Learning? An Examination of Data Based Concurrency Bug Detection

Jefferson Lee, *jeffersonlee@college.harvard.edu,* Korey N. Tucker, *tucker@college.harvard.edu,*

***Abstract*—In this paper we propose and study various machine learning approaches to detect bugs in concurrent programs. We present two different machine learning approaches and a non-learning approach and their performance measured across two vectors: operational overhead and the ability to classify correctly various bugs. The first approach seeks to use a machine learning model in an online fashion, to learn invariants from unlabeled executions. The second learns invariants in an offline fashion with executions being labeled. Our third and final approach uses a simple model on higher level features calculated on communication graphs. We test these models on a series of injected bugs and find the simpler model on better features to be more impressive. We propose future modifications to our final approach and list the approach's shortcomings.**

I. 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 [15] are concurrency bugs). They are particularly difficult to diagnose and fix for two reasons. First is that developers must keep in mind and reason about the interactions of many pieces of code that is often executing in many threads. By simply observing one thread, you may be unable to identify the cause of the bug. The second reason is non-determinism. In multithreaded environments the exact behavior of an application may vary from run to run (even from one buggy run to another). This can make it very difficult to pin-point and reproduce the bug. Thus, some developers faced with a concurrency bug have just given up and others apply incomplete solutions [16]. As a result, these bugs remain free and unmanaged in the wild, and have caused real world disasters such as the 2003 Northeast blackout [14].

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.

II. Related Work

In general, one can divide the research into concurrency bug detection in three categories: symptom based detection, invariant based detection, and dynamic bug avoidance. [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 [10], and context-switch bounded interleavings [3].

While these techniques do exist, they leave us with substantial problems, most important of which are: false negatives and positives, and heavy operation overhead. [18] Some approaches have sought to remedy this with sampling techniques to reduce the overhead, but again you are left with some problems [9].

Current invariant based approaches have their own strengths and weaknesses. Proposals like AVIO [10] detect atomicity violation bugs, and find interleaving in supposedly atomic instructions. Recently, works like Vulcan [12] 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 lose 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.

The third type of concurrency bug detection has been dynamic bug avoidance. Some work proposes architecture support for dynamic bug avoidance like the Interleaving Constrained Multiprocessor, PSets [17].Other work like Falcon, gets a stream of of memory operations to single addresses and looks for a matching pattern [8]. In some cases interthread communication invariants have also been used to specify correct communication at the function or method level.

III. Introduction

We explore machine learning based solutions for this problem. We explore a variety of features based on labeled executions of concurrent programs and combinations thereof. And we explore different models to identify and isolate concurrency bugs, and provide simple and useful bug reports.

Our models range from simply feeding traces to a multilayer perceptron, to developing complicated factors based on the interleaving schedule surrounding shared-memory communication. They do not look for specific patterns, therefore do not make assumptions about the nature of bugs.

Our solutions begin when a programmer observes a bug or receives a bug report. The programmer will then derive a test case designed to trigger the bug and then will run the

program several times under our approach. We collect and record summary information about each run including the label of each run as buggy or non-buggy. We compute features and use them to determine the location of the bug.

We make several contributions:

- We hypothesize and explore various factors and approaches to using Machine Learning to detect concurrency bugs.
- We test these approaches using the same standards and propose an optimal approach.
- We report performance over two vectors: overhead and accuracy of classification

The rest of the paper is organized as follows. In section four we discuss concurrency bugs and give an example. In section five we discuss the data that we extract from our executions and how we represent it. In section six we discuss the features that we used in our models. In section seven we discuss the machine learning models that we used. In section eight we discuss implementation of both models. In section nine we discuss how we evaluated the models. And in section ten and eleven we discuss our conclusions and future work.

IV. Bugs

A. Concurrency Bugs

These bugs can be loosely classified into three different categories: data races, ordering violations, and atomicity violations.

Data races happen when two different thread access the same memory location. One of the threads must be a write. And both threads' accesses are not ordered by synchronization.

In order violations two or more memory access from multiple threads happen in an unexpected order. This can be due to either absent or incorrect synchronization.

