International Journal of Hybrid Information Technology
Vol.9, No.3 (2016), pp. 355-374
http://dx.doi.org/10.14257/ijhit.2016.9.3.34

Mocklinter: Linting Mutual Exclusive Deadlocks with Lock Allocation Graphs

Zhen Yu[1, a], Xiaohong Su[1, b], Tiantian Wang[1, c] and Peijun Ma[1, d]

[1]*School of Computer Science and Technology, Harbin Institute of Technology, China*

[a]*yuzhen_3301@aliyun.com*, [b]*sxh@hit.edu.cn, sweetwtt@126.com*, [d]*ma@hit.edu.cn*

Abstract

Deadlocks are serious runtime bugs and are difficult to expose, reproduce and diagnose. Once suffering from them, programs may be afflicted with increasing response time, decreasing throughputs, or even crashes. We present Mocklinter, a dynamic deadlock detection tool to capture a deadlock as soon as it happens and spit out enough information to support source-level debugging. Mocklinter tracks the synchronization state of the target program by dynamically constructing and maintaining a lock allocation graph. Mocklinter uses this graph to decide whether a deadlock is confronted or not. Mocklinter handles all types of pthread mutexes and can detect any number of deadlocks at a time. Each deadlock captured by Mocklinter can involve any number of threads. We implemented Mocklinter in Linux-3.2.0 and evaluated it with ten applications, including Dining-Philosophers, Sshfs, SQLite, OpenLDAP, MySQL and so on, whose sizes varies from 0.1K to 1021.0K in terms of LOC. The results demonstrate effectiveness against real or artificial deadlock bugs, while incurring modest performance overhead and scaling to more than one thousand of threads.

Keywords*: Dynamic analysis, Software testing, Mutual exclusive locks, Cycle detection, Deadlock detection, Deadlock debug*

1. Introduction

Writing and debugging concurrent programs are difficult because of inherent concurrency and non-determinism. Humans are good at handling tasks one by one, however, concurrency requires programmers to think in a parallel way. Scutter and Larus observe [1]: "*Humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people may miss possible interleavings among simple collections of partially ordered operations*". Lots of concurrency bugs happen due to poor coordination between threads. Many programmers consider concurrency bugs to be some of the most insidious because they are hard to expose, detect and debug. One concurrency bug may or may not manifest itself even across executions with the same input.

Among various types of concurrent bugs, deadlock is one of the most common and important [3-5]. Deadlock may occur whenever multiple threads interact. Two or more threads are deadlocked when each of them is waiting for a resource, typically a lock, which has been acquired and is being held by another thread. Deadlock is a potential problem in all multithreaded programs. Once suffer from it, programs may be afflicted with increasing response time, decreasing throughputs, or even crashes. Timely detection of deadlock and its cause is essential for resolving the error and maintaining forward progress.

Pthread and its synchronization infrastructures are prevalently used in writing concurrent programs under various OS circumstances. For example, pthread mutexes are often used to protect series of operations that should be executed exclusively and

ISSN: 1738-9968 IJHIT

atomically. However, without proper handling, mutexes may introduce deadlock bugs into programs. Figure 1 is an example illustrating a deadlock bug [6] caused by mutexes in SQLite, a widely used embedded database engine. The bug occurs when thread T1 first acquires mutex1 at L1 and tries to acquire mutex2 at L2, and then thread T2, who has acquired mutex2 already, tries to acquire mutex1 at L3. Figure 2 shows a test case that can trigger this bug.

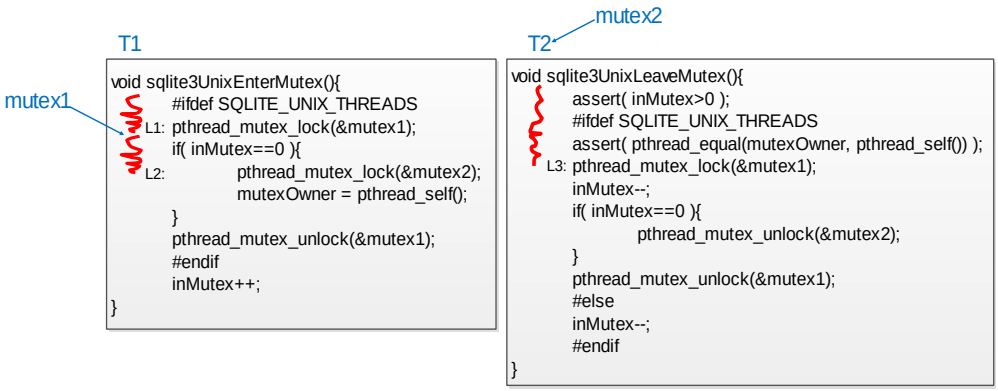


Figure 1. The Deadlock Bug#1672 in SQLite-3.3.3 and its Manifestation

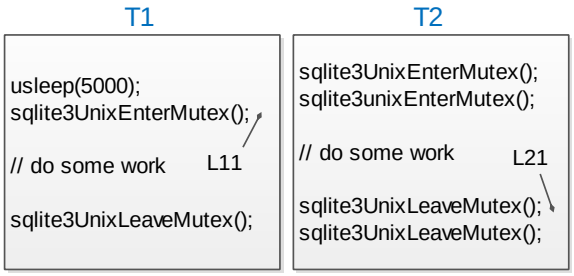


Figure 2. A Trigger for the Bug#1672 in SQLite-3.3.3

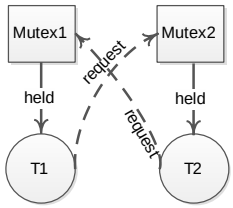


Figure 4. The LAG for the Trigger in Figure 2

During in-house testing, if a program hangs for a long time, testers don't know whether it is deadlocked or just is processing a time-consuming task. Some mechanism is needed so that whenever a deadlock is encountered, testers are informed of that immediately, additionally with enough information to debug and repair it at the source level. This is exactly what Mocklinter has done. Mocklinter works in two phases. In the first phase, it monitors the target program for every lock/unlock operation on mutexes and record these information in a lock allocation graph (LAG), which is a simplified version of RAG [7]. LAG is dynamically constructed and maintained in order to depict a scenario that which mutexes are being requested or held by which threads. In the second phase, Mocklinter uses a novel cycle detection algorithm, named ticketed-DFS, to detect whether a cycle exists in the LAG or not. If so, it will terminate the target program and emit thread IDs and mutex IDs involved in the cycle. The target program is terminated by Mocklinter with the signal SIGSEGV, which will cause kernel to dump a core file for it. With the help of gdb and information output by Mocklinter, testers can quickly locate the root cause of the reported deadlock at the source level if the target program is compiled with debugging flags.

We contribute in the following aspects: (1) dynamically detecting deadlocks caused by mutexes of all the four types; (2) capturing any number of deadlocks at a time; (3) proposing a new cycle detection algorithm; (4) supporting source-level debug.

In the rest of the paper, we survey the related work (Section 2), provide an overview of Mocklinter (Section 3), gives details of our techniques (Section4 and Section 5), evaluate it (Section 6), discuss its limitations (Section 7) and conclude (Section 8).

2. Related Work

Recently, much effort has been made to help detect and avoid deadlock bugs. In terms of techniques adopted, they form a spectrum from purely static approaches to purely dynamic ones.

At the beginning of the spectrum are verification and model checking approaches [8-10], which try to verify whether a program is deadlock free or not. If deadlock exist, they provide full counterexample from initial states to deadlock states. Compositional logic is often used by them to cope with the state explosion problem. However, programs in real world are too complicate for them to build models for verifying or checking. Now, they are only applicable to toy programs.

