View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Technical Disclosure Common

Technical Disclosure Commons

Defensive Publications Series

August 14, 2018

Detection of Deadlocks and Race Conditions in Computing Systems

Karthik Ravi Shankar

Follow this and additional works at: https://www.tdcommons.org/dpubs_series

Recommended Citation
Shankar, Karthik Ravi, "Detection of Deadlocks and Race Conditions in Computing Systems", Technical Disclosure Commons, (August 14, 2018)
https://www.tdcommons.org/dpubs_series/1407

This work is licensed under a Creative Commons Attribution 4.0 License.
This Article is brought to you for free and open access by Technical Disclosure Commons. It has been accepted for inclusion in Defensive Publications Series by an authorized administrator of Technical Disclosure Commons.

Detection of deadlocks and race conditions in computing systems

ABSTRACT

A race condition is a phenomenon wherein the output of an electronic device or computer process (thread) depends on the relative timing of events outside the control of the process or device. A deadlock is a state in which multiple computing processes that share a common resource are stalled due to the processes mutually locking each other out of access to the resource. Deadlocks and race conditions are difficult bugs to detect, or even reproduce for debugging.

This disclosure presents techniques that detect deadlocks and race conditions using machine-learning models to analyze the control flow graph of a program. Predictions of potential race or deadlock conditions are accompanied by justifications, e.g., potential scenarios that cause a race conditions or deadlock to arise. The classifying and generalizing abilities of machine-learning models are applied such that these difficult to detect bugs are caught at design stage, most advantageously for large code bases.

KEYWORDS

deadlock; race condition; race hazard; deadlock detection; machine learning; control flow graph; depth-first search

BACKGROUND

A race condition is a phenomenon wherein the output of an electronic device or computer process or thread depends on the relative timing of events outside the control of the process, thread or device. If the outside events do not occur in the sequence envisaged by the programmer, the behavior of a process or thread may become incorrect or non-deterministic.

In concurrent computing, a deadlock occurs when each member of a group of processes waits for another member of the group to take certain actions, e.g., sending a message or releasing a lock. Processes involved in a deadlock are unable to progress because the resources requested are held in mutual lock. Deadlock is a common problem in multiprocessing systems, parallel computing, distributed systems, operating systems, etc., where software and hardware locks are used to handle shared resources or synchronize processes.

Detecting deadlocks and race conditions is a difficult problem, one whose complexity increases exponentially with the size of the code. Detecting these bugs during testing is hard since they often manifest only when low-probability events occur in a particular sequence. For example, deadlocks and race conditions are sensitive to timing dependencies, workloads, the presence or absence of print statements, compiler options, slight differences in memory models, etc. This sensitivity increases the risk that these bugs elude pre-release regression tests yet manifest when the software is released to large numbers of users.

Further, even if a test case happens to trigger error, it can be difficult to determine from the test output that an error took place. For example, a race condition may violate data structures rather than cause a process crash, making it hard to observe. The effects of these data-structure violations may manifest substantially later, e.g., millions of cycles after the error occurred, making it hard to determine root cause in a test laboratory setting. Further, errors that occur in production code can disappear in debug mode, e.g., due to debug-instrumentation temporarily obscuring the race condition.

DESCRIPTION

This disclosure utilizes machine-learning techniques to detect race conditions and deadlocks statically, based on locks used, execution threads, memory accessed within lock scope, etc.

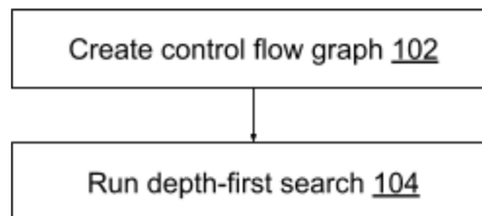

Fig. 1: Detecting deadlocks and race conditions

Fig. 1 illustrates the detection of deadlocks and race conditions, per techniques of this disclosure. A control flow graph (102) is created from a source code base. A control flow graph (CFG) is a directed acyclic graph of the code; locks and resources that are shared between threads of the code are represented in the CFG. A depth-first search (DFS) (104) is run on the CFG to check if any two sections, e.g., code pathways or threads, intersect for any interval of time.

Threads or paths within the code that intersect in time are a signature of a potential race condition. The DFS also checks for commonality, e.g., simultaneous or mutual demand for shared resources, which is a signature of a potential deadlock. Cyclic sub-graphs within the CFG, e.g., wherein a thread A requests a resource locked by a thread B, and thread B requests a resource locked by thread A, are an indication of a potential deadlock. In this manner, the DFS identifies threads or code pathways that converge on a single resource.

