UNDEAD: Detecting and Preventing Deadlocks in Production Software

Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai* and Tongping Liu
Department of Computer Science, University of Texas at San Antonio, USA
*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
Jinpeng.Zhou@utsa.edu, Sam.Silvestro@utsa.edu, liuhyscc@gmail.com, ycai.mail@gmail.com, Tongping.Liu@utsa.edu

Abstract—**Deadlocks are critical problems afflicting parallel applications, causing software to hang with no further progress. Existing detection tools suffer not only from significant recording performance overhead, but also from excessive memory and/or storage overhead. In addition, they may generate numerous false alarms. Subsequently, after problems have been reported, tremendous manual effort is required to confirm and fix these deadlocks.**

This paper designs a novel system, UNDEAD, that helps defeat deadlocks in production software. Different from existing detection tools, UNDEAD imposes negligible runtime performance overhead (less than 3% on average) and small memory overhead (around 6%), without any storage consumption. After detection, UNDEAD automatically strengthens erroneous programs to prevent future occurrences of both existing and potential deadlocks, which is similar to the existing work—Dimmunix. However, UNDEAD exceeds Dimmunix with several orders of magnitude lower performance overhead, while eliminating numerous false positives. Extremely low runtime and memory overhead, convenience, and automatic prevention make UNDEAD an always-on detection tool, and a "band-aid" prevention system for production software.

I. INTRODUCTION

The lock is possibly the most widely used synchronization primitive in multithreaded programs. It ensures that, at any time, at most one thread is allowed to execute inside a critical section. Common programming practice utilizes different locks to protect different shared data in order to avoid unnecessary conflicts, which may easily cause a notorious concurrency problem — deadlock.

Generally, deadlocks can be divided into two types: resource deadlocks and communication deadlocks [1]. This paper only deals with resource deadlocks that are caused by locks, without involving signals or other communication mechanisms (e.g., shared buffer). Deadlock occurs when a set of competing threads are simultaneously waiting for a lock held by their competitors, as shown in Figure 1. In this example, thread T_1 has acquired lock l_1 and waits for lock l_2. However, thread T_2 has acquired lock l_2 and waits for lock l_1. This situation prevents the program from making any further progress [1], [2]. Since deadlocks (as well as other concurrency bugs) only occur under specific inputs and with a precise timing requirement (e.g., two threads should acquire their first locks simultaneously), they cannot be completely expunged during development phases due to incomplete tests. They may be unavoidably leaked into deployed software, which may result in unpredictable losses, as programs must be restarted when deadlocks occur. Deadlocks continue to be a serious concern, accounting for more than 30% of reported concurrency bugs [3].

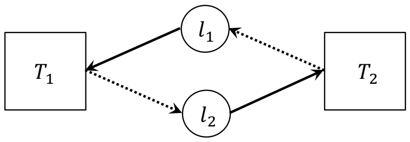

Fig. 1. A simple deadlock example.

A significant amount of work focuses on detecting deadlocks using static and dynamic approaches. Static detection tools, such as [4], [5], may report numerous false positives, and are difficult to scale to large programs [6]. Dynamic techniques primarily focus on detecting potential cycles within the execution trace [1], [7], [8], [9], [10]. Existing dynamic approaches have the following problems. *First*, existing systems record the full execution context, including call stacks and the order of all lock acquisitions and releases, which imposes significant overhead in the recording phase. For instance, the performance overhead of DeadlockFuzzer [1] is between 3× and 12×. *Second*, the execution trace can be extremely large for long-running applications (e.g., server applications), thus leading to unaffordable memory and/or storage overhead. *Third*, detecting potential cycles within a large execution trace may require a considerable amount of time, as long as two days [11]. Together, these shortcomings prevent existing detection techniques from gaining widespread adoption within deployed systems.

Some approaches aim to fix known deadlocks [12], [13], [14], [15], [16] by relying on the precise information of deadlocks that is difficult or impossible to obtain currently (due to the problems discussed above), and they may require additional manual effort (e.g., recompilation) before users can actually prevent problems. Also, many existing techniques may additionally generate new deadlocks through the introduction of gate locks, as discovered by DFixer authors [16].

Dimmunix pioneers the combination of detection and prevention in a single system, without involving additional manual effort [17]. For detection, Dimmunix employs a monitor thread to periodically check (100ms by default) whether recent locks generate a cycle of lock acquisitions. If so, Dimmunix stops a susceptible program immediately and outputs the signature of locks into a persistent file used to guide its prevention during future executions. To prevent deadlocks, Dimmunix obtains the call stacks (by invoking the very slow `backtrace` API) upon every lock acquisition, and confirms whether this acquisition matches one of the known deadlock patterns. If so, the current thread will be forced to wait. For every lock release, Dimmunix checks whether a potential deadlock has been broken. If so, it may wake other threads waiting for the release event of this lock. Dimmunix allows a deadlock-prone system to function correctly while developers are fixing these discovered bugs, which is a very interesting and helpful idea that could be potentially suitable for the deployment environment. However, Dimmunix suffers from a large number of false positives (a fact that is self-acknowledged [17]), incurs large runtime overhead (as high as $726\times$ slower, based on our evaluation), and could possibly miss some deadlocks, as it only checks recent locks. These serious limitations prevent Dimmunix from actually being deployed in production software.

This paper presents a novel system, UNDEAD, to defeat deadlocks in production software. It shares the same target as Dimmunix: it can detect deadlocks, and protect a program immediately and automatically whenever potential or actual deadlocks are found during prior executions. UNDEAD makes several key design choices to reduce overhead and improve accuracy of detection, and also proposes a novel, lock-variable-based approach to prevent deadlocks automatically. All of these details are further described in Section II.

A. Contributions

Overall, UNDEAD makes the following contributions:

1) UNDEAD *provides an efficient and effective detection tool:* UNDEAD's detection tool imposes only negligible performance overhead, less than 3% on average, and avoids the significant number of false positives. It only adds 6% memory overhead in total.
2) UNDEAD *proposes a novel, lock-variable-based approach to prevent deadlocks:* This novel approach supports deadlock prevention based on an incomplete execution trace: any call stacks related to these lock variables – even non-exercised call stacks – will be unable to cause the deadlock. UNDEAD's prevention can also be seamlessly integrated with its detection, without any manual intervention.
3) *We performed substantial evaluation on existing benchmarks and real applications, and compared it with existing work:* Evaluation results confirm the efficiency and effectiveness of UNDEAD.

B. Outline

The remainder of this paper is organized as follows. Section II first describes the background of deadlocks, as well as the basic idea of UNDEAD. Section III and Section IV describe the implementation of UNDEAD's detection and prevention mechanisms, respectively. Then, Section V presents experimental results. In the end, we discuss the related work in Section VI, and conclude in Section VII.

