Lockout: Efficient Testing for Deadlock Bugs

Ali Kheradmand[*]
School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne
a.i.kheradmand@gmail.com

Baris Kasikci
School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne
baris.kasikci@epfl.ch

George Candea
School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne
george.candea@epfl.ch

ABSTRACT

Deadlocks are hard to find via traditional testing, and they manifest rarely during program execution. We introduce Lockout, a technique and a tool that increases the probability of deadlock manifestation in multithreaded programs, while preserving the program semantics and requiring no perturbation to the runtime and the testing infrastructure. Lockout produces binaries that are more prone to deadlock compared to native binaries. We evaluated Lockout on a suite of multithreaded programs, and preliminary results show that it is effective in increasing the deadlock probability.

1. INTRODUCTION

A deadlock is a condition under which the progress of a program is halted, because each thread in a given set of threads attempts to acquire a lock already held by another thread in the set [26]. Deadlocks are hard to identify by manual code inspection and occur only in rare cases among an exponential number of thread interleavings a program may have; consequently, deadlocks may not easily manifest via traditional testing.

A number of mechanisms have been developed to aid developers in deadlock detection. Static deadlock detectors [9, 24, 26] attempt to detect possible deadlocks by analyzing the program's source code. Such detectors impose no runtime overhead or interference on the normal execution of programs. However, they tend to report many false positives, each of which could take a considerable amount of the developers' time only to verify manually.

Dynamic deadlock detectors [2, 3, 17, 20] try to detect deadlocks by monitoring the program during execution or by analyzing a trace of the program after its execution. Dynamic detectors operate on feasible program paths and are more accurate in terms of pointer aliasing information [9] than static detectors. Still, dynamic detectors report both false negatives and false positives, and they typically impose high runtime overhead on the running program.

Model checkers [4, 7, 11] systemically explore the state space of the program in order to find deadlocks. While they report no false positives, the state space explosion problem [6] and the need for the abstract specification of the system under test [9] limit their application to relatively small programs.

Deterministic execution and reachability testing techniques [23, 13, 19] aim to systemically explore a program's thread interleavings to find deadlocks. These tools do not report false positives, however, they only function properly if non-determinism is in thread scheduling and not in other external events (e.g., inputs). For example CHESS [23] cleans the state of memory before each run, and logs and replays the values returned by the functions that read the current time and generate random numbers.

We present a tool called Lockout that increases the probability of deadlock occurrence in multithreaded programs, while preserving program semantics and requiring no perturbation to the runtime and test infrastructure. Lockout takes the source code of a program, and it produces a binary (similar to the native binary) that is more prone to deadlock at runtime. This way, more deadlocks can be detected, using the same test suite.

There are several challenges to be addressed when designing such a tool. Ideally, such an approach should have minimum perturbation to the target program's execution, with the only difference being an increased probability of a deadlock. To achieve this, Lockout should not change program semantics and also impose low runtime overhead. In addition, Lockout's actions should not cause any unreal deadlocks or starvation. Furthermore, the ideal approach should not be dependent on the characteristics of a single execution of the program and work in the face of input non-determinism. Finally, some deadlocks happen because of data races. In such cases, data races can influence the locking order of a program. For example, if the result of a branch depends on a racy variable, the race may change the expected outcome of the branch and consequently the locking order, which in turn may result in a deadlock.

We attempt to address the aforementioned challenges with Lockout. By preempting threads before lock acquisitions and some memory accesses, we affect thread scheduling to produce interleavings that are more likely to deadlock. Even though we change the program's binary, the results of these changes are limited to effects on the program's scheduling and do not alter the program's semantics.

Lockout collects information about lock acquisition order during program execution and preserves it across different executions. Then, Lockout modifies the program schedule based on this information that is not directly dependent on a single execution and input. Therefore, Lockout is able to perform just as well under input non-determinism. This feature is likely important in interactive programs which introduce a lot of non-determinism through the inputs. In addition, applying preemptions before racing memory accesses could increase the probability of encountering data

