unlock(lx)         4 signal(cvx)        4 unlock(l1)
5 Unlock(l1)         5 unlock(lx)         5 unlock(l2)
6 signal(cvy)
```

Figure 3: A deadlock in MySQL-5.6.9.

from lock dependency in that lock dependencies acquiring different locks are never treated as conflicts. To determine such a conflict, one solution is to analyze the predicates by tracking and analyzing memory accesses, which can be impractical especially for large-scale applications. Instead, UnHang employs the following rule: if two related condition variables in two dependencies share the same associated lock, they have a conflict. Figure 2 shows such an example: the producer thread's signal dependency is $SD_p = (t_p, s_{cv_1}, \{s_{cv_2}\})$, and the consumer thread's signal dependency is $SD_c = (t_c, s_{cv_2}, \{s_{cv_1}\})$. Because condition variables related to s_{cv_1} and s_{cv_2} share the same associated lock l, these two generalized dependencies are assumed to have a conflict. That is, there is no deadlock in this example. Note that this rule could incur false negatives when a deadlock involves two condition variables with non-conflict predicates but the same associated lock. However, it will never cause any false positives.

2.4.3 Spurious Cycles. UnHang aims to predict deadlocks latent in real-world applications with the following assumption: real-world applications are thoroughly tested so that they should not have "*obvious bugs*" that will definitely lead to the deadlock. Figure 4 shows a deadlock cycle where thread t_1 and t_2 wait for the signal from each other at the same time. This cycle will definitely lead to deadlocks. That is, programmers should have already identified and fixed such obvious bugs during testing. Hence, we believe that applications may utilize some sophisticated mechanisms (e.g., ad hoc synchronization [42]) to avoid them, and we treat such a cycle as a spurious cycle. In theory, UnHang may have some false negatives due to this assumption. To identify a spurious cycle, UnHang checks if all dependencies in the cycle are signal dependencies. If so, every thread in the cycle is waiting for a signal from another thread in the cycle, indicating an obvious deadlock.

```
  // Thread t1              // Thread t2
1 wait(cv1, l1)          1 wait(cv2, l2)
2 signal(cv2)            2 signal(cv1)
```

Figure 4: A cycle indicating an obvious deadlock.

Figure 5 shows another example of spurious cycles. In this figure, thread t_2 cannot send any signal without holding lock l_1. If thread t_1 acquires lock l_1 first, the program will deadlock. That is, the deadlock has a fair chance to occur and should have been exposed and fixed by the developers. Although a cycle exists between $(t_1, s_{cv}, \{l_1\})$ and $(t_2, l_1, \{s_{cv}\})$, UnHang will not report it as it's a spurious cycle. More specifically, UnHang also considers a cycle as spurious if the cycle involves only one thread waiting for a lock while holding a signal in the cycle and this thread sends the corresponding signal while holding the lock.

```
  // Thread t1              // Thread t2
1 lock(l1)               1 lock(l1)
2 lock(l2)               2 lock(l2)
3 while (p == false)     3 p = true
4     wait(cv, l2)       4 Signal(cv)
5 unlock(l2)             5 unlock(l2)
6 unlock(l1)             6 unlock(l1)
```

Figure 5: A cycle indicating a spurious deadlock.

3 IMPLEMENTATION

We implemented UnHang as a dynamic library for C/C++ applications with Pthreads [5]. It can be deployed easily without changing and recompiling applications.

3.1 Recording Generalized Dependencies

During the execution, UnHang intercepts synchronization operations on mutex locks and condition variables to record generalized dependencies. To reduce contention, UnHang employs a per-thread recording method so that there is no need to introduce additional locks when updating the generalized dependencies for each thread. As mentioned in Section 2, a generalized dependency includes a requested resource and some held resources. It is relatively easy to identify the requested resource: if a thread acquires a lock or waits on a condition variable, then the lock or the corresponding condition signal is the requested resource. For held resources, it is easy to identify the held locks of a thread. However, the held condition signal(s) is unknown until the future invocations of signaling functions (e.g., `signal()` or `broadcast()`). Taking thread t_1 of Figure 1 as an example, the condition signal s_{cv_y} can be only recognized as the held resource of t_1 at line 3 (for line 1-2) and line 9 (for line 4-8).

To record new dependencies when identifying a held signal, UnHang tracks the *recent synchronization statuses* (denoted as *RS*) and the *held locks* (denoted as *LS*). Algorithm 1 shows how UnHang tracks these information and records generalized dependencies for each thread. In particular, UnHang tracks three types of statuses in *RS*: (1) generalized dependencies, such as mutex dependencies (line 5-6) or signal dependencies (line 14-15), (2) first-level lock acquisitions where the thread holds no locks (line 3), (3) condition waits where the thread holds no additional locks other than the associated lock (line 12). It is important to track the latter two types of statuses, as they may form new dependencies with future condition signals. When a thread invokes a signaling function, UnHang checks the *RS* to update existing dependencies (line 19-20) or record new ones (line 22-23), where the function $Merge()$ is used to update the *RS* with newly recorded dependencies.

Table 1 illustrates how UnHang records generalized dependencies for the threads in Figure 1, assuming one maintenance round. UnHang constructs new dependencies on three types of synchronization operations: lock acquisitions, waits, and signals. For the worker thread t_1, UnHang updates the *LS* and *RS* with l_2 from the 1st synchronization. Since there's no held resource identified in t_1, UnHang does not record any dependency. When t_1 performs the 2nd synchronization, UnHang knows that t_1 has been holding s_{cv}, thus it checks the thread's recent status and builds a mutex dependency $MD_1 = (t_1, l_2, \{s_{cv}\})$. Then, UnHang updates *RS* with MD_1 as the

Algorithm 1: Record Dependencies

LS: per-thread lockset for held locks.
RS: per-thread cache for recent synchronization statuses.
DEP: per-thread hashmap for dependencies.