Finally, atomicity violations result from a lack of constraints on the interleavings of the program. For example, consider the simple atomicity violation presented in Figure 1. In this piece of code we are trying to find the reciprocal of a global variable and store it back in the variable. However, consider the example where thread 1 performs the function `global_reciprocal` and thread 2 performs the function `to_zero`. Thread 1 can get to line 3, and check to see that `global_var` is not set to zero. It can succeed and more to line 6. However between its operation on line 3 and 6, thread 2 can run. It can set `global_var` to 0, and then the program will throw a division by zero error.

```
1
2  void global_reciprocal (void* arg) {
3    if (global_var != 0)
4    {
5      // then set our reciprocal
6      global_var = 1/global_var;
7    }
8
9  }
10
11 void to_zero (void* arg){
12
13   global_var = 0;
14
15 }
```

Fig. 1. Atomicity Violation

V. Data Representation

A. Traces and RAW dependencies

In order to simulate the type of online system seen in the work of Alam et. al, we generated full traces of all memory operations in the main function of the program. These operations were then parsed in order to detect read after write dependencies or RAW. If a write instruction writes to a memory location, all read instruction that read what was written at that location can be said to be dependent on that instruction.

Most programs will have many of these types of dependencies. If we could enumerate all dependencies that could occur during a correct execution of a program, it would be simple to determine if a read after write occurred in a buggy way. However, enumerating all possible dependencies is infeasible - rather, we only observe the dependencies that occur during a given program run, and use machine learning to label them as buggy or non-buggy.

Using RAWs is also advantageous in the context of hardware-based solutions, as tracking read after writes at the hardware level is relatively simple compared with other possible invariants. Alam et. al also seem to suggest that incorrect RAWs are correctable - namely,they suggest flushing the read from the pipeline and re-executing it, repeating the process as necessary until an acceptable RAW is achieved.

However, there are a number of issues with this data representation, at both a practical and theoretical level. From a practical standpoint, instrumenting every read and write to any memory location is very computationally intensive. This is particularly true not only for generating traces, but any processing or computation that must be carried out on these traces after the fact. From a theoretical standpoint, RAWs may not be an appropriate data representation due to the fact that most RAWs will not be relevant to concurrency bugs. For example, a RAW that occurs from within the same thread is very unlikely to be related to a concurrency bug. Another concern is that these dependencies represent very little information. There is no context attached to the information - one could imagine a scenario in which a RAW is safe when it occurs at a certain point in a programs execution, and buggy when it occurs in another. In order to address this issue, we use the last five RAWs as a state representation for the program's execution. One imagines, however, that at least the thread id would be a necessary feature for a machine learning approach.

B. Communication Graphs

In prior work communication graphs have proved useful for detecting concurrency bugs. A communication graph is a set

of nodes $v \in V$ and a set of edges $e \in E$ [16]. A node v in a communication graph represents a memory instruction at some point in the program. An edge from a source to a sink represents the sink read or write on data written by the source. Both the sink and the source must be in different threads.

In such a way we have mapped the complicated interleavings of a program to the edges and nodes of a graph. In a concurrent program bugs then can often be encapsulated by a single edge in a graph. Therefore a common technique that can be employed is targeting edges that occur frequently in buggy graphs and infrequently in a normal run [17].

For an example consider our atomicity violation Figure 1. Then the edge that would represent the bug would most likely be the edge from an instruction on line 13 to an edge to line 6.

However with a quick check to ourselves we can understand why this approach can only capture a subset of bugs. And bugs that depend on two separate data could not be caught using this sort of a graph. Therefore when building features, we will need more information. Thus we move to a aware communication graph.

C. Aware Communication Graph

An aware communication graph was first proposed by Bugaboo (another machine learning approach to detect concurrency bugs) to solve this problem exactly. The communication context C is a bounded history of events of the set: local reads, local writes, remote reads, remote writes. Where a local event is a thread manipulating its own local data. And a remote event is a thread manipulating some sort of global data. And finally I will note that context is a property of threads, and thus a history is associated with a particular thread.

In an aware communication graph a node is a pair (I, C), where C is the context previously defined, and I is the most recent instruction in context. Again edges between a source and a sink represent the sink reading or writing data that was written by the source. Note that there can be more than one node per instruction (because the instruction was carried out with various contexts). That being said, the size of the graph is bounded by the number of nodes times the bounded size of C.

While an aware communication graph captures more information than does a non-aware one, there are still some artifacts of the code and the execution that cannot be expressed using a bounded context. The final modification that we will present is a timestamped aware communication graph.