[*]This work was done while the author was a summer intern at EPFL in 2013

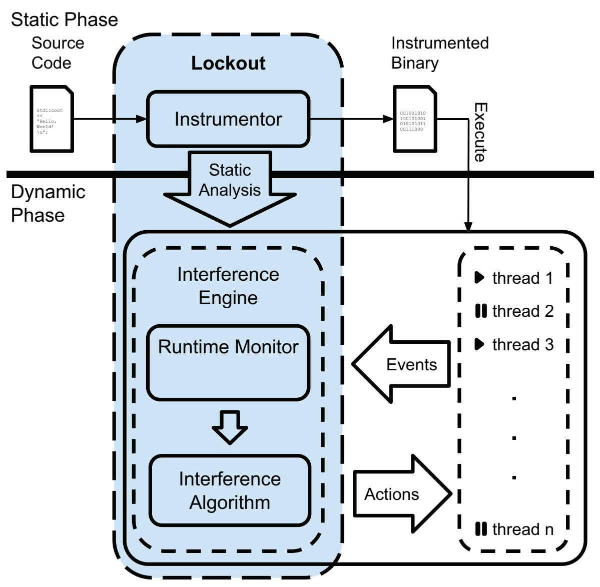

Figure 1: Lockout's Architecture.

race dependent deadlocks during the program's execution.

In the rest we describe the design of Lockout (§2) and its implementation (§3), evaluate it (§4), discuss the limitations (§5) and related work (§6), and finally conclude (§7).

2. DESIGN

Figure 1 depicts the architecture of Lockout. Lockout works in two phases. In the static phase, its instrumentor takes the program's source code and produces an instrumented program binary (§2.1). In the dynamic phase, Lockout's Interference Engine (IE) gathers information about the program's locking behavior (§2.2), and at certain points during the program's execution, IE uses this information to decide whether to interfere with the scheduling or not (§2.3).

2.1 Producing Target Binary

The instrumentor instruments the program's binary at specific locations including lock acquisition and release calls and memory accesses. This instrumentation is later used by IE to gather information about the locking behavior of the program and to interfere with the thread scheduling. Lockout selects appropriate instrumentation locations depending on the interference algorithm it uses (more in §2.3).

During the static phase, Lockout may also run static analyses on the target source code to produce information which may come in handy later in the dynamic phase. For example, Lockout can run a static deadlock detector to find potential deadlocks. IE may use this information when interfering with the thread schedule. Also, Lockout can use static data race detection to detect potential data races, and use this information to reduce the number of memory access instrumentation. Generally, static analysis can assist in reducing the runtime overhead at the cost of losing accuracy. However, in our early prototype of Lockout, we do not use any static analysis.

2.2 Capturing the Program Behavior

When the target program is executed, IE gathers information about the execution by keeping track of the threads and the locks that they hold. It also keeps a directed graph that we call the runtime lock order graph (RLOG). In this graph, each node represents a lock. We put a directed edge E from node N_1 to node N_2 if a thread T with a current lockset (i.e locks being held by T) S that contains N_1, attempts to acquire N_2.

E can potentially contain various information about the particular event that caused the edge to appear such as the number of edges between N_1 and N_2. This can give an estimate (albeit not always accurate) of the likelihood of occurrence of such an acquisition order. IE may use this information for more effective preemptions. In the current implementation, Lockout establishes an edge with no extra information once a thread acquires N_2 after acquiring N_1.

When the program ends, IE stores the RLOG in non-volatile memory. At the beginning of each run, IE loads the previously stored RLOG and continues updating it. This approach has a number of benefits. First, IE has enough information to start interference at the beginning of each run. Second, it can capture more information about the program's locking behavior as the program follows different flows in different executions, each execution capturing part of the entire locking behavior. Finally, after enhancing the RLOG for a sufficient number of runs, the IE could use any anomaly (e.g a brand new edge between two nodes) as a sign of a potential deadlock. An anomaly might be a rare correct behavior, in which case the algorithm only causes (rare) performance overhead.

The RLOG can also be built using static analysis. However, this option might be less accurate as static analysis is usually inaccurate in determining feasible paths and more importantly pointer aliasing information [9].

2.3 Deciding How to Interfere

To deadlock a program, Lockout should make a set of threads circularly wait for each other. To do this, Lockout needs to limit the number of locks a thread can acquire. Otherwise, a thread is likely to acquire all the locks it needs, do its job, and release all the locks; making those locks available for other threads that wait for them, and thus, diminish the chance of a deadlock. To prevent this, we try not to let threads acquire all the locks they need by interfering with scheduling. This way, threads acquire part of the locks they need, and the rest can be acquired by other threads.

