Fault Prophet: A Fault Injection Tool for Large Scale Computer Systems

by

Tal Tchwella

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 2014

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
May 23, 2014

Signature redacted

Certified by ....
Prof. Martin C. Rinard
Thesis Supervisor

Signature redacted

Certified by ....
Stelios Sidiroglou-Douskos, Research Scientist
Thesis Co-Supervisor

Signature redacted

Accepted by ....
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Fault Prophet: A Fault Injection Tool for Large Scale Computer Systems

by

Tal Tchwella

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I designed and implemented a fault injection tool, to study the impact of soft errors for large scale systems. Fault injection is used as a mechanism to simulate soft errors, measure the output variability and provide analysis of the impact of soft errors on the program. The underlying framework for the tool is based on LLFI, a LLVM fault injection tool, which I modified to support an end-to-end scenario for program testing purposes. The modifications and addition provide greater modularity of the tool by abstracting the input and output of the different components of the tool, support multiple fault scenarios and models, and supply an extensive visualizations framework. I evaluated the effectiveness of the new tool based on a set of benchmark programs as well as showcased the impact of soft errors on programs. The results demonstrate that while the sensitivity of instructions is program dependent, certain instruction opcodes are generally more sensitive than others, such as binary and memory operations, however well placed protection mechanisms can decrease the sensitivity of those instructions.

Acknowledgments

This work is dedicated to Oded 'Akram' Tchwella and Ida Tzouk.

First and foremost, I would like to thank my parents, Yaron and Michal, and my sisters, Shir and Ophir, for their love, support and their presence in times of difficulty. I would like to thank my grandparents for their love and appreciation throughout my life.

I would also like to thank my family abroad, the Aharoni family, the Youssefmir and Youssefzadah families, and the Zinger family, as well as the people who provided me with a home away from home, Joshua Blum, Orit Shamir, Itai Turbahn, Tom and Anat Peleg, Naor Brown, Jason Black and Edan Krolewicz. I would like to seize the opportunity to thank my friends in Israel for standing by me these past four years, showing me what true friendship is all about.

It is with great pleasure that I acknowledge Professor Martin C. Rinard for inspiring and motivating me to do brilliant things. It is also with immense gratitude that I acknowledge the support and help of Michael Carbin, Sasa Misailovic, and Dr. Stelios Sidiroglou-Douskos throughout this process, for believing in me and providing me with this opportunity to work with them.

In addition, a thank you to the Dependable Systems Lab in University of British Columbia for allowing me to use LLFI as an underlying framework for my research. A special thank you to Sara Achour of MIT and the Accelerator Architecture Lab, Intel Labs for providing me with fault models used by Fault Prophet.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Errors may affect the result that the program produces. In particular, *soft errors* are a type of error caused by uncontrollable events: cosmic rays hitting memory cells, voltage droops, missing deadlines, etc. [1], and as such they are hard to identify, they occur non-deterministically, and may silently corrupt computation. Understanding these types of errors is especially important for systems consisting of hundreds of thousands or millions of processors [2, 21]. In this thesis I present Fault Prophet, a system for measuring, analyzing, and visualizing the impact of soft errors on the behavior and output of a program.

1.1 Fault Prophet

Fault Prophet enables developers to reason about the impact of soft errors on a program using three components: a fault instrumentor, a sensitivity measurement tool, and an error impact visualizer.

1.1.1 Fault Instrumentor

The Fault Instrumentor component enables Fault Prophet to intelligently select and inject faults, controlled errors, to various execution points in a program. The component profiles user defined instructions, which it then instruments in order to find the potential execution points for fault injection. Based on the instrumentation output, a histogram of the program is created to randomly select instructions that are on the critical path of the program flow. Faults are then injected to those instructions, and the output is saved to measure the sensitivity of those instructions to the faults. Fault Prophet builds upon LLFI [24], LLVM [9] Fault Injection Tool, created by the Dependable Systems Lab at University of British Columbia, as the underlying infrastructure, providing an improved fault selection mechanism, as well as several fault scenarios and fault models empowering the developers to extensively test the fault tolerance of a program.

1.1.2 Sensitivity Measurement Tool

The sensitivity measurement tool enables to measure the variation in output a fault caused when compared to the baseline execution. The baseline execution is the control group in the experiment, producing the standard output of the program based on the normal flow of operation. The amount of variation in output as a result of the injected fault is defined as sensitivity.

There are two main methods to measure the sensitivity of the computation a fault induces:

- Compare the flow of the program (trace of instructions) to a baseline trace.
- Compare the output to a baseline output.

While using both methods will provide a comprehensive level of detail of the effect of the fault, for this fault injection tool, the sensitivity is measured using the output comparison method. The reason behind this choice, is that the analysis and comparison of traces is resource intensive, both for processing and storage. However, the tool does provide a way to output traces for later analysis if needed.

1.1.3 Error Impact Visualizer

Fault Prophet visualizes the impacts of the soft-errors simulations enabling the developers to quickly and easily deduce the critical sections in a program. Critical sections are portions of the program that have an impact on the integrity and reliability of the program, if modified would cause a deviation in the expected output. Fault Prophet classifies a fault into one of the fault categories based on the sensitivity of the instruction the fault was injected to. The developers can classify the faults according to their own output abstraction methodology. Fault Prophet aggregates the results of the various injected faults for each source line, function, and the entire program, which it then displays using different visualizations. Fault Prophet maps the outcomes of the fault injections to the source code, enabling the developer to view the impacts in a popular source code editor, as well as inject faults directly from the editor and view the results from there.

1.2 Usage

The idea of a fault injection tool is motivated by the concept that not all parts of a program are created equal: some parts are more critical to a program's reliability and correctness than others. Protecting these critical sections is essential, as many programs are resilient to errors: after experiencing soft errors they may continue, executing and producing acceptable exact or approximate, results, often without any guidance from the recovery mechanisms. The existing recovery mechanisms for soft error: Checkpoint / Restart [3], Triple module redundancy [12, 14], and Instruction level replication [7, 18, 16], are often not used, due to the large overhead to the energy, the performance, and the cost of the systems they add [3].

Fault Prophet would enable a new class of recovery and protection mechanisms which are selective in their nature to the critical section of the program, instead of protecting the entire program. The selective protection would reduce the overhead costs associated with existing recovery mechanisms, thereby making it more feasible to ensure a reliable and correct flow of the program.

1.3 Contributions

Fault Prophet has several contributions:

- Implemented a modular fault injection framework.
- Provided support for multiple fault scenarios.
- Created a visualizations framework supporting an interactive work environment
- Tested the framework on several benchmark programs.

The contributions extend the capabilities of existing fault injection tools, by helping developers and testers alike, to identify critical sections under various circumstances in a program, via an interactive visualizations framework.

Chapter 2

Related Work

Multiple fault injection tools have been developed over the years, most with emphasis on software faults, while others simulated hardware faults as well. The earlier fault injection tools, such as FERRARI [8], a flexible software based fault and error injection tool, were designed to emulate both transient errors and permanent faults in software.

As system developers and researchers alike, recognized the potential fault injection tools has to validate the fault tolerance of a system, the features were enhanced beyond the realm of fault tolerance to performance testing as well. FTAPE, Fault Tolerance And Performance Evaluator, is a tool that compares the performance of fault-tolerant systems under high stress conditions [25].

Recent developments in fault injection research enabled targeted software injection faults. New tools use static analysis alongside heuristics to identify locations in code that are highly error prone [23]. Other tools, such as SymPLFIED, Symbolic Program Level Fault Injection and Error Detection Framework [17], focused on uncovering undetected errors via random fault injections by using symbolic execution. BIFIT, Binary Instrumentation Fault Injection Tool [10], on the other hand enables a user to evaluate how soft errors impact an application in specific execution points and specific data structures. While BIFIT does provide an analysis on fault injection, and is geared towards extreme scale systems, it focuses only on silent data corruption (SDC) outcomes of general soft errors via the injection single random bit flips as faults. In that respect, BIFIT is limited to a specific set of fault scenarios, and testing and analyzing those specific kinds of faults.

While hardware injection faults have been around for a while [4], simulation of hardware faults has not advanced as much, until recently. Relyzer [5] was one of the first tools in recent years to put an emphasis on hardware faults. Relyzer deviates from the standard software fault detection tools, detection of software corruptions, to the detection and monitoring of silent data corruption (SDC), enabling corruption detecting at a lower level.

LLFI [24], LLVM Fault Injection tool, is a recent fault injection tool that is developed by the Dependable Systems Lab at University of British Columbia. LLFI is used for high level fault injection, mainly to test SDCs for computation errors. LLFI works in three steps:

1. Injects function calls to find potential fault injection candidates.
2. Instruments the application for execution points for fault injection.
3. Randomly selects an execution point, and injects a fault.

LLFI can inject faults into instruction buffers as well as and operands, while picking the fault injection method via a fault factory. However, LLFI does not provide a choice of fault scenarios, and injects faults randomly. If LLFI did have that detailed knowledge on the frequencies of instructions, it could intelligently inject faults to random execution points of prioritized instructions. Moreover, LLFI does not provide a visualizations framework to analyze the outcomes of the fault injection pass, but instead compares the fault output to a baseline output, and as such, uses an internal output analysis mechanism that might not be suitable for all applications.

Another recent tool is ASAC [20], Automatic Sensitivity Analysis for Approximate Computing, which analyzes the fault tolerance of approximation based programs. Approximation based programs compute an approximation to a precise value when the deviation from the precise value is not critical, thereby allowing the system to save energy, by reducing the number of computations. The tool instruments these types of programs to analyze which variables must produce exact calculations by employing a technique using hyberbox, to test the range of allowable values a variable can store after which they measure the deviation from the baseline output.

A similar tool to ASAC is iACT, Intel's Approximate Computing Toolkit [15], which analyzes and studies the scope of approximations in applications. The tool utilizes both LLVM [9] and Pin [11]; LLVM is used as a compiler to embed approximate computing knobs as well as to provide a runtime framework for approximate memoization, and Pin is used as a hardware simulator that handles the approximations. Using LLVM and Pin, the tool tests the application level error tolerance, while providing knowledge about the native support hardware could provide for approximate computations. While the focus of the tool is specifically for approximate computing, some of the fault models written for the tool were found to be resourceful in the understanding the sensitivity of instructions to faults, and are used by Fault Prophet.

None of the tools above provide an extensive fault injection platform that supports multiple fault scenarios, and provides an extensive analysis framework to provide a better understanding of the impact faults have on the flow of programs. Therefore, combining the benefits of existing fault injection tools would provide a comprehensive fault injection tool, which at the same time would be designed for the next generation of programs and systems.

Chapter 3

Motivating Example

Given that not all parts of a program are created equal, there is a need to identify which parts of a program are more fault-tolerant than others by identifying the critical parts in a program automatically. The motivating example for this tool is a freely distributed program, the Jacobi program. Jacobi is an important computational kernel used in various scientific computations. Fault Prophet will throughly test Jacobi to ensure its reliability and correctness even in situations of faulty operations.

3.1 Motivations

To analyze the effectiveness of the tool, and to understand its impact, the tool was evaluated based on several key questions:

- Can Fault Prophet identify the critical sections of a program? What is their sensitivity to faults?
- What computation patterns are critical and what are approximate critical sections within a program? Across programs?
- How do different fault scenarios and models help identify the critical sections of a program?
- What is the performance cost of running the tool?

Those questions are motivated on the hypothesis that a program has critical sections that are exposed to Single Event Upsets (SEU), which can cause Silent Data Corruptions (SDC). There are three main areas that were explored as a part of the motivation to answer these questions: instruction set sensitivity to faults, fault model analysis, and performance.

3.2 Example Program: Jacobi

The Jacobi method is an algorithm used to approximate a solution for a system of linear equations, $Ax = b$, where x and b are vectors of length n, and a given $n \times n$ matrix A, by iteratively solving $x^{(k+1)} = D^{-1}(b - Rx^{(k)})$ until convergence, where $A = D + R$ holds for a diagonal matrix D.

The Jacobi program[1] is a freely distributed program, which solves a finite difference discretization of Helmholtz equation: $f = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - \alpha u$ using Jacobi iterative method.

The program has four functions excluding the main function: `driver`, `initialize`, `jacobi` and `error_check` in addition to `main` function. The `main` function calls the `driver` function to run the program. The `driver` function calls the `initialize` function that allocates and initializes the two arrays used by the program to their initial values: one for $\frac{\partial^2 u}{\partial x^2}$ values, and the other for the $\frac{\partial^2 u}{\partial y^2}$ values. The `driver` then calls `jacobi`, shown in Figure 3-1, to solve the Helmholtz equation, after which it calls the `error_check` function to measure the distortion of the computed value by the program, to check for the absolute relative error. Since `error_check` is comparing the values of the relative error, in our evaluation we assume that it is error free, to ensure the computation is reliable, i.e. running on reliable hardware. We are trying to evaluate the fault sensitivity of the other three functions.

```
void jacobi(void);

/*      subroutine jacobi (n,m,dx,dy,alpha,omega,u,f,tol,maxit)
```

[1]The program was originally written by Joseph Robicheaux, and later modified by Sanjiv Shah in 1998 [19] until translated to its C version by Chunhua Liao in 2005.

```
******************************************************************
* Subroutine HelmholtzJ
* Solves poisson equation on rectangular grid assuming :
* (1) Uniform discretization in each direction, and
* (2) Dirichlect boundary conditions
*
* Jacobi method is used in this routine
*
* Input : n,m   Number of grid points in the X/Y directions
*         dx,dy Grid spacing in the X/Y directions
*         alpha Helmholtz eqn. coefficient
*         omega Relaxation factor
*         f(n,m) Right hand side function
*         u(n,m) Dependent variable/Solution
*         tol    Tolerance for iterative solver
*         maxit  Maximum number of iterations
*
* Output : u(n,m) - Solution
*****************************************************************/

void jacobi( )
{
  double omega;
  int i,j,k;
  double  error,resid,ax,ay,b;
  //      double  error_local;

  //      float ta,tb,tc,td,te,ta1,ta2,tb1,tb2,tc1,tc2,td1,td2;
  //      float te1,te2;
  //      float second;

  omega=relax;
  /*
   * Initialize coefficients */

  ax = 1.0/(dx*dx); /* X-direction coef */
  ay = 1.0/(dy*dy); /* Y-direction coef */
  b  = -2.0/(dx*dx)-2.0/(dy*dy) - alpha; /* Central coeff */

  error = 10.0 * tol;
  k = 1;

  while ((k<=mits)&&(error>tol))
  {
    error = 0.0;

    /* Copy new solution into old */
#pragma omp parallel
    {
#pragma omp for private(j,i)
      for(i=0;i<n;i++)
        for(j=0;j<m;j++)
          uold[i][j] = u[i][j];

#pragma omp for private(resid,j,i) reduction(+:error) nowait
      for (i=1;i<(n-1);i++)
        for (j=1;j<(m-1);j++)
        {
          resid = (ax*(uold[i-1][j] + uold[i+1][j])
              + ay*(uold[i][j-1] + uold[i][j+1])
              + b * uold[i][j] - f[i][j])/b;

          u[i][j] = uold[i][j] - omega * resid;
          error = error + resid*resid ;
        }

    }
    /*  omp end parallel */

    /* Error check */

    k = k + 1;
    if (k%500==0)
      printf("Finished␣%d␣iteration.\n",k);
    error = sqrt(error)/(n*m);

  }           /*  End iteration loop */

  printf("Total␣Number␣of␣Iterations:%d\n",k);
  printf("Residual:%E\n", error);

}
```

Figure 3-1: The source code of the Jacobi program in C.

3.3 Walkthrough

The tool was used on the Jacobi program to analyze the sensitivity to faults of the program. The tool injected 100,000 faults of each fault model as described in Section 4.4.3 to each of the binary, floating point and memory instruction sets. To run the tool on the Jacobi program, the user must follow these steps:

1. Install the Fault-Prophet tool (See Section A.1.1).
2. Copy the contents of the fault-inject/scripts folder to the jacobi program folder.
3. Modify the settings file (settings.cfg) as outlined in Section A.1.3 to suit the needs of the intended testing. A sample configuration for the jacobi program is shown in Figure B-2.
4. Run the tool by typing 'python script.py settings.cfg' into the terminal.
   - 4.1. The tool will filter all candidate instructions for fault injection.
   - 4.2. The tool will instrument the program to provide dynamic analysis of the program.
   - 4.3. The tool will inject random faults according to histogram of the program execution by line number, and instruction opcode.