D. Timestamped Aware Communication Graph

This type of graph allows for one more augmentation. Each node is now labeled with a timestamp. In such a way we encode an ordering between non-communicating nodes. Note that this does not increase the complexity of a graph. The nodes will be the same and are identified by (I, C), however each node is labeled with a timestamp that represents the global time when instruction I was last executed with context C.

This implementation looses some data, because old timestamps are overwritten, but overall carries more information

Using this modification, we are able to translate execution traces to a data structure that is able to convey much more information. That being said, this representation almost has too much information. We therefore propose one final modification to these graphs, a sparse timestamped aware communication-graph (STAC).

E. Sparse Timestamped Aware Communication Graph

One of the problems using the full communication graph structure operational overhead. Constructing such graphs can increase the overhead on a normal execution by 100 fold [11]. This is often prohibitive. Thus we propose STAC.

We implemented two optimizations: first read only and first write only.

For the first read only optimization, we assumed that repeated reads to a memory location by the same thread are most likely redundant. In this optimization threads only record their first read to a location after a remote write to that location. Thus even if a thread reads from an updated location many times, it will only record it once.

This optimization, because of the frequency and temporal locality of reads, eliminated many updates.

For the first write optimization we implemented a similar procedure. A thread will only record a write to a memory location if another thread was the last writer of that location.

VI. Features

We can then use STACs to create factors. Remember what we are trying to predict. If we can predict which edge of the STACs is buggy, we can then narrow the bug down to specific memory operations in a specific context.

What is very important is that the features are general. A feature that targets only one specific type of bug or pattern is only so useful. The tool should be able to predict any type of bug. Previous works have focused much more on using specific features to detect one type of bug like atomicity violations [13].

Thus our features should try to capture as much information as possible in a parsable form. The three features that we will study are: the buggy frequency ratio, the context variation statistic, and the reconstruction consistency ratio.

A. Buggy Frequency Ratio

The buggy frequency ratio, BFS, measure the frequency of specific edges in STACs made by buggy executions, vs. their frequency in non-buggy executions. One can expect that edges that only and always occur in buggy executions are highly suspect when it come to identifying the cause of the bug.

Formally we define the BFS as a ratio of ratios. Define $F_b(e)$ to be the percent of buggy runs which edge e occurs in. And define $F_n(e)$ as the percent of non-buggy runs which edge e occurs in. To ensure that percent is not zero we calculate it with the formula:

$$\frac{R_b(e) + 1}{R_b + 1}$$

Where R_b is the number of buggy runs and $R_b(e)$ is the number of buggy runs that contain edge e. Therefore we can define BFS as:

$$BFS(e) = \frac{F_b(e)}{F_n(e)}$$

Thus you can see that if an edge occurs in lots of buggy runs and few non-buggy runs, then it will have a high BFS, and vice-versa.

B. Context Variation Statistic

The context variation statistic, CVS, is the ratio of context variations between buggy and non-buggy executions. One can imagine that an edge that has nothing to do with the bug, will have similar number of contexts between buggy and non-buggy executions. However if buggy executions have many contexts and non-buggy executions have very few, one could expect these deviations to have cause the bug. Or if non-buggy executions have many contexts, and buggy only has a few, one could expect that in this limited number of contexts, bugs exist.

This is formally calculated by the normalized absolute differences in the number of contexts between buggy and non-buggy executions. Let I_v and I_u be two instructions. Let $C_b(I_v, I_u)$ be the number of distinct contexts in which the pair communicated during buggy executions. Let $C_n(I_v, I_u)$ be the number of distinct contexts in which the pair communicated during non-buggy executions. Therefore we have:

$$CVS(I_v, I_u) = \frac{|C_b(I_v, I_u) - C_n(I_v, I_u)|}{C_b(I_v, I_u) + C_n(I_v, I_u)}$$

C. Aggregate Reconstruction Consistency

To define aggregate reconstruction consistency, ARC, we must build an aggregate reconstruction. For any given edge e, a reconstruction is the tuple (e, p, b, s). Where $e = (u, v)$ is the edge going from the source u to the sink v. Where p is a set of fixed length L of nodes in timestamp order immediately preceding u. Where s is a set of nodes of fixed length L in timestamp order immediately following v. And b is the set of nodes between u and v.