Deciding when and how to interfere with scheduling is the most important functionality of Lockout. Before each lock acquisition attempt by any thread, IE decides either to let the thread acquire the lock, or delay the thread before letting it acquire the lock. IE also decides on how long to delay an acquisition. We implemented three different algorithms which differ in terms of their interference rate, effectiveness and run time overhead. The algorithms are elaborated in the following subsections.

2.3.1 Simple Preemption

In this algorithm, we aim for low overhead. The instrumentor instruments the program before all lock acquisitions and after all lock releases. IE does not create and preserve the RLOG, because the algorithm does not use it; it simply preempts any thread exactly before trying to acquire a lock and exactly after releasing the lock.

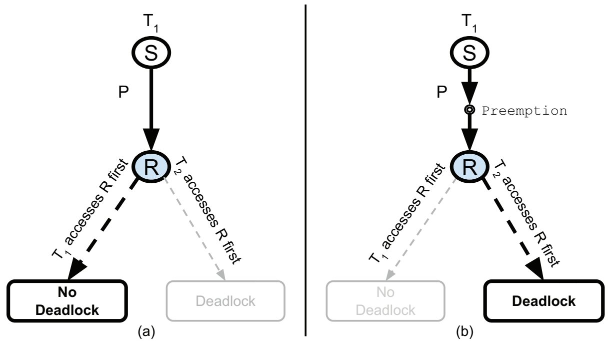

Figure 2: Data race dependent deadlocks involving one data race.

```
        (a)                                  (b)
Time  Thread 1          Thread 2    |  Thread 1          Thread 2
 |    ...                           |  ...
 |    unlock(l);                    |  unlock(l);
 |    write(a, 1);                  |  preempt();
 |    ...                           |  ...
 |
 |                   lock(l);       |                   lock(l);
 |                   v = read(a);   |                   v = read(a);
 |                   if(v == 1)     |                   if(v == 1)
 |                     //no deadlock|                     //no deadlock
 |                   else           |                   else
 v                     //deadlock   |                     //deadlock
                                    |  write(a,1);
```

Figure 3: Example of a data race dependent deadlock.

By preempting threads before acquisition of each lock, we make sure that a thread (T_1) relinquishes the CPU before acquiring two locks (l_1 and l_2) in a row. This way, another thread (T_2) may get a chance to acquire l_2, making T_1 which already holds some locks, wait for T_2 to release l_2. T_2 may wait for a third thread in a similar way and so on... Eventually, this may lead to threads circularly waiting for each other, and cause a deadlock.

Data races may change the locking order in the program and eventually lead to a deadlock. The reason for preempting threads after lock releases is to target a subset of such deadlocks. More precisely, we target changes that are caused by a single data race between two threads.

Fig. 2a shows a data race between memory accesses in two threads (T_1 and T_2). If T_1 accesses the racy memory before T_2, then there is no deadlock; but if T_2 accesses it first, locking order changes and the program might deadlock. We assume that normally T_1 has higher chances to access the memory first, otherwise the normal program is already likely to deadlock with high probability. Consider the beginning of the racy region for T_1 (S) where the intersection of the locksets of T_1 and T_2 is empty. Also, consider the racy memory access (R) and the path that T_1 takes from S to R (P). There is probably no instruction along P that causes T_1 to relinquish the CPU since otherwise T_2 could more often have the chance to acquire the CPU and access the memory before T_1. For the same reason, S and R probably have spatial locality. To make T_2 more likely to access the memory before T_1, we can preempt T_1 somewhere along P (Fig. 2b). In this algorithm, Lockout preempts T_1 right after S.

Figure 3a shows an example for which this approach is effective. Assume T_1 only holds the lock l and runs concurrently with T_2. There is a data race between the `write` in T_1 and the `read` in T_2. This data race may cause a deadlock if the `read` executes before the `write`. However, T_2 does not usually have the chance to execute the `read` before T_1 executes the `write` and a deadlock does not happen. By preempting T_1 after the `unlock` (Fig. 3b), we boost the probability of the deadlock. This approach is not as effective if the change in the locking behavior is dependent on affecting more than one racy variable between two threads. We address such situations (with higher overhead) in §2.3.3.

2.3.2 Targeted Deadlock Induction

This algorithm chooses a specific potential deadlock and tries to rise its probability of manifestation. The instrumentor instruments all lock acquisition and release calls (before and after) for IE's use. IE keeps track of the RLOG.

Each directed loop in the RLOG corresponds to a potential deadlock. Consider a directed loop $N_1 \rightarrow N_2 \rightarrow ... \rightarrow N_n \rightarrow N_1$. As mentioned earlier, each edge $N_i \rightarrow N_j$ means there might exist a thread in the program that after acquisition of N_i, acquires N_j. By preventing the thread T_i currently holding N_i from acquiring N_j, hopefully a thread T_j may acquire N_j; making T_i dependent on T_j and eventually, with a circular dependency, a deadlock may happen.

The algorithm chooses a specific directed loop in the graph at the beginning of the execution (assuming information from previous runs exists) and tries to make threads circularly dependent on each other. This is done by delaying any thread holding at least one lock from the loop just before attempting to acquire another lock from the loop. Delay time is dependent on the size of the loop. The algorithm adds a random delay to prevent a common delay from not affecting the schedule at all.

We implemented this algorithm, but did not test it, as it evolved into the third algorithm explained below. The initial motivation for this algorithm was to statically detect potential deadlocks, produce different binaries each targeting only one or a group of potential deadlocks, and run the binaries on different machines. This approach, inspired by the RaceMob system[16], has the advantage of achieving higher accuracy while keeping the overhead low by instrumenting and preempting only a few locations in the binary.

2.3.3 Component Based Delays

This algorithm finds strongly connected components (SCC) of the RLOG at the beginning of the program's execution, and it assigns numbers to the components based on their topological order (root is assigned 1 and so on). After each change in the RLOG, IE updates the components accordingly. Since any set of nodes within a strongly connected component form a directed loop, this is a sign of a potential deadlock.

The algorithm decides whether or not to delay a thread, which previously acquired a lock l_p, before acquiring a lock l_c based on the following relations among locks' components (C_i shows the component number of the lock i):

- If $C_{l_p} < C_{l_c}$, the algorithm does not delay the thread. Since the C_{l_c} is topologically ordered after the component of C_{l_p}, there is no directed path from l_c to l_p. Therefore, making the thread holding l_p wait for another thread containing l_c will probably not lead to a deadlock.

- If $C_{l_p} = C_{l_c}$, the algorithm delays the thread because l_c and l_p form a directed loop.
- If $C_{l_p} > C_{l_c}$, it means a new edge will be added from l_p to l_c just after the thread acquires l_c. It also means the edge will be the first from l_p to l_c (otherwise $C_{l_p} \leq C_{l_c}$). If there is a directed path from l_c to l_p, there is a directed loop involving l_p and l_c (including the edge that will be added after the acquisition). Therefore, the algorithm delays the thread. If such path does not exist, the algorithm still delays the thread since this new edge can be a sign of an anomaly in locking behavior and potential for a deadlock.

Deadlocks caused by locking orders will eventually create a directed loop in the RLOG (at least when they manifest). But SCCs are not a necessary condition for finding potential deadlocks as some edges may still have not been added to RLOG before the deadlock manifests. However, because deadlocks manifest in a very particular order of lock acquisition, it is reasonable to assume that the edges in a real deadlock's directed loop will appear in the RLOG after an adequate number of runs, and thus form an SCC. In addition, the algorithm does not only rely on SCCs as it always preempts threads if $C_{l_p} > C_{l_c}$.

Each delay duration is random to prevent a common delay from not affecting the schedule. If a thread does not have locks in its lockset, we assign $C_{l_p} = \infty$ to it to prevent delaying the thread before acquiring its first lock.

The algorithm also preempts threads before any *w*rite access in order to give rise to deadlocks caused by data races. This approach is more general than the one described in §2.3.1, as the former is only effective for cases in which the change in the locking order is a result of a single data race. Figure 4a illustrates the situation in which several data races (R_1, R_2, ..., R_n) are involved, and the lock order would be affected only if T_1 accesses R_1 to R_{n-1} before T_2 but accesses R_n after T_2. Here, unlike in §2.3.1, preempting T_1 somewhere along P (Fig. 4b) would not be as effective in increasing the deadlock probability as preempting it along P' (Fig. 4c). Currently, we preempt threads before all memory accesses to increase the probability of T_2 accessing R_n before T_1. We plan to use static analysis to detect such situations and limit the preemption (and instrumentation) to R_n in order to increase the deadlock probability and decrease runtime overhead in future work.

3. IMPLEMENTATION

We implemented Lockout in nearly 1000 lines of C++ code. We targeted the Pthreads API; however, extension to other APIs would be fairly straightforward as Lockout just needs to know the calls to lock acquisition and release functions to instrument them. The instrumentor is an LLVM [18] pass. Lockout compiles source code into LLVM bitcode using the clang [5] compiler. We chose to use LLVM, because we aim use LLVM static analyses like static deadlock and data race detection in the future. After the instrumentation and static analyses (if any), Lockout links the resulting LLVM bitcode with IE and any additional libraries to produce the final executable.

IE uses Pthreads mutexes to synchronize the threads when accessing data it maintains such as the RLOG and current per-thread lockset. Lockout uses `pthread_yield` for preemption and `usleep` for delays.

We could build Lockout by modifying the Pthreads APIs. However, this would have required changes to system platforms, which is intrusive and not desirable. Also, directly changing program binaries provides a higher degree of control over other events such as memory accesses.

Our prototype is at https://github.com/kheradmand/Break.

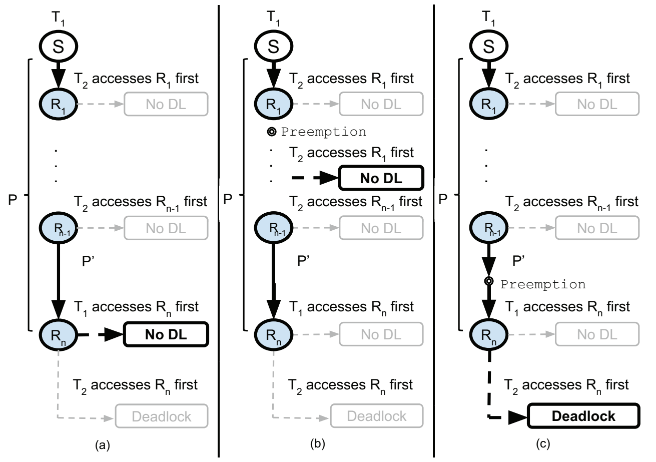

Figure 4: Data race dependent deadlocks involving more than one data race.

Table 1: Experimental targets of Lockout. The last column reports whether the programs include known data race dependent deadlock bugs or not.

Program	**Size**(LOC)	**#Threads**	**Race-based**
benchmark	34-226	2-4	Yes
SQLite 3.3.0	113,326	3	Yes
HawkNL 1.6b3	9,807	2	No
Pbzip2 1.1.6	4,523	4	No
httpd 2.0.65	300,140	128	No

4. EVALUATION

In this section, we answer two questions about Lockout: How effective is it in increasing the deadlock probability (§4.1)? How efficient is it in doing so(§4.2)?

To answer the questions, we evaluated Lockout's performance on on a set of real-world C/C++ programs that use the Pthreads API. We tested SQLite, a widely deployed database engine [25]; HawkNL, an open source, fairly low level game-oriented network API [1]; Pbzip2, a parallel implementation of the bzip2 file compressor [10]; and Apache httpd, a popular web server [12]. We also implemented and evaluated a microbenchmark with two locks that can potentially deadlock. Table 1 summarizes the properties of the tested programs.

The experiments were performed on an Intel Core 2 Duo E6600 CPU with 4 GB of RAM running Ubuntu Linux 12.04. Reported results are all averages of 10 experiments.

4.1 Effectiveness

To evaluate Lockout's effectiveness, we measured the probability of a deadlock by counting the number of successful

Table 2: Effectiveness of Lockout.