4.4. The tool will analyze the results and will display them to the user.

The tool produces the outputs as described in Section 4.5, once it has finished running. As shown in Figures 3-2, 3-3, 3-4, the tool aggregates all the fault data, and based on the given output abstraction (see Section 4.5.1), gives a visual representation of the critical sections the tool identified.

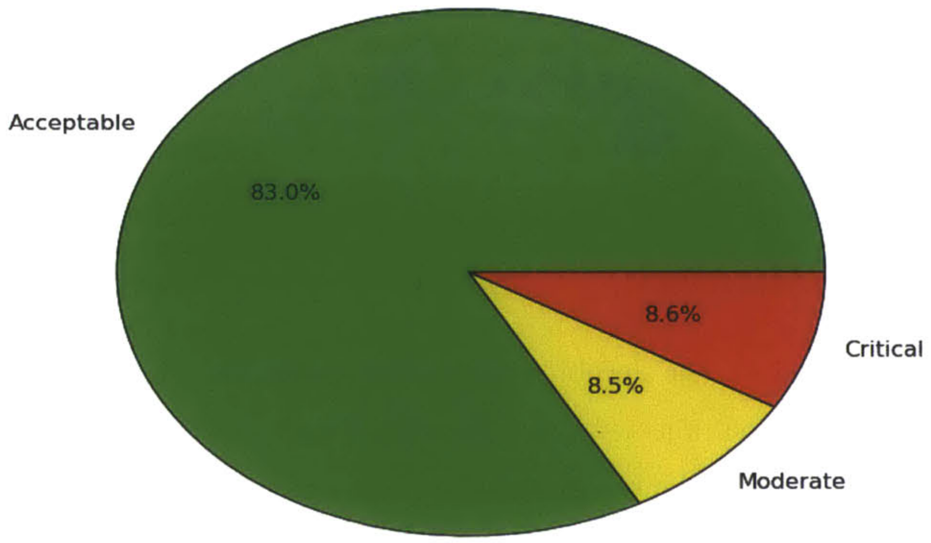
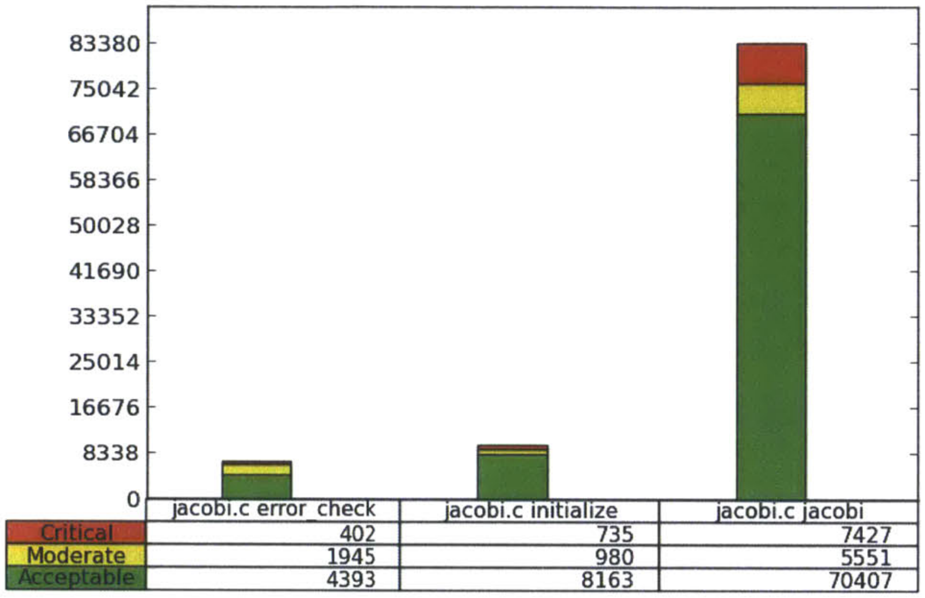


Figure 3-2: Pie chart example.

3.4 Analysis

3.4.1 Output Abstraction

An output abstraction is needed to be able to quantitatively compare the output of faulty and normal computations. The output of the Jacobi program is the output of the error function, which measures the deviation of the calculation of the solution



	jacobi.c error_check	jacobi.c initialize	jacobi.c jacobi
🟥 Critical	402	735	7427
🟨 Moderate	1945	980	5551
🟩 Acceptable	4393	8163	70407

Figure 3-3: Bar graph example.

from the exact solution. The output abstraction class illustrated in Figure B-3 was used is by the tool to analyze the output of the Jacobi program. The class uses a script, 'distortion.py' shown in Figure B-4 that calculates the distortion of the baseline output from the fault injected output, assuming both outputs are numbers, producing a range of statistics: minimum, mean, median, max, standard deviation, range, and zero elements. The output abstraction class specifically uses the mean to analyze the impact of the fault on the output of the program, given certain boundaries.

3.4.2 Fault Classification

The value of the mean the distortion algorithm outputs is between 0 and inf and as such, the fault classifications were defined as the following, for a given output o_i for all i in the set of faults:

- Acceptable : $0.0 < o_i \leq 0.0$

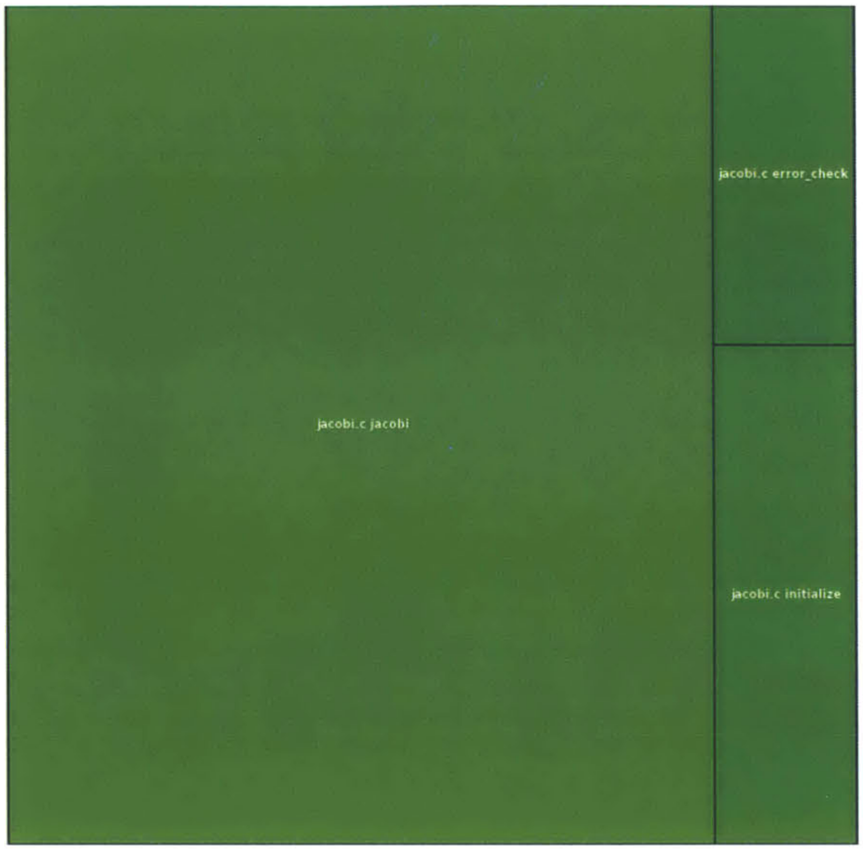


Figure 3-4: Treemap example of showing function level overview.

- Moderate : $0.0 < o_i \leq 1.0$
- Critical : $1.0 < o_i \leq inf$

'Acceptable' is defined to produce no change in the output of the program, and as such as having a mean of 0.0. Faults are classified as 'Moderate' if they produced a deviation in the output that is within ± 1 unit of the baseline, while faults are 'Critical' if the deviation is greater than 1 unit from the baseline output. There are two edge cases to the fault output: segmentation faults, i.e. program crashes, and time outs, i.e. the fault injected program lasted for longer than the alloted time that cannot be analyzed by the output abstraction script. Both edge cases were treated the same in this manner, and were given the default classification, which is 'Critical'. In the results of the Jacobi program shown in Figures 3-2, 3-3, the green correlates to 'Acceptable,' the yellow to 'Moderate,' and red to 'Critical'. Figure 3-4 shows the aggregate fault classifications for a function, where the color of each rectangle varies from green to red, correlating to the scale of 'Acceptable' to 'Critical' classification of faults, based on a score calculation of the number of 'Acceptable', 'Moderate,' and 'Critical' faults the function has.

3.4.3 Experimental Setup

The experimental setup is a EC2 instance hosted on Amazon AWS service in region US East (Virginia). The operating system installed on the virtual machine is Ubuntu 12.04 precise LTS_20120612, from Canonical Group Limited. The specification of the machine is reported in Table 3.1.

Resource	**Allocation**
Memory	3.75 GiB
CPU	2 EC2 Compute Units (1 virtual core with 2 EC2 Compute Units)
Storage	1 x 410 GB
Platform	64-bit
Network performance	Moderate
API Name	m1.medium
AMI	ami-967edcff

Table 3.1: Experimental setup machine specifications.

3.4.4 Experiments Automation

To ensure that the experiments generated by the tool on the program are consistent and reproducible, an experiments script was created. The experiments script runs the tool with different parameters, pertaining to different settings such as profiled instruction sets, fault models, quota and line coverage tests. Figure B-5 presents the script that was used for this program and other benchmark program explored in detail in Section 5.1.

3.5 Results

3.5.1 Sensitivity to Instruction Sets

The Figures 3-2, 3-3, 3-4 were only a part of results of the case study. The analysis of the faults injected, based on the single fault scenario described in Section 4.4.2, to the Jacobi program, demonstrated the fault outcomes were dependent on both the fault model and the instruction set. Therefore, there is reason to believe that sections in the program that contain specific types of instructions will be more critical than others. While the `error_check` function had faults injected to it, those are not analyzed or displayed as a part of the report, as it assumes that function runs on reliable hardware, however, this results are available.

Binary Instructions

The binary instruction set encapsulates all instructions that deal with arithmetic and logic operations. The results of 100,000 faults that were injected to the binary instruction set are shown in Table 3.2. In the table, the columns correspond to the names of the fault model described in Section 4.4.3, and thus each column contains the number of faults classified as 'Acceptable,' 'Moderate,' or 'Critical' as explained in Section 3.4.2.

Fault Type	**Random Bit Flips**	**Sticking Bits**	**XOR**	**Bit Flip**	**Byte Flip**	**Buffer Flip**
Acceptable	82,963	88,303	69,798	67,835	62,564	18,001
Moderate	8,476	5,418	11,171	17,320	18,634	31,071
Critical	8,564	6,282	19,034	14,848	18,805	50,931
Segmentation Faults	7,427	5,433	16,501	10,812	13,219	20,005
Time Outs	0	0	0	0	0	0

Table 3.2: Comparison of 100,000 generated faults by fault model for the binary instruction set of the Jacobi program.

The data in Table 3.2 shows that there is a lot of variation in the outcome of a fault that is injected to the binary instruction set based on the chosen fault model. The number of 'Moderate' and 'Critical' faults combined varies from 17% to 82%, which is high when compared to the results of faults that were injected floating-point operations and memory operations. This result can be attributed to the fact that binary operations do most of the computations in a program, some of which could be conditionals such as in loops, or pointer arithmetics calculations used to calculate memory addresses. In addition, the bottom of Table 3.2 explores the reasons for the 'Critical' faults. As observed, a large portion of the 'Critical' faults were a result of a segmentation fault, which could be the result of invalid calculations of memory addresses.

Results Analysis Examining the distribution of the faults per source line, the critical lines in the Jacobi program reside in the `jacobi` function, specifically the lines pertaining the calculation of a value in the solution. For those lines, the 'Critical' faults were more than 50% of the total faults injected into each line.

```
for (i=1;i<(n-1);i++)
    for (j=1;j<(m-1);j++)
    {
         double u1 = uold[i-1][j];
         double u2 = uold[i+1][j];
         double u3 = uold[i][j-1];
         double u4 = uold[i][j+1];
         ...
```

The loaded values are then weighted, to create the new value in the solution for the next iteration.

```
resid = (ax * (u1 + u2) + ay*(u3 + u4)+ b * u5 - u6)/b;
u[i][j] = uold[i][j] - omega * resid;
error = error + resid*resid ;
```

If the calculation of which value to load from memory, i.e. array, to calculate the solution, is faulty, then especially in an iterative method, this could be compounded over the next iterations, skewing the calculation, or even more so, accessing a non-existent memory location, causing a segmentation fault. Therefore, the calculations in these lines must be safeguarded to ensure a less faulty outcome.

In addition, the **initialize** function has multiple lines that have over 10% 'Critical' faults, pertaining to the calculation of values that used later to calculate the solution in each iteration. Again, the same reasoning applies as before that if there is a deviation in the initialize values, it will be compounded with each iteration, causing a skew in the end result.

```
xx =(int)( -1.0 + dx * (i-1));
yy = (int)(-1.0 + dy * (j-1)) ;
...
f[i][j] = -1.0*alpha *(1.0-xx*xx)*(1.0-yy*yy)\
               - 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy);
```

Statistical Significance Table 3.3 presents the results of the T-test calculations for the average error of each pair of fault models, based on the assumption that the sample of fault injection outcomes is independent for each fault model. A 'False' value in the table means that $p_{value} > 0.05$, while 'True' means that $p_{value} \leq 0.05$. As observed from the table, the fault models have significantly different averages, as represented by the 'True' value, except for the case of the Bit Flip model and the Byte Flip model results. The test was conducted using `test_ind` function in the stats package of `scipy`, an open-source python extension for mathematics, science, and engineering.

Fault Model	**Random Bit Flips**	**Sticking Bits**	**XOR**	**Bit Flip**	**Byte Flip**	**Buffer Flip**
Random Bit Flips	–	True	True	True	True	True
Sticking Bits		–	True	True	True	True
XOR			–	True	True	True
Bit Flip				–	False	True
Byte Flip					–	True
Buffer Flip						–

Table 3.3: Statistical significant analysis of the fault models based on 100,000 generated faults for the binary instruction set of the Jacobi program. True means the standard significance level was lower than 0.05, and therefore there is a statistical significance, where False means it was higher than 0.05.

Floating-Point Instructions

Floating-point operations are a subset of binary operations, pertaining only to floating point arithmetic and logic operations. As such, it is reasonable to believe that faults injected to those operations would cause deviation in output, but will not cause the program to crash, as floating point operations do not have control over the flow of a program. This hypothesis was verified by the results shown in Table 3.4, as some fault models caused no 'Critical' faults at all, and the deviation in the output was only 'Moderate', ranging from 0% – 4.3% for those fault models. The fault models that did cause 'Critical' faults, Bit Flip, Byte Flip and Buffer Flip, less 'Moderate' and 'Critical' errors than in the binary instruction set. Moreover, none of the faults that were classified as 'Critical' were the results of a segmentation faults, i.e. program crashes, which aligns with the hypothesis.

Fault Type	Random Bit Flips	Sticking Bits	XOR	Bit Flip	Byte Flip	Buffer Flip
Acceptable	95,734	99,999	97,277	85,324	81,087	22,579
Moderate	4,265	0	2,722	11,605	13,427	35,544
Critical	0	0	0	3,070	5,485	41,876

Table 3.4: Comparison of 100,000 generated faults by fault model for the floating-point instruction set of the Jacobi program.

Results Analysis Mapping the results of the faults to the source lines, there were two main lines that are 'Moderate' and somewhat 'Critical' in the program across all fault models. Those lines are a subset of the lines that were error prone in the binary instruction set, which is an expected result, as floating point operations are a subset of the binary instruction set. In the `initialize` function, initializing an array that is later used by the solution proved to be error prone:

```
f[i][j] = -1.0*alpha *(1.0-xx*xx)*(1.0-yy*yy)\
              - 2.0*(1.0-xx*xx)-2.0*(1.0-yy*yy);
```

In the `jacobi` function, the line that aggregates the results of the different coefficients proved to be error prone:

```
resid = (ax * (u1 + u2) + ay*(u3 + u4)+ b * u5 - u6)/b;
```

Memory Instructions

The memory instruction set used by the Jacobi program has only two instructions: load and call. As such, it is expected to see variation in the output, by modifying the load instruction, as well as segmentation faults to some extent if the call instruction is modified to call an illegal address. Table 3.5 aggregates the results of 100,000 faults injected to memory operations in the Jacobi program, which show that all fault models caused some variation in the output, unlike the floating point operations, in which one fault model caused no change. Moreover, the fault models that caused no 'Critical' faults for the floating point operations, caused some 'Critical' faults, but the faults caused less deviation than the benchmark output. Another important finding is that out of all the faults classified as 'Critical', expect for at most one occurrence per fault model, none of those faults were a result of a segmentation fault.