```
1  FUNCTION OnLOCK(l):
2      if LS.Empty() then
3          RS.Insert(l)
4      else
5          d := DEP.TryRecord(l, LS)
6          RS.Insert(d)
7      LS.Insert(l)
8  FUNCTION OnUNLOCK(l):
9      LS.Remove(l)
10 FUNCTION OnWAIT(cv):
11     if LS.OnlyContains(l_assoc) then
12         RS.Insert(cv)
13     else
14         d := DEP.TryRecord(cv, LS \ {l_assoc})
15         RS.Insert(d)
16 FUNCTION OnSIGNAL(cv):
17     RS_new := ∅
18     for e ∈ RS do
19         if e.IsDependency() then
20             e.AddHeldResource(cv)
21         else
22             d := DEP.TryRecord(e, {cv})
23             RS_new.Insert(d)
24     RS.Merge(RS_new)
```

most recent status. For the maintenance thread t_2, UnHang first creates a mutex dependency MD_2 upon the 2nd lock acquisition. When t_2 performs the wait operation, UnHang examines the held locks (i.e., l_1 and l_2), then builds a signal dependency $SD_1 = (t_2, s_{cv}, \{l_1\})$ while ignoring l_2 (the associated lock of cv). UnHang may update recorded dependencies upon a signal event (line 19-20 in Algorithm 1). For example in Table 1, if thread t_2 sends a signal on condition variable cv_x right after the 5th synchronization, UnHang will update MD_2 and SD_1 by adding the s_{cv_x} as a new held resource, respectively: $MD_2 = (t_2, l_2, \{l_1, s_{cv_x}\})$ and $SD_1 = (t_2, s_{cv}, \{l_1, s_{cv_x}\})$.

There is an important implementation question related to the most recent statuses: *how many recent statuses should we keep for each thread*. Typically, a larger number of statuses should help reduce false negatives, but with the cost of more memory consumption, higher recording overhead, as well as higher prediction overhead (due to the increased number of generalized dependencies). Moreover, the older a synchronization status is, the more likely it is followed by some implicit synchronizations (e.g., ad hoc synchronization) neglected by UnHang. That is, an older status may introduce more false positives. Therefore, UnHang currently only tracks the eight most recent statuses. Our experiments with real applications show that this is sufficient to find all known and new deadlocks with low overhead.

Table 1: Updating dependencies and other information for the two threads in Figure 1, assuming one maintenance round. The first column lists the synchronization operations of each thread in execution order, respectively.

Worker thread t_1			
Synchronization	**LockSet** *LS*	**Recent Status** *RS*	**Dependency**
1: $lock(l_2)$	$\{l_2\}$	$< l_2 >$	No Update
2: $signal(cv)$	$\{l_2\}$	$< l_2, MD_1 >$	$MD_1 = (t_1, l_2, \{s_{cv}\})$
3: $unlock(l_2)$	$\emptyset$	$< l_2, MD_1 >$	No Update
4: $lock(l_1)$	$\{l_1\}$	$< l_2, MD_1, l_1 >$	No Update
5: $unlock(l_1)$	$\emptyset$	$< l_2, MD_1, l_1 >$	No Update
Maintenance thread t_2			
Synchronization	**LockSet** *LS*	**Recent Status** *RS*	**Dependency**
1: $lock(l_1)$	$\{l_1\}$	$< l_1 >$	No Update
2: $lock(l_2)$	$\{l_1, l_2\}$	$< l_1, MD_2 >$	$MD_2 = (t_2, l_2, \{l_1\})$
3: $wait(cv, l_2)$	$\{l_1, l_2\}$	$< l_1, MD_2, SD_1 >$	$SD_1 = (t_2, s_{cv}, \{l_1\})$
4: $unlock(l_2)$	$\{l_1\}$	$< l_1, MD_2, SD_1 >$	No Update
5: $unlock(l_1)$	$\emptyset$	$< l_1, MD_2, SD_1 >$	No Update

In addition, UnHang collects the call stacks of synchronization events such that it can provide a meaningful report for programmers to understand the bugs. It further utilizes the call stacks for performance optimizations, as discussed in the following section.

3.1.1 Performance Optimizations. To be employed in the production environment, UnHang needs to minimize the recording overhead. Thus, UnHang first employs some recording policies from UnDead [43], a dynamic tool that detects resource deadlock based on lock dependencies. Although UnDead cannot detect deadlocks caused by condition variables, it defines some policies for efficient recording: (1) it does not record duplicated dependencies for each thread (which have the same requested resource and held resources). (2) It does not collect the call stack for the first-level lock acquisitions, as some applications (e.g., Fluidanimate) may have a large number of single-level lock acquisitions. (3) It does not repeatedly collect the call stack for the same lock acquisition that occurs multiple times.

In addition to the aforementioned policies, UnHang proposes a *call-stack based optimization* to decrease the total number of dependencies as it has a significant impact on detection efficiency and memory consumption [10, 12], especially in applications that create numerous synchronization objects (each of which may lead to a unique dependency). The basic idea is to record a subset of unique dependencies from each call stack. Since the number of synchronization patterns on a call stack is predetermined by the program, a subset of dependencies on the same call stack can be adequate to represent most (if not all) patterns on that call stack. More specifically, given a call stack, UnHang records the first N unique dependencies, then records new dependencies in a sampling manner with decreasing sampling rate: the more dependencies UnHang records from a call stack, the less likely it will record again on that call stack. In Algorithm 1, the function $TryRecord()$ implements these optimizations to record a dependency (with specified requested resource and held resources) only if necessary. In addition, UnHang avoids recording redundant condition variables by grouping them based on their initialization call stack, and only tracks the first M condition variables from each group. After that, it also gradually reduces the possibility of the recording. Currently, UnHang chooses N and M to be 32 and

2 correspondingly, considering both efficiency and effectiveness for deadlock prediction.