II. OVERVIEW

A. Background of Deadlocks

As described in Section I, deadlocks occur when the relationship between threads and locks generates a cycle. Deadlocks may be easily diagnosed by experts when they actually occur. Experts could attach a debugger to the hanging process, check the call stacks of lock acquisitions made by each thread, then possibly infer the exact causes of the deadlock. However, this discovery process requires expert knowledge and is not suitable for normal users in the production environment. More importantly, this discovery can only uncover actual deadlocks, while leaving behind potential deadlocks that may incur unpredictable losses in a deployed environment.

Based on our observation, deadlocks have the following **key properties**.

1) Only nested locks are capable of producing deadlocks, since a single-level lock will not be involved in a hold-and-wait situation, and thus will not generate a circle (as shown in Figure 1). In reality, the majority of lock acquisitions are not nested (without lock dependencies), but rather single-level locks. This fact is further confirmed by the statistics listed in Table I.
2) Deadlocks are very sensitive to timing, and only occur when all participating threads can successfully acquire their first locks prior to any thread subsequently requesting an additional lock. As illustrated in Figure 1, if one thread can first acquire both locks l_1 and l_2 successfully — which may cause the competing thread to fail to acquire its first lock — the deadlock will not occur.

B. Deadlock Detection

UNDEAD's detection aims to detect deadlocks for use by its prevention system, as well as for normal users, while keeping low performance and memory/storage overhead. UNDEAD makes several design choices to achieve this target.

We make the first key observation: *a program typically only has a bounded number of unique lock dependencies, although a program may execute for a long time*. This key observation is further confirmed via extensive experiments, where the results are shown as Table I. The lock dependency is first introduced by prior work DeadlockFuzzer [1], as discussed in more detail in Section III-C. When a program has n distinct locks, the maximum number of unique lock dependencies consisting of two locks will be equal to the permutation $P(n, 2)$. In reality, the actual number is even lower than this maximum number. Although the bounded property may not hold if a program dynamically creates and destroys locks, we did not find such cases in our experiments.

Firstly, based on this key observation, UNDEAD only records unique lock dependencies (under nested locks) for every thread during the execution, instead of recording the entire history of lock acquisitions and releases. It will skip all duplicated lock dependencies. Note that this will not compromise detection effectiveness, as all distinct lock dependencies (with their unique call stacks) are still recorded completely, which can achieve the same guarantee as existing work. This method prevents an ever-increasing trace log for long-running applications; instead, the size of the execution trace becomes stable after seconds or minutes of execution. A small execution trace greatly reduces not only the memory/storage requirement, but also the amount of time required for the detection.

Secondly, UNDEAD also avoids unnecessary information during the recording phase. Single-level locks are never recorded because they will never cause potential deadlocks, as Property 1 discussed in Section II-A. UNDEAD also discards lock acquisition events when only a single thread exists, since deadlocks typically involve more than one thread.

Thirdly, UNDEAD minimizes the overhead of collecting call stacks. UNDEAD utilizes the following two methods to reduce the overhead. (1) UNDEAD does not fetch the call stacks of first-level locks. This method reduces a large number of `backtrace` invocations, but creates some inconvenience. Although UNDEAD reports the address of locks involved in deadlocks, as well as where threads are hanging, it cannot report statements where the first-level locks were acquired. Users are required to manually identify the information of first-level locks for the purpose of detection. However, this problem will not affect UNDEAD's prevention. (2) UNDEAD further avoids invocations of `backtrace`, when both stack offset and lock address of a lock acquisition are the same as existing ones. Stack offset is utilized here, since different threads may possess the same stack offset for the same call stack. It is possible (although unlikely) that two different call stacks will have the same stack offset, where users may have to identify correct call stacks manually. Again, this shortcoming only affects the reporting functionality, and will not reduce the effectiveness of UNDEAD's detection and prevention.

C. Deadlock Prevention

UNDEAD starts from a novel observation concerning the prevention of deadlocks: *replacing multiple locks (within the same deadlock set) with the same lock eliminates the possibility of deadlock* (shown in Figure 4). Replacing multiple locks in the same deadlock set mimics the effect of using a coarse-grained lock, rather than using multiple fine-grained locks. This method eliminates the timing property of deadlocks. In the example shown in Figure 1, when two locks (l_1 and l_2) are actually using the same lock l_{12}, the deadlock will never occur. If one thread acquires lock l_{12} successfully, it will prevent another thread from acquiring the same lock due to the exclusivity property of mutex locks.

This method is significantly different from signature-based approaches [14], [16], [17], [18]. For signature-based approaches, they require call stacks of all lock acquisitions that are involved in a deadlock, so that they can explicitly locate and fix buggy acquisitions to avoid possible deadlocks. They cannot prevent deadlocks if a single statement is neglected to be changed. Compared with these approaches, UNDEAD supports *prevention based on an incomplete execution trace*: as long as some call stacks of suspect lock variables are detected to have potential deadlocks, any call stacks consisting of these variables – even call stacks not encountered in previous detection – will be unable to cause the deadlock. UNDEAD replaces all of these locks with the same physical lock during the initialization phase, and prevents any possible deadlock afterwards. Also, UNDEAD requires no change to the program and no re-compilation, while the prevention can be enabled automatically during future executions. UNDEAD also has one advantage when compared to Dimmunix, a signature-based approach requiring no additional manual effort. UNDEAD significantly reduces the prevention overhead, since there is no need to check the signature on every lock acquisition and release.

Performance Concern: This naturally raises a concern with respect to the performance impact: that is, whether merging multiple locks into the same lock will significantly increase lock contention. In practice, lock contention may not be substantially increased due to the following two reasons. First, deployed software has been extensively tested, and typically will not contain a large number of deadlocks inside. This fact will limit the number of distinct locks that should be merged. Second, all locks belonging to the same deadlock set are less likely to be acquired frequently, since deadlocks caused by frequently-acquired locks are most likely to have been eliminated during development phases. Evaluation results in Section V confirm that UNDEAD only introduces around 3% performance overhead, which is comparable to the state-of-the-art—DFixer [16].

```
   Thread 1 :                      Thread 2 :
1:     Lock l1 ;                       Lock l3 ;
2:     // Intermediate lock            Lock l2 ;
3:     Lock l3 ;
4:     Lock l2 ;
```

Fig. 2. A potential new deadlock due to the lock merge.

```
   Thread 1 :                      Thread 2 :
1:     Lock l1 ;                       Lock l1 ;
2:     Lock l2 ;                       Unlock l1 ;
3:     Cond_wait(cond, l2) ;           Lock l2 ;
4:     Unlock l2 ;                     Cond_signal (cond) ;
5:     Unlock l1 ;                     Unlock l2 ;
```

Fig. 3. Atomicity violation due to the lock merge.

