technische universität
dortmund

Masterarbeit

Vergleich von Fehlerinjektionsexperimenten auf Logikgatter-Ebene mit höheren Abstraktionsebenen

Mark Breddemann
25. April 2016

Betreuer:
Dipl.-Inf. Horst Schirmeier
Prof. Dr.-Ing. Olaf Spinczyk

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
`http://ess.cs.tu-dortmund.de`

Zusammenfassung

Die Simulation von Prozessorfehlern durch Fehlerinjektion ist ein zentraler Bestandteil in der Erforschung und Entwicklung von Fehlertoleranzmechanismen. Durch Fehlerinjektionsexperimente werden die Mechanismen getestet und bewertet. Es ist daher essentiell wichtig, dass die Ergebnisse der Simulationen zu den korrekten Schlussfolgerungen führen. Die Simulatoren abstrahieren häufig stark von den realen Schaltungen der Prozessoren, da genaue Modelle komplex sind und zu langen Simulationszeiten führen. Bei einer großen Abstraktion besteht jedoch die Gefahr, dass das gebildete Modell ein anderes Verhalten als die Realität wiedergibt.

In dieser Arbeit wird zunächst eine Plattform entwickelt, die in einen ARM Cortex-M0-Prozessor Fehler auf Basis verschiedener, gebräuchlicher Fehlermodelle injizieren kann. Diese läuft auf einem FPGA und kann mehrere Experimente gleichzeitig mit einer, gegenüber anderen Simulatoren, sehr hohen Geschwindigkeit ausführen. Anschließend werden Experimentergebnisse verglichen, welche durch die verschiedenen, implementierten Fehlermodelle gewonnen werden können. Es wird gezeigt, dass die uniform verteilte Fehlerinjektion in Gates, Flip-Flops sowie CPU-Registern auf dem ARM Cortex-M0-Prozessor zu vergleichbaren Ergebnissen führt. Die Injektion in die CPU-Register ausschließlich während der Schreibzugriffe lässt jedoch andere Schlussfolgerungen bei der Bewertung der Ergebnisse zu. Ebenfalls wird der Einfluss der Metriken Fault-Coverage sowie der extrapolierten, absoluten Fehlerzahl (EAF) verglichen und anhand eines Beispiels gezeigt, dass die Fault-Coverage zu falschen Schlüssen bei der Bewertung von Fehlertoleranzmechanismen führen kann.

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel vermittelt einen Überblick über diese Arbeit. Dazu wird zunächst der Kontext und die daraus resultierende Problemstellung dargestellt. Am Ende dieses Kapitels wird knapp der Ansatz vorgestellt, der benutzt wird, um die Problemstellung zu untersuchen.

1.1 Kontext

Elektronische Geräte sind im Betrieb stets Umwelteinflüssen ausgesetzt, durch die das Systemverhalten variieren kann. Die Auswirkungen der Einflüsse können unterschiedlicher Natur sein, mechanische oder thermische Belastungen schädigen die Elektrik häufig dauerhaft, Neutronen in der Atmosphäre oder kleine Spannungsschwankungen beeinflussen das Verhalten lediglich für eine kurze Zeit [1]. Je kleiner die Strukturgröße der elektrischen Schaltung, desto anfälliger ist diese für verschiedene Einflüsse aus der Umgebung. Moderne Prozessoren werden in einer Strukturgröße von nur 14 Nanometern hergestellt, sodass die elektrischen Komponenten nur noch extrem wenig elektrische Kapazität aufnehmen können. Dadurch sind die Prozessoren kompakt und günstig, können mit wenig Energie betrieben werden und mit einer Frequenz im Gigahertzbereich schalten. Gleichzeitig sinkt jedoch die Robustheit gegenüber Umwelteinflüssen, welche zu dauerhaften oder transienten, also kurzzeitigen Fehlern führen können. In der Digitaltechnik kann man vereinfacht sagen, dass bei einem Fehler der Wert eines elektrischen Signals, welches eine Null oder eine Eins darstellt, von seinem korrekten Wert abweicht.

Eine Studie aus dem Jahr 1986 besagt, dass bis zu 85% aller Computerfehler transiente Fehler sind [2]. Im Jahr 2000 stellte Sun bei einem ihrer Serversysteme fest, dass diese sehr anfällig für Alpha-Teilchen seien, welche aus zwei Protonen und zwei Neutronen bestehen. Diese Serversysteme waren bereits im Produktiveinsatz bei America Online oder eBay und sorgten dort für Ausfälle [3]. Ein weiteres Beispiel ist Toyota, wo durch mangelhafte Absicherung gegen diese Fehlerart Autos unbeabsichtigt beschleunigen konnten. Dies führte in den Jahren 2009 bis 2011 zu einer großen Rückrufaktion [4].

Zuverlässige Systeme, deren Fehler fatale Folgen hätten, sind aus diesem Grund meist durch zusätzliche Redundanz abgesichert. Die Hardware des Flugcomputers einer Boeing 777 beispielsweise ist dreifach redundant abgesichert, mit verschiedenen Architekturen in den drei Einheiten [5]. Da dies mit hohen Kosten, sowohl bei der Entwicklung als auch in der Herstellung, verbunden ist, verfolgt die Forschungsgemeinschaft andere Ansätze zur Absicherung gegenüber Hardwarefehlern.

Ein neuer Ansatz zum Erhöhen von Fehlertoleranz eines Systems ist die *Software-basierte Hardware-Fehlertoleranz* (*SIHFT, Software-Implemented Hardware Fault Tolerance*). Bei dieser Technik wird die auf dem Computersystem laufende Software so modifiziert, dass diese Hardwarefehler erkennen und behandeln kann. Dies geschieht beispielsweise durch Konsistenzüberprüfung der benutzten Daten, Kontrollflussüberwachung oder mehrfacher Ausführung von Berechnungen [6].

1.2 Problemstellung

Bei der Entwicklung von Fehlertoleranzmaßnahmen ist es essentiell, diese zu bewerten und miteinander zu vergleichen. Hierfür gibt es verschiedene Ansätze. Ein trivialer wäre, das System so lange laufen zu lassen, bis ein solcher Fehler auftritt, um anschließend das Systemverhalten zu beobachten. Die Wahrscheinlichkeit, dass in freier Natur innerhalb eines bestimmten Zeitrahmens ein Hardwarefehler jedweder Art auftritt, ist jedoch extrem gering [7]. Um den Prozess zu beschleunigen ist es möglich, das *Device Unter Test* (*DUT*) in einer Strahlenkammer mit Neutronen zu bestrahlen. Dadurch erhöht sich die Wahrscheinlichkeit für das Auftreten von transienten Fehlern. Diese Bestrahlung ist jedoch sehr kostenintensiv und es ist nicht möglich, zu einem bestimmten Zeitpunkt einen Fehler an einem bestimmten Ort zu erzeugen, sodass in einer Strahlenkammer einzelne Fehlerfälle nicht reproduzierbar sind.

Ein anderer Ansatz sind *Simulator-basierte Fehlerinjektionsexperimente*. Darunter versteht man das bewusste injizieren eines Fehlers in einer simulierten Umgebung. Grundlage für diese Simulation ist ein *Systemmodell* und darauf aufbauend ein *Fehlermodell*.

Es gibt verschiedene Fehlermodelle mit unterschiedlichen Abstraktionsniveaus. Manche stellen viele Details der Hardware nach, zum Beispiel die physikalischen und elektrotechnischen Eigenschaften der Hardwarebausteine. Andere Modelle abstrahieren stark und simulieren nur noch Fehler in den CPU-Registern, welche die Schnittstelle der Software zu der Hardware darstellen. Die Benutzung abstrakter Modelle hat viele praktische Vorteile, da diese einfach und dadurch schnell simulierbar sind. Daher werden sie von den Entwicklern von Fehlertoleranzmaßnahmen präferiert. Jedoch ist es essentiell wichtig, dass eine Bewertung der Maßnahmen unter Verwendung dieser abstrakten Modelle zu den selben Schlüssen führt, wie die Fehlerinjektion in einer Strahlenkammer. In der Literatur gibt es Hinweise, dass dies nicht für alle Fehlermodelle und Bewertungskriterien zutrifft, obwohl diese in der Forschungsgemeinschaft nach wie vor gebräuchlich sind.

1.3 Ansatz

In dieser Arbeit wird experimentell untersucht, ob die Verwendung abstrakter Fehlermodelle sinnvoll ist. Außerdem werden Metriken zur Bewertung der Fehlertoleranz einer Software miteinander verglichen. Dazu wird überprüft, ob eine Bewertung der

Fehlertoleranz eines Programms nach einer Reihe von Fehlerinjektionsexperimenten mit verschieden abstrakten Fehlermodellen zu den selben Schlüssen führt. Dabei beschränkt sich die Arbeit auf die transienten Fehler von Prozessoren, welche insbesondere durch Neutronen in der Atmosphäre ausgelöst werden. Für die Untersuchung wird zunächst eine Fehlerinjektionsplattform entworfen und deren Implementierung erläutert. Diese nutzt einen ARM Cortex-M0-Prozessor, welcher um *Saboteure* erweitert wurde, um Fehler in diese CPU zu injizieren. Durch das Nutzen eines FPGAs als Ausführungsumgebung ist die Plattform gegenüber anderen Simulator-basierten Ansätzen sehr performant.

Die Plattform kann Fehlerinjektionsexperimente auf Basis verschieden abstrakter Fehlermodelle ausführen. Diverse auf diese Plattform portierte Benchmarks werden für den anschließenden Fehlertoleranzvergleich benutzt. Die unter Benutzung der unterschiedlichen Modelle gewonnenen Ergebnisse der Experimente werden anschließend analysiert. Es wird gezeigt, dass nicht alle Fehlermodelle für einen Vergleich der Fehlertoleranz geeignet sind. In diesem Zusammenhang werden auch zwei verschiedene Metriken zur Auswertung analysiert und verglichen. Anhand eines Beispiels eines Fehlertoleranzmechanismus wird gezeigt, dass die Metrik einen Einfluss auf die gezogenen Schlüsse hat und zu einer falschen Bewertung führen kann.

Nachdem in Kapitel 2 die nötigen Grundlagen in Form der verwendeten Fehlermodelle und Vergleichsmetriken erläutert wurden, wird in Kapitel 3 der Entwurf der Fehlerinjektionsplattform und die Idee der zu implementierenden Saboteure für die Fehlerinjektion erläutert. In Kapitel 4 wird erklärt, wie diese Ideen anhand eines ARM Cortex-M0-Prozessors umgesetzt wurden. Außerdem werden die wesentlichen Aspekte der entwickelten Plattform dokumentiert. Die Evaluation in Kapitel 5 ist zweigeteilt. Zunächst werden die Fehlermodelle und der Einfluss der Metrik analysiert, anschließend die Fehlerinjektionsplattform in Bezug auf die Geschwindigkeit und die benötigten Ressourcen. Abschließend folgt in Kapitel 6 eine Zusammenfassung und ein Ausblick auf mögliche weitere Untersuchungen, die auf dieser Arbeit aufbauen.

2 Grundlagen

Dieses Kapitel stellt die Grundlagen vor, welche im Kontext dieser Arbeit benötigt werden. Zunächst werden im Abschnitt 2.1 verschiedene System- und darauf aufbauend Fehlermodelle vorgestellt. Abschnitt 2.2 erläutert den Ablauf und mögliche Ergebnisse eines Fehlerinjektionsexperiments. Anschließend werden in Abschnitt 2.3 zwei verschiedene Metriken vorgestellt, die zum Auswerten der Experimente benutzt werden können. Abschließend folgt in Abschnitt 2.4 ein Blick auf andere Arbeiten, die sich mit einer ähnlichen Fragestellung beschäftigt haben.

2.1 System- und Fehlermodelle

Ziel eines Modells ist es, die Realität so gut wie möglich vereinfacht abzubilden, so auch bei den System- und Fehlermodellen. Bei den Systemmodellen besteht die Realität aus den analogen Bauteilen einer CPU und deren physikalischen und elektrotechnischen Eigenschaften. Eine moderne CPU beispielsweise besteht aus mehreren Milliarden Transistoren, welche durch Neutronen sowie elektromagnetische Effekte beeinflusst werden können.

Da es extrem rechenintensiv ist, alle Transistoren einer CPU mit ihren analogen Eigenschaften zu simulieren, wird dieses physikalische und elektrotechnische Systemmodell vereinfacht. Dazu wird angenommen, dass die Transistoren nur digital mit Binärwerten arbeiten und ein Verbund mehrerer Transistoren ein *Gate* darstellt. Nun ist es möglich, eine CPU auf Basis der Gates zu modellieren. Davon ausgehend kann man das Modell weiter vereinfachen, indem man nur die Gates betrachtet, welche *Flip-Flops* darstellen und den CPU-Zustand speichern. Zwischen den einzelnen Takten gibt es Zustandsübergänge, in denen vom alten Zustand ausgehend der neue beschrieben wird. Weitergehend ist es möglich, lediglich die *Instruction Set Architecture* (*ISA*), welche die CPU der Software bereitstellt, zu betrachten und daraus ein Systemmodell eines Prozessors zu bilden. Es ist auch denkbar ein Computersystem so weit zu abstrahieren, dass nur noch die in der Software benutzten Variablen simuliert werden.

Auf Basis dieser verschiedenen Systemmodelle ist es möglich, Fehlermodelle zu entwickeln. Ein zu dem Gate- oder Flip-Flop-Systemmodell gängiges Fehlermodell ermöglicht es, den digitalen Wert am Ein- oder Ausgang eines Gates bzw. Flip-Flops zu modifizieren [8]. Zu dem ISA-Systemmodell stellt das Modifizieren von Bits oder Bytes eines CPU-Registers entweder zu einem beliebigen Zeitpunkt oder bei Register-Schreibvorgängen jeweils ein gängiges Modell dar [9]. Betrachtet das Systemmodell

nur noch die Software, können Fehler in die verwendeten Variablen eingefügt werden [10].

Häufig besteht das Problem, dass für viele kommerziell verfügbare CPUs kein Gate oder Flip-Flop-Systemmodell verfügbar ist und dementsprechend kein darauf aufbauendes Fehlermodell entwickelt werden kann. Der Grund ist, dass die CPU-Hersteller die für die Modellentwicklung benötigte Implementierung der CPU-Logik unter Verschluss halten und nur die Dokumentation der ISA veröffentlichen. Eine andere Problematik mit detaillierten Fehlermodellen stellt die Größe und Komplexität des Modells sowie die daraus resultierende langsame Geschwindigkeit bei der Simulation dar. Dies führt dazu, dass viele Fehlertoleranzmaßnahmen nur mittels eines ISA-Level-Fehlermodells getestet werden, (z.B. [11]).

Je nachdem, welcher Fehlertyp simuliert werden soll, kann verschiedenes Fehlverhalten an den von den Fehlermodellen bereitgestellten Fehlerinjektionsorten implementiert werden:

- *Permanente Fehler*: Bei dieser Art werden bleibende Fehler eines Bauteils simuliert, indem an einem Fehlerinjektionsort stets eine 0, 1 oder ein beliebiger, nichtdeterministischer Wert angelegt wird.

- *Single Event Upset* (*SEU*): Bei diesem Modell wird der Wert am Injektionsort für einen sehr kurzen Zeitraum invertiert. Dieses stellt transiente Fehler dar.

Die beiden Kategorien lassen sich weiter unterteilen. Es ist möglich, Fehler nur an einem Fehlerort oder an mehreren gleichzeitig zu injizieren. Auch können während einer Simulation mehrere SEUs nacheinander ausgelöst werden.

Diese Arbeit beschränkt sich auf die Betrachtung von Single Event Upsets, bei denen ein Fehler für die Länge eines Taktes injiziert wird. Dadurch entsteht ein diskreter *Fehlerraum*, welcher jede mögliche Kombination aus Injektionsort und -zeitpunkt enthält. Das zeitdiskrete Injizieren von Fehlern ist bereits eine Abstraktion von der Realität, da die realen Fehler natürlich nicht mit dem CPU-Takt synchronisiert sind. Ebenso wird von einer uniformen Verteilung des Auftretens der Fehler ausgegangen. Das heißt, dass alle Fehlerinjektionspunkte mit derselben Wahrscheinlichkeit ausgewählt werden. Auch dies entspricht nicht der Realität, da manche Komponenten durch bessere Abschirmung seltener von einem Fehler betroffen sind oder benachbarte Komponenten mit einer erhöhten Wahrscheinlichkeit gleichzeitig ein fehlerhaftes Verhalten aufweisen. Eine verbreitete Abstraktion bei Fehlermodellen ist das Beschränken der Fehlerinjektionszeitpunkte auf Schreibzugriffe, zum Beispiel bei CPU-Registern. Dadurch wird der Fehlerraum stark verkleinert.

Unter der Annahme, dass die Fehlermodelle wie die Systemmodelle aufeinander aufbauen, besteht der Fehlerraum eines abstrakten Fehlermodells aus einer Teilmenge des Fehlerraums eines weniger abstrakten Modells. In Abbildung 2.1 ist dies illustriert. Alle Fehlerpunkte gehören zu dem Gate-Fehlermodell, während nur eine Untermenge der Fehlerorte auch zu dem Flip-Flop-Fehlermodell gehören. Dies setzt sich bei dem ISA-Fehlermodell weiter fort. Zu dem Fehlerraum des Register-On-Write-

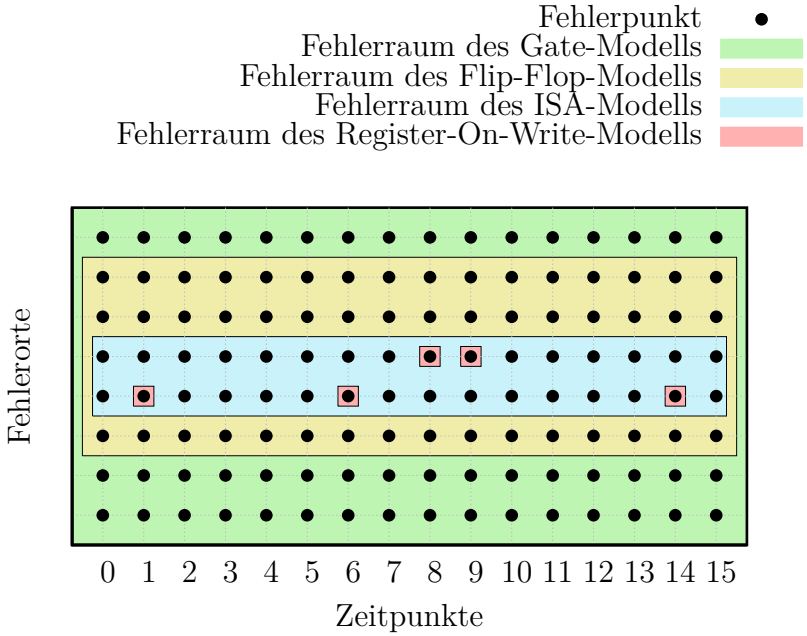

Abbildung 2.1: Illustration der Fehlerräume verschiedener Fehlermodelle

Fehlermodells zählen nur die Fehlerpunkte, welche einen Fehler in ein zu genau diesem Zeitpunkt beschriebenes CPU-Register injizieren.

2.2 Fehlerinjektionsexperimente

Die Fehlerinjektion geschieht im Rahmen eines *Fehlerinjektionsexperimentes*. Dazu wird die CPU vor Beginn des Experiments in einen definierten Zustand gebracht. Üblicherweise wird dafür der Zustand nach dem Einschalten der CPU benutzt. Anschließend führt diese die Instruktionen auf den gegebenen Daten aus, wobei nach einer gewissen Zeit ein Fehler in das System injiziert wird.

Die Auswirkungen auf das System der in dieser Arbeit betrachteten Single Event Upsets sind vielseitig. Tritt ein solcher in einer zu diesem Zeitpunkt ungenutzten CPU-Einheit auf, hat er häufig keinerlei Auswirkungen. Tritt der Fehler jedoch in einer Komponente auf, deren Ergebnis weiterverarbeitet wird, weicht das weitere Systemverhalten von dem eines fehlerfreien Systems ab. Dann rechnet die CPU beispielsweise mit falschen Daten oder der Kontrollfluss der CPU kann in eine Endlosschleife geraten.

Das Verhalten nach einem Fehler wird im Kontext dieser Arbeit in verschiedene Kategorien eingeteilt:

- *Benign/no Error*: Der Fehler hat keine Auswirkung auf das Ergebnis
- *Silent Data Corruption* (*SDC*): Die Ausführung der Software terminiert, liefert jedoch ein falsches Ergebnis

- *Timeout*: Das Programm terminiert nicht. Durch externe Maßnahmen (z.B. Watchdog) kann dies erkannt werden.
- *Lockup*: Die CPU stellt selbstständig fest, dass sie nicht in der Lage ist, weiter zu arbeiten. Dies kann ebenfalls durch externe Maßnahmen erkannt werden.

Die Einteilung in die Kategorien Benign und SDC geschieht ausschließlich anhand des Ergebnisses. Wenn das Systemverhalten nach einem Fehler zunächst von dem eines fehlerlosen Systems abweicht, das System anschließend dennoch das korrekte Ergebnis unterhalb der Timeout-Zeit liefert, wird dieses Verhalten der Gruppe Benign zugeordnet.

Ein einzelnes Fehlerinjektionsexperiment alleine hat nicht viel Aussagekraft. Im Idealfall wird im Rahmen einer *Fehlerinjektionskampagne* zunächst das fehlerfreie Systemverhalten anhand eines *Golden Runs* analysiert und das "goldene" Ergebnis gespeichert. Anschließend wird für jeden Punkt im Fehlerraum ein Experiment ausgeführt und das Ergebnis mit dem des Golden Runs verglichen. Dies ist jedoch nur in den seltensten Fällen praktisch realisierbar, da der Fehlerraum zu groß ist und die Experimente zu lange dauern. Daher ist es üblich, nur einen Teil der zur Verfügung stehenden Experimente auszuführen, sodass man statistisch aussagekräftige Ergebnisse erhält.

Um einen allgemeinen Fehlertoleranzmechanismus zu bewerten, werden mehrere Kampagnen mit verschiedenen *Benchmarks* ausgeführt. Die Benchmarks stellen unterschiedliche Anwendungen dar und sollten möglichst verschieden sein, um eine hohe Aussagekraft bezüglich der allgemeinen Verwendbarkeit eines Toleranzmechanismus zu erhalten.

2.3 Vergleichsmetriken

Das Ergebnis einer Fehlerinjektionskampagne besteht aus den absoluten Häufigkeiten, wie oft ein Benchmark ein bestimmtes Verhalten (Benign, SDC, Timeout, Lockup) nach einer Fehlerinjektion zeigt. Die Werte dieser Zahlen sind insbesondere von der Anzahl der ausgeführten Experimente abhängig. Um davon unabhängig die Ergebnisse zweier Fehlerinjektionskampagnen miteinander vergleichen zu können, müssen die Rohdaten mittels einer Metrik weiter verarbeitet werden.

Weit verbreitet ist die Angabe des *Fault-Coverage-Faktors*, kurz *Fault-Coverage*. Diese gibt das Verhältnis von fehlerhaften und nicht-fehlerhaften Experimenten an und berechnet sich durch

$$c = 1 - \frac{\text{Anz. SDCs}}{\text{Anz. Experimente}}. \tag{2.1}$$

Eine hohe Fault-Coverage deutet hierbei auf einen fehlertoleranteren Benchmark hin.

Die Fault-Coverage ist nicht ganz unproblematisch, da sich diese beispielsweise durch das Hinzufügen von vielen a priori fehlertoleranten *nop*-Operationen leicht manipulieren lässt, ohne dass das eigentliche Programm fehlertoleranter geworden

ist. Die Ursache liegt darin, dass die Metrik die Größe des Fehlerraums vernachlässigt. Dadurch ist sie nicht geeignet, Kampagnen mit unterschiedlichen Laufzeiten der Benchmarks oder verschiedenen Fehlermodellen miteinander zu vergleichen [12].

Eine Alternative zur Fault-Coverage ist die *extrapolierte absolute Fehlerzahl* (*EAF*). Diese berücksichtigt im Gegensatz zur Fault-Coverage die Fehlerraumgröße, die sich insbesondere durch die Laufzeit eines Benchmarks ergibt. Die EAF gibt die absolute Anzahl der Punkte des Fehlerraums an, welche ein fehlerhaftes Verhalten bzw. eine SDC verursachen. Je kleiner dieser Wert, desto fehlertoleranter ist der getestete Benchmark. Um diesen Wert genau zu bestimmen, müsste für jeden Punkt im Fehlerraum ein Experiment ausgeführt werden. Da der Fehlerraum häufig viele Milliarden Punkte umfasst, ist dies in der Praxis nicht realisierbar. Würde die Ausführung jedes Experiments bei einer Fehlerraumgröße von einer Milliarde Punkten nur eine Sekunde benötigen, läge die Gesamtlaufzeit der benötigten Fehlerinjektionskampagne bei über 30 Jahren.