3.2 Detecting Cycles of Generalized Dependencies

UnHang predicts deadlocks by identifying a cycle of generalized dependencies, as discussed in Section 2.4. A traditional method is to start from the threads by scanning the dependencies of every thread and to confirm whether each dependency can form a potential cycle with other dependencies in other threads via DFS-based cycle detection [9, 10, 25, 43]. A recent work AirLock [12] demonstrates the inefficiency of such a method on a large DFS search space and proposes the concept of "reachability graph" to reduce the search space.

The reachability graph is actually a directed graph, where each node is related to a lock and the edge between nodes implies a direct or indirect hold-and-wait relation. More specifically, a direct edge from l_i to l_j indicates a thread is requesting lock l_j while holding lock l_i. An indirect edge indicates that there is a path between different nodes, composed of multiple direct edges. Typically, predicting deadlocks consists of the following three steps: (1) check whether there is a cycle between different nodes; (2) construct the detailed cycle using DFS when the cycle has indirect edges; (3) check the conflict among dependencies related to the found cycle. If there is a set of non-conflicting dependencies that matches the found cycle, a potential deadlock is reported. Otherwise, it is a spurious cycle that should not be reported. With the reachability graph, one can decide if a pair of nodes have a cyclic path before starting the DFS search, thus reducing the search space in previous work [7, 10, 25, 43]. In addition, AirLock updates the reachability graph during execution such that it can save some edges to the external storage in order to reduce memory consumption.

UnHang adopts the idea of reachability graph from AirLock [12] and extends it as follows: (1) UnHang treats both locks and condition signals as nodes to detect communication deadlocks. (2) UnHang minimizes the interference to the original execution by only constructing the graph after recording (at program exits), instead of at execution time. Based on our understanding, AirLock builds a global graph and uses a global map to record lock dependencies without thread-specific data structures. Thus, it can unnecessarily introduce concurrency issues (e.g., high contention) for applications with a large number of threads and intensive synchronization. UnHang trades the memory consumption for the low overhead by constructing the graph later and reduces the contention by using thread-local recording. More specifically, UnHang constructs the graph by examining the recorded dependencies and grouping the synchronization objects by their initialization call stacks (or memory addresses if they are statically initialized). During cycle detection, UnHang maps a group back to the actual objects only if it's involved in a cycle. (3) UnHang further reduces the prediction overhead by embedding the reachability path information (a sequence of dependency groups connecting two nodes) into the graph, and prunes any path that has conflicts during graph construction. With this improvement, UnHang could directly report a cyclic path, instead of using the DFS search to construct the cyclic path again as AirLock.

By default, UnHang performs the prediction at program exits. It registers its analysis with the `destructor` attribute and intercepts all failure handling functions, such as `SIGHUP`, `SIGQUIT`, `SIGINT`, `SIGTERM`, and `SIGSEGV`. For long-running applications such as server applications, UnHang creates a monitor thread that periodically examines the current program status to provide timely detection. More specifically, UnHang periodically takes a snapshot of the threads' waiting statuses, then detects cycles in the current snapshot and reports cycles when statuses are not changed across two periods.

3.2.1 Detecting Lost Signals. There is one type of deadlock that cannot be represented by a cycle of generalized dependencies, which is the "Lost Signals". Lost Signals can happen when the signaling thread did not hold or is not holding the associated lock, which may introduce race conditions on the predicate. That is, the signal can be delivered before a thread is actually waiting on the related condition variable. Existing tools, such as Valgrind DRD [2, 31], report invocations of `pthread_cond_signal()` when the associated lock is not held. However, a signaling thread can release the lock before sending the signal without causing lost signals. Given that UnHang records the held locks and recent statuses for each thread, it reports a lost signal if the associated lock is not acquired recently by the signaling thread before the signaling.

4 EXPERIMENTAL EVALUATION

The experiments were conducted on a two-socket machine, where each socket is an Intel(R) Xeon(R) Gold 6230 processor with 20 cores. For the evaluation, we utilize 16 hardware cores in node 0 to exclude the NUMA effect. This machine has 256GB of main memory, 20MB of L2 cache, and 1280KB L1 cache. The underlying OS is Ubuntu 18.04.6 LTS, installed with the Linux-5.4.0-81.

4.1 Effectiveness

In this section, we evaluate whether UnHang can predict known resource and communication deadlocks. In particular, we compare UnHang with another recent work – UnDead [43]. There exist other works, such as CheckMate [24] or AirLock [12]. CheckMate is not open-source. It focuses on Java applications and relies on source code annotations and model checking. AirLock has a similar prediction ability as UnDead [43] in that they both only predict resource deadlocks. We have contacted the authors of AirLock for the source code, but they cannot provide it. However, based on the results shown in Table 2, UnHang should impose a similar overhead (if not less) as AirLock.

The evaluation is performed on nine known deadlock bugs from real-world applications, such as Apache, Bodytrack [8], HawkNL, Memcached, MySQL, and SQLite. Among these bugs, six of them are related to condition variables, while the other three are related to resource deadlocks (collected from existing work [9, 10, 12, 26, 43]). For Apache, HawkNL, MySQL, and SQLite, we utilized the test cases collected from the related bug reports, which is similar to existing work [9, 10, 12, 43]. We did not use specific test cases for Bodytrack and Memcached as the relevant logic can be exercised easily using given normal inputs. SQLite-3.25.2 is reported by AirLock to have a cycle inside [12]. In fact, we confirm that this is a false positive caused by memory reuse, as discussed in the following.

Table 2: Evaluation of UnHang's detection effectiveness.