Correctness Concern: We have identified two possible correctness issues related to lock merging, shown as Figure 2 and Figure 3. Figure 2 shows an example of a potential new deadlock. In this example, the acquisition of lock l_3 is located between l_1 and l_2 in Thread 1, and Thread 2 will acquire l_3 before acquiring l_2. When l_1 and l_2 are merged into a new lock l_{12}, it may create a potential deadlock caused by the lock set of $\{l_{12}, l_3\}$. Figure 3 shows another example in which the merging of l_1 and l_2 may cause an atomicity violation. Thread 1 should release the merged lock l_{12} at the invocation point of `Condwait` (line 3 of Thread 1), and can possibly violate the original atomicity semantics since other threads may be able to acquire the lock l_1 in the middle. However, invoking a conditional wait while holding two locks is actually a well-known deadlock pattern reported by existing work [8], [19].

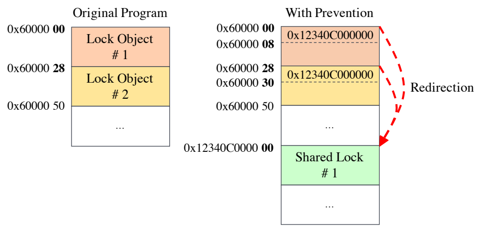

Fig. 4. The idea of deadlock prevention.

In Figure 3, Thread 2 may never acquire l_1 successfully, and would subsequently be unable to wake up Thread 1.

UNDEAD provides *the following guarantees for these two correctness concerns*: (1) All locks (e.g., l_3 of Figure 2) occurring in the middle of a known deadlock set (e.g., $\{l_1, l_2\}$), and which may potentially cause deadlocks, will be merged into the same deadlock set. Thus, the situation listed as Figure 2 will never create new deadlocks, if such a lock acquisition (e.g., l_3) has been observed during previous executions. (2) For cases such as Figure 3, UNDEAD only reports the deadlocks to users, but never performs automatic prevention. Thus, it does not introduce potential atomicity violations. All conditional waits inside nested locks are recorded to avoid this problem.

III. DEADLOCK DETECTION

For deadlock detection, UNDEAD first records necessary lock-related events during executions (Section III-A). Then, it can detect actual deadlocks periodically (Section III-B), or detect actual and **potential** deadlocks on program exit, or upon receiving instruction from users (Section III-C).

A. Logging Phase

UNDEAD logs only necessary lock-related information. It prunes all unnecessary information during the logging phase, including single-level locks, unnecessary lock acquisitions during the single-threaded period, and duplicated lock dependencies.

UNDEAD intercepts mutex lock operations, including locks, unlocks, and `try locks`. For `try locks`, UNDEAD only records successful acquisitions. UNDEAD also records the conditional waits inside nested locks in order to avoid the problems listed in Figure 3. Additionally, UNDEAD obtains call stacks of mutex initializations, as this information will be employed by its prevention system to accomplish deadlock prevention (Section IV).

UNDEAD collects the call stacks of lock acquisitions using `backtrace` [20], but handles it differently from all existing work. UNDEAD maintains the number of locks held by each thread separately. It does not obtain the call stacks of first-level locks, but rather simply records the addresses of these locks. When a thread is acquiring more than one lock, UNDEAD first confirms whether the corresponding lock dependency already exists. If the lock dependency exists, UNDEAD further determines whether it is necessary to obtain the call stack for the purpose of reporting. Currently, UNDEAD confirms whether both stack offset and lock address (inside the lock dependency) of a lock acquisition are the same as existing ones, since acquiring the call stack using `backtrace` can be very expensive. If both of these values and the lock dependency are the same, UNDEAD does not obtain the call stack for this lock acquisition.

The combination of the stack offset and the lock address is generally an adequate indicator of the call stack, as the chance of sharing the same stack offset and lock address, but with a different call stack, is extremely low. The only situation with confusion is when there are multiple acquisitions of the same lock within the same function. For this case, users may be required to check all lock acquisitions inside the same function manually, if they would like to identify which acquisition is involved in the deadlock. Note that UNDEAD's prevention mechanism does not rely on the call stacks of acquisitions, and requires no manual effort. The reason why UNDEAD uses the stack offset is that multiple threads inside a multithreaded program have the same offset, but different absolute addresses. If the call stack is new, UNDEAD additionally records it. The call stacks of lock acquisitions could be utilized by existing prevention tools [14], [16], [17], [18], or used to facilitate manual fixes.

In the implementation, all lock dependencies associated with each thread are stored in a per-thread hash table for performance reasons. This method may introduce slightly more memory overhead used to hold unique lock dependencies, but avoids concurrent accesses from multiple threads. Also, it eliminates the necessity of using locks to protect the actions of searching and adding new lock dependencies. To save lock dependencies, UNDEAD utilizes the combination of two locks as the key for the hash table. For instance, if lock l_1 is dependent on locks $\{l_2, l_3\}$, UNDEAD utilizes the XOR value of the addresses of the two most internal locks (e.g., l_1 and l_2 here) as the search key. This method substantially reduces conflicts on hash keys, and thus considerably reduces the number of comparisons required to check the duplicates of a lock dependency.

UNDEAD also improves the performance of obtaining per-thread data. During development, we discovered significant performance overhead associated with retrieving per-thread data upon every intercepted function invocation, if we simply utilized the thread-local storage area (declared using the keyword "`__thread`") [21]. When using thread-local storage, the system assigns a unique global ID to each thread-local variable, and maintains a Thread Local Storage (TLS) lookup table for all threads. Searching the thread-local area involves at least the cost of an external library call, as well as a lookup in the indexed table. Thus, UNDEAD *proposes a novel method to circumvent the cost of using TLS variables*. Because every thread will be allotted its own stack, the address of a stack variable can be utilized to identify a particular thread. Thus, UNDEAD initially allocates a global map, which will be used to create the stack areas for every thread. Upon creation of each thread, UNDEAD assigns a specific stack area to the thread based on its internal thread index, such as 8MB per thread by default. During execution, UNDEAD is able to compute the thread index by dividing the offset between the address of a stack variable and the starting address of the global stack map with the stack size. With the computed thread index, UNDEAD can efficiently obtain the per-thread data from a shared global array.

B. Periodic Detection

UNDEAD creates a monitor thread to perform detection periodically during normal executions, which is similar to existing work [22]. This option allows UNDEAD to timely detect deadlocks in long-running applications.

Every thread maintains its own current locking set. The monitor thread is awakened periodically. Upon waking up, the monitor thread takes a snapshot of the current locking set for each thread, and utilizes the simple loop detection (as shown in Figure 1) to check whether there is a cycle inside. If a cycle is found, the monitor thread will confirm again whether the current lock-holding status of related threads has been changed. The same status indicates that the program is actually in deadlock status, and cannot proceed. Otherwise, it is a potential deadlock that may not actually occur. UNDEAD allows a program with a potential deadlock to proceed as normal. Note that UNDEAD does not change the original lock logic for periodic detection, which simplifies implementation and possibly benefits performance. Instead, UNDEAD utilizes another confirmation to avoid any possible false positives caused by using the snapshots. If a deadlock is actually detected, UNDEAD further performs comprehensive detection to identify any other potential deadlocks, as discussed in Section III-C.