Type system approaches, such as ownership [11] and lock capability [12], which stem from programming language design field, aim for static deadlock freedom guarantees by imposing a strict (non-cyclic) lock acquisition order that must be respected throughout the entire program. However, most such approaches won't scale to large programs, neither work if no annotations are provided.

Effects approaches [13, 14] statically compute the effect of lock/unlock operations and dynamically decide whether a lock operation should be entered according to its effect. A lock operation is permitted to execute if the locks relating to it are all available. Effects approaches are fast and need no annotations. However, they won't work if no source code is given. Gadara [15, 16] transforms programs to/from Petri nets to synthesize additional locks that are able to avoid deadlocks. Gadara is poor at scalability, and is highly sensitive to pointer aliasing. In the worst case, it may insert so many locks that the concurrency degree of the original programs is decreased.

Dataflow approaches [5, 17, 18] usually use a combination of various static analysis techniques, such as call-graph analysis, pointer-alias analysis, thread-escape analysis, to compute a static lock order graph [2, 19] and report cycles in it as possible deadlocks. For example, based on these techniques, Jade [5] reports possible deadlocks involving two threads/locks while Williams [17] can report deadlocks involving more than two threads/locks, and can handle reentrant locks as well. RacerX [18] performs flow-sensitive inter-procedural analysis to find deadlocks and rank them in decreasing order according to likelihood. For lack of precise runtime information, these approaches all suffer from high false positives and consequently require amounts of time and energy of the user to manual confirmation.

At the end of the spectrum are the dynamic approaches. DeadlockFuzzer [4] uses iGoodlock algorithm to discover potential deadlock cycles in a normal execution of a multithreaded program, and then, in the next execution try to push the program into a real deadlock state corresponding to the previously reported cycle. MagicFuzzer [20], TeamWork [31] and MagicLock [32] improve iGoodlock by proposing the Magiclock algorithm to dramatically reduce the size of the lock order graph. These techniques may report lots of false positives for lacking of happen-before relationship. For example, in the Jigsaw benchmark, iGoodlock reports 283 potential deadlocks and only 29 of them are confirmed as real deadlocks by DeadlockFuzzer. Dimmunix [21, 22] helps a program develop immunity against future occurrences of a given deadlock once the program suffers from the deadlock. Dimmunix can only handle deadlocks caused by mutexes of the normal type, and only detect one deadlock at a time. Besides, it can't give any source level information about how to repair the detected deadlock. Glock [23] is similar to Dimmunix, except that it exhibits and heals deadlocks in the same execution. Chess [24] introduces preemptions at synchronization points in order to amplify the contention for

synchronization resources. This helps expose deadlocks with high probability. Pulse [25] can detect multiple types of deadlocks through speculative execution of "dead" processes. However, it couldn't achieve this without modifying the kernel code, which is difficult and impractical. ConTeGe [26] tries to find deadlock bugs in Java thread-safe libraries by calling to different synchronization methods of a given class's object from multiple threads. It only reveals deadlock bugs located in the same class and has limited support for detecting inter-class bugs.

Mocklinter hits the dynamic end of the spectrum. It tries to capture dynamic deadlocks caused by pthread mutexes and emits enough information to support debugging at the source level. Mocklinter can detect any number of deadlocks, rather than only one, at a time. Each deadlock captured by Mocklinter can involve any number of threads and mutexes, rather than only two. With respect to one execution, Mocklinter is sound and complete to detect deadlocks caused by pthread mutexes. Additionally, Mocklinter achieves this without needing any annotations or source code modification.

3. Mocklinter Overview

We present a dynamic analysis to detect deadlocks caused by pthread mutexes. This section gives the key ideas of the analysis. Section 4 and Section 5 fill in the details.

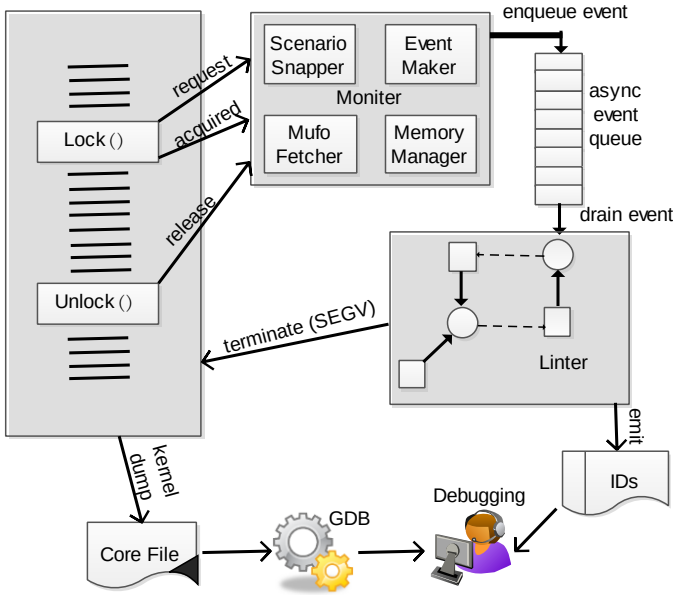


Figure 3. Mocklinter Architecture

There are four requirements for detecting deadlocks with a dynamic analysis. First, whenever an interesting deadlock occurs, the analysis should capture it. Second, the analysis should capture the deadlock as soon as possible. Third, it should never reports false warnings, that is, reports a deadlock that doesn't happen in reality. Fourth, the analysis should impose modest impact on the original program. Previous work has no complete solutions to all of these requirements. Dimmunix [21] can't capture the self-deadlock which occurs when a thread tries to acquire a normal type mutex already held by it. In addition, if more than one deadlock happens simultaneously, Dimmunix can only capture one of them. Pulse [25] may miss deadlocks or report false alarms. Helgrind [27, 28] slowdowns the original programs in a factor of similar to or more than 22.2X.

Mocklinter satisfies all the above four requirements. We give the evaluation of Mocklinter on these requirements in Section 6. Figure 3 illustrates its architecture. There are mainly two parts: Monitor and Linter. Monitor has four components: Scenario

Snapper, Mufo Fetcher, Event Maker and Memory Manager. Scenario Snapper hijacks every lock or unlock operation on mutexes, i.e., snaps the scenario that some thread is about to request, acquire, or release some mutex. During hijack, Scenario Snapper calls Mufo Fetcher, standing for Mutex Information Fetcher, to determine two things: what the type of a mutex is (normal, recursive, errorcheck, or default), and whether or not a mutex is currently held by a given thread. After hijack, Scenario Snapper calls Event Maker to make events and en-queue events into a lock-free queue that is drained by Linter. Memory Manager is responsible for memory allocation and de-allocation for all activities happening in Monitor.

Linter wakes up periodically to maintain a LAG according to drained events and search for deadlock cycles in the LAG. The delay between the occurrence of a deadlock and its detection has an upper bound determined by the wakeup frequency. If Linter detects cycles in the LAG, it emits thread and mutex IDs involved in the cycles, terminates the program with the signal SIGSEGV which causes kernel to dump a core file for it. The user then can load the core file with gdb to debug the root cause.

We now show how Mocklinter works with the test case shown in Figure 2. When T1 blocks at L11 and T2 blocks at L21, a deadlock happens. Mocklinter will construct a LAG, something like what shown in Figure 4, to represent this deadlock. The LAG is a directed graph with two types of vertices: threads, shown as circles, and mutexes, shown as squares. There are two types of edges connecting threads to mutexes: request edges and held edges. Request edges indicate that a thread T wants to acquire mutex M, however, has not already acquired it. Held edges indicate that thread T has already acquired and presently holds mutex M. Once a cycle appears in the LAG, Linter will detect it during its next wakeup interval.

Mocklinter can be used by software testers during in-house testing to detect deadlocks in general purpose systems, such as desktop applications, server software, and so on. Coupled with some deadlock exposing methods, like stress testing, schedule-based ConTest [29], preemption-based Chess [24], Mocklinter can help testers quickly locate and remove deadlocks.

4. Monitor

In this paper, we give details about hijack algorithms and some implementation issues related to them.

4.1. Operations Monitored

Given an execution of a multithreaded program P, we use *t* to identify a thread in the execution. There are two kinds of critical operations that Mocklinter needs to monitor:

- *lock*(*t*, *m*): thread *t* tries to acquire mutex *m*;
- *unlock*(*t*, *m*): thread *t* realeases mutex *m*.

Pthread defines three lock operations on mutexes, including *lock*, *trylock*, *timedlock*, and one unlock operation on mutexes, *i.e.*, *unlock*. Mocklinter monitors all these four operations, differentiating from Dimmunix [21], which only monitors the *lock* and *unlock* opertions.

Mutexes defined in pthread have four types: normal, recursive, errorcheck and default. For a given mutex *m* in an operation, say *lock*, Mocklinter needs to know what the type of it is. However, in pthread, there are functions to set the type for a mutex, but no functions to get the type of a mutex. Once a mutex is set as a given type, there is no way to reset its type, and no way to know what its type is. In order to circumvent this limitation, we refer to the internal structure definition of a mutex and retrieve the value of the fourth field as its type. This method works in Linux. However, it may not work in other unix-like OSes or Windows. We suggest that pthread, in its next version, should encompass functions responsible to set/get types for an initialized mutex. Thanks to our customized type

retriever, Mocklinter can handle mutexes of all four types, again, differentiating from Dimmunix, which handles normal mutexes only.

4.2. Hijack Algorithms

Mocklinter dynamically instruments the code of the original program, so that all lock and unlock operations are intercepted. We name the hijacked operations as "natives", and the hijacking operations as "wrappers" to the hijacked ones. A wrapper replaces a call to a native with wrapping codes. According to the type of a mutex being handled, the wrapping codes determine two things: whether or not to relay events to the Linter module and whether or not to call the native operations. The hijack algorithms for lock and unlock operations are shown in Figure 5 and Figure 6, respectively. In the two algorithms, "gettype" refers to the type retriever described in Section 4.1.

```
Algorithm: lock_wrapper
Input: native lock
Output: wrapped lock
1.  t := current thread;
2.  m := current mutex;
3.  type := gettype(m);
4.  if t currently holds m
5.      if type is RECURSIVE
6.          m.lock_count++;
7.          return SUCCEED;
8.      else if type is ERRORCHECK
9.          return native_lock(m);
10. enqueue event (t, m, REQUEST) to t.eq;
11. ret := native_lock(m);
12. if ret is SUCCEED
13.     if type is RECURSIVE
14.         m.lock_count++;
15.     insert m to t.held_locks;
16.     enqueue event (t, m, ACQUIRED) to
    t.eq;
17. return ret;
```

Figure 5. Hijack Algorithm for the Native Lock Operation

```
Algorithm: Ticketed-DFS
Input: lock allocation graph lag, cycle
container cycles, initially empty
Output: whether cycles exist and what
they are
1.  ticket := a boolean variable;
2.  cycle := a set of edges;
3.  foreach vertex v in lag
4.      v.color = WHITE;
5.  foreach vertex v in lag //actually, in
    reqthrs
6.      if v.color == WHITE
7.          set cycle empty;
8.          ticket = FALSE;
9.          if visit(lag, v, cycle, ticket) is true
10.             add cycle into cycles if cycles
    hasn't contained it;
11. return !(cycles.empty());
```

Figure 6. Hijack Algorithm for the Native Unlock Operation

Within a lock wrapper (see Figure 5), if thread *t* currently doesn't hold mutex *m*, we en-queue a REQUEST event which indicates *t* is trying to acquire *m*, to *t*'s private event queue (line 10). Then we call the native lock on *m* and check whether it succeeds or not. If it succeeds, we know that *t* has successfully acquired *m*, so we mark that *m* is held by *t* and en-queue an ACQUIRED event to *t*'s event queue. In addition, if *m*'s type is RECURSIVE, we increase its lock count (from zero) by one (lines 12-16). If thread *t* currently holds *m*, we act according to *m*'s type (lines 4-10):

- If *type* is RECURSIVE, we increase *m*'s lock count and do nothing else but to return SUCCEED to indicate that the original program's lock operation succeeds (lines 5-7). We neither call the native lock, nor en-queue a REQUEST event. This makes sense because if a thread tries to acquire a recursive mutex which is currently held by it, the native lock will return immediately with success. Moreover, we want to construct a simple LAG, meaning that there are no multiple edges between two vertices. So no matter how many lock operations executed on a recursive mutex, we only en-queue one REQUEST event.

- If *type* is ERRORCHECK, a proper error code should be returned to the original program's lock operation to indicate that the operation fails. We simply call the native lock on *m* and return its result (lines 8-9).
- If *type* is NORMAL or DEFAULT, this means *t* is about to trap into a self-deadlock state. We en-queue a REQUEST event to *t*'s event queue (line 10), so that Linter will detect this self-deadlock during its next wake-up interval.

Within an unlock wrapper (see Figure 6), if *t* tries to release *m* which is not held by it, we directly call the native unlock on *m* and return its result (line 15). This is sensible because *t* may have acquired *m* by calling functions related to conditional variables (for example, pthread_cond_wait), which are transparent to Mocklinter. If *t* tries to release a recursive mutex, before doing anything else, we decrease its lock count. Only if the lock count is decreased to zero will we en-queue a RELEASE event and call to the native lock operation (lines 6-9). Otherwise, we just simply return SUCCEED to make the original program's unlock operation happy (lines 10-11). Of course, if *m*'s type is not RECURSIVE, as expected, we will do two things: en-queue a RELEASE event and call the native unlock on *m* (lines 12-14).

Hijack algorithms for *trylock* and *timedlock* are similar to the algorithm shown in Figure 5, we omit them for briefness.

5. Linter

In this section, we present how Mocklinter constructs and maintains the LAG using drained events (Section 5.1); detect cycles using our ticketed-DFS algorithms (Section 5.2).

5.1. Maintaining LAG

In our current implementations, Linter is embodied as a thread residing in the address space of the target program. For minimizing interference with the original program, Linter periodically wakes up and falls into sleep with a period τ (for example, 1 second or 0.1 second). The value of τ doesn't affect correctness, because it merely introduces a delay from the time the target program becomes deadlocked to the time this condition is detected. In practice, we use τ as a knob to tune the trade-off between computation overhead and detection speed. A higher τ reduces the CPU time consumed on updating the LAG and detecting cycles, while a lower τ leads to faster detection.