Application	Bug ID	Deadlock Info				Execution Info			Report (Known/New/FP)		Detection Time	
		Type	#Thd	#Lock	#Cv	#Thd	#Lock	#Cv	UnDead	UnHang	UnDead	UnHang
HawkNL-1.6b3	n/a	Resource	2	2	0	401	403	0	1/0/0	1/0/0	2.6s	86ms
MySQL-5.6.10	62614	Resource Resource	3 2	3 2	0 0	23	311	33	2/0/0	2/0/0	1s	1ms
SQLite-3.3.3	1672	Resource	2	2	0	3	2	0	1/0/0	1/0/0	1ms	1ms
SQLite-3.25.2 [12]	None	None	n/a	n/a	n/a	81	299	0	0/0/1	0/0/1	67s	3ms
Apache-2.3.0-dev	42031	Communication	2	1	1	6	4	2	0/0/0	1/0/0	1ms	1ms
Bodytrack	n/a	Lost Signal	1	0	1	18	6	3	0/0/0	1/0/0	1ms	8ms
Memcached-1.5.19	567	Lost Signal **Communication**	1 2	0 1	1 1	22	16719	4	0/0/0	1/**1**/0	68s	67ms
MySQL-5.1.57	60682	Communication	3	2	1	237	1458	27	0/0/0	1/0/2	>3h	83ms
MySQL-5.5.8	50038	Communication	2	1	1	24	319	26	0/0/0	1/0/0	1ms	2ms
MySQL-5.6.9	68251	Communication **Communication**	3 3	2 1	1 2	29	359	123	0/0/0	1/**1**/0	1ms	3ms

Table 2 lists more details of these bugs, including the application names, bug ID, deadlock information, execution information, the number of reported deadlocks and the time spent on detection. More specifically, we list the number of threads ("#Thd"), the number of locks ("#Lock"), and the number of condition variables ("#Cv") involved in each deadlock and execution, respectively. In column "Report", "Known" indicates the number of the detected deadlocks corresponding to the known bug, while "New" and "FP" indicate the number of newly-detected deadlocks and false positives, respectively. Note that the "Detection Time" is the time spent at program exit, where both tools perform the detection. In addition, we highlight the two new bugs detected by UnHang with bold text.

Overall, UnHang can predict all known deadlocks and report corresponding cycle information, as shown in Figure 6. UnHang also uncovers one new bug in each of the widely-used applications: Memcached-1.5.19 and MySQL-5.6.9. UnHang reports three false positives. In the following, we explain how UnHang could help guide the bug fixes. Then we explain the new bugs detected by UnHang. In the end, we further explain why UnHang could introduce false positives, sharing the same shortcoming as existing work [7, 12, 15, 43].

```
A communication bug caused by 2 threads.
Thread 1 is waiting for the signal related to
     condition variable 0x63d060 at:
    0: wait_for_thread_registration at thread.c:125
    1: (inlined by) pause_threads at thread.c:188
    ......
while it is holding the lock (0x63d140), acquired at:
    0: pause_threads at thread.c:156
    1: assoc_maintenance_thread at assoc.c:259
    ......

Thread 2 is waiting for the lock (0x63d140) at:
    0: memcached_thread_init at thread.c:851
    1: main at memcached.c:9622
while it is holding the signal related to condition
     variable 0x63d060 at:
    0: register_thread_initialized at thread.c:132
    1: thread_libevent_process at thread.c:530
    ......
```

Figure 6: Report for the new deadlock shown in Figure 1.

Reporting deadlock details: Figure 6 shows an example of a bug report generated by UnHang, corresponding to the **new** deadlock in Memcached-1.5.19. Basically, it is able to show the deadlock type and threads involved in the bug. It also shows the relevant hold-and-wait relation in each thread, which can help programmers understand the bug. For the shown bug, if a lock belongs to the first-level acquisition, UnHang reports the initialization call stack instead, as it does not record the call stack for first-level lock acquisitions due to performance concerns.

New deadlocks: UnHang detects two new deadlocks from Memcached and MySQL, respectively. The deadlock from Memcached is shown in Figure 1, which has been discussed before. The new deadlock from MySQL is shown in Figure 7. This deadlock involves three dependencies: $(t_1, s_{cv_{prep\_xids}}, \{l_{log}\})$, $(t_2, s_{cv_{log\_send}}, \{cv_{prep\_xids}\})$, and $(t_3, l_{log}, \{s_{cv_{log\_send}}\})$. These dependencies form a cycle. In particular, thread t_1 waits on cv_{prep_xids} (line 2), t_2 waits on cv_{log_send} (line 1), and t_w is blocked on lock l_{log} (line 1) which is held by t_1.

False Positives: Similar to existing work [7, 12, 15, 43], UnHang may report some false positives. There are two types of false positives. The first type can be caused by ad hoc synchronization, as shown in Figure 8. MySQL implements a synchronization based on a custom mutex m which includes a lock l_m, a condition variable cv_m , and an integer $lock\_word$. It provides two operations (mutex_enter and mutex_exit) and relies on two operations (mutex_test_and_set and mutex_reset_lock_word) to

```
// Thread t1          // Thread t2          // Thread t3
1 lock(l_log)         1 Wait(cv_log_send)   1 lock(l_log)
2 Wait(cv_prep_xids)  2 Signal(cv_prep_xids) 2 unlock(l_log)
3 unlock(l_log)                             3 Signal(cv_log_send)
```

Figure 7: A new deadlock found in MySQL-5.6.9. The associated locks and predicates are omitted for the simplicity.

```
  mutex_enter(m) { // customized mutex acquisition
1     if (!mutex_test_and_set(m))
2         return
3     wait(cv_m, l_m)
  }

  mutex_exit(m) {  // cutomized mutex release
1     mutex_reset_lock_word(m))
2     signal(cv_m)
  }

  // Thread t1                    // Thread t2
1 lock(l)                      1 lock(l)
2 unlock(l)                    2 mutex_enter(m)
3 mutex_enter(m)               3 mutex_exit(m)
4 mutex_exit(m)                4 unlock(l)
```

Figure 8: False positive from MySQL. Operations on cv_m's associated lock and predicate are omitted for the simplicity.

atomically check and set the *lock_word* which indicates if the mutex is acquired. When invoking `mutex_enter(m)`, a thread will wait on cv_m if m's *lock_word* equals 1 (i.e., m has been acquired). Otherwise, it will set *lock_word* to 1 without waiting. When a thread invokes `mutex_exit(m)`, it will reset *lock_word* to 0 and send the corresponding signal. In Figure 8, a cycle exists between $(t_1, l, \{cv_m\})$ and $(t_2, cv_m, \{l\})$. However, the two threads will not deadlock: if t_2 is waiting for t_1 to release m while holding l, t_1 must have already acquired m and will not wait for l (due to the ad hoc synchronization based on the *lock_word*). `UnHang` is unaware of such a synchronization, and thus may report a deadlock mistakenly. To solve this false positive, it is beneficial to employ deadlock confirmation techniques or identify ad hoc synchronization, which is out of the scope.

