Copyright
by
Ameya Suhas Chaudhari
2014

The Thesis committee for Ameya Suhas Chaudhari

Certifies that this is the approved version of the following thesis:

Fiesta++: A software implemented fault injection tool for transient fault injection

APPROVED BY

SUPERVISING COMMITTEE:

____

Jacob Abraham, Supervisor

____

Saurabh Bagchi

Fiesta++: A software implemented fault injection tool for transient fault injection

by

Ameya Suhas Chaudhari, B.Tech; M.Tech

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2014

Fiesta++: A software implemented fault injection tool for transient fault injection

Ameya Suhas Chaudhari, M.S.E.
The University of Texas at Austin, 2014

Supervisor: Jacob Abraham

Computer systems, even when correctly designed, can suffer from temporary errors due to radiation particles striking the circuit or changes in the operating conditions such as the temperature or the voltage. Such transient errors can cause systems to malfunction or even crash. Fault injection is a technique used for simulating the effect of such errors on the system. Fault injection tools inject errors in either the software running on the processors or in the underlying computer hardware to simulate the effect of a fault and observe the system behavior. These tools can be used to determine the different responses of the system to such errors and estimate the probability of occurrence of errors in the computations performed by the system. They can also be used to test the fault tolerance capabilities of the system under test or any proposed technique for providing fault tolerance in circuits or software.

As a part of this thesis, I have developed a software implemented fault injection tool, Fiesta++, for evaluating the fault tolerance and fault response of software applications. Software implemented fault injection tools inject errors to simulate the effects of a fault into the software state of the application as it runs on a processor. Since such fault injection tools are used to conduct experiments on applications executing natively on a processor, the experiments can be carried out at almost the same speed as the application execution and can be run on the same hardware as used by the software application in the field.

Fiesta++ offers two modes of operation: whitebox and blackbox. The whitebox mode assumes that users have some degree of knowledge of the structure of the software under test and allows them to specify fault injection targets in terms of the application variables and fault injection time in terms of code locations and events at run time. It can be used for precise fault injection to get reproducible outcomes from the fault injection experiments. The blackbox mode is targeted for the case where the user has very little or no knowledge of the application code structure. In this mode, Fiesta++ provides the user with a view of the active process memory and an array of associated information which a user can use to inject faults.

Table of Contents

Abstract **iv**

List of Tables **viii**

List of Figures **ix**

Chapter 1. Introduction **1**

Chapter 2. Fault Injection Techniques **4**

2.1 Hardware based fault injection tools 4

2.1.1 Simulation based hardware fault injection 6

2.1.2 Emulation based hardware fault injection 7

2.2 Software implemented fault injection (SWIFI) 8

2.3 Compiler based Fault Injection and Profiling 11

2.4 Importance of Software based Fault Injection 12

Chapter 3. Fiesta++ **14**

3.1 Motivation 14

3.2 Fault injection framework 15

3.2.1 Modes of operation 16

3.2.2 Fault Models 17

3.3 Black Box Mode Fault Injection 18

3.3.1 Fault Injection Timing and Location 18

3.3.2 Obtaining Runtime Allocated Dynamic Memory 19

3.3.3 Runtime Stack Memory 21

3.3.4 Extracting Global Variable Addresses 22

3.4 White Box Mode Fault Injection 22

3.4.1 Fault Injection Target and Location 23

Chapter 4. Fault Injection Experiments using Fiesta++ **26**
4.1 Fault vulnerability over application runtime 27
4.1.1 Dynamic memory fault vulnerability 27
4.1.2 Stack memory fault vulnerability 29
4.1.3 Global variable fault vulnerability 29
4.2 Fault vulnerability of different stack regions 33
4.2.1 Fault vulnerability in the top most stack frame 33
4.2.2 Fault injection in function call signature 34
4.2.3 Fault injection in local variables and function arguments 35
4.3 Fault Injection Overhead 36
4.4 Conclusion 37

Chapter 5. Future Work **39**

Bibliography **41**

List of Tables

2.1 Comparison of different fault injection techniques 5

4.1 Fault injection outcome distribution for fault injected in the application dynamic memory 28

4.2 Fault injection outcome distribution for errors in the top stack frame. The numbers in parentheses are the error outcome probabilities when fault is injected at a random address chosen from the full application stack. 34

4.3 Fault injection outcome distribution for errors in function signature 35

4.4 Fault injection outcome distribution for errors in the local variables and function arguments. The numbers in parentheses are the error outcome probabilities when fault is injected at a random address chosen from the full application stack. 36

4.5 Run time overhead in seconds posed by the Fiesta++ fault injection mechanism 37

List of Figures

2.1 Hardware based fault injection by instrumenting flip-flops . . . 8

2.2 General framework used by software implemented fault injection tools . . . 10

3.1 Fiesta++ framework.The blocks shaded green are a part of the Fiesta++ framework. . . . 15

3.2 The runtime stack frame structure for applications. . . . 21

3.3 An example of using scope and variable access specification to describe fault timing in whitebox mode . . . 23

3.4 Flowchart for deciding the fault injection time in whitebox mode 24

4.1 Dynamic memory fault vulnerability for SPEC2000 benchmark applications . . . 30

4.2 Local variable fault vulnerability for SPEC2000 benchmark applications . . . 31

4.3 Global variable fault vulnerability for SPEC2000 benchmark applications . . . 32

Chapter 1

Introduction

Transient faults, in reference to computer systems, are abnormalities in circuit operation that occur for a short time period, generally for a single or a few clock cycles. Transient faults can be caused by radiation particles hitting circuit elements or variations in the voltage, current, or the temperature of the circuit. Even a properly verified and correctly functional circuit can encounter an error during operation due to a transient fault. With the scaling in the transistor sizes over years, the noise margins for memory elements have decreased and variations in the circuit environment parameters such as voltage and temperature have increased [20]. All these factors have made the circuits more vulnerable to transient faults and today most circuits have to be designed to be tolerant to such faults [12].

Dependability is a critical feature for most computer systems which needs to be properly evaluated and quantified. For computer systems, transient faults pose a significant challenge to the system dependability and their effect on dependability needs to be evaluated. Observing transient faults in an actual system is not a feasible mechanism for such an evaluation as the occurrence of transient faults is rather unpredictable and often rare when compared

with the usual system runtime. Fault injection tools provide a way to inject faults in an otherwise functional system so that sufficient experimental data can be obtained to evaluate the dependability of the system in their presence. Fault injection tools are used to simulate the effect of both the transient faults and the permanent faults. This thesis mainly focuses on injection of errors to simulate transient faults in applications and limits the discussion to those.

Fault injection tools inject a fault by changing signal/variable values at run time and then observing the effect on the output or behavior of the system. Most fault injection systems are designed to simulate the fault behavior of microprocessors as they are the most widely used integrated circuits and the systems employing them are too complex to be analyzed without empirical methods.

Fault injection tools are used to find answers to two important questions. First, what is the effect of a fault on the output and behavior of the processor or the software executing on it? Second, what is the probability of these observed effects, given that a fault has occurred?

The effect of a transient fault has been very well understood at the circuit level. It has been shown [6] that most of the time a transient fault results in the bit value stored in a circuit memory element being flipped. Most fault injection tools use this fault model and build on it to simulate the effects of a transient fault.

Today's processors contain millions of memory elements susceptible to

transient faults. Applications that run on these processors execute over trillions of clock cycles and a transient fault can occur during any execution cycle. Thus, the total number of possible transient faults in different application contexts is enormous. It is impossible to evaluate the outcomes of all possible faults or enumerate all faults to compute the exact probabilities of occurrence of any particular outcome. Fault injection tools generally operate by injecting random faults at random time points. If a sufficiently large number of such fault injection experiments are conducted, it can be assumed that the statistics for the fault outcomes obtained are close to the actual outcome probabilities.

This thesis describes a software implemented fault injection tool, Fiesta++, that can be used to evaluate the dependability of a software application in the presence of transient faults. It provides two modes of operation: blackbox and whitebox modes, which allow the user to evaluate the application under test with or without knowledge of the internals of the application software.

The rest of this thesis is structured in the following way. Chapter 2 provides a brief description of the prior work on fault injection tools, the different approaches taken, the advantages and disadvantages of each approach. Chapter 3 describes the framework, internals and working of Fiesta++. In Chapter 4, I discuss the results of a few fault injection experiments conducted and the conclusions drawn from those using Fiesta++. Chapter 5 concludes the thesis and provides a future path for extending this work.

Chapter 2

Fault Injection Techniques

Processor fault injection tools can be classified based on the level of abstraction at which that they operate. The following sections discuss the classes of fault injection tools available and give brief summaries of the popular techniques. Table 2.1 provides a comparison chart for the different classes of fault injection tools using a few relevant parameters.

2.1 Hardware based fault injection tools

Hardware based fault injection methods estimate the effect of a hardware level fault by simulating or emulating the circuit behavior after a fault. These tools either modify the circuit to introduce a mechanism to modify specific signal values in the circuit or modify the hardware simulators to do the same during circuit simulation. Since the single bit flip fault model used by the hardware based fault injectors accurately represents the actual phenomenon of transient faults in circuits, the statistics produced by hardware based fault injection tools generally give an accurate picture of the behavior of system in case of a hardware transient fault. All the hardware based fault injection techniques can be further categorized into two sub-classes, simulation based

Fault Injection Technique	Fault Model Accuracy	Experiment Speed	Experiment Cost and Portability	Experiment Repro-ducibility	Possible Use Cases
Hardware (emulation based)	High	Medium	High	High	Testing hardware fault tolerance for small embedded processors and applications with short run time. The cost of emulating hardware limits the size of circuit under test.
Hardware (simulation based)	High	Extremely Slow	Low	High	Can be used where emulation based hardware injection is not possible, yet accuracy is important. Slow simulation speeds implies only low latency errors can be observed.
Software based	Low	Fast	Low	High	Testing fault tolerance of large and complex applications on complex processors where accuracy of fault outcome probabilities is not important.
Radiation based	High	Fast	High	Low	Validating fault model for radiation based transient faults, evaluating dependability for multi-component systems
Compiler based	High	Fast	High	High	Predicting effects of the fault, collapsing the number of fault injection experiments to be carried out

Table 2.1: Comparison of different fault injection techniques

and emulation based techniques.

2.1.1 Simulation based hardware fault injection

Simulation based hardware fault injection techniques simulate the hardware description of the circuit under test using HDL simulators and during the simulation inject faults. Most of such techniques [3, 13, 22, 24] modify the hardware description of the circuit under test to include components necessary for injecting faults. These fault injection components can be modeled to inject different fault behavior depending on the fault model. Faults can also be injected using hardware description language (HDL) simulator commands which allow the variables and signals of the circuit under simulation to be modified. Such methods [13] do not require modification of the hardware description of the circuit under test for fault injection and hence are easier to implement.

A major disadvantage of simulation based techniques is that they are extremely slow. Simulating the register transfer level (RTL) description of a circuit is multiple orders of magnitude slower than the actual circuit operation speed. Hence, even for relatively small processors, simulation based fault injection tools can only evaluate the fault propagation for a few seconds of processor operation. It can only be used to observe the low latency faults that manifest as errors within a few seconds after injection.

However, these techniques provide an easy and accurate way to test the behavior of different fault models in different parts of the circuit. These techniques can be used to observe the effect of low latency errors which man-

ifest in either memory or register errors in short time periods. Once the error manifests itself in registers or memory contents, software implemented fault injection tools can be used to speed up the simulation of the error effect.

2.1.2 Emulation based hardware fault injection

Since simulation based fault injection experiments tend to be very slow, emulation based techniques were proposed. These techniques use reconfigurable circuits such as FPGAs to create modified versions of the test circuit that is capable of injecting faults at run time. The modified circuit contains flip-flops which can flip their output bit based on a control signal value. Such techniques require an additional control mechanism to specify the time and location of fault injection in the circuit. If such a control mechanism is implemented in circuit, its complexity increases with the number of fault injectable memory elements. And thus only a small set of flip-flops or memory elements can be instrumented for fault injection and the circuit needs to be reconfigured and reprogrammed on the FPGA if the fault needs to be injected in a different part of the circuit.

Antoni et al. [2] proposed a technique which reconfigured the circuit at run time on a FPGA. This eliminates the need for having a complex control circuit to determine the fault injection location. However, the time required to reconfigure the circuit could be significant when compared to the total application run time.

Civera et al. [7] proposed another solution for providing a more flexible

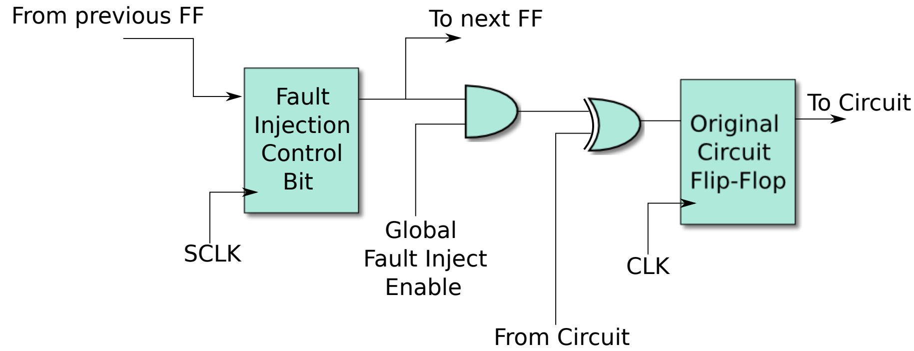

Figure 2.1: Hardware based fault injection by instrumenting flip-flops

control over runtime fault injection. They used modified flip-flops capable of injecting faults based on a control bit associated with each flip-flop. All these control bits are tied together like a scan-chain and at run time can be programmed to inject fault in any desired flip-flop in the circuit. Figure 2.1 illustrates the fault injection mechanism used such techniques.

A few recent techniques [8, 21] use on-chip debugging resources such as scan chains and JTAG debugging support for changing the state of the flip-flops in the circuit at run time. This allows the actual circuit to be tested for fault tolerance and the experiments can be carried out at speed and hence large applications can be tested.

2.2 Software implemented fault injection (SWIFI)

Software implemented fault injection tools use a software level abstraction of the single bit flip model to inject errors in software while it runs. These tools trade-off fault model accuracy for speed of fault injection exper-

iments and ability to test large applications. Also, these tools do not need any hardware changes or additional hardware to run experiments. Software implemented fault injectors provide a way to test complete systems with real hardware and software, including the operating system, for fault tolerance and effects of fault. This makes software implemented fault injection techniques quite popular and a large number software implemented fault injection tools exist [1, 5, 9, 16–18, 23].

One of the earliest software implemented fault injectors, FIAT [23] injected single bit flips in the instruction memory of the applications to model different type of faults. FERRARI [16] uses a parallel deamon process running on the host processor to control the application process in which the faults need to be injected. It is capable of injecting faults in the address, data or the control lines of the processor. DOCTOR [9] is a software implemented fault injection tool for injecting errors which can be used for injecting faults in distributed computing systems. It is able to inject CPU faults, memory faults and communication faults in such systems. DEFINE [17] provides a similar framework for injecting faults and analyzing results in distributed real-time systems. PROPANE [11] is a runtime fault injection tool that can inject software errors by mutating the source code at run time or changing the variable and memory contents.

Figure 2.2 illustrates the general framework on which these software implemented fault injectors are built. They generally consist of an experiment manager that decides which fault to inject and when. The fault injector run-

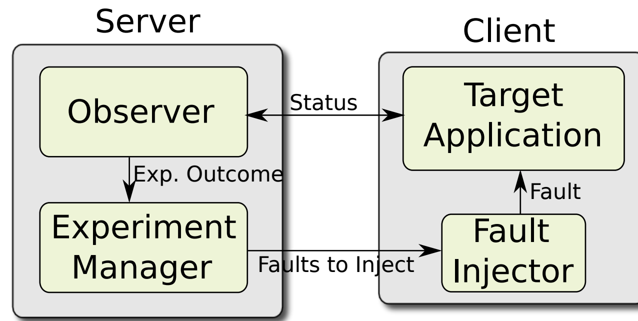

Figure 2.2: General framework used by software implemented fault injection tools

ning on the client processor inject faults into the application under test. An application observer monitors the application under test's behavior and reports the outcome of the experiment to the experiment manager. The experiment manager and the observer are generally resident on a separate machine to ensure that a system crash caused by any fault injection doesn't affect the experiment.

Jin et al. [14] have developed a fault injector using the dynamic binary instrumentation tool PIN. PIN allows insertion of code dynamically at run time allowing a fault to be injected at specific points in the application code. This tool targets software faults that result from common coding errors. They profile the application binary for possible fault sites and instrument these fault sites with modified instruction sequences.

2.3 Compiler based Fault Injection and Profiling

LLFI [25] is a compile time fault injection tool based on the LLVM compiler infrastructure. LLFI inserts fault injecting code into the compiler intermediate representation code of the application software. One of the advantages of using compiler based fault injection is that some level of static analysis can be done on the fault behavior to infer the outcome of a fault without simulating it.

Relyzer [10] is another technique which uses compiler based techniques to determine the equivalence of different fault sites in an application. As mentioned earlier, the total number of possible fault locations, even for a small processor running a moderately sized application, can be enormous. Random fault injection, as done by the hardware and software implemented fault injection techniques, may not be able to distinguish between equivalent fault sites and would lead to redundant fault injection experiments.

Benso et al. [4] have done similar work in estimating criticality of data variables in a program and provide a mathematical framework for estimating the data criticality of application data variables. Such techniques can help in aiding experimental fault injection improve coverage by determining the equivalence of different faults or by predicting the outcome of a fault.

2.4 Importance of Software based Fault Injection

Software based fault injection limits the fault injection at a higher, software visible abstraction level such as memory contents and registers. The actual transient faults occur at the device level and a single bit flip at the device level may not be accurately represented by a single bit flip at the software abstraction level [6]. The hardware based fault injection techniques, on the other hand, can accurately replicate the effect of a transient fault. This brings to question the relevance of software implemented fault injection techniques.

Many applications run on large complex super-scalar processors for which designing hardware based fault injection may be impractical. Most of the hardware based fault injection techniques have been demonstrated on small processors with a few hundreds or thousand flip-flops in their design. Developing hardware based fault injection would be difficult for the large and complex processors used in most practical applications.

Also, the hardware based fault injections have been carried out only on small snippets of code. Our experiments presented in the later chapters show that the fault outcome probabilities vary significantly in different parts of the software. Thus, projecting the fault outcome probabilities of a small set of instructions to the entire application run may not be accurate.

Software based fault injection tools are indispensable for estimating the fault outcome and probabilities for large supercomputing applications. These applications which run on hundreds of processors for extended periods of time

face significant risk of fault occurrence during the application run. Hardware based fault injection cannot be used to evaluate such large applications.

Chapter 3

Fiesta++

3.1 Motivation

Most software based fault injection tools provide only an interface to inject faults at random memory locations. This limits the fault models that can be used to specify fault location and timing.

Fiesta++ was developed with the aim of providing the user with a meaningful interface with useful information about the software state for developing detailed and meaningful fault models. It provides a mechanism to specify custom fault models as programmable C code. Along with hardware faults, it can also model software faults and thus can be used for testing for software bugs. Fiesta++ provides two modes of operation, whitebox and blackbox mode, that allows fault injection campaigns to be carried out on software irrespective of whether the user has knowledge about the software code structure of the application under test. It can be used to carry out fault injection experiments even in parallel and distributed applications running on large computing clusters. It can be directed to inject faults in a specific thread or multiple threads of an application, simultaneously.

3.2 Fault injection framework

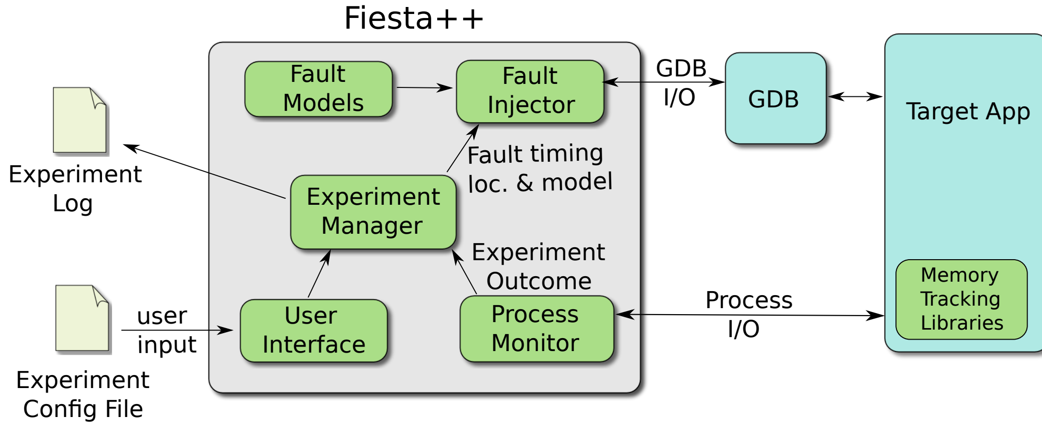

Figure 3.1: Fiesta++ framework.The blocks shaded green are a part of the Fiesta++ framework.

Figure 3.1 describes the Fiesta++ framework. The framework for Fiesta++ is similar to most software based fault injectors. The experiment manager, is the main controller that oversees the fault injection experiment. The experiment parameters are supplied by the user using a configuration file.

The experiment manager supplies the fault location, timing and model information to the fault injector whenever a fault is to be injected in the application under test. The Fiesta++ fault injector utilizes the GNU debugger tool (GDB) that is available on most Linux systems, to inject faults in the application process memory. GDB apart from providing fault injection support, also provides an interface to query some of the runtime memory information

such as the active stack frames and information about the application binary.

The process monitor observes the runtime behavior of target application on which the fault injection needs to be carried out. It analyzes the output of the application to determine whether the injected fault affects the application output and behavior, and how it does so. It communicates the output of the experiment to the experiment manager. The target application needs to be linked with special memory tracking libraries which Fiesta++ uses to query the runtime memory state.

Fiesta++ uses the GNU debugger (GDB) which is used attach to any process running on either the host or remote machines and is used to control the process and inject faults. GDB is an open source debugging application which uses the *ptrace* system call in Unix systems to attach to a running process. GDB allows the user to set breakpoints in the target application code, set watchpoints to monitor the value of variables, examine/modify the process memory at any point during the application run time and control the process execution. Fiesta++ leverages these capabilities provided by GDB to provide a highly configurable and programmable fault injection tool.

3.2.1 Modes of operation

Fiesta++ has two basic modes of operation, blackbox mode and white-box mode. The blackbox mode assumes that a user has no knowledge of the underlying code. In this mode the user specifies the fault model for the faults and the time instants at which these faults need to be injected. The tool at

run time stops the target process at the user specified time instants, extracts the valid data memory (on the heap,the stack and globals) and injects the faults as specified by the fault model. This mode is useful for testing software for which the source code is unavailable. For many applications that use third party software, such a fault injection tool would be useful.

In the whitebox mode, it is assumed that the user has some knowledge and access to the source code. This mode can be used by software developers to test specific parts of their software for soft-error vulnerability. In the whitebox mode, Fiesta++ allows the user to specify which variables to inject the faults in and when during the application execution to inject faults. The timing of fault injection is specified in terms of the number of times a particular region of code has been executed and the number of times the target variable has been accessed. Fiesta++ uses the hardware breakpoint capability provided by the microprocessors to monitor the target data variables without slowing down the target process significantly. These operating modes are explained in detail in later sections of this chapter.

3.2.2 Fault Models

Fiesta++ currently provides four basic fault models that emulate both hardware faults and software bugs.

1. Single bit flips
2. Increment variable value

3. Decrement variable value
4. Reset to zero.

Custom fault models can also be developed for more specific fault injections. For example, Fiesta++ provides functions to query the runtime memory state. Using these functions, a fault model could be developed that injects fault just in the local variables inside the currently executing function. Since the fault injection is controlled by the Fiesta++ wrapper code around GDB, it can be easily configured to add any fault model that can be specified using a C program.

3.3 Black Box Mode Fault Injection

The blackbox mode of operation in Fiesta++ provides a mechanism for a user to carry out fault injection campaigns on an application without knowing its internal structure. The user can treat the software under test as a black box. For injecting a fault in an application at runtime, the temporal (timing of the fault injection) and spatial (the fault target) parameters for the fault need to be specified. The following sections describe the mechanisms for specifying these parameters.

3.3.1 Fault Injection Timing and Location

The time for fault injection in the blackbox mode is specified in seconds of run time from the start of the program. Such a mechanism does not provide

an accurate, replicable fault injection, since the execution time of the application will vary, depending on the runtime environment. However, as the user has little knowledge of the application under test, there is very little scope for improving on the fault injection timing accuracy. This mode allows the user to test the fault vulnerability of an application at different points during the run time.

The fault location can be specified as one of three memory regions: heap, stack and global data. The exact fault location is determined by the fault model. The default fault model choses a random address from the allocated memory for the application. The total memory in use in the specified memory region is extracted at the fault injection time and a random memory address in this computed list is chosen as the target of fault injection. Although the current version cannot specify registers as a fault target, it can be trivially inserted.

Many fault injectors randomly inject errors throughout the application's virtual memory space. However, many regions of the virtual space may not be valid or in use and fault injection in these areas would be meaningless. Fiesta++, therefore, extracts the memory regions in use at the time of fault injection and then injects faults only in the valid memory.

3.3.2 Obtaining Runtime Allocated Dynamic Memory

The blackbox mode requires automatic extraction of the memory regions in use at the fault injection time for determining the fault location. The

tool needs a list of all the dynamic memory blocks in use. Fiesta++ uses the dynamic memory management hooks provided by the GNU C Library (GLIBC) to get the runtime state of the heap.

The dynamic memory management functions (*malloc*,*realloc* and *free*) are called using function pointer variables (*__malloc_hook*,*__realloc_hook* and *__free_hook*). These function pointers can be reassigned to a user specific functions so that the user functions are called instead of the GLIBC memory management functions. In Fiesta++ we use these hooks to insert our own dynamic memory allocation tracking functions before the calls to *malloc*,*realloc* and *free* to log each memory allocation and deallocation operation.

These logs are stored in a standard vector (a dynamic data structure that provides random access with amortized constant time appends) data structure and the log for each memory allocation or deallocation operation is appended into a vector like data structure. Since a *push_back* operation on a vector has an amortized constant time complexity, the insertion of these hooks does not pose significant time overhead on the memory allocation functions. To estimate the time overhead of inserting these memory tracking libraries, the run time of eight SPEC benchmark applications (listed in Chapter 4) compiled with and without the libraries were observed. The difference in run time for all the benchmarks was insignificant (in the order of milliseconds).

These memory allocation tracking functions are provided as a static library which needs to be included during the linking of the test application object files. Fiesta++ uses GDB to query the state of the memory allocation

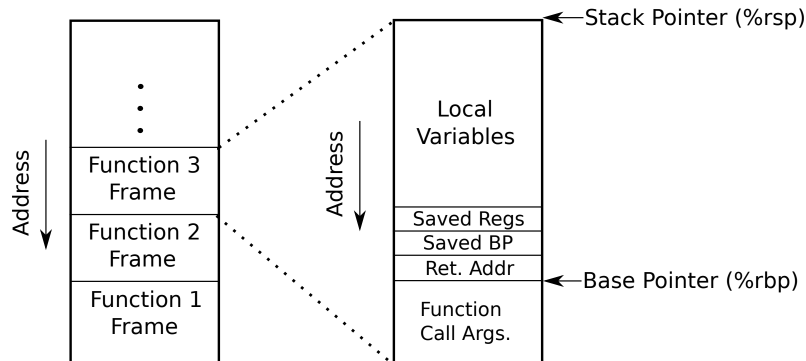

Figure 3.2: The runtime stack frame structure for applications.

log data structure provided by the memory allocation tracking functions. This log is processed to get the pointers to the currently allocated blocks on the heap and their sizes. The log maintains the order of allocation which could be used to determine the most/least recently allocated blocks. For the basic random bit flip fault model, an address is randomly chosen from all the allocated memory addresses as the target for fault injection.

3.3.3 Runtime Stack Memory

The stack memory in use can be obtained by querying the value of the stack pointer in GDB. The stack pointer(%esp) is a pointer to the top of the stack. GDB commands *backtrace* and *info frame* can be used to get the bottom of the stack. All the memory addresses between the top and bottom of the stack are considered to be in use.

The interface with GDB allows Fiesta++ to get information about the different stack frames present at the time of injection. Fig. 3.2 describes

the stack frame structure for applications compiled using GCC on x86 ISA processors. Fiesta++ extracts all the information shown in the figure for each stack frame. Users can use this information to develop specialized fault models and fault injection campaigns targeting specific regions of the stack such as function arguments, local variables or the stored registers.

3.3.4 Extracting Global Variable Addresses

The global variable addresses can be fetched from the application binary address itself. The global variables are stored in *.data* and *.bss* segments in the application binary. The starting virtual address for these sections and sizes can be obtained using the GNU *objdump* utility. Fiesta++ gets the starting address and size of these sections before starting the application simulation.

3.4 White Box Mode Fault Injection

The white box mode of fault injection is useful when the user has some knowledge of the application software structure and access to the source code. The white box mode allows the user to specify the fault targets in terms of application code variables and the injection time in terms of code execution events. This allows the user to conduct deterministic and replicable fault injection campaigns for applications.

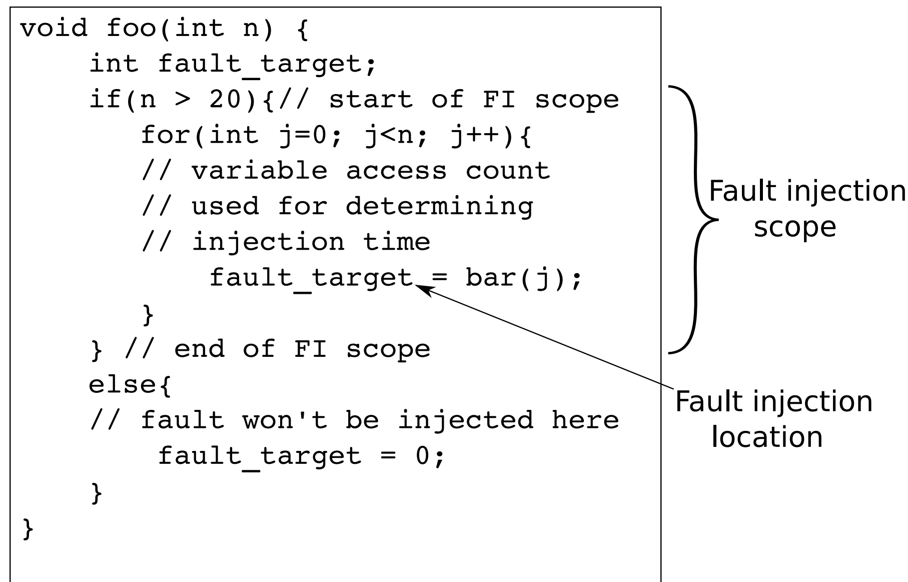

Figure 3.3: An example of using scope and variable access specification to describe fault timing in whitebox mode

3.4.1 Fault Injection Target and Location

The fault targets in the whitebox mode are specified using the application code variables. The fault injection timing is specified using a combination of two parameters: code scope trigger and a variable access trigger.

For each fault injection target the user needs to specify the code scope during which the fault injection is activated. The code scope is specified in terms of either a function or line numbers in the source code file. The fault target should be a valid variable in this scope. Fiesta++ monitors the number of times this scope is hit during code execution. Once the hit count is greater than the code scope trigger, Fiesta++ sets up a variable access monitor for

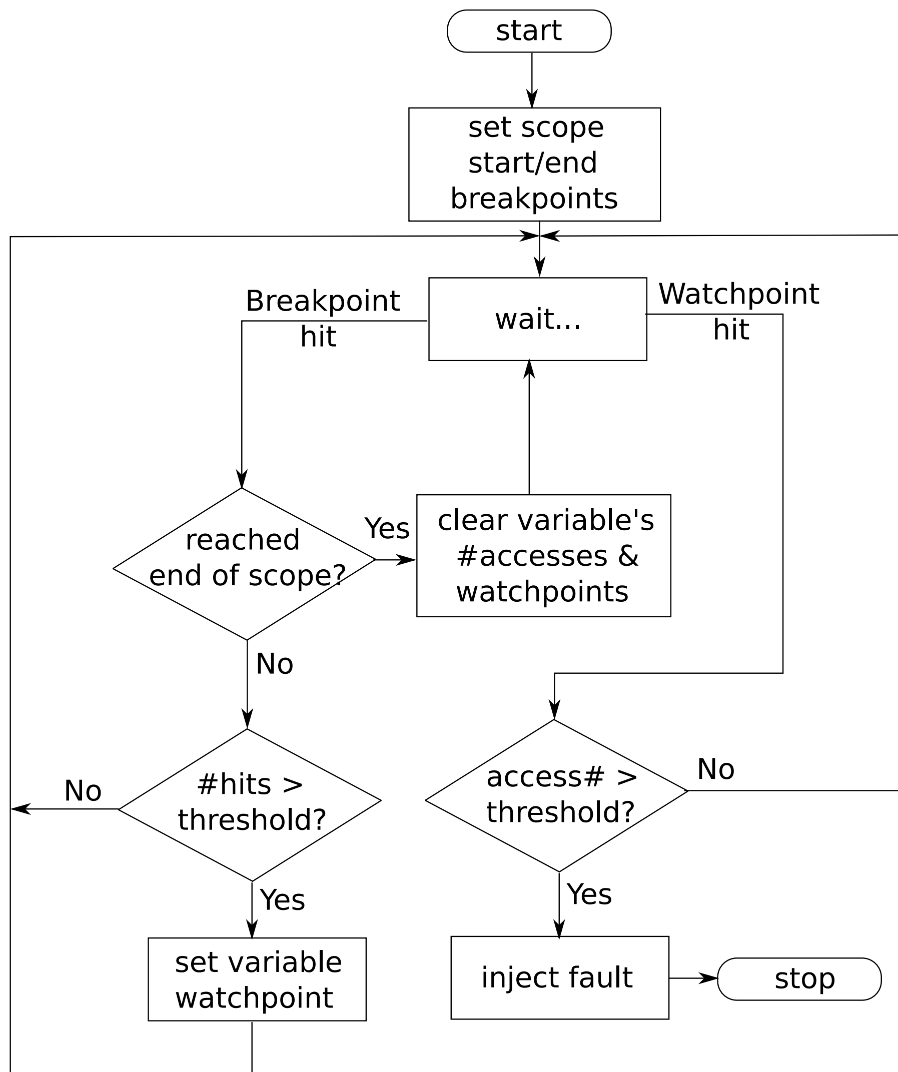

Figure 3.4: Flowchart for deciding the fault injection time in whitebox mode

the target variable.

The variable access monitor is set up using a hardware breakpoint which stops code execution whenever the target variable address is accessed. The fault is injected when the variable access count is greater than the variable access trigger specified by the user. The access count is reset to zero if the program execution goes out of the specified scope. The flowchart in Figure. 3.4 describes the decision process taken by Fiesta++ to determine the timing of fault injection in the whitebox mode.

The number of hardware breakpoints that can be simultaneously active is generally limited by the processor architecture. Our test processor allowed a maximum of 4 hardware breakpoints to be active at any given time. Fiesta++ has a queue to store the set of active fault targets that need to be monitored. Based on the number of hardware breakpoints allowed, access monitors are set up for the targets in the queue.

Such a dual monitoring mechanism enables specifying accurate injection timings for all kinds of variables. Often variables are declared in the local scope (such as functions, loops, blocks) and may not be visible at a global scope. This code scope where the variable is valid can be specified using the code scope trigger mechanism. Inside the valid scope the variable may be access numerous times, for example, inside a loop and the accurate injection timing can be specified using the variable access count trigger.

Chapter 4

Fault Injection Experiments using Fiesta++

This chapter discusses the results of fault injection experiments that were conducted using Fiesta++. The effect of a fault in the software state depends on various properties of the application's code. The aim of these fault injection experiments is to find and learn about any relationship that might exist between the different outcomes, the fault locations and timing. Fiesta++ allows us to inject faults at different points of time during the application run and target faults in specific memory regions. The fault injection experiments described in this chapter were carried out using the Fiesta++ black box mode for eight SPEC2000 benchmark applications. The *ref* inputs of the benchmark suite were used for evaluation. The application runtime for these inputs range from 30 seconds to 3 minutes on a six core, 2.8GHz AMD PhenomII processor with 8GB RAM. Each application run spans hundreds of billions of application instruction executions.

In each experiment, a single fault was injected using the single bit flip model. The behavior and outcome of the application under test was observed and the result was categorized in one of the following four categories.

- Application Hang: The fault injection resulted in the application running

for more than twice its usual runtime.

- Crash: The application terminated prematurely with a possible exception as a result of fault injection.
- Silent Data Corruption: The application terminated normally however, its final output did not match the output from an error free run.
- No error: The fault injected did not have any affect on the application behavior or output.

4.1 Fault vulnerability over application runtime

Applications over different phases during run time perform different tasks. The effect of faults during these different phases may differ depending on the code structure for these tasks.

In order to determine how the fault vulnerability of an application changes over the run time, I conducted fault injection campaigns where faults were injected in applications at different time points during its run time. A single bit error was injected in the application memory during each run. The following sections discuss the results obtained when these faults were injected in the different memory regions.

4.1.1 Dynamic memory fault vulnerability

Dynamic memory, or the heap, stores the variables that are allocated at runtime using the memory management functions such as *malloc*, *realloc* or

new. For most applications, the dynamic memory is the largest of the three different data memory regions.

The SPEC2000 benchmark applications used for these experiments had dynamic memory sizes ranging from 3 MB to 190 MB. We injected faults at random time instants distributed uniformly over the application's runtime. For each application, over 2000 fault injection experiments were carried out. The graphs in Figure 4.1 plots the variation in percentage of different outcomes for fault injection carried out at different points during the application's run. Table. 4.1 gives the average outcome percentages for the eight benchmark applications.

Benchmarks	Fault Outcome			
	No Errors	SDC	Crash	Timeout
ammp	94.47 %	0.55 %	4.97 %	0.00 %
art	89.50 %	6.50 %	3.50 %	0.50 %
equake	71.50 %	11.50 %	17.00 %	0.00 %
gzip	69.50 %	20.00 %	0.00 %	10.50 %
mcf	91.50 %	3.50 %	4.50 %	0.50 %
mesa	95.50 %	1.50 %	3.00 %	0.00 %
twolf	27.69 %	7.17 %	24.61 %	40.51 %
vpr	53.50 %	23.50 %	18.00 %	5.00 %
Average	74.05 %	9.39 %	9.45 %	7.11 %

Table 4.1: Fault injection outcome distribution for fault injected in the application dynamic memory

Most injected faults in the dynamic memory do not cause any observable error in the application's output or behavior. However, the average outcome percentages for a fault when injected in the three memory regions, the

number of silent data corruptions and timeouts are significantly more for dynamic memory. Given the fact that the dynamic memory size is greater than 10× the rest of all the memory regions combined, this difference in outcome percentages seems to be even more significant. The number of faults injected compared to the total in-use memory is extremely small.

4.1.2 Stack memory fault vulnerability

The application runtime stack stores the local variables for the executing functions, the function call signature and the function arguments. Figure 4.2 shows the fault vulnerability of the stack memory over the application runtime. Analysis of how the fault vulnerability varies for each of the different classes of data on the stack will be presented in a later section of this chapter.

4.1.3 Global variable fault vulnerability

The global variables for an application are stored in two sections in the application binary, .data and .bss. Figure 4.3 shows the global variable fault vulnerability for eight SPEC benchmark applications. For some of the applications such as *ammp* and *equake*, a fault in a global variable has almost no effect. However, faults in the global variables have a significant impact on the output of the *Art* benchmark.

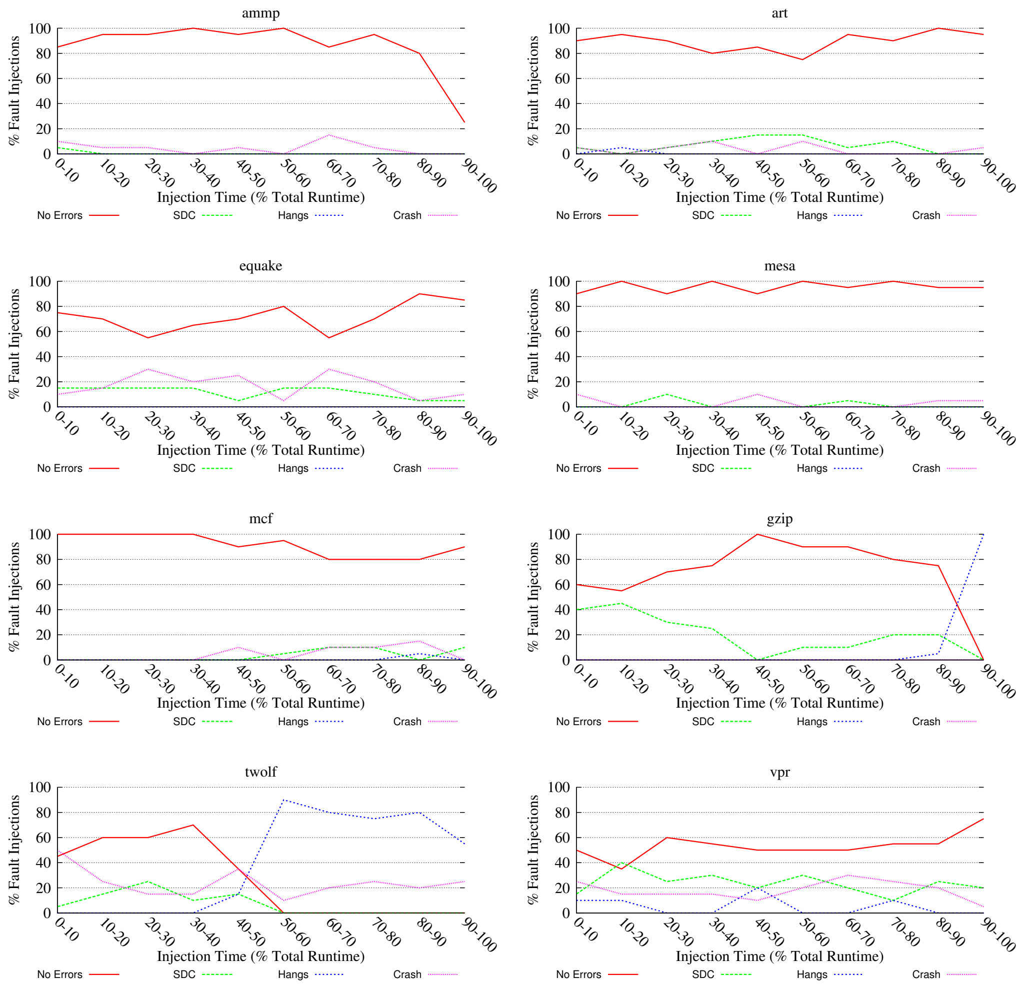

Figure 4.1: Dynamic memory fault vulnerability for SPEC2000 benchmark applications

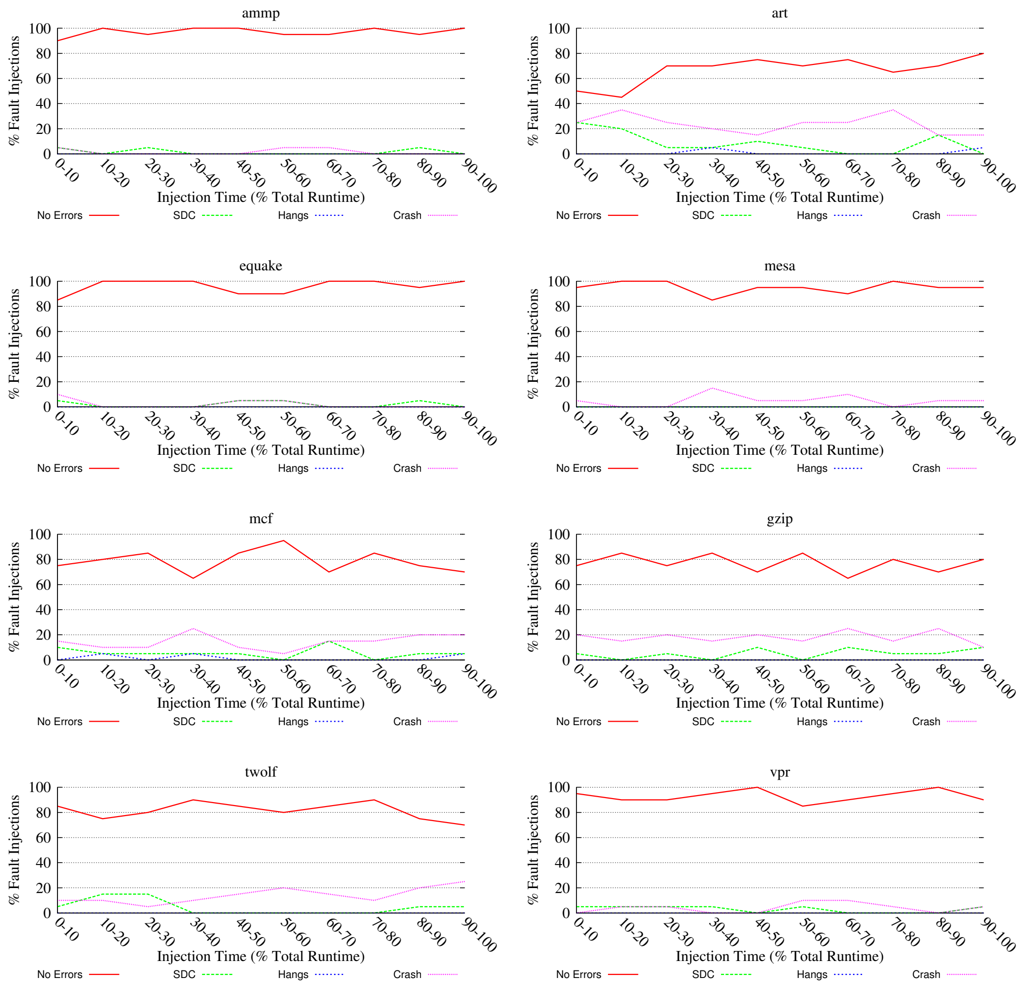

Figure 4.2: Local variable fault vulnerability for SPEC2000 benchmark applications

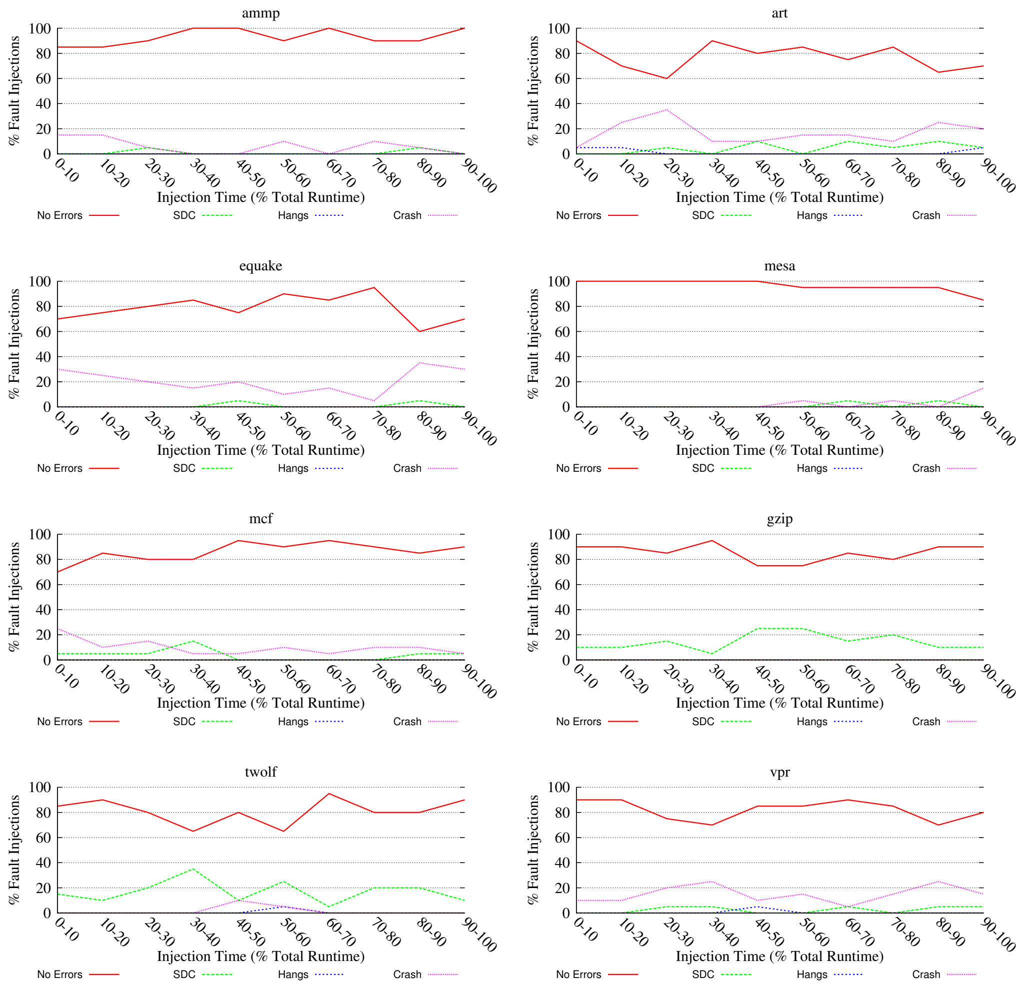

Figure 4.3: Global variable fault vulnerability for SPEC2000 benchmark applications

4.2 Fault vulnerability of different stack regions

The stack structure stores the different types of data variables such as the function signature, local variables and the function arguments in separate sections in the stack frame. Fiesta++ allows us to access each section independently and target fault injection to one or more of these areas and evaluate their fault vulnerability. The following sections discuss the experimental results for fault injections targeted for regions in the application stack frame.

4.2.1 Fault vulnerability in the top most stack frame

Fiesta++ provides information on all the active stack frames in the application at any given point of time. A fault model can be developed that injects faults only in the top frame on the stack. The top frame on the stack is the frame for the currently executing function.

If it is assumed that the faults mainly occur in the processor hardware (assuming memories are protected with ECC), there is a larger probability that the errors due to faults would manifest themselves in the variables recently accessed at the time of fault occurrence. As far as local variables are concerned, the top frame would contain the local variables for the currently executing function and would have been recently accessed. A fault injection campaign was carried out to see if the fault behavior for injections in the full application stack differ significantly from those in the top stack frame.

Benchmarks	Fault Outcome			
	No Errors	SDC	Crash	Timeout
ammp	70.63% (97%)	9.37%(1.5%)	18.75% (1.5%)	1.25% (0.0%)
art	75.00% (67%)	2.50% (8.5%)	22.50% (23.5%)	0.00% (1%)
equake	67.50% (96%)	2.50% (2%)	30.00% (2%)	0.00% (0%)
gzip	61.11% (77%)	1.11% (5%)	37.78% (18%)	0.00% (0%)
mcf	32.50% (78.5%)	8.50% (5.5%)	57.00% (14.5%)	2.00% (1.5%)
mesa	63.89% (95%)	0.00% (0%)	36.11% (5%)	0.00% (0%)
twolf	46.50% (81.5%)	7.50% (4.5%)	46.00% (14%)	0.00% (0%)
vpr	60.00% (93%)	3.50% (3%)	35.50% (4%)	1.00% (0%)
Average	59.28 % (85.63%)	4.34 % (3.75%)	35.85 % (10.31%)	0.53 % (0.31%)

Table 4.2: Fault injection outcome distribution for errors in the top stack frame. The numbers in parentheses are the error outcome probabilities when fault is injected at a random address chosen from the full application stack.

4.2.2 Fault injection in function call signature

The function call signature consists of the return address for the function and the stored base pointer. Modifying the return address causes the program execution to jump to an erroneous location after the return instruction in the current executing function is executed. Modifying the base pointer leads to a wrong address computation for the local variables. Both these errors lead to corruption of a large number of variables in the application program which I expected would lead the application to crash most of the time.

One of the aims of this experiment was to see of the outcome probability for error injection in these variables is constant across different applications. If this is so, these can be excluded from fault injection experiments all together as their outcome is mostly known. Table 4.3 shows the fault outcome probability for the eight SPEC benchmark applications.

As the data shows, although the assumption that most fault injections

Benchmarks	Fault Outcome			
	No Errors	SDC	Crash	Timeout
ammp	48.33%	1.11%	47.78%	2.78%
art	4.50%	16%	76.50%	3.00%
equake	20.50%	10.50%	69.00%	0.00%
gzip	49.00%	1.50%	49.50%	0.00%
mcf	24.00%	13.00%	63.00%	0.00%
mesa	36.00%	0.00%	64.00%	0.00%
twolf	15.50%	11.00%	72.50%	1.00%
vpr	58.00%	8.5%	33.50%	0.00%
Average	31.77 %	7.78 %	59.62 %	0.82 %

Table 4.3: Fault injection outcome distribution for errors in function signature

would result in an application crash stands true, the probability of the application crash varies widely across different benchmark applications. For example, while art, equake and twolf result in crashes for more the two-thirds of the total injections, the crash probability for ammp and vpr is extremely low.

It must be noted that the probability of silent data corruptions for fault injections in the function signature is greater than the probability for the same for injections in the full application stack. This indicates that the function signature errors contribute significantly to the silent data corruption statistics.

4.2.3 Fault injection in local variables and function arguments

Local variables and function arguments of the application stack correspond directly to the application variables. Hence, fault injection in the local variables region of the stack frame is essentially the same as fault injection

at the source code level in the function local variables. Table 4.4 compares the fault injection outcome percentages for the local variable fault injection versus the outcomes for fault injection in the entire valid stack frame. This comparison illustrates how the outcomes would differ if fault injection were carried out at the source code level versus when carried out at the assembly instruction level.

Benchmarks	Fault Outcome			
	No Errors	SDC	Crash	Timeout
ammp	91.67% (97%)	0.00%(1.5%)	7.78% (1.5%)	0.56% (0.0%)
art	94.50% (67%)	3.00% (8.5%)	2.50% (23.5%)	0.00% (1%)
equake	22.00% (96%)	1.50% (2%)	76.50% (2%)	0.00% (0%)
gzip	81.00% (77%)	2.50% (5%)	16.50% (18%)	0.00% (0%)
mcf	85.00% (78.5%)	0.00% (5.5%)	25.00% (14.5%)	0.00% (1.5%)
mesa	75.00% (95%)	0.00% (0%)	25.00% (5%)	0.00% (0%)
twolf	88.00% (81.5%)	2.00% (4.5%)	10.00% (14%)	0.00% (0%)
vpr	97.78% (93%)	1.11% (3%)	1.11% (4%)	0.00% (0%)
Average	78.94 % (85.63%)	1.32 % (3.75%)	19.67 % (10.31%)	0.06 % (0.31%)

Table 4.4: Fault injection outcome distribution for errors in the local variables and function arguments. The numbers in parentheses are the error outcome probabilities when fault is injected at a random address chosen from the full application stack.

4.3 Fault Injection Overhead

Fiesta++ poses a run time overhead for each fault injection. This overhead is a result of the time spent communicating via GDB with the target application for memory status queries and fault injection commands. Table 4.5 shows the average observed run time overhead in seconds the fault injection mechanism posed for fault injection in each of the three memory regions.

Memory Region	Time Overhead (seconds)
Dynamic Memory	2.9
Stack Memory	1.2
Globals	0.11

Table 4.5: Run time overhead in seconds posed by the Fiesta++ fault injection mechanism

The time overhead for injecting faults into dynamic memory is higher due to the large number of dynamically allocated memory blocks in the SPEC benchmark applications. Fiesta++ spends significant amount of time reading the dynamic memory block list from the memory tracking libraries. For stack memory fault injection, Fiesta++ reads the stack frame information through GDB which results in a moderate run time overhead. For global memory variables, the size and location of the variables can be obtained from the application binary and does not need to be queried at runtime using GDB. This results in a minimal fault injection time overhead.

4.4 Conclusion

We can make some notable observations from the experiments described in the preceding sections. Although, there are some visible trends between the fault injection timing, location and the fault outcome, these cannot be generalized for all applications. The graphs for fault vulnerability over injection time show that for various applications, certain time periods during the application run, errors in memory have a greater probability of affecting the output or

behavior.

The fault outcome probability heavily depends on the fault model assumed for the software level faults. The significant difference in the outcome probability when faults were concentrated in the top stack frame versus when they were injected throughout the application stack illustrates the fact. Faults originating from the processor will be concentrated in the memory regions being accessed by it, while those originating in the memory will be evenly distributed across the memory.

The faults in data other than the application variables, such as the function signature, contributes significantly to the silent data corruptions and application crashes. These faults cannot be ignored and software based fault injection tools that operate only on software defined variables will miss these faults.

Chapter 5

Future Work

There are two directions in which this work can be carried forward. First is adding more functionality to Fiesta++ to enable it to support a wider variety of fault models. It currently lacks the ability to inject faults in the application code. Adding this ability will enable it to mimic faults in the processor fetch and decode stage which lead to the instruction being interpreted incorrectly. Some support for identifying the recently accessed memory addresses at any given point of time during application runtime would be useful for emulating the fault model where a fault in the processor leads to a corrupted instruction output.

A second direction in which this work could be extended is in developing accurate fault models for different applications. For hardware faults, currently there is lack of clear understanding about how a fault in the processor would translate into fault in the software state. There have been results illustrating that a memory bit flip model may not correctly represent the fault outcome at software level[6], however, they do not specify what could be a reasonable fault model for software level fault injection.

Also, the fault models in Fiesta++ could be extended to include a

wider variety of software faults. Currently it supports fault models to increment/decrement software variables which would test for off-by-one bugs in software. These fault models could be extended to cover a larger variety of software bugs.

Bibliography

[1] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: Generic object-oriented fault injection tool. In *International Conference on Dependable Systems and Networks, 2001. DSN 2001.*, pages 83–88. IEEE, 2001.

[2] L. Antoni, R. Leveugle, and M. Feher. Using run-time reconfiguration for fault injection in hardware prototypes. In *Proceedings of 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2002. DFT 2002.*, pages 245–253. IEEE, 2002.

[3] J. Baraza, J. Gracia, D. Gil, and P. Gil. A prototype of a vhdl-based fault injection tool: description and application. *Journal of Systems Architecture*, 47(10):847–867, 2002.

[4] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri. Data criticality estimation in software applications. 2003.

[5] J. Carreira, H. Madeira, and J. Silva. Xception: A. technique for the experimental evaluation of dependability in modern computers. *IEEE Transactions on Software Engineering*, 24(2):125–136, 1998.

[6] H. Cho, S. Mirkhani, C. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In

Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE, pages 1–10. IEEE, 2013.

[7] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Exploiting circuit emulation for fast hardness evaluation. *IEEE Transactions on Nuclear Science*, 48(6):2210–2216, 2001.

[8] André V. Fidalgo, Gustavo R. Alves, and José M. Ferreira. Real time fault injection using a modified debugging infrastructure. In *12th IEEE International On-Line Testing Symposium, 2006. IOLTS 2006.*, pages 6–pp. IEEE, 2006.

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

[10] S. Hari, S. V. Adve, H. Naeimi, and P. 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.

[11] M. Hiller, A. Jhumka, and N. Suri. Propane: an environment for examining the propagation of errors in software. *ACM SIGSOFT Software Engineering Notes*, 27(4):81–85, 2002.

[12] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba. Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule. *Electron Devices, IEEE Transactions on*, 57(7):1527–1538, 2010.

[13] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mefisto tool. In *Digest of Papers., Twenty-Fourth International Symposium on Fault-Tolerant Computing, 1994. FTCS-24.*, pages 66–75. IEEE, 1994.

[14] A. Jin, J. Jiang, J. Hu, and J. Lou. A pin-based dynamic software fault injection system. In *The 9th International Conference for Young Computer Scientists, 2008. ICYCS 2008.*, pages 2160–2167. IEEE, 2008.

[15] B. W. Johnson and J. A. Profeta. Fault injection technique for behavioral-level models. *IEEE Transactions on Design and Test of Computers*, 13(4):24–33, 1996.

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A. flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, 1995.

[17] W-I Kao and R. K. Iyer. Define: A. distributed fault injection and monitoring environment. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, 1994.*, pages 252–259. IEEE, 1994.

[18] W-I Kao, R. K. Iyer, and D. Tang. Fine: A. fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, 1993.

[19] N. Krishnamurthy, V. Jhaveri, and J.A. Abraham. A design methodology for software fault injection in embedded systems. In *IFIP International Workshop on Dependable Computing and Its Applications, 1998.*, pages 237–248. IEEE, 1998.

[20] K. J. Kuhn, M. D. Giles, D. Becher, P. Kolar, A. Kornfeld, R. Kotlyar, Sean T. Ma, A. Maheshwari, and S. Mudanai. Process technology variation. *Electron Devices, IEEE Transactions on*, 58(8):2197–2208, 2011.

[21] M. Portela-García, C. López-Ongil, M. García-Valderas, and L. Entrena. A rapid fault injection approach for measuring seu sensitivity in complex processors. In *13th IEEE International On-Line Testing Symposium, 2007. IOLTS 2007*, pages 101–106. IEEE, 2007.

[22] P. Prinetto, A. Benso, F. Corno, M. Rebaudengo, M. Sonza R, A. Amendola, L. Impagliazzo, and P. Marmo. Fault behavior observation of a microprocessor system through a vhdl simulation-based fault injection experiment. In *Proceedings of the conference on European design automation*, pages 536–541. IEEE Computer Society Press, 1996.

[23] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Ysskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. Fiat-fault injection based

automated testing environment. In *International Symposium on Fault Tolerant Computing, 1995, FTCS 1995*, page 394. IEEE, 1995.

[24] V. Sieh, O. Tschache, and F. Balbach. Verify: evaluation of reliability using vhdl-models with embedded fault descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 32–36. IEEE, 1997.

[25] A. Thomas and K. Pattabiraman. Llfi: An intermediate code level fault injector for soft computing applications. *IEEE Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.