Fault Type	**Random Bit Flips**	**Sticking Bits**	**XOR**	**Bit Flip**	**Byte Flip**	**Buffer Flip**
Acceptable	97,480	99,901	99,587	87,249	81,626	25,479
Moderate	2,498	77	392	12,082	13,990	39,425
Critical	2	2	1	649	4,364	35,076

Table 3.5: Comparison of 100,000 generated faults by fault model for the memory instruction set of the Jacobi program.

Results Analysis Inspecting that lines that produced deviation from the expected output, pertaining to faults that are classified as 'Moderate' or 'Critical', it becomes clear those faults were injected into the lines that are on the critical path of calculating the solution. Those lines are mostly found in the `jacobi` function, which load calculated values that are used in the next iterations of the function:

```
uold[i][j] = u[i][j];
...
double u1 = uold[i-1][j];
double u2 = uold[i+1][j];
double u3 = uold[i][j-1];
double u4 = uold[i][j+1];
double u5 = uold[i][j];
double u6 = f[i][j];
...
error = sqrt(error)/(n*m);
```

Since these values are used for the different iterations, their impact will be compounded, and therefore will produce a much larger deviation than the expected output, than a variable that is used only a handful of times in the program.

All Instructions

The instructions that run in the Jacobi program and were profiled include: bitcast, load, sub, sitofp, fdiv, icmp, fmul, fsub, fptosi, sext, getelementptr, mul, fadd, add, call, alloca, fcmp, srem. Of these instructions, getelementptr, the instruction that explicitly provides address computations but does not access memory, and sext, the sign extension instruction, together account for about 50% of all instruction executions. The next instruction is load, which accounts for roughly 14% of instruction executions. This distribution helps to explain the results shown in Table 3.6, in which more than 50% of faults, on average, were classified as 'Moderate' or 'Critical', as address calculations and sign extension instructions were the most prominent instructions. This is also clear from the bottom of Table 3.6, which shows that most 'Critical' faults were a result of a segmentation fault, supporting the fact it is due to injection of critical instructions and operations.

Fault Type	**Random Bit Flips**	**Sticking Bits**	**XOR**	**Bit Flip**	**Byte Flip**	**Buffer Flip**
Acceptable	52,552	65,894	35,915	45,410	36,350	9,589
Moderate	6,922	9,424	6,195	14,364	14,717	17,891
Critical	40,567	24,723	57,931	40,267	48,974	72,561
Segmentation Faults	38,796	18,611	56,914	38,149	45,009	58,027
Time Outs	0	0	0	0	0	0

Table 3.6: Comparison of 100,000 generated faults by fault model for the all instructions of the Jacobi program.

Chapter 4

Design

4.1 System Overview

To make Fault Prophet applicable to a variety of scenarios, and to different program architectures, Fault Prophet was to designed as a modular fault injection tool. Such a tool can accept input from a variety of sources, as long as they adhere to the same input and output format. The four main modules Fault Prophet is comprised of are: Fault Selection Analysis, Profiler, Fault Injector, and Sensitivity Report Analysis, as shown in Figure 4-1. Each module depends on the previous module's output, as it uses it for input.

There were two assumptions that were made when designing the fault injection tool. The first assumption is that the program is deterministic, and has a definite termination point. The second assumption is that the program is compiled with debug information, to extract relevant information about each instruction, such as source file and line number.

4.2 Fault Selection Analysis

Fault Prophet allows a user to analyze the impact soft-errors have on specific instructions within a program. For example, a user might want to examine the impact of soft-errors on floating point operations, while another user might want to under-

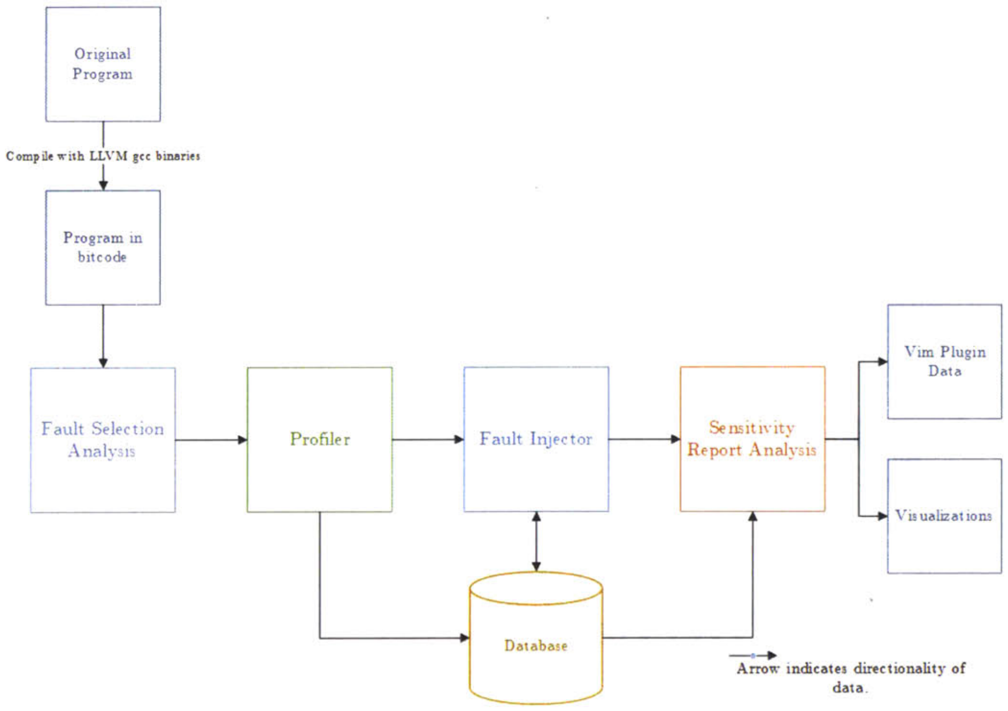


Figure 4-1: The system overview diagram, showing the dependences of each the component.

stand the impact on memory operations. To support this functionality, the Fault Selection Analysis module performs a static analysis of the program, to capture the instructions the user is interested in. After aggregating all the instructions, the Fault Selection Analysis module filters out unwanted opcodes from an instruction set, as well as unwanted source lines from specific source files.

4.2.1 Static Analysis

The static analysis performed by the Fault Selection Analysis is done using LLVM passes on a program. An LLVM pass allows to programmatically go through every instruction in LLVM bit code (middle layer between source code and machine code), and analyze it as needed. The pass that was written for the tool profiles instructions if they belong to a specific instruction set: floating point operations, binary operations (including floating point), memory operations, use-define chains, function calls, arrays and all instructions, saving the output of each instruction set that the user selected to be instrumented to a separate file. At least one instruction set has to be selected for the tool to work, which will be defined as the white-listed instruction set(s); faults will be injected only to the white-listed instruction sets. The output of the static analysis is of the following format:

<function name>,<instruction count within the function>,<is function call>,<is an array>,<opcode>,<line number>,<file name>

The function name, opcode, line number and file name are data points about the instruction that are provided either from the instruction itself, or the debug information of the instruction. The rest of the data points are a bit more complicated to gather. Since the LLVM bit code does not change unless the original source code is modified and compiled again, the count of every instruction traversed in a function is unique. As such, the instruction count within a function is a way to uniquely identify an instruction in the LLVM bit code across the other modules of the tool. Because the instructions are traversed in a deterministic order set forth by the compiler implementation, this algorithm might be dropped in future iterations of the tool given the identifiers of the instructions will be cached during compilation. Figure 4-2 presents an example of code that provides the instruction count of an instruction in a function. The is-function-call parameter is set to 1 if the instruction calls a function and 0 otherwise. Likewise, the is-an-array parameter is set to 1 if the instruction has an array as parameter and 0 otherwise. While checking whether is a function call is trivial - simply checking the instruction signature, to check whether an instruction has an array in its operands the meta-data of the instruction must be traversed completely, to find a specific tag LLVM generates that is associated with arrays. Those two parameters allow much greater control over the the fault injection capabilities to these type of instructions as mentioned in Section 4.4.2.

4.2.2 Filters

Once the instructions of interest have been selected, filters can provide a finer granularity to select the instructions of choice. There are three filtering options that are

```
unsigned int getProfileInstr(Instruction *I) {

    Function* F = I->getParent()->getParent();
    string funcname = F->getNameStr();

    unsigned int func_counter = 0;

    for (inst_iterator In = inst_begin(F), E = inst_end(F); In !=
        E; ++In) {
        Instruction *Instr = dyn_cast<Instruction>(&*In);
        func_counter ++;

        if (I == Instr)
           break;
    }
    return func_counter;
}
```

Figure 4-2: Sample code of an instruction counter in a function. The input is an instruction and the output is the count of that instruction in the function.

employed by the tool: blacklisting, opcode filter, and source line filter. The instructions that remain after running the filters are the only the instructions that will be potential candidates for fault injection by the tool.

Whitelisting & Blacklisting Instruction Sets

A user can choose whether they would like to blacklist an instruction set or white-list it, in the same manner that is described in Section 4.2.1. The difference between blacklisting and white-listing is that by blacklisting an instruction set, the instructions that fall into that instruction set will be excluded from the fault injection process, while white-listing an instruction set, will include those instructions in the fault injection process. Whenever there is a conflict, the blacklisting has priority. The blacklisting feature can come in handy whenever specific instructions overlap in at least two instruction sets.

Example 1 A user wants to include only binary operations for integers and not floats. In such a case, binary operations will be white-listed, while floating point operations will be blacklisted.

Example 2 A user wants to include binary operations, such as `add` but leave out any instructions that involve use-define chains, such as `getelementptr` or `sext`, which include operations to calculate pointer addresses. In such a case, binary operations would be white-listed, while use-define chains will be blacklisted.

Granularity of Program Object Selection

The user has the option to filter specific opcodes and source lines from specific source files.

Opcode Filter The user has the option to exclude specific opcodes on top of the instruction sets that are white-listed. This filter might come handy if the user wants to inject faults to an instruction set but leave out specific opcodes. For example, a user might want to profile binary operations but leave out specific binary operations such as `srem` or `xor` instructions, which calculate the remainder of integer division and performs an xor two operands, respectively.

Source Line Filter The user might want to inject faults only to specific lines of code, by either including specific lines and excluding everything else, or excluding specific lines and including everything else. While only one line filter is needed - to either include or exclude lines, as one is a negation of the other, in some scenarios it will be easier to use one or the other, and therefore both options are provided.

4.3 Profiler

The Profiler module has two responsibilities: providing a baseline output to be used for comparison analysis, and profiling the program to understand the instructions frequencies and flow of the program. The Profiler depends on the output of the Fault Selection Analysis module, to decide which instructions to instrument, and saves the profiling data only for those instructions.

4.3.1 Dynamic Analysis

The Profiler augments the original program in order to trace the execution of the program, and to keep a running count of the number of instructions that are executed. The running count allows the program to map an execution of an instruction, an execution point, to a unique number. The count helps determine how many times each instruction has been executed, as the same instruction could be executed multiple times, for example, if it is in an function or a loop. This data enables the Fault Injector module to inject a fault to a specific execution point in the program.

The tool augments the program by adding a function call before each instruction. If the instruction is an instruction of interest (as provided by the Fault Selection Analysis module, the called function saves the instruction execution data to a log file, which includes:

<opcode>,<instruction execution count>,<line number>,<file name>,<function name>

If the instruction is not an instruction of interest, then a function called only increases the running count of instruction executions in the program.

4.3.2 Database

The tool uses a sqlite3 database to store all the dynamic analysis data, to deal with programs that generate large amounts of data during the dynamic analysis process. The tool chunks the data gathered into many log files, each containing 100,000 profiled instructions, keeping the log files small. The dynamic analysis and log file process occur simultaneously, allows the tool to store the data in the database while the program is instrumented, parallelizing the process. The tool employs the watchdog module for python [13], to monitor the directory to which all the log files are saved, and processes a file after all the instructions were written to it. After the log file has been processed it is deleted to free up space. The database layout is shown in Figure 4-3.

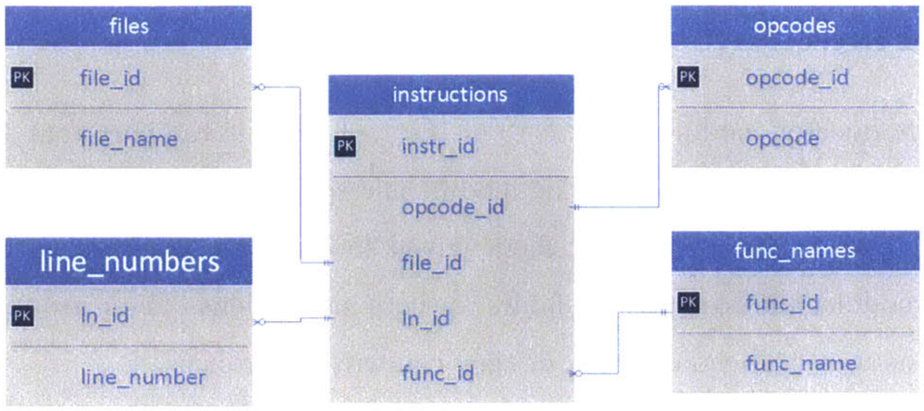


Figure 4-3: The database schema used by the tool.

In addition, while the tool processes the log files, it saves a histogram of the instruction frequencies by source file, line number and opcode. This data is later used to pick the faults according to the instruction distribution presented by the histogram.

Debugging Information

If needed, the debugging information can be outputted during the Profiler stage. The debugging information maps an instruction in LLVM bitcode to a source line number, and source file. This data is later used by the Vim plugin, fltmark, in order to show the number of instructions in each source line, and the type of instructions, making it critical to output this data if the user wants to view the data in the plugin.

Dynamic Trace

The user can output the trace of the original program: saving the entire execution path of the program. However, it is important to realize that the overhead is high, as the number of executed instructions in the program are at least doubled and each of those added instructions is an I/O operation used to record the data of each instruction in the program, meaning the run-time of the program is about 100x-1000x the regular run-time. The dynamic trace of the original program can later be compared to traces of programs that had faults injected to them, in order to understand how the fault modified the flow of the program.

4.4 Fault Injector

The Fault Injector module is the core of the tool, as it is the part that injects the actual faults into the program. The module is responsible to select the instructions that will have faults injected to them, given the dynamic analysis output of the Profiler module, and to inject the faults to those instructions. There are several fault models the user cam choose from, or they can write their own.

4.4.1 Selecting Faults

There are two ways faults are picked by the module: either by a fixed quota set by the user, or a quota set by a line coverage percentage.

Fixed Quota

In the fixed quota model, the tool selects the number of faults based on the histogram of the executed instruction in the program. The histogram provides the weight of each source line in the overall flow of the program, $W_{source\ line=i}$, based on the number of instruction executions mapped to that source line, i over the total number of profiled instruction executions. The tool then calculates the weight of each opcode in a source line, $R_{opcode=o}$, by dividing $W_{source\ line=i}$ by the number of executions of an opcode in source line i. $f_{opcode=o\ in\ source\ line=i} = W_{source\ line} \times R_{opcode=o}$ is the fraction of the overall faults that would be selected to an opcode in a given source line.
The tool samples $Q \times f_{opcode=o\ in\ source\ line=i}$, random execution points[1] that have the opcode o and are in source line i.

Line Coverage

In the line coverage model, the user specifies the source line coverage percentage, p, the percent of lines of code that must have a fault injected to them before halting. The tool retrieves the total number of lines, $count_{number\ of\ lines}$ that have instructions of interest, and calculates $N = p \times count_{number\ of\ lines}$ the number of lines that must

[1]Q is the fixed quota of faults set by the user.

have faults injected to them. The tool randomly selects N source lines, and randomly selects C, a configurable number of execution points for each of those source lines that will have faults injected to them, for a total of $C \times N$ faults will be injected. The default configurable number of faults, C, is 3.

