Extract Useful Features for Detecting Transient Faults

Zhiqiang Sui, Zhefan Ye, Karthik Desingh

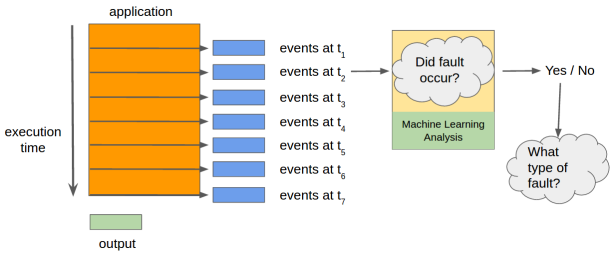

Figure 1: Overview

ABSTRACT

1. INTRODUCTION

In this report, we present a pipeline that can inject faults into programs and analyze how injected fault can alter program outcomes. To detect transient fault has been a research topic in robust system design for years. We utilize gemFI to study how faults are going to affect programs. Our contribution is two-fold: we implemented a fault inject pipeline, and provide analysis on how to predict fault and program outcome based on a machine learning approach. Figure 1 shows the overview of our approach.

2. RELATED WORK

3. APPROACH

In this section, we present our implementation for injecting faults, as well as analysis for predicting program outcome and fault types.

3.1 Pipeline

Figure 2 shows the pipeline.

3.2 Fault Injection

GemFI [1] supports two kind of architectures: ALPHA and X86. In our project, we choose ALPHA as our test architecture. In order to inject faults into the test program, it needs to be modified to initialize the fault injection. The target application is cross-compiled by the alpha gnu toolchain and linked with FI libraries. Afterwards, the generated binary should be moved to the disk image serving as the virtual disk of GemFI. The specific faults are provided by the user in a file at command line. Each line of the input file specifies the attributes of a single fault. Faults are characterized by four attributes: Location, Thread, Time and Behavior.

An example below shows the a sample input of faults.

```
RegisterInjectedFault Inst:2457 Flip:21 Threadi
```

This example describes a fault that injects into the 21*st* bit of the register R1 of the CPU, when the application fetches 2457*th* instruction after the initiation of fault injection. For our current tests, we just generate faults by randomly sampling from the number of instructions of the test program, the components of ALPHA and the operations needed. However, this method couldn't cause the program to generate false results some time as different programs make use of different components at different time. So we will explore the running characteristics of each test program and generate faults from a finer and smaller space.

3.3 Test Bench

We use MiBench [2] as our test bench for injecting faults and evaluating our system. MiBench is a representative embedded benchmark. We choose MiBench because of its compact program size and versatile program characteristics. We choose qsort as our test program.

3.4 Feature Extraction

We extract all the common features from the *stats* files generated by GemFI. Since some fault injected program will generate fewer features than non-fault injected program, we only choose those the features appeared in both programs.

3.5 Machine Analysis

We use the random forest algorithm to predict program outcome and fault type. A random forest is essentially an ensemble of single decision trees, as illustrated in 4 [3]. It captures different models of the data, with each decision tree representing a model, and allows us to analyze the importance of different features.

3.5.1 Training and Testing

We randomly select 60% of instances for training and 40% of instances for testing. Our dataset consists of 98,000 data instance.

4. EXPERIMENT

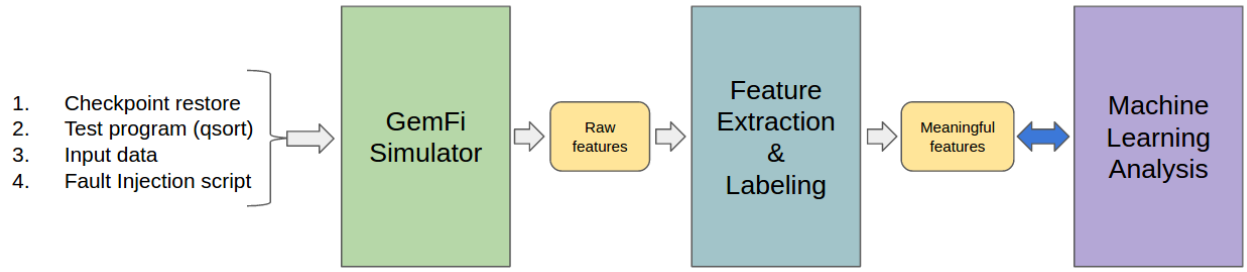

Figure 2:

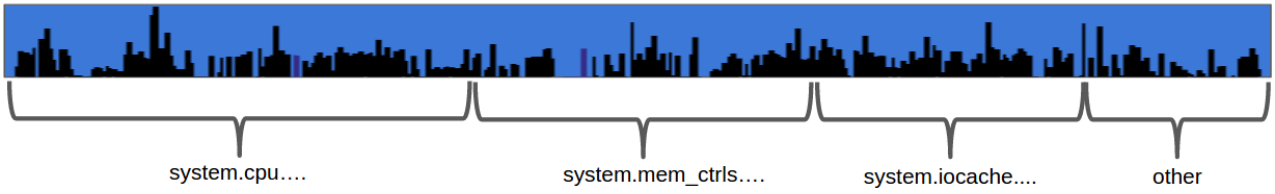

Figure 3:

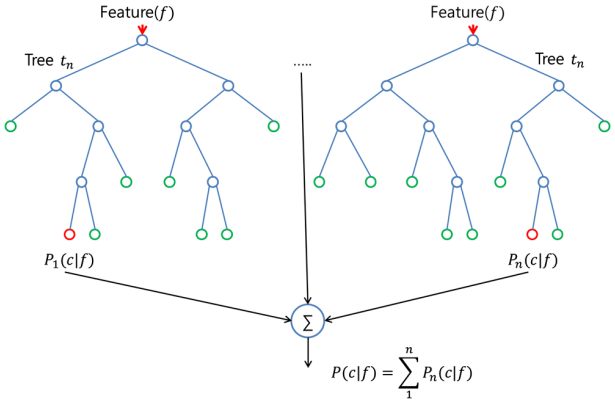

Figure 4:

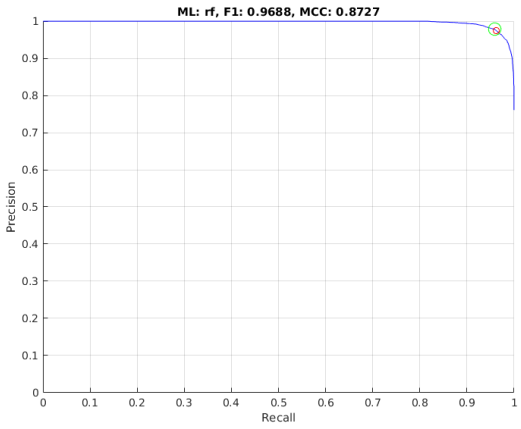

Figure 5:

4.1 Metrics

we use Precision-Recall (PR) curve and F_1 score as our evaluation criteria. More precisely, we have

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}, \quad (1)$$

where $precision = \frac{tp}{tp+fp}$, $recall = \frac{tp}{tp+fn}$, tp is the number true positive samples, fp is number of false positive samples, and fn is the number of false negative samples. For multi-class classification, we use confusion matrix to describe the performance of our classifier.

4.2 Experiment Setups

Here are our four experiment setups: 1) same input data with all features, 2) same input data with handpicked subsets of features, 3) different input data with all meaningful features, and 4) different input data with handpicked subsets of features.

5. RESULT

5.1 Same Input All Features

(same input all features SIAF)

The random forest algorithm output the importance score of each feature based on its discriminative power. The importance feature ranking is depicted in Figure 7.

5.2 Same Input Different Features

(same input handpicked features SIHF)

5.3 Different Input All Features

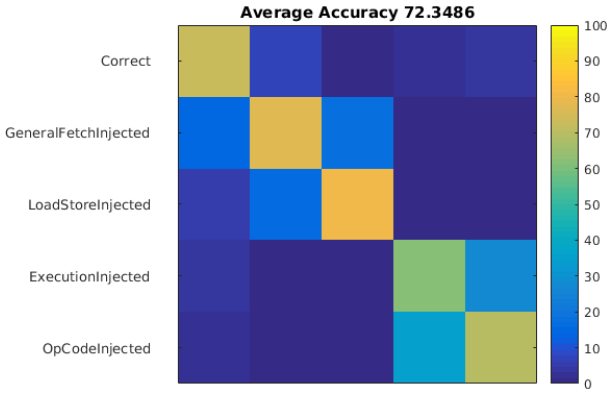

Figure 6:

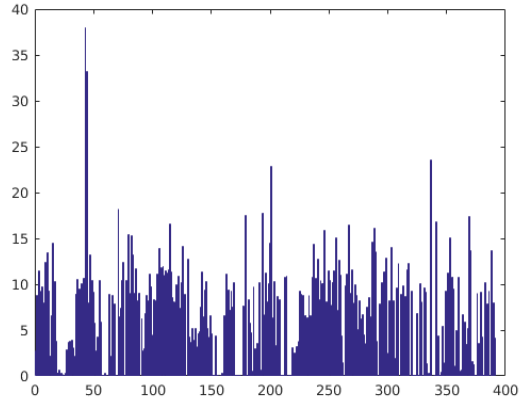

Figure 7:

(different input all features DIAF)

5.4 Different Input Different Features

Different Input Handpicked Features DIHF

5.5 Cross Application Predication

6. DISCUSSION AND CONCLUSION

In the same input scenario, a few features alone can determine the results in same input condition. For same input the most important features are mainly describing the execution length. For d different input scenario: the features such as, âĂIJtype of functional units issuedâĂİ, âĂIJfetch instructionsâĂİ and âĂIJcache read and writeâĂİ, play important role in the prediction. Overall features based on âĂIJL2 cacheâĂİ events perform higher than other set of hand-picked features. However, when all the features are used, the performance of the random forest algorithm is high. We extract meaningful fault signatures from the architectural events that can be used to predict transient fault occurrence.