Program	Deadlock Probability (%)			
	Native	SP	CBD	
			MP	LP
benchmark	0.000658	0.568414	50	50
SQLite	<0.000640	4.345292	50	50
HawkNL	22.668514	63.511905	50	50
Pbzip2	<0.000001	<0.000001	<0.000001	<0.000001
httpd	<0.000001	<0.000001	<0.000001	<0.000001

Table 3: Efficiency of Lockout

Program	Overhead (%)		
	SP	CBD	
		MP	LP
Pbzip2	0.16	56.67	0.54
httpd	5.99	3303.82	20.87

runs (served requests for httpd) before any deadlock manifests (N). We report $\frac{1}{N+1}$ as the deadlock manifestation probability. Table 2 shows the results of our evaluation. The columns in Table 2 report the probability of a deadlock in different setups. The native column reports results for native programs, the SP column reports the results for instrumented programs using the *Simple Preemption* algorithm (§2.3.1). The CBD column represents the *Component Based Delays* algorithm (§2.3.3) with 1-3 microsecond delays: the MP subcolumn represents using preemptions before writes, and the LP subcolumn represents using preemptions after lock releases. We stopped our tests after 1,000,000 executions.

Lockout significantly increases the deadlock probability in the programs with known bugs. However, it was unable to reveal new bugs. SP made the microbenchmark, SQLite, and HawkNL approximately 864, 6780, and 3 times more prone to deadlock compared to normal executions, respectively. CBD made the same programs always deadlock every other execution. No algorithm could reveal any deadlocks in Pbzip2 and httpd, but we do not know whether or not these programs actually have any deadlocks. In our experiments, there was no loop in the RLOG of these two programs.

4.2 Efficiency

To determine Lockout's efficiency, we measure its runtime overhead using the elapsed time for successful executions (request throughput for httpd) both with instrumented and uninstrumented programs. For the microbenchmark, SQLite, and HawkNL, programs compiled with the `clang` compiler (without instrumentation) incur respectively 87%, 2727%, and 65% overhead compared to programs compiled using `gcc`. However, this high overhead is actually a constant factor, as these programs execute for less than a millisecond, and the overhead is negligible for programs which execute more than 100 milliseconds (e.g. Pbzip2). Therefore, we only compare the runtime overhead for Pbzip2 and httpd.

Table 3 reports the results. SP incurs less than 6% overhead. The overhead is due to the instrumentation and preemption overheads. CBD incurs more than 3300% overhead. The overhead is due to the instrumentation, maintaining the locking behavior, the delays before lock acquisitions, and the preemptions before memory accesses. Since the preemptions before all the write memory accesses account for most of the runtime overhead, we tested a modified version of CBD which, similar to the SP algorithm, preempts threads after lock releases, instead of memory accesses as these are considerably fewer than the memory accesses in the programs we tested. There was no change in the deadlock probability. However, the runtime overhead substantially decreased. The modified version incurs less than 21% overhead.

The overhead for httpd was approximately 37 times the overhead for Pbzip2 for SP and the modified version of CBD. This is due to the greater number of threads (32×) and lock acquire/release operations in httpd compared to Pbzip2. For the unmodified CBD, the overhead for httpd is more than 58 times the overhead for Pbzip2. For this case, in addition to the greater number of threads and lock operations, the greater number of memory accesses contribute to the difference in the runtime overhead.

5. DISCUSSION

The primary focus of the current work is increasing the probability of deadlocks, therefore, in some cases, Lockout incurs high overhead. To incur lower overhead, the *Simple Preemption* algorithm, can limit the number of preemptions to locks that are more likely to cause a deadlock. Moreover, the number of preemptions per lock or per thread can be limited. For the *Component Based Delays* algorithm, the delay time can be set more intelligently. One option is to determine it dynamically based on the gathered information regarding the target program. As mentioned, some data races are likely to be involved in data race dependent deadlocks. They can be identified via static analysis, and preemptions before memory accesses can be limited to only such accesses.

One of the major limitations of the current work is that it only targets deadlocks caused by locking orders. Generally, any synchronization between threads that has the potential to make any set of threads circularly wait for each other may lead to a deadlock. Moreover, the proposed technique can be used to trigger other types of concurrency bugs rather than merely deadlocks.

One challenge we encountered in the implementation of Lockout was the identification of the same mutex across multiple program executions. We needed to store per-execution information about locks and use this information in subsequent executions by recognizing to which mutex the stored information belonged to. For a globally defined mutex, we use the relative address in memory and identify the same mutex accurately. But for dynamically allocated mutexes, relative address cannot be used. Debug information such as the line number where the mutex is defined or the name of the mutex variable does not help either, as several unnamed mutexes can be dynamically allocated in the same line in a loop. While [14] proposes two heuristic techniques to overcome this challenge, there seems to be no trivial comprehensive solution for this challenge.

Although we focused at revealing more deadlocks in the testing phase, Lockout (specially with algorithms with lower overheads) can potentially be combined with automatic failure avoidance techniques (e.g [21, 15]) to provide faster and better resilience against deadlocks in deployed software.

6. RELATED WORK

A few prior systems perturb thread scheduling and increase the probability of deadlock manifestation.

DeadlockFuzzer [14] dynamically detects potential deadlocks and biases a random scheduler to increase the probability of those deadlocks. It works in two phases; in the first phase, it dynamically detects potential deadlocks, and in the second phase, its random scheduler takes as input a single deadlock cycle reported in the previous phase, and tries to increase the manifestation probability of that deadlock. Similar to our approach, DeadlockFuzzer tries to make threads circularly wait for each other. Instead of delays, it pauses threads involved in the input deadlock cycle. The threads will be paused until a real deadlock happens, or all the enabled threads become paused (*thrashing*), in which case it randomly un-pauses a thread and continues. DeadlockFuzzer may outperform our approach in the ability to increase the probability of a particular deadlock because it pauses all the involved threads until a deadlock or *thrashing* occurs, whereas our approach only delays threads for a limited amount of time. On the other hand, DeadlockFuzzer incurs high runtime overhead (approximately from 200% to 600% overhead for its second phase). Moreover, DeadlockFuzzer only targets a single deadlock per execution; it requires a dynamic deadlock detection phase; it does not use information from previous executions; and it does not target data race dependent deadlocks, though its random scheduler might be effective in triggering such deadlocks.

ConTest [8] is similar to our work in that, it preempts some threads before and after the accesses to shared memory and synchronization primitives. However, ConTest focuses on achieving enough coverage in thread interleaving space rather than targeting deadlocks.

PCT [22] uses a priority-based scheduler and assigns random priorities to threads at random synchronizations points in the program in order to increase the bug manifestation probability. PCT provides probabilistic guarantees for bug manifestation in each execution. However, it does not target any specific bug. In addition, it does not improve its probability guarantee using previous executions.

7. CONCLUSION

We presented Lockout, a technique and a tool that increases the probability of deadlock manifestation in a program, while preserving program semantics. Lockout requires no changes to the runtime and the testing infrastructure. Lockout works by altering the thread schedule to produce interleavings that are more likely to deadlock. We evaluated Lockout on a suite of multithreaded programs. Early results suggest that Lockout rises the probability of deadlock manifestation at reasonable overhead.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their invaluable feedback. This work was supported in part by ERC Starting Grant No. 278656.

9. REFERENCES

[1] HawkNL. http://hawksoft.com/hawknl.

[2] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2006.

[3] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Hardware and Software, Verification and Testing*, pages 208–223. Springer, 2006.

[4] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17(4):461–483, 2005.

[5] The Clang compiler. http://clang.llvm.org/.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT, 1999.

[7] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software: Practice and Experience*, 29(7):577–603, 1999.

[8] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM systems journal*, 41(1):111–125, 2002.

[9] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symp. on Operating Systems Principles*, 2003.

[10] J. Gilchrist. PBZIP2. http://compression.ca/pbzip2, 2013.

[11] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[12] Apache httpd. http://httpd.apache.org, 2013.

[13] G.-H. Hwang, K. chung Tai, and T. lu Huang. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 5:493–510, 1995.

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

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

[16] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced data race detection. In *Symp. on Operating Systems Principles*, 2013.

[17] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. In *ACM Symp. on Parallelism in Algorithms and Architectures*, 2008.

[18] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[19] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *Software Engineering, IEEE Transactions on*, 32(6):382–403, 2006.

[20] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *USENIX Annual Technical Conf.*, 2005.

[21] B. Lucia and L. Ceze. Cooperative empirical failure avoidance for multithreaded programs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[22] M. Musuvathi, S. Burckhardt, P. Kothari, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[23] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[24] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Intl. Conf. on Software Engineering*, 2009.

[25] SQLite. http://www.sqlite.org/, 2013.

[26] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *European Conf. on Object-Oriented Programming*, 2005.