4.4.2 Fault Scenarios

Errors are unexpected by nature, their frequencies, their consequences, and their origins. Errors can also occur in different locations, whether it is within the same buffer, within consecutive instructions, all of which could be independent or correlated. To support this myriad of fault scenario, the tool allows a user to use four different fault injection scenarios: single faults, burst of faults, function call faults and array faults. The tool allows the fault injected program to run for a specified amount of time, which if exceeded the program is terminated, deeming that fault as one that caused timeout.

Single Fault

Under the single fault scenario, the tool injects a single fault to a specific execution of an instruction. Before the fault is injected, the tool will verify that the given execution point matches the opcode, as depicted by the Profiler, and will inject a fault only if that condition is true. The tool also allows the user to inject repeated faults into the same execution point as many times as they specify.
With this fault scenario, the user can specify a fault model they would like to use to inject a fault to the buffer of the chosen execution point, as described in Section 4.4.3. Regardless of the fault model that is specified, the tool saves the fault information for a later analysis if needed, including the original instruction buffer as well as the new one, along with the execution point identifier of the injected fault.

Burst Mode

The bursts mode scenario, allows the injection of multiple faults at a time. It employs the same basis of the single fault model as described in Single Fault under Section 4.4.2, injecting a fault at a specific execution of an instruction, after which it continues to inject faults, at a specified interval of instruction until a quota of faults have been injected. For example, the tool will inject a fault every 2 consecutive instructions for a total of 5 faults, after instruction I_i, where i is the initial execution point that had a fault injected to it. Under these configuration, the tool will inject faults to $I_{i+2}, I_{i+4}, I_{i+6}, I_{i+8}, I_{i+10}$, regardless if they were profiled or not. If a fault causes a segmentation fault, no further faults will be injected.
This fault model allows the simulation of a burst of soft-errors that might occur from cosmic rays. Since the basis is the same as the single fault model, it uses the same fault models as described in Section 4.4.3. However, unlike the single fault model, the analysis of these faults performed by the Sensitivity Report Analysis module described in Section 4.5, is not deterministic, as the affects on the output can arise from any of the faults injected. As such, the analysis of this fault scenario requires the comparison of the faulty trace to the baseline trace, which is not implemented due to reasons mentioned in Section 1.1.2.

Function Call Faults

Under this scenario, which was inspired by Rely and Chisel, a function call instruction is modified to call a new function. This new function, will replace the call instruction buffer with a user defined probability,p, to a randomly generated value, and with probability $1-p$, the original function will be called. To utilize this functionality, the user must specify a certain flag in the tool configuration.

Array Faults

Much like the function call faults described in Function Call Faults under Section 4.4.2, faults can be injected specifically into instruction that use arrays, by modifying the instruction that uses an array to a call instruction. The call instruction calls a function that with a user defined probability, p, will replace the array pointer of the original instruction with a new random value, and with probability $1 - p$ will keep the original array pointer. Since pointers cannot be differentiated at run-time, any pointer of an instruction that has been marked for fault injection, i.e., has an array pointer, will be modified with probability p. If the user would like to inject only a single fault, rather than inject faults with some probability, the user can use the single fault scenario. To utilize this functionality, the array instruction set must be profiled by the Fault Selection Analysis module along with the appropriate flag in the configuration of the tool.

4.4.3 Fault Models

The tool supports a robust and dynamic configuration of fault models that can be utilized across different fault scenarios. As such, fault models can be added, removed and modified, and used quite easily. To achieve these characteristics, the tool uses a function factory, which selects the user specified fault function. To add a new fault model, the user needs to write a function that adheres to the function signature shown in Figure 4-4. The function signature provides the user with the original buffer, which they will need to modify, the size of the buffer, which is 4 or 8 if 32 bits or 64 bits respectively, and the opcode of the instruction. To add the fault function, the user will need to add it to the factory, as shown in Figure 4-5. There are six fault models by default: random bit flips, sticking random bits, XOR mantissa bits, random bit flip, random byte flip, and a buffer flip. Random bit flips and XOR mantissa bits were burrowed from Sara Achour, based on fault models created by Accelerator Architecture Lab, Intel Labs. Sticking random bits fault model is Sara Achour's own fault model. Random bit flip, byte flip and buffer flip are taken from LLFI [24].

```
void fi_func(int size, char* buf, char* opcode);
```

Figure 4-4: The function signature of a fault model function. Size is the size of the buffer, buf is the original buffer, which the function must modify, and opcode is the instruction opcode.

```
#include "fi_func.c"

void init_factory() {

    create_factory();
    add_fi_func(&fi_func);
}
```

Figure 4-5: The factory function found in fi_factory.c. The file where the fault model is located must be included in the file, and the function must be added to the factory. Please make note of the location of the function in the factory, as it serves the the fault model id number later used by the tool.

Random Bit Flips

The random bit flips fault model randomly flips bits in the buffer. The most number of bits that can be flipped by this model is 8 regardless of the buffer size. However, due to the nature the bits are chosen, the same bit could be flipped multiple times, returning it to its original state.

Sticking Random Bits

The sticking random bits randomly generates a mask that is used to pick a value from one of two buffers. The first buffer is the original instruction buffer, while the second buffer is generated using the XOR fault model using the original instruction buffer. The mask is set for each byte individually, for a total of 8 bit flips across all the masks.

XOR Mantissa Bits

The XOR mantissa bits fault model flips the mantissa bits, which are the lower 7 bits of a float (32 bits), and the lower 44 bits of a double (64 bits). If used for an integer (32 bits) or a long (64 bits), the same bits will be flipped.

Random Bit Flip

The random bit flip fault model generates a random number that corresponds to a byte within the given buffer, and a random number that corresponds to a bit within the selected byte, and flips the value of that bit, by xoring the bit with $0x1$.

Random Byte Flip

The random byte flip fault model generates a random number that corresponds to a byte within the given buffer, and sticks a byte sized random number in that position.

Buffer Flip

The buffer flip fault model generates a byte sized random number for each byte in the original buffer.

4.5 Sensitivity Report Analysis

The Sensitivity Report Analysis module looks at the data produced by the Fault Injector module, and analyzes it to determine the effect the fault(s) had on the program. A fault can be classified into 3 categories: *Acceptable*, *Moderate*, and *Critical*. The classification of each fault is later used to visually show the impact of the faults by the vim plugin or the other visualizations.

4.5.1 Output Abstraction and Fault Classification

The Sensitivity Report Analysis module determines the effect a fault had on a program, by comparing the output of the fault injected program to that of the baseline output of the program. The tool allows the user to write their own output abstraction, by using a template class, `Output_Abs`. The user has to write two functions:

- analyze_fault – The function classifies a fault based on comparing the baseline output and the output of the injected program. The input to the function is two file paths, one for each output file, and the expected output is the fault classification: acceptable, moderate or critical.

- calculate_score – The function analyzes the aggregate score of a line in a source file, given the number of acceptable, moderate, and critical faults that were observed for that line, by the user's classification, and outputs the score and classification of the source line based on the score.

Figure B-3 shows an example of an output abstraction class that is explained in Section 3.4.1.

While there is no distinction between faults in the same fault class, i.e. two critical faults are the same regardless of the variation they caused, the output that is saved the summary described in Section 4.5.2, allows the user to see that information. This is useful, if for example, the user wants to know how many faults were caused by segmentation faults and / or timeouts.

Acceptable

The *Acceptable* category is used to classify faults that cause acceptable or preferably no variation in the output of the fault injected program when compared to the original output.

Moderate

The *Moderate* category is used to classify faults that cause a moderate variation in output. An example of such a case would be a valid output that within multiple standard deviations of the original output.

Critical

The *Critical* category pertains to any output that causes an extreme variation in output, and thus it is recommended that segmentation faults will be considered a part of this category, as well as timeouts.

4.5.2 Summary

The tool will record the output abstraction results into a summary file for each injected fault, as well as the source file, line number, opcode and the execution point identifier of the injected fault. This file is useful for a quick analysis of the output abstraction script, over all the injected faults, as well as a future reference point.

4.5.3 Vim Plugins

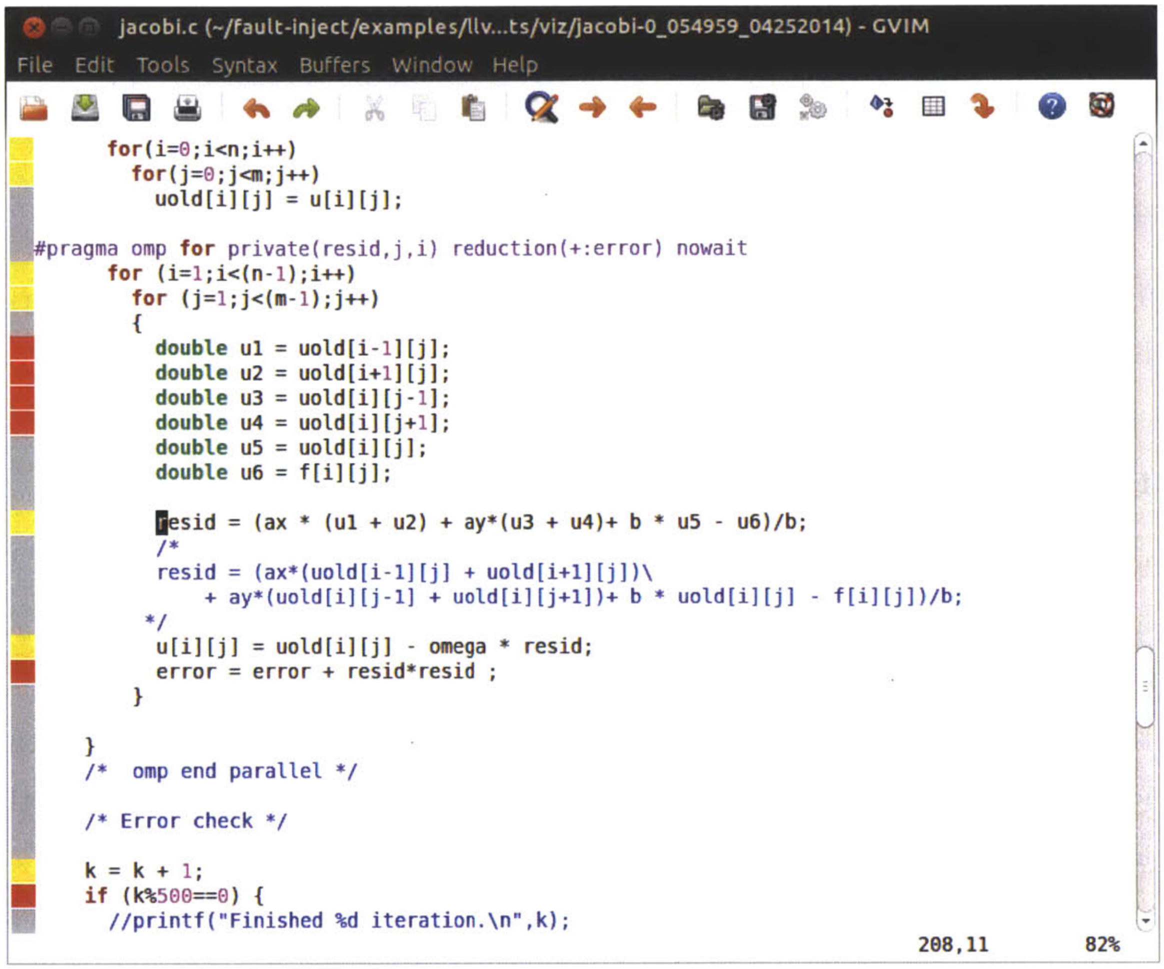
There are two vim plugins: FltMark and InstShow that were written as a part of the tool, to help visualize the data in a text editor, on top of the source code data.

FltMark

FltMark, as the name suggests, shows fault markers based on the classifications, for lines of code that had faults injected to them. Since each line of code can have multiple instructions, and multiple points of execution, the aggregated score for all those data points is used to classify the line as explained in Section 4.5.1. Thus, this vim plugin allows the developer to easily see the sensitivity of each source line to the injected faults.
On top of the visual functionality, the plugin can run Fault Prophet from within vim itself. In order to run the tool, the user must tag lines, and only the tagged lines will have faults injected them. Moreover, the user must generate a configuration file that will be used by the plugin, from an existing configuration file, by running the 'config.py' script. While most settings will be used, some settings will be changed automatically to suit the configuration to run the tool from within vim, such as the quota. Please see Section A.2 for full documentation of the plugin.

InstShow

InstShow is a plugin for vim that shows the instructions that are encapsulated in each source line from the LLVM Internal Representation, IR. The plugin itself is based on PyFlakes [6] and requires vim that has been compiled with python. A part of the Fault Prophet installation, such a vim is compiled and is installed on the machine as 'vimpy'.


jacobi.c (~/fault-inject/examples/llv...ts/viz/jacobi-0_054959_04252014) - GVIM
File Edit Tools Syntax Buffers Window Help
208,11
82%

```
      for(i=0;i<n;i++)
        for(j=0;j<m;j++)
          uold[i][j] = u[i][j];

#pragma omp for private(resid,j,i) reduction(+:error) nowait
      for (i=1;i<(n-1);i++)
        for (j=1;j<(m-1);j++)
        {
          double u1 = uold[i-1][j];
          double u2 = uold[i+1][j];
          double u3 = uold[i][j-1];
          double u4 = uold[i][j+1];
          double u5 = uold[i][j];
          double u6 = f[i][j];

          resid = (ax * (u1 + u2) + ay*(u3 + u4)+ b * u5 - u6)/b;
          /*
          resid = (ax*(uold[i-1][j] + uold[i+1][j])\
              + ay*(uold[i][j-1] + uold[i][j+1])+ b * uold[i][j] - f[i][j])/b;
         */
          u[i][j] = uold[i][j] - omega * resid;
          error = error + resid*resid ;
        }

    }
    /*  omp end parallel */

    /* Error check */

    k = k + 1;
    if (k%500==0) {
      //printf("Finished %d iteration.\n",k);
```

Figure 4-6: FltMark vim plugin example.

4.5.4 Visualizations

There are currently three supported visualizations: pie chart, bar graph and treemap that visually show the categorization of faults in different ways.

Pie Chart

The pie chart visualizes the overall distribution of the different fault classifications as described in Section 4.5.1 for the whole program. There are at most three parts of the chart, green, yellow and red, which represent the fault classifications 'Acceptable', 'Moderate', and 'Critical', respectively, as shown in Figure 3-2. The goal of the visualization is to quickly understand how sensitive the program is, for the specific fault scenario and fault model.

Bar Graph

The bar graph visualizes the distribution of the faults for each function in the source file(s) of the program, as well as the categorization of the faults as described in Section 4.5.1. The height of each bar represents the total number of faults injected to each function, and the bar itself is divided into three parts, green, yellow and red, which represent the fault classifications 'Acceptable', 'Moderate', and 'Critical', respectively, as shown in Figure 3-3. The goal of the visualization is for the user to understand how sensitive a function is, for the specific fault scenario and fault model. Using this visualization, the user can quickly understand which functions must be explored in depth, using the treemap visualization.

Treemap

The treemap graph is used to show two things: the weight each entity, whether function or source line, has in terms of instruction executions, and the aggregated sensitivity of that entity. To exhibit those two qualities, the size of each rectangle in the graph symbolizes the weight of each entity in relation to other entities, by using the number of instructions executions that are mapped to that entity; the color of the rectangle is the average score of a function based on the source lines in that function, on a scale from 0 to 1, where 0 is not sensitive at all, and 1 is the most sensitive, as shown in Figures 3-4, B-1. The tiling algorithm, which is used to determine the size of each rectangle in the visualization, was taken from an existing Treemap library [22]. The treemap is the only interactive graph from the current visualizations. The initial view shows the sensitivity of functions by source code file. Upon clicking on a rectangle in the view, a new view will appear, showing the sensitivity of lines in that function. Clicking on a rectangle in this view, will open gvim, a graphical vim interface, to the source line in the source code file, the user clicked on. Using this view, the user can quickly drill down, from a function, to a source line, to instructions, using the vim plugins described in Section 4.5.3.

Chapter 5

Evaluation

As mentioned in Chapter 3, the motivation for the experiments was to evaluate the tool in three different areas: instruction set sensitivity to faults, fault model analysis, and performance. This Chapter completes the evaluation of the tool.

5.1 Benchmark Programs

We used several programs to test the effectiveness of the tool, all of which were provided by the Department of Aeronautics & Astronautics at MIT.

5.1.1 Jacobi

Please see the description of the Jacobi program in Section 3.2.

