TeamWork: Synchronizing Threads Globally to Detect Real Deadlocks for Multithreaded Programs

Yan Cai[†], Ke Zhai[‡], Shangru Wu[†], and W.K. Chan[†,*]

† Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
{yancai2, shangruwu2}@student.cityu.edu.hk
wkchan@cityu.edu.hk

‡ Department of Computer Science
The University of Kong Kong
Pokfulam Road, Hong Kong
kzhai@cs.hku.hk

Abstract

This paper presents the aim of TeamWork, our ongoing effort to develop a comprehensive dynamic deadlock confirmation tool for multithreaded programs. It also presents a refined object abstraction algorithm that refines the existing stack hash abstraction.

Categories and Subject Descriptors D.2.5 [**Software Engineering**]: Testing and Debugging – testing tools. D.4.1 [**Operating Systems**]: Processing Management –deadlocks, synchronization, threads.

General Terms Reliability, Verification

Keywords deadlock detection, object abstraction, thread scheduling

1. Introduction

Deadlocks in multithreaded programs are difficult to both expose and reproduce, and yet they are critical bugs that should be fixed. Existing dynamic detection techniques that rely on systematic thread scheduling only guarantee a low probability to trigger real deadlocks [2]. Dynamic randomized deadlock confirmation techniques [3][6] that attempt to heuristically trigger a real deadlock in a new run with respect to some predictive run show their promise to significantly improve the deadlock detection probability over the systematic thread scheduling approach. Nonetheless, the current generation of randomized deadlock confirmation actively manipulate thread schedules of an execution *only when* a thread is about to acquire a lock at a position (site) predicted to form a deadlock in a predictive run that has been run beforehand. These techniques have no ability to coordinate threads (that are predicted to form a deadlock occurrence) and yet deadlock is a *global* property that threads should be carefully synchronized (instead of in a randomized manner) to trigger them.

This paper reports the status of our ongoing project for the dynamic confirmation of *resource deadlocks* in large-scale multithreaded programs. This work has not been generalized to consider deadlocks that involve communication primitives (e.g., the *wait−notify* primitives in Java) or conditional variables (e.g., to implement user-defined synchronization idioms).

In a program execution, a resource deadlock occurs if every thread in a set waits for a lock that another thread in the same set holds. Existing predictive techniques usually either identify cyclic subgraphs in lock-order graphs [1] or infer cyclic dependency chains from a lock dependency set [3][6]. We refer to their reported cases as ***tentative deadlocks***.

Figure 1 is a resource deadlock example: Two threads t_1 and t_2 compete for two locks l_1 and l_2. A deadlock occurs when (1) t_1 has acquired l_1 at line 2 and requests to acquire l_2 at line 3 and (2) t_2 has acquired l_2 at line 9 and requests to acquire l_1 at line 10. On the other hand, if t_1 executes lines 1−6 followed by t_2 executing lines 7−12, then the execution produces no deadlock occurrence. Based on the second execution, deadlock prediction technique such as [3] is able to report a tentative deadlock in the form of cyclic lock dependency chain [3]: $\langle\langle t_1, l_2, \{l_1, m\}\rangle, \langle t_2, l_1, \{l_2\}\rangle\rangle$, which means that the thread t_1 requests the lock l_2 while holding the set of locks $\{l_1, m\}$ and the thread t_2 requests the lock l_1 while holding a set of locks $\{l_2\}$.

In the rest of this paper, we present a brief overview of our ongoing effort to trigger resource deadlocks hidden in multithreaded programs. It also presents an object abstraction algorithm that our framework is expected to use.

2. Our Framework

2.1 The Aim of the TeamWork Component

We are developing a framework to supports the dynamic detection of concurrency bugs in multithreaded programs. The first two components of the framework include a lock trace reduction component [4] for predictive data race detection [5] and a dynamic predictive deadlock detector [3] to produce tentative deadlocks. We are developing TeamWork as its third component, which is for dynamic confirmation of the tentative deadlocks if such deadlocks are real deadlocks. All these three components share the common theme that the casual dependencies among threads with respect to the threading and locking operations can be harvested to significantly improve the cost-effectiveness of the framework.