C. Comprehensive Detection

UNDEAD performs comprehensive detection when the program is about to exit, or when it receives special instructions from users. As discussed above, UNDEAD only keeps unique lock dependencies, and prunes all unnecessary information. This reduces UNDEAD's detection time due to handling many fewer items.

1) Intercepting Exits and User Instructions: A program may terminate in various ways. Aside from exiting normally, a program may exit by explicitly calling `abort`, or due to failed assertions or other program errors such as segmentation faults. UNDEAD handles these different exit patterns. To intercept normal exits, UNDEAD places the detection procedure within a function marked with the `destructor` attribute. Furthermore, UNDEAD also registers a signal handler for signals such as SIGHUP, SIGQUIT, SIGINT, and SIGSEGV, from which it invokes `exit()`, such that the default exit handler will then be invoked. Additionally, users may send a special signal using the SIGUSR2 signal number to invoke the detection procedure on-demand.

2) Detection Algorithm: UNDEAD utilizes the known deadlock detection method, iGoodLock, which was proposed by DeadlockFuzzer [1] and extended by MagicFuzzer [9]. However, the idea of pruning unnecessary information is novel to this paper, which is the key reason that UNDEAD keeps a low performance and memory/storage overhead, and is different from these existing works [1], [9].

iGoodLock defines the following basic concepts:

- "Lock dependency": A triple $D = \{t, l, L\}$ indicates that a thread t is waiting for lock l while holding all locks in set L.
- "Lock dependency chain": A sequence of lock dependencies, $C = \langle D_1, ..., D_k \rangle$, describes a relationship among the lock dependencies of multiple threads: (1) each lock dependency inside C is from a distinct thread. (2) $\forall i \neq j, L_i \bigcap L_j = \varnothing$, which means no locks are held in common by any distinct pair of threads. (3) $\forall i \in [1, k-1], l_i \in L_{i+1}$, which indicates the $(i+1)^{\text{th}}$ thread has already acquired the lock l_i, which the i^{th} thread is trying to acquire.
- "Cyclic lock dependency chain": A cyclic chain is a special type of lock dependency chain, where $l_k \in L_1$ in order to form a circle.

As described above, each thread records its own lock dependencies within its own per-thread hash table, which eliminates the necessity of including the thread information from the lock dependency. For detection, UNDEAD searches for lock dependency chains by checking all possible permutations of lock dependencies among two threads, then three threads, and so on. UNDEAD utilizes a depth-first algorithm to iteratively explore dependency chains. If there exist lock dependency chains, UNDEAD further checks for cycles within these dependency chains. A cyclic dependency chain indicates a potential deadlock problem, which UNDEAD will report in the end.

3) Reporting Deadlocks: UNDEAD generates two different reports: one for normal users or signature-based prevention systems, the other for its prevention system.

For the first report, UNDEAD reports all information necessary to assist deadlock fixes, such as the call stack (with source code line information) of initialization and acquisitions of each involved lock, not including first-level locks, and the address of all involved locks. Programmers can proceed to fix these discovered deadlocks based on UNDEAD's report.

For its prevention system, the report currently takes the format of a text file, in which every deadlock set is separated by a special symbol. When multiple deadlock sets exist, UNDEAD further confirms whether or not they share any common locks. If true, UNDEAD merges various lock sets containing the common lock into the same deadlock set, which will utilize the same physical lock in prevention. For the example of Figure 2, l_3 will be merged into the same deadlock set of $\{l_1, l_2\}$. Inside the same deadlock set, each entry on a separate line will represent a different lock, the format for which appears as such: *lock address: initialization call stack*. If a lock is a global variable that is initialized via the `PTHREAD_MUTEX_INITIALIZER` macro, its initialization call stack will be empty. Thus, UNDEAD's prevention system can handle them accordingly. Depending on the preferences of the user, UNDEAD may skip a deadlock set that contains a conditional wait while holding multiple locks.

D. Limitations

UnDead may experience false negatives. Like other dynamic tools, UnDead cannot detect deadlocks if the corresponding code is not exercised. Currently, UnDead cannot detect resource deadlocks caused by conditional variables. Similar to existing work [1], [7], [8], [9], [10], UnDead does not recognize other types of locks, such as spin locks or read/write locks, which will be a focus of our future work.

In theory, UnDead can produce false positives as well. When two lock sets have a strong happens-before relationship which prevents their concurrent appearance, such as those ordered by conditional waits or barriers, UnDead may incorrectly generate false alarms. However, experimental results in Section V-D confirm that UnDead is very effective at capturing potential deadlocks in reality, without any false alarms.

IV. Deadlock Prevention

This section describes how UnDead's prevention system works automatically, based on its detection results.

UnDead's detection system reports potential deadlocks, where different lock sets are separated using a special symbol in the deadlock history file. As discussed in Section II-C, all locks within the same deadlock set are redirected into the same physical lock, as shown in Figure 4, which eliminates the deadlock problem.

For the example shown in Figure 1, if lock l_1 and lock l_2 are replaced with a new lock l_{12}, then T_1's acquisition of l_1 (now l_{12}) will prevent T_2's acquisition of l_2, due to their uses of the same lock l_{12}. Therefore, it is impossible to generate a circle, preventing the deadlock caused by lock l_1 and lock l_2. As shown in Figure 4, the new shared lock l_{12} will be allocated from an easily identifiable memory-mapped region, such as a continuous region starting from $0x12340C000000$. UnDead will store the address of l_{12} into the first word of both l_1 and l_2. Originally, this word includes two fields, __lock (the lock state) and __count (the number of recursive locks), which should not fall into the special range discussed above. Thus, when the first word of a lock falls within the special range, it is a pointer to a shared lock. UnDead then forces the lock acquisition onto the shared lock. This redirection mechanism is adapted from our existing work—SyncPerf [23].

The detailed implementation is discussed as follows.

A. Implementation

1) Initialization: One type of lock variables are initialized via the `PTHREAD_MUTEX_INITIALIZER` macro, while another type are initialized by explicitly invoking a mutex initialization procedure. Locks of the first type have no call stacks inside the deadlock history file, and these locks can only be global variables. Therefore, they will be redirected before entering the application's main routine. UnDead sets its initialization function with the `constructor` attribute in order to perform this initialization before entering into the main routine. For the other type of locks, UnDead intercepts the initializations of all locks, and confirms the call stack of these initialization invocations. When an initialization has the same call stack as those listed in the deadlock history file, UnDead redirects it appropriately, as shown in Figure 4.