5.1.2 Ignition Cases

There are three programs reporting the observations made when simulating the ignition times of three chemical compounds: hydrogen, ic8 and methane, in tubes, under different temperature conditions. Each program uses the Cantera, a suite of object-oriented software tools for problems involving chemical kinetics, thermodynamics, and transport processes. Both the inputs and the output of the programs has been modified to run for shorter intervals, still employing the same functionality, and to produce a single number that sums up all the observations made by the program on the data, respectively.

5.2 Performance

There are overheads costs to augmenting the program, in order to either profile the program as described in Section 4.3 or to inject faults to it as described in Section 4.4. This overhead cost has several factors, including processing time, as well storage needed to run the tool on programs.

5.2.1 Processing Power

Table 5.1 compares the runtime of a program without Fault Prophet, to the time it takes Fault Prophet reach the point of first fault injection to the program for binary operations, in seconds.

Program	**Runtime without FP**	**Runtime with FP until 1^{st} Injection**
Jacobi	2.9	60 <
Ignition Case: Hydrogen	8	1,380
Ignition Case: ic8	13	7,080
Ignition Case: Methane	29	24,780

Table 5.1: The processing overhead comparison of Fault-Prophet on different programs in seconds for a binary instruction set.

5.2.2 Storage

Table 5.2 shows the number of profiled instructions for binary operations by Fault Prophet, and the amount of storage it required on the machine. The data that is stored for all those profiled instructions, includes the execution id, as well as function name, source file name, line number and opcode. In addition visualization data will get aggregated for each source file name, and line number in different tables in the database. However, since this data pertains only for instruction that had faults injected to them it is insignificant compared to the rest of the data stored.

Program	Number of Instructions Profiled	Sqlite3 Database Size
Jacobi	557,375	11
Ignition Case: Hydrogen	13,996,793	253
Ignition Case: ic8	155,294,211	2,944
Ignition Case: Methane	193,650,619	3,701

Table 5.2: The storage overhead comparison of Fault-Prophet on different programs, in Megabytes.

5.3 Fault Models

5.3.1 Output Abstraction and Fault Classification

The output of the benchmark programs was modified in order to produce a single number as an output, which is the aggregation of the results from observed simulations. Therefore, the same output abstraction class and fault classification techniques that were used by the Jacobi program described in Sections 3.4.1, 3.4.2, were used to analyze the rest of the benchmark programs.

5.3.2 Fault Model Comparison

Based on the results of the different fault models of the case-study outlined in Section 3.5.1 as well as additional results of the ignition cases, shown in the tables in this Chapter, including but not limited to Tables 5.3, 5.4, 5.5, the 'Buffer Flip' fault model is the one that causes the most disruption in the output of the program, averaging at 50% of the time that the output has varied. However, the likelihood that a whole buffer will be flipped as a result of a SEU, is rather low, making the fault model an unlikely candidate to understand the behavior of the program. The fault model that causes the least amount of disruption in the output when a fault is injected is 'Sticking Bits' that on average causes variation in 1% of outputs, with the exception of the data displayed in Table 3.2.

Fault Type	Random Bit Flips	Sticking Bits	XOR	Bit Flip	Byte Flip	Buffer Flip
Acceptable	851	886	755	768	777	649
Moderate	50	27	105	74	63	66
Critical	38	26	79	97	99	224

Table 5.3: Comparison of generated fault types by fault model, based on the Hydrogen ignition case, with profiling binary operations, and 1,000 fault quota.

Fault Type	Random Bit Flips	Sticking Bits	XOR	Bit Flip	Byte Flip	Buffer Flip
Acceptable	841	895	789	789	799	685
Moderate	74	28	109	99	76	83
Critical	32	14	49	59	72	179
Segmentation Faults	16	9	28	52	60	156
Timeouts	12	4	18	7	6	1

Table 5.4: Comparison of generated fault types by fault model, based on the Methane ignition case, with profiling binary operations, and 1,000 fault quota.

5.4 Instruction Set Sensitivity

This section analyzes the sensitivity of the different instruction sets based on the results of the case study outlined in Section 3.5.1, as well as of the ignition cases.

5.4.1 Binary

Tables 5.3, 5.4 provide additional insight to the fault outcomes that occur as a result of injecting faults in the binary instruction set. As observed for the hydrogen and methane ignition cases, faults injected to the binary instruction set produced similar outcomes based on the number of fault classifications for each fault model. The 'Moderate' and 'Critical' faults which combined make up 9% − 30% of the faults, are less than the 17% − 82% that was observed in the Jacobi program as shown in Table 3.2. The reason for this could be attributed to the fact that the ignition cases perform many calculations that are not on the critical path of the program, pertaining to calculations of species in the reaction. This strengthens the hypothesis that the sensitivity of an instruction set depends on the program itself, rather than the instruction set.

Results Analysis of Ignition Cases

The results of the hydrogen, ic8 and methane are not only similar in terms of displaying close numbers of fault classifications, but also in the outcomes of the fault themselves when mapped back to source lines. As the three programs use the same libraries, and most of the instruction executions happen in those libraries, there is a commonality in the faulty lines. A deeper inspection of the 'Moderate' and 'Critical' faults shows that there is a greater likely for those faults to occur in the processes that compute the reactions defined by the input parameters. An example would be in the GasKinetics module which provides a module for a reaction mechanism, in which all the reaction update functions in the module provides are fault prone.

```
void GasKinetics::_update_rates_T() {
          ...
          if (fabs(T - m_kdata->m_temp) > 0.0) {
                    ...
                    m_rates.update(T, logT, &m_kdata->m_rfn[0]);
                    ...
          }
          ...
}
...
void GasKinetics::updateKc() {
          ...
          doublereal rrt = 1.0/(GasConstant * thermo().temperature
              ());
          for (i = 0; i < m_nrev; i++) {
                    ...
                    m_rkc[irxn] = exp(m_rkc[irxn]*rrt -
                              m_dn[irxn]*logStandConc);
          }
}
...
void GasKinetics::updateROP()  {
          ...
```

```
        for (int j = 0; j != m_ii; ++j) {
                ropnet[j] = ropf[j] - ropr[j];
        }
        ...
}
```

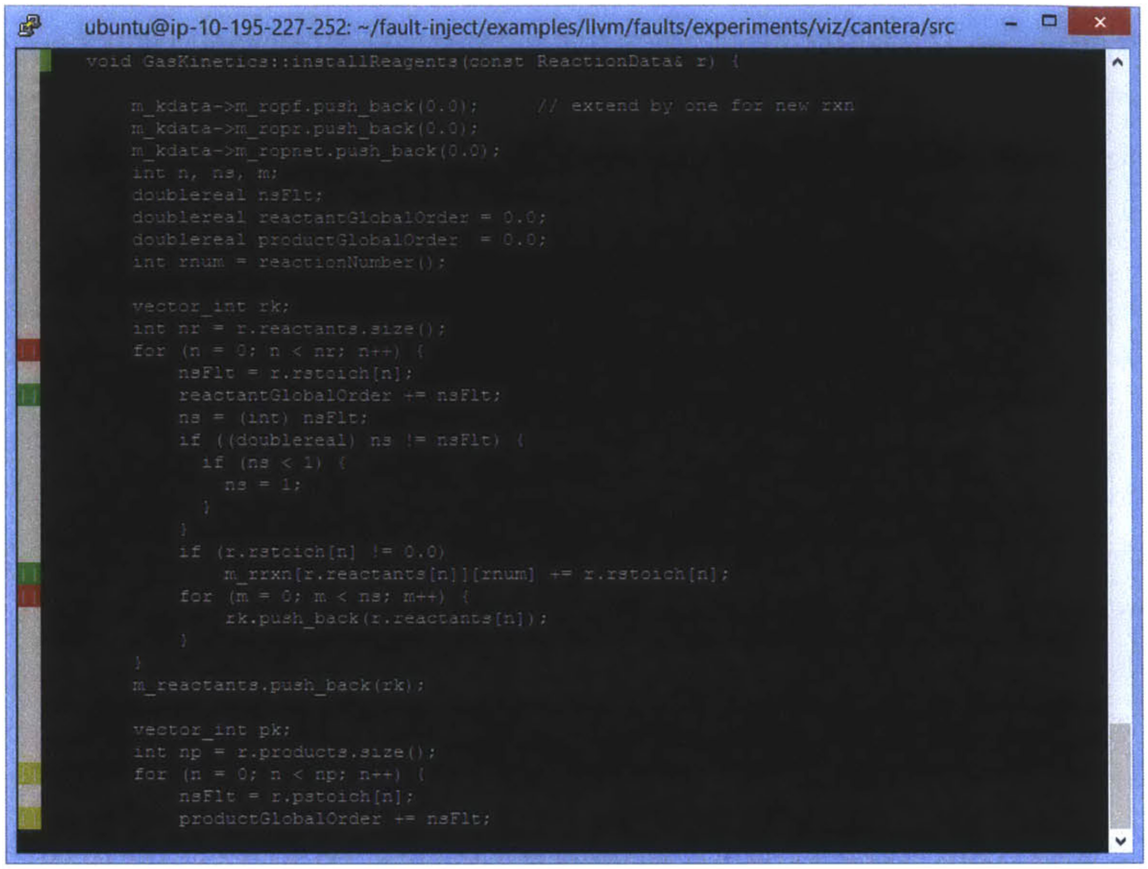
The lines shown were classified had more than 50% of the faults classified as 'Moderate' or 'Critical' across the different fault modules and programs, based on the aggregate faults that were injected in those lines, advising that the reaction update process, wherever it is taking place, is a sensitive one. In addition, Figure 5-1 shows the lines in another function function, which is used to update the reactants of a reaction with the data, dealing with the actual update process are sensitive to faults. The update proceeses had the same outcomes in other modules of the libraries such as `IdealGasPhase`, in which the update thermodynamic properties function is also prone to 'Moderate' and 'Critical' faults. Therefore, the update functions in general, and the lines pertaining to the actual updating, can cause faulty outcomes, as they deal directly with the output data.

5.4.2 Floating Point

As hypothesized in Section 3.5.1, fault that are injected into floating point operations causes variations in the output, however, those variations will not cause a segmentation fault, as those operations are not on the critical path of the program control. A fault injected to floating point operations results in $3\% - 13\%$ 'Moderate' faults, and $0\% - 5\%$ 'Critical' faults as observed from Tables 3.4, 5.5, with the exception of Buffer Flip that caused higher variations in output.

Fault Type	Random Bit Flips	Sticking Bits	XOR	Bit Flip	Byte Flip	Buffer Flip
Acceptable	908	931	832	854	857	709
Moderate	57	31	126	91	81	101
Critical	2	0	9	22	29	157

Table 5.5: Comparison of generated fault types by fault model, based on the Hydrogen ignition case, with profiling floating point operations, and 1,000 fault quota.


ubuntu@ip-10-195-227-252: ~/fault-inject/examples/llvm/faults/experiments/viz/cantera/src

```
void GasKinetics::installReagents(const ReactionData& r) {

    m_kdata->m_ropf.push_back(0.0);     // extend by one for new rxn
    m_kdata->m_ropr.push_back(0.0);
    m_kdata->m_ropnet.push_back(0.0);
    int n, ns, m;
    doublereal nsFlt;
    doublereal reactantGlobalOrder = 0.0;
    doublereal productGlobalOrder  = 0.0;
    int rnum = reactionNumber();

    vector_int rk;
    int nr = r.reactants.size();
    for (n = 0; n < nr; n++) {
        nsFlt = r.rstoich[n];
        reactantGlobalOrder += nsFlt;
        ns = (int) nsFlt;
        if ((doublereal) ns != nsFlt) {
          if (ns < 1) {
            ns = 1;
          }
        }
        if (r.rstoich[n] != 0.0)
            m_rrxn[r.reactants[n]][rnum] += r.rstoich[n];
        for (m = 0; m < ns; m++) {
            rk.push_back(r.reactants[n]);
        }
    }
    m_reactants.push_back(rk);

    vector_int pk;
    int np = r.products.size();
    for (n = 0; n < np; n++) {
        nsFlt = r.pstoich[n];
        productGlobalOrder += nsFlt;
```

Figure 5-1: A sample of GasKinetics.cpp from the Cantera library, showing the different fault outcomes, using the Bit Flip fault model on Binary instructions.

Results Analysis of Ignition Cases

As the floating point operations are a subset of the binary instructions, much like the Jacobi program, the same source lines that were classified as 'Moderate' or 'Critical' in binary instruction set, were classified the same way in the floating point operations instruction set. For example, in GasKinetics module the lines that update the species in updateKc function, which directly influence the output, were found to be 'Moderate' and 'Critical' amongst the different fault models.

```
doublereal rrt = 1.0/(GasConstant * thermo().temperature());
...
m_rkc[irxn] = exp(m_rkc[irxn]*rrt -
          m_dn[irxn]*logStandConc);
```

Moreover, much like the hypothesis made in the Jacobi study case, faults injected solely to fault point operations in the hydrogen ignition case caused almost no segmentation faults and / or timeouts, due to the calculations they execute, which are by large not on the critical path of the program flow.

5.4.3 Memory

Table 5.6 shows the results of injecting at most 1,000 faults into the memory instruction set of the hydrogen ignition case. In addition Table 5.7 presents the results for the Random Bit Flips fault model for the methane ignition case, injecting 2 faults to each profiled source line to achieve 100% line coverage. As shown in both tables, the Random Bit Flips model generates roughly 66% 'Acceptable' faults, with the rest 33% faults as 'Moderate' or 'Critical'. In both experiments there were only three instructions that were profiled: call, invoke and load. Comparing the combined results results to the results from Table 3.5, which shows the analysis of the Jacobi program, there are huge differences. First, there are many more 'Critical' faults in the hydrogen ignition case, while the 'Moderate' faults stay around the same range of around $1\% - 10\%$. Moreover, in the hydrogen ignition case, almost all of the 'Critical' faults were a result of a segmentation fault, again, unlike the Jacobi program as analyzed in Section 3.5.1. Last but not least, every fault model, except for the XOR model, caused between $1 - 2$ timeouts in the hydrogen ignition case. These strike differences between the two programs, makes the injection of faults into memory operations program dependent, and apparent that memory operations are critical for the ignition cases.

Fault Type	**Random Bit Flips**	**Sticking Bits**	**XOR**	**Bit Flip**	**Byte Flip**	**Buffer Flip**
Acceptable	636	753	529	562	549	425
Moderate	41	23	73	48	50	40
Critical	183	84	258	250	281	395

Table 5.6: Comparison of generated fault types by fault model, based on the Hydrogen ignition case, with profiling memory operations, and 1,000 fault quota.

Fault Type	Random Bit Flips
Acceptable	1645
Moderate	74
Critical	822
Segmentation Faults	816
Timeouts	3

Table 5.7: Results for the Random Bit Flips fault model when profiling memory instructions based for the Methane ignition case and 100% line coverage.

Results Analysis of Ignition Cases

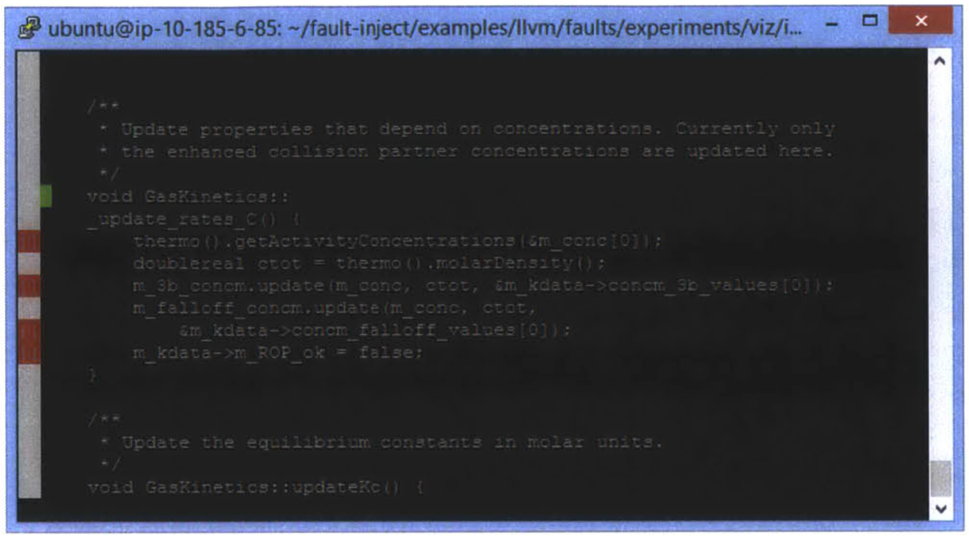
The results of the memory instructions across the ignition cases were consistent as many faults that were injected into memory operations, caused many 'Critical' faults, while almost no 'Moderate' faults. Figure 5-2 shows an example of how the `_update_rates_C` function in the `GasKinetics` module, which is mainly composed of memory instructions, caused all 'Critical' faults. A possible explanation to this would be that multiple memory instructions are encapsulated within each line, and a deviation in each one, could potentially cause a fault. For example, the line

```
m_3b_concm.update(m_conc, ctot, &m_kdata->concm_3b_values[0]);
```

accesses an array location in the function call itself, which could cause an error if the array output is invalid. The reason for the parity with the Jacobi program could be that there are many more memory accesses, and therefore more potential deviation in the memory calculation due to the larger memory size allocated to the ignition cases. However, it remains that memory instructions cause little to no timeouts, as suggested in Section 3.5.1.