In this section, we outline our onging work on TeamWork.

We conjecture that in real-world applications, a program execution visits a series of critical states before reaching a tentative deadlocked state. We are investigating methods to identify such critical states. In general, a critical state over multiple threads forms a thread synchronization condition, which this paper refers to it as a *barrier*.

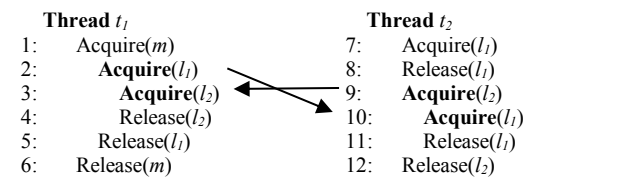

Figure 1. An example deadlock scenario (the deadlock is highlighted)

* contact author.

This work is supported in part by the General Research Fund of the Research Grant Council of Hong Kong (project nos. 111410 and 123512).

Once they could be identified, we plan to synchronize selected threads at selected barriers so that the operations performed by these threads could be executed by stages. TeamWork aims at escorting threads to concurrently pass through all the selected barriers before reaching the deadlocked state. If this is successful, TeamWork will be novel in its barrier-based strategy to confirm real concurrency bugs in general and real deadlocks in particular.

There are some considerations that we are studying for this component of our framework. For instance, a subset (possibly a singleton set) of all identified barriers can be used to synchronize threads at the same time. The same subset may be used multiple times to manipulate the same execution before the execution reaching the deadlocked state. Moreover, the number of threads selected among different occurrences of the same subset (or different subsets) of all the selected barriers can be non-identical. The thread selection strategy with respect to each barrier will be investigated. We will also study the complexity of large-scale real-world multithreaded programs to make TeamWork both scalable and effective to deal with this interesting class of program. As such, TeamWork is an umbrella of techniques instead of one technique.

2.2 Object Frequency Abstraction

Our framework relies on an effective but abstract representation to both effectively and efficiently models various entities (e.g., a program state needed in the barriers).

For dynamic confirmation of concurrency bugs, one key challenge is to compute an *object abstraction* [3][6] so that a thread or an event in a confirmation run is able to approximately map to the "same" thread or the "same" event occurred in another (e.g., predictive) run. An exact mapping may not be able to be developed.

We have developed an *instance* form of object abstraction [3][6][8] with a reference to *memory indexing* [7]. This form of object abstraction used the hash value for the program stack content (refer to as *stack hash*) [8] for matching efficiency. It also refines the precision of the existing proposal [3][6] by distinguishing the number of times that the same combination of a particular *thread* (and *lock*) *abstraction* (in the sense of existing form [3][6]) and a particular *stack hash value* has been used for the creation of the abstraction. We refer to it as the ***object frequency abstraction***:

- For a lock object or a thread object o, its abstraction $abs(o)$ is denoted by $\langle thread\_abs, call\_stack\_hash, newObj_{counter}\rangle$, where the couple $\langle call\_stack\_hash, newObj_{counter}\rangle$ is the site (i.e., $site(o)$) of the object.
- For a lock acquisition event e, the abstraction $abs(e)$ is denoted by $\langle thread\_abs, lock\_abs, call\_stack\_hash, acq_{counter}\rangle$, where the couple $\langle call\_stack\_hash, acq_{counter}\rangle$ is the site (i.e., $site(e)$) of the event.