2) Redirection: During prevention, UnDead intercepts every lock acquisition and release, as well as conditional waits. It checks whether the lock's first word falls within the special range. If so, UnDead redirects the operation to the shared lock. Otherwise, it simply performs the operation on the original `pthread_mutex_t` object. At every lock acquisition and release, the overhead imposed by UnDead is limited to a fetching operation and a checking operation, which is significantly lower than that of Dimmunix.

3) Handling Recursive and Irregular Locks: UnDead also handles recursive and irregular locks carefully. If a lock has been successfully acquired by the current thread, another invocation on the same lock will fail. Therefore, a simple and naive solution is to change the shared lock to be a recursive one, by setting the `PTHREAD_MUTEX_RECURSIVE` attribute. However, this simple solution cannot handle a program with an irregular lock pattern, as shown in Figure 5. There is no problem for this example initially (without the redirection mechanism), since releasing an already-released lock, such as l_2 in Figure 5, will be silently turned into a no-operation. However, if a recursive lock is utilized to replace these two locks, such as l_1 and l_2 in the example, the code (after line 4 of Figure 5) will not be protected by any lock, which violates the original semantics of the program.

```
1:    Lock l1 ;
2:    Lock l2 ;
3:    Unlock l2 ;
4:    Unlock l2 ; //Irregular lock pattern
5:    Unlock l1 ;
```

Fig. 5. Irregular lock pattern.

UnDead handles this problem by tracking the locking count of every specific lock (with its original address) in the deadlock sets. For every shared lock, each thread maintains a per-thread lock count, and an array of lock counts that track the status of all original lock variables inside the same deadlock set. The actual shared lock will only be acquired at the first acquisition, while subsequent acquisitions only increment the lock count of its corresponding original lock variable. This also helps report some potential problems of original lock uses. If an original lock variable is acquired multiple times, which indicates a double-lock problem in the original program, UnDead can report this problem. During a release, UnDead first confirms whether the local lock count is larger than 1. If true, it decrements the local and per-thread lock counts at the same time, and only invokes the actual release operation when the per-thread lock count is 0. Otherwise, it simply omits this operation and returns 0. This method matches the original semantics.

B. Limitations

UnDead's prevention is only enabled when evidence of potential deadlocks is detected with a cyclic lock dependency chain. Note that UNDEAD's prevention does not require a deadlock to occur.

Performance Concern: UNDEAD's prevention can increase the lock contention rate, thus affecting the performance of applications by merging multiple locks into a single lock. However, it only requires simple engineering effort to track the contention rate of merged locks, and disable them if the rate is higher than a specified percentage. The other candidate is to request confirmation from users; UNDEAD's prevention may skip a deadlock set based on the input of users.

Correctness Concern: UNDEAD provides the guarantees as discussed in Section II-C. In extremely rare situations, UNDEAD may add potential new deadlocks, or violate atomicity, when the critical information is not observed during prior executions. However, the deadlock and atomicity violation may not actually occur during executions, since they are both sensitive to timing. Also, upon the end of an execution, UNDEAD will adjust its deadlock sets based on new evidence, and further prevent the potential problems in future executions.

V. EVALUATION

A. Experimental Setup

We performed experiments on a 16-core quiescent machine, with two sockets installed with Intel(R) Xeon(R) CPU E5-2640 processors. This machine has 256GB of main memory, and 256KB L1, 2MB L2 and 20MB L3 cache. The experiments were performed on the unchanged Ubuntu 16.04, installed with Linux-4.4.25 kernel. We used GCC-4.9.1 with `-O2` and `-g` flags to compile all applications.

B. Performance Overhead

For the performance overhead, we compared UNDEAD with the only available work with similar functionality — Dimmunix [17]. The source code of Dimmunix was downloaded from Google Code [24]. Because Dimmunix halts a program immediately after detecting potential deadlocks inside, it often cannot complete applications normally due to many false positives. For performance comparison, we allowed Dimmunix to run until program completion by disabling the halting mechanism.

We evaluated 17 multithreaded applications in total. Among these applications, ten of them are from the PARSEC suite [25], while others are real applications, including `MySQL-5.6.10`, `Memcached-1.4.25`, `Apache httpd-2.4.25`, `SQLite-3.12.0`, `Aget`, `Pfscan`, and `Pbzip2`. PARSEC applications were exercised using native inputs, with 16 threads [26]. `MySQL` was tested using the `sysbench` application, with 16 threads and $100,000$ maximum requests. `Memcached` was tested using the `python-memcached` script [27], but changed to loop 20 times in order to obtain sufficient runtime. `SQLite` was tested using a program called "threadtest3.c" [28] and the total amount of iterations is shown in the figure. `Apache` was tested by sending $100,000$ requests via `ab` [29] and the requests per second is shown in the figure. For `Aget`, we collected the execution time of downloading 600MB of data

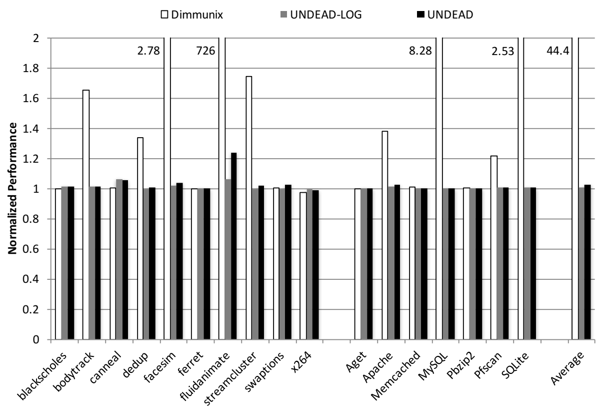

Fig. 6. Normalized runtime overhead of Dimmunix and UNDEAD. "UNDEAD" lists the overhead with both detection and prevention, which is comparable to Dimmunix's overhead. "UNDEAD-LOG" shows the runtime overhead of UNDEAD's logging mechanism only.

from another quiescent server located on the local network. For `Pfscan`, we performed a keyword search within 800MB of data. For `Pbzip2`, we performed compression on a file containing 150MB of data.

The average results of ten executions of these applications are shown in Figure 6, where all values of UNDEAD and Dimmunix are normalized to that of the default `Pthreads` library. A taller bar indicates a larger overhead.

UNDEAD has two sets of performance data in Figure 6: "UNDEAD-LOG" is a special build with logging only, where both detection and prevention mechanisms are disabled in order to confirm the efficiency of UNDEAD's logging strategy; "UNDEAD" represents the full version with all functionalities enabled (logging, detection, and prevention). Because Dimmunix combines logging with its detection and prevention, we only show one set of data. "Dimmunix" should be compared to "UNDEAD" in Figure 6, with the same functionality.