An aggregate reconstruction for a set of graphs is the tuple (e, p, b, s). Where $e = (u, v)$ is the edge going from the source u to the sink v. Where p is the union of all prefixes of the reconstruction of e from all the graphs. Where s is the union of all suffixes of the reconstruction of e from all the graphs. And where b is the union of all between nodes of the reconstruction of e from all the graphs.

This means that p, s and b can have duplicate nodes and their intersection might not be empty.

Then the idea behind ARC is that we focus on the buggy runs. We then look for edges that always seem to have the same prefix, suffix and between with little variation, and we assume they are buggy.

Let P_e be the nodes in the prefix region of the aggregate reconstruction around e. And let S_e and B_e be the set of nodes in the suffix and between regions of the aggregate reconstruction around e respectively. Let $F_e(v, R)$ be the number of times that node v happens in R divided by the number of times v happens in the aggregate reconstruction. Therefore:

$$ARC(e) = \frac{\sum_{v \in P_e} F_e(v, P_e) + \sum_{v \in B_e} F_e(v, B_e) + \sum_{v \in S_e} F_e(v, S_e)}{|P_e| + |B_e| + |S_e|}$$

Edges with high ARC values are likely to represent problematic interleavings.

D. Efficacy of Features

Prior work has shown the efficacy of these features using Weka's ReliefF [7]. The magnitude of a feature's ReliefF value is greater is the distance between the different classes (in our case buggy and non-buggy) is greater along that feature's dimension. It was shown that all features had a positive (sometimes near one) ReleifF across a variety of applications [11].

VII. Machine Learning Models

A. Multilayer Perceptron (Online)

A multilayer perceptron is a neural network model that maps sets of input data onto an appropriate set of outputs. A multilayer perceptron, MLP, consists of many layers of nodes in a directed graph with each layer fully connected to the next one.

The first layer of nodes represent the set of inputs and the last layer represents the outputs. The middle layers are referred to as hidden layers.

Each node aside from the inputs are neurons or a nonlinear activation function acting on the inputs, or all the nodes from the previous layer. It could easily be seen that if the nodes had linear activation functions then the entire network could be reduced to just two layers: inputs and outputs. However, some nodes have nonlinear activation functions (just like the firing of neurons in the brain).

MLPs use sigmoid functions as their nonlinear activation functions that range from 0 to 1:

$$\sigma(v) = (1 + e^{-v})^{-1}$$

Where σ is the output of our neuron and v is the weighted sum of the input synapses where each synapse (edge between nodes) has a weight w_{ij}.

Learning in an MLP is done through backpropogation. Let $e_j(i)$ be the error of the jth output node on the ith data point. The error function is then:

$$E = \sum_i \sum_j e_j^2(i)$$

This function is minimized (we in particular used stochastic gradient decent) by finding the optimal value for the weights w_{ij}.

Our initial design used this model to continuously learn and verify, as in the work of Alam et. al. However, there are a number of issues with this approach. Given the online nature in which this approach was framed, each read after write dependence detected during the learning phase will be unlabeled. The suggestion by Alam et. al is to assume, due to the nature of concurrent programs, that all of the read after writes we detect are valid. While this may not be problematic in many scenarios, we now have the issue of providing negative data - that is, false read after write dependencies - to our multilayer perceptron. The suggestion here is that we use the write before the last write, if the read does not dependent on it, as a negative example. However, this involves tracking all possible writes upon which the read may depend, in order to determine if the previous write could be used to generate a negative example. Therefore, this methodology was quite computationally expensive and seemed to perform poorly in practice.

B. Random Forest Classifier (Online and Offline)

Random Forests are an ensemble learning method. They generate many decision trees during training and output a classification (or a mean prediction) by taking a vote from all of the decision trees.

A decision tree is a flow-chart like structure, where each internal node represents a test on an attribute, each branch represents the outcome of the test, and each leaf represents the final classification. A tree can be learned through a recursive partitioning of the source set into subsets. Each internal node poses the set a binary attribute question that splits the subset into two groups. The recursion ends when all nodes in a set contain the same classification value. This is made a leaf.

Decision trees are classically known to be very unbiased and have high variance. That is where random forest come in.

A random forest uses a collection of decision trees. Its classification is a simple majority vote by the decision trees.

Learning a random forest is the similar to learning a decision tree. The random forest samples with replacement subsets of the source set. It then feeds these sets to the trees (one per tree). With this method, one can reduce variance without affecting expected bias.

In our model we used 10 trees and the scikit learn classifier.

VIII. Implementation

We implemented two versions of a machine learning approach for detecting bugs, a simpler version using traces fed into a Multilayer Perceptron and a version that extracts features out of the traces through graph construction.

A. Multilayer Perceptron (online)

The creation of a true online solution in the form of hardware is outside the scope of this project (outside of the scope of the paper of Alam et. al, as well). Instead, we simulated program execution by generating traces of all memory operations using the Pin tool, recording the type of memory operation (read or write), the instruction pointer, and the affected address to the trace. We then wrote a Python script to calculate all dependencies found in each trace, and write the dependencies (and their associated states) to a csv, along with a label. For each "true" dependency, we attempt to create a false one as well by using the previous write, as suggested in the paper.

This data was then taken as input by a multilayer perceptron created in Pylearn2. We used stochastic gradient descent as our learning algorithm, and, rather than alternating between learning and verification, have a single learning phase, ending when a threshold of 0.75 was achieved. This choice of threshold is discussed in the evaluation portion of the paper.

B. Random Forest Classifier (online)

Eventually, the multilayer perceptron was abandoned for our online model in favor of a random forest classifier. The large number of parameters and the sort of black-box nature of the multilayer perceptron made the model extremely difficult to tune. We used the random forest classifier found in Scikit-learn, formatting the data as with the multilayer perceptron, again with a single learning phase ending when we label at least 75% of the RAWs as valid.

C. Random Forest Classifier (offline)

In the offline version of our learning approach, traces are labeled as buggy or not based on their return value. We label all dependencies within a buggy run as buggy. We train on 80% of the dependencies generated, and validate our model on the remaining 20%.

D. Feature Based Implementation

1) Storing Metadata: Graphs are constructed by keeping a record for each memory location. In the record we store: (1) the timestamp of the last write, (2) the thread T that made the last write, (3) the node associated with that last write (I, C), and (4) a list of threads that share the resource S.

Each thread also has its own data store. Each thread carries with it the context it currently represents C.

When a thread writes to a memory location, it updates that location with its thread ID, the instruction address of the write (I), the threads current context and the current timestamp.

If the thread that wrote has a different thread ID than T, the one currently associated with the memory location, then we do four things: (1) update the current thread's context with a local write event, (2) update the context of each thread on the sharing list S with a remote write event, (3) clear the sharing list, and (4) add the edge associated with this write.

When a thread whose ID is different from T reads a location it does four things: (1) updates its context with a local read, (2) updates the context of thread with ID T with a remote read event, (3) add its ID to the sharing list S, and (4) adds the edge associated with this read.

This metadata tracking was implemented through a hash table within our Pin tool. As in the original Recon paper, our hash table is lossy and does not handle collisions.

For accesses to the memory location table we allow some potential concurrency bugs. To ensure proper behavior when accessing and writing this table we would require each thread to obtain a lock, however the cost of acquiring the lock is often greater than the check itself. Therefore we allow data race on this table. These races occur when checking whether data should be written - for example, checking if the last writer was of the same thread. When it is determined that we should write, we lock the table. Because these checks are relatively fast and the consequences of overwriting are somewhat small, there are very little inconsistencies [11].

2) Graph Construction: Each thread maintains its own partial STAC. In such a way adding an edge to the communication graph is a thread local operation that has high performance. One caveat is that whenever a thread seeks to add edge, it must check if the edge already exists. If it does, it simply overwrites the existing timestamps with the new ones being added. When the thread ends it merges its partial graph with a global graph.

3) Generating Reconstructions: As discussed earlier, we rank the edges seen in these graphs using a number of different metrics, the primary being the Buggy Frequency Ratio. After we choose an edge as the best candidate for causing a bug, we examine the timestamps associated with the source and the sink of the edge. We then find nodes whose time stamps fill in a window before, between, and after these time stamps. After collecting the nodes for each of these windows, we choose those nodes that appear in those windows with the highest frequency.

IX. Evaluation

We used two basic examples in our evaluation. The first was the atomicity violation showed earlier, and the second a program with two threads incrementing a shared counter. Two, slightly more complex examples - a dot product program and a matrix multiplication program - were also created using C's OMP library. In all cases, we generate fifty traces.

We ran tests on the efficacy of each approach and on the overhead of each.