Aus diesem Grund wird während der Kampagne nur ein Bruchteil der Punkte im Fehlerraum ausgewählt und zur Fehlerinjektion benutzt. Die ermittelten Ergebnisse werden anschließend hochgerechnet, wodurch die absolute Fehlerzahl extrapoliert wird. Die Rechenvorschrift lautet hier

$$\text{EAF} = \text{Fehlerraumgröße} \ * \ \frac{\text{Anz. SDCs}}{\text{Anz. Experimente}} \quad [12] \tag{2.2}$$

Da es sich bei der EAF um einen extrapolierten Wert handelt, der mit hoher Wahrscheinlichkeit von dem realen Wert abweicht, muss die Genauigkeit des ermittelten Wertes angegeben werden. Für diese Angabe eignet sich das Konfidenzintervall, dessen Größe insbesondere von der Anzahl der gemessenen SDCs und damit von der Anzahl der Experimente abhängt [13].

2.4 Abstraktionsebenen-übergreifende Vergleiche

Es gibt bereits andere Arbeiten, welche sich mit dem Vergleich von Fehlerinjektionsexperimenten auf unterschiedlichen Fehlermodellebenen beschäftigen.

Im Paper *Soft-error resilience of the IBM POWER6 processor* [14] wurde die Fehlertoleranz des genannten Prozessors mittels eines Benchmarks überprüft, der stets den Architekturstatus auf ungültige Zustände überprüft und somit Fehler detektieren kann. Einerseits wurden Fehler in die Flip-Flops des Prozessors injiziert, andererseits wurde er in einer Strahlenkammer mit Protonen und Neutronen bestrahlt. Die Ausgänge der Experimente wurden ähnlich kategorisiert wie in dieser Arbeit (siehe Abschnitt 2.2). Die Ergebnisse der ausgeführten Experimentreihen unter Benutzung der verschiedenen Injektionsmethoden weichen nur minimal voneinander ab. Es wird daher gefolgert, dass eine Fehlerinjektion in Flip-Flops zu denselben Ergebnissen führt, wie eine Injektion in einer Strahlenkammer.

Das Paper *Quantitative evaluation of soft error injection techniques for robust system design* [15] vergleicht Fehlerinjektionsergebnisse, die durch Benutzung ver-

schiedener Fehlermodelle erlangt worden sind. Implementiert wurde ein Flip-Flop-Fehlermodell (ohne Cache und CPU-Register) sowie ein ISA- und Programmvariablen-Modell (jeweils uniform oder bei Schreibzugriffen). Benutzt wurde dafür ein LEON3-Prozessor, basierend auf der SPARC V8 Architektur [16].

Festgestellt wurde eine hohe Ungenauigkeit der Ergebnisse zwischen der Fehlerinjektion in Flip-Flops und der Injektion in CPU-Register oder Programmvariablen. Es wurden Kampagnenergebnisse gemessen, bei denen die anteiligen Raten der verschiedenen Experimentausgänge (ähnlich zu Benign, SDC, Timeout, Lockup) um den Faktor 0.07 bis 45 bei der Verwendung unterschiedlicher Fehlermodelle voneinander abweichen. Außerdem sei kein Trend erkennbar, welcher diese Abweichungen erklären könne.

Ein Kritikpunkt an beiden Arbeiten ist insbesondere die ausschließliche Verwendung des Fault-Coverage-Faktors, da dieser, wie im vorherigen Abschnitt beschrieben, ungeeignet ist, Fehlerinjektionskampagnen mit unterschiedlicher Fehlerraumgröße zu vergleichen.

2.5 Zusammenfassung

Dieses Kapitel stellte die wesentlichen Grundlagen vor, die für den Kontext dieser Arbeit benötigt werden. Zunächst wurden die verschiedenen abstrakten Fehlermodelle, die aus den Systemmodellen eines Prozessors hervorgehen, vorgestellt. Hierbei wurden insbesondere SEU-Fehler betrachtet, die für diese Arbeit benutzt werden, um die durch Neutronen verursachten Fehler nachzubilden. Anschließend wurde der Ablauf eines Fehlerinjektionsexperiments und dessen möglichen Ergebnisse vorgestellt. Die Ergebnisse einer Fehlerinjektionskampagne, die aus vielen Experimenten besteht, werden mit Hilfe einer Metrik weiterverarbeitet. Einerseits wird dafür der Fault-Coverage-Faktor benutzt, welcher jedoch eine große Schwäche hat, da er die Fehlerraumgröße nicht betrachtet. Andererseits wird die EAF-Metrik vorgestellt, welche die Größe des Fehlerraums berücksichtigt. Aus der Problematik einer eventuell zu großen Abstraktion der Fehlermodelle sowie der Ungleichheit der Metriken ergibt sich die Motivation für die Entwicklung einer Fehlerinjektionsplattform, um diese Probleme anschließend analysieren zu können.

Das Ziel dieser Arbeit ist es, die Widersprüche der in Abschnitt 2.4 vorgestellten Arbeiten zu untersuchen, insbesondere auch in Hinblick auf die verwendete Metrik.

Zentral ist die Untersuchung, ob verschiedene Fehlermodelle unterschiedliche Ergebnisse erzeugen und voneinander abweichende Folgerungen zulassen. Dazu wird zunächst ein möglichst realitätsnahes Gate-Level-Fehlermodell einer CPU erstellt. Weitere Modelle entfernen sich schrittweise von diesem, beispielsweise ein Flip-Flop- oder ein ISA-Level-Modell. Speicherfehler werden nicht betrachtet. Zur Bewertung wird die extrapolierte absolute Fehlerzahl (EAF) und der Fault-Coverage-Faktor benutzt, damit diese Metriken miteinander verglichen werden können. Darüber hinaus wird das im Rahmen dieser Arbeit entwickelte Fehlerinjektionsframework mit einem um Fehlerinjektionen erweiteten Gem5 Simulator aus Fail∗ [17] gegenübergestellt.

3 Entwurf einer Fehlerinjektionsplattform

In diesem Kapitel werden zunächst in Abschnitt 3.1 die Anforderungen an das benötigte System erläutert, um Fehlerinjektionsexperimente auf verschiedenen Abstraktionsebenen ausführen zu können. Aus den Anforderungen ergeben sich mehrere zur Verfügung stehende Designalternativen, welche in den Abschnitten 3.2 und 3.3 beschrieben werden. Die getroffenen Designentscheidungen werden ebenfalls in den jeweiligen Abschnitten erläutert. In Kapitel 3.4 wird der Entwurf der zu entwickelnden Fehlerinjektionsplattform vorgestellt und erläutert, wie die gewählte CPU um Fehlerinjektionsorte erweitert wird. Abschnitt 3.5 erklärt abschließend knapp den Ablauf einer Fehlerinjektionskampagne.

3.1 Anforderungen

Grundgedanke bei der Anforderungsanalyse war stets das Hauptziel dieser Arbeit, die Untersuchung der Ergebnisse von Fehlerinjektionsexperimenten mit verschiedenen Fehlermodellen.

Folgende Anforderungen wurden zu Beginn der Arbeit identifiziert:

1. **Experimentelle Plattform:** Die zu implementierende Plattform soll Fehlerinjektionsexperimente ausführen können. Eine rein analytische Betrachtung der Folgen nach einer Fehlerinjektion scheidet aus, da die zu betrachtenden Systeme sehr komplex sind und daher von einer Zustandsexplosion auszugehen ist.

2. **Realistisches System:** Es soll ein möglichst typisches und "normales" System betrachtet werden, wie es in der Realität benutzt wird, um eine möglichst hohe Aussagekraft bezüglich der Ergebnisse zu erhalten.

3. **Verschiedene Fehlermodelle:** Damit Vergleiche zwischen mehreren Fehlermodellen getätigt werden können, müssen natürlich mehrere dieser Modelle zur Verfügung stehen. Mindestens untersucht werden sollen uniform verteilte Fehler auf Basis des Gate-, Flip-Flop und ISA-Systemmodells. Es ist wünschenswert, das ISA-Modell zu unterteilen, um zum Beispiel uniform verteilte Einzel- und Mehrbitfehler sowie Fehler bei Schreibzugriffen auf die Register simulieren zu können. Das Gate-Modell soll bei den Vergleichen als Referenzmodell benutzt werden, da dieses den realen elektrischen Schaltungen am nächsten ist.

4. **Benchmarks:** Um nach der Entwicklung des Systems den gewünschten Vergleich verschiedener Fehlermodelle tätigen zu können, sollen möglichst viele unterschiedliche Benchmarks auf die Fehlerinjektionsplattform portiert werden können.

5. **Fehlerinjektionskampagne:** Die zu entwickelnde Plattform soll nicht nur in der Lage sein, ein einzelnes Experiment auszuführen, sondern soll möglichst automatisiert komplette Fehlerinjektionskampagnen ausführen können, damit die Ergebnisse eine hohe statistische Aussagekraft haben.

6. **Reproduzierbar:** Die Fehlerinjektionskampagnen sollen reproduzierbar sein, sodass man bei mehrfacher Ausführung desselben Experiments oder derselben Kampagne übereinstimmende Ergebnisse erhält.

7. **Performant:** Die zu entwickelnde Fehlerinjektionsplattform soll auch größere Experimente und Kampagnen performant abarbeiten können.

8. **Kostengünstig:** Die Fehlerinjektionsplattform soll günstig in der Herstellung sowie im Betrieb sein.

3.2 Wahl der Ausführungsumgebung

Dieser Abschnitt beschäftigt sich mit der Frage, in welcher Umgebung das zu entwickelnde System ausgeführt werden soll.

Alternativlos ist die Benutzung einer Hardware-Beschreibungssprache (*HDL, Hardware Description Language*) wie Verilog oder VHDL. Die CPU muss in einer solchen vorliegen, damit sie flexibel weiterverarbeitet und um Fehlerinjektionsorte, sogenannte *Saboteure*, ergänzt werden kann. Die Benutzung einer solchen Sprache stellt keine Einschränkung dar, da die Entwicklung von CPUs üblicherweise mittels einer HDL geschieht.

Ein in einer HDL beschriebenes Hardwaresystem ist simulierbar. Ein verbreiteter kommerzieller Simulator ist Modelsim von Mentor Graphics [18]. Eine Open-Source-Alternative ist Icarus Verilog [19]. Problematisch bei Fehlerinjektionssystemen ist die Komplexität. Saggese et al. entwickelten einen entsprechenden Simulator, welcher auch in der Lage ist, Fehler auf Gate-Level zu injizieren [20]. Die Laufzeit, um 289 Takte einer DLX-CPU zu simulieren, betrug 30 Sekunden auf einem Intel 3,0 GHz Pentium 4 Prozessor unter Verwendung von Modelsim. Die heutigen Prozessoren sind zwar wesentlich schneller und mehrere Fehlerinjektionsexperimente sind sehr gut parallelisierbar, jedoch ist bei größeren Experimenten mit vielen Millionen zu simulierenden Takten pro Experiment und einer hohen Zahl an Experimenten eine sehr lange Kampagnenlaufzeit zu erwarten.

Eine Alternative zu dem Software-basierten Hardware-Simulator ist die Implementierung der benötigten Schaltung auf einem *ASIC* (*Application-specific Integrated Circuit*). Die Schaltungen sind zur Laufzeit sehr schnell und haben eine hervorragende Energiebilanz. Die Herstellung von wenigen maßgefertigten ASICs kostet jedoch

viele tausend Euro und ist daher für den Rahmen dieser Arbeit keine Option, zudem diese Kosten für eventuell fehlerbereinigte Hardwarerevisionen wiederholt anfallen.

Eine andere Alternative ist die Verwendung eines *FPGA*s (*Field Programmable Gate Array*), in das beliebig oft eine digitale Schaltung programmiert werden kann. Die konfigurierbaren Logikzellen des FPGAs werden bei der Programmierung so verknüpft, dass sie die programmierte Schaltung darstellen. Gegenüber ASICs ist die Kapazität eines FPGAs um den Faktor 40 kleiner, die Geschwindigkeit der Schaltung ist um den Faktor 3.2 verringert und der Energieverbrauch um den Faktor 12 erhöht [21]. Die Anschaffungskosten für ein FPGA halten sich jedoch im Rahmen und sind für diese Arbeit akzeptabel.

Aufgrund der relativ geringen Kosten eines FPGAs gegenüber eines ASICs und der hohen Geschwindigkeit gegenüber einem Simulator, wurde für die Entwicklung der Fehlerinjektionsplattform eine FPGA-Basis ausgewählt. Wären hierbei schwerwiegende Probleme aufgetreten, hätte weiterhin die Möglichkeit bestanden, die bis zu diesem Zeitpunkt entwickelte Plattform auf einen Simulator zu portieren. Dies wäre ohne eine komplette Neuentwicklung möglich, da beide auf Basis einer HDL arbeiten.

3.3 Wahl der CPU

Dieser Abschnitt beschäftigt sich mit einer Auswahl von CPUs, welche für die Fehlerinjektion infrage kommen.

Voraussetzung für die Nutzbarkeit einer CPU für die zu entwickelnde Fehlerinjektionsplattform ist die Verfügbarkeit eines Gate-Level-Modells der CPU, um das dazu entsprechende Fehlermodell implementieren zu können. Liegt die CPU in einer HDL vor, ist es mittels Tools wie dem kommerziellen Synopsys Design Compiler [22] oder dem freien Yosys [23] (nur Verilog) möglich, daraus ein Gate-Level-Modell zu erzeugen.

Der Synopsys Design Compiler eignet sich aufgrund hoher Lizenzkosten nur bedingt für diese Arbeit. Yosys ist ein sehr junges Open-Source Projekt, welches "fast jedes synthetisierbares Verilog-2005-Design" [23] [1] verarbeiten kann.

Bei der folgenden Vorstellung verschiedener, infrage kommender CPUs werden nur solche betrachtet, die ohne Kosten zugänglich sind. Von vielen kommerziell erhältlichen CPUs kann man Lizenzen erwerben, sodass man auch Zugriff auf den HDL-Quellcode erhält. Diese Lizenzen sind jedoch in der Regel sehr teuer und daher keine Option.

Die Entscheidung bei der Wahl einer für die Fehlerinjektionsplattform passenden CPU erfolgt anhand der folgenden Kriterien:

- **Open-Source-Lizenz:** Steht eine CPU unter einer Open-Source-Lizenz, ist der Zugriff auf den HDL-Quellcode sowie dessen Manipulation und spätere Veröffentlichung kein Problem. Zudem sind weitergehende Analysen bezüglich der

[1] "Process almost any synthesizable Verilog-2005 design" [23]

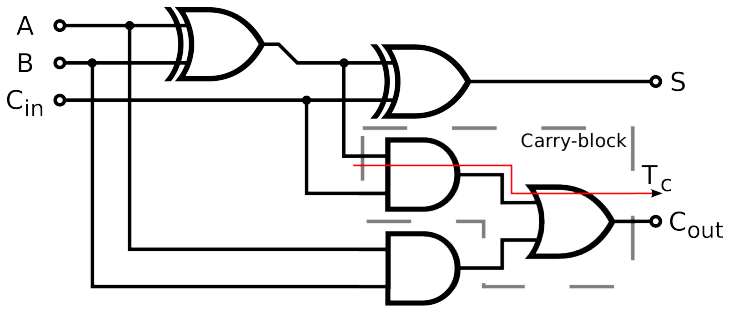

Abbildung 3.1: Gatestruktur eines Volladdierers (Quelle: [24])

Fehlerausbreitung nach einem transienten Fehler oder der Fehlerempfindlichkeit einzelner CPU-Einheiten denkbar.

- **Verbreitung:** Es sollen Fehler in ein typisches System injiziert werden. Ein sehr guter Indikator, ob dies für eine CPU zutrifft, ist deren Verbreitung. Ein Nebeneffekt verbreiteter CPUs ist die häufig gute Unterstützung von Entwicklertools, auch von freien. Bei wenig verbreiteten CPUs, insbesondere die, die auch eine komplett neu entwickelte Mikroarchitektur enthalten, ist die Wahrscheinlichkeit auf Fehler in der Mikroarchitektur oder Toolchain zu treffen, umso höher. Diese Fehler sind extrem aufwändig zu finden und zu beheben, und stellen daher ein Risiko für eine zeitlich begrenzte Arbeit dar.

- **Komplexität der CPU:** Die CPU, in die Fehler injiziert werden sollen, muss eine ausgewogene Komplexität besitzen. Sie sollte nicht zu komplex sein, insbesondere damit die Fehlerinjektion mit Hilfe eines ressourcenbeschränkten FPGAs möglich ist. Es ist davon auszugehen, dass die erforderliche Logik zum Implementieren der Saboteure viele Ressourcen auf dem FPGA benötigt.

- **Verilog HDL oder Source in Gate-Level:** Um aus HDL-Code eine Gate-Level-Repräsentation der CPU zu generieren, kommt aufgrund der Lizenzkosten nur Yosys infrage. Dieses kann nur Verilog verarbeiten, jedoch kein VHDL. Wenn eine CPU in VHDL beschrieben ist, muss sie daher zwingend bereits in einer Gate-Level-Beschreibung verfügbar sein.

  Im weiteren Verlauf dieses Kapitels werden viele Aspekte der Gate-Level-Repräsentation anhand eines Volladdierers beispielhaft beschrieben. In Abbildung 3.1 ist die Gate-Struktur eines solchen Addierers abgebildet, in Abbildung 3.2 der Verilog-Code eines Volladdierers. Auf der linken Seite ist eine Variante zu sehen, wie man den Addierer üblicherweise in Verilog implementieren würde. Auf der rechten Seite ist der Code nach der Konvertierung in eine Gate-Level-Repräsentation mittels Yosys abgebildet, bei dem die einzelnen Gates aus Abbildung 3.1 erkennbar sind.

```
1  module full_adder(a_in, b_in,
       c_in, s_out, c_out);
2
3  input a_in;
4  input b_in;
5  input c_in;
6  output s_out;
7  output c_out;
8
9  assign s_out = a_in ^ b_in ^
       c_in;
10 assign c_out = (a_in & b_in) |
11                (a_in & c_in) |
12                (b_in & c_in);
13
14 endmodule
```

```
1  /* Generated by Yosys 0.5 */
2
3  module full_adder(a_in, b_in,
       c_in, s_out, c_out);
4
5  wire _00_;
6  wire _01_;
7  wire _02_;
8  wire _03_;
9  wire _04_;
10
11 input a_in;
12 input b_in;
13 input c_in;
14 output c_out;
15 output s_out;
16
17 assign _04_  = a_in ^ b_in;
18 assign s_out = _04_ ^ c_in;
19 assign _00_  = a_in & b_in;
20 assign _01_  = a_in & c_in;
21 assign _03_  = _00_ | _01_;
22 assign _02_  = b_in & c_in;
23 assign c_out = _03_ | _02_;
24
25 endmodule
```

Abbildung 3.2: “Normaler” Verilog-Code eines Volladdierers (links) und deren Repräsentation als Verilog-Gate-Level-Beschreibung (rechts)

CPU	Open-Source	Verbreitung	Komplexität	Verilog/Gate
OpenRISC 1200	✓	✗	✓	✓
openMSP430	✓	✗	✓	✓
Amber 2 Core	✓	✗	✗	✓
MBLite+	✓	✗	✓	✗
ZPU	✓	✗	✓	✗
LEON3	✓	✓	✗	✗
ARM Cortex-M0	✗	✓	✓	✓

Tabelle 3.1: Bewertung verschiedener CPUs bezüglich dessen Eignung für die Fehlerinjektionsplattform

In Tabelle 3.1 ist eine Übersicht verschiedener infrage kommender CPUs mit deren Bewertung bezüglich der genannten Kriterien dargestellt. Im Folgenden werden die einzelnen CPUs vorgestellt:

- **OpenRISC 1200 [25]:** Diese 32-Bit-CPU besteht aus einer 5-stufigen Pipeline und enthält eine MMU. Sie ist eine Implementierung der neu entwickelten OpenRisc-1000-Mikroarchitektur [26] und steht unter einer Open-Source-Lizenz. Die CPU wurde bisher auf verschiedene FPGAs programmiert und wird noch weiterentwickelt, daher könnte sie noch Fehler enthalten. Es gibt einen Port der *GNU Compiler Collection* (*GCC*) für die OpenRISC1000 Architektur [27].

- **openMSP430 [28]:** Diese CPU ist ein Open-Source-Klon der pipelinelosen 16-Bit-CPU aus der MSP430-Mikrocontrollerfamilie von Texas Instruments [29]. Diese CPU wurde sowohl auf einem FPGA programmiert, als auch auf einem ASIC realisiert. Für die MSP430-Architektur pflegt Texas Instruments einen Port der GNU Compiler Collection [30].

- **Amber 2 Core [31]:** Es gibt zwei Versionen dieser CPU, eine mit einer 3-stufigen und eine mit einer 5-stufigen Pipeline. Das dazugehörige System ist relativ komplex und beinhaltet unter anderem einen DRAM- und einen Ethernet-Controller. Beide Versionen der CPU sind kompatibel zur ARM-v2a-ISA, welche in der GNU Compiler Collection abgedeckt ist.

- **MBLite+ [32]:** Diese CPU ist ein Open-Source-Klon der MicroBlaze-CPU von Xilinx und kann einen Großteil der Instruktionen dieser Architektur verarbeiten. Sie hat eine 5-stufige Pipeline und wurde bereits im LavA-Projekt verwendet [33]. Bei der dortigen Benutzung der CPU wurden in dieser bereits Fehler entdeckt und behoben. Die Architektur wird von der GNU Compiler Collection unterstützt.

- **ZPU [34]:** Die ZPU ist eine sehr kompakte, stackbasierte 32-Bit-Open-Source CPU, welche vom GCC unterstützt wird. "Der Instruktionssatz wurde in den letzten Jahren nicht verändert und kann als recht stabil betrachtet werden" [34][2]

- **LEON3 [16]:** Dieser von der ESA entwickelte Prozessor [35] basiert auf der verbreiteten SPARC-V8 Architektur und wird sowohl für FPGAs als auch ASICs verwendet. Er steht unter einer Open-Source-Lizenz. In einer erweiterten Version wurde diese CPU bereits für Satelliten verwendet [36]. Der LEON3 besitzt eine 7-stufige Pipeline, eine FPU, einen SRAM-Controller, einen Cache und eine MMU. Daher ist er wahrscheinlich zu komplex für eine umfassende Erweiterung um Saboteure auf Basis einer ressourcenbeschränkten FPGA-Plattform.

[2] "The current ZPU instruction set and architecture has not changed for the last couple of years and can be considered quite stable." [34]

- **ARM Cortex-M0 [37]:** Diese kommerziell erhältliche 32-Bit-CPU adressiert insbesondere kosten- und energieeffiziente Systeme und besitzt eine 3-stufige Pipeline. Den ARM Cortex-M0 findet man in vielen kommerziell erhältlichen Mikrocontrollern. Für Universitäten ist diese CPU in Form von verschleiertem HDL-Code kostenfrei zugänglich. Dieser verschleierte Code ist sehr nah an einer Gate-Level-Repräsentation der CPU. Zu Diagnosezwecken kann die CPU den Inhalt der CPU-Register sowie des Instruktions- und Stackpointers ausgeben.

Der MBLite+, die ZPU sowie der LEON3 kommen für die Verwendung für die Fehlerinjektionsplattform nicht infrage, da sie in VHDL vorliegen und keine Gate-Level Repräsentation verfügbar ist. Der Amber 2 Core ist recht komplex, was zu Kapazitätsproblemen auf dem FPGA führen könnte, wenn die CPU um Fehlerinjektion erweitert wird. Übrig bleiben der OpenRISC 1200, der openMSP430 sowie der ARM Cortex-M0. Die ersten beiden sind durch eine Open-Source Lizenz frei verfügbar, jedoch wenig verbreitet. Der ARM Cortex-M0 ist nur als verschleierter Verilog-HDL-Code zugänglich, dafür wird er in verschiedenen verbreiteten Mikrocontrollern benutzt.

Aufgrund der Erfahrungen aus dem LavA-Projekt, bei dem der wenig verbreitete MBLite+-Prozessor eingesetzt wurde und in diesem Fehler nach langwieriger Suche gefunden und behoben wurden, wird für diese Arbeit der ARM Cortex-M0-Prozessor präferiert. Diese CPU muss aufgrund des verschleierten Quellcodes als Blackbox betrachtet werden. Insbesondere sind Analysen wie die Fehlerempfindlichkeit einzelner Komponenten der CPU nicht möglich. Auf einer solchen Analyse liegt jedoch nicht der Fokus dieser Arbeit, während eine stabile CPU hingegen essentiell ist. Das Erkennen, Suchen und Beheben von CPU-Fehlern ist sehr zeitintensiv und soll nach aller Möglichkeit vermieden werden.

