# Fault Localization in Concurrent System

IN4MATX 215

Milestone 1 – Literature Review

Team Member:

Chiyu Cheng

Yi Zhou

# Introduction

Testing is one of most critical and expensive tasks in the development of software. The process of detecting and locating bugs in software system is called fault localization. Manually fault localization is acceptable for the small software system but unbearable for the large-scale system. Therefore, an automated fault localization technique is necessary for the real-world complex software system. A concurrent system is one where a computation or operation can be executed without waiting for all other operations to complete. It generally involves the coordination of multi-thread/process/cores to run a program. The concurrency fault is difficult to find because a concurrent system may have large number of threads interleaving, which makes it is hard to reproduce a fault in one specific thread interleaving. The research question we have is, is it possible to locate the concurrent bugs in the large-scale system efficiently? In this project, we want to develop an evaluation framework to study and evaluate the current state of art in finding the concurrency fault. The plan is surveying 8-10 papers in this topic and evaluating the methods provided in those papers under the same standard. The basic criteria are feasibility, efficiency, and accuracy. We will test if each method is feasible and record the performance of each method on the real system like the SPLASH-2 programs from Stanford university with concurrency bug.

Based on the paper we read, the current state of arts in the field of concurrent fault localization are focused on the memory-access pattern, variable access pattern, random execution between interleaving, and etc. The brief summaries of each paper present in the next section.

# Paper Summaries

In order to solve concurrency bugs, we should identify what is a concurrency bug.
Eitan et al. present and categorize a taxonomy of concurrent bug patterns [6]. For a run of a given concurrent program, its interleaving is the sequence of concurrent events that occurred in this run. Thus a concurrent program P is associated with a set of possible interleavings I(P), i.e., the set of interleavings that can occur in some run of the concurrent program P. One way of viewing a concurrent bug pattern of a given concurrent program P is to look at the relation of the space of possible interleavings, I(P), to maximal space of interlevings for P under which the program is correct, C(P). A concurrent bug pattern can be viewed as defining interleavings in I(P) – C(P).

He then defines three concurrent bug patterns:

1. A bug pattern occurs when a code segment is assumed to be protected but is actually not. In practice this bug pattern occurs in many flavors such as nonatomic operations, wrong lock or no lock, double-checked locking.
2. Another bug pattern is interleavings assumed never to occur which means the programmer assumes that a certain interleaving never occurs because of the relative

execution length of the different threads, or because of some assumptions on the underlying hardware.

3. "Blocking" critical section bug pattern, which means a thread is assumed to eventually return control, but it never does. This situation may occur in a critical section protocol.

There are also many frameworks are prompted to test concurrent and the key methods which those frameworks are based on can be divided into two categories: heuristic algorithms or active testing.

E. Farchi et al [7]. used to prompt ConTest timing heuristics which are written to increase the probability that known kinds of concurrent bugs will occur. At that time ConTest is passed various kinds of information such as the name of the statement being executed or a reference to the variable being accessed. In addition, sometimes ConTest is given control of whether to invoke the concurrent event at all or override it by a different, semantically equivalent operation. Thus, ConTest architecture can be viewed as an instance of the filter design pattern. At runtime, when control is passed to ConTest before and after a concurrent event is executed, ConTest uses Java primitives such as sleep () and yield() to change the order of concurrent event execution. It follows that a timing heuristic is a description of code segments executed before or after a concurrent event is encountered and a statement of whether the original concurrent event is executed.

The idea of another way is quite intuitive that is trying to systematically explore all thread schedules when such approach is done it will give very high confidence in the correctness of the system. Whereas, there is an obvious problem with such approach for the number of thread schedules is nearly infinite when you come to a large-scale distributed system. In order to solve such problem, many interesting algorithms have been prompted.

Joshi et al. prompt a framework called CALFUZZER which is based on active testing [8]. Active testing first uses an existing predictive off-the-shelf static or dynamic analysis, such as Lockset, Atomizer, or Goodlock, to compute potential concurrency bugs. Each such potential bug is identified by a set of program statements. For example, in the case of a data race, the set contains two program statements that could potentially race with each other in some execution. For each potential concurrency bug, CALFUZZER runs the given concurrent program under random schedules. Further, active testing biases the random scheduling by pausing the execution of any thread when the thread reaches a statement involved in the potential concurrency bug. After pausing a thread, active testing also checks if a set of paused threads could exhibit a real concurrency bug. For example, in the case of a data race, active testing checks if two paused threads are about to access the same memory location and at least one of them is a write. We do believe the idea of active testing might be work for our project.

Another idea, given by Koushik Sen from UC Berkeley can be concluded as random partial ordering algorithm or RAPOS [10]. Koushik finds that traditional random scheduling approach often explores some states with very high probability compared to the others. And the RAPOS can partly removes this non-uniformity in sampling the state. According to his result, RAPOS can find bugs more quickly if bugs exist in partial orders and concurrency related bugs often exist in partial order in a multi-thread system.

One thing which makes testing concurrent program quite hard is called “Heisenbugs”, that occasionally surface in systems that have otherwise been running reliably for months. Sometimes, even a slight change to a program can drastically reduce the likelihood of erroneous interleavings, adding frustration to the debugging process. Fortunately, the state-of-art work based on the idea of active testing called CHESS can solve the problem quite well.

Musuvathi et al. from Microsoft implements a tool called CHESS in order to find and reproduce such Heisenbugs in concurrent programs [9]. The key idea of CHESS is like active testing in concurrent program. When attached to a program, CHESS takes control of thread scheduling and uses efficient search techniques to drive the program through possible thread interleavings. When CHESS reaches a statement, which might have potential bugs it will stop and check whether such interweaving of threads could exhibit a real concurrency bug. Besides, another advantage of CHESS is it could provide custom schedulers for active testing which allows users to specify their own arbitrary schedules.

There are also some techniques to test concurrency program:

In Dr.Park’s paper[1], he provides an dynamic fault-localization technique that extends the functionality of traditional method from sequential and deterministic program to the multi-thread program. This basic idea of this technique is detecting the data access patterns in multi-thread concurrent programs. It monitors the memory access sequence in different thread interleaving to find the data access patterns and mark it with fail or success that associates with the result of program tests. Based on the data access patterns collected, the technique generates a suspicious score for each pattern to help programmer to find concurrent bugs.

In the paper “Cooperative Crug Isolation” [2], the authors provides a scalable technique to locate the root of the concurrent bugs with low-head. They use a novel approach named cross-thread random sampling strategy to reduce the overhead of this method and track successive accesses to a memory location to keep recording the occurrence of specific interleavings at run-time. With the collection of those shared variable access, programmers can design the predicates to check whether a previous variable access is thread-local or thread-remote. Based on the fail/success result of those predicates, CCI method generates the suspiciousness score of each predicate.

In Stoller’s paper [3], he presents a novel random scheduling function to test it on a Java program. The scheduling function use heuristic to weight the random generated choice and find the choice that leads to all the reachable deadlock and assertion violations with Non-Zero probability.

In Musuvathi’s paper [4], they proposes a new algorithm that systematically search and iterate the executions of multi-threaded program with different priority order based on the number of context switch. They detect and distince the difference between preempting and non-preempting context switches of each execution, and find that the limitation of the number of preempting context-switch mitigates the explosive possible states of multi-thread executions.

In Ken’s paper [5], he proposes a randomized dynamic analysis technique with a focus on concurrent faults caused by data race condition. This technique gets the possible data race condition from an analysis tool and use it in a random scheduler of threads to create real race

conditions and resolve it in the runtime. With this technique, programmers are be able to find the real race condition on their program with low over-head.

# Reference

[1] Park, S., Vuduc, R., and Harrold, M. J., “Falcon: Fault localization in concurrent programs,” in Proc. of Intl Conf. on Softw. Eng, pp. 245–254, May 2010.

[2] A Thakur, R. Sen, B. Liblit, and S. Lu. Cooperative crug isolation. In WODA, pages 35–41, 2009.

[3] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In Proc. Second Workshop on Runtime Verification (RV), volume 70(4) of Electronic Notes in Theoretical Computer Science. Elsevier, July 2002

[4] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. PLDI., 42(6):446–455, 2007.

[5] K. Sen. Race directed random testing of concurrent programs. In PLDI, 2008.

[6] Farchi, Eitan, Yarden Nir, and Shmuel Ur. "Concurrent bug patterns and how to test them." Proceedings international parallel and distributed processing symposium. IEEE, 2003.

[7] E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. Java Grande/ISCOPE 2001 Special Issue of Concurrency and Computation: Practice and Experience, 2001

[8] Joshi, Pallavi, et al. "CalFuzzer: An extensible active testing framework for concurrent programs." International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, 2009.

[9] Musuvathi, Madanlal, et al. "Finding and Reproducing Heisenbugs in Concurrent Programs." OSDI. Vol. 8. 2008.

[10] Sen, Koushik. "Effective random testing of concurrent programs." Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007.