A. Random Forest Classification (online)

1) Atomicity Violation: We began by generating 50 traces for the atomicity violation. Of these 50, 19 were incorrect. These were used to generate 9507 "correct" dependencies and 2808 "incorrect" dependencies, which were then used to train our random forest classifier. It took 59.785 seconds to generate these traces, and 237.347 seconds for the entire learning process. However, when we attempt to use a threshold of 0.95 as suggested in Alam et. al, we find that our classifier never achieves this rate of classification. Here we see the importance of the assumption that most dependencies will be correct, and the unsatisfying nature of using a threshold to determine the behavior of our system. In fact, our system never labels significantly more than 79% of the RAWs as safe - this makes sense, given that roughly 77% of the data we provide it during training is labeled as safe. However, even given more appropriate thresholding - around 75% - the system is still unable to label any of the RAWs as unsafe.

It is perhaps worth noting that this simple atomicity violation problem should be solved quite easily by any system correctly learning RAW invariants. There are very few options for RAWs, none are particularly context specific, and they are all very easy to recognize as either safe or unsafe behavior.

The argument could be made that our choice of model is influencing the results. This may be true to some degree - however, such an incredible failure on such a simple program suggests not a failure with a model but with our methodology. The predictive power of a random forest classifier is not this significantly worse than that of a neural network.

Given the inability to learn anything at all from this simple example, we opted instead to move towards an offline procedure.

B. Random Forest Classification (offline)

1) Atomicity Violation: In our offline version of the process, we collect our fifty traces and train a model on the first forty. Then we attempt to use this model to identify buggy read after writes by running on the remaining ten traces. The generation of the traces took about the same amount of time - 63.564 seconds - but the learning portion occurs much faster, as it can be done in bulk, leading to a total time of 65.863. This method proved less useful simply due to the sheer amount of output - more than fifty read after writes were determined as buggy, many erroneously.

C. Recon

1) Atomicity Violation: As in the earlier examples, we generated 50 traces of the atomicity violation example. It took 51.187 seconds to generate these traces, and 51.991 seconds for the entire feature calculation and reconstruction process. As noted earlier, the output traces in this scenario are far more succinct than in the online learning case, and it thus takes far less time to process them.

This system is able to isolate the issue almost immediately. No reconstruction is created or provided due to the nature of the program, as the bug is a very obvious read from one thread after a write from another.

It's worth noting that only seven buggy execution were created out of fifty. Due to the nature of the Pin tool, it appears that more heavy instrumentation - say, at the level of every read and write as in the last approaches - will cause more failures.

2) Shared Counter: It took 48.472 seconds to generate the traces, and 48.515 seconds for the entire process. During the collection of fifty traces, only two buggy executions were collected.

Given the simplicity of this problem, our implementation of Recon was still able to correctly isolate the correct buggy edge - that one thread was overwriting the other thread when it incremented the shared counter, without any intermediary reads.

3) Further Evaluation: Given the relative promise of our implementation of Recon compared to our other approaches, we decided to evaluate it further. We injected a bug into a program to calculate the dot product of two vectors, implemented using

C's OMP library. Our system took 117.254 seconds to generate the traces and 118.516 seconds to complete the reconstruction.

Unlike in the previous examples, there are now a number of different memory locations of interest - namely, every element of each vector. This throws a significant amount of noise into the data. Our reconstruction is not particular helpful in this scenario, as all reads and writes simply point to the same line of code, and are not labeled by type or by thread id. Some of this information may still be useful for debugging purposes, but it is far less clear than in the simpler examples.

X. Conclusion

Concurrency bugs are extremely difficult. Even when using sophisticated approaches and nearly all the data about the program's run, it can still be hard to find them.

Perhaps one thing that we learned from the project most of all is that one cannot apply machine learning naively to all sets of problems. Features and feature selection is very important, and as much work should be done to isolate the relevant data before learning as possible.

The conclusion of the paper is that feature extraction through STACs is more predictive than feeding RAW dependencies into a MLP and more computationally tractable. We were able to more accurately detect a program's bug and give a more useful report. Perhaps the model in conjunction with a more feature heavy approach might yeild similar results, however it seems like further goals will be more focused on reducing rather than adding complexity.

XI. Future Work

Our future work will be divided into two parts: one part on current limitations, and another on potential augmentations.