In the above two abstraction definitions, *thread_abs* is the abstraction of the thread [3] to produce the *new object* or *lock acquisition* events, *lock_abs* is a lock abstraction [3], and *call_stack_hash* is the *hash value* [8] for the combination of call stack value and a program statement *stmt* being executed by the thread. Both $acq_{counter}$ and $newObj_{counter}$ are thread-local mappings from *thread_abs* and *call_stack_hash* and from *thread_abs*, *lock_abs*, and *call_stack_hash* to an integer, respectively. Algorithm 1 shows the algorithm to compute the object frequency abstraction for object and locking event creations. The object frequency abstractions for the other events and objects can be defined and computed similarly.

In the algorithm, each call to a function `getProgramCallStack(k)` at lines 3 and 8 returns the sequence of the top k (inputted at line 1) values of the program call stack (or the whole call stack if there are less than k values). Line 2 initializes the abstraction of the main thread (denoted by `main_thread`) by arbitrary values as its abstraction. The procedure `OnCreateAnObject` (lines 3–7) computes $abs(o)$ for an object o. It firstly computes the above-mentioned hash value, finds out the number of times that the couple of this hash value and the thread associated with o have previously been mapped via $Occurrence_{counter1}()$ (which is a map from the inputted pair to the occurrence times of this pair). It then constructs $abs(o)$. The procedure `OnAcquireALock` (lines 8–12) computes $abs(e)$ for a lock acquisition event e, which can be interpreted similar to `OnCreateAnObject`. The function $Occurrence_{counter2}()$ can be interpreted similar to $Occurrence_{counter1}()$.

Algorithm 1: Object Frequency Abstraction

Initialization

1: k := *input*(); //a user specified value, by default, it is 8 according //to [8].
2: abs (*main_thread*) := ⟨–1, –1, –1⟩;

OnCreateAnObject (Thread t, Object o, Statement *stmt*):

3: Vector st := *getProgramCallStack*(k);
4: st.push (*stmt*);
5: *call_stack_hash* := *hash*(st);
6: $newObj_{counter} = Occurrence_{counter1}(abs(t), call\_stack\_hash)$;
7: $abs(o) := \langle \boldsymbol{abs}(t), \boldsymbol{call\_stack\_hash}, \boldsymbol{newObj_{counter}} \rangle$;

OnAcquireALock (Thread t, Lock m, Statement *stmt*, Event e):

8: Vector st := *getProgramCallStack*(k);
9: st.push (*stmt*);
10: *call_stack_hash* := hash(st);
11: $acq_{counter} := Occurrence_{counter2}(abs(t), abs(m), call\_stack\_hash)$;
12: $abs(e) := \langle \boldsymbol{abs}(t), \boldsymbol{abs}(m), \boldsymbol{call\_stack\_hash}, \boldsymbol{acq_{counter}} \rangle$

Using Algorithm 1, in our experimentation, the executions for deadlock confirmation only encounters very few occurrences of object mismatches. We leave the report of the experimental results as a future work.

3. Conclusion

In this paper, we have presented the aim of TeamWork. We have also presented a refined object abstraction algorithm. Future work includes the formulation of the concrete strategy of TeamWork.

References

[1] R. Agarwal, L. Wang, and S. D. Stolle, 2005. Detecting potential deadlocks with static analysis and run-time monitoring. In *Proceedings of the 2005 IBM Verification Conference*.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of ASPLOS'10*, 167–178.

[3] Y. Cai and W.K. Chan, 2012. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *Proceedings of ICSE'12*, 606–616.

[4] Y. Cai and W.K. Chan, to appear. Lock trace reduction for multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems*.

[5] Y. Cai and W.K. Chan, 2011. LOFT: Redundant synchronization event removal for data race detection. In *Proceedings* of *ISSRE'11*,160–169.

[6] P. Joshi, C.S. Park, K. Sen, and M. Naik, 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of PLDI'09*, 110–120.

[7] W. N. Sumner and X. Zhang, 2010. Memory indexing: Canonicalizing addresses across executions. In *Proceedings of FSE '10*, 217–226.

[8] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, 2008. Deadlock Immunity: enabling systems to defend against deadlocks. In *Proceedings of OSDI'08*, 295–308.