As shown in Figure 6, UNDEAD's logging only introduces around 1% performance overhead on average. Overall, UNDEAD introduces around 2.7% overhead on average with all mechanisms enabled. For all applications, except `fluidanimate`, UNDEAD's performance overhead is less than 6%. In contrast, Dimmunix imposes more than $44\times$ performance overhead on average. Without `fluidanimate`, Dimmunix's average overhead is still more than 170%.

We further collected the lock characteristics of these applications, which helps explain the performance problems of both UNDEAD and Dimmunix. The data is shown as Table I. The "Time" column shows the total execution time of each application. The "Locks" column shows the number of distinct lock objects, and the "Lock Acqs" column shows the number of lock acquisitions. "Total" and "Unique" indicate the amounts of all and unique lock dependencies, respectively.

UNDEAD imposes around 23% performance overhead for the `fluidanimate` application, which is still orders of magnitude lower than Dimmunix's overhead ($726\times$). The data shows that this application issues approximately 1.7 billion lock acquisitions within 30 seconds' execution, and contains

TABLE I: CHARACTERISTICS OF EVALUATED APPLICATIONS

Applications	Time (s)	Locks (#)	Lock Acqs (#)	Dependencies (#)	
				Total	Unique
blackscholes	37.6	0	0	0	0
bodytrack	26.8	7	1858970	0	0
canneal	55.9	1	15	0	0
dedup	16.6	2199	999468	0	0
facesim	72.7	17	5244475	407092	120
ferret	4.6	5	2594	0	0
fluidanimate	30	92396	1723264500	0	0
streamcluster	62.6	2	1274992	0	0
swaptions	20.1	0	0	0	0
x264	66.9	35	202776	0	0
Aget	5.5	2	147110	0	0
Apache	23.6	105	1454437	0	0
Memcached	4.6	30	50427	18208	53
MySQL-5.6.10	44.6	651	20049715	887	113
Pbzip2	1.47	20	2350	340	3
Pfscan	1.16	2	36	0	0
SQLite	20.2	6	5022776	35	2

TABLE II: MEMORY CONSUMPTION (MB)

Applications	Default	Dimmunix	UNDEAD
blackscholes	612	613	634
bodytrack	33	45	52
canneal	944	943	960
dedup	1643	1630	1654
facesim	324	393	343
ferret	66	69	80
fluidanimate	408	696	426
streamcluster	110	121	155
swaptions	9	9	24
x264	485	487	499
Aget	6	9	21
Apache	4	300	17
Memcached	5	7	15
MySQL-5.6.10	123	143	156
Pbzip2	95	105	113
Pfscan	813	799	838
SQLite	39	41	57
Total	5719	6410	6044

92,396 different lock objects. UNDEAD should collect the call stacks for each distinct lock initialization, and also record the address of every lock acquisition. This explains the significant performance overhead for this application. However, we expect that normal applications should not have such extensive lock acquisitions, such as `MySQL` and `Memcached` here. Thus, UNDEAD's performance overhead should be totally acceptable.

Table I also helps explain the performance overhead of Dimmunix as well. To our understanding, Dimmunix imposes high overhead because it must obtain call stacks of all lock acquisitions. For applications with a significant number of lock acquisitions, such as `fluidanimate`, `MySQL`, `facesim`, and `SQLite`, Dimmunix imposes significant performance overhead.

Table I also confirms several design choices employed by UNDEAD. (1) The number of single-level lock acquisitions is much larger than the number of nested locks. For instance, `fluidanimate` has more than 1.7 billion lock acquisitions, but with zero nested locks. Similarly, `MySQL`'s quantity of single-level lock acquisitions is 22,603 times more than the number of nested locks. Thus, avoiding the overhead of obtaining call stacks for first-level locks can greatly reduce logging overhead. (2) The total number of lock dependencies is much larger than the number of unique dependencies. `facesim` has around 407,902 lock dependencies, but with only 120 that are unique. Thus, recording only the unique lock dependencies can largely reduce the detection time and memory/disk consumption. The mechanism of recording all lock acquisitions employed by existing work is not suitable for long-running applications, as the ever-expanding trace will consume too much memory/disk capacity, and significantly increases the detection time.

C. Memory Overhead

The physical memory overhead from using `Pthreads`, Dimmunix, and UNDEAD is shown in Table II. For small-footprint applications, such as `aget` and `swaptions`, UNDEAD may add a high percentage of startup memory overhead introduced by its recording mechanism, due to utilizing some pre-allocated mechanisms inside. However, if we sum the memory usage across all applications, UNDEAD imposes only 6% memory overhead when compared to the default `Pthreads` library, which is similar to Dimmunix's memory overhead.

We further performed another experiment to verify whether the memory overhead introduced by UNDEAD is stable across different sizes of input. We verified the memory consumption of `MySQL-5.6.10` server, and showed the results in Figure 7. The memory consumption of UNDEAD remains almost the same across varying numbers of requests, where the number of unique lock dependencies has a similar trend as well.

D. Detection Effectiveness

This section confirms the detection effectiveness of both UNDEAD and Dimmunix: false negatives and false positives. We evaluated all applications in Section V-B, as well as real-world deadlock problems frequently utilized by previous work [16], [17]. The bug in `MySQL-6.0.4` is a synthetic bug used in existing work [30]. Effectiveness results are shown in Table III. Those applications without false positives and false

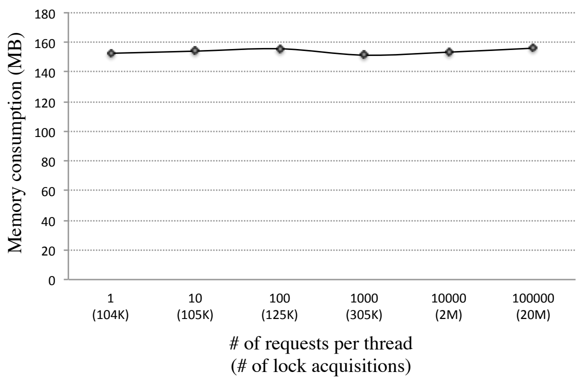

Fig. 7. Memory consumption of UNDEAD, when `sysbench` utilizes different numbers of requests and different numbers of lock acquisitions to exercise the `MySQL-5.6.10` server.

TABLE III: Comparison of detection effectiveness

Applications	# of Cycles		Real Deadlocks (#)
	Dimmunix	UnDead	
bodytrack	2	0	0
dedup	2	0	0
facesim	1	0	0
ferret	3	0	0
fluidanimate	1	0	0
streamcluster	5	0	0
HawkNL-1.6b3	1	1	1
MySQL-5.6.10	1	2	2
MySQL-6.0.4	2	1	1
pbzip2	1	0	0
SQLite-3.3.3	1	2	2

negatives for both UnDead and Dimmunix are not shown in this table.