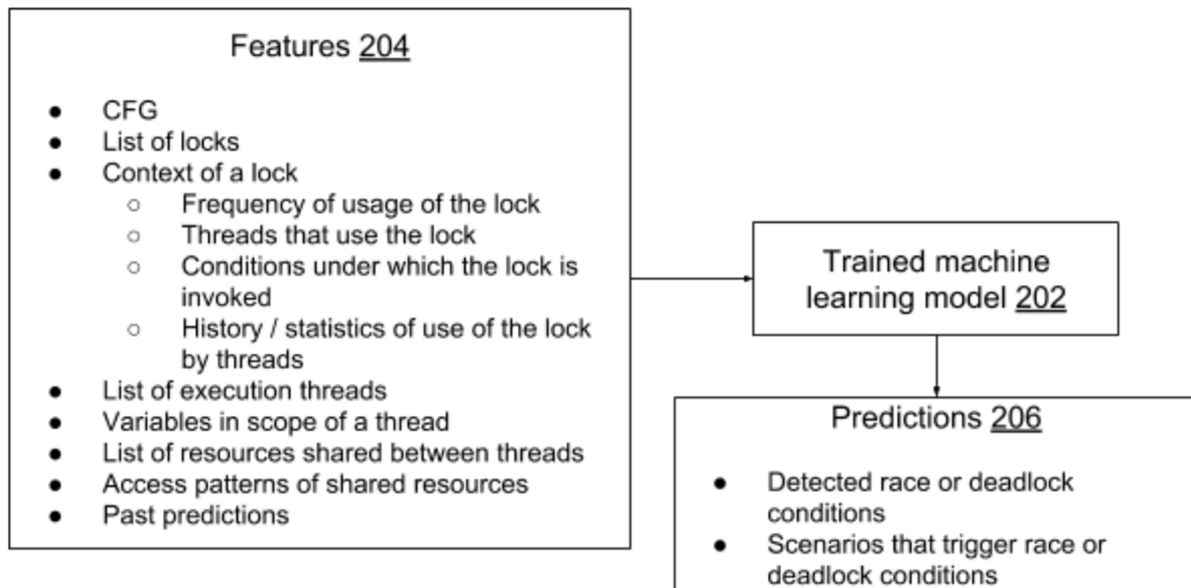

Fig. 2: Use of machine learning to detect race or deadlock conditions

The number of threads that can potentially intersect in time, or the numbers of shared resources that can potentially be mutually locked by threads, is relatively large in any code base of practical significance. The number of possibilities for potential race and deadlock conditions increases exponentially with the size of code base. Per techniques of this disclosure, the DFS make advantageous use of machine learning techniques, as illustrated in Fig. 2, to prune the search space. At each step of the DFS, a trained machine-learning model (202) accepts as input a number of features (204) of the CFG and makes predictions (206) regarding potential race or deadlock conditions.

Examples of features that are provided as input to the machine-learning model include:

- the CFG;
- a list of locks within the CFG;
- context of each lock, including:
    - frequency of usage of the lock,

    - threads that use the lock,
    - conditions under which the lock is invoked,
    - history or statistics of the use of the lock by threads;
- a list of execution threads;
- variables within the scope of a thread;
- a list of resources shared by threads;
- access patterns of shared resources, e.g., threads that use a lock, or access a common memory location, or other common resource;
- past predictions; etc.

The predictions of the machine-learning model include a list of detected race or deadlock conditions, and also scenarios, e.g., particular sequences of events, that likely trigger such conditions. Justification or reasoning for the identification of race conditions is also provided as an output.

The trained machine learning model can be, e.g., regression learning models, neural networks, etc. Example types of neural networks include long short-term memory (LSTM) neural networks, recurrent neural networks, convolutional neural networks, etc. Other machine learning models, e.g., support vector machines, random forests, boosted decision trees, etc., can also be used. Reinforcement learning can be used to advantageously incorporate past predictions as feedback to improve machine-learning performance. The machine-learning model is trained with known pairs of input features and optimal predictions. For faster and energy-efficient inferences, the machine-learning model can be advantageously run on special-purpose hardware, e.g., GPU, DSP, processors or co-processors optimized for machine learning, etc.

The techniques of this disclosure train machine-learning model(s) and apply the trained models to detect deadlock and race conditions in a static manner, e.g., by analyzing code at design or compile time, without having to execute code. Machine learning is used to advantageously prune the number of race or deadlock possibilities to be checked, thereby enabling relatively large code bases to be assessed for race or deadlock conditions. Along with identification of potential race or deadlock condition, the techniques provide reasoning for such identification, e.g., a possible sequence of events that leads to race or deadlock. This enables a developer of the code to make informed decisions. The techniques are advantageously applied as a pre-check in a back-end tool prior to submission of code to a code base.

CONCLUSION

This disclosure presents techniques to detect deadlocks and race conditions by making use of trained machine-learning models to analyze the control flow graph of a program. Predictions of potential race or deadlock conditions are generated by the trained models, and are accompanied by justifications, e.g., potential scenarios that give rise to races or deadlocks. The classifying and generalizing abilities of machine-learning models are applied such that these difficult to detect bugs are caught at design stage, most advantageously for large code bases.