Once waking up, Linter drains events from the event queue of each thread that performs lock/unlock operations on mutexes during Linter's sleep interval. For a given drained event, Linter updates the LAG according to the type of the event. Figure 7 shows the main loop for processing events, maintaining LAG, and detecting cycles (line 18). The cycle detection part will be explored in detail soon later.

```
Algorithm: main_loop
Input: lock or unlock events
Output: updated LAG and whether cycles exist
1.  lag := the global lock allocation graph
2.  reqthrs := the global set for recording threads which request but have not acquired
3.  for each thread t who performs lock/unlock operations during Linter's sleep
4.    num := length of t.eq (at this moment)
5.    while num != 0
6.      num--;
7.      event := dequeue(t.eq);
8.      switch(event.type)
9.        case REQUEST:
10.         add a REQUEST edge from event.t to event.m in lag;
11.         add t to reqthrs;
12.       case ACQUIRED:
13.         remove the REQUEST edge from event.t to event.m in lag;
14.         add a HELD edge from event.m to event.t in lag;
15.         erase t from reqthrs;
16.       case RELEASE:
17.         remove the HELD edge from event.m to event.t in lag;
18. check_cycles(lag, reqthrs);
```

Figure 7. Linter's Main Loop Algorithm

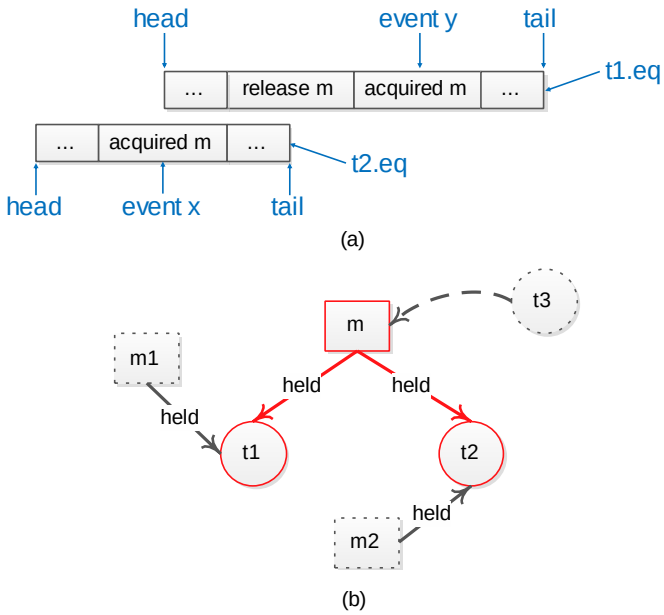


Figure 8. A Possible Scenario in *Lag*

For a given thread *t*, Linter drains its events of *num*, which is obtained when Linter is about to drain events from *t.eq* (line 4). The event queue of *t*, *t.eq*, may be accessed simultaneously by thread *t* and thread Linter. When Linter is draining an event from the tail of the queue, thread *t* may be adding an event to the head of it. No matter whether thread *t* adds events to *t.eq* or not, Linter just drains *num* events. The rest events (if any) will be drained in Linter's next wake-up interval. In order to obtain high efficiency, *t.eq* is implemented as a lock-free queue [2], meaning that there is underlying measures to synchronize accesses to it. This relieves the requirement to protect *t.eq* with explicit synchronizations.

```
Algorithm: Ticketed-DFS
Input: lock allocation graph lag, cycle container cycles, initially empty
Output: whether cycles exist and what they are
12. ticket := a boolean variable;
13. cycle := a set of edges;
14. foreach vertex v in lag
15.     v.color = WHITE;
16. foreach vertex v in lag //actually, in reqthrs
17.     if v.color == WHITE
18.         set cycle empty;
19.         ticket = FALSE;
20.         if visit(lag, v, cycle, ticket) is true
21.             add cycle into cycles if cycles hasn't contained it;
22. return !(cycles.empty());
```

```
Procedure: visit
Input: lag, v, cycle, ticket, whose values are all passed by reference
Output: whether a cycle exists and what it is
2.  v.color = GREY;
3.  foreach edge (v, u) in lag
4.      if u.color == GREY
5.          ticket = TRUE;
6.          u.color = RED;
7.          put (v, u) into cycle;
8.          return TRUE;
9.      else if u.color == WHITE
10.         if visit(lag, u, cycle, ticket) is TRUE
11.             if ticket == TRUE
12.                 put (v, u) into cycle;
13.             if v.color == RED
14.                 ticket = FALSE;
15.             return TRUE;
16. v.color = BLACK;
17. return FALSE;
```

Figure 9. The Ticketed-DFS Algorithm

As stated before, Mocklinter uses a lock allocation graph *lag* to represent the synchronization state of the target program (line 1). In pthread, a mutex can be held and only be held by one thread. This requires that when Linter is about to detect cycles in *lag*, any vertex of mutex type must have no more than one outgoing edge. However, during the time Linter updates *lag*, this point is not necessary to be guaranteed. For example, as shown in Figure 8(a), if Linter first processes events in *t*2.*eq* and ends with "event *x*", then switches to process events in *t*1.*eq* and drains "event *y*", it will construct a LAG *lag*, something like what shown in Figure 8(b), in which a vertex of mutex type has two outgoing edges. Fortunately, this scenario exists temporarily. Linter will continue to process RELEASE events, so the extra held edges will be removed finally before it detects cycles in *lag* (line 18).

5.2. Detecting Cycles

To find cycles in *lag*, we propose a depth first research algorithm named ticketed-DFS (as shown in Figure 9), which is a modified version of colored-DFS [30]. Owning to the ticketed-DFS algorithm, not only can we determine whether a cycle exists, which is what

colored-DFS only does, but also we can tell: how many cycles exists totally, what the vertexes and edges are for each cycle.

The ticketed-DFS algorithm (see Figure 9), initially marks all vertices with WHITE. When a vertex is visited at the first time, it is marked with GREY. When all of its descendants are completely visited, it is marked with BLACK. If a GREY vertex is encountered for the second time, then there is a cycle. We reset the GREY vertex with RED and backtrack from this vertex. In order to specify what a cycle is when it is detected, we use *ticket* to control that which edges should be collected while returning back recursively. When starting to backtrack, *ticket* is set as TRUE. During backtracking, *ticket* is set as FALSE once a RED vertex is seen. An edge is collected if and only if *ticket* is TRUE. This is just like the case you need a ticket when you go home by bus.

In Figure 9, we find cycles from every vertex in *lag*. Actually, it is not necessary. Because only REQUEST events can introduce new cycles into *lag*, we look for cycles that containing requesting vertexes (line 18 in Figure 7). The requesting vertexes represent the threads that pend for requesting a mutex. We achieve this by replacing *lag* with *reqthrs* in line 5 of Ticketed-DFS algorithm in Figure 9.

Now we analyze the theoretical complexity of ticketed-DFS. The minimal operation of ticketed-DFS is to determine whether *cycle* exists in *cycles* or not, that is, whether *cycle* is equal to some element in *cycles*. For *lag*[*V, E*], we determine in two steps whether two cycles are equal or not. Firstly, for each cycle, we sort the vertexes in it using the red-black-tree algorithm. This needs $O(\log|V|)$ complexity for each cycle, because a cycle involves $|V|$ vertexes at most. Secondly, we compare two ordered arrays. This needs $O(|V|)$ complexity. If *cycles* now contains N_C cycles, the complexity for checking whether *cycles* encompasses *cycle* is $O(N_C \cdot (2\log|V|+|V|))$. In addition, the complexity for detecting cycle is $O(|V|+|E|)$. So we can conclude that the complexity of ticketed-DFS is $O(N_T \cdot ((|V|+|E|)+N_C \cdot (2\log|V|+|V|)))$, where N_T is the number of threads in *reqthrs*.

6. Evaluation

6.1. Implementation and Applications

We have implemented Mocklinter as a preload library on Linux-3.2.0 for C/C++ programs written with pthread library. For each thread or mutex, Mocklinter maintains a shadow memory location to store its data, such as a set *held_locks* for a thread, or an integer *lock_count* for a mutex.

Our work aims at dynamically capturing all deadlocks caused by mutexes effectively and efficiently. To empirically evaluate whether Mocklinter has achieved this goal, we use a total of ten applications that have been widely used in previous bug detection and avoidance researches [13-15, 16, 20-22], as shown in Table 1. The suite of applications, the triggers and the source code of Mocklinter are available at http://sse.hit.edu.cn/yz/index.php/mocklinter.

The ten applications contain ten deadlock bugs, which can be divided into four groups: the born, the inserted, the real and the modified. The first group deadlocks are generated intrinsically because of wrong solutions to the bank transaction and dining philosopher problems. The second group deadlocks are inserted manually by us into deadlock-free applications, such as Ngorca, Sshfs-fuse and Tgrep. The third group contains deadlocks that originally exist in applications. The last group contains deadlocks that are modified from the original deadlocks whose causes are not mutexes only. For example, the original deadlock bug#3494 in OpenLDAP-2.2.20 is caused by the case that two threads in different orders try to acquire a customized read/write lock and a mutex, as illustrated in Figure 10. The deadlock happens, when a thread blocks at line 555 and another thread blocks at line 880. In order to simulate this deadlock with mutexes, we add a lock operation on a mutex following the lines 555 and 225

respectively. We also add an unlock operation on the same mutex following the lines 559 and 882 respectively. The bug#37080 and bug#38804 in MySQL are modified in a similar way.

Table 1. Evaluated Applications and Deadlock Bugs (NA Means Unknown)

Application	LOC(K)	Bug ID	Bug Group	Bug Type		
				DN	RECUR	MN
Bank Transactions	0.1	NA	Born	1	Yes	2
Dining Philosophers	0.1	NA	Born	1	No	5
Ngorca-1.0.2	1.3	NA	Inserted	1	No	2
Tgrep	1.7	NA	Inserted	1	No	2
Sshfs-fuse-2.2	3.8	NA	Inserted	3	No	6
SQLite-3.3.3	50.7	1672	Real	1	No	2
HawkNL-1.6b3	8.7	NA	Real	1	No	2
OpenLDAP-2.2.20	149.4	3494	Modified	1	No	2
MySQL-6.0.4-alpha	1021.0	37080	Modified	1	No	2
MySQL-5.1.24-rc	924.1	38804	Modified	1	No	1

```
./servers/slapd/back_bdb/cache.c
int bdb_cache_find_id(...){ //593
    ......
    ldap_pvt_thread_mutex_lock(&bdb->bi_cache.lru_mutex); //784
    ......
    bdb_cache_lru_add(bdb, ...); //799
} //804

static void bdb_cache_lru_add(bdb, ...){ //510
    ......
    ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock); //555
    ......
    ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock); //559
    ......
} //567
```

(a) bdb_cache_find_id

```
./servers/slapd/back_bdb/cache.c
int bdb_cache_add(...){ //829
    ......
    bdb_entryinfo_add_internal(bdb, ...); //864
    ......
    ldap_pvt_thread_mutex_lock(&bdb->bi_cache.lru_muttex); //880
    ......
    ldap_pvt_thread_rdwr_wunlock(&bdb->bi_cache.c_rwlock); //882
    ......
} //888

static int bdb_entryinfo_add_internal(bdb, ...){ //214
    ......
    ldap_pvt_thread_rdwr_wlock(&bdb->bi_cache.c_rwlock); //225
    ......
} //258
```

(b) bdb_cache_add

Figure 10. The bug#3494 in OpenLDAP

The BugType column in Table 1 has three components: DN means the number of deadlock cycles, RECUR refers to whether the mutexes involved in deadlocks are recursive, MN means the number of mutexes involved in deadlocks. We insert into Sshfs-fuse-2.2 three deadlocks that can happen simultaneously. We will check whether or not Mocklinter can detect all of the three deadlocks if they happen at the same time. Besides, we modify the bug#38804 in MySQL and convert it into a self-deadlock bug caused by one mutex. We will check whether or not Mocklinter can detect this self-deadlock.

6.2. Results

Our experiments are performed on a 4-core processor (Intel Q8200, 2.33GHZ) machine, 2GB RAM, running Ubuntu 12.04 (whose kernel is Linux-3.2.0). Mocklinter is configured with τ =0.1s.

Table 2. Overall Bug Detection Results

Application	Deadlock Detected?	# of threads	# of mutexes	# of events
Bank Transactions	√	2	2	709.8
Dining Philosophers	√	5	5	4148.4
Ngorca	√	193.5	2	226.9

Tgrep	√	16.5	5	501.3
Sshfs-fuse	√	374.7	6	641.7
SQLite	√	2	2	9
HawkNL	√	3	4	20
OpenLDAP	√	4.9	1895.9	1280537.1
MySQL-6.0.4	√	23.4	354.7	1812489.9
MySQL-5.1.24	√	13	418	1361101.5

The deadlock triggers are different from each other. The deadlock in Dining-Philosophers and Ssh-fuse are triggered by directly running the corresponding applications, with no special inputs. The deadlock in Bank-Transaction can be triggered in a high probability, if given two agents of different deposits as input. We trigger the deadlock in Ngorca by feeding it an encoded string and trigger the deadlock in Tgrep by searching a fifteen-letter word in directory containing 10000 text files. We use the test cases attached with Dimmunix [21] to trigger the deadlocks in SQLite and HawkNL. For the last three deadlocks, we refer to their report pages and make our own triggers to trigger them.

We report our results, as shown in Table 2, with average performance on ten runs for each application. The second column shows that Mocklinter successfully capture all the ten deadlocks, including the multi-cycles deadlocks in Sshfs-fuse, the self-deadlock in MySQL-5.1.24 and the deadlock involving recursive mutexes in Bank-Transaction. The last three columns show, respectively, the number of threads, mutexes and events, handled by Mocklinter before it captures a given deadlock.

Mocklinter can detect multi-cycle deadlocks and self-deadlocks, so it can capture an interesting deadlock if this deadlock occurs (the first requirement in Section 3). Mocklinter captures an interesting deadlock within 0.1s or less time, so Mocklinter satisfies the second requirement posed in Section 3. Mocklinter reports a deadlock if and only if this deadlock happens in runtime, so Mocklinter never reports false warnings (the third requirement in Section 3).

6.3. Performance Overhead

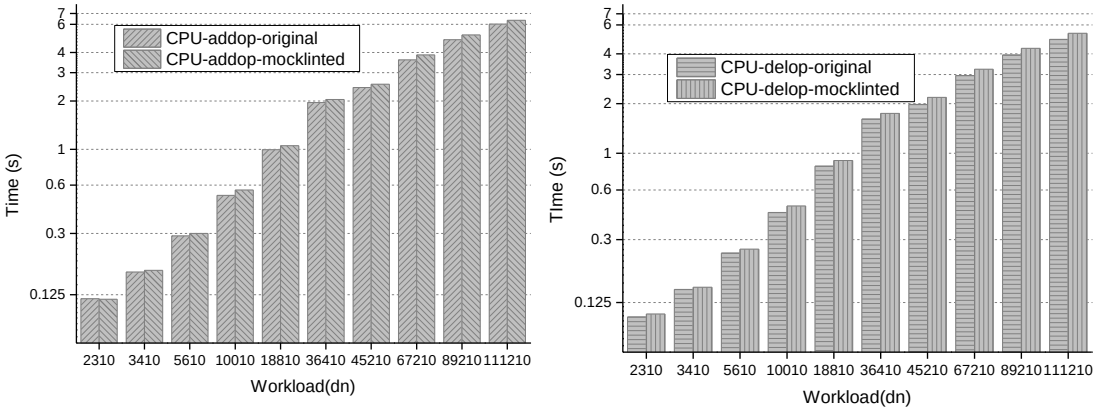
To measure Mocklinter's overhead under various clients and workloads, we prepare some test cases for performance evaluation and run them against OpenLDAP-2.2.20 equipped with or without Mocklinter. All our experiments are done in the case that no deadlock is encountered. As shown in Figure 10, the deadlock bug#3494 does exist in OpenLDAP. However, our test cases intentionally bypass it by sending requests of different types separately.

6.3.1. One Client with Various Workloads: The first test case, denoted as *addop*, automatically sends INSERT requests of variant sizes (from 2 thousands to 100 thousands) to slpad, the OpenLDAP daemon. The second test case, denoted as *delop*, sends DELETE requests of the same size as the first. The two test cases run from the same client, successively. We measure the average performance on ten runs for each workload. We count the CPU occupation and the slowdown for the two test cases, as shown in Table 3. Figure 11 is a visualization form of Table 3.

Table 3. CPU Occupation, Slowdown and Overhead for the Case that one Client with Various Workloads

Time / Workload	original(s)		mocklinted(s)		slowdown(s)		overhead	
	addop	**delop**	**addop**	**delop**	**addop**	**delop**	**addop**	delop
2310	**0.1180**	**0.1020**	**0.1168**	**0.1064**	**-0.0012**	**0.0044**	**-1.02%**	4.31%

3410	**0.1728**	**0.1500**	**0.1768**	**0.1544**	**0.0040**	**0.0044**	**2.31%**	2.93%
5610	**0.2896**	**0.2484**	**0.3000**	**0.2628**	**0.0104**	**0.0144**	**3.59%**	5.80%
10010	**0.5172**	**0.4384**	**0.5588**	**0.4800**	**0.0416**	**0.0416**	**8.04%**	9.49%
18810	**0.9956**	**0.8352**	**1.0540**	**0.9044**	**0.0584**	**0.0692**	**5.87%**	8.29%
36410	**1.9544**	**1.6132**	**2.0416**	**1.7468**	**0.0872**	**0.1336**	**4.46%**	8.28%
45210	**2.4264**	**1.9784**	**2.5384**	**2.1792**	**0.1120**	**0.2008**	**4.62%**	10.15%
67210	**3.6032**	**2.9608**	**3.8668**	**3.2260**	**0.2636**	**0.2652**	**7.32%**	8.96%
89210	**4.8120**	**3.9444**	**5.1448**	**4.3220**	**0.3328**	**0.3776**	**6.92%**	9.57%
111210	6.0148	4.9016	6.3464	5.3284	0.3316	0.4268	5.51%	8.71%



(a) addop **(b) delop**

Figure 11. A Visualization form of Table 3

From Table 3, we can say that, generally, if slpad runs with Mocklinter, both *addop* and *delop* need more time to complete. However, this is not true for *addop* running with the first workload. For the first workload, *addop* surprisingly achieves a better performance (a speed up by 1%). We believe that this phenomenon is related to the warm-up problem. Our measurements are carried out in two stages. We first run various workloads against the original slpad. When finishing, we shut down slpad and clean up all resources consumed by it. Then we launch the mocklinted slpad in a brand-new environment and run *addop* with the first workload. So the mocklinted slpad is not warmed up and may quickly process requests from *addop*. This explains why *addop* may require less time to finish its operations, compared with the time used when *addop* runs against the original slpad.

Experiment results indicate that the performance overhead of Mocklinter is low. The max overhead for *delop* is 10.15%, while only 8.04% for *addop*. In addition, the overhead doesn't monotonously increase with the workload's size. This is because that Mocklinter detects cycles in an asynchronous way and only check cycles for the requesting threads.

6.3.2. Various Clients with Constant Workload: The third test case, named as *searchop*, sends a total of 20480 SEARCH requests to slpad from clients of a varying number (from 2 to 1002). The workload of each client is the result of 20480 being divided by the number of clients. This means that if there are two clients, each client will send 10240 requests. We simulate a client with a thread which connects to the slpad through a unique connection. Clients send requests from a client machine with AMD Athlon X2 5000+ CPU, 2GB memory. A 100Mb Ethernet connects the client machine and the server machine, on which the slpad runs. We measure the average performance on ten runs for each number of clients. Table 4 shows the different time slpad needs to finish all of the

20480 requests when it is coupled with/without Mocklinter, along with the slowdown and overhead. Figure 12 plots Table 4.

Table 4. CPU Occupation, Slowdown and Overhead for the Case that Various Clients with Constant Workload

Time / Clients	original(s)	mocklinted(s)	slowdown(s)	overhead
2	**4.1751**	**5.4947**	**1.3196**	31.61%
4	**3.8426**	**5.0879**	**1.2453**	32.40%
8	**3.6926**	**5.1351**	**1.4425**	39.06%
16	**3.7034**	**5.2843**	**1.5809**	42.69%
32	**3.7814**	**5.3883**	**1.6069**	42.49%
64	**4.5683**	**8.5077**	**3.9394**	86.24%
128	**4.5034**	**9.2233**	**4.7199**	104.81%
256	**4.4550**	**9.1069**	**4.6519**	104.42%
512	**4.0596**	**8.8893**	**4.8297**	118.97%
1002	3.6500	8.2304	4.5804	125.49%

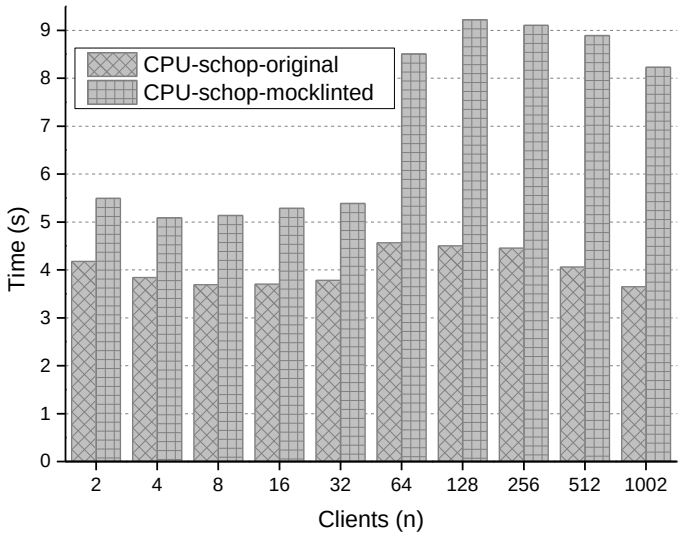


Figure 12. A Visualization form of Table 4

According to Table 4, Mocklinter introduces moderate overhead into slpad when slpad processes requests sent by multiple clients simultaneously. For 1002 clients, Mocklinter only slows down the original slpad by 125.5%. Although the overhead increases with the number of clients, the slope is not sheer. For example, Mocklinter introduces 31.6% overhead into slpad for 2 clients, while only 42.5% overhead for 32 clients. We therefore believe that Mocklinter is applicable to large scale, real world applications.

6.4. Memory Consumption

Another aspect of Mocklinter we want to measure is how much additional memory it consumes, compared to the original applications. In order to minimize the interference of various factors in the original programs, we select non-deadlock solutions to the Dining-Philosophers (from 5 to 1024 philosophers) problem as our baselines, because they only consume a small and stable size of memory. For an N-Dining-Philosophers problem, the non-hungry solution to it uses N threads to represent N philosophers, and N mutexes for

N chopsticks. In our solutions, each philosopher eats 1000 times before it terminates. We measure the average memory usage on ten runs for each solution. Table 5 shows the different memory consumption of ten Dining-Philosophers solutions when they run with/without Mocklinter.

Table 5. Peak Memory usage and Overhead for 10 Solutions

Memory / Philos	original(byte)	mocklinted(byte)	additional(byte)	overhead
5	**482099.2**	**1588019.2**	**1105920.0**	229.3968%
8	**460800.0**	**1920614.4**	**1459814.4**	316.8000%
10	**460380.4**	**2141388.8**	**1681008.4**	365.1347%
20	**489062.4**	**3486105.6**	**2997043.2**	612.8141%
30	**727859.2**	**4306944.0**	**3579084.8**	491.7276%
50	**726630.4**	**8527872.0**	**7801241.6**	1073.6189%
100	**996556.8**	**18824806.4**	**17828249.6**	1788.9848%
200	**1268121.6**	**38782156.8**	**37514035.2**	2958.2364%
500	**2619801.6**	**98928640.0**	**96308838.4**	3676.1882%
1024	4893900.8	203355750.4	198461849.6	4055.2896%

As indicated in Table 5, Mocklinter consumes about 2 ~ 40 times more memory than that of the original. We check out Mocklinter carefully, and find that the amount of memory consumed by Mocklinter is mainly related to the number of lock/unlock operations in the original solution. The more times each philosopher eats, the more overhead of memory usage will be introduced. Mocklinter makes three events for each lock/unlock pair on a given mutex. If there are lots of threads performing lock/unlock operations frequently, huge amounts of events will be generated and stored into event queues, waiting for being drained. This is why Mocklinter consumes so much additional memory. However, luckily enough, when Linter wakes up, it will quickly drain events and free the memory occupied by them. Once all events are drained completely, the memory usage will go back to the normal level.

According to the performance overhead (Section 6.3) and memory overhead (Section 6.4) introduced into the target program by Mocklinter, we can conclude that Mocklinter impose merely modest impact on the original program (the fourth requirement in Section 3).

6.5. Comparison

We compare Mocklinter with Dimmunix in terms of two aspects: deadlock detection capability and memory consumption. According to the paper [21] and our experiences, Mocklinter has performance overhead similar to Dimmunix. We don't compare them on how much they slow down the original programs.

Table 6. Comparison on Deadlock Detection Capability (√: Yes, ×: No, *: Uncertain)

Application	Deadlock Detected?	
	Mocklinter	Dimmunix
Bank Transactions	√	√
Dining Philosophers	√	*
Ngorca	√	√
Tgrep	√	√

Sshfs-fuse	√	×
SQLite	√	√
HawkNL	√	√
OpenLDAP	√	*
MySQL-6.0.4	√	*
MySQL-5.1.24	√	×

The deadlock detection capability of Dimmunix is not as strong as Mocklinter, as shown in Table 6. Dimmunix can't detect the self-deadlock in MySQL-5.1.24, neither the multi-cycles deadlock in Sshfs-fuse. In addition, Dimmunix is not sound because it may miss deadlocks. For example, for the deadlock in Dining-Philosophers, Dimmunix capture it each time it happens in a single-core machine. However, when the deadlock happens in a multi-core machine, Dimmunix sometimes will capture it, sometimes will not. This happens because Dimmunix wrongly implements the lock-free queue, which is used to relay events between Dimmunix's Avoidance module and Monitor module. When multiple threads write to the lock-free queue, some events may be lost. Because of the wrong implementation of the lock-free queue, Dimmunix is even not dynamically complete. It may misleadingly report a deadlock that doesn't exist. For example, for the deadlock in OpenLDAP and MySQL-6.0.4, Dimmunix sometimes can correctly report them when they happen, however, sometimes Dimmunix will report false deadlocks.

Dimmunix consumes more additional memory than Mocklinter, because it never frees memory occupied by events. As time goes by, Dimmunix will consume more and more memory, until it runs out of the memory space. We try to quantitatively measure how many additional memories consumed by Dimmunix using the non-deadlock Dining-Philosophers solutions. However, we fail to do that. In our try, Dimmunix falsely report deadlocks for the non-deadlock solution to the 50-Dining-Philosophers problem, and blocks forever (we wait for one hour) for the solution to 100-Dining-Philosophers problem. These drawbacks of Dimmunix make the measurement infeasible.

7. Limitations and Future Work

Mocklinter can only detect deadlocks involving threads residing in the same process. If threads from different process become deadlocked because of cyclic waiting on mutexes, Mocklinter will not help.

Another drawback of Mocklinter is that it can't see anything happening in a child process which is forked by its parent process using the system call "fork". If threads in the forked process trap into deadlock states, Mocklinter can do nothing about that. For example, we once inserted a deadlock into Sshfs-fuse-2.2 and had Mocklinter detect it. However, Mocklinter failed. The reason is that, after Ssh-fuse finishing initialization, it forked a process to serve as a daemon for interacting with the remote ssh server. The deadlock was inserted to the forked process, so Mocklinter couldn't capture it.

The third limitation is that only deadlocks caused by mutexes can be detected by Mocklinter. Real deadlocks often happen as a result of mixed effects of multiple synchronizations, such as mutexes, read/write locks, conditional variables or even semaphores, instead of pure effects of mutexes. How to detect the mixed deadlocks will be our future work. We have made some progress in that direction.

8. Conclusions

This paper presents Mocklinter, a dynamic analysis and code instrumentation tool that helps to detect deadlocks caused by mutexes of all the four types. We evaluate Mocklinter using ten deadlocks in different applications with greatly varying code sizes. The result shows that Mocklinter can detect all the ten deadlocks, including self-deadlocked

deadlocks and multi-cycle deadlocks, with no dynamic false positives and no dynamic false negatives.

We quantitatively evaluate the performance overhead and memory consumption of Mocklinter. Mocklinter scales to 1002 threads while incurring only 125% overhead. Only when lots of events are generated and stored into event queues will Mocklinter consume much more memory than the original program. Once Mocklinter drains all events, the additional memory will be freed.

In addition, Mocklinter uses ticketed-DFS, a novel cycle detection algorithm, to check not only whether cycles exist or not, but also what they are. Thanks to ticketed-DFS, Mocklinter can output IDs of threads or mutexes that are involved in deadlocks. Combining this information with the core file, testers can quickly locate causes of deadlocks in the source level.