In these experiments, UnDead reports no false positives, and identifies all known deadlocks. Thus, UnDead is very effective at detecting deadlocks, although it cannot completely avoid false positives and false negatives in theory, as discussed in Section III-D.

In contrast, Dimmunix reports 16 false alarms on evaluated applications, and misses an actual deadlock in MySQL-5.6.10. When an application has more than one deadlock, Dimmunix may only detect one deadlock and then kill the program, inducing false negatives. Dimmunix may report different results on different executions, as its monitor thread may check deadlocks at random points. False positives of Dimmunix are due to its implementation fault, based on our understanding of their source code. Dimmunix accidentally has two different states for the same lock in its Resource Allocation Graph [17], which mimics the effect of two continuous lock acquisitions of the same lock from the same thread. Dimmunix treats this behavior as an irregular lock pattern that can cause a program to hang, so it reports this to users. This explains why Dimmunix has many false positives, even when a program (e.g. `fluidanimate`) only contains single-level locks.

E. Prevention Effectiveness

We evaluated the effectiveness of UnDead's prevention system for applications containing known deadlocks, by comparing with Dimmunix and the state-of-the-art (DFixer [16]). For all of these buggy applications, including `HawkNL-1.6b3`, `SQLite-3.3.3`, and `MySQL-5.6.10`, these three prevention systems can successfully prevent those deadlocks.

As shown in Table IV, the "Merged Locks" column lists the number of locks that are actually merged by UnDead. "Real" indicates the number of real locks contained in the deadlock set, while the "Add" column lists the number of locks that should be additionally merged in order to avoid new deadlocks. The "Conflict Acqs" column lists the number of lock conflicts during the execution, with `Pthreads` ("Orig") and UnDead, using buggy inputs. UnDead adds one additional lock into the deadlock set for `HawkNL-1.6b3` and `MySQL-5.6.10` to avoid the problem as shown in Figure 2.

F. Prevention Performance

Figure 6 shows that UnDead's prevention mechanism imposes little overhead for programs without deadlocks, since UnDead's overhead with prevention is similar to the one ("UNDEAD-LOG") without prevention.

The performance results with deadlocks are shown in Table IV. When unnecessary invocations of `sleep` inside these programs were removed, these executions finished very quickly. For example, `HawkNL` and `SQLite` only execute for less than 0.002 seconds, which makes the performance comparison meaninglessly. However, both UnDead and DFixer perform better than Dimmunix, but worse than the original executions (marked as "Orig"). UnDead may introduce some startup overhead: UnDead should read the bug report file, redirect locks, and check whether a lock operation should be performed on a shared lock. The number of lock conflicts can be slightly increased with the prevention system of UnDead.

We further confirmed the prevention overhead on normal executions, which is more important than small inputs. We utilized `sysbench` to exercise both `MySQL-5.6.10` and `MySQL-6.0.4` with 16 threads and $100,000$ maximum requests. As shown in Table V: (1) both UnDead and DFixer significantly outperform Dimmunix. The overhead of both UnDead and DFixer is less than 1%, while Dimmunix runs up to 25× slower. For `MySQL-6.0.4`, Dimmunix encounters many false positives caused by the levels of call stacks, and its performance deteriorates significantly; (2) UnDead achieves similar prevention performance as DFixer, despite its additional workload such as call stacks fetching and lock redirection. The comparison between DFixer and UnDead is further discussed in Section VI.

VI. Related Work

A. Detection and Prediction

There are two types of dynamic tools used for detecting deadlocks. One type is very efficient [22], [31], comparable to UnDead, but which cannot report potential deadlocks. For instance, Pulse [22] periodically checks whether the system has been blocked, by checking whether the resource graph contains a possible cycle.

The other type of detection tools focus on detecting potential deadlocks that do not occur in the execution. Many works aim to predict cycles [1], [4], [5], [7], [9], however, these works typically incur significant performance overhead, as high as multiple orders of magnitude [1], [8], [9], [10], [32]. For example, Sherlock [10] introduces more than 6× overhead compared to that of DeadlockFuzzer, while DeadlockFuzzer itself operates with up to 12× overhead. Wolf [32] reduces its recording overhead over DeadlockFuzzer by about 20%. In contrast, UnDead exhibits much lower recording overhead than these tools, a result of its careful design. Also, UnDead imposes low memory overhead by pruning all unnecessary information during execution.

There exist works that also attempt to utilize scheduling to trigger deadlock occurrences [1], [2], [8], [9], [10], [33], [34]. They typically utilize dynamic analysis to identify potential deadlocks at first, then direct a scheduler to reproduce these potential deadlocks with high probability. However, these approaches reduce convenience by adding an additional

TABLE IV: CHARACTERISTICS OF UNDEAD'S PREVENTION

Applications	Bug ID	Time (s)				Merged Locks(#)		Conflict Acqs(#)	
		Orig	UNDEAD	Dimmunix	DFixer	Real	Add	Orig	UNDEAD
HawkNL-1.6b3	N/A	0.002	0.006	0.007	0.004	2	1	≤ 2	≤ 1
SQLite-3.3.3	1672	0.001	0.005	0.006	0.004	2	0	≤ 1	≤ 1
MySQL-5.6.10	62614	0.015	0.015	0.031	0.015	3	1	≤ 6	≤ 7

TABLE V: NORMALIZED PERFORMANCE AND LOCK CONFLICTS UNDER NORMAL INPUTS

Applications	Bug ID	Normalized Slowdown(%)			Normalized Conflicts(%)	
		UNDEAD	Dimmunix	DFixer	Orig	UNDEAD
MySQL-5.6.10	62614	0.6%	762%	0.4%	100%	109%
MySQL-6.0.4	37080	0.9%	2556%	0.8%	100%	107%

confirmation stage. Armus detects barrier deadlocks on a set of synchronization models [34]. STEPDAD helps reproduce the deadlocks inside database problems [35]. The lock confirmation can serve as a great supplement to UNDEAD, though with a different focus.

ConTeGe [36] and Omen [37] generate concurrent test cases toward triggering deadlocks and other concurrency bugs. They could possibly integrate with UNDEAD's detection to find additional deadlock problems.

B. Prevention, Fixing, and Recovery

Some previous work prevents deadlock occurrences by patching programs, relying on precise signatures of deadlocks that are difficult or impossible to obtain due to the lack of efficient detection tools. Gadara [38] synthesizes the Discrete Control Theory (at source code level) to avoid deadlocks by inserting some gate locks during compilation, then dynamically chooses to enforce them at runtime. However, as observed by DFixer, Gadara can also introduce potential deadlocks, and it reports a maximum of 18% performance overhead. Lock capabilities can statically verify a deadlock problem, then may be used to prevent deadlock occurrences [39]. Various other works also take similar approaches, such as AFix [12], CFix [13], Axis [14], Grail [15], and [40], which can introduce potential deadlocks as well.