Der Verilog-Quellcode, in dem der ARM Cortex-M0 vorliegt, ist bereits sehr nah an einem Gate-Level-Modell der CPU. Es befinden sich lediglich einige wenige Additionen und Subtraktionen von Bitvektoren im Quellcode, damit die Synthesesoftware diese Funktionen in eigens für diesen Zweck auf dem FPGA dedizierten ALU-Komponenten platzieren kann. Diese Komponenten können für die Fehlerinjektion jedoch nicht verwendet werden, da auch in die Addier- und Subtrahierlogik Fehler injiziert werden müssen. Schwieriger als bei anderen CPUs wird die Implementierung einer ISA-Level Fehlerinjektion für den ARM Cortex-M0, da die Diagnosesignale, welche die Registerinhalte ausgeben, nur lesbar sind. Diese müssen zurückverfolgt werden zu den Flip-Flops, welche die Inhalte speichern, um dort Fehler injizieren zu können.

3.4 Fehlerinjektions-Plattform

Wie in Kapitel 3.2 beschrieben, soll die Fehlerinjektionsplattform mit Hilfe eines FPGAs umgesetzt werden. Dadurch ist es möglich, die Architektur der benutzten Hard- und Software komplett selbst zu bestimmen und an die zu entwickelnde Plattform anzupassen.

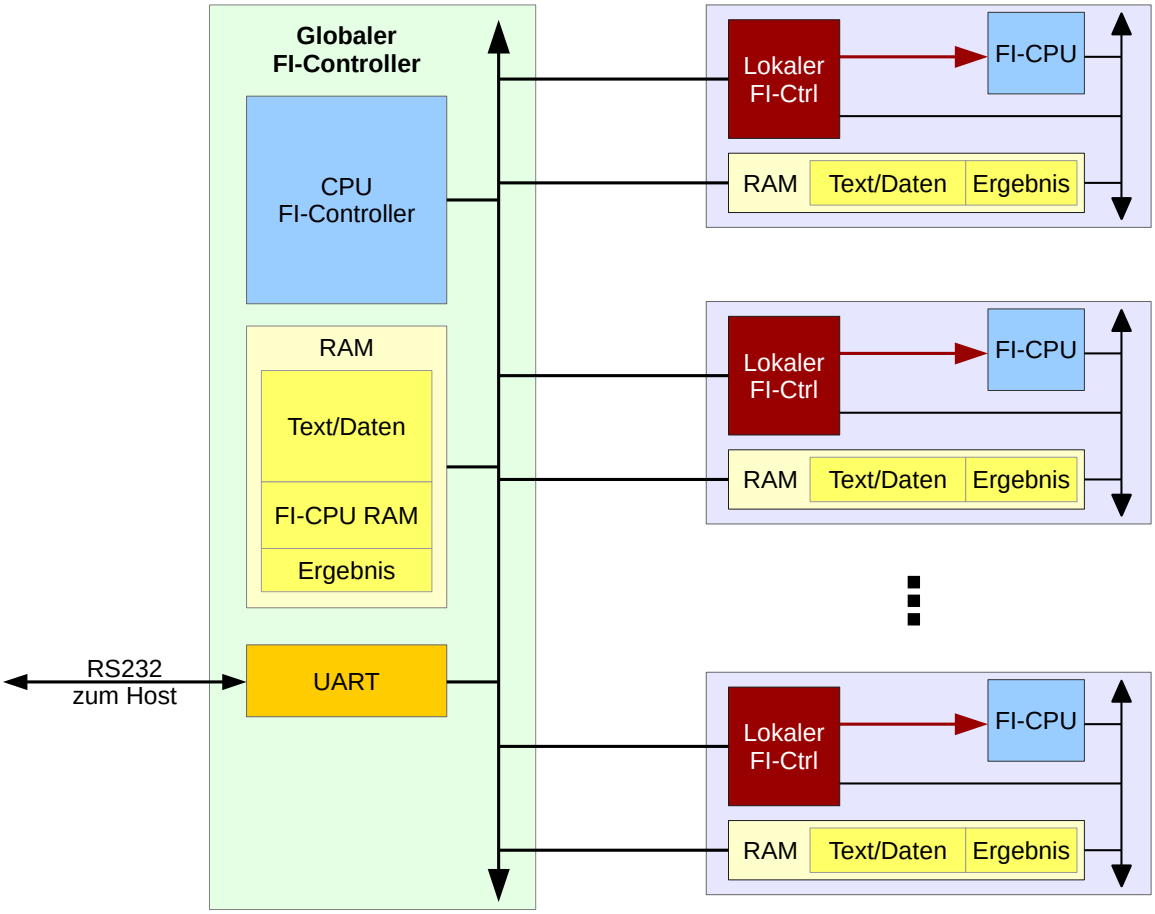

Abbildung 3.3: Architektur des Hardwareentwurfs für die Fehlerinjektionsplattform

Abbildung 3.3 zeigt die entwickelte Architektur des Fehlerinjektionssystems. Auf der linken Seite ist die globale Controller-Einheit zu erkennen, welche die Fehlerinjektionskampagnen verwaltet. Diese Einheit hat eine eigene CPU, sodass der Ablauf der Kampagnen in Software beschrieben werden kann. Die Umsetzung mit einer separaten Controller-CPU wurde gewählt, da es sehr aufwändig wäre, diesen Vorgang in Hardwarestrukturen umzusetzen. Der Arbeitsspeicher der CPU enthält neben den Daten und Instruktionen für die Controller-Software noch Platz für den initialen Speicher der Fehlerinjektions-CPUs (*FI-CPUs*), in die während der Kampagne Fehler injiziert werden. Außerdem ist ein Speicherbereich für das "goldene" Ergebnis eines Benchmarks reserviert, welches man erhält, wenn ein Benchmark fehlerfrei abläuft. Der globale FI-Controller kann mittels einer UART-Schnittstelle mit einem Hostcomputer kommunizieren und über diese beispielsweise Befehle entgegen nehmen oder den initialen FI-CPU-Speicher in die Fehlerinjektionsplattform laden.

Auf der rechten Seite von Abbildung 3.3 sind mehrere Komponenten zu sehen, in die Fehler injiziert werden können. Jede dieser FI-Komponente besitzt eine CPU mit dem dazugehörigen Speicher sowie einen lokalen Fehlerinjektionscontroller. Der lokale FI-Controller übernimmt die Steuerung eines Fehlerinjektionsexperiments und ist vollständig in Hardwarestrukturen implementiert. Er ist in der Lage, die CPU zu starten, Fehler zu injizieren, und die CPU zu stoppen. Außerdem hat die FI-CPU die Möglichkeit, Rückmeldung an den lokalen FI-Controller zu geben, um so beispielsweise zu signalisieren, dass der zu testende Benchmark terminiert ist. Das Ergebnis des Experiments wird in eine separate Sektion des Arbeitsspeichers geschrieben.

Um eine bessere Performance der Fehlerinjektionsplattform zu erreichen, ist eine variable Anzahl an FI-Komponenten vorgesehen, wobei der Fehlerraum der zu injizierenden Fehler zwischen den Komponenten aufgeteilt wird. Dadurch kann durch die parallele Abarbeitung der Fehlerinjektionsexperimente eine höhere Gesamtgeschwindigkeit erreicht werden.

Die Software des globalen FI-Controllers kann die lokalen FI-Controller mittels Zugriff auf Memory-Mapped Register konfigurieren und die Experimente starten bzw. deren Status auslesen. Außerdem hat der globale FI-Controller Lese- und Schreibzugriff auf den Arbeisspeicher der FI-CPUs. Dies dient dazu, um vor dem Ausführen eines Experiments den Speicher zu initialisieren und um nach dem Beenden des Experiments den Ergebnisspeicher auszulesen und mit dem Ergebnis des Golden Runs zu vergleichen.

Die CPU, in die Fehler injiziert werden, soll Software ausführen können, welche nicht stark an das Fehlerinjektionssystem angepasst werden muss. Sicher wird ein eigener Startupcode sowie ein angepasstes Linkerscript benötigt, da es sich um ein neu entwickeltes System handelt. Die Benchmarks sollen aber normale C oder C++-Programme sein, welche ohne Einschränkungen mit der GNU Compiler Collection kompilierbar sind.

Die auf der FI-CPU laufenden Benchmarks sollen in der Lage sein, bestimmte Rückmeldungen an ihren lokalen FI-Controller zu geben. Zwingend erforderlich ist die Signalisierung, dass die Software terminiert ist. Zusätzlich ist es sinnvoll, dass der Benchmark im Golden Run ein zeitliches Fenster bestimmen kann, in das bei

```
1  module full_adder(a_in, b_in, c_in, s_out, c_out, fi_in);
2
3  wire _00_;
4  wire _01_;
5  wire _02_;
6  wire _03_;
7  wire _04_;
8
9  input a_in;
10 input b_in;
11 input c_in;
12 output c_out;
13 output s_out;
14 input [6:0] fi_in;
15
16 assign _04_  = (a_in ^ b_in) ^ fi_in[0];
17 assign s_out = (_04_ ^ c_in) ^ fi_in[1];
18 assign _00_  = (a_in & b_in) ^ fi_in[2];
19 assign _01_  = (a_in & c_in) ^ fi_in[3];
20 assign _03_  = (_00_ | _01_) ^ fi_in[4];
21 assign _02_  = (b_in & c_in) ^ fi_in[5];
22 assign c_out = (_03_ | _02_) ^ fi_in[6];
23
24 endmodule
```

Abbildung 3.4: Der Code des Gate-Level Volladdierers aus Abbildung 3.2, erweitert um Saboteure auf Basis eines breiten Arrays

den folgenden Experimenten die Fehler injiziert werden. So kann die Fehlerinjektion beispielsweise auf einzelne Funktionsaufrufe beschränkt werden. Insbesondere kann verhindert werden, dass Fehler während der Startupphase injiziert werden. Diese Signalisierung, von der FI-CPU ausgehend, soll durch Zugriff auf Memory-Mapped Register des lokalen FI-Controllers geschehen.

Zudem muss die Möglichkeit bestehen, dass der globale FI-Controller feststellen kann, ob das Ergebnis eines Experiments mit dem Ergebnis des Golden Runs übereinstimmt. Dies geschieht durch einen festen Speicherbereich, in welchen der Benchmark das Ergebnis schreibt. Da der globale FI-Controller Zugriff auf den Speicher der FI-CPU hat, kann er diesen Speicherbereich mit einem Abbild des korrekten Ergebnisses vergleichen.

3.4.1 Implementierung der Gate-Level Saboteure

Damit in die Hardwarestrukturen der FI-CPU Fehler injiziert werden können, muss der HDL-Code um die Saboteure erweitert werden. Hierfür wird ausgehend von einer Gate-Level Beschreibung, wie die auf der rechten Seite in Abbildung 3.2, jede einzelne Zuweisung (`assign`) erweitert, sodass das binäre Ergebnis der Anweisung invertiert werden kann. Dazu müssen dem entsprechendem HDL-Modul Steuersignale

```
1  module full_adder(a_in, b_in, c_in, s_out, c_out, fi_in);
2
3  wire _00_;
4  wire _01_;
5  wire _02_;
6  wire _03_;
7  wire _04_;
8
9  input a_in;
10 input b_in;
11 input c_in;
12 output c_out;
13 output s_out;
14 input [2:0] fi_in;
15
16 assign _04_  = (a_in ^ b_in) ^ (fi_in == 3'b000);
17 assign s_out = (_04_ ^ c_in) ^ (fi_in == 3'b001);
18 assign _00_  = (a_in & b_in) ^ (fi_in == 3'b010);
19 assign _01_  = (a_in & c_in) ^ (fi_in == 3'b011);
20 assign _03_  = (_00_ | _01_) ^ (fi_in == 3'b100);
21 assign _02_  = (b_in & c_in) ^ (fi_in == 3'b101);
22 assign c_out = (_03_ | _02_) ^ (fi_in == 3'b110);
23
24 endmodule
```

Abbildung 3.5: Der Code des Gate-Level Volladdierers aus Abbildung 3.2, erweitert um Saboteure auf Basis von vielen Logikvergleichern

für die Saboteure hinzugefügt werden, welche vom FI-Controller zur Laufzeit geschaltet werden können. Dadurch wird eine Invertierung des Wertes an einer bestimmten Zuweisung ausgelöst.

In Abbildung 3.4 ist der Volladdierer zu sehen, wie er um Saboteure erweitert wurde. Dem Modul wurde das eingehende Signalarray `fi_in` hinzugefügt, welches aus 7 Bits (0 bis 6) besteht (Zeilen 1 und 14). Die Anzahl der Bits dieses Arrays entspricht der Zahl der Zuweisungen, in welche Fehler injiziert werden sollen. Die rechten Seiten der Zuweisungen in Zeile 16 bis 22 wurden um ein XOR mit einem Bit aus diesem Signalarray erweitert. Hat ein Bit dieses Arrays den Wert `1`, wird das Ergebnis der dazugehörigen Zuweisung verfälscht. Die Bitwerte dieses Arrays werden vom lokalen FI-Controller in Abhängigkeit von der Experimentkonfiguration gesetzt.

In Abbildung 3.5 ist eine andere Implementierung der Saboteure für den Volladdierer zu sehen. Hier enthält das Signalarray `fi_in` bei n Zuweisungen lediglich $\lceil \log_2 n \rceil$ Bits. Dafür fällt die Erweiterung der Zuweisungen komplexer aus. Hier wird bei jeder Erweiterung der `assign`-Anweisung überprüft, ob die Bits des Eingangssignals einem eindeutigen Bitmuster entsprechen.

Obwohl die Saboteure in den Abbildungen 3.4 und 3.5 scheinbar dieselbe Wirkung erzielen, gibt es große Unterschiede zwischen beiden Implementierungen der Fehlerinjektion. Zunächst einmal ist es offensichtlich, dass es bei der ersten Variante mit dem

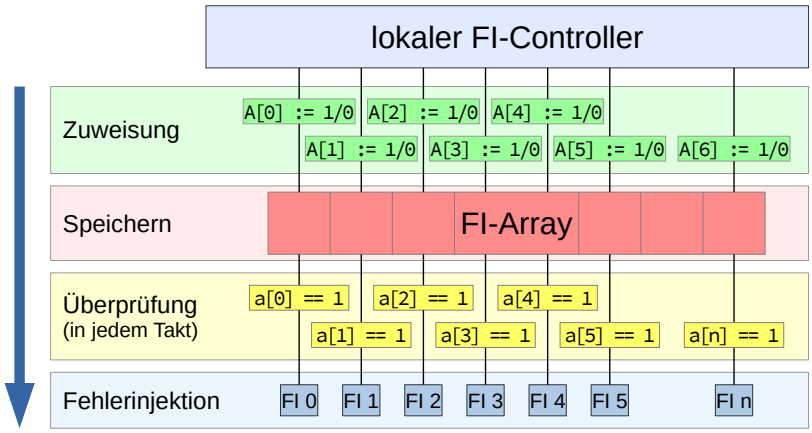

Abbildung 3.6: Darstellung der benötigten Komplexität zur Fehlerinjektion in den einzelnen Abschnitten, auf Basis eines breiten FI-Arrays mit einfacher Überprüfungslogik

größeren Array möglich ist, an mehreren Stellen gleichzeitig einen Fehler zu injizieren. Dies ist bei der zweiten Variante nicht möglich, da hier das ganze Array mit einem eindeutigen Bitmuster verglichen wird, welches für jeden Injektionsort verschieden ist. Somit ist beispielsweise eine Fehlerinjektion in ein ganzes Byte eines CPU-Registers mit der zweiten Variante nicht möglich.

Darüber hinaus gibt es bei der FPGA-Synthese beider Verfahren sehr große Unterschiede. In Abbildung 3.6 wird die Komplexität der einzelnen Teile der zuerst vorgestellten Saboteurimplementierung aus Abbildung 3.4 beschrieben. Durch die hohe Bitzahl im `fi_in`-Array werden viele Flip-Flops auf dem FPGA benötigt. Die benötigte Logik für jeden Saboteur ist sehr einfach, da nur ein Bit des Arrays überprüft werden muss. Dafür fällt die Zuweisungslogik, welche die Bits des `fi_in`-Arrays setzt, umso komplexer aus. Die Komplexität der zweiten Saboteurimplementierung aus Abbildung 3.5 ist in Abbildung 3.7 grafisch dargestellt. Durch die geringere Anzahl an Bits werden auf dem FPGA weniger Flip-Flops benötigt und die Zuweisungslogik fällt entsprechend einfacher aus. Dafür müssen jedoch bei jedem Saboteur $\lceil \log_2 n \rceil$ Bits überprüft werden, sodass an dieser Stelle viel Hardwarelogik in Form von Gates erforderlich ist.

Um eine möglichst hohe Taktrate des finalen Systems zu erreichen, muss darauf geachtet werden, dass zwischen den Flip-Flops auf dem FPGA möglichst wenige hintereinander geschaltete Gates liegen, da jedes Gate eine gewisse Schaltverzögerung besitzt. Abbildung 3.8 zeigt beispielhaft die Schaltlatenz des Volladdieres auf der rechten Seite in Abbildung 3.2 bei einer Schaltgeschwindigkeit von $10ns$ pro Gate bzw. Zuweisungszeile. Es wird zum Zeitpunkt $0ns$ an die Eingänge `a_in` sowie `b_in` eine 1 angelegt. Es ist zu sehen, dass das korrekte Ergebnis erst nach $30ns$ an den Ausgängen `c_out` und `s_out` anliegt.

Die originale CPU ohne Saboteure besitzt zwischen ihren Flip-Flops, in denen zum Beispiel die CPU-Registerinhalte gespeichert sind, Logikbausteine, welche beispiels-

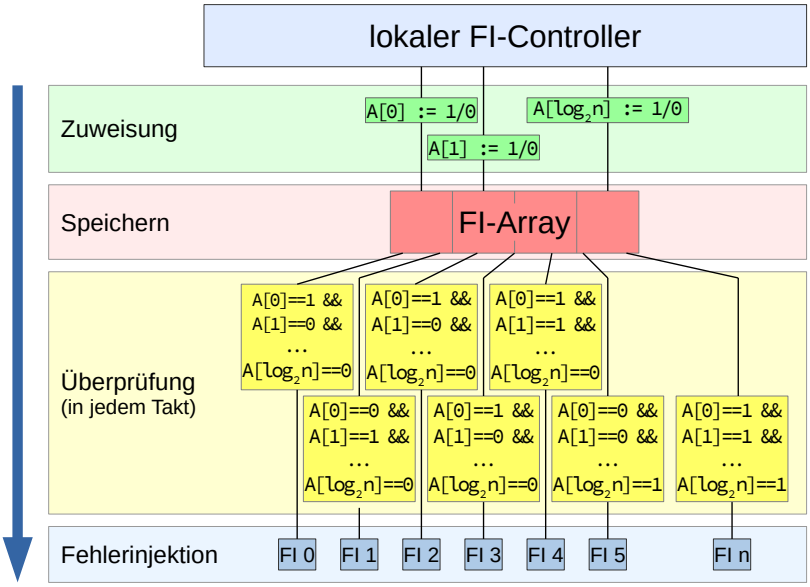

Abbildung 3.7: Darstellung der benötigten Komplexität zur Fehlerinjektion in den einzelnen Abschnitten, auf Basis eines kleinen FI-Arrays mit komplexer Überprüfungslogik

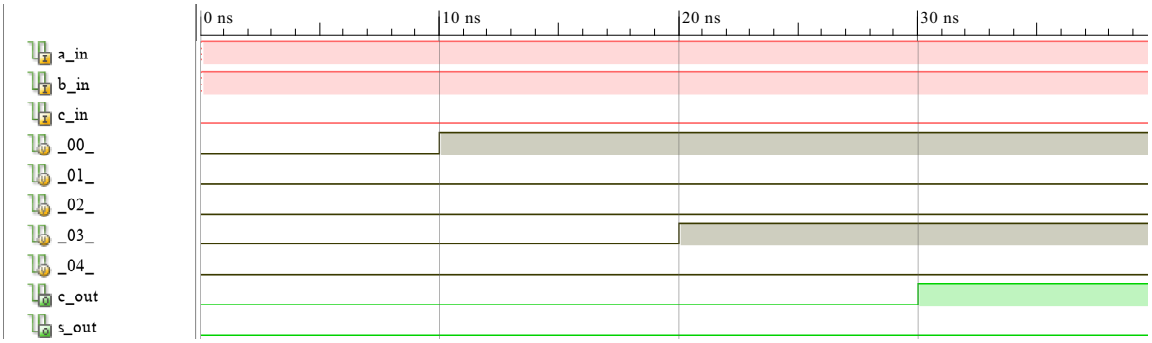

Abbildung 3.8: Simulierte Schaltlatenzen des Gatelevel-Volladdieres aus Abbildung 3.2 bei 10 ns Schaltzeit pro Gate

weise eine Addition zweier CPU-Register durchführen. Dafür müssen bereits viele Gates hintereinander geschaltet werden, was zu einer entsprechenden Verzögerung führt.

Für die Implementierung einer Fehlerinjektion muss an jedem einzelnen Gate entweder ein (erste Saboteurimplementierung aus Abbildung 3.4) oder mehrere (zweite Saboteurimplementierung aus Abbildung 3.7) Gates hinzugefügt werden, sodass sich die gesamte Schaltverzögerung der Logik weiter erhöht. Um die durch die Fehlerinjektion zusätzliche Schaltlatenz so gering wie möglich zu halten, bietet es sich daher an, die erste Methode zur Implementierung der Saboteure zu benutzen. Diese hat den Nachteil, dass viele Flip-Flops benötigt werden. Ein FPGA stellt generell viele Flip-Flops bereit, und die anderen Komponenten der Fehlerinjektionsplattform benötigen nur wenige von diesen, sodass dieser hohe Flip-Flop-Verbrauch an dieser Stelle akzeptabel ist, wenn das gesamte System auf das FPGA passt. Bei der ersten Implementierungsmethode ist mehr Aufwand bei der Zuweisung des Arrays notwendig, aufgrund der sehr einfachen Logik im Überprüfungsschritt ist jedoch zu erwarten, dass der gesamte Bedarf an Logikkomponenten bei der ersten Saboteurtechnik kleiner ist, als bei der zweiten.

3.4.2 Gate-Level-Modell der Flip-Flops

Die grundlegende Umsetzungsweise zur Fehlerinjektion in Hardwarelogik bestehend aus Gates wurde im vorherigen Kapitel 3.4.1 erläutert. Gesondert betrachten muss man jedoch die Flip-Flops, welche sowohl in der HDL-Beschreibung als auch auf dem FPGA eine Sonderstellung einnehmen, auf einem echten Prozessor jedoch auch aus Gates bestehen. Um auch für Flip-Flop-Schaltungen eine Fehlerinjektion zu ermöglichen, muss daher ein separates Gate-Level-Modell der benutzten Flip-Flops erstellt werden.

FPGAs unterscheiden zwischen der umgesetzten Logik und den Flip-Flops als Speicherelementen. Die Gate-Logik wird auf eine *Look-Up Table* abgebildet, während die Speicherkomponenten in real auf dem FPGA existierenden Flip-Flop-Schaltungen untergebracht werden (vgl. [38]). Da diese fix in den *Slices*[3] vorhanden sind, ist es nicht möglich, in einzelne Gates dieser Flip-Flops Fehler zu injizieren.

Eine Lösungsmöglichkeit wäre die Transformation der CPU in eine asynchrone Schaltung, bei der die Taktleitung der CPU keine Sonderstellung (als Takt) einnimmt, sondern als weiteres, normales Signal der zu realisierenden Logik behandelt wird. Die Flip-Flops würden dann durch entsprechende (asynchrone) Logikschaltungen ersetzt. Hier besteht das Problem, dass FPGAs generell für synchrone Schaltungen ausgelegt sind. Zwar sind asynchrone Designs prinzipiell möglich, jedoch sind die Timing-Eigenschaften der resultierenden FPGA-Konfiguration nur schwer vorherzusagen, da insbesondere die Unterstützung der Synthesetools für komplexe asynchrone Schaltungen fehlt.

[3] Gruppierung mehrerer LUTs und Flip-Flops auf einem FPGA

Eine andere Lösung, welche im Rahmen der Entwicklung umgesetzt wurde, ist das modellhafte Abbilden der Auswirkungen der transienten Fehler in den Flip-Flops. Dazu wurde ausgehend vom Gate-Aufbau evaluiert, welche Folgen die Fehler in einzelnen Gates auf das Flip-Flop als ganzes haben. Dieses Fehlermodell wurde anschließend in die CPU eingefügt.

Abbildung 3.9 zeigt ein ausschließlich mittels NAND-Flip-Flops umgesetztes vorderflankengesteuertes D-Flip-Flop. Dieses Flip-Flop erfüllt die benötigten Anforderungen für die Speicherzellen der ARM Cortex-M0-CPU.

Zur Analyse der Auswirkungen von transienten Fehlern in den einzelnen Gates wurde eine Simulationsumgebung erstellt, in der die Schaltung asynchron nachgebildet wurde und Fehler zum Zeitpunkt der positiven Taktflanke injiziert wurden. Anhand dessen wurde beobachtet, welche Auswirkungen diese Fehler haben. Das Ergebnis ist in Tabelle 3.4.2 dargestellt. Es ist zu sehen, dass es insgesamt 5 verschiedene Fehlerauswirkungen gibt.