5.4.4 All

Tables 3.6,5.8 present the results of injecting faults into all instructions. As observed, without differentiation of the instruction sets faults are injected to, the program are error prone, causing on average 50% 'Moderate' and 'Critical' faults. The ignition cases have many more instructions that are profiled and have faults injected to them: call, invoke, getelementptr, bitcast, icmp, load, sext, add, zext, shl, ptrtoint, sub,


ubuntu@ip-10-185-6-85: ~/fault-inject/examples/llvm/faults/experiments/viz/i...

```
/**
 * Update properties that depend on concentrations. Currently only
 * the enhanced collision partner concentrations are updated here.
 */
void GasKinetics::
_update_rates_C() {
    thermo().getActivityConcentrations(&m_conc[0]);
    doublereal ctot = thermo().molarDensity();
    m_3b_concm.update(m_conc, ctot, &m_kdata->concm_3b_values[0]);
    m_falloff_concm.update(m_conc, ctot,
        &m_kdata->concm_falloff_values[0]);
    m_kdata->m_ROP_ok = false;
}

/**
 * Update the equilibrium constants in molar units.
 */
void GasKinetics::updateKc() {
```

Figure 5-2: A sample of `GasKinetics.cpp` from the Cantera library, showing the different fault outcomes, using the Bit Flip fault model on Memory instructions.

ashr, extractvalue, and, trunc, ret, inttoptr, insertvalue, fcmp, sitofp, fmul, fadd, fsub, fptosi, sdiv, mul, fdiv, store, xor, or, br, select, lshr. The additional opcodes such as ptrtoint, extractvalue, inttoptr, insertvalue, amongst the many different ones, are critical to the proper flow of the program and the output, as they convert a point to an integer, extract a value from an array or a struct, covert an integer to a pointer, or insert a value to an array or a struct, respectively. The two major opcodes that are utilized in the ignition cases are getelementptr and load, which pertain to memory operations, collectively making up 66% of executed opcodes. As observed from the analysis of the memory operations, those are error prone, with causing roughly 33% 'Critical' faults on average. The third most executed opcode is icmp is a logical comparison opcode, which makes up about 12% of total executions. As the icmp pertains to the logical operations, such as conditionals, it is very likely that faults injected to those operations would produce a deviation in the output.The other 25% of the executions are distributed amongst the other 31 opcodes listed above, with the majority (over 50%) of them belonging to binary instruction set, which are known to be fault prone, possibly explaining the makeup of the rest of the 'Moderate' and 'Critical' faults.

Fault Type	**Random Bit Flips**	**Sticking Bits**	**XOR**	**Bit Flip**	**Byte Flip**	**Buffer Flip**
Acceptable	559	822	401	485	450	309
Moderate	39	27	41	45	57	37
Critical	450	199	606	518	541	702
Segmentation Faults	420	157	564	486	509	656
Timeouts	20	40	38	27	28	38

Table 5.8: Comparison of generated fault types by fault model, based on the Hydrogen ignition case, with profiling all instructions operations, and $1,000$ fault quota.

Chapter 6

Discussion

The Fault Prophet fault injection tool provides the ability to test the resilience of programs to faults caused by SEU (Single Event Upsets), by identifying the critical sections in the program, and visualizing the findings. The additional value on top of existing fault injection tools: greater modularity by abstracting the input and output of the different components of the tool, choice of fault scenarios, and a visualizations framework supporting an interactive work environment.

6.1 Suggested Preventive Actions

Several commonalities were found amongst the results of the experiments, enabling suggestions for fault preventive actions.

6.1.1 Instructions Separation

For code cleanliness and optimizations, multiple lines of code can be combined into a single line. For example, when loading a value from an array, the index can be calculated in the memory access code, as shown below from the Jacobi program.

```
double u1 = uold[i-1][j];
```

While this is a typical line of code, combining the index calculation with the memory access, separating the two can provide means to mitigate errors, as the index calculation can be verified for correctness before the memory access. While this creates an overhead, it is only for memory index and address calculations that could salvage an execution of a program.

6.1.2 Conditionals

As observed from the results analysis, timeouts do occur in program due to soft-errors in binary instructions, usually due to a change in the execution flow extending the execution time. Moreover, if variables are calculated in an iterative manner, additional iterations can cause deviations in the output of the program, which are more critical in loops, as such deviations are compounded. An example of such a case is in the main function of the ignition cases, in which a loop checks for cantera errors in the reaction. Additional iterations would change the calculated error deviation, producing a different results.

```
while (errFlag)
{
    /* Repeat to resolve errors. */
    ctrErr++;
    tFPerturb = 1.0e-5 * ctrErr;
    errFlag = chem.react(tFPerturb);
}
```

To mitigate the consequences of soft-errors on the execution flow and output, and allow normal operation of the program, an additional conditional check in conditional iterations, could serve as a fail-safe mechanism to stop extraneous iterations from taking place. While this will cause additional overhead, it is only for conditionals, and would protect critical calculations.

6.2 Design Challenges

6.2.1 Scalability

As the development of the tool progressed, larger programs were experiments with as described in Section 5.1. Those larger programs exhibited a certain behavior that was unnoticed beforehand - the size of the log outputted by the Profiler module, grew from MBs to GBs. As a consequence, objects that were once held in memory, grew too big, impeding the system, and at times causing errors in interpreter itself. To cope with these problems, the tool uses *sqlite3* to store the data, allowing the tool unlimited space to hold objects needed. However, this comes at a cost to performance, as sqlite3, and sql in general are not optimized to perform the queries the tool needs; moreover, pulling data from a database is much slower than in-memory objects. Therefore, the tool uses a hybrid approach, to store minimal data that is essential for most fault selection calculations in-memory while keeping the log outputs in a database. This hybrid approach is able to reduce the size of the log by normalizing the data by 75% as well as select faults within a reasonable amount of time, ranging from seconds to a few hours, depending on the program size. Additional optimizations could be performed to reduce the time for fault selection even further.

6.3 Future Work

6.3.1 Code Optimizations

There are multiple code optimizations that could reduce the overall runtime of the tool.

SQL queries

The transition to a Sqlite3 database to hold and retain the data of the profiled instructions, and the injected faults has a performance cost as mentioned in Section 6.2.1. Proper, normalized and optimized database choice, structure and queries can reduce the average runtime of the tool.

Redundant Fault Selection Analysis Runs

Under ideal settings, the Fault Selection Analysis module would have to run only once per profile option, i.e. instruction set, and will save that data for future iterations of the tool that use the same profiling option. While the current architecture of the tool cannot support this feature, profiling a fairly sized program execution creates a bearable overhead as discussed in Section 6.2.1.

6.3.2 Fault Model Factory for Function Calls and Arrays

The current architecture of the tool supports different fault models only for two fault scenarios: single fault and burst mode (see Section 4.4.3. While the current fault model for function calls and array faults is to replace the existing buffer, whether call instruction of pointer, with a generated random value, as shown in the Section 5.3, the fault model has a great impact on the outcomes. As such, providing support for multiple fault models is essential to fully support the identification of critical section via a myriad of fault scenarios.

6.3.3 Fault Injection into Operands

The tool was designed to inject faults into instruction buffer. However, there are times in which there is a need to test the behavior of an instruction when an operand changes. Therefore, there is a need to inject faults into a more granular object than instructions. As demonstrated by the array fault scenario, which modifies only pointers in an instruction, this is possible, and should be exploited whenever possible.

6.3.4 Profiling Options

Loops

Loops are a critical asset of any program, and there could be variations in the sensitivity of the program in general, and of critical sections in particular, if a fault in injected in a loop, depending on which iteration the fault is injected. A loops profiling option would measure the number of times a loop runs, and will allow to inject faults to the loop in the first and last iterations.

6.3.5 Faulty Trace Analysis

As mentioned in Section 1.1.2, to measure the sensitivity of a fault, the tool compares the faulty output to the baseline output, since it is cheapest analysis in terms of resources. However, comparing a faulty trace comparison to the baseline trace would provide a more accurate analysis; this requires optimizations on the trace and the storage mechanism of the traces, as they can take up a lot of space, as well as the comparison if the traces are long.

6.3.6 Extension of the Visualizations Framework

A possible extension of the visualization framework, could offer comparison of the fault models. Given faults were injected using multiple fault models, a visualization, much like a heat map, based on the composite score on all fault models, could provide a better visual identification and indication of the sensitivity of critical sections in the program.

Appendix A

Documentation

This is the documentation of how to use the Fault Prophet Fault Injection Tool, and the Vim plugins designed for it.

A.1 Fault Prophet

After the tool has been installed, the main resource users will have to deal with is the configuration file of the tool. The configuration file allows the user to setup all the options pertaining to the program they want to run their tool on. Once the configuration file has been set, and the program compiled, all that is left to do is run the tool by running:

```
python script.py <name of configuration file>
```

A.1.1 Installation

it is recommended to use Ubuntu 12.04.3 with the tool.

Preconfigred Virtual Machine

A virtual machine has been preconfigured with the tool that can be obtained by contacting the Computer Architecture research group at CSAIL, Computer Science and Artificial Intelligence Lab. The user name of the machine is 'neo' and the password is '1234'. The tool itself can be found under the home folder in the fault-inject folder.

A.1.2 Compilation Suggestions

In order for the tool to provide the best possible data on the tested program, it is suggested to compile the program with the '-O0 -g' flags. The '-O0' will not optimize the program, allowing a mapping of instructions to source line with utmost accuracy. The '-g' **must** be used in order to map each instruction to its source code file and source line. This is especially necessary if the user would like to make use of the visualization and vim plugins.

A.1.3 Configuration

There are seven sections in the settings file: general, scripts, filter, modes, profiler, faultinjection and analysis. Each section has one or many options that can be set. While not all options are mandatory, it is recommended the user will fill all the options, in order to make sure the output is to their liking. Please see Figure B-2 for an example of a configuration file. Settings marked with (*) the asterisk mark are mandatory settings.

General section

This section includes three mandatory settings that must be set.

- Program Name * (string) : the name of the executable without the .bc extension. The tool expects the executable to reside in the same directory as the tool itself.
- Program Input * (string) : the path of the file that has the program inputs. If no inputs are needed or required, a path to file must still be specified, and the file must exist. In such a case, use the *touch* command.
- SO-Libs * (string) : the path to the Fault Prophet LLVM libraries. In accordance with the installation script, those will exist in <specified home path>/llvm/llvm-2.9/Relase+Asserts/lib

Scripts section

This section specifies the path of the scripts that the tool runs: the profiler, the fault injection, fault injector module compilation, and fault analysis. Those settings should not be modified unless there is a reason to do so.

- run-profiler * (string) : profiler script path. The default value is './run_profiler.sh'.
- run-inject * (string) : fault injection script path. The default value is './run_inject.sh'.
- run-inject-compile * (string) : fault injection compilation script path. The default value is './faultinject-compile.pl'.

Filter section

This section pertains to the settings of the Fault Selection Analysis module.

- white-list-filters * (special format) : the instructions sets that will be profiled.
  - Note: The tool allows a user to create a filtering tool themselves, and run it through the tool. As such, the tool allows the user to specify a custom filtering script using the following format:

    [<filter script 1>,<filter option 1>];[<filter script 2>,<filter option 2>]

    The tool includes a filter tool that can be used with the following parameters:

    [./run_filters.sh,<profiling option>]

    where the profiling option can be picked from the following:
    - a : all instructions
    - b : binary operations (include floating point operations)
    - f : floating point operations
    - c : function calls
    - v : arrays

        * m : memory operations
        * u : use-define chains
        * s : all instructions except invoke and cast

- black-list-filters (special format) : the instructions sets that will be filtered out. This option takes precedence over the white-list-filters options.
  - Note: Like the white-list-filters, the black-list-filters can be created by the user, and as such, the format for this setting is the same as for the white-list-filters. Please see note in white-list-filters to understand the input format.
- filter-instrs (comma separated strings) : comma separated list of opcodes that will be filtered out, regardless of whether they are a part of the profiled instruction set(s).
- exclude-lines : Source lines from specific source files that will be filtered out, if instructions on those lines are being profiled.
  - Note: the tool allows a user to specify any source file name, and corresponding line numbers. As such the format is:

    [<source file name 1, including extension>,<starting line number 1>,<ending line number1>];[<source file name 2, including extension>,<starting line number2>,<ending line number2>]

    The starting line number must be less than or equal to the ending line number, for those lines to be included.
- include-lines (special format) : the opposite of exclude-lines. This settings allows the user to specify the source lines from specific source files that will be included in the profiling.
  - Note: uses the same formatting as exclude-lines option. Please refer to the note under that setting to obtain information about the input format for this setting.
- is-skip * (boolean) : allows the user to specify whether lines will be excluded or included, as only one option can be used.The default value is True, meaning to use the exclude lines option, rather than include lines.

Modes Section

- run-native (boolean) : the mode in which the program needs to run, whether native, i.e. the program will be compiled to an executable, or not, i.e. the LLVM interpreter, lli, will be used, as some program can run only in one mode out of the two. The default value is False.
- run-time * (integer) : the alloted runtime in seconds. If the program exceeds this time, it will be timed-out. The input time should be the original runtime, i.e., the average amount of time it takes the program to run, as the tool accounts for the overhead costs.

Profiler Section

This section pertains to the settings of the Profiler module.

- trace (boolean) : an option to dynamically trace instructions for each fault injection. The default value is False.
  - Note: The overhead costs might be expensive, if set to True.
- static-trace (boolean) : output a static trace of the program. The default value is False.
  - Note: This must be True in order to use the InstShow vim plugin described in Section 4.5.3.

Fault-Injection Section

This section pertains to the settings of the Fault Injector module.

- quota * (integer) : number of faults to inject to the program.
- burst-mode (boolean) : use the burst of faults scenario. The default value is False.
- burst-repeat (integer) : if in burst mode, this option specifies the number of instructions to pass before injecting a fault.

- burst-num-faults (integer) : if in burst mode, this option specifies the total number of faults to inject.
- repeat-fault (integer) : specifies the number of faults to inject to the same buffer. This does not count against the quota, meaning $quota * repeat - fault$ faults will be injected total. The default value is 1.
  - Note: This option is useful if the fault model is based on random values, allowing to exhaustively test the given buffer.
- test-line-coverage (boolean) : if True, given a percentage, the tool will stop injecting faults into the program once the percentage of source lines have been tested. The tool will inject two faults into each source line in each source file, regardless of number of instructions it encapsulates or the number of instruction executions it has, making the quota parameter extraneous in this case.
- line-coverage-percent (boolean in rage of 0-100) : the percentage of source lines of those with profiled instructions to inject faults to.
- func-faults (boolean) : use of function call and arrays fault scenarios described in Section 4.4.2.
  - Note: If the intended use is for array faults, then must specify 'v' as a profiled instruction set in the white-list-filter setting.
- func-fault-prob (float) : the probability a fault will be injected to a function call if func-faults is True.
- arr-fault-prob (float) : the probability a fault will be injected to an instruction that has an array if func-faults is True, and the instruction set for arrays is profiled.
- fault-model * (integer) : the choice of fault model. The fault model is assigned a number by the fault factory as described in Section 4.4.3.
  - Note: Given the current fault factory settings, the fault models described in Section 4.4.3 are mapped to the following numbers:
    * 0 : Flip Random Bits
    * 1 : Sticking Random Bits
    * 2 : XOR Mantissa Bits

    * 3 : Flip a Random Bit
    * 4 : Flip a Random Byte
    * 5 : New Buffer

Analysis Section

- output-file * (string) : name of the file where to save the results of the output abstraction script.
- output-abstraction * (string) : path to the output abstraction tool.
- show-viz (boolean) : the option whether to display the visualizations. The default value is False.

A.2 Vim Plugin

The FltMark vim plugin allows the user to inject faults directly into the program itself from vim as described in Section 4.5.3. However, preliminary steps must take place before this functionality can be used.

A.2.1 Configuration File

In order to utilize functionality of injecting faults directly from vim, the user must first create a configuration file that is suitable for use by the plugin. The configuration script lets the user generate a suitable configuration file, by copying the settings it needs from an existing configuration file, by typing:

```
python config.py <path of existing configuration file>
```

into the terminal window. This command will generate a new file cfgCreator.py that will be used by the vim plugin whenever fault are injected through vim. Some settings are modified from the original settings, whatever they might be. The plugin configuration file will profile all instruction sets, filter only the selected lines and will inject 2 faults to instructions executed in each of those lines (see Section A.2.3). Moreover, the configuration will be modified to save no traces, static or dynamic, of those fault injections to optimize the fault injection time.

A.2.2 Source Line Sensitivity

There are three commands to control whether to show the sensitivity of source lines in a source file that is opened with vim, given that Fault Prophet was used beforehand.

FaultMarkersEnable

This command enables the sensitivity view in vim.

FaultMarkersDisable

This command disables the sensitivity view in vim.

FaultMarkersOnOff

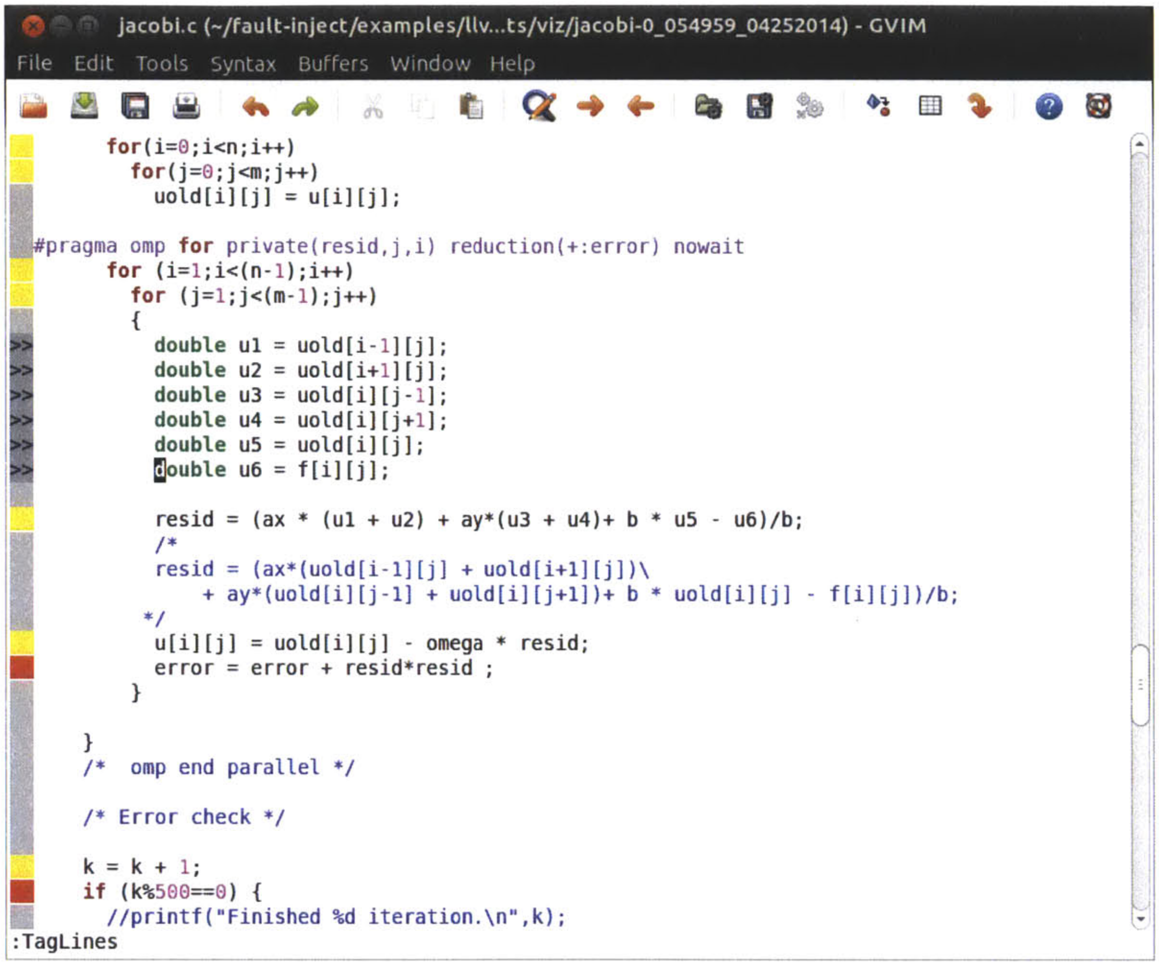
This command enables or disables the sensitivity view in vim, depending on the current view.

A.2.3 Fault Injection

There are two classes of commands that are supported by the plugin to inject faults: TagLine(s) and InjectFault(s).

TagLine(s)

The TagLine command tags the current line the user is on for fault injection. Therefore, the user can tag multiple lines and inject faults into all those lines at once. The TagLines command allows the user to tag multiple lines at once. However, the user must enter the command twice: once for the start line, and another for the end line. The order of the start line and end line does not matter.


jacobi.c (~/fault-inject/examples/llv...ts/viz/jacobi-0_054959_04252014) - GVIM
File Edit Tools Syntax Buffers Window Help

```
      for(i=0;i<n;i++)
        for(j=0;j<m;j++)
          uold[i][j] = u[i][j];

#pragma omp for private(resid,j,i) reduction(+:error) nowait
      for (i=1;i<(n-1);i++)
        for (j=1;j<(m-1);j++)
        {
>>        double u1 = uold[i-1][j];
>>        double u2 = uold[i+1][j];
>>        double u3 = uold[i][j-1];
>>        double u4 = uold[i][j+1];
>>        double u5 = uold[i][j];
>>        double u6 = f[i][j];

          resid = (ax * (u1 + u2) + ay*(u3 + u4)+ b * u5 - u6)/b;
          /*
          resid = (ax*(uold[i-1][j] + uold[i+1][j])\
              + ay*(uold[i][j-1] + uold[i][j+1])+ b * uold[i][j] - f[i][j])/b;
         */
          u[i][j] = uold[i][j] - omega * resid;
          error = error + resid*resid ;
        }

    }
    /*  omp end parallel */

    /* Error check */

    k = k + 1;
    if (k%500==0) {
      //printf("Finished %d iteration.\n",k);
:TagLines
```

Figure A-1: Example of tagged lines in FltMark vim plugin.

InjectFault(s)

The InjectFaults command injects faults to tagged lines. If no lines are tagged, the command will not run, and will alert the user lines need to be tagged first. The InjectFault command on the other hand, injects faults to the current line the user is on, regardless of any lines are tagged. The tagged lines will be removed once the fault have been injected. In order to inject faults, the tool must reside in the same directory as the source file, as well as the binary on which it will execute.

Appendix B

Figures

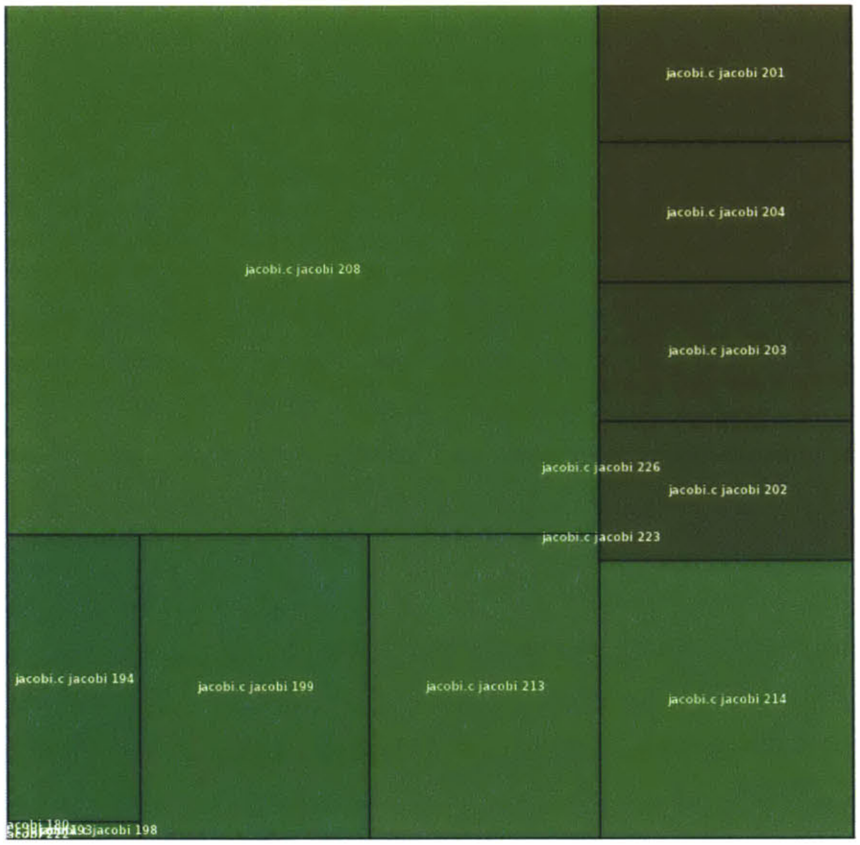


Figure B-1: Treemap example of showing source line breakdown.

```
[general]
# program name without .bc extension
program-name = jacobi

# must specify an input file and must exist
program-input = input.jacobi

# fault injection libs
so-libs = /home/neo/llvm/llvm-2.9/Release+Asserts/lib

[scripts]
run-profiler = ./run_profiler.sh
run-inject = ./run_inject.sh
run-inject-compile = ./faultinject-compile.pl

[filter]

# profiled instructions to include (white) exclude (black),
# include script and profile option
# separate scripts and options using ;
# example: [./run_filters.sh,b];[./run_filters.sh,f]
# profile options:
# a - all insturctions
# b - binary operations (include float)
# f - float operations
# c - function calls
# v - arrays
# m - memory operations
# u - usedef chains
# s - backward slice = all except invoke and cast

white-list-filters = [./run_filters.sh,f]
black-list-filters =

# instructions to exclude from analysis and fault injection
filter-instrs = getelementptr,sext
```

```

# profiled lines to include or exclude
# include the source file name and starting and ending line
    numbers,
# both inclusive the format is [src file name, start line, end
    line]
# include additional files and line numbers by separating them
    with ;
# must specify if excluding or including by setting skip to True
    if
# skipping, else would include

exclude-lines = [jacobi.c,229,249] #error_check function
include-lines =
is-skip = True

[modes]
# Whether to run the program in native mode, i.e. create an
    executable
run-native = True

# Runtime in secs
run-time = 5

[profiler]
# True to dynamically trace instructions for each fault injection
trace = False

# Static trace of program.
# This must be True in order to use the GUI component
static-trace = True

[faultinjection]
# number of faults to inject
quota = 1000

```

```
# True for burst mode, otherwise False
burst-mode = False

# number of instructions to jump before injecting a fault in
    burst mode
burst-repeat = 2

# number of total faults to inject in fault mode
burst-num-faults = 5

# number of fault to inject in the same buffer
# (useful if flipping a random bit or byte)
repeat-fault = 1

# line coverage percentage
# if test-line-coverage is set to true, then won't reach quota,
# but will stop when reaching line coverage percentage
test-line-coverage = False
line-coverage-percent = 90

# fault injection noise
noise-max = 11000
noise-control = 0.002

# True if want to inject faults to function calls, false
    otherwise
# if True, must specify "c", function calls to be profiled in
    filter section
func-faults = False

# choose a fault model by specifying the number of fault model
# 0 - flip random bits
# 1 - xor random bits
# 2 - xor entire buffer
# 3 - flip a random bit in buffer
# 4 - flip a random byte in buffer
# 5 - flip the entire buffer with random bytes
# 6 - stick noise in the lower bits (mantissa when float)
# for 6, you can set the noise level under fault-injection noise

fault-model = 5

[analysis]
output-file = summary.txt
output-abstraction = distortion.py
mean-green-lower = 0.0
mean-green-upper = 0.0
mean-yellow-lower = 0.0
mean-yellow-upper = 1.0
mean-red-lower = 1.0
mean-red-upper = inf
mean-segfault = 10000000.0
mean-default = critical
show-viz = True
```

Figure B-2: A sample configuration file of Fault-Prophet for the Jacobi program.

```
import os.path
import subprocess

settings =      {'mean-green-lower' : 0.0,
                 'mean-green-upper': 0.0,
                 'mean-yellow-lower' : 0.0,
                 'mean-yellow-upper' : 1.0,
                 'mean-red-lower' : 1.0,
                 'mean-red-upper' : float('inf'),
                 'mean-segfault' : -1,
                 'mean-default' : 'critical',
                 'output-abstraction': 'distortion.py'}

class OutputAbs:
    def __init__(self):
        pass

    def calculate_score(self, greens, yellows, reds):
        score = 1.0 * (yellows * 0.5 + reds) / (greens + yellows + reds)
        classification = self.classify_fault(score)
        return score, classification

    def output_analysis(self, fault_output, fault_path, baseline_path):
        output = ""
        mean_ = 0.0
        if fault_output == "":
            mean_ = settings["mean-segfault"]
            output = "Segmentation␣Fault"
        elif fault_output == "TimeOut":
            mean_ = settings["mean-segfault"]
            output = "Time␣Out"
        else:
            proc = subprocess.Popen(['python', settings["output-abstraction"],
                baseline_path, fault_path], stdout=subprocess.
                   PIPE)
            output = proc.stdout.read()
            _meanindx = output.find("Mean")
            if _meanindx != -1:
                    _meanindx += 5
                    mean_ = output[_meanindx : output.find("\n",
                        _meanindx)]
                    mean_ = float(mean_)
            output = output.strip().replace("␣", "").replace("\n"
               , "␣")

        return output, mean_

    def classify_fault(self, mean_):

        classification = settings["mean-default"]

        if mean_ == settings["mean-segfault"]:
                classification = "critical"
        elif settings["mean-green-lower"] <= mean_
                and mean_ <= settings["mean-green-upper"]:
                classification = "acceptable"
        elif settings["mean-yellow-lower"] < mean_
                and mean_ <= settings["mean-yellow-upper"]:
                classification = "moderate"
        elif settings["mean-red-lower"] < mean_
                and mean_ <= settings["mean-red-upper"]:
                classification = "critical"

        return classification

    def analyze_fault(self, baseline_path, fault_path):

        if not os.path.exists(fault_path):
                return
        fault_file = open(fault_path)
        fault_output = fault_file.read().strip()
        fault_file.close()

        output, mean_ = self.output_analysis(fault_output,
            fault_path, baseline_path)
        classification = self.classify_fault(mean_)

        return output, classification
```