The second type of false positive can be caused by memory reuse for dynamically allocated objects, such as the cycle reported in SQLite-3.25.2. Note that AirLock reports such a cycle as well. However, we investigate this more carefully, and find that this is not a real deadlock. The problem is caused by a lock object reusing the memory address of a previously-freed lock. Such a bug can be avoided if we could record the timestamps of synchronization initialization so that we could differentiate synchronizations with different timestamps. However, this method may significantly increase the recording overhead. As such bugs appear very rarely, `UnHang` prefers to reduce the recording overhead.

4.2 `UnHang`'s Overhead

In this section, we focus on both the performance and memory overhead of `UnHang`. To further evaluate `UnHang`'s prediction overhead, we also add another version of `UnHang` – `UnHang-Log` that only traces the execution without performing any prediction.

Evaluated Applications: We evaluated the performance and memory overhead on 19 applications. 13 of them are from the PARSEC suite [8], while others are real applications: Apache, Memcached, MySQL, Pbzip2, Pfscan, and SQLite. For the PARSEC applications, we used the native input set with 16 threads. Apache was evaluated by sending 10,000,000 requests via the ab [1] with 16 requests at a time. Memcached was evaluated by running the Memtier_benchmark [34] with 16 threads for 300 seconds. MySQL

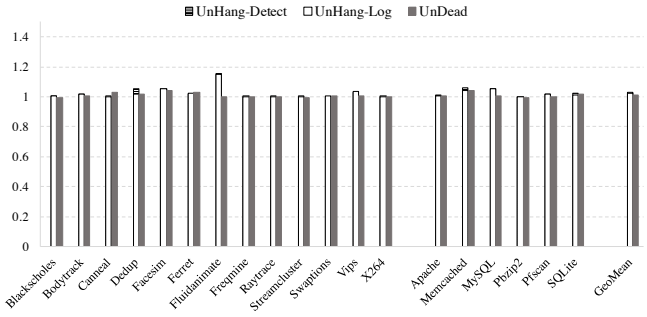

Figure 9: Normalized performance overhead over the default Pthreads library.

was evaluated by using the Sysbench to send 100,000,000 queries with 500 threads. SQLite was tested using "threadtest3.c" [3], a stress test program for SQLite. Pfscan was evaluated by performing a keyword search within 500MB of data. Pbzip2 was evaluated by compressing on a 600MB data file. The results shown in this section are the average results of 10 executions. *Note that `UnHang` does not report any false positive for all applications in this evaluation.*

4.2.1 Performance Overhead. Figure 9 shows `UnHang`'s performance overhead. For server applications, including Apache, Memcached, and MySQL, Figure 9 shows their throughput. In this figure, all results are normalized to the original values without using the tools. We compared `UnHang` with UnDead, which can only predict resource deadlocks. On average (geomean), `UnHang` only imposes a performance overhead of 2.7% , which is slightly higher than UnDead (0.9%). In addition, `UnHang` imposes more than 5% overhead on only five applications, including Dedup (5.4%), Facesim (5.4%), Fluidanimate (15.2%), Memcached (5.9%), and MySQL (5.3%). We also evaluated `UnHang-Log`, which is the performance overhead when `UnHang` only performs the logging. By comparing the difference between `UnHang-Log` and `UnHang`, we can know better about `UnHang`'s detection overhead (shown as "`UnHang-Detect`" in the figure). On average, `UnHang-Log` imposes a performance overhead of 2.2%, indicating that `UnHang`'s detection overhead is only 0.5%.

Table 2 also lists the detection overhead of `UnHang` and UnDead on real deadlocks. Since UnDead neither performs the optimizations discussed in Section 3.1.1 nor uses the reachability graph, its DFS-based detection can be quite inefficient given a huge search space. As shown in the table, UnDead's detection overhead can be many orders of magnitude larger than `UnHang`'s, for example on HawkNL-1.6b3, SQLite-3.25.2, and MySQL-5.1.57. This also confirms the results from AirLock [12]. In comparison, `UnHang` significantly reduces the detection overhead to the level of dozens of milliseconds. Although we cannot compare with AirLock directly, we believe that such overhead is comparable to AirLock.

`UnHang` imposes a higher overhead than UnDead due to the following reasons: (1) `UnHang` needs to record all single-level locks (where a thread only acquires a lock) to identify generalized dependency, while UnDead skips such locks because single-level locks will not cause traditional resource deadlocks. For generalized dependency, a single-level lock may combine with condition signals to

Table 3: Characteristic of evaluated applications.

Applications	Time (s)	Locks		Condition Variables				Dependencies	
		Objects	Acquisitions	Objects	Tracked	Signals	Waits	UnHang	NoOpt
Blackscholes	20.4	0	0	0	0	0	0	0	0
Bodytrack	15.1	6	1858851	3	3	3397	36856	9	4097
Canneal	31.3	1	15	0	0	0	0	0	0
Dedup	12.9	2199	999527	168362	19	259627	4397	2176	53586
Facesim	256.6	17	8168748	2	2	369247	1970860	423	65536
Ferret	58.7	5	35034	10	10	35034	16696	229	73728
Fluidanimate	20.3	92396	1723264500	1	1	8000	61337	155	10491
Freqmine	58.7	0	0	0	0	0	0	0	0
Raytrace	40.9	16	9431	16	16	3200	3209	85	7502
Streamcluster	36.6	1	2688	1	1	168	2042	6	168
Swaptions	12.6	0	0	0	0	0	0	0	0
Vips	20.7	71	649013	19	13	318687	139385	1345	54940
X264	34.4	35	202776	35	14	34816	4179	1392	218060
Apache	440	72	48500155	2	2	13573437	13573458	8	4101
Memcached	300	16735	169617675	4	4	20	15	15151	77824
MySQL	224	4328	939779967	3608	51	38268	215455	20966	157863
Pbzip2	6.1	7	304	6	6	71	24	127	312
Pfscan	0.7	4	36	3	3	19	12	34	35
SQLite	307	465	170877368	0	0	0	0	3623	169427

form new dependencies (e.g., Figure 1). (2) `UnHang` further records call stacks of the initialization of all locks to assist the identification of locks, which adds significant overhead for applications with a large number of locks (e.g., Fluidanimate). (3) `UnHang` records information related to condition variables, which is unnecessary for UnDead. In fact, `UnHang` imposes much less detection overhead than UnDead, as shown in Table 2.

We further investigate the source of `UnHang`'s overhead by collecting the characteristics of locks and condition variables as shown in Table 3. The information related to locks includes the number of objects and lock acquisitions. For condition variables, we collect the number of condition variables ("Objects"), the number of condition variables tracked by `UnHang` ("Tracked"), the number of condition signals ("Signals") and waits ("Waits"). Note that the "Signals" column actually includes the number of signal broadcasts, which explains why the number of signals can be smaller than that of waits (e.g., Facesim), since a signal broadcast may wake up all threads waiting on the related condition variable. We also collected the number of generalized dependencies recorded by `UnHang` and a special build of `UnHang` (named `UnHang-NoOpt`, or "NoOpt" for simplicity), respectively. `UnHang-NoOpt` does not have the call-stack based optimizations described in Section 3.1.1 and will record all dependencies on each call stack.

Based on our investigation, Fluidanimate has around 1.7 billion lock acquisitions on 92K lock objects, and 69K condition variable operations in around 20 seconds of execution. Therefore, collecting the call stack of initializations and intercepting the lock acquisitions together adds non-negligible overhead. Similarly, `UnHang` imposes more overhead for Dedup, Facesim, Memcached, and MySQL, due to a large number of synchronizations.

We also observed that `UnHang` can significantly reduce the number of synchronizations and dependencies to be recorded. For instance, Dedup has 168362 condition variables, but `UnHang` only tracked 19 of them. Similarly, `UnHang` also only tracked 51 out of 3606 condition variables in MySQL. When collecting data for `UnHang-NoOpt` in Table 3, we limited the maximum number of dependencies recorded for each thread to 4096, instead of allowing `UnHang-NoOpt` to keep recording. Even with this limitation, we can still observe a significant difference between `UnHang` and `UnHang-NoOpt`. For instance, `UnHang-NoOpt` records 218060 dependencies from X264, which is 156X more than `UnHang`. Nevertheless, `UnHang-NoOpt` does not detect more bugs than `UnHang` in our evaluation. Thus, `UnHang`'s call-stack based optimization is essential considering its efficiency and effectiveness.

Overall, `UnHang` imposes little performance overhead for most applications. Compared to UnDead, `UnHang` imposes a similar overhead for almost all applications, except Fluidanimate, while being capable of predicting deadlocks caused by condition variables.

4.2.2 Memory Overhead. We collected memory overhead in two ways depending on the application type. For non-server applications, such as Pbzip2, Pfscan, and all PARSEC applications, the maximum memory consumption is collected from the `maxresident` field in the output of the `time` utility. For server applications, such as Apache, Memcached, and MySQL, the maximum memory consumption is collected from the `VmHWM` field in the `/proc/PID/status` file.

Table 4 shows the memory consumption of `UnHang`, compared with UnDead and the default Pthreads library [5]. We also collected the memory consumption of `UnHang-NoOpt` which records all dependencies on each call stack. We further divided applications into two types, one with small footprints (less than 100 MB), and the other one with large footprints (no less than 100MB).

From Table 4, we conclude that `UnHang` imposes reasonable overhead for applications with a large footprint, as the geomean overhead is only 2.7%. For applications with a large number of synchronization objects and events, such as Fluidanimate and MySQL, `UnHang` consumes more memory to record the call stacks. Of course, UnDead's memory overhead is even smaller, with 1.4% for these applications. Table 4 also shows that `UnHang-NoOpt`'s geomean overhead is much larger (10% for the same applications). For example, `UnHang-NoOpt`'s overhead on X264 is 195.2% which

Table 4: Memory consumption (MB) and overhead over the default Pthreads library.

Applications	Default	UnDead		UnHang		UnHang-NoOpt	
		Mem	Overhead	Mem	Overhead	Mem	Overhead
Large Footprint (≥ 100MB)							
Blackscholes	613	619	0.9%	621	1.3%	622	1.4%
Canneal	851	856	0.6%	858	0.8%	858	0.8%
Dedup	1505	1534	1.9%	1617	7.4%	2389	58.7%
Facesim	310	317	2.2%	320	3.2%	596	92.2%
Ferret	106	113	6.6%	116	9.4%	429	304.7%
Fluidanimate	209	214	2.4%	332	58.8%	377	80.3%
Freqmine	1275	1276	0.1%	1280	0.1%	1280	0.1%
MySQL	319	450	41%	541	69.5%	1560	389.0%
Pbzip2	1285	1291	0.4%	1289	0.3%	1298	1.0%
Pfscan	516	524	1.6%	524	1.5%	524	1.6%
Raytrace	1287	1288	0.1%	1289	0.1%	1289	0.1%
Streamcluster	109	117	7.3%	117	7.3%	118	8.2%
X264	480	490	2.1%	490	2.1%	1417	195.2%
Total	8865	9089	2.5%	9394	6.0%	12757	43.9%
GeoMean Overhead			**1.4%**		**2.7%**		**10.0%**
Small Footprint (< 100MB)							
Apache	3	13	333.3%	15	400%	32	966.7%
Bodytrack	34	39	14.7%	41	20.6%	58	70.6%
Memcached	7	50	614.3%	59	742.9%	85	1114.2%
SQLite	31	40	29.0%	42	35.5%	796	2467.7%
Swaptions	7	12	71.4%	14	100%	14	100%
Vips	55	64	16.4%	68	23.6%	293	432.7%
Total	137	218	59.1%	239	74.5%	1278	958.7%
GeoMean Overhead			**68.4%**		**89.5%**		**448.3%**
Overall Total	9002	9307	3.4%	9633	7.0%	14035	55.9%
Overall GeoMean Overhead			**4.9%**		**8.2%**		**33.2%**

is 92X more than UnHang. This indicates that UnHang's optimizations significantly reduce the memory overhead (and the prediction runtime overhead). We also notice that both UnDead and UnHang impose higher overhead for applications with small footprints, with 68.4% and 89.5% respectively. This is understandable as they both include some initialization overhead for bookkeeping or other purposes. However, these applications with small footprints should not be a big concern, as their memory consumption is small in general. On average, UnHang's memory overhead is only 8.2%, which is slightly higher than UnDead (4.9%). This is because UnHang has to record much more information, as discussed in Section 3. Meanwhile, UnDead's overhead is significantly less than UnHang-NoOpt (33.2%), indicating the effectiveness of UnHang's optimizations.

5 LIMITATIONS

As a prototype built on generalized dependency, UnHang still has some limitations: (1) As discussed in Section 4.1, UnHang may report some false positives, which shares the same shortcomings with almost all existing dynamic tools [7, 12, 15, 43]. False positives can be caused by ad hoc synchronization and memory reuse. One possible solution for eliminating these false positives is to integrate some dynamic confirmation tools [9, 11, 25, 35]. As shown in Section 4.1, UnHang only reports a small number of false positives, while it reports all known deadlocks and reports some new bugs. (2) Similar to all dynamic tools, UnHang cannot predict a deadlock if its relevant synchronization events are not exercised at runtime. (3) UnHang currently only predicts deadlocks caused by mutex locks and condition variables. However, the idea of generalized dependency could be applied to other synchronization primitives, which will be our future work. (4) Although UnHang detects lost signals by checking if the signaling thread ever acquired the associated lock before the signaling (as discussed in Section 3.2.1), it may miss some lost signals because acquiring the associated lock does not necessarily ensure race-free on the predicate.

6 RELATED WORK

In the following, we discuss existing tools for detecting communication and resource deadlocks separately.

Detecting communication deadlocks: Only a few existing works detect deadlocks caused by condition variables. Among them, Agarwal et al. [6] detect lost signals by checking if a wait operation always happens before the corresponding signaling event. However, they do not consider the interactions between condition variables and locks. Pattern-based approaches [22, 40] may have an oversimplified pattern that can easily lead to false positives. For instance, FindBugs [22] reports an alarm if a thread waits on condition variables while holding multiple locks, although such a pattern does not necessarily lead to a deadlock. Further, it provides no information to guide bug fixes, as developers cannot know how deadlocks may occur.

CheckMate [24] is a previous state-of-the-art detector for both resource and communication deadlocks. CheckMate records more information (which relies on source code annotations) than UnHang, including changes to the predicate, and the wait/notify events that did not occur because the predicate was false in the execution. It then builds a simple program based on the records and model-checks the simple program to find deadlocks. Thus, theoretically, CheckMate can detect more deadlocks than UnHang. CheckMate can have false positives, as the deadlocks detected from the simple program may not be feasible in the original program. It also can have false negatives as the simple program is built from single execution. CheckMate's

overhead is prohibitive: it imposes more than 10× overhead on its recording phase and 60× on its model checking phase. In addition, CheckMate requires users to manually annotate the source code for its recording. These issues can become more serious if CheckMate is used in large-scale applications.

Some existing work aims to verify or guarantee deadlock-freedom in a concurrent program built on condition variables: Laneve et al. [29] propose a behavioral type system to analyze programs that use wait-notify coordination and guarantee deadlock-free for well-typed programs. Hamin et al. [21] ensure deadlock-freedom by verifying that each wait has a corresponding notification. Gomes et al. [20] apply Petri Net analysis to verify if every thread synchronizing under a set of condition variables eventually exits the synchronization block. In [28], the authors use a theorem prover to analyze the boolean conditions related to condition variables, focusing on the system design rather than the misuse of synchronization primitives. These approaches typically rely on sophisticated but heavy static analysis. They are orthogonal to `UnHang`.

Detecting resource deadlocks: Resource deadlocks have been extensively studied in the past. Among them, static approaches [16, 17, 19, 30, 37, 41] detect deadlocks based on the analysis of the source code. They can easily produce numerous false positives and often do not scale to large programs. Dynamic detection tools [7, 9, 10, 12, 18, 25–27, 36, 38, 43] perform detection based on observed events from real executions, thus are more likely to find real deadlocks. GoodLock [7] constructs a lock order graph based on lock acquisitions and detects cycles within the graph to identify potential deadlocks. DeadlockFuzzer [25] introduces the lock dependency concept and the iGoodLock algorithm to detect cyclic lock dependency chains. Multiple tools are proposed to reduce the search space. MulticoreSDK [15] performs cycle detection in a location-based lock order graph, where lock acquisitions from different threads on the same code location are grouped together and different groups sharing a lock are further merged. MagicLock [10] prunes locksets that can never form a cycle and designs a thread-specific strategy DFS algorithm to find cycles. AirLock [12] further improves the detection algorithm by using a reachability graph to reduce the search space and quickly determine if there's a cycle between two nodes. UnDead [43] is a deadlock detection and prevention tool with a novel prevention approach and various optimizations (such as pruning duplicated dependencies and reducing collections of call stacks) to reduce its performance overhead. It has a similar workflow as Dimmunix [26] in that they both detect deadlocks in current execution and prevent these deadlocks in future executions. UnDead's detection is based on the existing DFS algorithm [9, 25], thus can be very inefficient when the search space is huge.

Note that the cycles detected by the aforementioned approaches are predictive, i.e., they do not necessarily indicate a real deadlock. Some tools further try to confirm the detected deadlocks by controlling the scheduling of threads [9, 10, 25, 38]. For example, PickLock [38] predicts the deadlocking schedule based on the lock-sets and lock-acquisition history from one execution. There are some works that provide sound deadlock prediction (i.e., without false positives) [13, 27]. Dirk [27] aims to provide maximal and sound deadlock prediction, relying on a Satisfiability Modulo Theory (SMT) solver to identify if specific constraints can be satisfied given a feasible trace, similar to RVPredict [23]. However, checking satisfiability still remains intractable for large traces, and constructing feasible traces requires data-flow analysis that can introduce heavy runtime overhead. SeqCheck [13] collects traces of memory accesses and synchronizations to check if a sequence of these events in a specified order is feasible and predict concurrency bugs offline. `UnHang` can be extended to trace memory accesses and adopt these approaches to further reduce false positives (but with significantly more overhead).

Some works aim to reveal concurrency bugs by generating tests [14, 32, 33, 39]. With these test generators, `UnHang` could record more synchronization patterns and find more deadlocks.

7 CONCLUSION

This paper proposes generalized dependency that can be utilized to detect deadlocks caused by both locks and condition variables. It further presents an efficient dynamic tool, `UnHang`, that predicts deadlocks based on generalized dependency. `UnHang` also includes multiple practical designs to reduce the recording and prediction overhead. Based on our experiments, `UnHang` only imposes around 3% performance overhead and 8% memory overhead, while predicting a range of known deadlocks. It further detects **two new** deadlocks in widely-used real applications. `UnHang` has few false positives, which shares the same issues with most dynamic tools. `UnHang` can be an always-on option for detecting program deadlocks due to its low overhead and high effectiveness.

8 ACKNOWLEDGEMENTS

We are thankful to the anonymous reviewers for their constructive feedback that has helped us improve this paper. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paidup, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This material is also based upon work supported by the National Science Foundation under Award CCF-2024253, and UMass Start-up Package. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] ab developers. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.
[2] Drd, valgrind. http://valgrind.org/.
[3] How sqlite is tested. https://www.sqlite.org/testing.html.
[4] Mysql bugzilla. https://bugs.mysql.com/bug.php?id=68251.
[5] pthreads - posix threads. https://man7.org/linux/man-pages/man7/pthreads.7.html.
[6] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 51–60, 2006.
[7] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conference*, pages 208–223. Springer, 2005.

[8] C. Bienia and K. Li. PARSEC 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[9] Y. Cai and W. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 606–616. IEEE, 2012.

[10] Y. Cai and W.-K. Chan. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering*, 40(3):266–281, 2014.

[11] Y. Cai and Q. Lu. Dynamic testing for deadlocks via constraints. *IEEE Transactions on Software Engineering*, 42(9):825–842, 2016.

[12] Y. Cai, R. Meng, and J. Palsberg. Low-overhead deadlock prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1298–1309. IEEE, 2020.

[13] Y. Cai, H. Yun, J. Wang, L. Qiao, and J. Palsberg. Sound and efficient concurrency bug prediction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 255–267, 2021.

[14] A. Choudhary, S. Lu, and M. Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 266–277. IEEE, 2017.

[15] Z. Da Luo, R. Das, and Y. Qi. Multicore sdk: A practical and efficient deadlock detector for real-world applications. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 309–318. IEEE, 2011.

[16] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 480–491. IEEE, 2009.

[17] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[18] M. Eslamimehr and J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 353–365. ACM, 2014.

[19] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[20] P. d. C. Gomes, D. Gurov, M. Huisman, and C. Artho. Specification and verification of synchronization with condition variables. *Science of computer programming*, 163:174–189, 2018.

[21] J. Hamin and B. Jacobs. Deadlock-free monitors. In *European Symposium on Programming*, pages 415–441. Springer, 2018.

[22] D. Hovemeyer and W. Pugh. Finding concurrency bugs in Java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[23] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, pages 337–348, 2014.

[24] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336, 2010.

[25] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. *ACM Sigplan Notices*, 44(6):110–120, 2009.

[26] H. Jula, D. M. Tralamazza, C. Zamfir, G. Candea, et al. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, volume 8, pages 295–308, 2008.

[27] C. G. Kalhauge and J. Palsberg. Sound deadlock prediction. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[28] E. Kamburjan. Detecting deadlocks in formal system models with condition synchronization. *Electronic Communications of the EASST*, 76, 2019.

[29] C. Laneve and L. Padovani. Deadlock analysis of wait-notify coordination. In *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy*, pages 50–67. Springer, 2019.

[30] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *2009 IEEE 31st International Conference on Software Engineering*, pages 386–396. IEEE, 2009.

[31] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[32] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 727–737. IEEE, 2012.

[33] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 521–530, 2012.

[34] N. Redis. Memcache traffic generation and benchmarking tool, 2020.

[35] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 473–489, 2014.

[36] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 29–42, New York, NY, USA, 2014. ACM.

[37] V. K. Shanbhag. Deadlock-detection in java-library using static-analysis. In *2008 15th Asia-Pacific Software Engineering Conference*, pages 361–368. IEEE, 2008.

[38] F. Sorrentino. Picklock: A deadlock prediction approach under nested locking. In *International SPIN Workshop on Model Checking of Software*, pages 179–199. Springer, 2015.

[39] V. Terragni and M. Pezzè. Statically driven generation of concurrent tests for thread-safe classes. *Software Testing, Verification and Reliability*, 31(4):e1774, 2021.

[40] C. von Praun. *Detecting synchronization defects in multi-threaded object-oriented programs*. PhD thesis, Citeseer, 2004.

[41] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *European conference on object-oriented programming*, pages 602–629. Springer, 2005.

[42] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *OSDI*, volume 10, pages 163–176, 2010.

[43] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu. Undead: Detecting and preventing deadlocks in production software. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 729–740. IEEE, 2017.