A. Current Limitations

One of the biggest limitations of this approach is that we need to notice the bug and be able to reproduce it. If we cannot reproduce it, then we will not have any training data to feed our models, and our approach will be effectively useless. Even more so, we require the developer to find the bug and generate test cases that can classify it into buggy and non-buggy executions. Overall the programmer is still left with a sizable amount of work that s/he needs to implement before s/he can even use this technique.

Another issue is the need for generating both successful and failing executions. Twenty-five of each are used in the original Recon paper - however, generating twenty-five failing executions may not be feasible in many situations. Consider, for example, that it took roughly a minute to create one buggy execution for our shared counter example. This is an extremely simple program, and it still may take upwards of 18 minutes to generate the necessary traces.

B. Potential Augmentations

There are three areas of potential work that we could see being very fruitful. First is to classify the bugs that we found. One of the boons to the approach is that we use general methods to find the bugs in our concurrent program. We give the user a summary bug report which contains communications in and around the communication that we believe to be the bug. However the user of the problem does not know what type of concurrency bug this is. Perhaps using machine learning to classify the bug into either a data race, atomicity violation or order violation might make debugging even easier.

Another potential improvement that we could make to the system is to introduce further sampling. We current use a STAC, a sparse representation of all the data available. It has been shown to be just as effective as using the full representation, because we mostly removed spurious edges. We could consider doing further sampling to further improve our runtime while leaving the accuracy unaffected.

And finally what would be the most titanic improvement that would could make is hardware support. The biggest way to improve our speeds and overheads is through hardware. This would be by far the largest but perhaps most beneficial of the future work options.

References

[1] Mejbah Ul Alam et al. "Concurrency Bug Detection and Avoidance Through Continuous Learning of Invariants Using Neural Networks in Hardware". In: ().

[2] Thomas Ball et al. "Thorough static analysis of device drivers". In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 4. ACM. 2006, pp. 73–85.

[3] Sebastian Burckhardt et al. "A randomized scheduler with probabilistic guarantees of finding bugs". In: *ACM Sigplan Notices*. Vol. 45. 3. ACM. 2010, pp. 167–178.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[5] Jong-Deok Choi et al. "Efficient and precise datarace detection for multithreaded object-oriented programs". In: *ACM SIGPLAN Notices*. Vol. 37. 5. ACM. 2002, pp. 258–269.

[6] Kevin Elphinstone and Gernot Heiser. "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 133–150.

[7] Mark Hall et al. "The WEKA data mining software: an update". In: *ACM SIGKDD explorations newsletter* 11.1 (2009), pp. 10–18.

[8] Christian Hammer et al. "Dynamic detection of atomic-set-serializability violations". In: *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE. 2008, pp. 231–240.

[9] Guoliang Jin et al. “Instrumentation and sampling strategies for cooperative concurrency bug isolation”. In: *ACM Sigplan Notices*. Vol. 45. 10. ACM. 2010, pp. 241–255.

[10] Shan Lu et al. “AVIO: detecting atomicity violations via access interleaving invariants”. In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 5. ACM. 2006, pp. 37–48.

[11] Brandon Lucia, Benjamin P Wood, and Luis Ceze. “Isolating and understanding concurrency errors using reconstructed execution fragments”. In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 378–388.

[12] Abdullah Muzahid, Shanxiang Qi, and Josep Torrellas. “Vulcan: Hardware support for detecting sequential consistency violations dynamically”. In: *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*. IEEE. 2012, pp. 363–375.

[13] Sangmin Park, Richard W Vuduc, and Mary Jean Harrold. “Falcon: fault localization in concurrent programs”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM. 2010, pp. 245–254.

[14] Kevin Poulsen. “Software bug contributed to blackout”. In: *Security Focus* (2004).

[15] Leonid Ryzhyk et al. “Dingo: Taming device drivers”. In: *Proceedings of the 4th ACM European conference on Computer systems*. ACM. 2009, pp. 275–288.

[16] Yao Shi et al. “Do I use the wrong definition?: DeFuse: definition-use invariants for detecting concurrency and sequential bugs”. In: *ACM Sigplan Notices*. Vol. 45. 10. ACM. 2010, pp. 160–174.

[17] Jie Yu and Satish Narayanasamy. “A case for an interleaving constrained shared-memory multi-processor”. In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 325–336.

[18] Wei Zhang et al. “ConSeq: detecting concurrency bugs through sequential errors”. In: *ACM SIGPLAN Notices*. Vol. 47. 4. ACM. 2011, pp. 251–264.