At present, Mocklinter can detect deadlocks caused by mutexes only. Future work will extend Mocklinter to capture deadlocks involving multiple synchronizations.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grant No. 61173021). We thank the anonymous reviewers for useful feedback, Yan Cai in City University of HongKong for discussing how to reproduce the bug#37080 in MySQL, Aaron Richton in Rutgers, The State University of New Jersey, for discussing how to reproduce the bug#3494 in OpenLDAP.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution", ACM Queue, vol.3, no.7, **(2005)**, pp.54-62
[2] R. Agarwal, L. Wang and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring", Proc. of the 1st International Haifa Verification Conference, **(2005)**, Haifa, Israel, pp.191-207.
[3] S. Lu, S. Park, E. Seo and Y. Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics", ACM SIGARCH Computer Architecture News, vol. 36, no. 1, **(2008)**, pp. 329-339.
[4] P. Joshi, C. S. Park, K. Sen and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks", Proc. of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, **(2009)**, pp. 110-120, Dublin, Ireland.
[5] M. Naik, C. S. Park, K. Sen and D. Gay, "Effective static deadlock detection", Proc. of the 31st International Conference on Software Engineering, **(2009)**, pp. 386-396, Vancouver, Canada.
[6] SQLite bug#1672, Available at: http://www.sqlite.org/src/info/a6c30be214/, Last visited: **(2014)** November 26.
[7] R. C. Holt, "Some deadlock properties of computer systems", ACM Computing Surveys, vol. 4, no. 3, **(1972)**, pp. 179-196.
[8] S. Bensalem, A. Griesmayer, A. Legay, T. H. Nguyen and D. Peled, "Efficient deadlock detection for concurrent systems", Proc. of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, **(2011)**, pp. 119-129, Cambridge, UK.
[9] C. Li, L. P. Huang, L. X. Chen and W. C. Luo, "Deadlock detection and recovery for component-based systems", Mathematical and Computer Modeling, vol. 58, no. 5, **(2013)**, pp.1362-1378.
[10] S. Bensalem, M. Bozga, T. H. Nguyen and J. Sifakis, "D-Finder: a tool for compositional deadlock detection and verification", Proc. of the 21st International Conference on Computer Aided Verification, **(2009)**, pp. 614-619, Grenoble, France.
[11] C. Boyapati, R. Lee and M. Rinard, "Ownership types for safe programming: preventing data races and deadlocks", ACM SIGPLAN Notices, vol.37, no.11, **(2007)**, pp.211-230.
[12] C. S. Gordon, M. D. Ernst and D. Grossman, "Static lock capabilities for deadlock freedom", Proc. of the 8th ACM SIGPLAN workshop on Types in Language Design and Implementation, **(2012)**, pp. 67-78, Philadelphia, USA.
[13] P. Gerakios, N. Papaspyrou, K. Sagonas and P. Vekris, "Dynamic deadlock avoidance in system code using statically inferred effects", Proc of the 6th Workshop on Programming Languages and Operating Systems, **(2011)**, pp. 134-141, Cascais, Portugal.

[14] P. Gerakios, N. Papaspyrou and K. Sagonas, "A type and effect system for deadlock avoidance in low-level languages", Proc. of the 6th ACM SIGPLAN Workshop on Types in Language Design and Implementation, **(2011)**, pp. 15-28, Austin, USA.
[15] T. Kelly, Y. Wang, S. Lafortune and S. Mahlke, "Eliminating concurrency bugs with control engineering", IEEE Computer, vol. 42, no. 11, **(2009)**, pp. 52-60.
[16] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune and S. Mahlke, "Gadara: dynamic deadlock avoidance for multithreaded programs", Proc. of the 8th Symposium on Operating Systems Design and Implementation, **(2008)**, pp. 281-294, San Diego, USA.
[17] A. Williams, W. Thies and M. D. Ernst, "Static deadlock detection for Java libraries", Proc. of the 19th European Conference on Object Oriented Programming, **(2005)**, pp. 602-629, Glasgow, UK.
[18] D. Engler and K. Ashcraft, "RacerX: efficient, static detection of race conditions and deadlocks", ACM SIGOPS Operating Systems Review, vol. 37, no. 5, **(2005)**, pp. 237-252.
[19] S. Bensalem and K. Havelund, "Scalable dynamic deadlock analysis of multithreaded programs", Proc. of the 1st International Haifa Verification Conference, **(2005)**, pp. 208-223, Haifa, Israel.
[20] Y. Cai and W. K. Chan, "MagicFuzzer: scalable deadlock detection for large-scale applications", Proc. of the 34th International Conference on Software Engineering, **(2012)**, pp. 606-616, Zurich, Switzerland.
[21] H. Jula, D. Tralamazza, C. Zamfir and G. Candea, "Deadlock immunity: enabling systems to defend against deadlocks", Proc. of the 8th Symposium on Operating Systems Design and Implementation, **(2008)**, pp. 295-308, San Diego, USA.
[22] H. Jula and G. Candea, "A scalable, sound, eventually complete algorithm for deadlock immunity", Proc. of the 8th Workshop on Runtime Verification, **(2008)**, pp. 119-136, Budapest, Hungary.
[23] Y. N. Buchbinder, R. Tzoref and S. Ur, "Deadlocks: from exhibiting to healing", Proc. of the 8th Workshop on Runtime Verification, **(2008)**, pp. 104-118, Budapest, Hungary.
[24] M. Musuvathi, S. Qadeer, T. Ball and G. Basler, "Fining and reproducing heisenbugs in concurrent programs", Proc. of the 8th Symposium on Operating Systems Design and Implementation, **(2008)**, pp. 267-280, San Diego, USA.
[25] T. Li, C. S. Ellis, A. R. Lebeck and D. J. Sorin, "Pulse: a dynamic deadlock detection mechanism using speculative execution", Proc. of the 2005 USENIX Annual Technical Conference, **(2005)**, pp. 31-44, Anaheim, USA.
[26] M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations", ACM SIGPLAN Notices, vol.47, no.6, **(2012)**, pp.521-530.
[27] Helgrind, Available at: http://valgrind.org/docs/manual/hg-manual.html/, Last visited: **(2014)** November 26.
[28] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", ACM SIGPLAN Notices, vol. 42, no. 6, **(2007)**, pp. 89-100.
[29] A. Bron, E. Farchi, Y. Magid, Y. Nir and S. Ur, "Applications of synchronization coverage", Proc. of the 10th ACM SIGPLAN symposium on Principle and Practice of Parallel Programming, **(2005)**, pp. 206-212, Chicago, USA.
[30] A. Kamil, "Graph algorithms", Available at: http://www.cs.berkeley.edu/~kamil/teaching/sp03/041403.pdf, Last Visited: **(2014)** November 26.
[31] Y. Cai, Z. Ke, S. R. Wu and W. K. Chan, "TeamWork: synchronizing threads globally to detect real deadlocks for multithreaded programs", Proc. of the 18th ACM SIGPLAN symposium on principles and practice of parallel programming, **(2013)**, pp. 311-312, Shenzhen, China.
[32] Y. Cai and W. K. Chan, "Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs", IEEE Transaction on Software Engineering, vol. 40, no. 3, **(2014)**, pp. 266-281.

Authors

Zhen Yu, He was born in 1987 and currently is a PhD candidate in the School of Computer Science and Technology at Harbin Institute of Technology (HIT). His research interests are in the areas of software static or dynamic analysis, implicit rules mining from large-scale software, and software bug (especially concurrent bug, including deadlock, data race and atomicity violation) detection and avoidance.

Xiaohong Su, She is a professor and PhD supervisor in the School of Computer Science and Technology at HIT. She has managed a number of research projects supported by the Ministry of National Defense and the National Natural Science Foundation of China. Her current research interests include software engineering, information fusion, image processing and computer graphics.

Tiantian Wang, She is an associate professor in the School of Computer Science and Technology at HIT. She has published multiple papers on some famous journals and conferences, such as JSS, WCLL and FSE. Her current research interests include program analysis, software bug detection and location, software testing, and so on.

Peijun Ma, He is a professor and PhD supervisor in the School of Computer Science and Technology at HIT. He is an IT education consultant of Japan Computer Service Company. He has managed many projects, such as the project supported by the Ministry of National Defense and National Natural Science Foundation of China as well as the international cooperative projects. His main research interests include software engineering, information fusion, image processing, embedded system simulation, and so on.