Figure B-3: The output abstraction wrapper for the analysis of the Jacobi program.

```
#!/usr/bin/env python

import sys
import os
import re
import math
import numpy
from pprint import pprint
from optparse import OptionParser


STATS = {
          "min" : numpy.min ,
          "max" : numpy.max,
          "mean" : numpy.mean,
          "median" : numpy.median,
          "stdev" : numpy.std,
          "range" : lambda x : numpy.max(x) - numpy.min(x),
          "zeros" : lambda x : 100.0*len([z for z in x if z == 0])
             /len(x)
         }

STATS_DESCRIPTION = [
                       ("min", "Minimum") ,
                       ("mean", "Mean"),
                       ("median", "Median"),
                       ("max", "Maximum"),
                       ("stdev", "Standard␣Deviation"),
                       ("range", "Range"),
                       ("zeros", "Zero␣elements")
                      ]

ROUND_PLACES = 10

## For all errors, the first argument is the reference value, the
     other is the comparison value
```

```
ERR_FUNCTION = {
                    "abserr"    :  lambda x, y: math.fabs(x - y),
                    "absrelerr" :  lambda x, y: math.fabs(1-y/x) if
                        0 != round(x, ROUND_PLACES) else (1e+308)
                       if 0 != round(y, ROUND_PLACES) else 0.0,
                    "err"     :  lambda x, y: x - y,
                    "relerr" :  lambda x, y: 1-y/x if 0 != round(x,
                        ROUND_PLACES) else (1e+308) if 0 != round(y
                       , ROUND_PLACES) else 0.0,
                 }

ERR_FUNCTION_DESCRIPTION = [
                                ("err", "Error"),
                                ("relerr", "Relative␣Error"),
                                ("abserr", "Absolute␣Error"),
                                ("absrelerr", "Absolute␣Relative␣
                                    Error")
                              ]

ERR_FUNCTION_DEFAULT = "absrelerr"
#ERR_FUNCTION_DEFAULT = "relerr"

WORD_DELIMITERS = "␣\t"
COMMENT_DELIMITERS = "#="
DIGIT_PRECISION = 10

##
## Calculates the distortion. Measures the whole contents of
## both files.
##

def readLines(FileInput, comments):
    ListOutput = []

    lcomments = [re.compile(c) for c in comments]

    for line0 in FileInput:
        line = line0.strip()
        if len(line) == 0: continue

        continueLoop=True
        for lcomment in lcomments:
            if re.match(lcomment, line):
                continueLoop = False
                break
        if not continueLoop: continue

        ListOutput.append(line)

    return ListOutput


def convertLineToNumbers(strLine, delimiters):

    r = re.compile('[%s]' % delimiters)
    return [float(s) for s in r.split(strLine) if len(s) > 0]


def convertToValues(ListStringInput, delimiters):
    ListIntOutput = []
    ListIntPerLineOutput = []

    for pl in ListStringInput:
        apl = convertLineToNumbers(pl, delimiters)
        ListIntOutput.extend(apl)
        ListIntPerLineOutput.append(len(apl))

    return ListIntOutput, ListIntPerLineOutput

def isNan(x):
    return type(x) is float and x != x

```

```

def calculateStats(ListDistortions, statsWhitelist = STATS.keys()):
    list = [d for d in ListDistortions if not isNan(d)]
    stats = dict( ( n , f (list) ) for n,f in STATS.items() if n in statsWhitelist)
    return stats


def readFromFile(inputFile, delimiters = WORD_DELIMITERS, comments = COMMENT_DELIMITERS):
    inputFile = open(inputFile, 'r');

    inputLines = readLines(inputFile, comments)
    inputValues, inputValuesPerLine = convertToValues(inputLines, delimiters)

    return inputValues, inputValuesPerLine



def calculateDistortions(refValues, cmpValues, errFunction, exitOnNaN = False):
    distortions = []
    for rv, cv in map(None, refValues, cmpValues):
        rv, cv = float(rv), float(cv)
        if (isNan(rv) or isNan(cv)):
            if not (isNan(cv) and isNan(rv)):
                if not exitOnNaN: print "[Warning]␣",
                print "Both␣values␣must␣be␣nan's␣(ref␣=␣%5.10f␣,␣cmp␣=␣%5.10f)" % (rv, cv)
                if exitOnNaN: exit(1)
            dist = 0.0
        dist = errFunction(rv, cv);
        distortions.append(dist)
    return distortions
    ###



def printDistortions(distortions, elementsStart = 0,
    elementsPerLine = [], digits = DIGIT_PRECISION, minimumPrint =
     False):

    format = "%%5.%df" % digits
    if elementsPerLine and not minimumPrint:
        start = 0
        for linecnt in elementsPerLine:
            print "%4d:␣%s" % (elementsStart + start, "␣".join([
                format % d for d in distortions[start : start +
                linecnt] ]))
            start += linecnt
    else:
        print "␣".join([str(d) for d in distortions])

def printSummary(summary, digits = DIGIT_PRECISION, minimumPrint
    = False):

    format = "%%5.%df" % digits
    for statId, statName in STATS_DESCRIPTION:
        try:
            s = summary[statId] ## throw if doesn't exist
            if not minimumPrint: print "%s:␣" % statName,
            print format % s
        except KeyError:
            pass ## this is fine, do nothing




def constructOptions():
    parser = OptionParser()

    parser.add_option("-c", "--comments",
                      action="store", dest="comments",
                      default=COMMENT_DELIMITERS,
                      help="Characters␣in␣input␣files␣that␣begins
                         ␣comments")
    parser.add_option("-d", "--delimiters",
                      action="store", dest="delimiters",
                      default=WORD_DELIMITERS,
                      help="Delimiters␣within␣line␣(default␣<spc
                         >,␣<tab>)")

#     parser.add_option("-m", "--mode",
#                       action="store", dest="mode",
#                       default=MODE_DEFAULT,
#                       help="Select mode of operation (one of %s)
    " % ", ".join(MODES))

    parser.add_option("-e", "--error",
                      action="store", dest="errorFunction",
                      default=ERR_FUNCTION_DEFAULT,
                      help="Error␣function␣for␣individual␣
                         elements␣(including␣following:␣%s;␣
                         default:␣%s)" %(",␣".join(ERR_FUNCTION.
                         keys()), ERR_FUNCTION_DEFAULT) )

    parser.add_option("", "--spartan",
                      action="store_true", dest="printSpartan",
                      default=False,
                      help="Print␣Minimum␣Additional␣Information"
                         )

    parser.add_option("-p", "--precision",
                      action="store", type="int", dest="precision
                         ",
                      default=DIGIT_PRECISION,
                          help="Number␣of␣significant␣digits␣while␣printing␣the␣output")

    parser.add_option("-f", "--field",
                          action="store", type="string", dest="fields",
                          default="all",
                          help="List␣of␣fields␣separated␣by␣comma␣or␣'all'␣for␣all␣fields")
    ## ^ todo add pattern langauge for this, such as odd, even, or a sequence....

    parser.add_option("", "--cmp-multi",
                          action="store_true", dest="cmpMultiple",
                          default=False,
                          help="Each␣line␣in␣comparison␣file␣corresponds␣to␣a␣different␣result")

    parser.add_option("-i", "--individual",
                          action="store_true", dest="printIndividual",
                          default=False,
                          help="Print␣individual␣distortions")

    parser.add_option("-s", "--skip-summary",
                          action="store_false", dest="printSummary",
                          default=True,
                          help="Print␣summary␣statistics")
    parser.add_option("-S", "--statistics",
                          action="store", dest="statistics",
                          default=",".join(STATS.keys()),
                          help="Summary␣statistics␣component␣(including␣following:␣%s,␣or␣'all')␣" % ",␣".join(STATS.keys()))

    return parser
```

```python
def prepareData(args, options):

    spartan = options.printSpartan

    if len(args) != 2:
        print "There␣should␣be␣exactly␣two␣arguments:␣a)␣<reference_src>␣and␣<comparison_src>"
        exit(1)

    if options.printSummary == False and options.printIndividual == False:
        print "Program␣must␣print␣at␣least␣one␣of␣summary␣or␣individual␣distortions␣(check␣parameters␣-s␣and␣-i)"
        exit(1)
    if options.printSummary and len(options.statistics) == 0:
        print "When␣printing␣summary␣at␣least␣one␣statistic␣must␣be␣chosen␣(check␣parameter␣-S)"
        exit(1)


    refInput = args[0]
    cmpInput = args[1]

    if options.statistics == "all": options.statistics = ",".join(STATS.keys())
    if options.errorFunction == "all": options.errorFunction = ",".join(ERR_FUNCTION.keys())

    if not spartan:
        #print "Reference Input: %s \nComparison Input: %s\n" % (refInput, cmpInput)
        #print "Options: "
        #pprint(options.__dict__, indent=4)
        #print "\n"
        pass


    refValues, elementsPerLine = readFromFile(refInput, delimiters = options.delimiters, comments= options.comments)
    cmpValues, _ = readFromFile(cmpInput, delimiters = options.delimiters, comments= options.comments)

    return refValues, cmpValues, elementsPerLine


def filterValues (refValues, cmpValues, options):

    if options.fields != "all":
        indices = [int(f) for f in options.fields.split(',')]
        refValues = [ refValues[i] for i in indices ]
        cmpValues = [ cmpValues[i] for i in indices ]

    return refValues, cmpValues



def computeDistortion(refValues, cmpValues, options):

    refValues, cmpValues = filterValues (refValues, cmpValues, options)

    if len(refValues) != len(cmpValues):
        print "Number␣of␣data␣elements␣must␣match␣"
        print "␣␣Presently␣ref␣=␣%d␣,␣cmp␣=␣%d" % ( len(refValues) , len(cmpValues) )
        exit(1)

    try:
        errorFunction = ERR_FUNCTION[options.errorFunction]
    except KeyError:
        print "Error␣function␣%s␣unknown!" % options.errorFunction
        exit(1)

    distortions = calculateDistortions(refValues, cmpValues, errorFunction)
    summary = calculateStats(distortions, statsWhitelist = options.statistics.split(','))
    return distortions, summary

def presentResults(distortions, summary, options, elementsStart = 0, elementsPerLine = []):

    spartan = options.printSpartan

    if options.printIndividual and distortions != None:
        s = "␣Print␣Individual␣Element␣Distortions␣"
        l = int(40 - len(s)/2)
        if not spartan: print "=" * l + s +  "=" * (l + len(s) % 2)
        printDistortions(distortions, elementsStart = elementsStart, elementsPerLine = elementsPerLine, digits = options.precision, minimumPrint = spartan)
        if not spartan and not options.printSummary: print "="*80
        if options.printSummary: print

    if options.printSummary and summary != None:
        s = "␣Print␣Distortion␣Summary␣"
        l = int(40 - len(s)/2)
        #if not spartan: print "="*l + s + "="*(l+len(s)%2)
        printSummary(summary, digits = options.precision, minimumPrint = spartan)
        #if not spartan: print "="*80

    #if options.cmpMultiple: print

    return 0


def main():
    parser = constructOptions()
    (options, args) = parser.parse_args(args = sys.argv[1:])

    refValues, cmpValues, elementsPerLine = prepareData(args,
        options)

    rl, cl = len(refValues), len(cmpValues)

    if rl == 0:
        print "At␣least␣one␣reference␣value␣must␣exist!"
        exit(1)

    if options.cmpMultiple:
        if cl <= rl and cl % rl != 0:
            print "The␣number␣of␣comparison␣values␣(%d)␣should␣be
                ␣a␣multiplicative␣of␣the␣reference␣values␣(%d)" %
                (cl, rl)
            exit(1)
    else:
        if cl != rl:
            print "The␣number␣of␣comparison␣values␣(%d)␣should␣be
                ␣equal␣to␣the␣number␣of␣reference␣values␣(%d)" % (
                cl, rl)
            exit(1)

    start = 0
    totaldistortions = []
    while start < cl:
        distortion, summary = computeDistortion( refValues,
            cmpValues[start : start + rl] , options)
        totaldistortions.extend(distortion)
        presentResults(distortion, None if options.cmpMultiple
            else summary, options, elementsStart = start,
            elementsPerLine = elementsPerLine)
        start += rl

    if options.cmpMultiple:
        summary = calculateStats(totaldistortions, statsWhitelist
             = options.statistics.split(','))
        presentResults(None, summary, options)


if __name__ == "__main__":
    main()
```

Figure B-4: The output abstraction algorithm used for the analysis of the Jacobi program.

```
#!/bin/bash

export PRG_NAME="main.x"
export PRG_INPUT="input.main.x"

export PRG_QUOTA="1000"

export FILTER_BLACK_LIST=""
export FILTER_INSTRS=""
export FILTER_EXCLUDE_LINES=""
export FILTER_INCLUDE_LINES=""
export FILTER_IS_SKIP="True"

export PRG_RUN_NATIVE="False"
export PRG_RUN_TIME="20"

export FI_REPEAT_FAULT="1"
export FI_FUNC_FAULTS="True"

export FI_LINE_COVERAGE="True"
export FI_LINE_COVERAGE_FAULTS="2"
export FI_LINE_COVERAGE_PERCENT="100"

export FI_NOISE_MAX="1000"
export FI_NOISE_CONTROL="0.123"

for option in 'b' 'f' 'm' 'a'
do
        export OPTION_OUTPUT_DIR="$PRG_NAME-$option$(date +_%H%M%
            S_%m%d%Y)"
        mkdir $OPTION_OUTPUT_DIR

        export FILTER_WHITE_LIST="[./run_filters.sh,$option]"
        for i in $(seq 0 5)
        do
                export PRG_FAULT_MODEL=$i
                export PRG_SETTINGS_PATH="settings-$PRG_NAME$(date +_%H%M_%m%d%Y).cfg"

                python configCreator.py
                python script.py $PRG_SETTINGS_PATH

                export OUTPUT_DIR="$PRG_NAME-$PRG_FAULT_MODEL$(date +_%H%M%S_%m%d%Y)"

                mkdir $OUTPUT_DIR

                mv $PRG_NAME\_output $OUTPUT_DIR
                mv $PRG_NAME\_baseline $OUTPUT_DIR
                mv faults $OUTPUT_DIR
                mv analysis $OUTPUT_DIR
                mv trace $OUTPUT_DIR
                mv logs $OUTPUT_DIR
                mv summary.txt $OUTPUT_DIR
                mv $PRG_SETTINGS_PATH $OUTPUT_DIR
                mv $PRG_NAME.mem2reg.bc.analysis.log $OUTPUT_DIR
                mv $PRG_NAME.mem2reg.bc.log $OUTPUT_DIR
                mv $PRG_NAME.db $OUTPUT_DIR
                mv fltmark* $OUTPUT_DIR
                tar -zcf "$OUTPUT_DIR.tar.gz" $OUTPUT_DIR
                rm -rf $OUTPUT_DIR
                mv "$OUTPUT_DIR.tar.gz" $OPTION_OUTPUT_DIR
        done
done
```

Figure B-5: The experiments script used to run the tool for the evaluation of both the tool and programs.

Bibliography

[1] Robert Baumann. Soft errors in advanced computer systems. *Design & Test of Computers, IEEE*, 22(3):258–266, 2005.

[2] Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[3] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[4] Seungjae Han, Kang G Shin, and Harold A Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213. IEEE, 1995.

[5] Siva Kumar Sastry Hari, Sarita V Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 123–134. ACM, 2012.

[6] Dafydd Harries. Pyflakes. `https://launchpad.net/pyflakes`, July 2005. Accessed: 2013-12-25.

[7] John G. Holm and Prithviraj Banerjee. Low cost concurrent error detection in a vliw architecture using replicated instructions. In *Proceedings of the International Conference on Parallel Processing*, pages 192–195, 1992.

[8] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: a flexible software-based fault and error injection system. *Computers, IEEE Transactions on*, 44(2):248–260, 1995.

[9] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[10] Dong Li, Jeffrey S Vetter, and Weikuan Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In

Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 57:1–57:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press, IEEE Computer Society Press.

[11] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[12] Robert E. Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[13] Yesudeep Mangalapilly. Watchdog. `http://pythonhosted.org//watchdog/`, October 2010. Accessed: 2014-04-30.

[14] Francis P. Mathur and Algirdas Avižienis. Reliability analysis and architecture of a hybrid-redundant digital system: generalized triple modular redundancy with self-repair. In *Proceedings of the May 5-7, 1970, spring joint computer conference*, AFIPS '70 (Spring), pages 375–383, New York, NY, USA, 1970. ACM.

[15] Asit K Mishra, Rajkishore Barik, and Somnath Paul. iact: A software-hardware framework for understanding the scope of approximate computing. 2014.

[16] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63–75, 2002.

[17] Karthik Pattabiraman, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 472–481. IEEE, 2008.

[18] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4):366–396, December 2005.

[19] Joseph Robicheaux. Jacobi.f. `http://www.openmp.org/samples/jacobi.f`, 1998. Accessed: 2014-04-23.

[20] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng-Fai Wong. Asac: Automatic sensitivity analysis for approximate computing. 2014.

[21] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

[22] Laszlo Simon. Treemap. `https://launchpad.net/treemap`, March 2010. Accessed: 2014-03-16.

[23] Anna Thomas and Karthik Pattabiraman. Error detector placement for soft computation. In *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2013.

[24] Anna Thomas and Karthik Pattabiraman. Llfi: An intermediate code level fault injector for soft computing applications. *ser. SELSE*, 2013.

[25] Timothy K Tsai and Ravishankar K Iyer. FTAPE: A fault injection tool to measure fault tolerance. In *AIAA, Computing in Aerospace Conference, 10 th, San Antonio, TX*, pages 339–346, 1995.