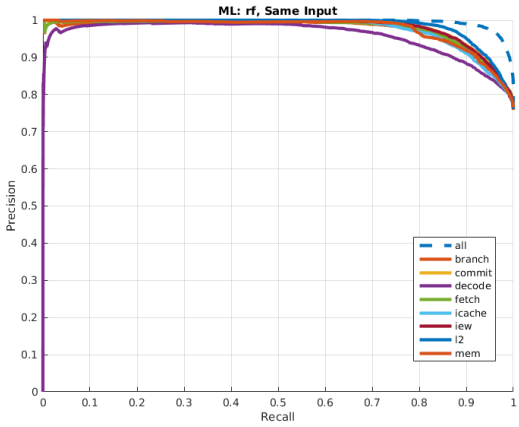

Figure 8:

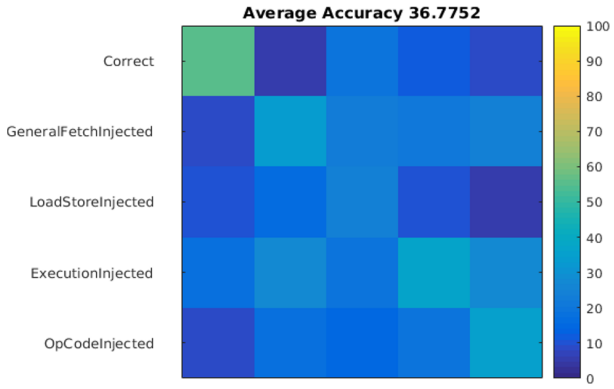

Figure 9:

7. REFERENCES

[1] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 622–629, IEEE, 2014.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pp. 3–14, IEEE, 2001.

[3] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

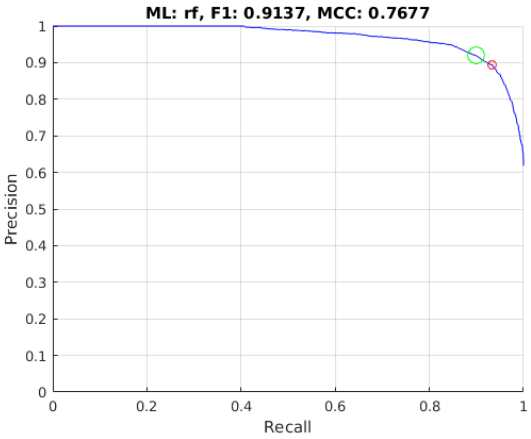

Figure 10:

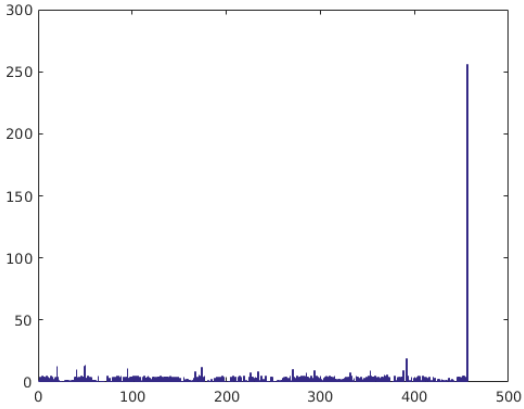

Figure 11:

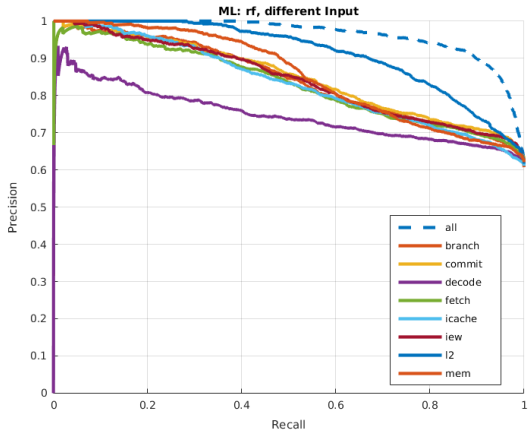

Figure 12: