Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

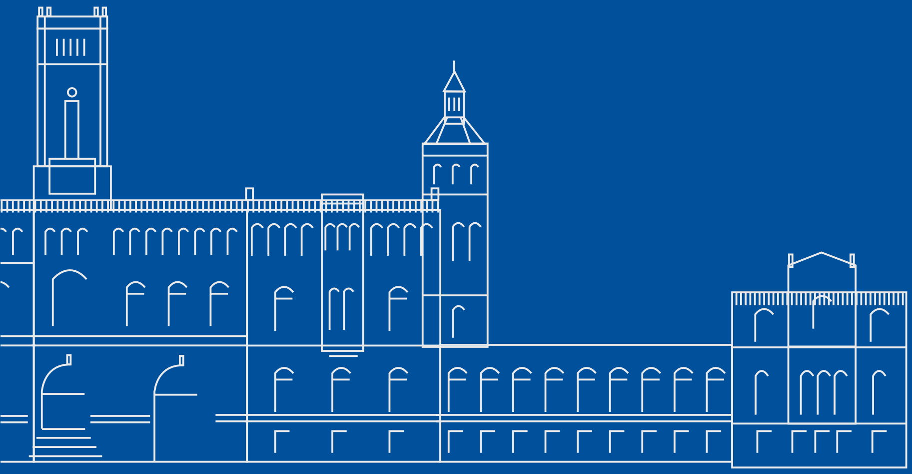
Marcel Budoj

Schotbruch: Automatisierte Ableitung von Injektionsplattformen für transiente Hardwarefehler aus formalen Prozessormodellen

Masterarbeit im Fach Informatik

08. Mai 2020

Please cite as:
Marcel Budoj, “Schotbruch: Automatisierte Ableitung von Injektionsplattformen für transiente Hardwarefehler aus formalen Prozessormodellen” Master’s Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, May 2020.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

Schotbruch: Automatisierte Ableitung von Injektionsplattformen für transiente Hardwarefehler aus formalen Prozessormodellen

Masterarbeit im Fach Informatik

vorgelegt von

Marcel Budoj

geb. am 13. Juni 1996
in Hannover

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer: **Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer: **Prof. Dr.-Ing. Bernardo Wagner**
Betreuer: **Dr.-Ing. Christian Dietrich**

Beginn der Arbeit: **24. Oktober 2019**
Abgabe der Arbeit: **08. Mai 2020**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Marcel Budoj)
Hannover, 08. Mai 2020

ABSTRACT

Transient errors caused by cosmic radiation in processors have been a problem for a long time. To better deal with these errors, it is necessary to find out how the systems behave when transient errors occur. Since the error injection into real hardware is often expensive and complex, the error injection into emulators has become an alternative. One such tool for error injection into emulators is the FAIL* framework. However, the problem with error injection into emulators is that the emulators must first be developed. However, this is time-consuming. The Sail tool solves the problem by enabling the generation of emulators from processor architecture models. The problem now is how to automatically integrate the generated emulators into the error injection framework. Schotbruch was developed to solve this problem. Schotbruch provides an interface between the generated emulator and the FAIL* framework, which abstracts all elements required by the fault injection framework and manages the communication between emulator and fault injection framework. In addition, it is possible to use Schotbruch to generate the architecture-specific parts of the FAIL* connection from the architecture model. This integration between emulator and FAIL* was done using the model generated from the RISC-V model. Subsequently, the emulator was examined for its performance and error-proneness and compared with the Bochs emulator.

KURZFASSUNG

Transiente Fehler, die durch kosmische Strahlung in Prozessoren auftreten sind schon seit langem ein Problem. Um mit diesen Fehlern besser umgehen zu können, muss herausgefunden werden, wie sich die Systeme beim Auftreten von transienten Fehlern verhalten. Da die Fehlerinjektion in die echte Hardware oft teuer und aufwendig ist, hat sich die Fehlerinjektion in Emulatoren als Alternative ergeben. Ein solches Tool für die Fehlerinjektion in Emulatoren ist das FAIL*-Framework. Das Problem mit der Fehlerinjektion in Emulatoren ist aber, dass die Emulatoren zuerst entwickelt werden müssen. Dies ist aber zeitaufwendig. Das Tool Sail löst das Problem, indem es die Generierung von Emulatoren aus Prozessorarchitekturmodellen ermöglicht. Das Problem ist jetzt, wie die Einbindung der generierten Emulatoren an das Fehlerinjektions-Framework automatisch ablaufen kann. Um dieses Problem zu lösen wurde Schotbruch entwickelt. Schotbruch sorgt für ein Interface zwischen dem generierten Emulator und dem FAIL*-Framework, welches alle vom Fehlerinjektions-Framework benötigten Elemente abstrahiert und die Kommunikation zwischen Emulator und Fehlerinjektions-Framework bewältigt. Zusätzlich ist es möglich mit Schotbruch die architekturspezifischen Teile der FAIL*-Anbindung aus dem Architekturmodell generieren zu lassen. Diese Anbindung zwischen Emulator und FAIL* wurde mithilfe des aus dem RISC-V-Modell generierten Modell durchgeführt. Anschließend wurde der Emulator auf seine Performance und seine Fehleranfälligkeit untersucht und mit dem Bochs-Emulator verglichen.

INHALTSVERZEICHNIS

Abstract **v**

Kurzfassung **vii**

1 Einleitung **1**

2 Grundlagen **3**
- 2.1 Fehler . . . 3
  - 2.1.1 Fault-Klassen . . . 4
  - 2.1.2 Single Event Upsets . . . 6
- 2.2 Fehlerinjektion . . . 9
  - 2.2.1 Hardwarebasierte Fehlerinjektion . . . 10
  - 2.2.2 Softwarebasierte Fehlerinjektion . . . 10
  - 2.2.3 Simulationsbasierte Fehlerinjektion . . . 11
  - 2.2.4 Emulationsbasierte Fehlerinjektion . . . 11
  - 2.2.5 Hybride Fehlerinjektion . . . 12
  - 2.2.6 Verwandte Arbeiten . . . 12
- 2.3 FAIL* . . . 13
  - 2.3.1 Architektur . . . 13
  - 2.3.2 Fault-Toleranz-Einschätzungskreislauf . . . 14
- 2.4 RISC-V . . . 15
  - 2.4.1 Architektur . . . 15
- 2.5 Sail . . . 16
- 2.6 Zusammenfassung . . . 18

3 Implementierung **19**
- 3.1 Schotbruch . . . 19
  - 3.1.1 Schritte einer Fehlerinjektion . . . 20
- 3.2 Das Schotbruch-Interface . . . 20
  - 3.2.1 Übersicht . . . 20
  - 3.2.2 SailController . . . 21
  - 3.2.3 SailCPU . . . 22
  - 3.2.4 SailSimulator . . . 22
  - 3.2.5 SailArchitecture . . . 22
  - 3.2.6 SailTimer . . . 22
  - 3.2.7 SailWrapper . . . 22

3.3 RISC-V 22
  3.3.1 RISC-V-Emulator 23
  3.3.2 RISC-V-Interface 24
    3.3.2.1 Automatisierung 25
3.4 Zusammenfassung 25

4 Evaluation **27**
4.1 Performance 27
  4.1.1 Versuchsaufbau 27
  4.1.2 Ergebnisse 30
  4.1.3 Zusammenfassung 35
4.2 Fehleranfälligkeit 36
  4.2.1 Versuchsaufbau 36
  4.2.2 Ergebnisse 38
  4.2.3 Zusammenfassung 41

5 Zusammenfassung **43**

Verzeichnisse **45**
Abbildungsverzeichnis 45
Tabellenverzeichnis 47
Quellcodeverzeichnis 49
Literatur 51

1

EINLEITUNG

Prozessoren werden immer kleiner und brauchen immer weniger Energie, was dazu führen kann dass sie immer anfälliger für transiente Fehler werden. Diese transienten Fehler können zu jeder Zeit und zufällig irgendwo im Prozessor auftreten und im schlimmsten Fall große Auswirkungen haben. Um dem Entgegenzuwirken müssen Prozessoren auf ihre Fehleranfälligkeit überprüft werden, damit entsprechende Maßnahmen getroffen werden können um die Fehleranfälligkeit der Prozessoren zu verringern. Die Methode um die Fehleranfälligkeit zu überprüfen ist die Fehlerinjektion. Die Fehlerinjektion in die Hardware ist aber teuer und aufwendig, weshalb es die Fehlerinjektion in Emulatoren als Alternative gibt. FAIL* ist ein Tool, welches die Fehlerinjektion in Emulatoren ermöglicht. Für die Fehlerinjektion in Emulatoren ist das große Problem aber, dass die Emulatoren erstmal entwickelt werden müssen. Sail ist ein Tool welches dieses Problem löst, in dem es die Generierung von Emulatoren aus Prozessorarchitekturmodellen ermöglicht. Das Problem ist jetzt aber, wie können der generierte Prozessoremulator und das Fehlerinjektions-Framework miteinander verbunden werden, sodass die Fehlerinjektion in den Emulator problemlos abläuft.

Das Ziel dieser Arbeit soll es sein eine automatische Möglichkeit zu bieten, wie man diese generierten Emulatoren an das Fehlerinjektions-Framework anbinden kann, sodass es möglichst einfach ist diese Emulatoren auszutauschen. Dadurch soll es möglich sein, verschiedene Prozessorarchitekturen auf ihre Fehleranfälligkeit zu testen und diese Ergebnisse dann miteinander vergleichen zu können. Diese Anbindung zwischen dem Fehlerinjektions-Framework soll möglichst generisch gehalten werden, sodass es möglich ist jeden von Sail generierten Emulatoren an das FAIL*-Framework anzubinden.

GRUNDLAGEN 2

Dieses Kapitel beschreibt die Grundlagen, die für den Rest der Arbeit von Bedeutung sein werden. Zuerst wird auf Fehler in einem System und anschließend auf die Fehlerinjektion im Allgemeinen eingegangen. Danach wird das in dieser Arbeit verwendete Fehlerinjektionsframework FAIL* genauer beschrieben. Desweiteren wird die Befehlssatzarchitektur RISC-V, die als Zielsystem der Fehlerinjektion dient, und die Befehlssatzarchitekturensprache Sail, mit der die Befehlssatzarchitektur-Back-Ends für den FAIL*-Fehlerinjektor generiert werden, beschrieben.

2.1 Fehler

In diesem Kapitel werden die verschiedenen Fehlerarten, die in einem System auftreten können, beschrieben. Diese Arten werden anschließend genauer aufgeteilt und es wird genauer auf einen bestimmten Teil der Fehler, den Single Event Upsets, eingegangen.

Der Begriff Fehler wird im deutschen Sprachgebrauch für eine Vielzahl von verschiedenen, feineren Systemaspekten verwendet. Das Online Wörterbuch LEO, zum Beispiel bietet mehr als 30 verschiedene, englische Übersetzungen für den Begriff Fehler an [Feh]. Um eine bessere Differenzierung der verschiedenen Fehlerarten zu gewährleisten, werden im folgenden, anstatt dem Begriff Fehler, die passenden englischen Begriffe benutzt. Im Bereich technischer Systeme sind drei Fehlerbegriffe wichtig: Fault, Error und Failure [ZAV04]. Ein Fault ist ein physischer Mangel, eine Imperfektion oder ein Defekt innerhalb einer Hardware- oder Softwarekomponente. Ein Error hingegen ist eine Abweichung von der Korrektheit und ist die Manifestation eines Faults. Und ein Failure ist die Nichterfüllung einer fälligen oder erwarteten Aktion.

Die Reihenfolge, in der in einem System Fehler auftreten, beginnt zuerst mit einem Fault. Dieser Fault führt anschließend zu einem Error, der schlussendlich einen Failure erzeugt. Nicht jeder Fault führt zwingend zu einem Error, denn die Faults können überschrieben werden oder sie können das System gar nicht erst beeinflussen. Ein Fault wird als aktiv bezeichnet, wenn er einen Error erzeugt, und wird anderenfalls als verborgen bezeichnet [Avi+04]. Wenn ein Fault einen Error erzeugt, tritt in den meisten Fällen der Error in einem internen Zustand einer Systemkomponente auf und der externe Zustand des Systems wird nicht sofort beeinflusst, sprich, von außen kann der Error nicht beobachtet werden [Avi+04]. Erst wenn der Error von außen betrachtet werden kann, existiert ein Failure im System. Dennoch muss nicht jeder Error zwingend den externen Zustand des Systems verändern und zu einem Failure führen. Ein Error, der nicht zu einem Failure führt, ist zum Beispiel ein falscher interner Zustand, der vom erwarteten internen Zustand abweicht, aber trotzdem noch das richtige Ergebnis nach außen liefert.

Auf Faults wird in der nächsten Sektion noch genauer eingegangen, da sie der erste Schritt in der Manifestation eines Failures sind und sie in dieser Arbeit eine große Rolle spielen.

2.1.1 Fault-Klassen

Nicht alle Faults sind gleich, denn Faults können verschiedenste Ursprünge haben. Die folgende Einordnung von Faults in Klassen, basiert auf der Klassifizierung aus [Avi+04]. Faults können anhand von acht grundlegenden Kriterien von einander unterschieden werden, die nachfolgend genauer erläutert werden.

Das erste Kriterium welches vorgestellt wird ist der Zeitpunkt des Auftretens des Faults. Wenn ein Fault während der Systementwicklung, der Systemwartung im Betrieb oder der Erzeugung von Verfahren zum Betrieb oder zur Wartung des Systems auftritt, dann ist dieser Fault ein Development-Faults. Ein Beispiel für solch einen Fault ist ein Tippfehler im Quellcode, der noch während der Entwicklung den Fault auslöst. Gegenüber der Development-Faults stehen die Operational-Faults, die während der Dienstbereitstellung in der Nutzungsphase auftreten. Würde der vorher beschriebene Tippfehler erst bei der Nutzung des Systems durch den Benutzer auftreten, dann wäre es ein Development-Fault.

Ein weiteres Kriterium anhand dessen Faults unterteilt werden können, sind die Grenzen eines Systems. Wenn der Ursprung des Faults innerhalb des Systemgrenzen liegt, wird von einem Internal-Fault gesprochen. Der Fault der durch den Tippfehler im Quellcode ausgelöst worden ist, ist ein Internal-Fault. Wenn der Ursprung des Faults aber außerhalb des Systems liegt, dann ist der Fault ein External-Fault. Ein Beispiel für einen External-Fault ist, wenn eine Person mit einem Hammer auf einen Computer einhämmert und dadurch einen Fault erzeugt. Dieser erzeugte Fault ist ein External-Fault.

Desweiteren können Faults noch anhand ihrer Ursache kategorisiert werden. Wenn der Fault aus natürlichen Begebenheiten und nicht durch menschlichen Einfluss entstanden ist, dann zählt er zu den Natural-Faults. Wenn zum Beispiel ein kosmisches Teilchen in einen Computer einschlägt und einen Fault erzeugt, dann ist dieser Fault ein Natural-Fault. Wenn der Fault aber keinen natürlichen Ursprung hat und aus den Handlungen eines Menschen entstanden ist, dann ist das ein Human-Made-Fault. Der Fault der durch den Tippfehler entstanden ist und der Fault, der durch den Menschen mit dem Hammer entstanden ist, sind beide Human-Made-Fault, da beide von einem Menschen verursacht worden sind.

Ein weiteres Kriterium ist die Dimension eines Faults. Faults die ihren Ursprung in der Hardware haben oder die Hardware beeinflussen, sind Hardware-Faults. Faults, die die Software beeinflussen, gehören stattdessen zu den Software-Faults. Der Fault, der aus dem Tippfehler entstanden ist, zählt zu den Software-Faults, wohingegen der Fault, der vom Menschen mit dem Hammer verursacht wurde, zu den Hardware-Faults zählt.

Zusätzlich können Faults auch noch anhand ihres Ziels klassifiziert werden. Ein Fault, der von einem Menschen in das System eingebracht wurde, mit dem böswilligen Ziel, dem System Schaden zuzuführen, zählt zu den Malicious-Faults. Wenn hingegen der Fault ohne ein böswilliges Ziel in das System eingebracht wurde, dann ist dies ein Non-Malicious-Fault. Der Fault der durch den Tippfehler verursacht wurde, zählt zu den Non-Malicious-Faults. Ein Beispiel für einen Malicious-Fault ist der Fault, der durch die Person mit dem Hammer verursacht worden ist, da die Person die Absicht hatte, dem System Schaden zuzuführen

Faults können auch noch anhand ihrer Intention in Gruppen eingeordnet werden. Wird der Fault vorsätzlich in das System eingebracht, gehört er zu den Deliberate-Faults. Der Fault der durch den Menschen mit dem Hammer verursacht wurde ist ein Beispiel für einen Deliberate-Fault. Ein Non-Deliberate-Fault, ist ein Fault der unbewusst in das System eingebracht worden ist. Ein Beispiel dafür ist der Fault der durch den Tippfehler entstanden ist.

Zusätzlich können Faults auch nach dem Kriterium der Kompetenz gruppiert werden. Ein Fault, der unbeabsichtigt in das System eingebracht wurde, ist ein Accidental-Fault. Wenn aber der Fault das Resultat aus einem an fachlicher Kompetenz der befugten Person oder das Resultat der Unangemessenheit der Entwicklungsumgebung ist, dann zählt er zu den Incompetence-Faults. Ein Beispiel für einen Accidental-Fault ist der schon genannte Fault, der aus dem Tippfehler hervorgegangen ist. Ein Beispiel für einen Incompetence-Fault ist ein Fault, der entstanden ist, weil eine Person nicht gewusst hat dass sie nicht mit einem Hammer auf einen Computer hämmern darf und trotzdem auf den Computer geschlagen hat.

Zu guter Letzt können Faults noch anhand ihrer Persistenz im System eingeordnet werden. Falls die Anwesenheit des Faults im System als zeitlich unbegrenzt angenommen wird, der Fault permanent im System ist, dann zählt er zu Permanent-Faults. Falls aber die Anwesenheit des Faults im System zeitlich begrenzt ist, der Fault transient im System ist, dann zählt er zu den Transient-Faults. Ein Permanent-Fault wäre zum Beispiel ein Fault, der entstanden ist, weil das System durch, zum Beispiel eine Person mit einem Hammer, permanent beschädigt worden ist. Ein Transient-Fault ist zum Beispiel ein Bit-Flip, der durch eine Ladungsumverteilung in einem Hableiter entstanden ist. Nach dem Neustart des Systems existiert dieser Fault nicht mehr.

Mithilfe dieser acht Kriterien können 31 verschiedene Fault-Klassen unterschieden werden, da nicht alle Kriterien miteinander kombiniert werden können. Zum Beispiel können Natural-Faults nicht vorsätzlich ins System eingebracht werden und gehören somit nicht zu den Deliberate-Faults. Diese 31 Fault-Klassen können zusätzlich noch in drei, teilweise überlappende Hauptkategorien eingeteilt werden. Diese drei Fault-Kategorien sind Development-Faults, Physical-Faults und Interaction-Faults. Zu den Development-Faults zählen alle Fault-Klassen, die während der Entwicklung auftreten. Die Physical-Faults haben alle einen Einfluss auf die Hardware. Und die Interaction-Faults beinhalten alle externen Faults. In Abbildung 2.1 sind die Fault-Kriterien, und ihre zugehörigen Klassen und Kategorien zusehen.

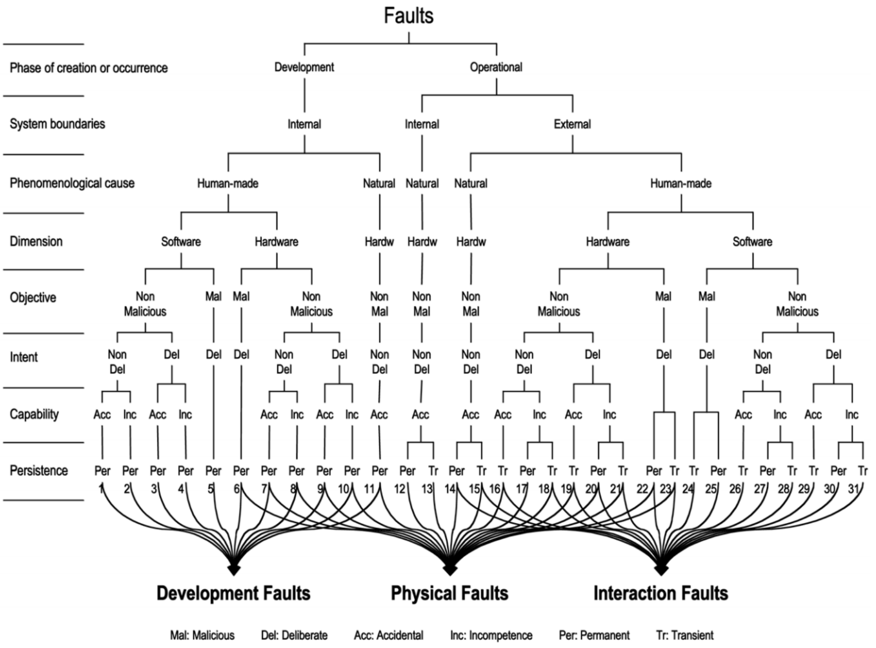


Abbildung 2.1 – Die Fault-Kriterien und ihre zugehörigen Klassen und Kategorien (*entnommen aus [Avi+04]*).

Die Fault-Klassen, die im Laufe dieser Arbeit genauer betrachten werden, sind die beiden Fault-Klassen aus Abbildung 2.1 mit den Nummern 13 und 15. Die Faults dieser Klassen treten während der Nutzungsphase des Systems auf, haben entweder einen Ursprung innerhalb oder ausserhalb der Systemgrenzen und sind natürlichem Ursprungs. Da diese Faults natürlichem Ursprungs sind, haben sie kein böswilliges Ziel und wurden unbewusst und unbeabsichtigt in das System eingebracht. Zusätzlich beeinflussen diese Faults ausschließlich die Hardware und sind transient. Zu diesen beiden Klassen gehören auch die Single Event Upsets, die im nächsten Abschnitt genauer betrachtet werden.

2.1.2 Single Event Upsets

Das folgende Kapitel zu Single Event Upsets basiert auf dem Paper [WA08] von F. Wang und V. D. Agrawal. Ein Single Event Upset wird vom NASA Thesaurus [Nas] als ein strahlungsinduzierter Error in einer mikroelektronischen Schaltungen definiert, der entsteht, wenn geladene Teilchen ihre Energie durch Ionisierung an das Medium, welches sie durchqueren, abgeben und dadurch eine Spur aus Elektronen-Loch-Paaren hinterlassen. Diese kann in Halbleitern zu einer Änderung der Ladungsverteilung führen und kann schlussendlich zu einem einzelnen Bit-Flip im System führen. Diese Ladungsverteilung beschädigt das System nicht permanent. Single Event Upsets zählen zu den sogenannten Soft Errors. Soft Errors bewirken nur eine temporäre Zustandsänderung im System und treten zufällig auf. Somit zählen Soft Errors auch zu den transienten Faults. Diese transienten Faults können spätestens durch den Neustart des System behoben werden, oft werden sie aber schon im Betrieb überschrieben und existieren anschließend nicht mehr im System.

Schon zwischen 1954 und 1957 gab es erste Berichte zu Single Event Upsets. Bei oberirdischen Atombombentests wurden eine Vielzahl von elektronischen Anomalien in den Überwachungsgeräten festgestellt. Diese Anomalien konnten später auf Single Event Upsets zurückgeführt werden. Diejenigen Geräte, die nach einer hohen Strahlungsdosis, nicht mehr funktionierten, wurden auch mit dem Begriff "Hard Fail" bezeichnet [Zie+96].

In dem im Jahr 1975 erschienen Paper [BSH75] von D Binder, E. C. Smith und A. B. Holman wurden erstmals Unregelmäßigkeiten, die während des Betriebs von Kommunikationssatelliten aufgetreten waren, untersucht. Ein Großteil der in den Satelliten aufgetretenen Unregelmäßigkeiten konnte auf den Ladungsaufbau durch Hochtemperaturplasma zurückgeführt werden. Der restliche Teil der Unregelmäßigkeiten soll durch kosmische Strahlung verursacht worden sein. Diese Strahlen hätten Flipflopschaltungen ausgelöst und dadurch ein Single Event Upsets verursacht. Da die Teilchen, die für die Unregelmäßigkeiten verantwortlich gemacht worden waren, zu schwach waren, um die Erdatmosphäre durchdringen zu können, stellen sie für die Elektronik auf der Erde keine Probleme dar [Zie+96].

Vier Jahre später, im Jahr 1979, wurden im Paper [MW79] von T. C. May und M. H. Woods, Intel-DRAMs untersucht, da es Probleme mit betriebsbedingten Errors gab. Sie fanden heraus, dass das Problem durch den Zerfall von radioaktiven Stoffen im Gehäuse des Speichers entstand. Diese Verunreinigungen im Gehäuse konnten auf eine einzelne Keramikfabrik zurückgeführt werden, die stromabwärts von einer Uranmine gebaut wurde. Das Wasser, welches für den Keramik für die Verpackung der Speicher benutzt wurde, hatte somit einen hohen Anteil an radioaktiven Stoffen, welche dazu führten, dass das Keramik schlussendlich mit radioaktiven Stoffen verunreinigt wurde. Diese Verunreinigungen im Gehäuse des Speichers führten schlussendlich zu Single Event Upsets im Speicher [Zie+96].

Der Begriff "Single Event Upset" wurde erstmals im 1979 veröffentlichten Paper [GWA79] von C. S. Guenzer, E. A. Wolicki und R. G. Allas benutzt. In diesem Paper berichten sie, dass durch

Atomreaktionen entstehende Hochenergie-Protonen und -Neutronen auch zu Single Event Upsets führen können.

Heutzutage stellen Single Event Upsets ein immer größer werdendes Problem dar. Integrierte Schaltungen werden immer kleiner und benötigen immer weniger Energie. Dies führt dazu, dass immer weniger Energie benötigt wird, um eine Ladungsverteilung in einem Halbleiter zu bewirken und somit einen Single Event Upset auszulösen.

Single Event Upsets in Halbleitern können vorallem auf drei verschiedene Ursprünge zurückgeführt werden.

Zum einen kann Alphastrahlung, die beim Zerfall von Atomkernen entsteht, beim Durchqueren eines Halbleiters, ein Single Event Upsets auslösen. Die, in der Natur vorkommenden, radioaktiven Isotope mit der meisten Emissionsaktivität, und somit auch mit der meisten Alphastrahlung, sind Uran und Thorium. Die Quellen der Alphastrahlung mit der größten Bedeutung sind radioaktive Unreinheiten, wie zum Beispiel bleibasierte Isotope in den Lötstellen von Halbleitern, das Gold, welches für die Verbindung von Kabeln und der Deckelbeschichtung verwendet wird, das Aluminium in den Keramikverpackungen oder die Rahmenlegierungen aus Blei.

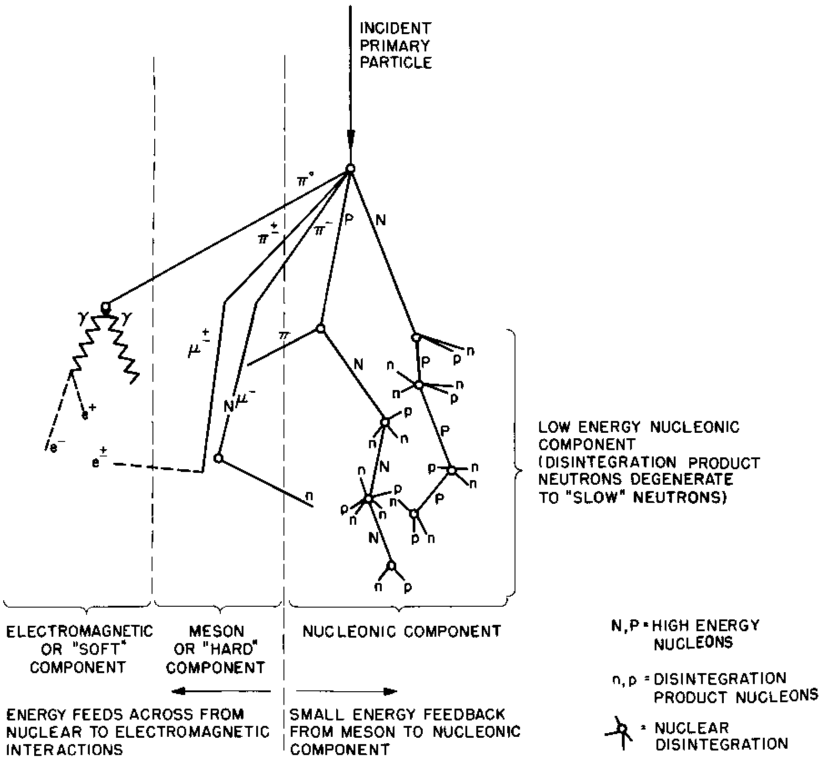
Eine weitere Quelle von Single Event Upsets können hochenergetische Neutronen aus kosmischer Strahlung sein. Diese Strahlung löst die Soft Errors in den Halbleitern durch Sekundärionen aus, welche bei der Reaktion von Neutronen mit Siliziumkernen entstehen. Wie in Abbildung 2.2 zu sehen ist, reagiert die kosmische Strahlung beim Eintreten in die Erdatmosphäre mit selbiger und erzeugt einen Schauer aus Sekundärteilchen. Die dabei entstehenden Teilchen sind vorwiegend Myonen, Neutronen, Protonen und Pionen. Weil Pionen und Myonen kurzlebig sind und Protonen und Elektronen durch Coulomb-Wechselwirkung abgeschwächt werden, sind Neutronen die wahrscheinlichste Quelle von Single Event Upsets in Halbleitern in Bodennähe. Die Dichte der Teilchenschauer ist abhängig von der Höhenlage, je höher, desto dichter ist der Teilchenschauer. Weniger als 1 % der entstehenden Teilchen erreichen Bodennähe.

Der dritte Ursprung ist die Sekundärstrahlung, die beim Zusammentreffen von energiearmen Neutronen der kosmischen Strahlung, mit dem Isotop Bor-10 entsteht. Beim Zusammentreffen vom Neutron und Bor werden Lithiumionen, Gammaphotonen und Alphateilchen erzeugt. Dies ist auch in Abbildung 2.3 zu sehen. Bor wird in Halbleitern vorallem für die Dielektrikumsschicht in Form von Borophosphosilikatglas verwendet.

In modernen Mikroprozessoren werden nur stark gereinigt Materialien verwendet, weshalb die Verunreinigung durch radioaktive Elemente und durch Bor verringert ist. Deshalb ist heutzutage die kosmische Strahlung der Hauptgrund für Single Event Upsets.

Um mit Soft Errors besser umgehen zu können, wurde von R. Raina in [Raj05] ein Vierphasenansatz vorgeschlagen. Die erste Phase ist die Prävention von Soft Errors. In dieser Phase sollen Methoden entwickelt werden, die das System vor Soft Errors schützen sollen. Die zweite Phase, Testen, beinhaltet Methoden die Soft Errors im System erkennen sollen. In der Einschätzungsphase, der dritten Phase, wird die Auswirkung von Soft Errors auf das System eingeschätzt. Die letzte Phase, Online-Wiederherstellung, beinhaltet schlussendlich Methoden, um das System nach Soft Errors wiederherzustellen.

Failures, Errors und Faults stellen, falls unbeachtet, in Systemen ein großes Problem dar. Single Event Upsets können zufällig auftreteten und dadurch schwerwiegende System-Failures auslösen. Der von R. Raina vorgeschlagene Vierphasenansatz zeigt, durch welche Methoden, die Auswirkungen von Faults minimiert werden können. Eine Methode der zweiten Phase, Testen, wird im nächsten Kapitel genauer betrachtet.



Schematic Diagram of Cosmic Ray Shower

Abbildung 2.2 – Kosmische Strahlung, bei ihrem Eintreten in die Erdatmosphäre und der dabei entstehende Schauer an Sekundärteilchen (*entnommen aus [Cos]*).

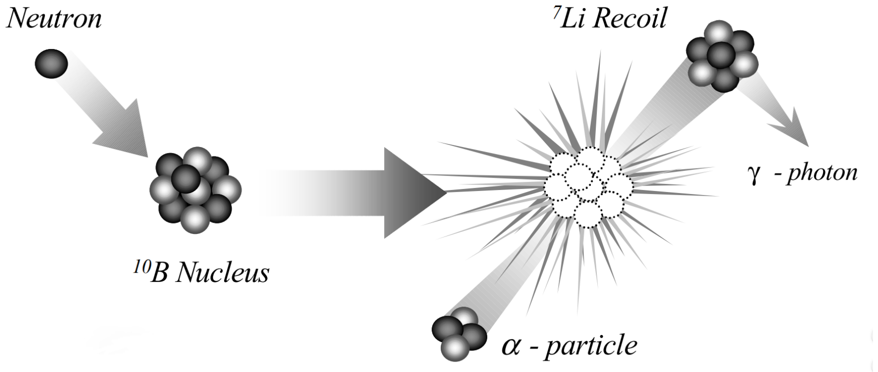


Abbildung 2.3 – Neutronen und Bor-10 kollidieren und erzeugen Sekundärstrahlung (*entnommen aus [Bau01]*).

2.2 Fehlerinjektion

In diesem Kapitel wird das Verfahren der Fehlerinjektion beschrieben. Anschließend werden verschiedene Formen der Fehlerinjektion vorgestellt und ihre Vor- und Nachteile werden aufgezeigt. Zusätzlich werden verwandte Arbeiten vorgestellt und in die verschiedenen Fehlerinjektionsformen eingeordnet.

Fehlerinjektion wird von Arla [Arl90] als ein Verfahren zur Validierung der Zuverlässigkeit von fehlertoleranten Systemen, welches daraus besteht kontrollierte Experimente durchzuführen, bei denen das Verhalten des Systems beim Vorhandensein von Faults, welche explizit vom Experiment ins System eingebracht werden, beobachtet wird, definiert. Dieses absichtliche Einbringen von Faults wird auch als injizieren bezeichnet.

Das Ziel der Fehlerinjektion ist es, herauszufinden, ob die Reaktion des Systems, während es einer bestimmten, vordefinierten Menge an Faults ausgesetzt ist, der Spezifikation des Systems entspricht. Eine einzelne Fehlerinjektionskampagne besteht aus einer Vielzahl an Experimenten, die jeweils bestimmte Faults in das System injizieren.

Die Daten, die bei der Durchführung von Fehlerinjektion in ein System gesammelt werden, können benutzt werden um Fault-Erkennungs- oder Fault-Behebungsmechanismen für das System zu entwickeln oder zu verbessern. Mithilfe dieser beiden Methoden kann die Fehlerinjektion in ein System folgende, sieben Vorteile für das System bringen [ZAV04]. Zum einen ist es möglich mithilfe der Fehlerinjektion ein Verständnis über die Auswirkungen von echten Faults und das zugehörige Verhalten des Systems in Bezug auf die Funktionalität und die Performance des Systems zu bekommen. Desweiteren kann eine Einschätzung der Wirksamkeit der Fault-Toleranz-Mechanismen im System und dadurch Feedback bezüglich der Verbesserung und Korrektur der Mechanismen gewonnen werden. Zusätzlich kann eine Vorhersage über das fehlerhaften Verhaltens des Systems, welches insbesondere eine Messung der Effizienz der Fault-Toleranz-Mechanismen umfasst getroffen werden. Zum anderen kann eine Einschätzung der Failure-Abdeckung und der Latenz von Fault-Toleranz-Mechanismen getroffen werden. Außerdem kann ein Überblick über die Auswirkungen verschiedener Workloads auf die Effektivität der Fault-Toleranz-Mechanismen geschaffen werden. Zusätzlich ist es auch noch möglich eine Überblick über Schwachstellen im Systemdesign, die durch einen einzelnen Fault zu schwerwiegenden Failures führen können zu bekommen. Und zu guter Letzt kann ein Verständnis über das Verhalten des Systems in der Gegenwart von Faults, zum Beispiel wie die Auswirkungen der Faults durch die Systemkomponenten propagiert werden, gewonnen werden.

Eine Fehlerinjektionskampagne wird normalerweise von einer Fehlerinjektionsumgebung durchgeführt. Diese Umgebung kümmert sich, zum Beispiel um die Injektion und um das Datensammeln nach und vor der Injektion. Typischerweise besteht eine Fehlerinjektionsumgebung aus den folgenden Komponenten [ZAV04]. Ein Teil der Fehlerinjektionsumgebung ist der Fault-Injektor, der dazu dient Faults, die aus den Befehlen des Workloadgenerator stammen, in das System zu injizieren. Die Faults, die in das System injiziert werden, werden zusammen mit ihrem Typ, dem Ort der Injektion und dem Zeitpunkt der Injektion, und eventuellen Hardware-Semantiken oder Software-Strukturen in der Fault-Bibliothek abgespeichert. Ein weiteres Teil der Fehlerinjektionsumgebung ist der Workloadgenerator der die Workloads, die als Eingabe für das Zielsystem verwendet werden sollen, generiert. In der Workloadbibliothek werden die generierten Workloads und zusätzliche Beispiel-Workloads für das Zielsystem abgespeichert. Das Herzstück einer Fehlerinjektionsumgebung ist der Controller, der das gesamte Experiment steuert. Der Monitor verfolgt die Ausführung der Befehle und leitet, wenn immer erforderlich, eine Datensammlung ein. Diese Datensammlungen werden online vom Datensammler durchgeführt. Diese Daten werden dann schlussendlich vom Datenanalysator verarbeitet und analysiert.

Fehlerinjektion wird schon lange als ein notwendiges Verfahren zur Feststellung der Zuverlässigkeit von Systemen angesehen, weshalb viele verschiedene Fehlerinjektionsverfahren entwickelt wurden. Diese Verfahren können in fünf Hauptkategorien eingeordnet werden die nachfolgend beschrieben werden [ZAV04].

2.2.1 Hardwarebasierte Fehlerinjektion

Bei der hardwarebasierten Fehlerinjektion wird das Zielsystem mithilfe von speziell entwickelter Test-Hardware erweitert, so dass das Injizieren von Fehlern in das System und die anschließende Untersuchung der Auswirkungen möglich ist. Das Zielsystem der Fehlerinjektion wird bestimmten Interferenzen ausgesetzt, die einen Fault auslösen können. Die Fehlerinjektion wird auf der physikalischen Ebene durchgeführt, indem zum Beispiel die Hardware durch Umweltfaktoren gestört wird, wie zum Beispiel durch Bestrahlung mit Schwerionenstrahlung oder durch elektromagnetische Interferenzen. Desweiteren können zum Beispiel Spannungsabfälle in die Hardware injiziert werden oder die Werte der Pins der Schaltkreise verändert werden.

Hardwarebasierte Fehlerinjektionen können zusätzlich noch in zwei Kategorien eingeteilt werden. Zum einen in die Hardwarebasierte Fehlerinjektion mit Kontakt, bei der der Injektor direkten, physischen Kontakt mit dem Zielsystem hat. Dadurch ist es zum Beispiel möglich Spannungs- oder Stromänderungen im Zielsystem zu erzeugen. Dem steht die Hardwarebasierte Fehlerinjektion ohne Kontakt gegenüber, bei der der Injektor keinen direkten, physischen Kontakt mit dem Zielsystem hat. Stattdessen wird eine externe Quelle benutzt, die zum Beispiel das Zielsystem mit Schwerionen bestrahlt.

Die Vorteile dieser Fehlerinjektionsmethode sind unteranderem, dass diese Form der Fehlerinjektion gut mit Systemen funktioniert, die eine hohe Zeitauflösung für das Hardware-Asulösen und die Hardware-Überwachung benötigen. Zusätzlich ist das Injizieren in echte Hardware in vielen Fällen die einzige Möglichkeit, die Fault-Abdeckung zu ermitteln. Desweiteren haben die injizierten Faults geringe Störeinflüsse auf das Zielsystem, abseits von der eigentlichen Fehlerinjektion. Ein weiterer Vorteil dieser Technik ist, dass sie nicht intrusiv ist. Dies kommt daher, dass keine Veränderung des Zielsystems für die Fehlerinjektion notwendig ist. Zusätzlich sind die Experimente schnell und können in nahezu Echtzeit durchgeführt werden. Desweiteren muss kein Modell entwickelt und validiert werden um Fehler in ein System injizieren zu können.

Die Nachteile dieser Fehlerinjektionsmethode sind unteranderem, dass durch die Fehlerinjektion dem Zielsystem dauerhafter Schaden zugeführt werden kann. Hinzu kommt noch, dass die Portabilität und die Beobachtbarkeit gering ist. Zusätzlich ist die Anzahl an Injektionspunkten und injizierbaren Faults limitiert. Desweiteren wird Spezial-Hardware für die Fehlerinjektion benötigt.

2.2.2 Softwarebasierte Fehlerinjektion

Bei der softwarebasierten Fehlerinjektion wird die Software, die auf dem Zielsystem läuft, modifiziert, sodass der Systemzustand gezielt verändert werden kann. In diesen Verfahren werden Errors, die aus Faults in der Hardware hervorgehen, in Software reproduziert. Die Faults die dabei injiziert werden könnten, reichen von Register- oder Speicher-Faults über fehlerhafte Netzwerkpakete, bis hin zu fehlerhaften Fehlerzuständen. Softwarebasierte Fehlerinjektion kann, abhängig vom Zeitpunkt der Fehlerinjektion, in zwei Kategorien eingeordnet werden.

Die erste Kategorie ist die Softwarebasierte Fehlerinjektion während der Kompilierung, bei der Programminstruktionen verändert werden, bevor das Programm vom Zielsystem geladen wird. Anstatt die Faults in die Hardware des Zielsystems zu injizieren, injiziert diese Methode die Errors in den Programmcode, um die Auswirkungen von Hardware-Faults, Software-Faults und transienten

Faults zu emulieren. Der modifizierte Programmcode verändert die Instruktionen des Zielprogramms und führt dadurch zur Injektion.

Die zweite Kategorie ist die Softwarebasierte Fehlerinjektion während der Ausführung. Der Unterschied zur vorherigen Fehlerinjektion ist, dass bei dieser Fehlerinjektion die Injektionen zur Laufzeit erfolgen. Dadurch werden Mechanismen benötigt, die die Injektion zur Laufzeit explizit auslösen. Häufig verwendete Auslösemechanismen sind zum Beispiel Time-outs, bei denen nach einer bestimmten Zeit ein Timer ausläuft und die Injektion durch die Generierung eines Interrupts auslöst. Der Timer kann entweder ein Hardware- oder ein Software-Timer sein. Ein weiterer Auslösemechanismus kann eine Exception oder eine Trap sein. Bei diesem Auslösemechanismus transferiert eine Exception oder ein Trap die Kontrolle zum Fault-Injektor, der dann den Fault injizieren kann. Dadurch können Faults injiziert werden, wenn bestimmte, vorher definierte, Ereignisse auftreten. Zu guter Letzt existiert noch der Auslösemechanismus der Code-Einfügung, bei der der Code des Zielprogramms um Instruktionen erweitert wird, die es ermöglichen vor bestimmten Instruktionen Faults zu injizieren. Anstatt, wie bei der Fehlerinjektion während der Kompilierung, nur schon vorhandene Instruktionen zu verändern, fügt dieser Mechanismus neue Instruktionen hinzu.

Die Vorteile dieses Verfahrens sind unteranderem, dass die Fehlerinjektion gezielt an Programmen und Betriebssystemen ausgerichtet werden kann. Zusätztlich ist es möglich die Experimente in nahezu Echtzeit auszuführen. Es wird keine Spezial-Hardware benötigt, was zu geringen Kosten führt. Ein weiterer Vorteil ist, dass kein Modell entwickelt und validiert werden muss. Desweiteren kann die Fehlerinjektion für weitere Fault-Klassen erweitert werden.

Zu den Nachteilen diese Verfahrens zählt, dass nur an Orten, die von der Software aus erreichbar sind, injiziert werden kann. Zusätzlich wird der Quellcode des Programms, welches bei der Fehlerinjektion ausgeführt wird, verändert und das Programm ist deshalb nicht das selbe Programm, wie das welches im Feld ausgeführt wird. Desweiteren ist die Beobachtbarkeit und die Steuerbarkeit der Fehlerinjektion limitiert.

2.2.3 Simulationsbasierte Fehlerinjektion

Bei der simulationsbasierten Fehlerinjektion wird ein High-Level-Simulationsmodell des Zielsystems erstellt, in das die Faults injiziert werden. Dies erlaubt es die Zuverlässigkeit des Systems zu evaluieren, auch wenn nur ein Modell des Systems existiert. Die Errors und Failures des simulierten Systems treten entsprechend einer vorgegebenen Verteilung auf.

Die Vorteile der simulationsbasierten Fehlerinjektion sind unteranderem, dass alle Abstraktionsebenen unterstützt werden. Desweiteren gibt dieses Verfahren dem Entwickler die volle Kontrolle über das Fault-Modell und die Injektionsmechanismen. Ein weiterer Vorteil ist, dass keine Spezial-Hardware benötigt wird, sondern diese Fehlerinjektion auf jedem beliebigen Computer ausgelöst werden kann. Desweiteren ist die Beobachtbarkeit und die Kontrollierbarkeit der Fehlerinjektion sehr hoch. Zusätzlich kann diese Fehlerinjektion sowohl transiente als auch permanente Faults modellieren.

Zu den Nachteilen dieser Methode zählt unteranderem, dass die Entwicklungskosten und der Zeitaufwand für das Entwickeln der Modelle hoch ist. Hinzu kommt noch, dass die Genauigkeit der Ergebnisse von der Genauigkeit des Modells abhängen. Desweiteren können dem Modell die Designfehler der echten Hardware fehlen.

2.2.4 Emulationsbasierte Fehlerinjektion

Die emulationsbasierte Fehlerinjektion wurde entwickelt, um mit den zeitlichen Beschränkungen, die durch die Simulation entstehen, fertig zu werden und die Auswirkungen der Schaltungsumgebung zu berücksichtigen. In diesem Verfahren wird das System mithilfe von Hardware-Prototyping in Hardware emuliert. Die Emulationshardware ist mit einem Host-Computer verbunden, der die Fehlerinjektion steuert und überwacht.

Die Vorteile dieser Fehlerinjektionsmethode sind unteranderem, dass das Verhalten, welches in der finalen Umgebungen erwartet wird, genauer beurteilt werden kann. Zusätzlich aknn die Experimentzeit verringert werden.

Die Nachteile diese Verfahrens sind unteranderem, dass der Aufwand ein Hardware-Emulations-System zu entwerfen hoch ist. Zusätzlich wird die Emulation nur benutzt, um die funktionalen Auswirkungen eines Faults zu analysieren. Die zeitlichen Auswirkungen werden nicht berücksichtigt.

2.2.5 Hybride Fehlerinjektion

Die hybride Fehlerinjektion kombiniert zwei oder mehr der anderen Fehlerinjektionsverfahren. Dadurch können die Vorteile verschiedener Verfahren miteinander kombiniert werden.

2.2.6 Verwandte Arbeiten

Schon seit der Entdeckung von Single Event Upsets wurde nach Möglichkeiten geforscht, um diese Faults gezielt in ein System zu injizieren und ihre Auswirkungen zu studieren. Zu diesem Zweck wurde eine Vielzahl von verschiedensten Tools und Techniken entwickelt. Im Bereich der hardwarebasierten Fehlerinjektion, injizieren G. Miremadi et al. in ihrem Paper [Mir+92] Faults in Mikroprozessoren mithilfe von Schwerionenbestrahlungen und Spannungsschwankungen. MESSALINE [Arl+90], entwickelt von J. Arlat et al., und RIFLE [Mad+94], entwickelt von H. Madeira, sind zwei Fehlerinjektionstools um Faults mithilfe von Sonden, die an den Pins von Prozessoren angebracht sind, in das System zu injizieren.

Im Bereich der softwarebasierten Fehlerinjektion existieren zum Beispiel Tools wie das von Z. Segall et al. entwickelte FIAT [Seg+88] und das von G. A. Kanawati et al. entwickelte FERRARI [KKA92]. Beide Tools emulieren Hardware-Faults und -Errors mithilfe von Software, die auf der Zielhardware läuft. Das Tool SASSIFI [Har+17] von S. K. S. Hari et al. hingegen ermöglicht es softwarebasierte Fehlerinjektion auf NVIDIA GPUs durchzuführen. GOOFI [Aid+01], entwickelt von J Aidemark et al., und GOOFI-2 [SBK10], entwickelt von D. Skarin et al. sind zwei Tools die die hardwarebasierte Fehlerinjektion mithilfe von Pins, aber auch softwarebasierte Fehlerinjektion vor und während der Laufzeit ermöglichen.

Simulationsbasierte Fehlerinjektionstools sind zum Beispiel die VHDL-basierten Tools MEFISTO [Jen+94], entwickelt von E. Jenn et al., und VERIFY [STB97], entwickelt von V. Sieh et al. Diese beiden Tools erlauben die Injektion von Faults in Modelle, die mithilfe der Hardwarebeschreibungssprache VHDL geschrieben wurden. DEPEND [Gos97] ist ein von K. K. Goswami entwickeltes Tool, mit welchem es möglich ist ein Hardwarearchitekturmodell zu designen und anschließend mit Faults zu injizieren. Auch zu schon vorhanden Simulatoren existieren Tools um Faults zu injizieren. FSFI [Cha+11] ist ein von W. Chao et al. entwickeltes Fehlerinjektionstool für den Systemsimulator SAM. Ein weiteres Tool ist das von K. Parasyris et al. entwickelte GEMFI [Par+14], welches ermöglicht Faults in den System- und Prozessorsimulator Gem5 zu injizieren. GEMFI unterstützt alle vom Gem5-Simulator unterstützen Prozessormodelle und Befehlssatzarchitekturen.

Im nächsten Kapitel wird das Fehlerinjektionsframework FAIL* vorgestellt. FAIL* ist primär ein simulationsbasiertes Fehlerinjektionstool erweitert um die Triggermechanismen aus der softwarebasierten Fehlerinjektion, wobei die hardwarebasierte und softwarebasierte Fehlerinjektion auch möglich ist.

2.3 FAIL*

Das folgende Kapitel basiert auf dem Paper [Sch+15] von H. Schirmeier et al. FAIL* - FAult Injection Leveraged - ist ein offenes und vielseitiges Fehlerinjektionsframework zur Fehlerinjektion auf Architekturebene. Das Framework erlaubt die kontinuierliche Einschätzung und Quantifizierung von Fehlertoleranz im iterativen Softwareentwicklungsprozess. Zusätzlich bietet FAIL* wiederverwendbare und zusammensetzbare Fehlerinjektionskampagnen, Vor- und Nachbearbeitungsanalysen zur Erkennung von sensiblen Stellen in der Software, eine abstrahierten Back-End-Implementierung für mehrere Hardware- und Simulatorplattformen und die Möglichkeit die Fehlerinjektionskampagnen durch Parallelisierung zu Skalieren, an. FAIL* ermöglicht zwischen verschiedenen Back-Ends hin und her zu wechseln und die selbe Fehlerinjektionskampagne auf mehreren Back-Ends auszuführen, ohne den Experimentcode umschreiben zu müssen. Back-Ends für die Simulatoren Bochs, Gem5 und QEMU, sowie für das Injizieren in ARM Cortex-A9 Entwicklungsboards, sind vorhanden. Durch das Wechseln der Back-Ends ist es möglich hardwarebasierte, softwarebasierte und simulationsbasierte Fehlerinjektion durchzuführen.

FAIL* ist in der Programmiersprache C++ implementiert und wird durch Skripte in Bash, Perl und Python unterstützt. Die Callback-Hooks der Simulator-Back-Ends sind in der aspektorientierten Programmiersprachenerweiterung von C++, AspectC++, implementiert. Die Callback-Hooks werden vom Back-End verwendet, um bestimmte Ereignisse an das Framework zurück zu melden. Diese Callback-Hooks wurden in AspectC++ geschrieben, damit der Code des Simulators nicht verändert werden muss.

2.3.1 Architektur

Die Architektur von FAIL* ist in Abbildung 2.4 zu sehen. FAIL* ist auf oberstem Level in eine Client-Server-Architektur aufgeteilt. Zum einen gibt es einen zentralen Kampagnen-Controller, der Aufgaben aus einer zentralen Datenbank an verschiedene FAIL*-Instanzen verteilt. Die von den einzelnen FAIL*-Instanzen an den Server zurückgesendeten Experimentergebnisse werden vom Kampagnen-Controller gesammelt und in der Datenbank gespeichert.

Zum anderen gibt es die einzelnen FAIL*-Instanzen, die die Fehlerinjektion parallelisiert durchführen. Eine einzelne FAIL*-Instanz ist ein Simulator oder Debugger, der durch Callback-Hooks an bestimmten Code-Stellen erweitert wurde. Diese Callback-Hooks ermöglichen die Steuerung und den Zugriff auf den internen Systemzustand des Back-Ends.

Die Hauptkomponente von FAIL* ist die Execution-Environment-Abstraction, welche ein gemeinsames Interface für die verschiedenen Back-Ends liefert. Dieses Interface bietet Zugriff auf die Metainformation und den Zustand der Back-Ends, und erlaubt es Ereignisbehandlungsroutinen für verschiedene Ereignisse zu erstellen. Die Metainformation, die das Interface abstrahiert sind: die Anzahl der CPUs, die Anzahl der Register, die plattformunabhängige Benennungen von speziellen Registern, die Wortgröße, die Byte-Ordnung und die Speichergröße. Auf folgende Zustände hat das Interface Zugriff: Schreib- und Lesezugriff auf die CPU-Register und den Speicher, die Injektion von externen Interrupts, die Zeit des Back-Ends, die Speicherung und Wiederherstellung des Zustandes des Back-Ends, und das Neustarten des Back-Ends. Für die folgenden Ereignisse können Ereignisbehandlungsroutinen erstellt werden: das Erreichen einer bestimmten Programminstruktion, der Zugriff auf eine bestimmte Speicheradresse, das Auslösen einer CPU-Exception, das Auftreten eines externen Interrupts, dem Vergehen einer bestimmten Zeit des Back-Ends und der seriellen Ein- oder Ausgabe.

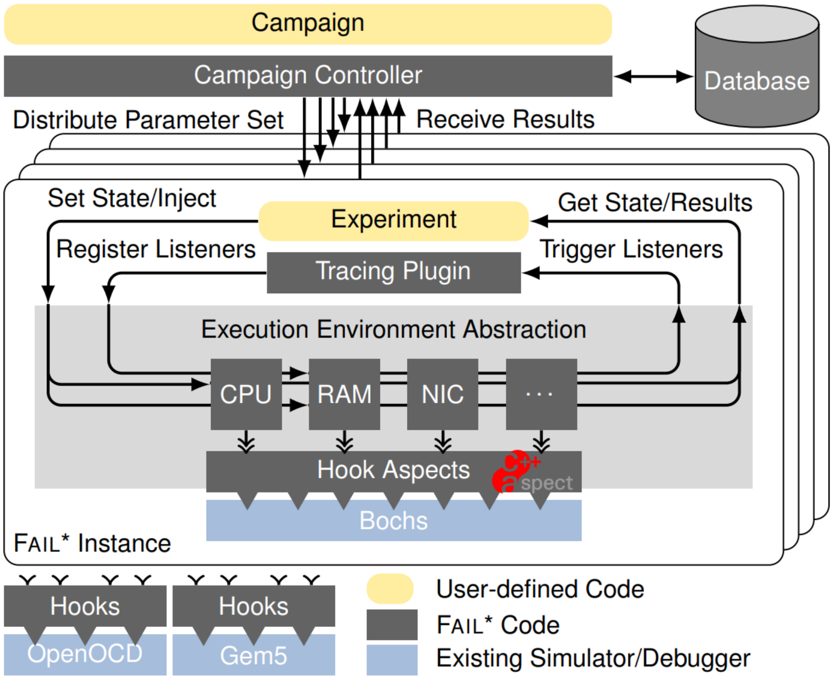


Abbildung 2.4 – FAIL*-Architektur: Der Kampagnen-Controller verteilt Parametersets an die einzelnen FAIL*-Instanzen. Diese steuern ihr Back-End mithilfe der Execution-Environment-Abstraction und senden nach der erfolgreichen Injektion und Ausführung ihr Ergebnis zurück an den Controller (*entnommen aus [Sch+15]*).

2.3.2 Fault-Toleranz-Einschätzungskreislauf

Der Fault-Toleranz-Einschätzungskreislauf beschreibt die Phasen, die bei der Einschätzung der Fault-Toleranz eines Systems mithilfe von FAIL* durchgeführt werden und er besteht aus vier Phasen. Diese vier Phasen sind in Abbildung 2.5 zu sehen.

In der ersten Phase wird der Experimentablauf definiert, welcher der Fehlerinjektionskampagne vorgibt, welches Fault-Modell verwendet werden soll und wie der Experiment-Parameter-Raum aussehen soll. Dieser Ablauf wird vom Kampagnen-Controller für jedes einzelne Fehlerinjektionsexperiment der Kampagne vorgegebenen.

Die zweite Phase besteht aus der Pre-Injektionsanalyse, in der ein kompiliertes, binäres Abbild der Software des Zielsystems ausgeführt wird ohne Faults zu injizieren. Dieser sogenannte goldene Run wird benutzt, um das normale Verhalten des Systems und die Laufzeit der Software zu dokumentieren. Zusätzlich wird ein Instruktions- und Speicherzugriffs-Trace erzeugt. Dieser Trace wird für die statische Analyse und für das Fault-Raum-Prunen verwendet. Durch das Prunen kann die Anzahl der benötigten Fehlerinjektionsexperimente stark verringert werden.

In der dritten Phase wird die vollautomatische Fehlerinjektionskampagne ausgeführt. Der Kampagnen-Controller verteilt die Auftragsparameter aus der Datenbank an die verschiedenen FAIL*-Clients. Die Ergebnisse der einzelnen Fehlerinjektionen werden anschließend in einer Datenbank gesammelt.

In der vierten und letzten Phase wird, nach dem Sammeln aller Fehlerinjektionsergebnisse, eine Post-Injektionsanalyse durchgeführt. Diese Analyse kann benutzt werden, um eine Bewertung der

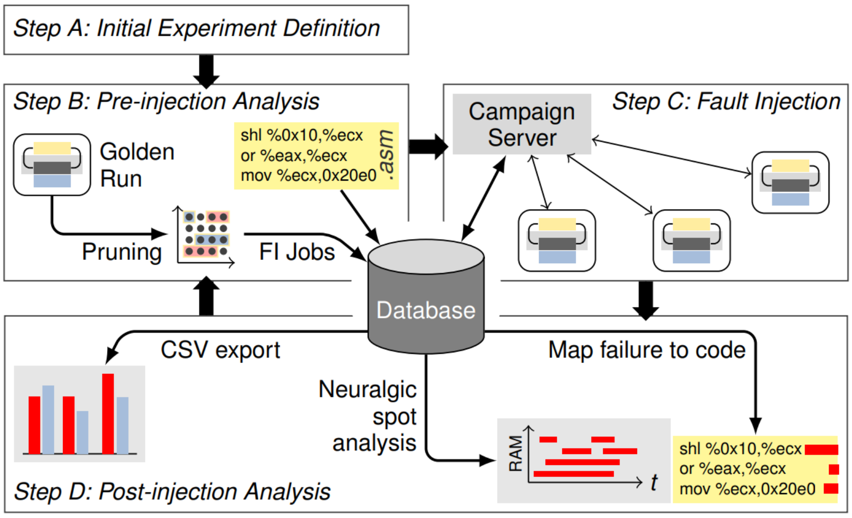


Abbildung 2.5 – Die vier Phasen des Fault-Toleranz-Einschätzungskreislauf (*entnommen aus [Sch+15]*).

Effektivität der Software-Fault-Toleranz zu bekommen oder um den Prozess der Systemhärtung und den Einbau von Fault-Erkennungs- oder Fault-Behebungsmechanismen zu unterstützen.

2.4 RISC-V

RISC-V ist eine offene Befehlssatzarchitektur, die auf den Desingprinzipien der Reduced Instruction Set Computer, kurz RISC, basiert [Rise]. RISC erlaubt es weniger Zyklen pro Instruktion zu haben als ein Complex Instruction Set Computer (CISC) [Risb]. RISC-V wird seit 2010 an der University of California, Berkeley entwickelt und ist unter der Open-Source-Lizenz BSD frei verfügbar [Risf]. Dies ermöglicht es jedem, eigene RISC-V Mikroprozessoren zu entwickeln, herzustellen und zu verkaufen [Risa].

2.4.1 Architektur

Die Spezifikation der RISC-V-Architektur ist in zwei Spezifikationen aufgeteilt [Risg], in den unprivilegierten Teil [Risc] und in den privilegierten Teil [Risd].

Die Basis-Befehlssatzarchitektur von RISC-V ist als little-endian definiert, wobei big-endian und bi-endian durch Erweiterungen unterstützt werden. Die Wortgröße der Basisarchitektur ist 32-Bit, wobei 64-Bit und 128-Bit auch durch alternative Basisarchitekturen unterstützt werden.

Die RISC-V-Architektur ist modular aufgebaut und besteht aus Basis-Befehlssatzarchitekturen und optionalen Erweiterungen. Der komplette Funktionsumfang der verwendeten RISC-V-Architektur wird durch eine Zeichenfolge aus Buchstaben und Zahlen dargestellt. Der Aufbau des Namens beginnt mit der Kennung der Basis-Befehlssatzarchitektur, gefolgt von den Kennungen der Erweiterungen in einer, in der Spezifikation definierten, Reihenfolge. In Tabelle 2.1 ist die Reihenfolge und die Kennungen der zurzeit ratifizierten Erweiterungen zu sehen. Die Kennung der Basis-Befehlssatzarchitektur beginnt mit dem Präfix *RV*, gefolgt von der Wortgröße und der Variante. Zum Beispiel steht *RV32I* für die Integer-Basis-Befehlssatzarchitektur mit einer Wortgröße von 32-Bit.

Diese Basis-Befehlssatzarchitektur alleine kann schon einen vereinfachten Allzweck-Computer mit voller Software-Unterstützung, einschließlich eines Allzweck-Compilers, implementieren. Die Erweiterungen fügen jeweils nur neue Features hinzu und sie sind mit allen Basis-Befehlssatzarchitekturen kompatibel.

Ein Beispiel für die komplette Benennung einer RISC-V-Architektur, mit optionalen Erweiterungen, ist *RV32IMAC*. Dies steht für die RISC-V-Basis-Befehlssatzarchitektur mit einer Wortgröße von 32-Bit und den Erweiterungen für die Integer Multiplikation und Division (*M*), für die atomaren Instruktionen (*A*) und für die komprimierten Instruktionen (*C*).

RISC-V besitzt 32 Integerregister, wie in Tabelle 2.2 zu sehen ist. Das erste Register ist das Nullregister, welches auf die logische Null festverdrahtet ist. Die restlichen 31 Register sind Allzweckregister.

Das RISC-V-Architektur ist eine der Architekturen, für die ein Sail-Modell geschrieben wurde. Die zu diesem Modell gehörende Programmiersprache Sail wird im nächsten Kapitel vorgestellt.

2.5 Sail

Sail ist eine Programmiersprache, mit der die Semantik von Befehlssatzarchitekturen von Prozessoren beschrieben werden kann [Saib]. Sail wird seit 2015 an der University of Cambridge entwickelt [Gra+15], mit dem Ziel eine ingenieurfreundliche, Hersteller-Pseudocode ähnliche Sprache für das Beschreiben von Instruktionssemantiken zu schaffen.

Mithilfe von Sail ist es möglich, Befehlssatzarchitekturspezifikationen automatisch in interaktive Theorembeweiserdefinitionen übersetzen zu lassen, wodurch es möglich ist die Befehlssatzarchitekturen für mechanisierte Beweise zugänglich zu machen [Arm+18]. Für die Theorembeweiser Isabelle, HOL4 und Coq können Definitionen generiert werden. Eine Übersicht über Sail ist in Abbildung 2.6 zu sehen.

Die Sail-Programmiersprache ist eine imperative Programmiersprache mit abhängiger Typisierung für numerische Typen und Bitvektorlängen [Arm+18]. Dies ermöglicht es universelle Bitvektormanipulationen in Befehlssatzarchitekturspezifikationen auf Längen- und Abgrenzungs-Errors zu überprüfen.

Eine Sail-Spezifikation besteht typischerweise aus einem abstrakten Syntax-Typ von Maschineninstruktionen, einer Dekodierfunktion, die binäre Werte zu abstrakten Syntax-Typ-Werten umwandelt; einer Ausführungsfunktion, die beschreibt, wie sich die einzelnen Instruktionen zur Laufzeit verhalten; und weiteren eventuell benötigten Funktionen und Typen [Saic]. Aus der Sail-Spezifikation kann, nach einer Typprüfung durch die Sail-Implementierung, folgendes generiert werden. Zum einen

Erweiterung	Beschreibung
M	Integer Multiplikation and Division
A	Atomare Instruktionen
F	Singleprecision Gleitkommazahlen
D	Doubleprecision Gleitkommazahlen
Q	Quadprecision Gleitkommazahlen
C	Komprimierte Instruktionen
Zicsr	Kontroll- und Statusregister Instruktionen
Zifencei	Instruction-Fetch-Fence

Tabelle 2.1 – Ratifizierte Erweiterungen der RISC-V-Architektur

Name	ABI Name	Beschreibung
x0	zero	Festverdrahtete Null
x1	ra	Rücksprungadress
x2	sp	Stapelzeiger
x3	gp	Globaler Zeiger
x4	tp	Threadzeiger
x5	t0	Temporäre/Alternative Link-Register
x6–7	t1–2	Temporäre Register
x8	s0/fp	Gespeicherte Register/Rahmenzeiger
x9	s1	Gespeicherte Register
x10–11	a0–1	Funktionsargumente/Rückgabewerte
x12–17	a2–7	Funktionsargumente
x18–27	s2–11	Gespeicherte Register
x28–31	t3–6	Temporäre Register

Tabelle 2.2 – Register des RISC-V-Basis-Integerinstruktions-Sets

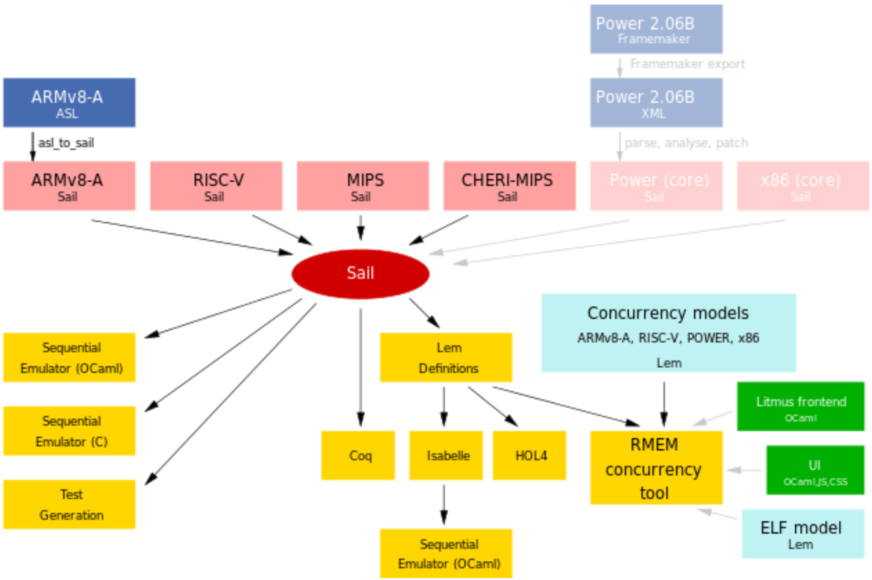


Abbildung 2.6 – Eine Übersicht über Sail. Zu sehen sind die schon vorhandenen Befehlssatzarchitekturmodelle und die Generate die mithilfe von Sail aus den Modellen erzeugt werden können.(*entnommen aus [Saib]*).

kann eine interne Darstellung der vollständigen, typkommentierten Definition, in einem Format das vom Sail-Interpreter ausgeführt werden kann, generiert werden. Der Sail-Interpreter kann zusätzlich benutzt werden, um die Instruktionsdefinitionen zu analysieren und deren potenziellen Register- und Speicherfußabdruck festzustellen. Zusätzlich kann noch eine einfachere Variante der Definition generiert werden, die einfacher ausgeführt oder zu Theorembeweiser-Code konvertiert werden kann. Desweiteren kann eine kompilierte Version der Spezifikation in der Programmiersprache OCaml generiert werden. Und zu guter Letzt kann eine effiziente, ausführbare Version der Spezifikation, kompiliert in C Code, generiert werden.

Sail-Modelle existieren für die Befehlssatzarchitekturen ARM, RISC-V, MIPS, CHERI-MIPS, IBM Power und x86, wobei nicht alle Modelle die komplette Spezifikation abbilden [Said]. Das RISC-V Modell basiert auf der RISC-V-Befehlssatzarchitektur RV32IMAC bzw. RV64IMAC (vergleich Abschnitt 2.3) [Saia].

2.6 Zusammenfassung

In diesem Kapitel wurden die verschiedenen Arten von Fehlern vorgestellt und es wurde gezeigt, welche Fehlerinjektionsmethoden existieren. Anschließend wurde das Fehlerinjektionsframework FAIL* vorgestellt, mit dem die Fehlerinjektion durchgeführt werden soll. Desweiteren wurde noch die Prozessorarchitektur RISC-V vorgestellt und das Sail Tool, mit dem es möglich ist, aus Architekturspezifikationen Emulatoren zu generieren.

IMPLEMENTIERUNG 3

In diesem Kapitel wird die implementierte Anbindung zwischen dem Emulator und dem FAIL*-Framework beschrieben. Dazu wird das Interface vorgestellt, welches die Verbindung zwischen Emulator und FAIL* darstellt. Anschließend wird auf den architekturspezifischen Teil der Anbindung anhand des Modells der RISC-V-Architektur eingegangen.

3.1 Schotbruch

Schotbruch ist eine Erweiterung des Fehlerinjektionsframework FAIL* und ermöglicht es eine automatisierte Ableitung von Injektionsplattformen für transiente Hardwarefehler aus formalen Prozessormodellen durchzuführen. Das Ziel dieser Arbeit war es, das Einbinden von formalen Prozessormodellen an das FAIL*-Framework zu erleichtern und damit das Überprüfen von verschiedenen Befehlssatzarchitekturen auf ihre Fehlertoleranz zu erleichtern. Schlussendlich soll es möglich sein schnell, einfach und ohne viel Aufwand ein schon vorhandenes Befehlssatzarchitekturmodell in das FAIL*-Framework einzupflegen und mit geringem Aufwand Fehler in das Modell injizieren zu können.

Die formalen Prozessormodelle, in die die Fehler injiziert werden sollen, sind in Sail geschriebene Modelle von Befehlssatzarchitektur, basierend auf den Spezifikation selbiger. Aus diesen Modellen werden mithilfe von Sail Emulatoren generiert, die als Back-End für das FAIL*-Framework dienen. Die transiente Fehlerinjektion wird vom FAIL*-Framework durchgeführt, indem die Fehler in Programme injiziert werden, die in den aus den Sail-Modellen generierten Emulatoren laufen.

Ein Designaspekt von FAIL* ist die Möglichkeit, spezifische Back-Ends austauschen zu können und dabei den selben Fehlerinjektionsexperimentcode zu behalten [Sch+15]. Dies ist möglich, da das FAIL*-Framework in zwei Teile aufgeteilt ist. Zum einen in das FAIL*-Framework an sich, welches die Fehlerinjektion durchführt und die Daten sammelt, und zum anderen in das Back-End. Die Back-Ends können beliebig ausgetauscht werden, da die Implementierungen der Back-Ends deren Elemente abstrahieren. Schotbruch verfolgt auch diesen Designaspekt und ermöglicht den einfachen Austausch von, von Sail generierten, Befehlssatzarchitekturemulatoren. Dies wird durch ein spezifisch für Sail-Emulatoren entwickeltes Interface erreicht, welches generisch genug ist, um mit allen von Sail generierten Befehlssatzarchitekturemulatoren kompatibel zu sein. Dieses Interface abstrahiert den Zugriff auf die vom FAIL*-Framework benötigten Komponenten der Sail-Emulatoren. Die Komponenten die abstrahiert werden, sind der emulierte Speicher und die emulierten Register der Sail-Emulatoren. Dies ermöglicht es beliebige Befehlssatzarchitekturen austauschen zu können, ohne die Abstraktionen für jeden neuen Befehlssatzarchitekturemulator neu implementieren zu müssen. Stattdessen müssen nur architekturspezifische Komponenten und Funktionen implementiert werden.

Als Schnittstelle zwischen dem FAIL*-Framework und den architekturspezifischen Komponenten dient das Interface.

Zusätzlich wurde die Sail-Bibliothek erweitert, um die Generierung der architekturspezifischen Implementierungen des Interfaces zu ermöglichen.

Die Implementierung des Sail-Emulators in das FAIL*-Framework wurde am Beispiel des Modells der Befehlssatzarchitektur RISC-V durchgeführt. Das RISC-V-Modell ist ein in Sail handgeschriebenes Modell und basiert auf der RISC-V-Befehlssatzarchitektur RV32IMAC [Saia].

3.1.1 Schritte einer Fehlerinjektion

In diesem Kapitel wird der Ablauf einer, mithilfe vom FAIL*-Framework durchgeführten, Fehlerinjektion in den Sail-Emulator beschrieben. Nur die Seite des Clients wird beschrieben, da der Server nur Aufträge an die einzelnen Clients verteilt und den Emulator nicht ausführt.

Bevor die Fehlerinjektion ansich starten kann, muss erstmal ein Trace aus dem auszuführenden Programm generiert werden. Dieser Trace enthält die ausgeführten Instruktionen und die Speicherzugriffe der Ausführung des Programms ohne Fehlerinjektion, die sogenannte goldene Ausführung. Dieser Trace wird vom FAIL*-Framework benutzt, um die Fehler die injiziert werden sollen, auszuwählen. Zusätzlich wurde in der goldenen Ausführung der Zustand des Emulators nach dem Starten des Programms gespeichert. Anschließend kann die Fehlerinjektion mit dem Starten des Clients beginnen. Beim Starten des Clients wird nicht zuerst das FAIL*-Framework gestartet, sondern zuerst wird der Sail-Emulator gestartet. Der Quellcode des Emulators wurde verändert und beinhaltet nun einen Aufruf der Startfunktion von FAIL*. Diese Funktion wird beim Starten des Emulators aufgerufen, aber noch bevor der Emulator einen Instruktionsschritt durchführt. Die Startfunktion initialisiert das FAIL*-Framework und anschließend die Experimente, die vom Client durchgeführt werden. Im nächsten Schritt fragt der Client den Server nach neuen Jobs. Falls der Server einen Job hat, wird er vom Client bearbeitet. Dazu setzt der Client den Emulator auf einen vorher gespeicherten Zustand. Anschließend wird der Emulator ausgeführt, bis er beim Injektionspunkt des Jobs angekommen ist. Dort wird dann der Fault injiziert und der Emulator wird weiter ausgeführt, bis ein vom Experiment vorher bestimmtes Ereignis auftritt. Dies kann zum Beispiel ein Trap sein, ein Timeout sein oder eine bestimmte vorher definierte Instruktion kann aufgerufen werden. All diese Ereignisse werden mithilfe von Listenern überwacht, die im Experimentcode erstellt worden sind. Nach dem Auslösen eines Listeners werden die gesammelten Ergebnisse dieser einen Fehlerinjektion zurück zum Server geschickt. Anschließend holt sich der Client einen neuen Job vom Server und führt diesen aus und die ganzen Schritte starten von vorne. Falls der Server keine Jobs mehr für den Client hat, wird der komplette Client beendet.

3.2 Das Schotbruch-Interface

3.2.1 Übersicht

Um die Möglichkeit zu bieten, einfach neue Sail-Emulatoren in das FAIL*-Framework einzupflegen, wurden ein Interface entwickelt, welches in mehreren Klassen implementiert wurde und es ermöglicht die Komponenten der Sail-Emulatoren zu abstrahieren.

Das zentrale Interface-Klasse von Schotbruch ist der *SailController*. Diese Klasse erbt von dem von FAIL* implementierten *SimulatorController*, der als Interface für die Steuerung der Back-Ends und den Zugriff auf die Experimentdaten fungiert. Der *SailController* bekommt zusätzlich noch den *SailMemoryManager*, der die Abstraktion des Speichers beinhaltet, und die *SailCPU*, die die Abstraktion für die gesamte restliche Architektur liefert, übergeben. Die *SailCPU* erbt von der konkreten Implementierung der *SailArchitecture*- und der *SailSimulator*-Klassen. Die konkrete Implementierung dieser beiden Interface-Klassen und damit auch die konkrete Sail-Architektur, wird in der *SailArchitectureConfig* festgelegt. Die *SailArchitecture*-Klasse beinhaltet generische Funktionen für den Zugriff auf die Register des Emulators. Die *SailSimulator*-Klasse beinhaltet eine Map, die den Bezeichner des Registers auf die spezifische Variable des Registers mappt, und eine Liste der Sprungbefehlsinstruktionen mit ihren zugehörigen Masken. Zusätzlich stellt die *SailSimulator*-Klasse abstrakte Funktionen für das architekturspezifische Speichern, Laden und Neustarten bereit.

3.2.2 SailController

Der *SailController* implementiert unter anderem Funktionen für das Speichern und das Laden des Zustandes des Emulators. Der Zustand eines Sail-Emulators beinhaltet den Inhalt der Register und den Inhalt des Speichers des Emulators. Die Speicher- und die Ladefunktion wurden beide generisch gehalten, damit neue Sail-Emulatoren möglichst einfach integriert werden können. Der Speicher des Sail-Emulators wird von der Sail-Bibliothek bereitgestellt und die Sail-Bibliothek bietet zudem noch Funktionen für den Zugriff auf diesen Speicher an. Dies ermöglicht einen generischen Zugriff auf den Speicher des Sail-Emulators.

Der Zugriff auf die Register wurde generisch gehalten dadurch, dass die *SailSimulator*-Klasse eine Map verwendet, die aus den Bezeichnern der Register und den Verweisen auf die Register des Emulators besteht. Diese Datenstruktur wird von jedem Sail-Back-End mit ihren eigenen Registern befüllt. Beim Speichern wird diese Datenstruktur durchlaufen und die entsprechenden Werte der Register werden ausgelesen und in eine Datei gespeichert. Beim Laden werden die Registerwerte, die aus der zu ladenen Datei ausgelesen wurden, den entsprechenden Register aus der Map zugeordnet.

Zusätzlich mussten die Speicher- und die Ladefunktionen in jeweils zwei Funktionen aufgeteilt werden, damit das Speichern und das Laden eines Zustandes immer an der selben Stelle in der Ausführung einer Instruktion im Emulator, am Ende, geschieht. Dadurch soll verhindert werden, dass beim Laden eines Zustandes, der Emulator noch Schritte durchführt, die nicht für die geladene Instruktion gedacht waren. Der Ablauf sieht vor, dass das Experiment eine Anfrage auf das Speichern beziehungsweise Laden des Zustandes stellt. Der Emulator überprüft nach jeder Ausführung einer Instruktion, ob eine Anfrage vorliegt und führt sie gegebenenfalls aus. Zusätzlich zu den generischen Speicher- beziehungsweise Ladeschritten, die für alle Architekturen gleich sind, führt die Speicher- beziehungsweise Ladefunktion zusätzlich noch die architekturspezifische Speicher- beziehungsweise Ladefunktion aus, die in der *SailSimulator*-Klasse deklariert ist und vom Architektur-Back-End implementiert wird. Diese architekturspezifische Speicher- beziehungsweise Ladefunktion dient dazu Register zu speichern beziehungsweise zu laden, die keine einheitliche Datenform haben und deshalb nicht mithilfe der generischen Speicherfunktion gespeichert beziehungsweise mithilfe der generischen Ladefunktion nicht geladen werden können.

Eine weitere Funktion, die der *SailController* implementiert ist die generische Funktion zum Neustarten des Emulators. Die generische Neustartfunktion bewirkt ein Neustarten des Emulators indem der Speicher genullt wird und der Instruktionszeiger auf Null gesetzt wird. Falls dies nicht ausreicht um den Emulator neu zu starten, kann stattdessen die architekturspezifische Neustartfunktion in der architekturspezifischen Implementierung der *SailController*-Klasse implementiert werden. Im Gegensatz zu der Speicher- beziehungsweise Ladefunktion, ist die architekturspezifische Neustartfunktion nicht ergänzend zur generischen Neustartfunktion, sondern wird, falls sie vorhanden ist, anstelle der generischen Neustartfunktion ausgeführt.

3.2.3 SailCPU

Die *SailCPU* bietet Abstraktionsfunktion für den Zugriff auf die einzelnen Register der gewählten Architektur. Um diese Funktionen möglichst generisch halten zu können, verwendet sie die vom *Sail-Simulator* geerbte Map der Register. Im Prinzip repräsentiert eine *SailCPU* einen Prozessorkern eines Emulators. Der *SailController* bekommt so viele *SailCPU*-Objekte, wie der Emulator Kerne emuliert. Da die von Sail generierten Emulatoren nur Single-Core emulieren, bekommt der *SailController* nur ein *SailCPU*-Objekt.

3.2.4 SailSimulator

Die *SailSimulator*-Klasse, von dem die *SailCPU* erbt, besteht vorwiegend aus rein virtuellen Funktionen, die noch von der konkreten Architektur implementiert werden müssen. Diese Funktionen sind die architekturspezifischen Speicher-, Lade- und Neustartfunktionen, und die Funktion um die Map der Register mit den Registern zu füllen. Zusätzlich beinhaltet diese Klasse Getter für die Register-Map und die Sprungliste.

3.2.5 SailArchitecture

Die *SailArchitecture*-Klasse, von dem die *SailCPU* erbt, besteht, wie auch schon die *SailSimulator*-Klasse, vorwiegend aus rein virtuellen Funktionen, die noch von der konkreten Architektur implementiert werden müssen. Diese Funktionen sind in diesem Fall die Getter für die Instruktionszeiger-, Stapelzeiger- und Instruktionszählerregister. Desweiteren erbt diese Klasse von der vom FAIL*-Framework implementierten *CPUArchitecture*-Klasse, welches die Register der Back-Ends verwaltet.

3.2.6 SailTimer

Zusätzlich zu den schon genannten Interface-Klassen existiert noch die *SailTimer*-Klasse, die die Verwaltung der generischen Timer-Listener von Schotbruch übernimmt. Damit die Durchführung einer Injektion deterministisch durchgeführt werden kann und immer zur selben Zeit an der selben Stelle injiziert werden kann, messen diese Timer die Zeit, die bei der Ausführung des Emulators vergeht, mithilfe der Anzahl der durchgeführten Instruktionen. Dazu greifen sie auf den Getter für das Instruktionszählerregister zu, welcher von der *SailArchitecture*-Klasse deklariert wird.

3.2.7 SailWrapper

Das FAIL*-Framework ist in der Programmiersprache C++ geschrieben, wohingegen der Sail-Emulator von Sail in C generiert wird. Damit der Sail-Emulator Zugriff auf die Funktionen des FAIL*-Frameworks hat, gibt es den *SailWrapper*, der die Funktionen verpackt und dem Emulator zugänglich machen. Die Funktionen die dabei verpackt werden, sind unteranderem die Auslösefunktionen für verschiedene Ereignisse, die im Emulator auftreten können und die Startfunktion des FAIL*-Frameworks.

3.3 RISC-V

Nach dem im vorherigen Kapitel die generischen Aspekte von Schotbruch vorgestellt wurden, geht dieses Kapitel auf die architekturspezifische Seite von Schotbruch ein. Dazu wird der von Sail generierte RISC-V-Emulator [Saia] als Back-End in das FAIL*-Framework eingegliedert. Um dies zu ermöglichen musste unteranderem, neben der Implementierung des Interfaces für die gewählte Architektur, in diesem Fall RISC-V, der Code des Modells angepasst werden, um die vom *SailWrapper* bereit gestellten Funktionen zu implementieren und damit die Kommunikation zwischen dem Emulator und dem FAIL*-Framework zu gewährleisten.

3.3.1 RISC-V-Emulator

In diesem Kapitel wird auf die Änderungen, die auf der Seite des Emulators durchgeführt werden müssen, um die Kommunikation von Sail nach FAIL* zu gewährleisten. Der RISC-V-Emulator besteht prinzipiell aus zwei Teilen. Zum einen aus dem von Sail generierten RISC-V-Emulator und zum anderen aus dem handgeschriebenen Initialisierungscode, der den generierten Emulator initialisiert und anschließend startet. Diese handgeschriebenen Initialisierungsfunktionen sind in der Regel nicht notwendig um einen vollständigen Sail-Emulator zum Laufen zu bekommen. In diesem Fall hat sich der Entwickler des RISC-V-Sail-Modells dazu entschlossen, zusätzlich zum generierten Emulator eigenen Code zu schreiben, der es ermöglicht den Emulator besser zu steuern.

Die wichtigste Funktion, die der Emulator implementieren und aufrufen muss, ist die Startfunktion von FAIL*. Diese Funktion muss vor dem eigentlichen Starten des Emulators aufgerufen werden, um das FAIL*-Framework zu initialisieren und die entsprechenden Experimente zu starten. Der ideale Platz für diese Funktion ist in diesem Fall der handgeschriebene Initialisierungscode des Emulators, da dort sichergestellt werden kann, dass diese Funktion vor dem eigentlichen Starten des Emulators ausgeführt wird.

Desweiteren muss der Emulator das FAIL*-Frameworks bei bestimmten Ereignissen aufrufen, zum Beispiel wenn der Emulator gerade einen Wert aus dem Speicher liest. Diese FAIL*-Aufrufe können nicht einfach in den C-Code des generierten Emulators geschrieben werden, da, falls sich das Modell des Emulators ändert und der Emulator neu generiert werden muss, diese Aufrufe nicht mit generiert werden. Stattdessen wurden sie direkt in das Sail-Modell eingetragen. Die Sail-Sprache bietet dazu Sprachkonstrukte an, um C-Funktionen aufzurufen. Das geschieht indem die C-Funktionen einer Sail-Funktionen zugewiesen werden und die Sail-Funktion als Verpackung für die C-Funktion dient. Das Sail-Modell wurde mithilfe dieser Sprachkonstrukte um die Aufrufe der Funktionen aus dem *SailWrapper* erweitert. Zusätzlich wurden die Funktionsaufrufe an die passenden Stellen im Modell geschrieben.

Die Rückruffunktionen die zum Modell hinzugefügt worden sind, sind zum einen die Auslöser für den Lesezugriff und den Schreibzugriff auf den Speicher, und die Auslöser für die Auslösung eines Interrupts und eines Traps. Desweiteren existiert eine Rückruffunktion, die nach jeder Ausführung einer Instruktion aufgerufen wird und den aktuellen Instruktionszeiger und die aktuelle Instruktion zurückgibt. Dieser Instruktionszeiger wird an das FAIL*-Framework übergeben um eventuelle Auslöser, die vom Fehlerinjektionsexperiment auf bestimmte Instruktionsadressen gesetzt wurden, auszulösen. Desweiteren wird geprüft ob die Instruktion ein Sprungbefehl der Zielarchitektur ist. Falls die Instruktion ein Sprungbefehl ist, wird der Sprungbefehls-Auslöser ausgelöst. Zusätzlich wird diese Funktion benutzt, um den internen Timer von Schotbruch zu erhöhen, da Sail nicht garantieren kann, dass das Modell den internen Timer von Sail benutzt. Der Timer von Schotbruch ist die Anzahl der ausgeführten Instruktionen, da FAIL* voraussetzt, dass der Timer deterministisch ist und zwei Ausführungen des gleichen Experiments genau das gleiche Ergebnis erzeugen.

Desweiteren sind noch die Speicher- und Ladefunktionen für den Zustand des Emulators zum Modell hinzugefügt worden. Bei ihrem Aufruf befragen sie das FAIL*-Framework, ob eine Speicher- beziehungsweise Ladeanfrage eingegangen ist und führen die Speicher- beziehungsweise Ladeanfrage, falls sie existiert, durch. Diese beiden Funktionen sind notwendig, damit sichergestellt werden kann, dass der Zustand des Emulators nicht gespeichert beziehungsweise nicht geladen wird, wenn der Emulator gerade eine Instruktion ausführt, sondern erst an einer vorher definierten Stelle in der Ausführung der Instruktion. Dies kann zum Beispiel passieren, wenn wärend der Ausführung einer Instruktion, ein Auslöser die Kontrolle zurück an das FAIL*-Framework übergibt und das zurzeit laufende Experiment zu diesem Zeitpunkt den Zustand lädt. Dies sollte verhindert werden, da falls das Laden beziehungsweise das Speichern mitten in der Ausführung der Instruktion passiert, es sein kann dass der Emulator sich in einem Zustand befindet, der die geladene Instruktion nicht ausgelöst hat. Zum Beispiel kann die vorherige Funktion eine Trap ausgelöst haben, anschließend lädt das Experiment den Zustand des Emulators neu und eine Funktion wird geladen die keinen Trap ausgelöst hat. Der Emulator versucht aber weiterhin die notwendigen Schritte für den Trap auszuführen. Dies gilt es zu verhindern. Die Speicher- und die Ladefunktionen sind die beiden letzten Funktionen die vom Emulator bei der Ausführung einer Instruktion ausgeführt werden.

Desweiteren mussten zum Modell des RISC-V-Prozessors noch zwei neue Traps hinzugefügt werden. Der eine neue Trap wird ausgelöst, wenn der Emulator einen internen Fehler feststellt, zum Beispiel wenn eine Adresse nicht übersetzt werden kann. Der andere Trap wird ausgelöst, falls das Modell in einen Zustand gelangt, der vom Modell noch nicht implementiert wurde, da das Modell nicht komplett ist. Diese Traps mussten hinzugefügt werden, da ansonsten sich der Emulator beendet hätte, wenn einer dieser Fehler ausgelöst worden wäre. Dies muss verhindert werden, da dadurch die Ergebnisse der Injektion verloren gehen, da der Rückruf zurück zu FAIL* nie aufgerufen werden würde.

3.3.2 RISC-V-Interface

In diesem Kapitel wird auf die Implementierungen des Sail-Interface eingegangen. Neben den Sail-Interfaces mussten auch noch zwei weitere Quellcodedateien erstellt werden, die wichtige Aspekte des Emulators verwalten. Diese zwei Quellcodedateien sind die *RiscvExterns*- und die *RiscvEnums*-Dateien.

In der *RiscvExterns*-Datei befinden sich die externen Deklarationen aller Registervariablen des Sail-Emulators und die Vorwärtsdeklarationen der zu den Registerdatenstrukturen zugehörigen Datentypen. Dies ist notwendig, um dem FAIL*-Framework Zugriff auf die Inhalte der Register zu ermöglichen. Zusätzlich bietet die *RiscvExterns*-Datei externen Zugriff auf eventuell notwendige Variablen und Funktionen des Emulators, wie zum Beispiel Emulator interne Funktion zum Neustarten des Emulators.

Die zweite Quellcodedatei, *RiscvEnums*, enthält die Enums von verschiedenen Aspekten der Architektur des Emulators. Zum einen enthält die Datei Enums mit den Registern des Emulators. Diese Enums dienen als Bezeichner für die einzelnen Register. Desweiteren befinden sich noch Enums für die Bezeichner der Sprungbefehle, für die Bezeichner der Traps und für die Bezeichner der Interrupts in der Datei. All diese Enums können von den Experimenten benutzt werden, um die einzelnen Sprungbefehl, Traps oder Interrupts zu unterscheiden. Zusätzlich werden die Enums der Register in der architekturspezifischen Implementierung des *SailSimulators* benutzt, um die Verweise auf die Registervariablen bestimmten Bezeichnern zuzuweisen.

Im *RiscSimulator*, der RISC-V-spezifischen Implementierung des *SailSimulators*, wird die Liste bestehend aus den Sprungbefehle und ihren zugehörigen Bitmaske erstellt. Zusätzlich wird, wie im vorherigen Absatz erwähnt, die Map aus den Verweisen auf die Registervariablen und ihren Bezeichnern erstellt. RISC-V-spezifische Speicher- und Ladefunktionen mussten auch implementiert werden, da der Sail-Emulator Register enthält, die nicht dem definierten Registerdatentyp entsprechen. Auch eine RISC-V-spezifische Neustartfunktion wurde implementiert, die die Neuinitialiserungsfunktion des Initialisierungscodes des Emulators aufruft.

In der *RiscvArchitecture*, der RISC-V-spezifischen Implementierung der *SailArchitecture*, wird die interne Repräsentation der Register erstellt. Dazu werden die Registerbezeichner mit der Registergröße, dem Namen des Registers als String und dem Typen des Registers verknüpft. Zusätzlich noch werden die Instruktionszeiger-, Stapelzeiger- und Instruktionszählerregister auf die richtigen Register der Architektur gesetzt.

3.3.2.1 Automatisierung

Um eine neue Architektur in das FAIL*-Framework einzupflegen, musste, im Falle von RISC-V, in vier verschiedenen Quellcodedateien Code für jedes einzelne vom Emulator implementierte Register geschrieben werden. Für jedes einzelne Register wird der Name, der Bezeichner, der Typ, der Verweis auf das Register und der Datentyp der Registervariable benötigt. Der Name, der Bezeichner des Registers und der Verweis auf das Register können von einander abgeleitet, da sich der Variablenname im Sail-Emulator aus einem Präfix und dem Namen des Registers zusammensetzt. Folglich wird nur der Variablenname, der Datentyp und der Typ des Registers benötigt. Der Variablenname und der Datentyp werden beide bei der Generierung des Sail-Emulators miteinander verknüpft.

Aufgrunddessen wurde die Sail-Bibliothek um eine Funktion erweitert, die automatisch die architekturspezifischen Implementierungen der Sail-Interfaces generieren kann. Generiert werden die beiden architekturspezifischen Implementierungen de Sail-Interface-Klassen, die Quellcodedatei mit den Enums der Register und die Quellcodedatei mit den externen Deklarationen der Register. Die Dateien werden parallel mit dem Emulator generiert. Die Funktion benötigt dafür nur den Namen der Architektur und die Typen der einzelnen Register.

Die restlichen Daten bekommt die Funktion aus dem Sail-Modell, welches in die internen Datenstrukturen der Sail-Bibliothek geladen worden wurde.

3.4 Zusammenfassung

In diesem Kapitel wurde die Anbindung zwischen den von Sail generierten Emulatoren und dem FAIL*-Framework gezeigt. Dazu wurde ein Interface entwickelt welches die Abstraktion der Elemente des Emulators und die Kommunikation zischen dem Emulator und FAIL* übernimmt. Zusätzlich wurde das Interface für den generierten RISC-V-Emulator implementiert und vorgestellt. Desweiteren wurde noch die automatische Generierung von architekturspezifischem Code vorgestellt.

EVALUATION 4

In diesem Kapitel wird das in implementierte Back-End, der von Sail generierte RISC-V-Emulator, des FAIL*-Frameworks evaluiert. Dazu wird das Back-End mit dem schon existierenden Bochs-Back-End in Punkten Performance und Fehleranfälligkeit verglichen. Zuerst wird erstmal der Versuchsaufbau beschrieben, der benutzt wird, um die beiden Emulatoren miteinander vergleichen zu können.

Die Analyse wird in zwei Teile geteilt. Der erste Teil fokussiert sich auf die Performance des RISC-V-Emulator und vergleicht diese dann mit der Performance des Bochs-Emulators. Der zweite Teil wird sich auf die Fehleranfälligkeit des im Emulator implementierten RISC-V-Modells beschränken. Diese wird wiederum der Fehleranfälligkeit der vom Bochs-Emulator implementierten Architektur, x86, gegenübergestellt

4.1 Performance

In diesem Kapitel wird die Performance des RISC-V-Emulators untersucht und mit der Performance des Bochs-Emulators verglichen.

4.1.1 Versuchsaufbau

Einer der Hauptaspekte der Fehlerinjektion ist die Performance der Fehlerinjektion und damit einhergehend auch die Performance des Back-Ends, in das die Fehler injiziert werden. Die geladenen Programme in einem Emulatoren werden im Laufe einer Fehlerinjektionskampagne mehrere hunderttausend Mal ausgeführt und ein kleiner Performanceunterschied beim Ausführen kann im Endeffekt eine große Auswirkung auf die Anwendbarkeit dieses Verfahrens haben, da durch eine höheren Zeitaufwand bei der Ausführung auch ein höherer Kostenaufwand entsteht. Aufgrund dessen wird in diesem Kapitel der Versuchsaufbau und die Versuchsdurchführung der Performance-Messung beschrieben.

Um die Performance der Implementierung des RISC-V-Emulators in das FAIL*-Framework zu testen, wurden mehrere Benchmark-Programme im RISC-V-Emulator ausgeführt und Fehler wurden mithilfe des FAIL*-Frameworks in das laufende Programm injiziert. Die selben Experimente wurden auch mit dem Bochs-Emulator [Boc] durchgeführt, für den eine Back-End-Anbindung an das FAIL*-Framework existiert, um einen Vergleich mit einem schon bestehenden Fehlerinjektionssystem zu bekommen. Bochs ist ein Open Source Emulator, für die Intel x86 Prozessorarchitektur [Boc].

Der Versuchaufbau sieht vor, dass die Client-Server-Struktur des FAIL*-Frameworks auf zwei verschiedenen Systemen läuft. Sprich der Server läuft auf einem System und der Client läuft auf einem anderen System. Dies soll verhindern, dass der Server und der Client sich gegenseitig negativ beeinflussen. Der Server wurde auf einem Computer mit einem eingebauten Intel Core i5-7400, mit eine Taktrate von 3.00 GHz, und 32 GB Arbeitsspeicher enthält. Der Client wurde auf einem Computer mit einem eingebauten Intel Core i5-6400, mit einer Taktrate von 2.70 GHz, und wieder 32 GB Arbeitsspeicher. Da der Prozessor, auf dem der Client läuft, vier Kerne hat, wird die Parallelisierbarkeit des FAIL*-Frameworks ausgenutzt, um den Client auf allen vier Kernen parallel auszuführen. Dazu wurde der Client viermal gestartet. Zusätzlich muss noch erwähnt werden, dass der Bochs-Emulator ein Speicherleck hat und deshalb nur 25 Aufträge am Stück ausführen kann, bevor der komplette Client neugestartet werden muss. Der RISC-V-Emulator hat dieses Problem nicht und kann deshalb ununterbrochen laufen und Aufträge ausführen.

Die Benchmark-Programme, die zur Evaluation der Performance dienen sollen, sind in C geschrieben und wurden für beide Zielarchitekturen kompiliert. Dazu wurde eine Umgebung zum Bauen der ausführbaren Dateien geschaffen, die leicht durch neue Architekturen oder Benchmarks erweiterbar ist. Die Benchmarks für den Bochs-Emulator wurden mit dem GCC Compiler, in der Version 9.2, kompiliert. Die Benchmarks für den RISC-V-Emulator wurden mit dem *riscv32-unknown-elf-gcc* Compiler, auch in der Version 9.2, der RISC-V GNU Compiler Toolchain kompiliert. Die RISC-V GNU Compiler Toolchain wurde für die RV32IMAC Architektur und für das ILP32 ABI konfiguriert. Beim Kompilieren wurde darauf geachtet, dass der Kompiliervorgang für beide Architekturen gleich abläuft und beide Compiler die gleichen Flags bekommen. Die C-Flags die beim Kompilieren verwendet wurden, sind für beide Architekturen folgende: "*-O1 -std=c11 -fno-pic -fno-pie -ffunction-sections -fno-stack-protector*". Die Flags optimieren den Code bezüglich der Codegröße und Ausführungszeit(*-O1*), setzten den Sprachstandard auf C11(*-std=c11*), schalten die Generierung von Positionsunabhängigem Code aus(*-fno-pic -fno-pie*), aktivieren die Platzierung jeder Funktion in einem eigenen Sektion in der Ausgabedatei(*-ffunction-sections*) und schalten das Hinzufügen von Code, der das Programm auf Pufferüberläufe prüft, ab(*-fno-stack-protector*). Zusätzlich bekommt der Compiler für den Bochs-Emulator die Flag "*-m32*" übergeben. Diese Flag aktiviert das Kompilieren des Codes für die 32-Bit-Umgebung der x86-Architektur. Der Compiler für den RISC-V-Emulator bekommt zusätzlich die Flag "*-mcmodel=medany*" übergeben. Diese Flag aktiviert die Generierung von Code für das Medium-Any-Code-Modell. Die Binder-Flags der beiden Compiler sind folgende: "*-nostdlib -static -nostartfiles -Wl,–build-id=none*". Diese Flags schalten die Verwendung der Standard-Systemstartdateien oder -bibliotheken beim Binden ab(*-nostdlib -nostartfiles*), deaktivieren das dynamische Binden(*-static*) und deaktivieren die Generierung der Build-ID(*-Wl,–build-id=none*). Zusätzlich bekommen beide Compiler noch ein benutzerdefiniertes Binder-Skript übergeben, welches den Startvorgang des Programms durchführt. Das Binder-Skript ist notwendig für das Starten des Programms im Emulator, da der Emulator selbst kein Betriebssystem und keinen Speicherschutz besitzt.

Zum Benchmarken der Geschwindigkeit des Emulators wurde ein einfaches Programm geschrieben, welches eine bestimmte Anzahl an Schleifendurchläufen durchführt. In Listing 4.1 ist die Hauptfunktion des Programms mit 1000 Schleifendurchläufen zu sehen. In jedem Schleifendurchlauf werden zwei vorzeichenlose Integer um eins erhöht. Die eine Variable wird in jedem Schleifendurchlauf mit der Anzahl der zu laufenden Schleifen verglichen. Die andere Variable wird erst am Ende der benötigten Schleifendurchläufe mit der Anzahl der zu laufenden Schleifen verglichen. Wenn sie gleich der Anzahl der zu laufenden Schleifen ist, dann wird diese Ausführung mithilfe der *ok_marker()*-Funktion als erfolgreich markiert. Falls nicht wird die Ausführung mithilfe der *fail_marker()*-Funktion als nicht erfolgreich markiert. Das Markieren der Ausführung als erfolgreich oder nicht erfolgreich durchgeführt, geschieht mithilfe von zwei Auslösern, die sich im Fehlerinjektionsexperimentcode befinden. Falls eine dieser beiden Funktionen im Programmablauf aufgerufen wird, wird der Auslöser ausgelöst und das entsprechende Ergebnis wird als Ergebnis für diese Fehlerinjektion genommen.

```c
void os_main() {
    unsigned count = 0;
    unsigned iter = 0;

    start_trace();
    while (iter < 1000){
        count++;
        iter++;
    };
    stop_trace();

    if(count == 1000) {
        ok_marker();
    }
    fail_marker();
}
```

Listing 4.1 – Beispielcode für das Benchmarkprogramm für die Geschwindigkeitsmessung mit 1000 Durchläufen

Um die Performance des Emulators bezüglich des Benchmarks zu testen, wird die Zeit gemessen, die eine einzelne Fehlerinjektion vom Zustand des Emulators laden bis hin zum Ergebnis der Ausführung braucht. An vier verschiedenen Zeitpunkten in der Fehlerinjektion wird die Zeit gemessen. Dafür wurde das generische Fehlerinjektionsexperiment des FAIL*-Frameworks angepasst und an vier Stellen wurde Code hinzugefügt, der den jetzigen Zeitpunkt speichert. Diese gemessenen Zeitpunkte werden dann zusammen mit den Ergebnissen der Fehlerinjektion zurück an den Server gesendet, der sie anschließend in der Datenbank abspeichert. In Abbildung 4.1 sind schematisch die Schritte die der FAIL*-Client bei einer Fehlerinjektion durchführt zu sehen und die Zeitpunkte, an denen gemessen wird. Der erste Zeitpunk wird am Anfang der Fehlerinjektion gemessen, bevor der Emulatorzustand geladen worden ist oder eine Instruktion vom Emulator für diese Fehlerinjektion ausgeführt worden war. Der zweite Zeitpunkt wird gemessen, bevor der Emulator bis zur Injektionsinstruktion ausgeführt wird. Zu diesem Zeitpunkt hat der Emulator in dieser Fehlerinjektion bisher nur seinen vorher spezifizierten Zustand geladen. Der nächste Zeitpunkt, der dritte, wird gemessen nachdem der Emulator die Instruktion an der injiziert werden soll erreicht hat und der Fehler in das System injiziert worden ist. Der vierte und letzte Zeitpunkt an dem gemessen wird, befindet sich am Ende der Fehlerinjektion. Nach dem einer der vom Experiment definierten Auslöser ausgelöst wurde, sei es durch ein Time-out oder dadurch, dass das Programm erfolgreich durchgelaufen ist, wird das letzte Mal die Zeit gemessen.

