A. Motivation

Concurrency bugs are one of the hardest to detect and most prevalent in software systems today (20% of driver bugs examined in a previous study [10] are concurrency bugs). They are pernicious to the extent that they have caused real world disasters such as the 2003 Northeast blackout [9].

And today with the ubiquity multicore machines, this becomes an ever more pressing problem. While reliability and debugging has made significant steps in sequential software [4], [2], and some have even shown steps to put scheduling in user control [6], the prospect for bug free concurrent code still remains bleak. The complexity lies in part in the idea of interleavings.

Interleavings are not only complicated to reason about heuristically, but also impossible to reason about holistically because of the dramatic increase in the state space.

In general, one can devise the research into concurrency bug detection into two categories: symptom based detection, and invariant based detection. [1]

Current symptom based techniques to address this problem have relied on overarching structural patterns of interleavings and zoomed in on the ones most probable to cause problems. The patterns include race conditions [5], atomicity violations [7], and context-switch bounded interleavings [3].

While these techniques do exist, they leave us with substantial problems, most important of which are: False Negative and Positives. [11]

Current invariant based approaches have their own strengths and weaknesses. Proposals like AVIO [7] detect atomicity violation bugs, and find interleaving in supposedly atomic instructions. Recently, works like Vulcan [8] focus on detecting sequential consistency violations, and the idea that some memory operations get preformed in a simply unintuitive manor. It then flags these violations and presents invariants.

Existing invariant based proposals generally work by collecting a large number of execution traces by using various test inputs either written before by the developers or explicitly for the trace. Then they analyze the traces and extract the required invariants. For this reason they suffer from four problems.

First, they cannot test all the possible thread interleavings, thus miss invariants and contain false negatives. Second, the proposals replicate the production run environment, thus a lack of diversity of tests. Third, as the program is extended, many of the invariants become invalid, and if run again, they loose information from the previous runs that had not been invalidated. And fourth, they cannot check the correctness of an invariant if it is not found during the extraction process.

Our project seeks to address these concerns and augment current techniques.

B. Ideal Goal

In our project we seek to prove that a machine learning algorithm can be used to detect and avoid a number of these concurrency bugs. These will include bugs that would not be caught by using read after write dependencies as the only invariant. In the cases where it cannot avoid the bug, we hope to provide the developer with program level invariants that will help direct the coding process and ensure reliability.

We will achieve this by creating a continuously learning neural network to verify invariants dynamically. We will provide a statistical and theoretical framework for interpreting our results and, ideally, will complete our paper with a study of concurrency bugs in the wild.

C. Minimal Goal

While we have lofty ambitions, we must remember that Burns poem "To a Mouse". Our minimal goal will be to replicate the results of Alam et. alius [1], on a few "toy programs," We will add to their theoretical framework and add a statistical interpretation to the results.

We will not get time to add different factors to the code, some of which might be: accounting for locks, non-atomic operations, and some types of shared resources.

One thing that worries us is the potential for a complete lack of applicability. While the use of simulation will allow us to more freely explore different invariants and learning algorithms, as well as avoid difficult low-level implementation issues, it also makes our work more theoretical - transitioning to a practical implementation will likely be extremely difficult, and there is the possibility that our project will not be able to find any concurrent bugs in practice.

Another potential issue is the possibility that the approach is not as extensible or valid as we currently believe. The Alam paper has only a very limited "preliminary results" section in terms of evaluation. While the results achieved are promising, they are also very non-descript. They describe using a PIN based tool to collect execution traces for data, but that is all the specificity they provide.

D. What technologies do you plan to use to accomplish your project's goals?

We plan to implement our project in dynamically typed language for easier work with interleaving. And because of our familiarity with python and in order to take advantage of the large number of machine learning libraries that are available, it seems to be the language of choice. The Pylearn2 library in particular has a variety of different types of neural networks that could potentially be applied to this problem - we will likely begin by using a multi-layer perceptron. We also plan on, at least initially, simulating buggy concurrent programs
in Python rather than writing them in C. This is perhaps the largest departure our work takes from that done in [1] - rather than implement a neural network at the hardware level, we plan to simulate the environment. This will allow us increased flexibility in our choice of invariants and our choice of learning algorithm.

E. What paper or papers would you like us to read relating to your paper?

Our project will be largely stemming from the work done in [1]. However, this may not be the best paper to read in class as there is little in terms of evaluation. It also focuses more heavily on the choice of read after write dependencies as an invariant for concurrency bugs rather than the use of invariants as a method for catching errors. Instead, we suggest reading a related paper about invariants, such as AVIO [7].

F. What is your one-week project?

Our one-week project is to write a simulation of a simple, buggy, concurrent program, generate a number of execution traces for the program, and then train an out-of-the-box multilayer perceptron to detect and avoid this simple bug. This is in line with our minimal goal of replicating the results from [1].

G. What is your collaboration plan?

This project will be split into two components. The first is the creation of an interface for the learning algorithm - this will involve the simulation of concurrent bugs. The second is the learning algorithm itself. We will begin by each taking one of these two aspects. Both aspects can be iterated on a number of times, as there are many different concurrency bugs, possible invariants and learning algorithms that we can investigate.

REFERENCES