Sammati proposes an online prevention system that is not based on detection results [41]. It can checkpoint the state of the program upon lock acquisitions, and monitor the possible cycles during execution. If a deadlock is found, the program is rolled back to a previous checkpoint, where the program is re-executed to avoid the susceptible schedule [41]. However, Sammati may impose more than 100% performance overhead for some applications.

Similar to Dimmunix, REDACT [42] utilizes a supervisor controller to prevent deadlocks in database applications. Differently, it uses static analysis to find hold-and-wait cycles in transactions, and feeds the analysis results to the controller. Afterwards, it utilizes a similar mechanism as that of Dimmunix to prevent the deadlocks. It delays a transaction if it is involved in a hold-and-wait cycle.

DFixer [16] is the previous state-of-the-art in the deadlock prevention. It proposes a lock pre-acquisition idea that enables one thread involved in a deadlock to acquire all locks in the same deadlock set at one time. This eliminates the hold-and-wait condition necessary for a deadlock to occur, which is similar to UNDEAD's idea. DFixer also proposes context-aware conditionals to avoid the introduce of new communication deadlocks, which may avoid problems as shown in Figure 3. DFixer has the following significant differences from UNDEAD: (1) DFixer requires an additional recompilation, which cannot work on legacy applications without the source code available. In contrast, UNDEAD directly works on commercial off-the-shelf (COTS) binaries without recompilations. (2) DFixer requires precise inputs on deadlocks, where it may not avoid deadlocks caused by non-instrumented statements. UNDEAD proposes a variable-based mechanism that can prevent all deadlocks related to known lock variables, preventing deadlocks based on incomplete trace. (3) DFixer may require the special handling for complexed situations, such as multiple deadlocks caused by the same set of locks, or irregular lock patterns. On the contrary, UNDEAD's mechanisms can handle complex situations easily, but with possible performance compromise.

VII. CONCLUSION

This paper introduces UNDEAD, a novel system that can efficiently detect and prevent deadlocks in production software. As a detection tool, UNDEAD only imposes 3% runtime overhead and 6% memory overhead. Evaluation results show that UNDEAD detects all known deadlocks, while reporting no false positives. UNDEAD also proposes a novel idea of "merging locks" to prevent deadlock problems temporarily, providing a "band-aid" for production software. UNDEAD's prevention system can automatically read detection results, and strengthen buggy programs by preventing potential deadlocks in future executions. UNDEAD is available at https://github.com/UTSASRG/UnDead.

ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their valuable suggestions and feedback that helped improve this paper. The work is supported by UTSA startup package, Google Faculty Award, and the National Science Foundation under Grants No. 1566154. The work is also supported by National Natural Science Foundation of China (NSFC) (grant No. 61502465), National 973 program of China (2014CB340702), and Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (2017151).

References

[1] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 110–120. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542489

[2] Y. Cai, S. Wu, and W. K. Chan, "ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 491–502. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568312

[3] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 329–339. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346323

[4] 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*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/512529.512558

[5] 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*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 237–252. [Online]. Available: http://doi.acm.org/10.1145/945445.945468

[6] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *Proceedings of the 19th European Conference on Object-Oriented Programming*, ser. ECOOP'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 602–629. [Online]. Available: http://dx.doi.org/10.1007/11531142_26

[7] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*. London, UK, UK: Springer-Verlag, 2000, pp. 245–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=645880.672085

[8] 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*. ACM, 2010, pp. 327–336.

[9] Y. Cai and W. K. Chan, "Magicfuzzer: Scalable deadlock detection for large-scale applications," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 606–616. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337294

[10] 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*. ACM, 2014, pp. 353–365.

[11] Z. D. 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*, March 2011, pp. 309–318.

[12] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 389–400. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993544

[13] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 221–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387902

[14] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware fixing of concurrency bugs," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635881

[15] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 299–309. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337259

[16] Y. Cai and L. Cao, "Fixing deadlocks via lock pre-acquisitions," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1109–1120. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884819

[17] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock immunity: Enabling systems to defend against deadlocks," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 295–308.

[18] Y. Nir-Buchbinder, R. Tzoref, and S. Ur, "Deadlocks: From exhibiting to healing," in *Runtime Verification*, M. Leucker, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 104–118. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89247-2_7

[19] 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.

[20] GLIBC Developers, "Backtraces - glibc," http://www.gnu.org/software/libc/manual/html_node/Backtraces.html#Backtraces.

[21] D. Gross, "TLS performance overhead and cost on gnu/linux," http://david-grs.github.io/tls_performance_overhead_cost_linux, 2016.

[22] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin, "Pulse: A dynamic deadlock detection mechanism using speculative execution," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247363

[23] M. M. u. Alam, T. Liu, G. Zeng, and A. Muzahid, "Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: ACM, 2017, pp. 298–313. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064186

[24] Authors of Dimmunix, "Dimmunix: Deadlock immunity system for Java/C/C++ software," https://code.google.com/archive/p/dimmunix, 2016.

[25] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[26] 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.

[27] "Pure python memcached client," https://pypi.python.org/pypi/python-memcached.

[28] SQL Developers., "How sqlite is tested," https://www.sqlite.org/testing.html.

[29] ab Developers., "ab - apache http server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html.

[30] T. W. Zhen Yu, Xiaohong Su and P. Ma, "Mocklinter: Linting mutual exclusive deadlocks with lock allocation graphs," *International Journal of Hybrid Information Technology*, vol. 09, no. 3, pp. 355–374, 2016.

[31] C. Zamfir and G. Candea, "Low-overhead bug fingerprinting for fast debugging," in *International Conference on Runtime Verification*. Springer, 2010, pp. 460–468.

[32] 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*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 29–42. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555262

[33] S. Bensalem and K. Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, ser. HVC'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 208–223. [Online]. Available: http://dx.doi.org/10.1007/11678779_15

[34] T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida, "Dynamic deadlock verification for general barrier synchronisation," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015, pp. 150–160.

[35] M. Grechanik, B. M. Hossain, and U. Buy, "Testing database-centric applications for causes of database deadlocks," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 174–183.

[36] 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, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 521–530. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254126

[37] 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*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 473–489. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660238

[38] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke, "Gadara: Dynamic deadlock avoidance for multithreaded programs." in *OSDI*, vol. 8, 2008, pp. 281–294.

[39] C. S. Gordon, M. D. Ernst, and D. Grossman, "Static lock capabilities for deadlock freedom," in *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. ACM, 2012, pp. 67–78.

[40] Y. Lin and S. S. Kulkarni, "Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 237–247. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610398

[41] H. K. Pyla and S. Varadarajan, "Avoiding deadlock avoidance," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 75–86.

[42] M. Grechanik, B. Hossain, U. Buy, and H. Wang, "Preventing database deadlocks in applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 356–366.