Die Timingfehler wurden für den weiteren Verlauf dieser Arbeit ignoriert, da diese auch schon bei den Gates nicht betrachtet wurden. Die Fehlerauswirkungen $Q' = 1$, $Q' = 0$ und $Q' = \neg D$ sind einfach zu implementieren. Besondere Aufmerksamkeit muss den Fehlern gewidmet werden, bei denen der Ausgang Q' undefiniert ist. Der Grund für den Nichtdeterminismus ist, dass während der Zeit des Fehlers ein ungültiger Eingangszustand am entsprechenden *Latch* anliegt, sodass der Status mit der maximalen Schaltgeschwindigkeit der Gates wechselt. In der Realität ist der Ausgangswert abhängig von der Zeit, in welcher der Fehler anliegt sowie von der Schaltgeschwindigkeit.

Dieser Nichtdeterminismus ist mit einem FPGA nicht darstellbar, sodass in diesem Fehlerfall ein gleichverteilter Zufallswert an den Ausgang des Flip-Flops angelegt wird.

Um Hardware-Logikzellen zu sparen, wurde für jede Fehlerauswirkung nur genau ein Saboteur implementiert. Um dennoch eine Gleichverteilung der Gate-Fehler zu erhalten, ist eine Gewichtung erforderlich, damit die Häufigkeiten der auftretenden Fehler zu den auslösenden Gates passen. Diese Gewichtung wird in der Software des globalen FI-Controllers implementiert, indem für jedes Flip-Flop 8 Gates dem Fehlerraum hinzugefügt werden, wobei mehrere Fehlerorte derselben Fehlerauswirkung und somit demselben Saboteur zugewiesen werden können.

3.4.3 Fehlerinjektion in Flip-Flops

In diesem Modell soll zu bestimmten Zeitpunkten der digitale Wert eines Flip-Flops invertiert werden. Das im vorherigen Kapitel 3.4.2 entwickelte Fehlermodell für die Gate-Level Fehlerinjektion in Flip-Flops enthält bereits die für dieses Modell erforderliche Fehlerauswirkung, das Negieren des Eingangwertes des vorherigen Takts (Nummer 7 in Tabelle 3.4.2, $Q' = \neg D$). Dadurch ist es für die Umsetzung der Flip-Flop-Fehlerinjektion lediglich erforderlich, den Raum der Fehlerpunkte auf genau diesen Ausgang der Gate-Level-Injektion eines jeden Flip-Flops zu beschränken und gleichverteilt in diese Fehlerpunkte zu injizieren.

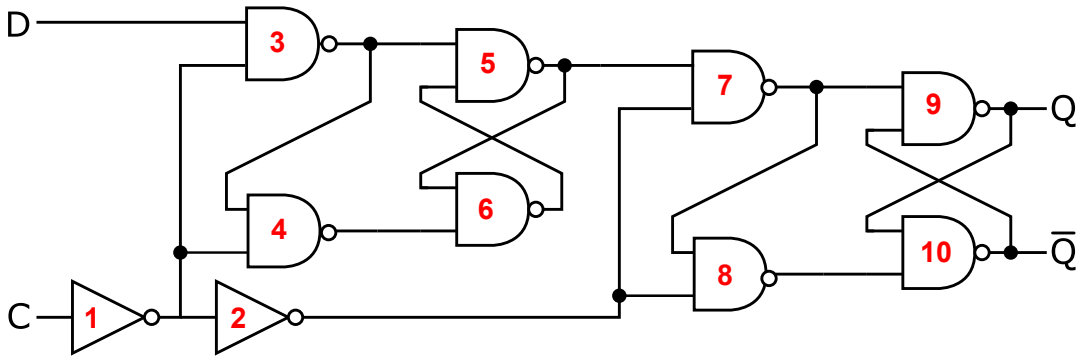

Abbildung 3.9: Gate-Level-Modell eines vorderflankengesteuerten D-Flipflops (Quelle: [39])

Gate	Auswirkung
1	Timingfehler
2	Timingfehler
3	$Q' = 1$
4	$Q' = 0$
5	Q' undefiniert
6	Q' undefiniert
7	$Q' = \neg D$
8	$Q' = D$
9	Q' undefiniert
10	Q' undefiniert

Tabelle 3.2: Auswirkung eines transienten Gate-Fehlers zum Zeitpunkt der positiven Taktflanke bei einem vorderflankengesteuerten D-Flip-Flop

3.4.4 Fehlerinjektion auf ISA-Level

Die Fehlerinjektion auf Basis der Instruction Set Architecture betrifft diejenigen Flip-Flops der CPU-Schaltung, welche den Wert eines CPU-Registers speichern. Bei der gewählten ARM Cortex-M0-CPU ist es leider nicht offensichtlich, welche Flip-Flops dies sind, da der Quellcode verschleiert ist. Da jedoch die Diagnosesignale der CPU die Registerinhalte ausgeben, ist es möglich, die Schaltung zu analysieren, um die Abbildung der Flip-Flops auf die Register herauszufinden.

Generell ist die Fehlerinjektion in die CPU-Register wie schon die Flip-Flop-Injektion eine Beschränkung des Fehlerraums auf bestimmte Gate-Level Fehlerinjektionen und somit leicht umsetzbar, sobald das Abbild der einzelnen Flip-Flops auf die Bits der CPU-Register feststeht. Im Gegensatz zu den vorherigen Modellen sollen jedoch nicht ausschließlich uniform verteilte Einzelbitfehler untersucht werden, sondern auch

Mehrbitfehler sowie Einzelbitfehler bei Schreibzugriffen auf die Register. Beides sind in der Forschung Modelle, welche häufig neben den Einzelbitfehlern betrachtet werden.

Die Umsetzung von Mehrbitfehlern erfordert, dass die Möglichkeit besteht, zu einem Zeitpunkt in mehrere Fehlerorte gleichzeitig zu injizieren. Daher muss hierfür zwingend die erste in Kapitel 3.4.1 vorgestellte Saboteurmethode benutzt werden. Für die Injektion bei Schreibzugriffen ist eine etwas andere Fehlerinjektionslogik erforderlich, welche einen bestimmten Schreibzugriff auf ein Registerbit abwartet, um bei diesem Schreibvorgang den geschriebenen Wert zu invertieren.

3.4.5 Parallelisierung

Fehlerinjektionsexperimente lassen sich sehr gut parallelisieren, indem mehrere Experimente gleichzeitig ausgeführt werden. Im Falle dieser Plattform erfordert dies mehrere CPUs, in die Fehler injiziert werden können. Allerdings ist davon auszugehen, dass die für die Saboteure benötigte Logik den Ressourcenbedarf einer gesamten CPU auf dem FPGA vervielfacht. Vermutlich gibt es einen linearen Zusammenhang zwischen dem Platzbedarf und der Anzahl der Saboteure. Um Ressourcen zu sparen, kann man den Fehlerraum partitionieren, sodass nicht mehr jede FI-CPU für jeden Fehlerort einen Saboteur erhält. Für jeden auf der ARM Cortex-M0-CPU vorhandenen Fehlerort muss jedoch eine FI-CPU im System vorhanden sein, welche in der Lage ist, ein Fehlerinjektionsexperiment mit einem bestimmten Fehlerort auszuführen.

Pro zusätzlicher FI-CPU werden durch die eigentliche Prozessorlogik und dem lokalen FI-Controller zusätzliche Ressourcen auf dem FPGA benötigt. Dafür ist jedoch die in jeder FI-CPU benötigte Logik für die Fehlerinjektion umso geringer, je mehr Partitionen vorhanden sind. Es ist davon auszugehen, dass dadurch der zuerst genannte Logikzuwachs kompensiert wird.

Ein Nachteil der Partitionierung ist, dass sich alle FI-CPUs den auf dem FPGA vorhandenen SRAM aufteilen müssen, da sie diesen als Speicher benutzen. Ein anderer ist der erhöhte Mehraufwand in der Steuersoftware des globalen FI-Controllers.

Da der gesamte Ressourcenmehrbedarf durch die Partitionierung sowie der benötigte Speicherbedarf der auf der Plattform laufenden Benchmarks unklar ist, soll die Anzahl der FI-CPUs bei der Implementierung flexibel gehalten werden. Der Parallelisierungsgrad kann solange erhöht werden, wie die FPGA-Ressourcen nicht vollständig aufgebraucht sind und der Speicher ausreicht.

3.5 Kampagnenablauf

Der globale FI-Controller verwaltet den Ablauf einer Fehlerinjektionskampagne. Da dies ein komplexer Vorgang ist, soll diese Verwaltung in Software programmierbar sein und nicht mittels reiner Hardwarelogik umgesetzt werden.

Der grobe Ablauf einer Kampagne ist in Abbildung 3.10 dargestellt. Zunächst wird der Golden Run ausgeführt. Dieser dient dazu, für das Experiment benötigte Infor-

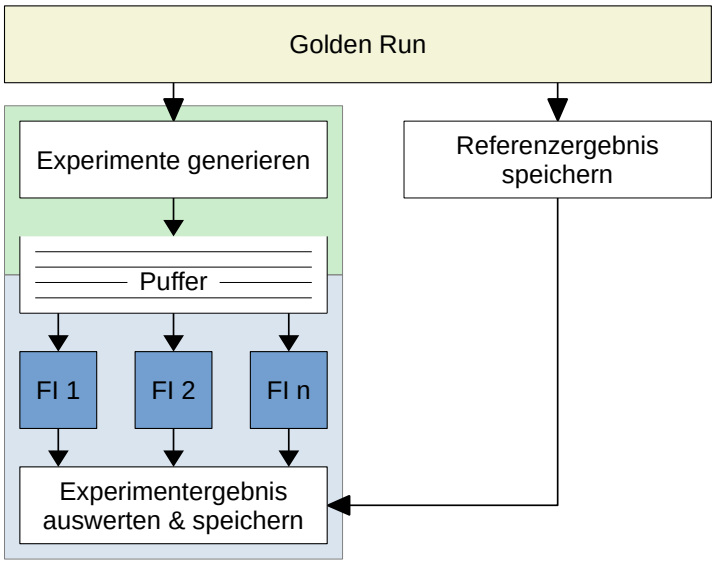

Abbildung 3.10: Ablauf einer Fehlerinjektionskampagne

mationen eines fehlerfreien Benchmark-Duchlaufs zu sammeln. Es wird die gesamte Laufzeit bis zur Terminierung sowie das Zeitfenster der Fehlerinjektion festgestellt, um daraus im folgenden Schritt die Experimente zu generieren. Außerdem wird das Ergebnis des Benchmarks im Golden Run als Referenzergebnis gespeichert, um dieses später mit dem Benchmarkergebnis nach einem Fehlerinjektionsexperiment zu vergleichen.

Die Experimentparameter werden anhand der gesammelten Daten generiert und zunächst in einer Queue gespeichert. Wenn eine FI-CPU im Leerlauf ist, kann ein zu der Partitionierung der CPU passender Experimentdatensatz aus der Queue entnommen und ausgeführt werden. Es werden bewusst keine für eine im Leerlauf befindliche FI-CPU passende Experimentparameter generiert, da dies zu einer Verzerrung der statistischen Ergebnisse führen könnte.

Nach Beendigung des Experiments durch Terminierung des Benchmarks wird der Ergebnisspeicher der FI-CPU mit dem Referenzergebnis verglichen, um festzustellen, ob es zu einer Datenkorruption gekommen ist. Wird das Experiment aufgrund eines Timeouts abgebrochen, entfällt dieser Schritt. Das Gesamtergebnis des Experiments kann nun gespeichert werden und die FI-CPU kann das nächste Experiment ausführen.

3.6 Zusammenfassung

In diesem Kapitel wurde der Entwurf einer Fehlerinjektionsplattform vorgestellt, die CPU-Fehler auf verschiedenen Ebenen erzeugen kann. Zunächst wurden die Anforderungen identifiziert, die für den Entwurf maßgeblich sind. Anschließend wurde festgestellt, dass sich ein FPGA vermutlich gut für die Umsetzung eignet, insbesonde-

re aufgrund der wesentlich höheren Geschwindigkeit gegenüber eines Simulators. Als nächstes wurde die CPU ausgewählt, in die Fehler injiziert werden. Der ARM Cortex-M0 ist in vielerlei Hinsicht besser als andere Alternativen geeignet, aber leider nicht Open-Source. Die Fehlerinjektionsplattform kann durch Saboteure Fehler auf Gate-Ebene injizieren. Die Fehlerorte der Flip-Flop- und ISA-Ebene sind eine Teilmenge der Gate-Level-Fehlerorte und daher einfach zu implementieren. Abschließend wurde der grobe Ablauf einer Fehlerinjektionskampagne entworfen. Im nächsten Kapitel wird nun erläutert, wie der Entwurf der Plattform implementiert wurde und die entwickelten Konzepte der Saboteure auf den ARM Cortex-M0-Prozessor angewandt wurden.

4 Implementierung

In diesem Kapitel wird die Implementierung der Fehlerinjektionsplattform dargelegt. Zunächst werden in Abschnitt 4.1 verschiedene Aspekte der Hardwareimplementierung beschrieben. In Kapitel 4.2 wird die Interaktion zwischen dem Host und dem globalen FI-Controller erläutert. Abschließend wird in Abschnitt 4.3 erklärt, wie eine Fehlerinjektionskampagne auf der Plattform ausgeführt werden kann.

4.1 Implementierung der Hardware

Dieser Abschnitt beschäftigt sich zunächst mit der Implementierung der Saboteure des ARM Cortex-M0-Prozessors, und wie diese die verschieden abstrakten Fehlermodelle implementieren. Anschließend werden Aspekte der Partitionierung, das Speicherlayout und die zur Verfügung stehenden Hardwareregister der lokalen FI-Controller erläutert.

4.1.1 Gate-Level-Fehlerinjektion im ARM Cortex-M0-Prozessor

In Abbildung 4.1 ist ein Ausschnitt des verschleierten Verilog-Codes zu sehen, in dem der ARM Cortex-M0 Prozessor vorliegt. Insgesamt besteht er aus 11109 `assign`-Anweisungen, welche überwiegend eine Gate-Level-Struktur besitzen. Dazu gibt es insgesamt 841 `always @...` Blöcke, in denen Zuweisungen in die von der CPU benutzen Flip-Flops getätigt werden. Diese entsprechen stets dem abgebildeten Schema.

Fast alle Zuweisungen sowie die Blöcke müssen modifiziert werden, um an diesen Punkten eine Fehlerinjektionsmöglichkeit zu implementieren. Dies ist manuell nicht praktikabel, sodass zu diesem Zwecke Skripte geschrieben wurden, welche den originalen Code des ARM Cortex-M0-Prozessors um die Saboteure erweitern. Zunächst werden die für die Steuerung der Fehlerinjektion benötigten Eingangssignale hinzugefügt, anschließend wird der Code modifiziert.

Im Folgenden werden mögliche Formen der `assign`-Zeilen im Prozessorquellcode erläutert und dessen eventuelle Transformation erklärt:

- `assign X = Y;`
  Zeilen dieser Form weisen dem Signal `X` denselben Wert wie Signal `Y` zu. Dies erfordert keinerlei Logik und bedarf damit keiner Erweiterung um eine Fehlerinjektion.

```
wire Mf8vx4, Eg8vx4, Ih8vx4, Ki8vx4, Mj8vx4, Mk8vx4, Ul8vx4, Cn8vx4;
wire Xq8vx4, Fs8vx4, Nt8vx4, Vu8vx4, Dw8vx4, Hx8vx4, Oy8vx4, Uz8vx4;
wire P39vx4, W49vx4, D69vx4, K79vx4, R89vx4, Y99vx4, Fb9vx4, Mc9vx4;
[...]
reg J6i2z4, Z7i2z4, H9i2z4, Wai2z4, Gci2z4, Pdi2z4, Zei2z4, Igi2z4;
reg Tki2z4, Emi2z4, Rni2z4, Fpi2z4, Uqi2z4, Isi2z4, Xti2z4, Mvi2z4;
reg Yzi2z4, M1j2z4, X2j2z4, M4j2z4, B6j2z4, Q7j2z4, F9j2z4, Uaj2z4;
[...]
assign Dbgvx4 = Aok2z4;
assign Srovx4 = (Btovx4 & Itovx4);
assign O9bwx4 = (~(vis_r11_o[3] & Dg9wx4));
assign Oqb2z4 = (Xmsvx4 ? Lcpvx4 : Vqb2z4);
assign Sxpvx4 = (!Zxpvx4);
[...]
always @(posedge hclk or negedge hreset_n)
  if(~hreset_n)
    Gxk2z4 <= 1'b0;
  else
    Gxk2z4 <= T2ivx4;
  end
```

Abbildung 4.1: Codefragmente des ARM Cortex-M0 Prozessors. Oben werden Signale (`wire`) definiert, darunter Flip-Flops (`reg`). Anschließend finden sich Zuweisungen in Gates (`assign ...= `) sowie Zuweisungen in Flip-Flops (`...<= ...`)

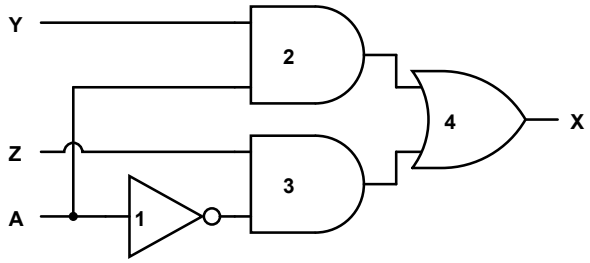

Abbildung 4.2: Gate-Modell eines 2:1-MUX zur bedingten Zuweisung

- `assign X = (Y & Z);`
  Hier werden zwei Signale mittels eine booleschen Funktion verknüpft. Neben dem `&` Operator gibt es noch `|` und `^`. Außerdem können die Signale `Y` und `Z` mittels einem vorgestelltem `!` oder `~` negiert werden. Für die Fehlerinjektion wird eine Zeile dieses Schemas erweitert zu:

  ```
  1 assign X = (Y ^ fi_in[i]);
  ```

  Eine eventuelle Negation vor den Variablen bleibt vorhanden und der Operator wird übernommen.

- `assign X = (!Y);`
  Diese Negation des Signals `Y` wird zu:

  ```
  1 assign X = (!Y) ^ fi_in[i];
  ```

- `assign X = (A ? Y : Z);`
  Diese bedingte Zuweisung (*2:1-MUX*) lässt sich nicht mit einem einzigen Gate beschreiben, sondern benötigt, wie in Abbildung 4.2 zu sehen, insgesamt vier. Dementsprechend gibt es auch vier Fehlerinjektionspunkte. Aus einer Zeile mit einer bedingten Zuweisung wird daher folgender Code generiert:

  ```
  1 assign X = (
  2  (fi_in[i + 0] == 1'b1) ?  (A & Y) |  ( A & Z) : // Fehler in 1
  3  (fi_in[i + 1] == 1'b1) ? !(A & Y) |  (!A & Z) : // Fehler in 2
  4  (fi_in[i + 2] == 1'b1) ?  (A & Y) | !(!A & Z) : // Fehler in 3
  5  (fi_in[i + 3] == 1'b1) ? !(A ? Y : Z) : // Fehler in 4
  6  (A ? Y : Z) // Keine Fehlerinjektion
  7 );
  ```

  Manche bedingte Zuweisungen dienen dazu, einem Flip-Flop, welches ein Bit eines CPU-Registers speichert, einen neuen Wert zuzuweisen. In diesem Fall wird der obige Code noch durch eine weitere Zeile ergänzt:

  ```
  1 (A && register_write_fi_register[1] == 1'b1 &&
       register_write_fi_bit[8] == 1'b1) ? (!( X )) :
  ```

  Hier bestimmt das Signal `A`, ob ein neuer Wert in das Register geschrieben wird. Das Signal wurde durch Zurückverfolgen der nur lesbaren Debug-Signale der ARM Cortex-M0-CPU ermittelt. Details dazu werden in Abschnitt 4.1.3

erläutert. Die obige Zeile überprüft, ob vom Register `r1` das Bit 8 bei einem Schreibzugriff invertiert werden soll.

4.1.2 Flip-Flop-Fehlerinjektion

Blöcke, welche im Originalcode der CPU die Zuweisungen der Flip-Flops tätigen, haben alle die folgende Struktur:

```
always @(posedge hclk or negedge hreset_n)
    if(~hreset_n)
        FF_X <= 1'b0;
    else
        FF_X <= GATE_Y;
    end
```

Um die vier Auswirkungen von Gate-Fehlern, die in Kapitel 3.4.2 beschrieben wurden, zu implementieren, wird der obige Block durch den folgenden Code ersetzt:

```
always @(posedge hclk or negedge hreset_n)
    if(~hreset_n)
        FF_X <= 1'b1;
    else
    begin
        if (fi_in[i + 0] == 1'b1) begin
            FF_X <= 1'b1; // Q' = 1
        end
        else if (fi_in[i + 1] == 1'b1) begin
            FF_X <= 1'b0; // Q' = 0
        end
        else if (fi_in[i + 2] == 1'b1) begin
            FF_X <= ~GATE_Y; // Invertierung
        end
        else if (fi_in[i + 3] == 1'b1) begin
            FF_X <= RANDOM_IN; // Undefinierter Wert
        end
        else FF_X <= GATE_Y; // Keine Fehlerinjektion
    end
```

Zusätzlich zu den vorgestellten Transformationen, welche sich gut automatisieren lassen, gibt es im Originalcode noch einige wenige Zeilen der Form

```
assign {s1, s2, s3, s4 } = {a1, a2, a3, a4} + {b1, b2, b3 b4};
```

In dieser Zeile werden zunächst die Signale `a1 - a4` sowie `b1 - b4` zu jeweils einem Array konkateniert. Diese beiden Arrays werden addiert und in einem Array gespeichert, welches sich aus den Signalen `s1 - s4` zusammensetzt. Diese Schreibweise der Addition im sonst verschleierten Quellcode der CPU hat den Vorteil, dass eine FPGA-Synthesesoftware in der Lage ist, solche Additionen auf die eventuell vorhandenen Addierer-Bausteine zu konfigurieren, die sich auf dem FPGA befinden.

Um Fehler in diese Additionsschaltungen zu injizieren, wurden passende *Carry-Look-Ahead* (*CLA*) -Addierer und -Subtrahierer entwickelt, mittels Yosys in ein Gate-Level-Format transformiert und um Fehlerinjektion erweitert (analog zu dem in den

```
1  wire Ndivx4 [...] Zz1wx4 [...] Ax1wx4 [...] ;
2  reg Asr2z4 [...] ;
3
4  assign vis_r1_o[8] = Asr2z4;
5  assign Ndivx4 = (Ax1wx4 ? vis_r1_o[8] : Zz1wx4);
6
7  always @(posedge hclk or negedge hreset_n)
8      if(~hreset_n)
9          Asr2z4 <= 1'b1;
10     else
11         Asr2z4 <= Ndivx4;
```

Abbildung 4.3: Zuweisungslogik eines CPU-Registerbits im ARM Cortex-M0 Quellcode (ohne Saboteure)

Kapiteln 3.3 und 3.4.1 vorgestellten Volladdierer). Eine Zeile wie die obige wird dann durch eine Instanz der entsprechenden Schaltung ersetzt.

4.1.3 Fehlerinjektion auf ISA-Modellebene

Die einzelnen Bits der CPU-Register sind in Flip-Flops gespeichert, in welche, wie oben beschrieben, Fehler injiziert werden können. Problematisch ist bei dem ARM Cortex-M0 jedoch die Zuordnung der Registerbits zu den Flip-Flops, da der HDL-Code der CPU verschleiert ist. Es existieren jedoch bekannte, nur lesbare Signale, an denen die Werte der aktuellen Bits der Registerbank anliegen. Diese dienen Debug-Zwecken. Diese Signale sind zurückverfolgbar und somit ist das Flip-Flop, welches ein bestimmtes Registerbit speichert, ermittelbar. Außerdem kann die Logik weiter zurückverfolgt werden, um Schreibzugriffe auf die Register festzustellen. Dies wird für die Register-On-Write-Fehlerinjektion benötigt.

Abbildung 4.3 zeigt die relevanten Codeausschnitte der CPU, welche den Wert des Bit 8 vom Register 1 setzt. In Abbildung 4.4 ist dieser Prozess grafisch aufbereitet. Es ist zu sehen, dass der Wert im Flip-Flop `Asr2z4` so lange erhalten bleibt, wie das Signal `Ax1wx4` den Wert `0` hat. Ansonsten wird der Wert, der zur entsprechenden Taktflanke in `Zz1wx4` anliegt, in dieses Flip-Flop bzw. das CPU-Register geschrieben.

Für das Register-On-Write Fehlermodell wurden der Hardware außerdem Zähler hinzugefügt, welche die Anzahl der Takte zählen, in denen die Schreibsignale (wie hier `Ax1wx4`) den Wert `0` annehmen und dementsprechend ein neuer Wert in das jeweilige Flip-Flop übernommen wird. Zur Register-On-Write Fehlerinjektion wird dann bei dem Zählerwert $i-1$ das entsprechende Signal gesetzt (siehe Kapitel 4.1.1), damit bei dem *nächsten* Schreibvorgang auf dieses Bit der Wert invertiert wird.

Das Prinzip der Registerzuweisung ist bei allen CPU-Registerbits identisch zu dem hier vorgestellten. Dadurch kann mittels automatischer Verarbeitung des Quellcodes der ARM Cortex-M0-CPU ein Mapping zwischen den Registerbits und den Flip-Flops erstellt werden. Davon ausgehend können anschließend die zu den Registerbits passenden Saboteure abgeleitet werden. Mehrere Bits eines Registers teilen sich außerdem

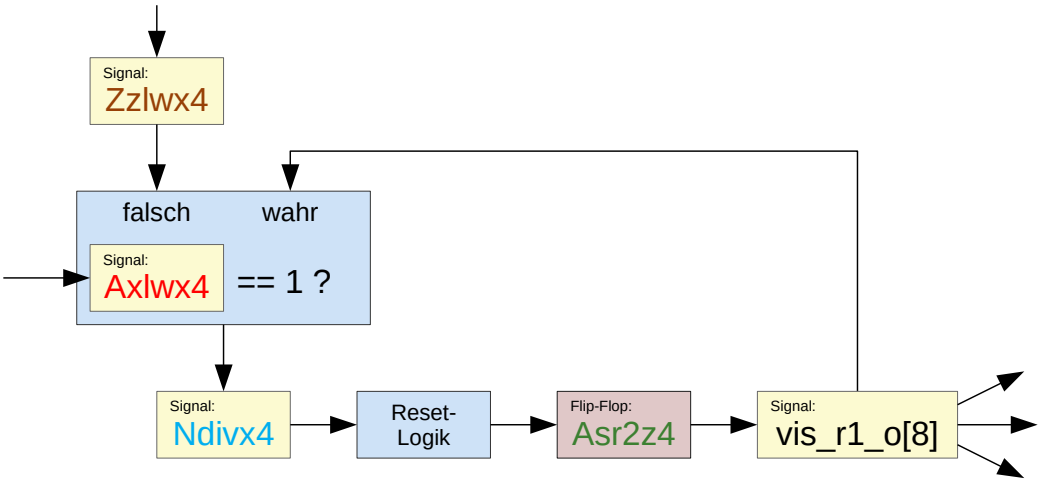

Abbildung 4.4: Grafische Darstellung der Registerbit-Zuweisungslogik aus Abbildung 4.3

ein Schreibsignal, welches bestimmt, ob ein neuer Wert in das Flip-Flop übernommen werden soll. Dadurch ist für das Register-On-Write-Fehlermodell nicht für jedes Registerbit ein separater Zähler notwendig.

4.1.4 Partitionierung der Saboteure

Im vorherigen Abschnitt wurden Techniken erläutert, um die ARM Cortex-M0-CPU um Saboteure zu erweitern, damit in jedes Gate (und darauf aufbauend Flip-Flop & CPU-Register) Fehler injiziert werden können. Um parallel so viele Fehlerinjektionsexperimente wie möglich ausführen zu können, sollten möglichst viele FI-CPUs vorhanden sein. Um die Anzahl der CPUs, die auf das FPGA passen, zu erhöhen, ist es erforderlich, den Ressourcenbedarf jeder FI-CPU so gering wie möglich zu halten. Dies kann erreicht werden, indem die Saboteure auf die zur Verfügung stehenden FI-CPUs partitioniert werden. Es steht dadurch für jedes Fehlerinjektionsexperiment, welches stets einen bestimmten Saboteur benötigt, genau eine FI-CPU bereit, welche diesen Saboteur besitzt.

Im vorangegangenen Abschnitt wurde bei der Erweiterung der CPU um die Saboteure keine Rücksicht auf die geplante Partitionierung genommen. Der Grund liegt darin, dass stets derselbe Quellcode für die FI-CPU benutzt werden soll, um später die Konfigurierbarkeit der Partitionierung zu erleichtern.

Um einer FI-CPU nur bestimmte Saboteure hinzuzufügen, werden daher die Optimierungsfunktionen der Synthesesoftware ausgenutzt. Wenn einem Signal ein fester Wert zugewiesen wird, erkennt die Synthesesoftware, dass an dem Eingang einer Logikstruktur stets derselbe Wert anliegt und vereinfacht diese Struktur entsprechend. Nachteile sind der vermutlich erhöhte Zeitaufwand bei der Synthese sowie die Ausgabe einer Warnung, da dies auf einen schlechten Programmierstil hinweist. Für die

Registerbezeichnung	Adresse	Zugriff bei...
CPU_DONE	`0x40001000`	Terminierung des Programms
FAULT_DETECTED	`0x40001004`	Fehler detektiert
FI_START	`0x40001008`	Beginn des Fehlerinjektionsfensters
FI_STOP	`0x4000100C`	Ende des Fehlerinjektionsfensters

Tabelle 4.1: Adressen der Register der lokalen FI-Controller

Partitionierung wird dies jedoch explizit ausgenutzt, um so bestimmte Saboteure aus der synthetisierten FI-CPU zu entfernen.

Die Umsetzung erfolgt, indem bei der Adressierung der Elemente des breiten Arrays, welches die Saboteure steuert, den niederwertigen Bits ein fester Wert zugewiesen wird, abhängig von der Instanznummer der FI-CPU. Daraus folgt, dass immer 2^n Partitionen angelegt werden müssen und alle Partitionen annähernd gleich viele Saboteure enthalten. Die Verteilung der Injektionspunkte auf die FI-CPUs muss bei der Entwicklung der Software für den globalen FI-Controller berücksichtigt werden, damit diese die Experimente auf der korrekten FI-CPU ausführt.

Konfiguriert wird die Anzahl der Partitionen in der Datei `config.v` des Verilog-Quellcodes der Fehlerinjektionsplattform durch Einkommentieren der entsprechenden Zeile. Anschließend muss die Hardware neu synthetisiert werden. Insgesamt können 1, 2, 4, 8, 16 oder 32 Partitionen konfiguriert werden. In dieser Datei ist auch der Speicher, welcher dem globalen FI-Controller sowie den FI-CPUs zur Verfügung steht, einstellbar.

Sehr wichtig ist, dass keine Partitionierung der Saboteure möglich ist, wenn Fehler an mehreren Orten gleichzeitig injiziert werden sollen. Hierfür müssten zwingend alle zusammen injizierten Fehlerorte auf derselben FI-CPU liegen, was nicht generell garantiert werden kann. Für die Ausführung dieser Arbeit betrifft dies lediglich das ISA-Fehlermodell, bei dem die Werte eines ganzen Bytes der CPU-Register invertiert werden. Bei der Ausführung von Fehlerinjektionsexperimenten nach diesem Modell muss daher die Partitionierung deaktiviert sein.

4.1.5 Speicherlayout und Controllerregister

Dieses Kapitel erläutert das entstandene Speicherlayout der verschiedenen CPUs der Fehlerinjektionsplattform. Grundlegend muss zwischen zwei CPU-Typen unterschieden werden: Einerseits die CPU des globalen FI-Controllers und andererseits die FI-CPUs.

Beide gemeinsam besitzen ab Adresse `0x0` einen SRAM-Speicher konfigurierbarer Länge. Dieser enthält die für den Betrieb der jeweiligen CPU benötigten Instruktionen, Daten und genügend Platz für Stack sowie einen eventuellen Heap.

Startadresse	Speicherbereich
0x00000000	Lokaler Hauptspeicher
0xA0000000	Hauptspeicher der ersten FI-CPU
0xA0800000	Adressbereich des ersten lokalen FI-Controllers
0xA1000000	Hauptspeicher der zweiten FI-CPU
0xA1800000	Adressbereich des zweiten lokalen FI-Controllers
0xA2000000	Hauptspeicher der dritten FI-CPU
0xA2800000	Adressbereich des dritten lokalen FI-Controllers
...	...

Tabelle 4.2: Speicherlayout des globalen FI-Controllers

Die FI-CPUs besitzen neben diesem Hauptspeicher vier Register, mit denen sie mit ihrem jeweiligen lokalen FI-Controller kommunizieren können. Tabelle 4.1 zeigt eine kurze Beschreibung der Register. Durch das Schreiben einer `1` in diese Register wird dem lokalen FI-Controller die entsprechende Situation signalisiert. In einem Fehlerinjektionsexperiment ist nur ein Zugriff auf das `CPU_DONE`-Register zwingend erforderlich. Ein Zugriff auf `FAULT_DETECTED` ist nur dann sinnvoll, wenn Fehlererkennungs- oder -toleranzmaßnahmen getestet werden sollen. Fehlt ein Zugriff auf die `FI_START`- oder `FI_STOP`-Register während des Golden-Runs, werden bei den anschließenden Experimenten die Fehlerzeitpunkte auf die gesamte Programmlaufzeit verteilt, vom Start bis zum Zugriff auf `CPU_DONE`.

Das Speicherlayout des globalen FI-Controllers ist in Tabelle 4.2 abgebildet. Der Controller hat Lese- und Schreibzugriff auf den Hauptspeicher von jeder im System vorhandenen FI-CPU und hat außerdem die Möglichkeit, mit dem zu der FI-CPU gehörenden, lokalen FI-Controller zu kommunizieren. Der Zugriff auf den Hauptspeicher ist notwendig, damit der Speicher vor jedem Experiment der FI-CPU zurückgesetzt werden kann, da durch injizierte Fehler Instruktionen oder vorinitialisierte Daten überschrieben werden können. Außerdem kann durch den Zugriff auf den Hauptspeicher der FI-CPUs das Ergebnis eines Benchmarks nach einem Experiment mit dem Sollergebnis verglichen werden, welches im Golden Run ermittelt wird.

Im Speicherbereich der lokalen FI-Controller befinden sich viele Register, die in Tabelle 4.3 und 4.4 aufgelistet sind. Im Folgenden werden die Funktionen der Register grob erläutert:

- **Statusregister**: Mit diesem Register werden wesentliche Funktionalitäten der lokalen FI-Controller gesteuert. Die Bedeutung der einzelnen Bits sind in Tabelle 4.5 zu sehen. Bei den Bits handelt es sich um Ja/Nein-Werte, wobei eine `0` für *Nein* und eine `1` für *Ja* steht.

Adresse	R/W	Beschreibung (Wertebereich)
0x__800000	RW	Statusregister
0x__800004	–	Reserviert
0x__800008	R	Niederwertige 32 Bit des Taktzählers
0x__80000C	R	Höherwertige 32 Bit des Taktzählers
0x__800010	RW	Niederwertige 32 Bit des Injektionstaktes
0x__800014	RW	Höherwertige 32 Bit des Injektionstaktes
0x__800018	RW	Nummer des zu benutzenden Flip-Flop-Saboteurs (0 – 4203)
0x__80001C	RW	Nummer des zu benutzenden Gate-Saboteurs (0 – 14235)
0x__800020	RW	Zufallswert (0 – 1)
0x__800024	R	Niederwertige 32 Bit des Taktes mit Zugriff auf FI_START
0x__800028	R	Höherwertige 32 Bit des Taktes mit Zugriff auf FI_START
0x__80002C	R	Niederwertige 32 Bit des Taktes mit Zugriff auf FI_STOP
0x__800030	R	Höherwertige 32 Bit des Taktes mit Zugriff auf FI_STOP
0x__800034	RW	Niederwertige 32 Bit des Experimenttimeouts (Takte)
0x__800038	RW	Höherwertige 32 Bit des Experimenttimeouts (Takte)
0x__80003C	RW	Niederwertige 32 Bit der Register-On-Write Schreibzugrif-finjektionsnummer
0x__800040	RW	Höherwertige 32 Bit der Register-On-Write Schreibzugrif-finjektionsnummer
0x__800044	RW	Register, in welches ein Register-On-Write Fehler injiziert wird (0-27)
0x__800048	RW	Bitnummer, welches beim Register-On-Write benutzt wird (0-31)
0x__80004C∗	RW	Zweites Flip-Flop, in welches Fehler injiziert werden (0 – 4203)
0x__800050∗	RW	Drittes Flip-Flop, in welches Fehler injiziert werden (0 – 4203)
...	...	...
0x__800064∗	RW	Achtes Flip-Flop, in welches Fehler injiziert werden (0 – 4203)

Tabelle 4.3: Speicherlayout des globalen FI-Controllers
∗ Nur verfügbar, wenn Mehrbitfehler zur Synthesezeit konfiguriert wurden

Adresse	R/W	Beschreibung (Wertebereich)
`0x__800100`	RW	Niederwertige 32 Bit des ersten Registerschreibzugriffzählers
`0x__800104`	RW	Höherwertige 32 Bit des ersten Registerschreibzugriffzählers
`0x__800108`	RW	Niederwertige 32 Bit des zweiten Registerschreibzugriffzählers
`0x__80010C`	RW	Höherwertige 32 Bit des zweiten Registerschreibzugriffzählers
...	...	...
`0x__8001D8`	RW	Niederwertige 32 Bit des 28. Registerschreibzugriffzählers
`0x__8001DC`	RW	Höherwertige 32 Bit des 28. Registerschreibzugriffzählers

Tabelle 4.4: Speicherlayout des globalen FI-Controllers (Fortsetzung)

Die nur-lesbaren Bits dienen der Auswertung eines Experiments. Sie werden im Laufe des Experiments gesetzt und geben an, ob das Ereignis, welches sie beschreiben, während des Experimentes aufgetreten ist. Die Bits halten ihren Status, bis die CPU ein neues Experiment startet (Wechsel von `1` zu `0` im CPU-Reset/Halt-Bit).

Bits `4` – `6` wählen das Fehlermodell aus, welches zur Fehlerinjektion genutzt werden soll. Das entsprechende Modell muss mittels der anderen zur Verfügung stehenden Register konfiguriert werden. Es ist auch möglich, innerhalb eines Experiments Fehler nach mehreren Fehlermodellen zu injizieren.

- **Taktregister**: Die Plattform benutzt stets 64 Bit zum Zählen der Takte. Da ein Register jedoch nur eine Breite von 32 Bit hat, werden für alle Taktfunktionen zwei Register benutzt, wobei jeweils eines die niederwertigen und das andere die höherwertigen Bits des Taktzählers beinhaltet.

- **Gate-/Flip-Flop Saboteur-Nummer**: Diese Register beschreiben die Adresse eines Saboteurs der Gates bzw. Flip-Flops. Für die Fehlerinjektion in CPU-Registerbits ist eine Zuordnung zwischen diesen Nummern und den Bits notwendig, welche der Controllersoftware bekannt ist.

- **Register-On-Write Konfiguration**: Im Gegensatz zur uniformen Fehlerinjektion in die CPU-Registerbits wird für das Register-On-Write-Fehlermodell direkt die betroffene Registernummer und -bit angegeben.

- **Optionale Flip-Flop-Register**: Um Mehrbitfehler in CPU-Register zu injizieren, müssen in einem Takt mehrere Flip-Flop-Saboteure angeregt werden. Diese Funktionalität ist konfigurierbar, um FPGA-Ressourcen sparen zu können. Ist die Unterstützung für Mehrbitfehler aktiviert, stehen sieben weitere

Bit	R/W	Beschreibung
0	RW	CPU-Reset/Halt
1	R	Zugriff auf `CPU_DONE` der FI-CPU
2	R	CPU-Lockup
3	R	Buszugriffsfehler der FI-CPU
4	RW	Injiziere Flip-Flop-Fehler
5	RW	Injiziere Gate-Fehler
6	RW	Injiziere Register-On-Write Fehler
7	R	Zugriff auf `ERROR_DETECTED` der FI-CPU
8	R	FI-CPU Timeout
9	RW	Stoppe FI-CPU bei Zugriff auf `FI_START`
10	RW	Stoppe FI-CPU bei Zugriff auf `FI_STOP`
11	R	Zugriff auf `FI_START` der FI-CPU
12	R	Zugriff auf `FI_STOP` der FI-CPU
13∗	RW	Injiziere Fehler in zweites Flip-Flop
14∗	RW	Injiziere Fehler in drittes Flip-Flop
15∗	RW	Injiziere Fehler in viertes Flip-Flop
16∗	RW	Injiziere Fehler in fünftes Flip-Flop
17∗	RW	Injiziere Fehler in sechstes Flip-Flop
18∗	RW	Injiziere Fehler in siebtes Flip-Flop
19∗	RW	Injiziere Fehler in achtes Flip-Flop

Tabelle 4.5: Beschreibung der Bits im Statusregister
∗ Nur verfügbar, wenn Mehrbitfehler zur Synthesezeit konfiguriert wurden

Register sowie Statusbits im Statusregister zur Verfügung, um insgesamt bis zu 8 Fehlerinjektionspunkte zu konfigurieren.

4.2 Globaler FI-Controller

Der globale FI-Controller enthält eine eigene CPU, auf der eine Software läuft, welche die Fehlerinjektionskampagnen ausführt. Außerdem enthält dieser Teil der Plattform einen UART-Anschluss, über den die Software mit einem Host-Computer kommunizieren kann. Die CPU ist ebenfalls ein ARM Cortex-M0, wobei diese Wahl recht beliebig und unabhängig vom CPU-Typ der Fehlerinjektionseinheiten ist. Daher ist es ohne weiteres möglich, andere FI-CPU-Architekturen anzubinden, solange diese entsprechende Saboteure enthalten.

Die Host-Schnittstelle ist ein UART, über den mittels des RS232-Protokolls in der Konfiguration 115200 8N1 (115.200 Bit/s, 8 Datenbits, kein Paritätsbit, ein Stopbit) kommuniziert werden kann. Bei dem für diese Arbeit vorliegenden Xilinx Virtex-7 FPGA VC707 Evaluation Kit wird der darauf enthaltene USB2Serial-Wandler benutzt, sodass am Host-PC eine USB-Schnittstelle zur Kommunikation verwendet werden kann.

Am Host wird lediglich ein Terminal-Client benötigt, welcher über die serielle Schnittstelle kommunizieren kann und außerdem eine Unterstützung für das *XMODEM*-Übertragungsprotokoll bietet. Dies ist beispielsweise Picocom [40]. Mit dem Aufruf von

```
picocom -b115200 -fn -pn -d8 --imap lfcrlf /dev/ttyUSB0 --send-cmd
    'sx -vv'
```

kann allumfassend mit der Plattform kommuniziert werden. Weitere Programme werden auf dem Host-Computer nicht benötigt.

Nach dem Programmieren des FPGAs mit dem synthetisierten Hardware-Bitfile startet zunächst der Bootloader. Über diesen kann mittels des XMODEM-Protokolls die Firmware des globalen FI-Controllers eingespielt werden.

Nach dem Start der Controllerfirmware erhält man eine Kommandozeile, in die Befehle eingegeben werden können. Unterstützt wird neben der reinen Texteingabe die Backspace-Taste. Außerdem steht eine kleine Kommando-Historie zur Verfügung, welche über die Pfeil-Hoch- bzw. Pfeil-Runter-Tasten ähnlich einer Bash-Shell bedient werden kann.

Tabelle 4.6 zeigt eine Übersicht der zur Verfügung stehenden Kommandos, welche über die Befehlszeile eingegeben werden können.

Befehl	Parameter	Beschreibung
`boot`		Lädt den Bootloader, kann zum Hochladen einer neuen Controllerfirmware oder zum Neustart genutzt werden
`campaign`	Experimentzahl, Fehlermodell, Zufalls-Seed	Führt eine Fehlerinjektionskampagne mit dem zu dieser Zeit geladenen Experimentcode aus. Die Anzahl der Experimente und das Fehlermodell auf dessen Basis injiziert wird (`gate`, `ff`, `reg`, `regbyte`, `row` oder `all`), muss übergeben werden
`debug`	`on/off`	(De-) Aktiviert den Debug-Modus, in dem viele Diagnoseausgaben ausgegeben werden
`dumpmem`	Startadresse, Länge	Gibt den gewünschten Speicherinhalt auf dem Terminal aus (Hex & ASCII)
`help`	Befehl	Zeigt eine Übersicht über die Befehle an bzw. eine genaue Beschreibung des im Parameter übergebenen Befehls
`setfault`	FI-CPU-Nummer, Fehlermodell, Saboteurnummer, FI-Takt	Konfiguriert manuell die gewünschte FI-CPU mit einem Fehlerinjektionsexperiment. Dient insbesondere zu Diagnosezwecken
`start`	FI-Einheit	Startet die zu der FI-Einheit gehörende FI-CPU (wenn der Parameter fehlt, werden alle gestartet)
`status`	FI-Einheit	Gibt den Status einer oder aller FI-Einheiten zurück.
`stop`	FI-Einheit	Stoppt die zu der FI-Einheit gehörende FI-CPU (wenn der Parameter fehlt, werden alle gestoppt)
`xmodem`		Lädt einen neuen Experimentcode mittels des XMODEM-Protokolls in die Fehlerinjektionsplattform

Tabelle 4.6: Beschreibung der Befehle des globalen FI-Controllers. Graue Parameter sind optional

4.3 Fehlerinjektionskampagne

Dieser Abschnitt beschreibt zunächst die Implementierung des Ablaufs einer Fehlerinjektionskampagne. Anschließend werden die Schritte, die notwendig sind um einen Benchmark für die entwickelte Fehlerinjektionsplattform zu portieren und eine Fehlerinjektionskampagne auszuführen, erläutert.

4.3.1 Ablauf der Fehlerinjektionskampagne

Der generelle Ablauf einer Fehlerinjektionskampagne wurde bereits in Abbildung 3.10 erläutert. In diesem Abschnitt werden einzelne Aspekte der Implementierung dieses Ablaufs vorgestellt und erläutert.

Zu Beginn einer Kampagne werden drei Golden Runs ausgeführt, um verschiedene Daten über den Benchmark zu sammeln. Das mehrfache Ausführen ist nötig, da die FI-CPU nach einem Anhalten nicht fortgesetzt werden kann, sondern zurückgesetzt wird. Die im Golden Run gesammelten sind:

1. Die Laufzeit des Benchmarks in Takten, um die Fehlerraumgröße sowie den Experimenttimeout zu bestimmen

2. Den Wert der Registerschreibzugriffszähler beim Setzen des `FI_START`-Registers (nur für Register-On-Write benötigt)

3. Den Wert der Registerschreibzugriffszähler beim Setzen des `FI_STOP`-Registers (nur für Register-On-Write benötigt)

Aus diesen Daten werden anschließend die Konfigurationen für die einzelnen Fehlerinjektionsexperimente generiert. Der Fokus liegt hier bei der Reproduzierbarkeit der Kampagnen. Die Fehlerinjektionspunkte der einzelnen Experimente werden aus dem Fehlerraum zufällig ausgewählt. Dazu wird der KISS Zufallszahlengenerator [41] benutzt, der eine gute Gleichverteilung der generierten Zufallszahlen sicherstellt. Zusammen mit einem angegebenen Startwert (*Seed*) generiert dieser stets reproduzierbare Zufallswerte. Anschließend werden die generierten Experimente in einem Puffer gespeichert.

Das Zwischenspeichern der Experimentparameter in einem Puffer trennt die Generierung von der Ausführung der Experimente. Dadurch wird zwar ein gewisser Overhead erzeugt, es wird jedoch auch sichergestellt, dass die ausgewählten Fehlerpunkte unabhängig von der Abarbeitung der Experimente sind. Somit sind die ausgeführten Fehlerinjektionsexperimente einer Kampagne auch bei verschieden partitionierten Systemen identisch und die Generierung aller Experimentparameter stets deterministisch.

Der Puffer hat nur eine bestimmte Größe und nimmt in der Regel nicht alle Experimentparameter einer Fehlerinjektionskampagne auf. Daher wird sie im Verlauf der Kampagne immer wieder aufgefüllt. Ebenso kann es passieren, dass für eine FI-CPU keine Experimentdaten im Puffer vorhanden sind, welche diese ausführen kann.

Dann entsteht ein Leerlauf dieser FI-CPU, bis wieder ein passender Datensatz für diese CPU generiert wurde.

Das Abarbeiten eines Experimentparametersatzes aus dem Puffer läuft nach den folgenden Schritten ab:

1. Suchen nach einem für die jeweilige FI-CPU passenden Datensatz mit Experimentparametern im Puffer
2. Beschreiben des FI-CPU Speichers mit dem per XMODEM hochgeladenen Benchmarkcode
3. Setzen der Fehlerinjektionsparameter des entsprechenden lokalen FI-Controllers
4. Starten der FI-CPU
5. Warten auf Terminierung durch Setzen des `CPU_DONE`-Registers oder auf einen Timeout
6. Stoppen der FI-CPU
7. Vergleichen des Ergebnisspeichers der FI-CPU mit dem Referenzergebnis aus dem Golden Run
8. Auswerten der gesetzten Flags (`FAULT_DETECTED` und Buszugriffsfehler) und Speichern des Experimentergebnisses

Diese Schritte werden von der Software des globalen FI-Controllers für alle vorhandenen FI-CPUs parallel durchgeführt. Insbesondere während des Wartens auf die Terminierung oder einen Timeout können andere FI-CPUs verwaltet werden. Ebenfalls wird zwischenzeitlich der Puffer aufgefüllt.

4.3.2 Benchmark vorbereiten

Die auf Fehlertoleranz zu testenden Funktionen eines Benchmarks benötigen in der Regel keine Anpassungen. Vor und nach dem Aufruf der entsprechenden Funktion müssen jedoch zusätzliche Codezeilen eingefügt werden, damit der Benchmark in einem Fehlerinjektionsexperiment verwendet werden kann.

Wie schon in Kapitel 4.1.5 erwähnt, stellt die FI-CPU die Register `FI_START`, `FI_STOP`, `CPU_DONE` und `FAULT_DETECTED` bereit. In jedem Benchmark ist ein Zugriff auf das `CPU_DONE`-Register zwingend erforderlich, welches die Terminierung der Software signalisiert.

Ein minimaler Quicksort-Benchmark ist in Abbildung 4.5 zu sehen. Die Fehler sollen nur in die `quicksort`-Funktion injiziert werden, welche dafür nicht modifiziert werden muss. In den Startupcode des Programms sowie in die Ergebnisaufbereitung sollen jedoch keine Fehler injiziert werden. Dazu wird *vor* dem Aufruf der zu testenden Funktion auf das `FI_START`-Register zugegriffen, direkt *nach* der Funktion auf das `FI_STOP`-Register. Dadurch kann bei dem Ausführen der Kampagne im Golden Run

```
1 #include "config.h"
2 #include "result.h"
3
4 void quicksort(int* array, int n) { [...] }
5
6 #define COUNT = 10;
7 int array[COUNT] = { 51, 23, 54, 93, 2, 43, 21, 65, 23, 97 };
8
9 int main() {
10     *FI_START = 1; // Beginn der Fehlerinjektion
11     quicksort(array, COUNT);
12     *FI_STOP = 1; // Ende der Fehlerinjektion
13
14     hashResultData(array, COUNT * sizeof(int));
15     *CPU_DONE = 1; // Terminierung
16 }
```

Abbildung 4.5: Minimalbeispiel eines Quicksort-Benchmarks, aufbereitet für die Fehlerinjektionsplattform

das Taktfenster bestimmt werden, in welches bei den anschließenden Experimenten Fehler injiziert werden. Das Ergebnis des Benchmarks, in diesem Fall das sortierte Array, muss in einen separaten Speicherbereich kopiert werden, damit der globale FI-Controller dieses Ergebnis findet. Nach dem Golden Run wird das "goldene" Ergebnis als Referenz gespeichert und anschließend, nach den Fehlerinjektionsexperimenten, verglichen.

Dieser Speicherbereich hat in der Standardkonfiguration eine Größe von einem Kilobyte. Um auch ein größeres Ergebnis verarbeiten zu können, wird in der Funktion `hashResultData(void* dataStart, unsigned int dataLen)` ein Hash-Wert über das Ergebnis berechnet. Dieser Hash-Wert hat die Größe des zur Verfügung stehenden Ergebnisspeichers.

Wenn das Ergebnis in kleinen Teilmengen zur Verfügung steht, sollte alternativ die Funktion `hashIncrResultData(void* dataStart, unsigned int dataLen)` verwendet werden, welche eine Verteilung der Teilmengen auf den gesamten Ergebnisspeicher sicher stellt. Außerdem können normale `printf`-Aufrufe benutzt werden. Die Ausgabe der `printf`-Funktion wird ebenfalls in den Ergebnisspeicher geschrieben.

Wenn ein eventuell vorhandener Fehlererkennungsmechanismus einen Fehler erkennt, kann dies mittels eines Zugriffs auf das `FAULT_DETECTED`-Register dem globalen FI-Controller mitgeteilt werden. Diese Fehlererkennung wird dann bei Auswertung der Kampagne berücksichtigt.

Abschließend muss noch ein Zugriff auf das `CPU_DONE`-Register stattfinden, wodurch der globale FI-Controller über den Abschluss des Benchmarks informiert wird und mit der Auswertung des Experiments beginnt.

Der Benchmark sollte mittels des erstellten Makefiles kompiliert werden, um sicherzustellen, dass die korrekten Parameter sowie das richtige Linkerscript und Startup-

Code benutzt werden. Außerdem wird das generierte Kompilat weiterverarbeitet und für das Hochladen in die Fehlerinjektionsplattform vorbereitet.

4.3.3 Fehlerinjektionskampagne ausführen

Dieser Abschnitt beschreibt die nötigen Schritte, um mittels der entwickelten Fehlerinjektionsplattform Kampagnen auszuführen. Zunächst muss das FPGA mit dem synthetisierten Bitfile programmiert werden. Da dieser Vorgang nicht für alle FPGA-Plattformen identisch ist, wird dies hier nicht weiter beschrieben und auf die Dokumentation der jeweiligen FPGA-Plattform verwiesen.

Nach dem Programmieren des FPGAs startet auf der seriellen Konsole der Bootloader. An dieser Stelle kann der Controller-Code mittels des XMODEM-Protokolls in das System geladen werden. Nach dem Abschließen der Übertragung wird der Controller automatisch gestartet.

Der Controller präsentiert nun eine Kommandozeile, in welche die Befehle aus Tabelle 4.6 eingegeben werden können. Um eine Fehlerinjektionskampagne auszuführen, muss als nächstes nach Eingabe des `xmodem`-Kommandos das Benchmark-Programm hochgeladen werden. Dieses muss wie in Kapitel 4.3.2 beschrieben vorbereitet sein. Anschließend kann mittels des `campaign`-Kommandos die Fehlerinjektionskampagne gestartet werden.

Als erster Parameter für das `campaign`-Kommando wird die Anzahl der auszuführenden Experimente angegeben. Selbstverständlich ist es ratsam, möglichst viele Experimente auszuführen, um die Aussagekraft des statistischen Ergebnisses zu erhöhen. Eine höhere Experimentanzahl führt aber auch zu einer höheren Kampagnenlaufzeit. Als zweiter Parameter wird das Fehlermodell angegeben. Die folgenden Optionen stehen zur Auswahl:

- `gate`: Fehlerinjektion auf Gate-Ebene
- `ff`: Fehlerinjektion auf Flip-Flop-Ebene
- `reg`: Fehlerinjektion auf ISA-Ebene (Einzelbitfehler)
- `regbyte`: Fehlerinjektion auf ISA-Ebene (Byte-Fehler)
- `row`: Fehlerinjektion nach dem Register-On-Write Fehlermodell. Nur verfügbar, wenn die Hardware entsprechend synthetisiert wurde
- `all`: Führt nacheinander Kampagnen auf Basis von allen Fehlermodellen aus (`row` nur, wenn verfügbar)

Zu Beginn der Kampagne werden durch Ausführen des Benchmarks ohne Fehlerinjektionen für die Experimente benötigte Daten gesammelt und deren Werte ausgegeben. Während der Ausführung der Experimente wird regelmäßig ein Fortschritt angegeben und nach Beendigung aller Experimente wird das Kampagnenergebnis ausgegeben. In Abbildung 4.6 ist ein beispielhaftes Ergebnis einer Fehlerinjektionskampagne

```
1        gate    |          |    Error Flags after Experiment
2 ---------------#----------#-------------------------------------------
3 Exp. Outcome   |    Total |     none    busErr  faultDet   busFaultErr
4 ---------------#----------#-------------------------------------------
5 noError:       |    15647 |    15628        19         0           0
6 lockup:        |        1 |        0         1         0           0
7 cpuTimeout:    |     2122 |     2122         0         0           0
8 resultDiff:    |     2230 |     2078       152         0           0
9
10 CPU Usage: 95%
11 Fault space size (width x time) = 20956x52623529 = 1102778673724
```

Abbildung 4.6: Beispielhafte Ausgabe eines Kampagnenergebnisses

abgebildet. Die Tabelle gibt die Ausgänge der einzelnen Experimente an. Die Spalten `Exp. Outcome` (Experimentergebnis) und `Total` geben die Anzahl der Experimente mit dem entsprechendem Ergebnis an. In den rechten Spalten wird die Anzahl der Experimente angegeben, bei denen entsprechende Fehlerflags gesetzt sind. `none` steht für kein gesetztes Flag, `busError` zeigt einen Zugriff des Benchmarks auf einen ungültigen Speicherbereich an. Das `faultDet`-Flag ist von dem Benchmark durch einen Zugriff auf das `FAULT_DETECTED`-Register setzbar und dient dazu, Fehlererkennungs- oder -korrekturmaßnahmen zu testen. Die Spalte `bufFaultErr` gibt die Anzahl aller Experimente an, bei denen beide Flags gesetzt sind.

4.4 Zusammenfassung

In diesem Kapitel wurde die Implementation der Fehlerinjektionsplattform vorgestellt. Zunächst wurde erläutert, wie die Konzepte der Saboteure aus Kapitel 3 auf den ARM Cortex-M0-Prozessor übertragen wurden. Anschließend wurde die Problematik der ISA-Level-Fehlerinjektion gelöst, die sich aufgrund des verschleierten Quellcodes der CPU ergibt. Weiterhin wurde die Schnittstelle der Hard- und Software der entwickelten Plattform dokumentiert und Implementierungsdetails erläutert. Abschließend wurde der Ablauf einer Fehlerinjektionskampagne erläutert, mit den notwendigen Schritten, um einen Benchmark auf die Plattform zu portieren und auszuführen. Auf dieser Basis wurden diverse Benchmarks portiert und Experimente ausgeführt, welche als Grundlage für die Evaluation im folgenden Kapitel dienen.

5 Evaluation

Dieses Kapitel evaluiert die in dieser Arbeit entwickelten Ansätze. Abschnitt 5.1 geht auf den Gütevergleich verschiedener Fehlermodelle ein und beantwortet die zentrale Fragestellung dieser Arbeit. Außerdem werden die Fault-Coverage- und die EAF-Metriken miteinander verglichen. Anschließend werden in Abschnitt 5.2 verschiedene Aspekte der entwickelten Fehlerinjektionsplattform evaluiert. Abschließend werden die Ergebnisse dieser Arbeit im Abschnitt 5.3 kritisch betrachtet und mögliche Schwächen identifiziert.

5.1 Vergleich der Fehlermodelle

In diesem Abschnitt werden zunächst die Benchmarks vorgestellt, die für den Vergleich der Fehlermodelle benutzt werden. Anschließend werden die Rohdaten der mit den Benchmarks ausgeführten Fehlerinjektionskampagnen dargestellt. Diese Daten werden dann anhand zweier Metriken qualitativ und mittels mathematischen Zusammenhangsmaßen analysiert. In diesem Rahmen wird die Güte der verschiedenen Fehlermodelle und der Metriken bewertet. Abschließend wird anhand eines Beispiels die Problematik der Fault-Coverage Metrik verdeutlicht.

5.1.1 Auswahl der Benchmarks

Um die Fehlermodelle vergleichen zu können, müssen diese mit einer möglichst großen Anzahl von Benchmarks getestet werden. Diese müssen verschiedene Aufgaben und Szenarien testen, da beispielsweise bei Programmen, welche viele Speicheroperationen tätigen, andere Fehlerauswirkungen auftreten können als bei Programmen, welche viele Berechnungen auf wenigen Daten tätigen. Problematisch bei der Portierung von Benchmarks sind die Ressourcenbeschränkungen der Plattform, insbesondere der Speicher, welcher den Benchmarks zur Verfügung steht. Außerdem können sie nicht über Peripheriegeräte interagieren, um zum Beispiel Daten einzulesen. Die Benchmarks müssen daher statisch mit den Daten, mit denen gerechnet werden soll, initialisiert sein.

Eine Sammlung von Benchmarks, welche einerseits repräsentative Aufgabenstellungen testet und andererseits überwiegend mit den beschränkten Ressourcen auskommt, ist die *MiBench Embedded Benchmark Suite* [42]. Die Benchmarks sind in verschiedene Kategorien einsortiert, portiert für diese Plattform wurden Benchmarks aus den Kategorien Automotive (*basicmath*, *bitcount*, *susan*) und Security (*blowfish enc/dec*, *rijndael enc/dec*, *sha*). Die Größe der Eingabedaten musste häufig verringert werden,

um mit dem Speicher auszukommen. Zum Beispiel betragen die Eingabedaten des *blowfish-small* Benchmarks bereits 305 KB, während den FI-CPUs bei einer Partitionierung mit 16 CPUs lediglich jeweils nur 128 KB zur Verfügung stehen.

Neben Benchmarks der MiBench-Suite wurden außerdem die Sortierbenchmarks aus einer vorangegangenen Bachelorarbeit [43] portiert (*bubblesort*, *gnomesort*, *heapsort*, *insertionsort*, *mergesort*, *quicksort*, *selectionsort*, *shellsort*). Diese erledigen zwar prinzipiell stets dieselbe Aufgabe, die Algorithmen erfüllen diese aber auf sehr unterschiedliche Weise. Außerdem wurden noch zwei Benchmarks portiert, welche mittels des Δ-Encodings [11] gegen Fehler abgehärtet wurden (*delta-bubblesort*, *delta-quicksort*). Auch bei diesen wurde die Eingabegröße verringert, um mit der Speicherbegrenzung auszukommen.

5.1.2 Ergebnisse der Fehlerinjektionskampagnen

Mit den in Abschnitt 5.1.1 vorgestellten Benchmarks wurden Fehlerinjektionskampagnen ausgeführt, deren Ergebnisse in den Abbildungen 5.1 und 5.2 zu sehen sind. Es wurde pro Benchmark und Fehlermodell eine Fehlerinjektionskampagne mit je 20000 Experimenten ausgeführt. Außerdem wurde pro Benchmark eine weitere Fehlerinjektionskampagne mittels der FAIL∗-Plattform ausgeführt, welche ähnlich zu dem entwickelten Register-Einzelbitfehler-Modell zu einem beliebigen Zeitpunkt zwischen den Zugriffen auf `FI_START` und `FI_STOP` einzelne Bits in den CPU-Registern invertiert. Eine Einschränkung der FAIL∗-Plattform ist, dass diese keine Fehler in den Instruktions-Pointer der simulierten ARM-CPU injizieren kann. Außerdem gibt es auf dieser Plattform keinen *Lockup*, wie bei der ARM Cortex-M0-CPU, allerdings stürzen manche Experimente während der Simulation ab. Diese nicht ausführbaren Experimente werden daher als *Lockup* gewertet.

Die Graphen in den Abbildungen 5.1 und 5.2 sind gruppiert nach den Benchmarks und zeigen die Ergebnisse der Fehlerinjektionskampagnen unter Verwendung verschiedener Fehlermodelle. Die möglichen Ausgänge eines Experiments sind, wie bereits in Kapitel 4.3.3 beschrieben, *no error*, *lockup*, *cpu timeout* und *result diff*. Die Balken geben die Anzahl der Ausgänge innerhalb einer Kampagne an und sind übereinander gestapelt, sodass die unterschiedlichen Ergebnisse der Kampagnen eines Benchmarks verglichen werden können.

Bei vielen Benchmarks ist festzustellen, dass die Anzahl der *no error*-Ergebnisse mit zunehmender Abstraktion der Fehlermodelle abnimmt. Dies war zu erwarten, da insbesondere die Fehlerpunkte wegabstrahiert wurden, welche vermeintlich wenige Auswirkungen auf das Ergebnis haben. Dieser Verlauf stimmt jedoch nicht überall, wie beispielsweise die Kampagne *selectionsort*/Register-On-Write zeigt.

Abbildung 5.3 zeigt die zusammengefasste Verteilung der Ergebnisse *no error* und *result diff* in einem Box-Whisker-Plot, gruppiert nach den Fehlermodellen. Es ist klar zu erkennen, dass die Anzahl der fehlerfreien Experimente bei zunehmend abstrakteren Fehlermodellen abnimmt und Streuung der Ergebnisse zunimmt.

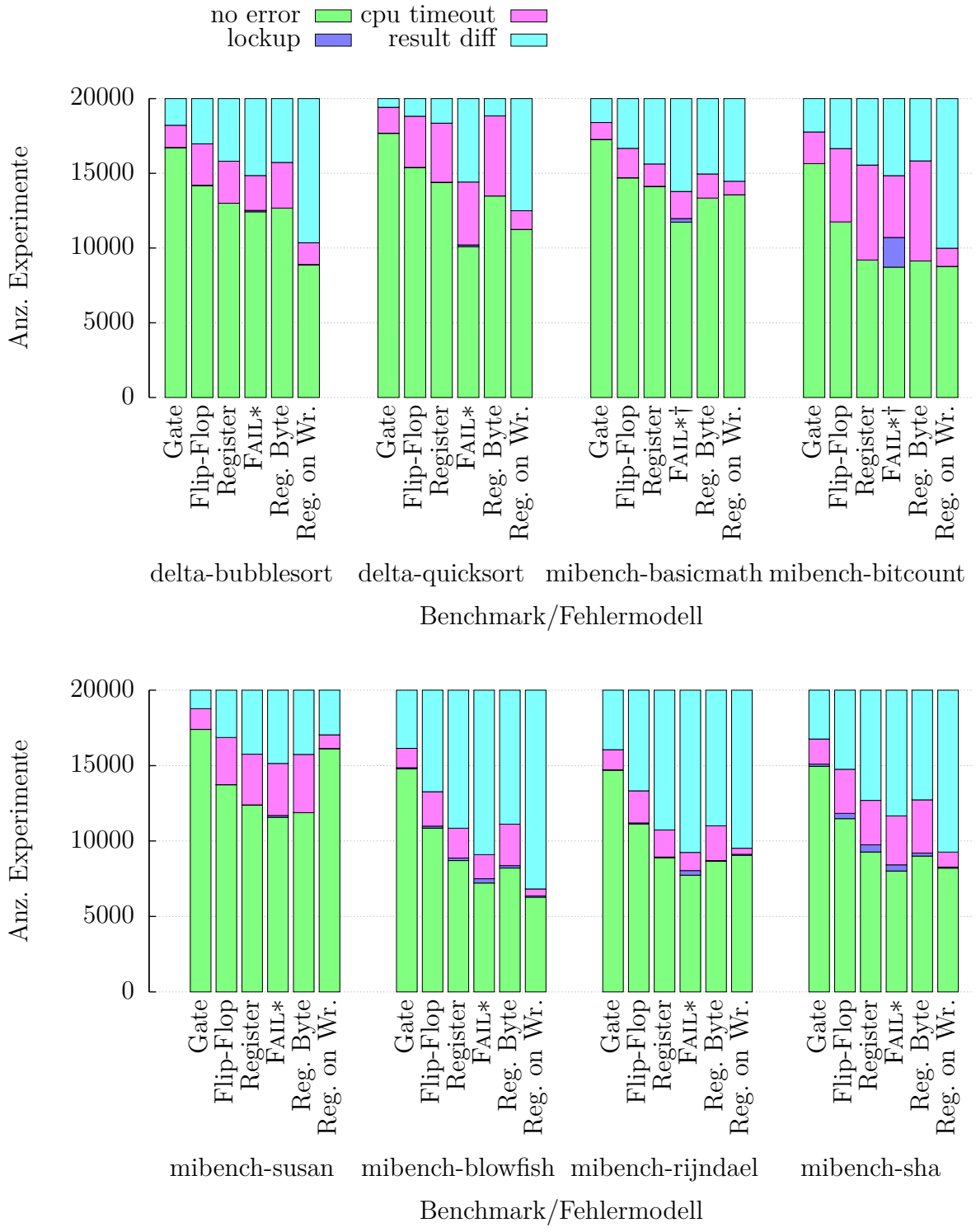

Abbildung 5.1: Ergebnisse nach einer Fehlerinjektionskampagne mit 20000 Experimenten der *MiBench-Suite* und der ungehärteten Delta-Benchmarks. Mit † gekennzeichnete Ergebnisse sind aufgrund zu langer Kampagnenlaufzeit von 5000 Experimenten auf die dargestellten 20000 extrapoliert

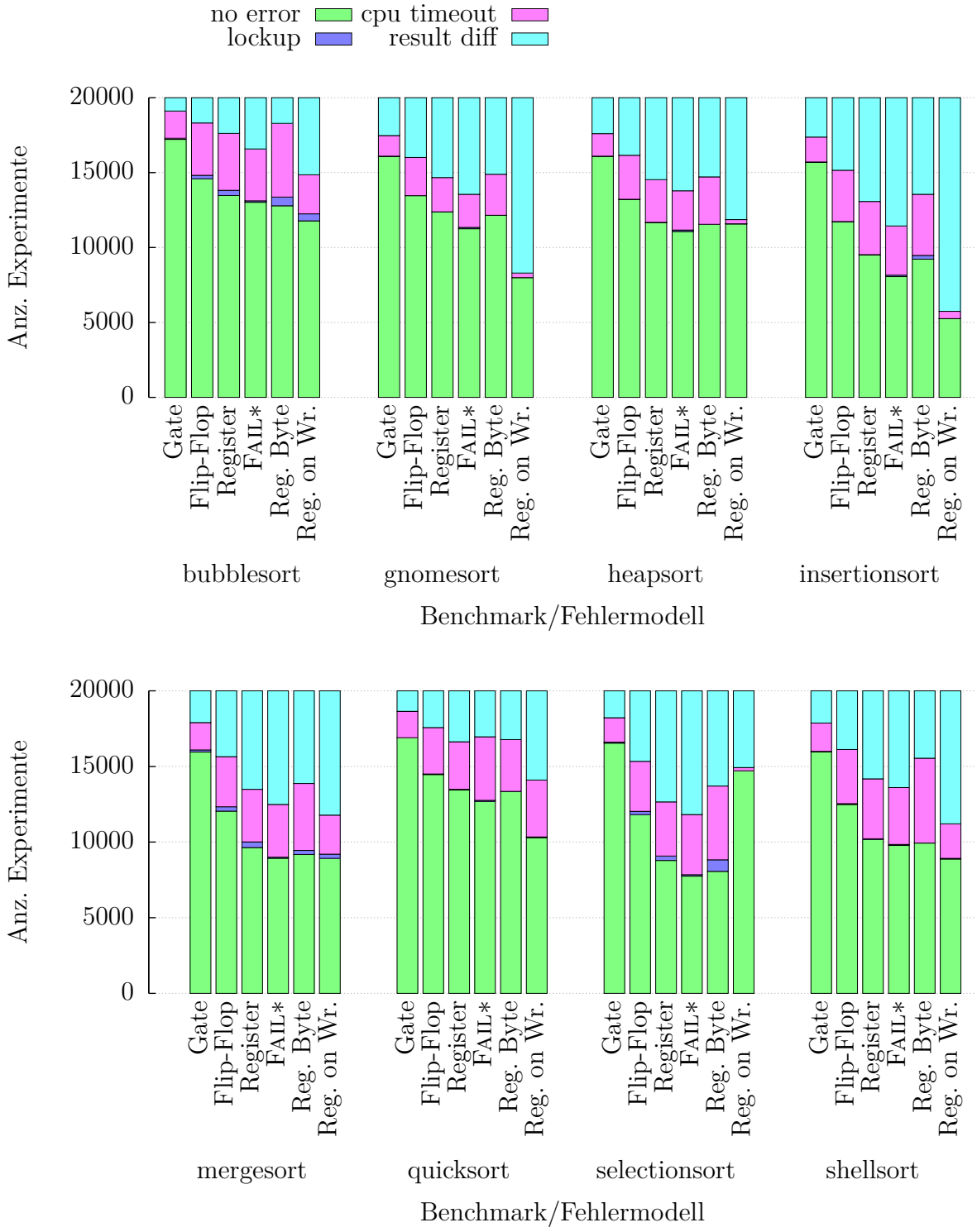

Abbildung 5.2: Ergebnisse nach einer Fehlerinjektionskampagne mit 20000 Experimenten der Sortierbenchmarks aus [43]

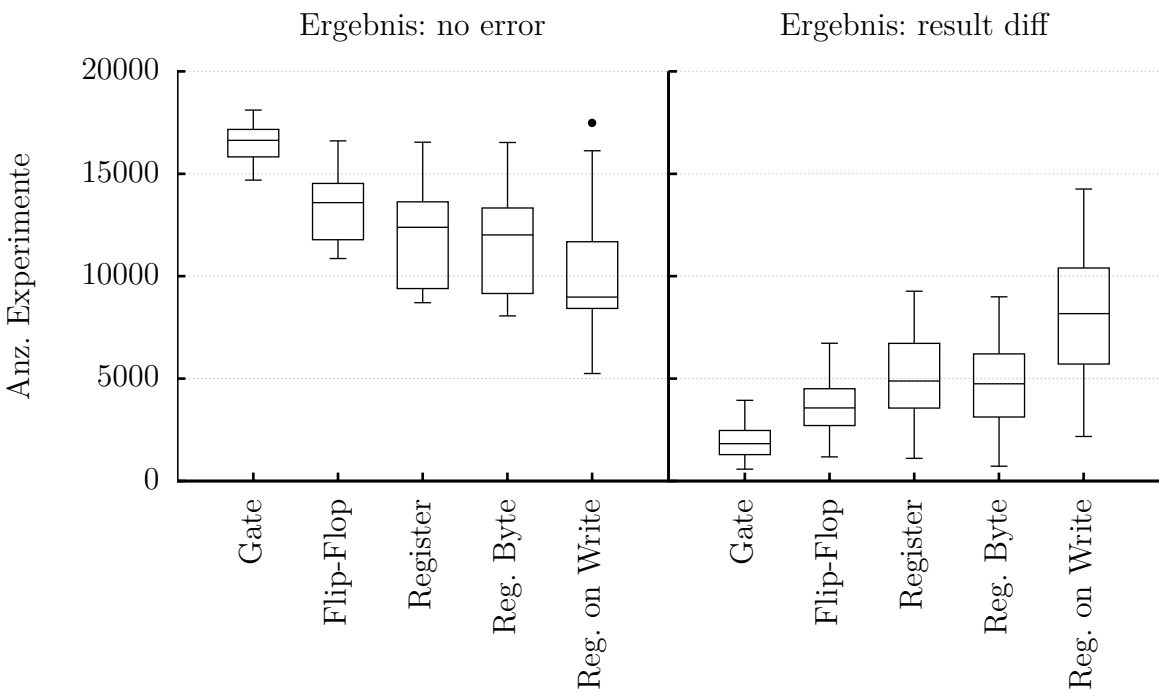

Abbildung 5.3: Box-Whisker-Plot der Kampagnenreihe mit 20000 Experimenten, gruppiert nach den Fehlermodellen und den Experimentergebnissen *no Error* und *result diff* (Whiskerlänge max. $1.5 * \text{IQR}$, siehe [44])

5.1.3 Einfluss der Metrik

Ein gewisser Trend bei den fehlermodellübergreifenden Ergebnissen ist bereits in den Abbildungen 5.1 und 5.2 erkennbar. Die Anzahl der *no error*-Experimentergebnisse nimmt überwiegend bei zunehmender Abstraktion des zugrunde liegenden Fehlermodells ab, während die Zahl der *result diff*-Ergebnisse zunimmt. Allerdings sind auch Abweichungen von diesem Trend erkennbar, beispielsweise bei dem Vergleich der *mibench-susan*/Register-On-Write- oder der *selectionsort*/Register-On-Write-Kampagnen mit den anderen Kampagnen des jeweiligen Benchmarks, welche mit anderen Fehlermodellen ausgeführt wurden. In diesem Abschnitt wird daher der Zusammenhang der Modelle unter Berücksichtigung verschiedener Metriken genauer untersucht.

Die Untersuchung beschränkt sich im wesentlichen auf die Betrachtung der Silent Data Corruption (SDC), da insbesondere bei Software-basierten Hardware-Fehlertoleranzverfahren angestrebt wird, diese zu vermeiden. Bei den bisher vorgestellten Benchmarks ist die Anzahl der Silent Data Corruption identisch mit der Anzahl der Experimente mit einem *result diff*-Ergebnis, da diese keine Fehlertoleranzmaßnahmen besitzen und somit keine Fehler detektieren können.

Abbildung 5.4 zeigt oben die absolute Anzahl aufgetretener SDCs aus derselben Reihe von Fehlerinjektionskampagnen wie im Abschnitt 5.1.2. Es wurden pro Kam-

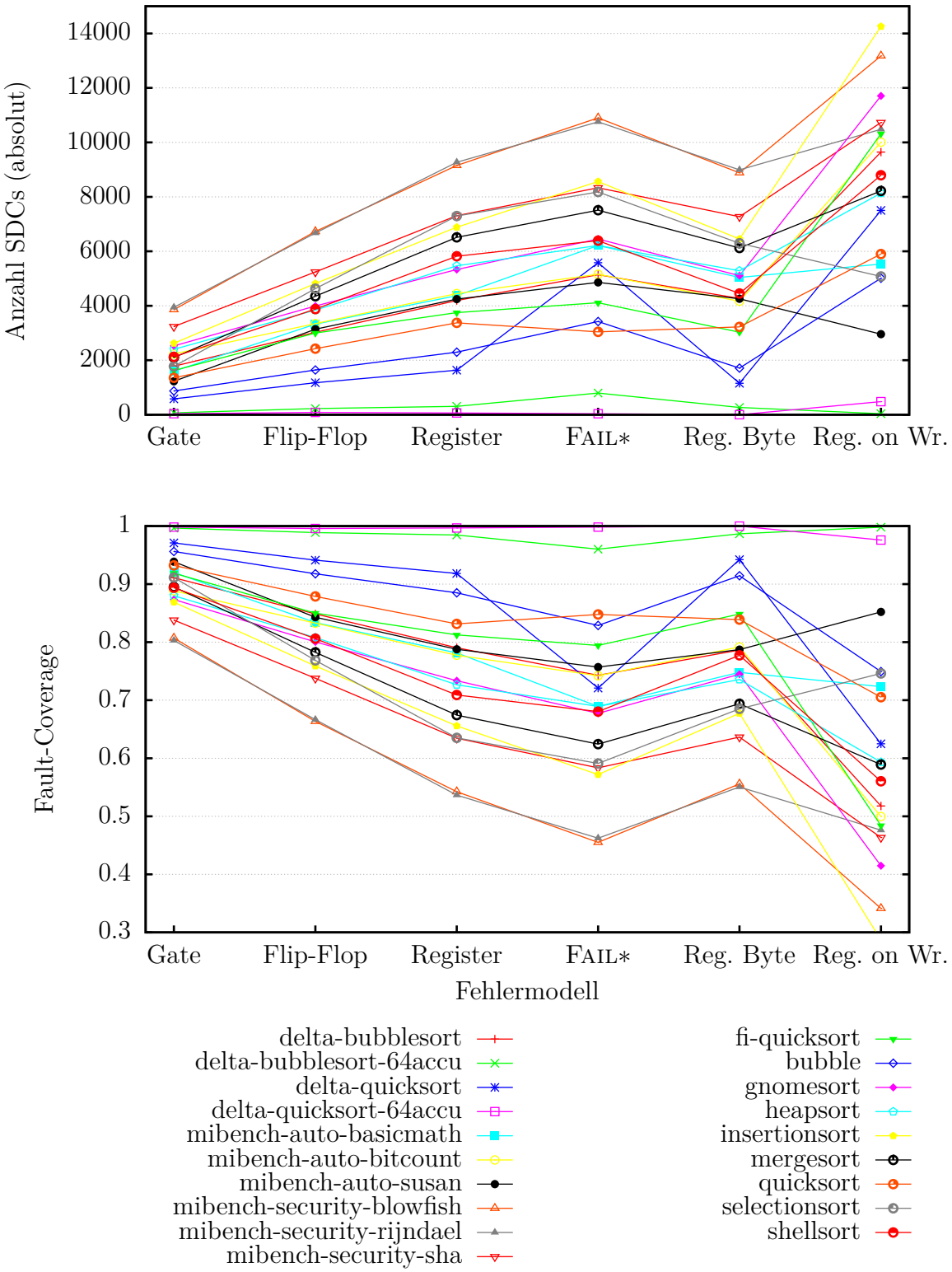

Abbildung 5.4: Absolute SDCs (oben) und daraus resultierende Fault-Coverage (unten) der Fehlerkampagnen mit 20000 Experimenten

pagne 20000 Experimente ausgeführt. Die Verbindungslinien zwischen den Punkten dienen dazu, den Verlauf der Ergebnisse eines Benchmarks über verschiedene Fehlermodelle hinweg zu verdeutlichen.

Der untere Graph von Abbildung 5.4 zeigt den Verlauf der Fault-Coverage (siehe Kapitel 2.3) für die ausgeführte Kampagnenreihe. Da zur Berechnung der Fault-Coverage der Absolutwert lediglich durch einen Faktor dividiert wird und dieses Ergebnis von 1 abgezogen wird, ist der Graph an der X-Achse gespiegelt und sieht sonst dem oberen Graphen der Absolutwerte sehr ähnlich.

Bei einer qualitativen Betrachtung der Graphen stellt man fest, dass es scheinbar einen linearen Zusammenhang der Ergebnisse der ausgeführten Kampagnen mit dem Gate-, Flip-Flop-, Register- und Register-Byte-Fehlermodell gibt. Bei dem durch Fail∗ implementierten Fehlermodell ist dieser Zusammenhang bei den meisten Benchmarks ebenfalls zu sehen, jedoch gibt es hier beispielsweise mit dem *delta-quicksort*-Benchmark einen deutlichen Ausreißer. Die Messung dieses Benchmarks wurde zwei mal wiederholt und auf Fehler überprüft, ohne dass Ungereimtheiten festgestellt wurden.

Die Benchmarks können anhand der Fault-Coverage bewertet werden. Die Idee dieser Metrik ist, dass ein hoher Fault-Coverage-Wert eine hohe Fehlertoleranz eines Benchmarks indiziert. Es kann somit pro Fehlermodell eine Rangfolge der Benchmarks gebildet werden. Diese Rangfolge ändert sich teilweise über mehrere Fehlermodelle hinweg. Bei den Gate-, Flip-Flop-, Register- und Register-Byte-Fehlermodellen sind relativ wenige Vertauschungen über viele Rangplätze hinweg erkennbar. Vertauschungen mit einer geringen Distanz der Rangfolgen sind bei stets nahezu identischen Werten nicht verwunderlich, beispielsweise bei dem *mibench-security-blowfish* und dem *mibench-security-rijndael*-Benchmark. Bei der Betrachtung der Rangfolge stellt der *delta-quicksort*-Benchmark wieder einen deutlichen Ausreißer dar.

Keinen offensichtlichen Zusammenhang scheint es zwischen dem Register-On-Write und den anderen Fehlermodellen zu geben, da die Graphen bei dem Übergang zu dem Register-On-Write Modell widersprüchliche Verläufe annehmen und auch sehr viele Änderungen in der Rangfolge beobachtbar sind.

Auffallend sind auch die Werte der gehärteten *delta-bubblesort-64accu* und *delta-quicksort-64accu*-Benchmarks, deren Fault-Coverage-Werte sich stets auf einem sehr niedrigen Niveau befinden. Wird die Fault-Coverage als Metrik benutzt, scheint dieser Härtungsmechanismus gut zu funktionieren.

Abbildung 5.5 zeigt die EAF-Werte (siehe Kapitel 2.3) der portierten Benchmarks. Die Experimentanzahl der Fehlerinjektionskampagnen wurde teilweise erhöht, um durchgängig ein Konfidenzintervall von unter $1,6\%$ zum Konfidenzniveau 99% zu erhalten. Aufgrund der Verteilung der EAF-Werte über mehrere Größenordnungen hinweg wurde für diesen Graph auf der Y-Achse eine logarithmische Skalierung gewählt. Eine Rangfolge der Benchmarks ergibt sich anhand der EAF-Werte, wobei ein geringerer Wert eine höhere Fehlertoleranz repräsentiert. Wie schon bei dem Graphen der Fault-Coverage in Abbildung 5.4 erkennt man bei qualitativer Betrachtung einen Zusammenhang zwischen den Werten der Gate-, Flip-Flop-, Register-, und Register-Byte Fehlermodelle, sowohl beim Verlauf als auch bei der daraus resultierenden Rangfolge.

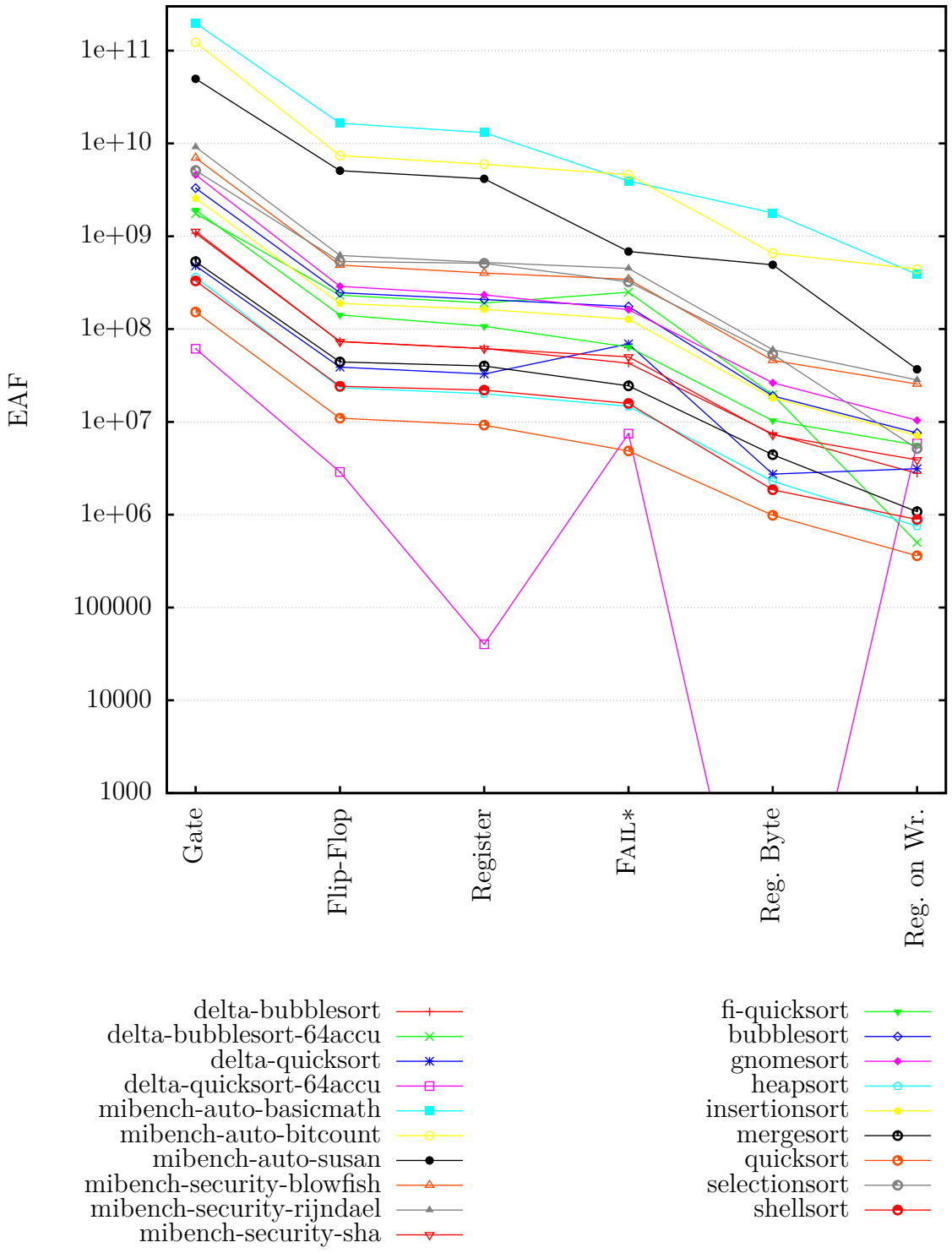

Abbildung 5.5: Extrapolierte absolute Fehlerzahl der Fehlerinjektionskampagnen (mit logarithmischer Skalierung der Y-Achse). Das Konfidenzintervall zum Niveau 99 % ist stets unter 1,6%

Im Vergleich zum Fault-Coverage-Graphen weichen die Ergebnisse, die mittels FAIL∗ ermittelt wurden stärker von den anderen Werten ab. Ein Grund ist, dass FAIL∗ in jedem simulierten Takt eine Instruktion ausführt, während der reale ARM Cortex-M0 teilweise mehrere Takte pro Instruktion benötigt, zum Beispiel bei Sprüngen oder Speicherzugriffen. Dadurch verändert sich die gemessene Laufzeit der Benchmarks, was wiederum die Größe des Fehlerraums und somit den EAF-Wert beeinflusst. Bei dem Register-On-Write-Fehlermodell sind, wie schon bei der Fault-Coverage, Abweichungen um teilweise mehrere Größenordnungen zwischen den erwarteten und den gemessenen Verläufen festzustellen und viele Rangvertauschungen über mehrere Ränge hinweg zu sehen.

Die EAF-Werte der gehärteten *delta-bubblesort-64accu* und *deta-quicksort-64accu* Benchmarks variieren zum Teil sehr stark und schneiden im Vergleich zu der Fault-Coverage nicht so gut ab. Der Hauptgrund wird die viel längere Laufzeit der Benchmarks gegenüber ihren ungehärteten Varianten sein. Der gehärtete *delta-bubblesort-64accu*-Benchmark wird dadurch bei allen Fehlermodellen, außer dem Register-On-Write-Modell, sogar schlechter bewertet, als seine ungehärtete Variante.

5.1.4 Untersuchung mittels Zusammenhangsmaßen

Bisher wurden die Benchmarkergebnisse nur qualitativ betrachtet. In diesem Abschnitt werden mathematische Methoden angewendet, um die Ergebnisse zu bewerten.

Gezeigt werden soll eine Fehlermodell-übergreifende Abhängigkeit der Ergebnisse. Dazu wird das Gate-Fehlermodell stets als Referenz benutzt, da dieses aufgrund von wenig Abstraktion am ehesten reale Fehler darstellen kann. Um den Zusammenhang zwischen dem Gate Modell und den anderen Modellen festzustellen, bieten sich sogenannte *Zusammenhangskoeffizienten* an. Diese stellen eine Metrik für einen Zusammenhang zweier Ergebnisreihen dar und bieten sich daher als Zusammenhangsmaß für die Fehlermodelle an. Die benutzten Maße werden nun vorgestellt. Alle haben gemeinsam, dass sie Werte zwischen -1 und 1 annehmen, wobei 1 für einen positiven und -1 für einen negativen Zusammenhang steht. Bei dem Wert 0 besteht kein Zusammenhang. In Abbildung 5.6 finden sich drei beispielhafte Verteilungen mit den Werten der vorgestellten Koeffizienten.

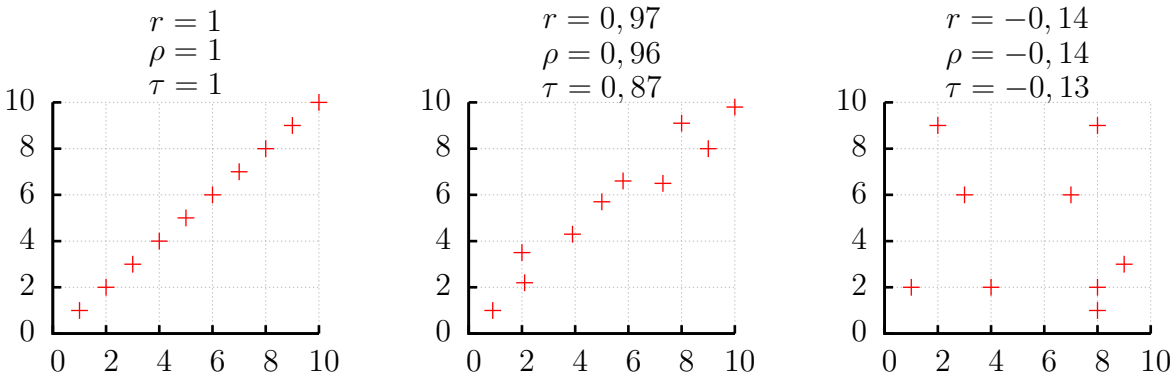

Abbildung 5.6: Beispielhafte Verteilungen mit Angabe von Pearsons r, Spearmans ρ und Kenndals τ

- **Pearsons** r (auch: *Korrelationskoeffizient*): Pearsons r gibt ein Maß für den linearen Zusammenhang zwischen zwei intervallskalierten Zahlenreihen an. Berechnet wird dieses Maß durch

$$r_{\mathrm{XY}} = \frac{\mathrm{Cov}(X,Y)}{\sigma(X)\sigma(Y)}, \tag{5.1}$$

  mit der Kovarianz *Cov* und der Standardabweichung σ. Pearsons r arbeitet auf den Rohdaten zweier Verteilungen und berücksichtigt auch die Abstände zwischen den Werten.

- **Spearmans Rangkorrelationskoeffizient** ρ: Wie der Name schon sagt, arbeitet dieser Koeffizient lediglich auf den Rängen der Werte. Dazu werden die Werte einer Reihe miteinander verglichen und deren Reihenfolge ermittelt. Anschließend wird die Berechnungsvorschrift von Pearsons r auf die Rangfolge zweier Wertereihen angewendet. Spearmans ρ betrachtet also die Rangvertauschungen. Dadurch entfällt die Annahme, dass die Werte linear voneinander abhängen. Außerdem ist der Koeffizient tolerant gegenüber Ausreißern.

- **Kendalls** τ: Dieses Korrelationsmaß arbeitet wie Spearmans ρ auf den Rängen, berechnet allerdings die Kovarianz der Ränge. τ strebt schneller gegen eine Normalverteilung, wenn sich die Samplegröße erhöht, wodurch sich τ besser für eine Analyse von wenigen Samples eignet als ρ [45]. Im Allgemeinen eignen sich jedoch beide Verfahren, da sie bei Hypothesentests fast immer zur gleichen Entscheidung führen [46].

Tabelle 5.1 zeigt die Zusammenhangsmaße nach Pearsons r, Spearmans ρ und Kendalls τ für die ausgeführte Kampagnenreihe unter Benutzung der Fault-Coverage. Die Werte von Pearsons r für den Zusammenhang mit dem Gate-Fehlermodell sind für das Flip-Flop-, Register- und Register-Byte-Fehlermodell auf einem hohen Niveau.

Gate vs ...	Flip-Flop	Register	FAIL*	Reg. Byte	Reg. on Write
r	0.964	0.941	0.897	0.947	0.819
ρ	0.916	0.898	0.847	0.886	0.872
τ	0.789	0.766	0.684	0.754	0.661

Tabelle 5.1: Korrelationsmaße r, ρ und τ zwischen dem Gate- und den anderen Fehlermodellen auf Basis der Fault-Coverage

Gate vs ...	Flip-Flop	Register	FAIL*	Reg. Byte	Reg. on Write
r	0.987	0.988	0.938	0.976	0.938
ρ	0.989	0.991	0.956	0.986	0.835
τ	0.942	0.953	0.860	0.942	0.719

Tabelle 5.2: Korrelationsmaße r, ρ und τ zwischen dem Gate- und den anderen Fehlermodellen auf Basis der EAF

Bei dem Wert für FAIL* machen sich die Ausreißer bemerkbar, welche bereits in der Grafik in Abbildung 5.4 zu sehen waren. Die Werte für ρ und τ sind alle weitaus geringer als die von r. Der Grund hierfür ist sicherlich, dass ρ und τ auf Basis der Rangfolge arbeiten und, wie in Abbildung 5.4 zu sehen, bereits beim Übergang vom Gate zum Flip-Flop-Fehlermodell viele Vertauschungen in der Rangfolge auftreten.

In der Tabelle 5.2 sind dieselben Zusammenhangsmaße angegeben, jedoch auf Basis der EAF. Es ist zu beobachten, dass die Werte aller Maße für den Vergleich des Gate-Fehlermodells mit dem Flip-Flop-, Register- und Register-Byte-Modell sehr hoch sind. Die Werte des Vergleiches mit dem FAIL*-Fehlermodell sind geringer, da hier neben den Ausreißern zusätzlich die geringere Fehlerraumgröße eine Rolle spielt.

Beide Tabellen haben gemeinsam, dass alle Zusammenhangsmaße für den Vergleich des Gate-Fehlermodells mit dem Register-On-Write-Modell wesentlich geringer sind als die anderen Werte. Dies deckt sich mit der qualitativen Beobachtung, dass die Ergebnisse dieses Modells nicht zu dem Trend, der sich durch die anderen Fehlermodelle ergibt, passen.