Durch diese vier Zeitpunkte ist es möglich die Fehlerinjektion in vier Abschnitte einzuteilen. Zum einen wurde die Zeit gemessen, die der Emulator benötigt um einen vorher gespeicherten Zustand zu laden, die Setup-Zeit. Desweiteren wurde der Zeitraum gemessen, den der Emulator braucht um die benötigte Anzahl an Instruktionen bis zum Injektionspunkt und das Injizieren von Fehlern auszuführen. Zusätzlich konnte die Zeit gemessen, die der Emulator nach der Fehlerinjektion benötigt um die restlichen Instruktionen bis zum Auslösen eines Auslösers auszuführen. Und zu guter Letzt wird noch die Zeit gemessen, die das FAIL*-Framework für die gesamte Durchführung einer einzelne Fehlerinjektion benötigt. Zusätzlich zu den Zeiten wird noch die Anzahl der Instruktionen, die vom Emulator vor und nach der Fehlerinjektion ausgeführt worden sind, gezählt. Dies ermöglicht es die Anzahl der Instruktionen des Emulators pro Sekunde auszurechnen.

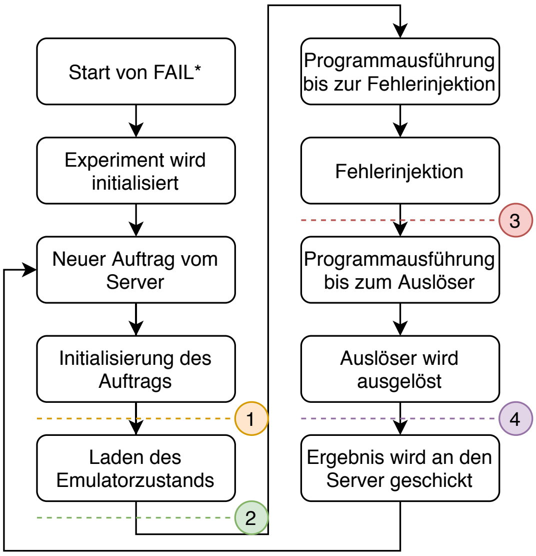


Abbildung 4.1 – Schematische Darstellung der Fehlerinjektion mit den Messzeitpunkten für die Performance-Analyse

Die Fehlerinjektion die auf den Benchmarks durchgeführt werden soll, injiziert in die Register und in den Speicher. Die Fehler die injiziert werden sind einzelne Bit-Flips. Zusätzlich wurde der Time-out für beide Emulatoren auf 100 000 Instruktionen gesetzt.

4.1.2 Ergebnisse

Wie in Abschnitt 4.1.1 beschrieben wurde die Performance-Evaluation mithilfe des Schleifenzählerbenchmarks durchgeführt, wobei die Benchmarks mit verschiedenen Anzahlen an Schleifendurchläufen ausgeführt worden sind. Die Benchmarks wurden mit jeweils 10, 100, 1000 und 10000 Schleifendurchläufen durchlaufen. In Tabelle 4.1 sind die Anzahl der ausgeführten Instruktionen in der goldenen Ausführung des Benchmarkprogramms mit unterschiedlicher Anzahl von Schleifendurchläufen zu sehen. Die goldene Ausführung ist die Ausführung des Programms, ohne Fehler in das Program zu injizieren. Zu erkennen ist, dass bei beiden Architekturen die Instruktionsanzahl in der goldenen Ausführung nahezu identisch sind. Beide Architekturen brauchen also ungefähr die selbe Anzahl an Instruktionen um das Benchmark erfolgreich auszuführen. Diese ähnliche Menge an Instruktionen kommt daher, dass bei beiden kompilierten Programmen die Schleife nur aus zwei Instruktionen besteht. Diese beiden Instruktionen sind bei beiden Architekturen eine Subtraktion und ein Sprung. Offenbar wurde der Quellcode für beide Architekturen ähnlich optimiert. Dies ist ein Vorteil, da durch die ähnlichen Anzahlen an Instruktionen, die kompilierten Programme und damit auch die Architekturen besser vergleichbar sind. In Tabelle 4.1 ist außerdem zu sehen, dass RISC-V 6 Instruktionen und Bochs 4 Instruktionen für die Ausführung von 10 Schleifendurchläufen benötigt. Der Grund für diese geringe Anzahl an Instruktionen ist der Optimierungslevel *01*, der beim Kompilieren der Benchmarks verwendet worden ist. Dieser Optimierungslevel hat die Schleife und die Vergleiche der Variablen rausoptimiert, sodass nur die Aufrufe der Funktionen *start_trace()* und *stop_trace()* im Trace der goldenen Ausführung verzeichnet wurden.

Schleifendurchläufe	RISC-V	Bochs
10	6	4
100	207	205
1000	2007	2005
10000	20008	20005

Tabelle 4.1 – Anzahl ausgeführter Instruktionen im goldenen Run, bei verschiedener Anzahl Schleifendurchläufe und verschiedenen Emulatoren

Obwohl beide Architekturen die beinahe gleiche Anzahl an Instruktionen in ihrer goldenen Ausführung haben, sind bei der Ausführung einer Fehlerinjektionskampagne klare Unterschiede in der Ausführungszeit beider Emulatoren sichtbar. In Abbildung 4.2 sind die Ausführungszeiten der beiden Emulatoren, bei verschiedenen Schleifendurchläufen zu sehen. Die Buchstaben stehen für den Emulatoren und die Zahl steht für die Anzahl an Schleifendurchläufen, die für diesen Benchmark benutzt wurde. Zum Beispiel steht *R1000* für den Benchmark mit 1000 Schleifendurchläufen, welcher vom RISC-V-Emulator ausgeführt wurde. Die Ausführungszeiten in Abbildung 4.2 sind die Zeiten einer kompletten Injektion.

Zu beachten ist, dass aus den Datensätzen für den Bochs-Emulator mit der Schleifendurchlaufsanzahl von 100, 1000 und 10000 der oberste, beziehungsweise die beiden obersten Datenpunkte entfernt worden sind, da diese Daten sehr weit abseits von den anderen Werten lagen. Diese Datenpunkte liegen alle bei über 13 000 ms. Ohne diese Datenpunkte liegt das Maximum bei nur ungefähr 360 ms. Die Ursache dieser Ausreißer kann nicht genau festgestellt werden, aber da sie nur vereinzelt und sehr weit ab von den anderen Werten liegen, werden sie in den nächsten Tabellen und Grafiken nicht auftauchen.

Aus dem Boxplot in Abbildung 4.2 ist zu sehen, dass bis einschließlich den Benchmarks mit 1000 Schleifendurchläufen, der RISC-V-Emulator durchschnittlich die Fehlerinjektionen schneller durchführt als Bochs. Erst ab 10 000 Schleifendurchläufen wird Bochs im Durchschnitt schneller als der RISC-V-Emulator. Dies lässt darauf schließen, dass Bochs Instruktionen schneller ausführt als der RISC-V-Emulator, da, je mehr Instruktionen ausgeführt werden müssen, desto schneller wird Bochs im Vergleich zum RISC-V-Emulator. Zu sehen ist auch, dass die Standardabweichung beim RISC-V-Emulator ähnlich groß ist wie der Mittelwert. Dass lässt darauf schließen, dass die unterschiedlichen, einzelnen Programmausführungen im RISC-V-Emulators sehr unterschiedliche Laufzeiten besitzen. Das 75-Quantil beim RISC-V-Emulator beim Benchmark mit 10000 Schleifendurchläufen ist ungefähr 117 % größer als der Mittelwert. Desweiteren ist beim Bochs-Emulator die Standardabweichung viel geringer als beim RISC-V-Emulator. Zusätzlich ist die Standardabweichung im Vergleich zum Mittelwert auch kleiner. Dies lässt darauf schließen, dass ein Großteil der einzelnen Ausführungen des Programms in Bochs eine ähnliche Laufzeit haben. Der einzige Teil der Programmausführung, der sich signifikant über die einzelnen Programmausführungen ändern kann, ist die Anzahl der Instruktionen die ausgeführt werden, da die Anzahl der ausgeführten Instruktionen stark vom Ergebnis der Fehlerinjektion abhängt. Bei einem Time-out ist die Anzahl der ausgeführten Instruktionen höher als bei einer erfolgreichen Ausführung. Daraus lässt sich schließen, dass die Ausführung der Instruktionen eine große Auswirkung auf die Gesamtlaufzeit der Fehlerinjektion im RISC-V-Emulator hat. Da beim Bochs-Emulator die Laufzeiten der unterschiedlichen Ausführung nah beieinander liegen, das 75-Quantil des Benchmarks mit 10000 Schleifendurchläufen ist nur ungefähr 2 % größer als der Mittelwert, lässt sich daraus schließen, dass die Anzahl der Instruktionen keine große Auswirkung auf die Laufzeit des Programms hat.

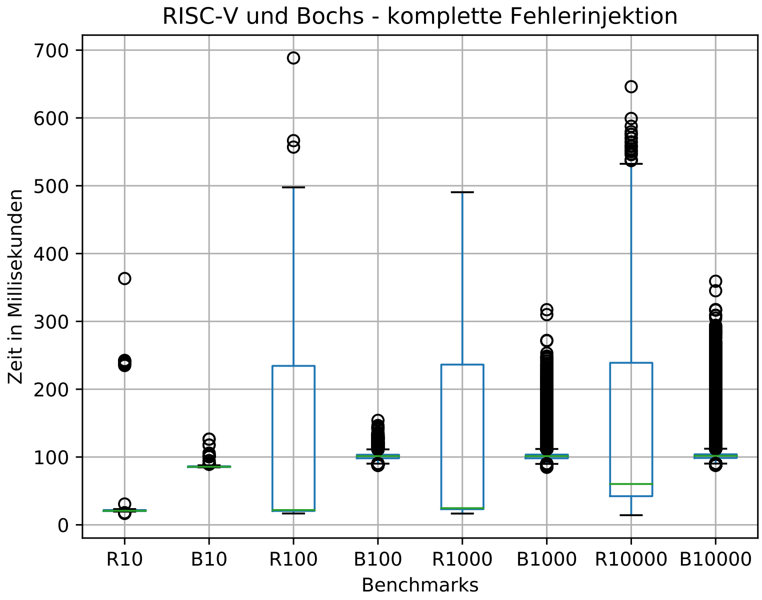


Abbildung 4.2 – Zu sehen ist die Zeit in Millisekunden, die die beiden Emulatoren, das B steht für Bochs und das R für RISC-V, benötigen, um eine einzelne Fehlerinjektion in ihrer Gänze durchzuführen. Die Zahlen in den Tabelleüberschriften stehen für die Anzahl an Schleifendurchläufen die im jeweiligen Benchmark durchlaufen werden.

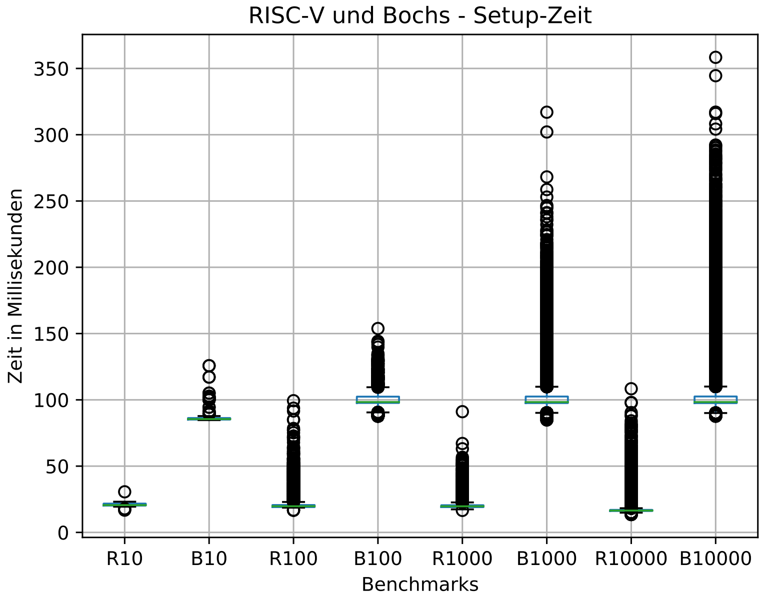


Abbildung 4.3 – Zu sehen ist die Zeit in Millisekunden, die die beiden Emulatoren für die Setup-Zeit der Fehlerinjektion benötigen.

In Abbildung 4.3 ist nur die Setup-Zeit der Benchmarks abgebildet. Zu sehen ist, dass die Setup-Zeit des Bochs-Emulators ungefähr fünfmal so lang ist, wie die Setup-Zeit des RISC-V-Emulators. Der einzige, zwischen den Architekturen unterschiedliche, Schritt, der beim Setup ausgeführt wird, ist das Laden des Zustands des Emulators. Folglich ist der Ladevorgang die Quelle des Unterschieds in der Setup-Zeit zwischen den einzelnen Emulatoren. Dies bedeutet, dass das Laden des Zustands des Bochs-Emulator signifikant länger braucht, als das Laden des Zustands des RISC-V-Emulators. Dies bestätigt auch, dass Bochs schneller Instruktionen ausführen kann als der RISC-V-Emulator. Zu beachten ist auch, dass die Setup-Zeit bei beiden Emulatoren einen kleinen Quartilsabstand haben. Das kommt daher, dass bei der Durchführung des Setups der Fehlerinjektion es keine Unterschiede zwischen verschiedenen Fehlerinjektionen gibt, da alle Fehlerinjektion in der generischen Fehlerinjektionskampagne den gleichen Zustand laden und dadurch auch die gleichen Schritte durchführen. Wieso die Setup-Zeit des RISC-V-Emulators besser wird, je mehr Schleifen durchlaufen werden und damit auch je mehr Instruktionen ausgeführt werden, kann ich an dieser Stellen nicht genau erklären, da die Setup-Zeit unabhängig von der Anzahl der durchgeführten Instruktionen sein sollte. Woher die große Anzahl an Ausreißern kommt kann ich an dieser Stelle auch nicht erklären, da bei jedem Setup der gleiche Zustand geladen wird. Wahrscheinlich haben diese beiden Punkte den selben Grund.

In Abbildung 4.4 und in Abbildung 4.5 sind die Instruktionen pro Millisekunde zu sehen, die beide Emulatoren jeweils bei der Ausführung der Benchmarks bis zur Fehlerinjektion, beziehungsweise bis zum Auslösen des Auslösers, ausführen.

Besonders deutlich ist zu sehen, dass der Bochs-Emulator um ein Vielfaches mehr Instruktionen pro Millisekunde ausführen kann als der RISC-V-Emulator. Im Besten Fall kann der Bochs-Emulator ungefähr 77-mal so viele Instruktionen pro Millisekunde ausführen.

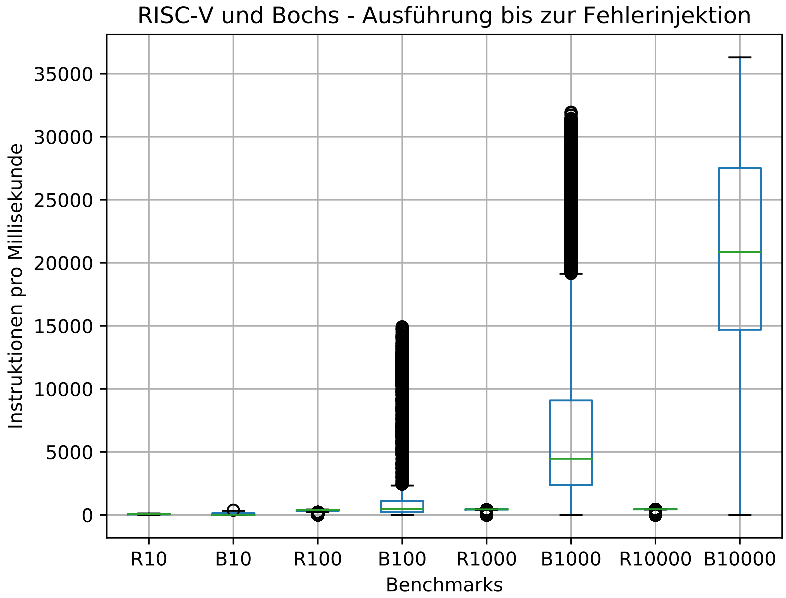


Abbildung 4.4 – Zu sehen sind die Instruktionen pro Millisekunde die die Emulatoren während der Ausführung bis zur Fehlerinjektion ausführen.(Abschnitt zwischen den Punkten 2 und 3 in Abbildung 4.1)

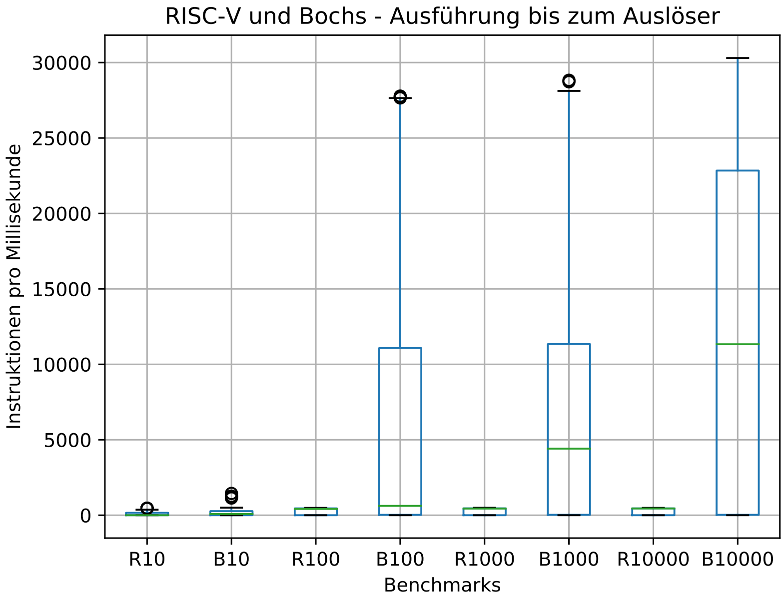


Abbildung 4.5 – Zu sehen sind die Instruktionen pro Millisekunde die die Emulatoren während der Ausführung bis zum Auslöser ausführen. (Abschnitt zwischen den Punkten 3 und 4 in Abbildung 4.1)

Zusätzlich kann gesehen werde, dass der Durchschnitt der Instruktionen pro Millisekunde bei beiden Emulatoren mit der Zunahme der Schleifendurchläufe auch zunimmt. Dies ist der Fall, da zusätzlich zum reinen Ausführen der Instruktionen noch der Overhead vom FAIL*-Framework dazukommt. Dieser Overhead fällt aber, je mehr Instruktionen im Durchschnitt ausgeführt werden, immer weniger ins Gewicht, so dass bei längeren Ausführungen, mehr Instruktionen pro Millisekunde ausgeführt werden können. Dies führt dazu, dass die durchschnittlich höchste Anzahl an Instruktionen pro Millisekunde bei Benchmark mit 10 000 Schleifendurchläufen ausgeführt werden.

Interessanterweise ist das Maximum der Instruktionen pro Millisekunde des Bochs-Emulators nicht in der Ausführung bis zum Auslöser zu finden, sondern in der Ausführung bis zur Fehlerinjektion. Dies ist interessant, da dies gegen die vorherige Annahme geht, dass mehr Instruktionen auch einen Anstieg der ausgeführten Instruktionen pro Millisekunde bedeutet, da die obere Schranke in den ausgeführten Instruktion in der Ausführung bis zur Fehlerinjektion die Anzahl der Instruktionen in goldenen Ausführung ist, während bei der Ausführung bis zum Auslöser die Anzahl der Instruktionen durch den Wert des Time-outs beschränkt ist. Im Falle der Ausführung mit 10 000 Schleifendurchläufen läge die obere Schranke des Ausführen bis zu Fehlerinjektion bei ungefähr 20 000 Instruktionen und die obere Schranke des Ausführen bis zum Auslöser bei 100 000 Instruktionen. Folglich sollte die Ausführung bis zum Auslöser ein höheres Maximum haben. Dies ist aber nicht der Fall. Wahrscheinlich ist der Overhead, der bei der Ausführung bis zum Auslöser auftritt, größer als der Overhead, der Ausführung bis zur Fehlerinjektion auftritt. Dies würde den Unterschied in den Maxima erklären. Dies ist aber nicht im RISC-V-Emulator zu beobachten. Dort findet man das Maximum bei der Ausführung bis zum Auslöser.

Zusätzlich ist im Vergleich zu sehen, dass das 25-Quantil bei der Ausführung bis zum Auslöser sehr nahe bei null Instruktionen pro Millisekunde liegt. Dies lässt darauf schließen, dass ein Großteil der

Ausführung direkt nach der Fehlerinjektion, zum Beispiel durch einen Trap, beendet werden. Dadurch würden nur wenige Instruktionen ausgeführt werden, was zu einer geringeren Geschwindigkeit führt.

Die großen Quartilsabstände bei der Ausführung bis zum Auslöser kommen daher, dass die Anzahl der Instruktionen die nach der Fehlerinjektion ausgeführt werden, stark schwankt und dadurch auch die Instruktionen pro Millisekunde stark schwanken.

In Abbildung 4.6 ist der Schnittpunkt der Gesamtlaufzeit vom RISC-V-Emulator und dem Bochs-Emulator zu sehen. Für diesen Graph wurde die beste Setup-Zeit beider Emulatoren, 13.46 ms beim RISC-V-Emulator und 84.71 ms beim Bochs-Emulator, und die beste Anzahl an Instruktionen pro Millisekunde, 492.55 Instruktionen pro Millisekunde beim RISC-V-Emulator und 36 300.61 Instruktionen pro Millisekunde beim Bochs-Emulator, genommen und mit den Instruktionen als Laufvariable gezeichnet.

Zu sehen ist, dass bei ungefähr 35 500 Instruktionen der Bochs-Emulator den Unterschied in der Setup-Zeit wettmachen kann und anschließend schneller als der RISC-V-Emulator läuft. Bis zu diesen 35 500 Instruktionen ist noch der RISC-V-Emulator aufgrund seiner besseren Setup-Zeit schneller. Der Bochs-Emulator schafft es den Nachteil der aus der längeren Setup-Zeit entstanden ist, in einer Millisekunde wieder einzuholen.

4.1.3 Zusammenfassung

Bei den durchgeführten Performance-Tests konnte gesehen werden, dass der RISC-V-Emulator bis zu einer bestimmten Anzahl an Instruktionen, ungefähr 36 000, schneller als der Bochs-Emulator läuft,

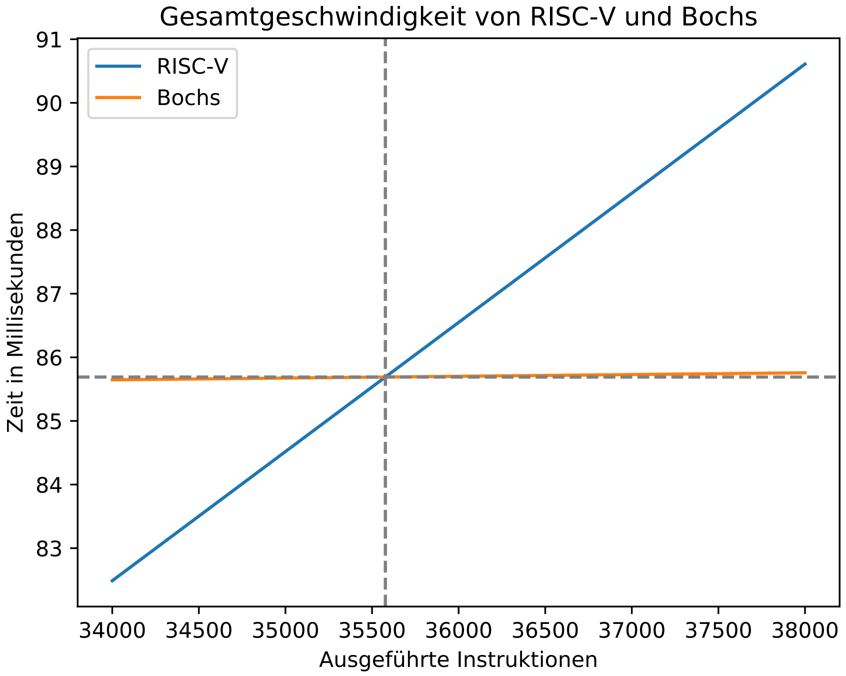


Abbildung 4.6 – Schnittpunkt der Gesamtlaufzeit vom RISC-V-Emulator und vom Bochs-Emulator

da die Setup-Zeit des Bochs-Emulators größer ist als die des RISC-V-Emulators. Ab diesem Punkt ist die Geschwindigkeit des Bochs-Emulators vom RISC-V-Emulators nicht mehr einzuholen, da der Bochs-Emulator ungefähr 70-mal schneller ist. Solange also kurze Programme im RISC-V-Emulator ausgeführt werden, läuft er performanter als der Bochs-Emulator.

4.2 Fehleranfälligkeit

In diesem Kapitel wird die Fehleranfälligkeit der RISC-V-Architektur mit der der x86-Architektur verglichen. Dazu wurden Fehler in Benchmarkprogramme injiziert, die im RISC-V-Emulator und im Bochs-Emulator laufen, und anschließend werden die Ergebnisse der Fehlerinjektion präsentiert und miteinander verglichen.

4.2.1 Versuchsaufbau

Der grundlegende Versuchsaufbau für die Tests der Fehleranfälligkeit ist der gleiche wie bei der Analyse der Performance.

Für diese Versuche wird das generische Fehlerinjektionsexperiment des FAIL*-Frameworks benutzt. Dieses Fehlerinjektionsexperiment kann mithilfe des FAIL*-Frameworks die in Tabelle 4.2 gelisteten Fehlerklassen detektieren.

Die Fehlerklasse *Erfolgreich* zeigt an, dass die Ausführung des Programms das erwartete Ergebnis erzielt. Dazu sind in den Benchmarks Vergleiche vorhanden, die das erwartete Ergebnis mit dem tatsächlichen Ergebnis vergleichen. Falls das tatsächliche Ergebnis dem erwarteten Ergebnis entspricht wird, wie in schon in Listing 4.1 zu sehen war, die *ok_marker()*-Funktion aufgerufen. Das RISC-V-Modell ruft bei jeder Ausführung einer Instruktion das FAIL*-Framework auf und teilt ihm mit, welche Instruktion aufgerufen wurde und welchen Wert der Instruktionszeiger hat. Diese Daten vergleicht das FAIL*-Framework mit den zurzeit aktiven Auslösern und löst den entsprechenden Auslöser aus. Im Falle der Instruktion der *ok_marker()*-Funktion wird diese Fehlerinjektion der Fehlerklasse *Erfolgreich* zugeordnet.

Falls das tatsächliche Ergebnis aber vom erwarteten Ergebnis abweicht, dann wird die *fail_marker()*-Funktion aufgerufen. Der zu dieser Funktion zugehörige Auslöser ordnet diese Fehlerinjektion der Fehlerklasse *Fehler* zu, die anzeigt, dass das tatsächliche Ergebnis vom erwarteten Ergebnis abweicht.

Die Fehlerklasse *Time-out* zeigt an, dass ein Time-out bei der Fehlerinjektion aufgetreten ist. Die Länge des Time-outs wird zu Beginn der kompletten Fehlerinjektionskampagne vom Client festgelegt und wurde für diesen Versuch zur Fehleranfälligkeit auf 100 000 ausgeführte Instruktionen gesetzt. Die interne *SailTimer*-Klasse verwaltet für die Sail-Emulatoren die Time-outs. Jede vom

Fehlerklassen
Erfolgreich
Fehler
Time-out
Trap
Textsegment
Außerhalb Speicher

Tabelle 4.2 – Die Fehlerklassen die FAIL* detektieren soll.

RISC-V-Emulator ausgeführte Instruktion führt dazu, dass der interne Zähler der *SailTimer*-Klasse inkrementiert wird und überprüft wird, ob einer der gesetzten Timer ausgelöst werden soll. Falls der Timer für die Time-out-Instruktionen erreicht wurde, wird er ausgelöst und diese Fehlerinjektion bekommt die Fehlerklasse *Time-out* zugeteilt.

Die Fehlerklasse *Trap* zeigt an, dass die Fehlerinjektion zu einem Trap geführt hat. Um dies zu detektieren sind im RISC-V-Modell an den Stellen im Code, an denen ein Trap ausgelöst wird, Aufrufe, die dem FAIL*-Framework mitteilen, dass ein Trap ausgelöst worden ist und welcher Trap ausgelöst worden ist.

Die Fehlerklasse *Textsegment* zeigt an, ob ein Speicher-Schreibzugriff auf das Textsegment erfolgt ist. Dazu wird das kompilierte, ausführbare Programm analysiert und ein Auslöser wird für den Speicherraum des Textsegments erstellt. Im RISC-V-Modell befinden sich, ähnlich wie für die *Trap*-Fehlerklasse, an den Stellen an denen ein Speicherzugriff erfolgt, Aufrufe des FAIL*-Frameworks, die ihm mitteilen, an welchen Speicheradressen gerade geschrieben beziehungsweise gelesen wurde. Falls einer dieser Speicher-Schreibzugriff in den Speicherraum des Textsegments fällt, dann wird der entsprechende Auslöser ausgelöst und diese Ausführung bekommt die Fehlerklasse *Textsegment* zugeteilt.

Die Fehlerklasse *Außerhalb Speicher* entspricht der Fehlerklasse *Textsegment* mit der einzigen Ausnahme, dass anstatt einen Schreibzugriff in das Textsegment anzuzeigen, zeigt diese Fehlerklasse den Schreibzugriff außerhalb des Speicherraums des Programms an. Falls dieser Schreibzugriff detektiert wird, bekommt diese Ausführung die Fehlerklasse *Außerhalb Speicher* zugeteilt.

Bei den Versuchen für die Fehleranfälligkeit wird nicht jede mögliche Fehlerinjektion durchgeführt. Stattdessen wird das Def-Use-Pruning-Verfahren benutzt um die Anzahl der injizierten Fehler zu reduzieren [Sch+15]. Die grundlegende Idee ist, dass alle Fehlerinjektionspunkte zwischen dem Schreiben von Daten, sei es in die Register oder in den Speicher, und dem Lesen selbiger Daten, äquivalent sind. Es ist egal wann nach dem Schreiben von Daten in diese Daten Fehler injiziert werden, solange dies vor dem Lesen der Daten geschieht. Diese äquivalenten Fehlerinjektionspunkte werden zu Äquivalenzklassen zusammengefasst und für jede Äquivalenzklasse muss nur eine Fehlerinjektion durchgeführt werden. Dadurch wird die Anzahl der auszuführenden Fehlerinjektionen verringert.

Zusätzlich zu den Benchmarks aus Abschnitt 4.1.1 wurden die Fehler auch in weitere Benchmarkprogramme injiziert, um die Fehleranfälligkeit der beiden Architekturen miteinander zu vergleichen. In Tabelle 4.3 ist eine Auflistung der verwendeten Benchmarks zu sehen. Neben den schon aus Abschnitt 4.1.1 bekannten Schleifenzählerbenchmarks 10 bis 10 000 sind noch zwei Qsort-

Benchmarks
Schleifenzähler10(10)
Schleifenzähler100(100)
Schleifenzähler1000(1000)
Schleifenzähler10000(10000)
Qsort(rekursiv)(QR)
Qsort(iterativ)(QI)
BerechnungInline(BI)
BerechnungNoInline(BN)
Fibonacci(Fib)
Mix(Mix)

Tabelle 4.3 – Die verwendeten Benchmarks für die Analyse der Fehleranfälligkeit. In den Klammern steht der Code der in den Diagrammen das Benchmark identifiziert.

Funktion, die ein 3-dimensionales Array sortieren, einmal rekursiv(Qsort(rekursiv)) und einmal iterativ(Qsort(iterativ)), zwei Berechnungsprogramme, die eine Variable mithilfe von Funktionen 1000-mal multiplizieren, addieren, subtrahieren und dividieren, einmal in der normalen Variante(BerechnungInline) und einmal in der Variante(BerechnungNoInline), bei der die Funktionen mit dem C-Attribut "*noinline*" versehen sind, ein rekursives Fibonacci-Programm(Fibonacci), welches die zehnte Fibonacci-Zahl errechnet, und zu guter Letzt einem Programm(Mix), welches mit einem Array aus Werten rechnet.

4.2.2 Ergebnisse

In Tabelle 4.4 sind die Anzahl der Schreib- und Lesezugriffe auf den Speicher des Emulators in einer normalen Ausführung der verschiedenen, verwendeten Benchmarks aufgelistet. Zusätzlich steht dort die Anzahl der durchgeführten Fehlerinjektionen pro Benchmark.

Da die Schleifenzählerbenchmarks eine ähnliche Anzahl an Instruktionen, vergleich Tabelle 4.1, haben und die Anzahl der Lese- und Schreibzugriffe auch ähnlich groß sind, ist es nicht verwunderlich, dass die Anzahl der Fehlerinjektionen beziehungsweise die Anzahl der Äquivalenzklassen der Fehlerpunkte auch ähnlich groß sind.

Interessanter wird es bei den Qsort-Benchmarks. Bei ihnen hat der für den Bochs-Emulator kompilierte Benchmark ungefähr doppelt so viele Schreib- und Lesezugriffe wie der für den RISC-V-Emulator kompilierte Benchmark. Dies lässt darauf schließen, dass die RISC-V-Architektur den Zugriff auf die Arrays mit weniger Speicherzugriffen ermöglicht. Zusätzlich ist zu sehen, dass in die für den Bochs-Emulator kompilierten Qsort-Benchmarks ungefähr 30 % mehr Fehler injiziert werden als in die im RISC-V-Emulator ausgeführten Benchmarks.

Generell ist zu sehen, dass in die Benchmarks, die im Bochs-Emulator ausgeführt werden, mit Ausnahme der Schleifenzählerbenchmarks und dem Mix-Benchmark, mehr Fehler injiziert werden und folglich auch mehr Äquivalenzklassen gefunden wurden. Dies lässt sich auf Unterschiede zwischen den beiden Architekturen zurückführen.

Interessanterweise ist zu sehen, dass die Anzahl der Schreibzugriffe beim Berechnungsbenchmark in der *NoInline*-Variante im Vergleich zur RISC-V-Variante dieses Benchmarks und zur *Inline*-Variante des Berechnungsbenchmarks, sehr groß ist. Anscheinend werden Funktionsaufrufe mit dem C-

	RISC-V			Bochs		
	Schreiben	Lesen	Injektionen	Schreiben	Lesen	Injektionen
Schleifenzähler10	1	0	304	2	1	208
Schleifenzähler100	1	0	13136	2	1	13040
Schleifenzähler1000	1	0	128336	2	1	128240
Schleifenzähler10000	1	0	1280400	2	1	1280240
Qsort(rekursiv)	2675	3050	913296	4120	5151	1270128
Qsort(iterativ)	2023	2806	895248	3852	4972	1264112
BerechnungInline	1	0	609072	2	1	1089392
BerechnungNoInline	1	0	2082512	20022	20021	4933200
Fibonacci	532	531	219632	710	709	236528
Mix	3	9	8400	4	10	5584

Tabelle 4.4 – Die Anzahl der Lese- und Schreibzugriff auf den Speicher des Emulators und die Anzahl der durchgeführten Fehlerinjektionen, beziehungsweise die Anzahl der Äquivalenzklassen, des RISC-V-Emulator und des Bochs-Emulator.

Attribut "*noinline*" bei dem Kompilieren für den Bochs-Emulator mit Speicherzugriffen übersetzt. Beim Kompilieren für den RISC-V-Emulator scheint dies nicht der Fall zu sein.

In 4.7 sind zuerst nur die Schleifenzählerbenchmarks zu sehen, die auch schon in Abschnitt 4.1 verwendet worden sind.

Beim Vergleich zwischen den einzelnen Benchmarks ist zu sehen, dass mit Ausnahme vom Benchmark mit einer Schleifendurchlaufsanzahl von 10, alle anderen Benchmarks ihr Verhältnis der Fehlerklassen beibehalten. Der Benchmark mit 10 Schleifendurchläufen hat nicht das gleiche Verhältnis wie die anderen Benchmarks, da bei ihm die Schleife beim Kompilieren rausoptimiert worden ist. Die anderen Benchmarks behalten das Fehlerklassenverhältnis bei, da sich zwischen den einzelnen Benchmarks nur die Anzahl der Schleifendurchläufe geändert hat.

Beim Betrachten der Fehlerklassen fällt auf, dass beide Architekturen eine hohe Anzahl an erfolgreich durchgeführten Ausführung haben und eine geringe Anzahl an Ausführungen bei der das errechnete Ergebnis vom erwarteten Ergebnis abweicht. Dies lässt erstmal vermuten, dass beide Architekturen gut gegen stille Datenverfälschung, dem Verfälschen von Daten, ohne dass das System mitbekommt, dass die Daten verändert wurden, geschützt sind, da die Anzahl der fehlerhaften Ausführungen sehr gering ist.

Beide Architekturen haben eine relativ ähnliche Anzahl an Ausführungen die eine Trap auslösen.

Zusätzlich ist noch zu sehen, dass die RISC-V-Benchmarks fast keine Schreibzugriffe ausserhalb des Speichers haben, wohingegen die Bochs-Benchmarks eine nicht kleine Menge an Schreibzugriffen ausserhalb des Speichers haben.

Noch anzumerken ist, dass bei der Fehlerinjektion in diese Benchmarks kein einziger Schreibzugriff in das Textsegment erfolgt ist.

In Abbildung 4.8 sind die restlichen Benchmarks aus Tabelle 4.3 zu finden.

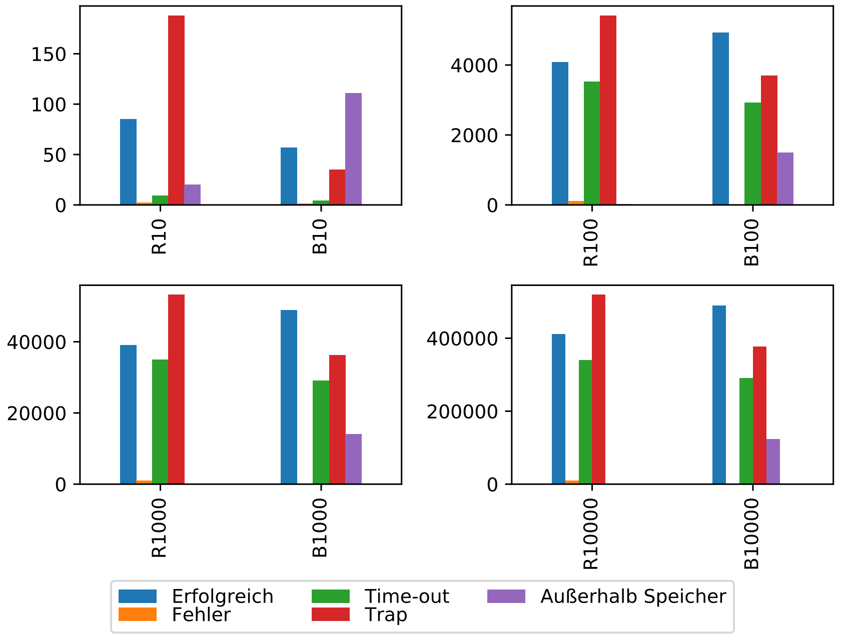


Abbildung 4.7 – Die verschiedenen Fehlerklassen der Schleifenzählerbenchmarks. Der Buchstabe R vor dem Code des Benchmarks(vergleich Tabelle 4.3) steht für den RISC-V-Emulator und das B steht für den Bochs-Emulator.

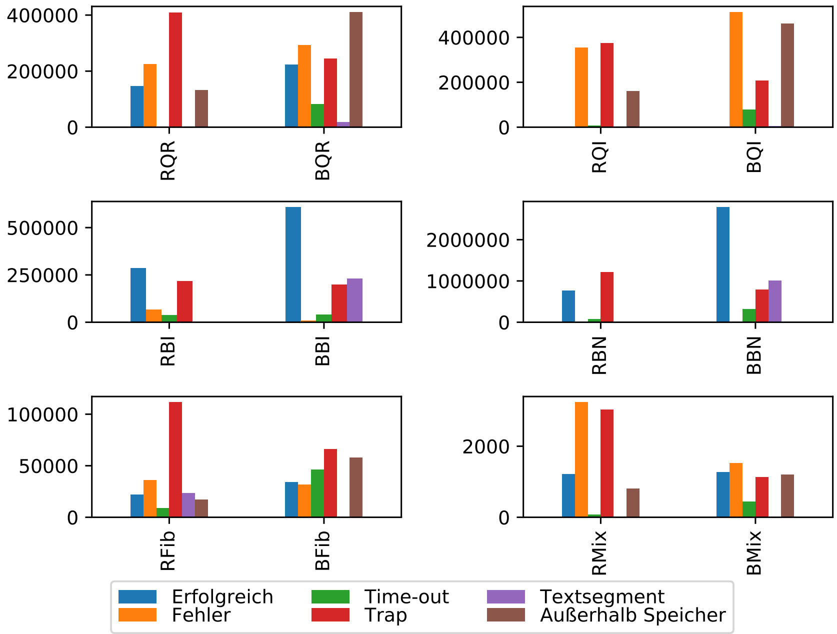


Abbildung 4.8 – Die verschiedenen Fehlerklassen der Benchmarks aus Tabelle 4.3. Der Buchstabe R vor dem Code des Benchmarks steht für den RISC-V-Emulator und das B steht für den Bochs-Emulator.

Bei den rekursiven Qsort-Benchmarks ist zu sehen, dass die Ausführung im RISC-V-Emulator eine große Anzahl von Traps ausgelöst haben. Bei der Ausführung im Bochs-Emulator wurden nur ungefähr die Hälfte der Traps, die im RISC-V-Emulator ausgelöst wurden, ausgelöst. Bei der Ausführung im Bochs-Emulator ist stattdessen die Anzahl der Schreibzugriffe ausserhalb des Speichers mehr als doppelt so groß.

Beim iterativen Qsort-Benchmark fällt auf, dass beide Architekturen eine sehr geringe Anzahl an erfolgreich durchgeführten Ausführungen haben, dies lässt vermuten, dass wahrscheinlich das Benchmark-Programm und das erwartete Ergebnis der Ausführung des Programms fehlerhaft waren. Der fehlerhafte Verleich mit dem erwarteten Ergebnis wirkt sich aber nur auf die Fehlerklassen der erfolgreichen und der fehlerhaften Ausführung aus, weshalb sich die anderen Klassen trotzdem noch vergleichen lassen. Wie schon beim rekursiven Qsort-Benchmark zu sehen, ist bei der Ausführung im Bochs-Emulator die Anzahl der Schreibzugriffe außerhalb des Speichers mehr als doppelt so groß wie bei der Ausführung im RISC-V-Emulator. An dieser Stelle ist aber zu beachten, dass die Anzahl der injizierten Fehler bei der Ausführung im Bochs-Emulator ungefähr 30 % größer ist als bei der Ausführung im RISC-V-Emulator.

Bei den Berechnungsbenchmarks ist interessanter Weise zu sehen, dass das Verhältnis der Fehlerklassen zwischen den beiden Varianten ungefähr gleich bleibt, obwohl die "*noinline*"-Variante viel mehr Speicherzugriffe hat. Nur die Anzahl der Time-outs ist minimal angestiegen. Bei der Ausführung mit dem RISC-V-Emulator kann gesehen werden, dass sich das Verhältnis der Fehlerklassen zwischen den beiden Varianten verändert. Bei der *Inline*-Variante sind mehr erfolgreiche Ausführungen, aber auch mehr fehlerhafte Ausführung, aber auch weniger ausgelöste Traps zu sehen. Beim direkten Vergleich zwischen den im RISC-V-Emulator und den im Bochs-Emulator ausgeführten Berechnungsbenchmarks kann gesehen werden, dass bei der Ausführung im Bochs-Emulator viel mehr Ausführung erfolgreich ausgeführt werden. Zu beachten ist aber, dass bei der Ausführung im Bochs-Emulator es viel mehr Fehlerinjektionen gab und diese keine Auswirkung auf das Ergebnis der Ausführung gehabt haben könnten. Wiederum ist bei diesen Benchmarks zu sehen, dass RISC-V keine Schreibzugriffe auf das Textsegment hat.

Bei dem Fibonacci-Benchmark fällt sofort auf, dass bei der Ausführung im RISC-V-Emulator viele Traps auftreten und die anderen Fehlerklassen in einer ähnlichen, aber geringen Anzahl auftreten. Bei der Ausführung im Bochs-Emulator kann gesehen werden, dass die Fehlerklassen, mit Ausnahme des Schreibens außerhalb des Textsegment, eine relativ ähnliche Anzahl haben und es keine besonders großen Anzahlen gibt.

Bei dem Mix-Benchmark fällt auf, dass bei der Ausführung im RISC-V-Emulator eine große Anzahl an Ausführungen ein fehlerhaftes Ergebnis haben oder einen Trap auslösen. Die Fehlerklassen des Bochs-Emulators sind dagegen gleichmäßiger verteilt, wieder mit der Ausnahme der Fehlerklasse des Schreibens in das Textsegment.

Beim Vergleichen der Fehlerklassen der Benchmarks fällt auf, dass bei der Ausführung im RISC-V-Emulator es oft nur sehr wenige Time-outs gibt. Dies ist aber nicht der Fall bei den Schleifenzählerbenchmarks. Anscheinend hängen die Fehlerklassen bei der Ausführung im RISC-V-Emulator stark vom Programm ab. Zusätzlich ist noch zu sehen, dass die Anzahl der Ausführung, die ein Trap erzeugen bei allen Benchmarks hoch. Der Grund dafür können die zwei zusätzlich zum RISC-V-Modell hinzugefügten Trap-Klassen sein, die auslösen, wenn der Emulator einen internen Fehler hat und ihn erkennt, und wenn die auszuführende Instruktion einen noch nicht implementierten Teil des Emulators aufrufen will. Diese beiden Trap-Klassen können die Anzahl der Ausführungen die einen Trap auslösen stark nach oben treiben.

4.2.3 Zusammenfassung

In diesem Kapitel wurde die Fehleranfälligkeit des RISC-V-Emulators mit der Fehleranfälligkeit des Bochs-Emulators verglichen. Dazu wurden Fehler in verschiedene Benchmarkprogramme, die in den Emulatoren liefen, injiziert und die einzelnen Fehlerklassen der beiden Emulatoren wurden miteinander verglichen. Zu sehen war, dass die Anzahl der unterschiedlichen Fehlerklassen stark vom Programm, in das injiziert wird , abhängen. Trotzdem konnte gesehen werden, dass bei der Ausführung im RISC-V-Emulator relativ viele Traps ausgelöst werden. Dies liegt aber sehr wahrscheinlich am Modell des Emulators, da dort zwei zusätzliche Traps hinzugefügt worden sind, die modellinterne Fehler markieren.

ZUSAMMENFASSUNG

5

In dieser Arbeit wurde ein generisches Interface für das FAIL*-Framework entwickelt, welches es ermöglicht verschiedene, von Sail generierte, Emulatoren an das FAIL*-Framework anzubinden, ohne das generische Interface ändern zu müssen. Dadurch ist es jetzt möglich in, aus Sail generierten, Emulatoren Fehler zu injizieren und dadurch die Fehleranfälligkeit verschiedener Prozessorarchitekturen zu testen. Um diese Anbindung zu untersuchen, wurde der von Sail generierte RISC-V-Emulator mithilfe dieses generischen Interfaces an das FAIL*-Framework angebunden. Die im RISC-V-Emulator implementierte Prozessorarchitektur wurde anschließend mit der im Bochs-Emulator implementierten Prozessorarchitektur in den Punkten Performance und Fehleranfälligkeit verglichen. Es stellte sich heraus, dass bis zu einer Anzahl von ungefähr 36000 Instruktionen der RISC-V-Emulator, aufgrund der besseren Setup-Zeit, bei einer Fehlerinjektion performanter läuft als der Bochs-Emulator. Zusätzlich wurde die Fehleranfälligkeit der beiden Prozessorarchitekturen miteinander verglichen und es stellte sich heraus, dass die Fehleranfälligkeit zum größten Teil sehr ähnlich zwischen den beiden Architekturen sind, wobei bei manchen Programmen bestimmte Fehlerklassen besonders oft vertreten waren. Desweiteren hängt die Fehleranfälligkeit der Architekturen stark von dem ausgeführten Programm ab.

Da jetzt eine Möglichkeit für die Anbindung zwischen FAIL* und den von Sail generierten Emulatoren existiert wäre es spannend noch andere Architekturen an das FAIL*-Framework anzubinden und auf ihre Fehleranfälligkeit zu testen. Zum Beispiel könnte ein Modell für die x86-Architektur erstellt werden und die Fehleranfälligkeit des Modells könnte mit der Fehleranfälligkeit der in Bochs implementierten x86-Architektur verglichen werden. Vielleicht könnte dies interessante Aufschlüsse liefern.

ABBILDUNGSVERZEICHNIS

2.1 Die Fault-Kriterien und ihre zugehörigen Klassen und Kategorien (*entnommen aus [Avi+04]*). 5
2.2 Kosmische Strahlung, bei ihrem Eintreten in die Erdatmosphäre und der dabei entstehende Schauer an Sekundärteilchen (*entnommen aus [Cos]*). 8
2.3 Neutronen und Bor-10 kollidieren und erzeugen Sekundärstrahlung (*entnommen aus [Bau01]*). 8
2.4 FAIL*-Architektur: Der Kampagnen-Controller verteilt Parametersets an die einzelnen FAIL*-Instanzen. Diese steuern ihr Back-End mithilfe der Execution-Environment-Abstraction und senden nach der erfolgreichen Injektion und Ausführung ihr Ergebnis zurück an den Controller (*entnommen aus [Sch+15]*). 14
2.5 Die vier Phasen des Fault-Toleranz-Einschätzungskreislauf (*entnommen aus [Sch+15]*). 15
2.6 Eine Übersicht über Sail. Zu sehen sind die schon vorhandenen Befehlssatzarchitekturmodelle und die Generate die mithilfe von Sail aus den Modellen erzeugt werden können.(*entnommen aus [Saib]*). 17

4.1 Schematische Darstellung der Fehlerinjektion mit den Messzeitpunkten für die Performance-Analyse 30
4.2 Zu sehen ist die Zeit in Millisekunden, die die beiden Emulatoren, das B steht für Bochs und das R für RISC-V, benötigen, um eine einzelne Fehlerinjektion in ihrer Gänze durchzuführen. Die Zahlen in den Tabelleüberschriften stehen für die Anzahl an Schleifendurchläufen die im jeweiligen Benchmark durchlaufen werden. 32
4.3 Zu sehen ist die Zeit in Millisekunden, die die beiden Emulatoren für die Setup-Zeit der Fehlerinjektion benötigen. 32
4.4 Zu sehen sind die Instruktionen pro Millisekunde die die Emulatoren während der Ausführung bis zur Fehlerinjektion ausführen.(Abschnitt zwischen den Punkten 2 und 3 in Abbildung 4.1) 33
4.5 Zu sehen sind die Instruktionen pro Millisekunde die die Emulatoren während der Ausführung bis zum Auslöser ausführen.(Abschnitt zwischen den Punkten 3 und 4 in Abbildung 4.1) 34
4.6 Schnittpunkt der Gesamtlaufzeit vom RISC-V-Emulator und vom Bochs-Emulator . . 35
4.7 Die verschiedenen Fehlerklassen der Schleifenzählerbenchmarks. Der Buchstabe R vor dem Code des Benchmarks(vergleich Tabelle 4.3) steht für den RISC-V-Emulator und das B steht für den Bochs-Emulator. 39
4.8 Die verschiedenen Fehlerklassen der Benchmarks aus Tabelle 4.3. Der Buchstabe R vor dem Code des Benchmarks steht für den RISC-V-Emulator und das B steht für den Bochs-Emulator. 40

TABELLENVERZEICHNIS

2.1 Ratifizierte Erweiterungen der RISC-V-Architektur 16
2.2 Register des RISC-V-Basis-Integerinstruktions-Sets 17

4.1 Anzahl ausgeführter Instruktionen im goldenen Run, bei verschiedener Anzahl Schleifendurchläufe und verschiedenen Emulatoren 31
4.2 Die Fehlerklassen die FAIL* detektieren soll. 36
4.3 Die verwendeten Benchmarks für die Analyse der Fehleranfälligkeit. In den Klammern steht der Code der in den Diagrammen das Benchmark identifiziert. 37
4.4 Die Anzahl der Lese- und Schreibzugriff auf den Speicher des Emulators und die Anzahl der durchgeführten Fehlerinjektionen, beziehungsweise die Anzahl der Äquivalenzklassen, des RISC-V-Emulator und des Bochs-Emulator. 38

QUELLCODEVERZEICHNIS

4.1 Beispielcode für das Benchmarkprogramm für die Geschwindigkeitsmessung mit 1000 Durchläufen 29

LITERATUR

[Arl90] Jean Arlat. „Validation de la sûreté de fonctionnement par injection de fautes : méthode, mise en oeuvre, application“. Thèse de doctorat dirigée par Laprie, Jean-Claude Informatique Toulouse, INPT 1990. Diss. 1990, 190 f. URL: `http://www.theses.fr/1990INPT094H`.

[Arm+18] Alasdair Armstrong u. a. „Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics“. In: *Proc. Automated Reasoning Workshop*. Two-page abstract. Proceedings available at `https://www.cl.cam.ac.uk/events/arw2018/arw2018-proc.pdf`. Apr. 2018, S. 23–24.

[Boc] *bochs: The Open Source IA-32 Emulation Project (Home Page)*. `http://bochs.sourceforge.net/`. (Accessed on 05/07/2020).

[BSH75] D. Binder, E. C. Smith und A. B. Holman. „Satellite Anomalies from Galactic Cosmic Rays“. In: *IEEE Transactions on Nuclear Science* 22.6 (1975), S. 2675–2680. ISSN: 1558-1578. DOI: `10.1109/TNS.1975.4328188`.

[Cos] *Cosmic Rays | NCEI*. `https://www.ngdc.noaa.gov/stp/solar/cosmicrays.html`. (Accessed on 04/23/2020).

[Feh] *Fehler - LEO: Übersetzung im Englisch ⇔ Deutsch Wörterbuch*. `https://dict.leo.org/englisch-deutsch/Fehler`. Accessed: 2020-3-11.

[Gra+15] Kathryn E. Gray u. a. „An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors“. In: *Proceedings of the 48th International Symposium on Microarchitecture, , Waikiki, HI, USA*. Dez. 2015, S. 635–646. DOI: `10.1145/2830772.2830775`.

[GWA79] C. S. Guenzer, E. A. Wolicki und R. G. Allas. „Single Event Upset of Dynamic Rams by Neutrons and Protons“. In: *IEEE Transactions on Nuclear Science* 26.6 (1979), S. 5048–5052. ISSN: 1558-1578. DOI: `10.1109/TNS.1979.4330270`.

[KKA92] G. A. Kanawati, N. A. Kanawati und J. A. Abraham. „FERRARI: a tool for the validation of system dependability properties“. In: *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*. 1992, S. 336–344. DOI: `10.1109/FTCS.1992.243567`.

[Mad+94] Henrique Madeira u. a. „RIFLE: A general purpose pin-level fault injector“. In: Jan. 1994, S. 199–216. DOI: `10.1007/3-540-58426-9_132`.

[MW79] T. C. May und M. H. Woods. „Alpha-particle-induced soft errors in dynamic memories“. In: *IEEE Transactions on Electron Devices* 26.1 (1979), S. 2–9. ISSN: 1557-9646. DOI: `10.1109/T-ED.1979.19370`.

[Nas] *NASA Thesaurus Volume 1 - Hierarchical Listing With Definitions*. 2012.

[Risa] *FAQ - RISC-V International*. `https://riscv.org/faq/`. (Accessed on 04/21/2020).

[Risb] *https://web.archive.org/web/20180612144729/http://faculty.cs.niu.edu/~berezin/463/lec/05/risc03.html*. `https://web.archive.org/web/20180612144729/http://faculty.cs.niu.edu/~berezin/463/lec/05/risc03.html`. (Accessed on 04/21/2020).

[Risc] *ISA Specification - RISC-V International*. `https://riscv.org/specifications/isa-spec-pdf/`. (Accessed on 04/21/2020).

[Risd] *Privileged ISA Specification - RISC-V International*. `https://riscv.org/specifications/privileged-isa/`. (Accessed on 04/21/2020).

[Rise] *RISC-V Geneology*. `https://content.riscv.org/wp-content/uploads/2016/02/EECS-2016-6.pdf`. (Accessed on 04/21/2020).

[Risf] *RISC-V History - RISC-V International*. `https://riscv.org/risc-v-history/`. (Accessed on 04/21/2020).

[Risg] *Specifications - RISC-V International*. `https://riscv.org/specifications/`. (Accessed on 04/21/2020).

[Saia] *rems-project/sail-riscv: Sail RISC-V model*. `https://github.com/rems-project/sail-riscv`. (Accessed on 04/21/2020).

[Saib] *Sail*. `https://www.cl.cam.ac.uk/~pes20/sail/`. (Accessed on 04/21/2020).

[Saic] *sail/manual.pdf at sail2 · rems-project/sail*. `https://github.com/rems-project/sail/blob/sail2/manual.pdf`. (Accessed on 04/21/2020).

[Said] *sail/README.md at sail2 · rems-project/sail*. `https://github.com/rems-project/sail/blob/sail2/README.md`. (Accessed on 04/22/2020).

[SBK10] D. Skarin, R. Barbosa und J. Karlsson. „GOOFI-2: A tool for experimental dependability assessment“. In: *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 2010, S. 557–562. DOI: `10.1109/DSN.2010.5544265`.

[STB97] V. Sieh, O. Tschache und F. Balbach. „VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions“. In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 1997, S. 32–36. DOI: `10.1109/FTCS.1997.614074`.

[WA08] F. Wang und V. D. Agrawal. „Single Event Upset: An Embedded Tutorial“. In: *21st International Conference on VLSI Design (VLSID 2008)*. 2008, S. 429–434. DOI: `10.1109/VLSI.2008.28`.

[ZAV04] Haissam Ziade, Rafic Ayoubi und Raoul Velazco. „A Survey on Fault Injection Techniques“. In: *Int. Arab J. Inf. Technol.* 1.2 (2004), 171–186.

[Aid+01] J. Aidemark u. a. „GOOFI: generic object-oriented fault injection tool“. In: *2001 International Conference on Dependable Systems and Networks*. 2001, S. 83–88. DOI: `10.1109/DSN.2001.941394`.

[Arl+90] J. Arlat u. a. „Fault injection for dependability validation: a methodology and some applications“. In: *IEEE Transactions on Software Engineering* 16.2 (1990), S. 166–182. ISSN: 1939-3520. DOI: `10.1109/32.44380`.

[Avi+04] A. Avizienis u. a. „Basic concepts and taxonomy of dependable and secure computing“. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), S. 11–33. ISSN: 2160-9209. DOI: `10.1109/TDSC.2004.2`.

[Bau01] R. C. Baumann. „Soft errors in advanced semiconductor devices-part I: the three radiation sources“. In: *IEEE Transactions on Device and Materials Reliability* 1.1 (2001), S. 17–22. ISSN: 1558-2574. DOI: 10.1109/7298.946456.

[Cha+11] W. Chao u. a. „FSFI: A Full System Simulator-Based Fault Injection Tool“. In: *2011 First International Conference on Instrumentation, Measurement, Computer, Communication and Control*. 2011, S. 326–329. DOI: 10.1109/IMCCC.2011.88.

[Gos97] K. K. Goswami. „DEPEND: a simulation-based environment for system level dependability analysis“. In: *IEEE Transactions on Computers* 46.1 (1997), S. 60–74. ISSN: 1557-9956. DOI: 10.1109/12.559803.

[Har+17] S. K. S. Hari u. a. „SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation“. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2017, S. 249–258. DOI: 10.1109/ISPASS.2017.7975296.

[Jen+94] E. Jenn u. a. „Fault injection into VHDL models: the MEFISTO tool“. In: *Proceedings of IEEE 24th International Symposium on Fault- Tolerant Computing*. 1994, S. 66–75. DOI: 10.1109/FTCS.1994.315656.

[Mir+92] G. Miremadi u. a. „Two software techniques for on-line error detection“. In: *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*. 1992, S. 328–335. DOI: 10.1109/FTCS.1992.243568.

[Par+14] K. Parasyris u. a. „GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates“. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, S. 622–629. DOI: 10.1109/DSN.2014.96.

[Raj05] Rajesh Raina. „Is the concern for soft-error overblown?“ In: *IEEE International Conference on Test, 2005.* 2005, 2 pp.–1273. DOI: 10.1109/TEST.2005.1584103.

[Sch+15] H. Schirmeier u. a. „FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance“. In: *2015 11th European Dependable Computing Conference (EDCC)*. 2015, S. 245–255. DOI: 10.1109/EDCC.2015.28.

[Seg+88] Z. Segall u. a. „FIAT-fault injection based automated testing environment“. In: *[1988] The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. 1988, S. 102–107. DOI: 10.1109/FTCS.1988.5306.

[Zie+96] J. F. Ziegler u. a. „IBM experiments in soft fails in computer electronics (1978–1994)“. In: *IBM Journal of Research and Development* 40.1 (1996), S. 3–18. ISSN: 0018-8646. DOI: 10.1147/rd.401.0003.