5.1.5 Gegenüberstellung der Fault-Coverage und der EAF-Metrik

Insbesondere bei dem Vergleich der Werte von Kendalls τ in den Abbildungen 5.1 und 5.2 fällt auf, dass die abstrakten Modelle unter Verwendung der Fault-Coverage eine geringere Korrelation zu dem Gate-Level-Modell aufweisen, als bei der EAF-Metrik. Diese Beobachtung beschränkt sich nicht nur auf τ, sondern ist auch in einer nicht ganz so starken Ausprägung bei r und ρ zu sehen. Dies lässt den Schluss zu, dass die Bewertung der Kampagnenergebnisse mit der EAF-Metrik über mehrere Fehlermo-

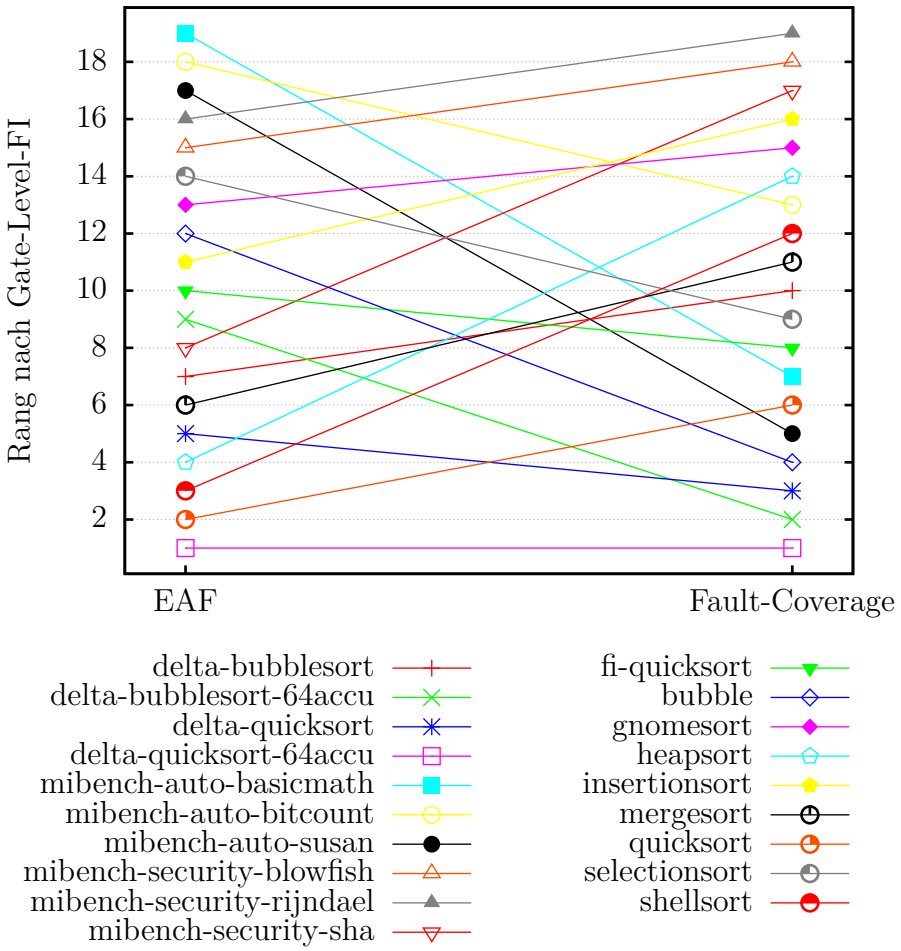

Abbildung 5.7: Vergleich der Rangbewertung durch die Fault-Coverage- und die EAF-Metrik von einer Kampagne mit Gate-Level-Fehlerinjektion

delle hinweg konstanter ist, als unter Verwendung der Fault-Coverage-Metrik. Diese Aussage muss jedoch aufgrund der relativ geringen Datenbasis mit 19 Benchmarks mit etwas Vorsicht betrachtet werden.

Wenn man verschiedene Benchmarks nach einer Fehlerinjektionskampagne miteinander vergleichen möchte, wird üblicherweise die Rangfolge der Benchmarks gebildet. Bei der Fault-Coverage ist ein hoher Wert besser, bei der EAF-Metrik ein niedriger. Für eine Gate-Level-Fehlerinjektion sind die resultierenden Rangbewertungen der benutzten Benchmarks in Abbildung 5.7 abgebildet, unter Verwendung beider Metriken. Anhand der Verbindungslinien ist erkennbar, dass es extrem viele Vertauschungen gibt und die beiden Rangfolgen kaum Gemeinsamkeiten aufweisen. Mit den Zusammenhangsmaßen stellt man die Werte $r = \rho = 0.282$, und $\tau = 0.205$ fest, welche ebenfalls für eine sehr geringe Korrelation der beiden Rangreihen sprechen.

Die Ursache für diese starke Differenz in der Bewertung der Benchmarks liegt sicherlich darin, dass die EAF-Metrik die Fehlerraumgröße und somit die Laufzeit be-

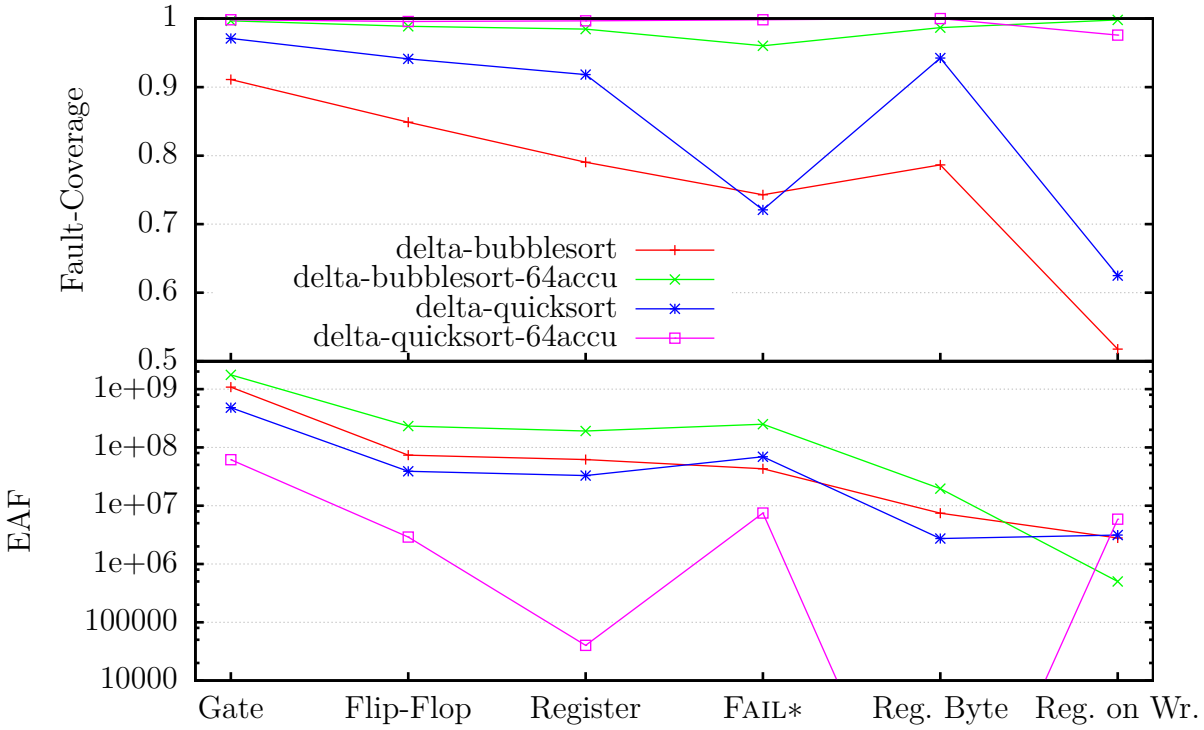

Abbildung 5.8: Gegenüberstellung der Delta-Benchmark Kampagnenreihe ausgewertet mittels der Fault-Coverage (oben) und der EAF (unten, Y-Achse logarithmisch skaliert)

trachtet, während die Fault-Coverage diese ignoriert. Da bereits gezeigt wurde, dass die Laufzeit bei der Bewertung der Fehlertoleranz eines Programms relevant ist [12] und die Fault-Coverage zu komplett anderen Ergebnissen führt, sollte diese nicht für einen Vergleich zweier oder mehrerer Benchmarks benutzt werden.

5.1.6 Vergleich der Delta-Benchmarks

Besonders deutlich wird der Einfluss der Metrik auf das Bewertungsergebnis, wenn man die ungehärteten Delta-Benchmarks mit den gehärteten vergleicht. In Abbildung 5.8 sind nochmal die Werte der Fault-Coverage sowie der EAF der Delta-Benchmarks zu sehen.

Der Graph der Fault-Coverage entspricht den Erwartungen, die man an einen guten Härtungsmechanismus hat: Die gehärteten Varianten sind fehlertoleranter als die Ungehärteten. Anders ist jedoch der Verlauf des EAF-Graphen. Hier ist zu sehen, dass der EAF-Wert des gehärteten *bubblesort*-Benchmarks sogar schlechter ist, als die ungehärtete Variante. Dies würde bedeuten, dass der Benchmark durch das angewandte Härtungsverfahren fehleranfälliger wird. Diese Diskrepanz zwischen den Interpretationen erklärt sich durch den Laufzeitzuwachs der gehärteten Varianten. Während der ungehärtete *bubblesort*-Benchmark eine Laufzeit von 5,8 Millionen Takten hat,

wächst die Laufzeit bei der gehärteten Variante auf 271 Millionen Takte an. Dieser Laufzeitzuwachs um den Faktor 47 wird von der Fault-Coverage komplett ignoriert. Bei der Berechnung des EAF-Wertes wird jedoch davon ausgegangen, dass mit einer Vergrößerung der Laufzeit auch die Anzahl der Fehlerpunkte steigt. Betrachtet man Neutronen in der Atmosphäre, führen diese in einem Zeitintervall mit einer gewissen Wahrscheinlichkeit zu einem Fehler in der CPU. Wird das Zeitintervall vergrößert, steigt dementsprechend auch die Wahrscheinlichkeit, dass diese einen Fehler auslösen. Eine Verdoppelung der betrachteten Zeit führt daher auch zu einer annähernden Verdoppelung der Fehlerauftrittswahrscheinlichkeit.

Um im gehärteten *bubblesort*-Benchmark einen besseren EAF-Wert zu erreichen als bei der ungehärteten Variante, müsste der Härtungsmechanismus die Fehlertoleranz des Benchmarks um mindestens den oben genannten Faktor 47 des Laufzeitzuwachs verbessern. Da die EAF-Werte des gehärteten Benchmarks jedoch schlechter sind, scheint dieser Faktor beim Zuwachs der Fehlertoleranz nicht erreicht zu werden.

Bei dem *quicksort*-Benchmark beträgt der Faktor des Laufzeitzuwachs *nur* 22. Da der EAF-Wert des gehärteten Benchmarks deutlich besser ist als bei der ungehärteten Variante, scheint der Härtungsmechanismus diesen Laufzeitzuwachs ausgleichen zu können.

5.2 Evaluation des FI-Systems

Der Umfang der in dieser Arbeit entwickelten Fehlerinjektionsplattform beträgt über 6000 Zeilen Code, von denen knapp 2800 auf den C-Quellcode, ca. 2350 Zeilen auf den Verilog-HDL-Code und ca. 550 auf die Codetransformations-Skripte zum hinzufügen der Saboteure entfallen. Nicht in diesen Zahlen enthalten sind die portierten Benchmarks sowie der Code des ARM Cortex-M0-Prozessors und der Verilog-UART Implementierung. Die Codezeilen der originalen CPU summieren sich auf 16606 Zeilen, nach der (automatisierten) Erweiterung um die Saboteure beträgt die Anzahl der Codezeilen 34079.

Die Kompilierzeit der Software beträgt nur wenige Sekunden, währen die Hardwaresynthese des kompletten Systems mehrere Stunden in Anspruch nimmt. Abbildung 5.9 zeigt die Synthesezeiten für verschiedene Systemkonfigurationen. Bemerkenswert ist auch der maximale Speicherbedarf des Syntheseprozesses, welcher vor dem Abbruch der Variante mit 32 FI-CPUs bei über 20 GB lag.

Um die Auslastung der Logikzellen eines FPGAs anzugeben, wird in der Regel die Anzahl der vom Hardwaredesign benutzten LUTs verwendet. Das verwendete FPGA (XC7VX485T) besitzt 303600 dieser LUTs [38]. In Abbildung 5.10 ist der LUT-Verbrauch von verschiedenen Konfigurationen der Fehlerinjektionsplattform angegeben. Der Gesamtverbrauch steigt bei zunehmender Anzahl von Partitionen kontinuierlich an. Eine Konfiguration mit 32 FI-CPUs passt nicht mehr auf das oben genannte FPGA, sodass hierfür keine Angaben über den LUT-Verbrauch der einzelnen Komponenten getroffen werden kann. Die Variante der Fehlerinjektionshardware

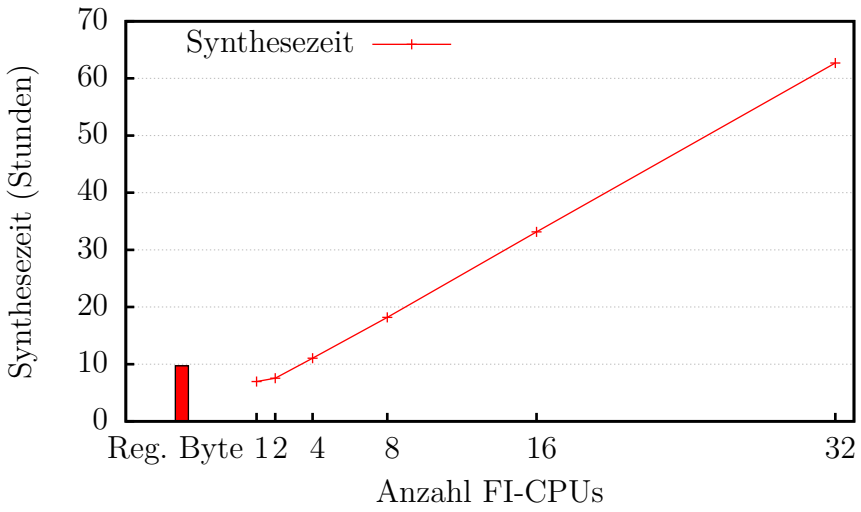

Abbildung 5.9: Synthesezeit verschiedener Konfigurationen der FI-Plattform

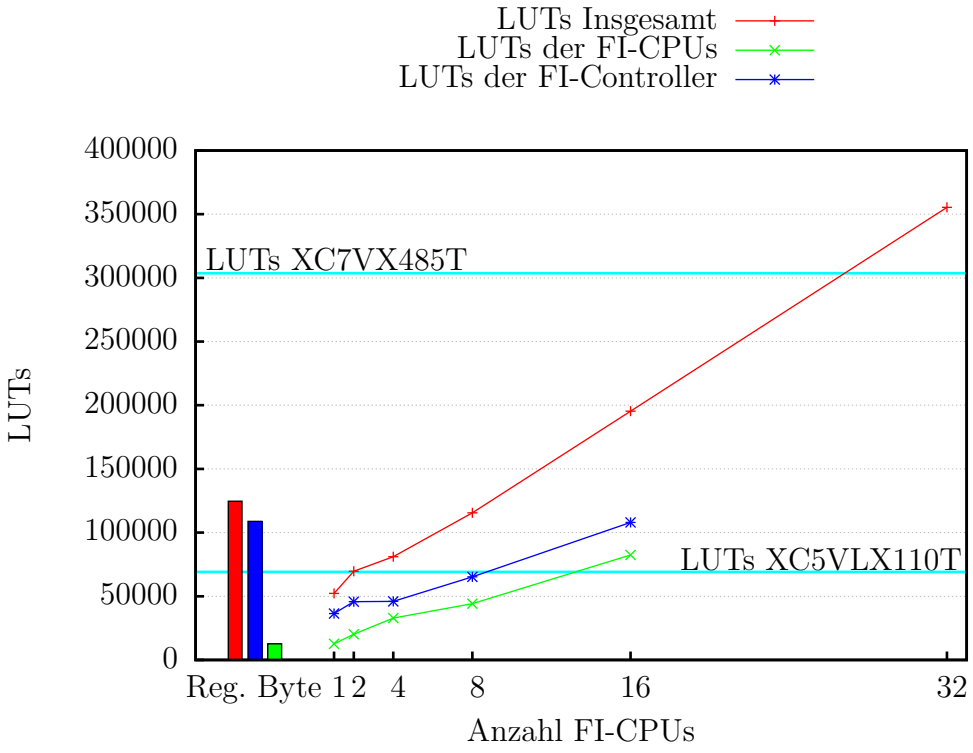

Abbildung 5.10: LUT-Verbrauch verschiedener Konfigurationen der FI-Plattform mit Angaben der zur Verfügung stehenden LUTs zweier ausgewählter FPGAs

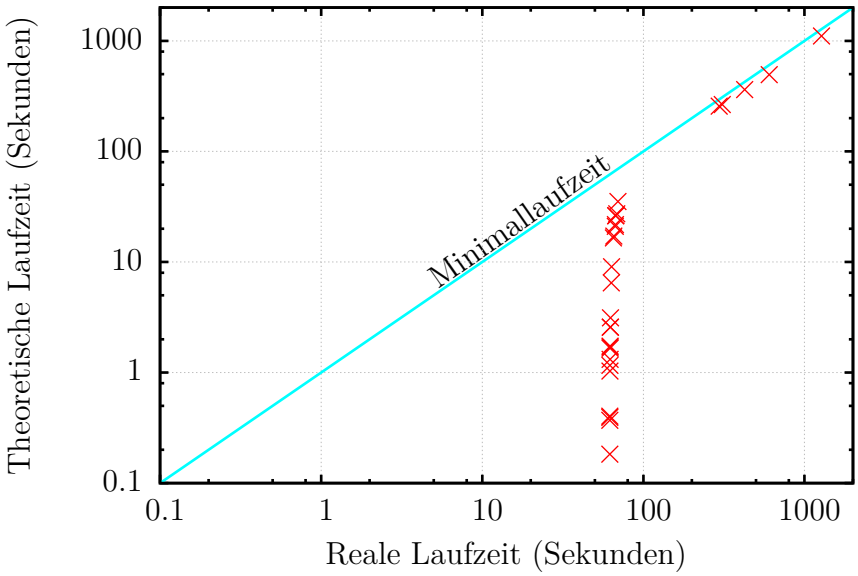

Abbildung 5.11: Vergleich der theoretischen und tatsächlichen Laufzeit von Kampagnen mit verschiedenen Benchmarks, 10000 Experimenten und 16 FI-CPUs. Beide Achsen mit logarithmischer Skalierung

mit einer FI-CPU, die das Register-Byte-Fehlermodell unterstützt, benötigt mehr als doppelt so viele Ressourcen wie die Variante ohne die Mehrbitfehlerunterstützung.

5.2.1 Laufzeit der Kampagnen

Die CPUs der Fehlerinjektionsplattform laufen auf einem XC7VX485T-FPGA mit 66,6 MHz, wobei auf diesem FPGA bis zu 16 Experimente zur selben Zeit ausgeführt werden können. Neben der eigentlichen Laufzeit eines Benchmarks kommen pro Experiment noch diverse Verwaltungsaufgaben im globalen FI-Controller hinzu.

Abbildung 5.11 stellt die realen Kampagnenlaufzeiten den theoretischen Zeiten gegenüber, welche sich aus der folgenden Rechenvorschrift ergeben:

$$t_{\text{theoretisch}} = \frac{\text{Benchmarklaufzeit in Takten}}{66.6 * 10^6 \text{ Hz} * 16 \text{ CPUs}} * \text{Anz. Experimente} \tag{5.2}$$

Diese theoretische Laufzeit vernachlässigt die Verwaltungsaufgaben des globalen FI-Controllers sowie die Situation, dass nicht alle FI-CPUs ausgelastet sind. Für kleine Benchmarklaufzeiten weicht die reale Zeit stark von der theoretischen ab. Bei 10000 Experimenten pro Kampagne ist eine Mindestlaufzeit von ca. 60 Sekunden festzustellen, egal wie kurz der Benchmark ist. Darüber hinaus bewegen sich die realen Laufzeiten in derselben Größenordnung nah an den theoretischen Werten. Ganz erreicht werden diese nicht, da der Overhead durch die Verwaltungsaufgaben natürlich weiterhin vorhanden ist.

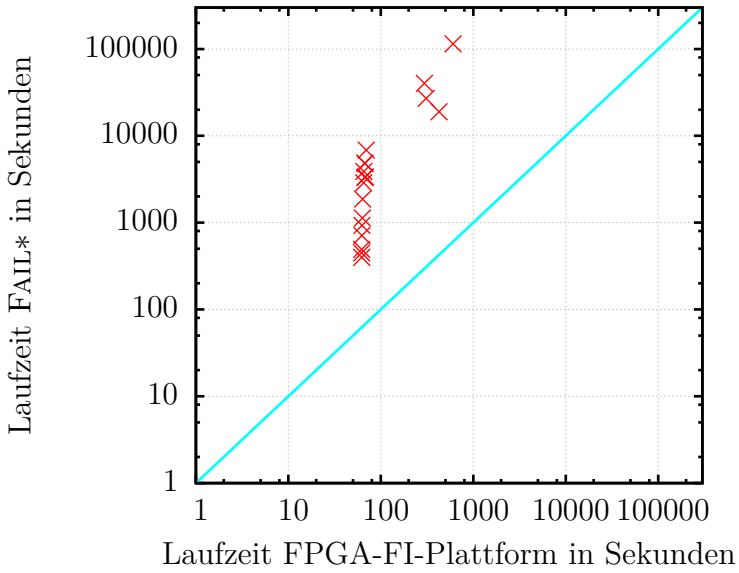

Abbildung 5.12: Vergleich der Kampagnenlaufzeiten der entwickelten Fehlerinjektionsplattform mit äquivalenten Kampagnen auf der FAIL*-Plattform, jeweils 10000 Experimente pro Kampagne. Beide Achsen mit logarithmischer Skalierung

Im Vergleich mit FAIL* sind die Kampagnen der entwickelten FPGA-Plattform wesentlich schneller. Abbildung 5.12 stellt die Laufzeiten verschiedener Fehlerinjektionskampagnen auf beiden Plattformen gegenüber. FAIL* lastete für diesen Test ein System mit 4 Intel Xeon E5-4640-Prozessoren für die angegebene Zeit aus. Die Kampagnen, die mit der FPGA-Plattform in unter 70 Sekunden durchlaufen, benötigen mit FAIL* bis zu zwei Stunden (*bubblesort*). Eine Kampagne, welche den *mibench-bitcount*-Benchmark testet, wird auf der FPGA-Plattform in ca. 10 Minuten abgearbeitet, benötigt mit FAIL* auf dem oben genannten System fast 32 Stunden. In Tabelle 5.3 sind die Laufzeitwerte von Kampagnen einiger Benchmarks sowie der Laufzeitfaktor zwischen FAIL* und der FPGA-Plattform aufgelistet. Der Faktor bewegt sich im Bereich 44-190. Betrachtet werden in dieser Tabelle nur die lang laufenden Kampagnen, bei denen die Laufzeit auf der FPGA-Plattform nicht hauptsächlich durch die Verwaltungsaufgaben des globalen FI-Controllers verursacht wird.

FAIL* ist nur in der Lage, Fehler in die CPU Register oder den Speicher zu injizieren. Um Fehler auch in weniger abstrakte Fehlermodelle zu injizieren, wurde im Entwurfskapitel 3.2 alternativ zu der FPGA-Plattform die Möglichkeit der Umsetzung mittels eines HDL-Simulators erwähnt. Der HDL-Code der FPGA-Plattform ist auch im Simulator lauffähig, sodass die Laufzeit des Simulators betrachtet werden kann.

Die Simulation eines Systems mit einer FI-CPU und mit Unterstützung des Register-Byte-Fehlermodells benötigt auf einem Intel Core i5-4570 31 Minuten Wallclock-Zeit,

Benchmark	Laufzeit FI-Plattform	Laufzeit theoretisch	Laufzeit FAIL*	Faktor FAIL*/FI-Plattf.
delta-bubblesort-64accu	295	257	40282	136
delta-quicksort-64accu	309	265	27015	87
mibench-bitcount	604	494	114500	190
mibench-susan	426	363	18927	44
bubblesort	69	35	6831	98

Tabelle 5.3: Laufzeitvergleich von Fehlerinjektionskampagnen repräsentativer Benchmarks auf der FPGA-FI-Plattform sowie auf Basis von FAIL*. Alle Angaben in Sekunden

	1	2	3	4	5	6	7	8
no error:	86408	86407	86408	86409	86408	86407	86407	86408
lockup:	658	658	658	658	658	658	658	658
cpu timeout:	8848	8848	8848	8848	8848	8848	8848	8848
result diff:	4086	4087	4086	4085	4086	4087	4087	4086

Tabelle 5.4: Ergebnisse von 8 Durchläufen derselben Fehlerinjektionskampagne mit 100000 Experimenten

um 10ms Systemlaufzeit bei 100 MHz zu simulieren. Wird dies umgerechnet auf eine Sekunde Systemlaufzeit bei 66,6 MHz, erhält man eine Simulationslaufzeit von ca. 125000 Sekunden oder ca. 34 Stunden.

Das simulierte System ist für die Ausführung auf einem FPGA optimiert. Wahrscheinlich ließe sich die Wallclock-Zeit mit einem für den Simulator optimierten Fehlerinjektionssystem deutlich verringern, außerdem ist die Simulation von vielen Fehlerinjektionsexperimenten gut parallelisierbar. Eine Verbesserung der Laufzeit um den Faktor 125000 erscheint jedoch ohne massive Parallelisierung und den damit verbundenen Kosten nicht realistisch.

5.2.2 Reproduzierbarkeit

Die Fehlerinjektionskampagnen sind leider nicht vollständig reproduzierbar. Tabelle 5.4 zeigt die Ergebnisse von 8 Durchläufen einer Fehlerinjektionskampagne mit 100000 Experimenten des *bubblesort*-Benchmarks unter Benutzung der Gate-Level-Fehlerinjektion und desselben Zufallsseeds. Es wäre zu erwarten, dass bei jedem Durchlauf dasselbe Ergebnis entsteht, stattdessen schwanken die Ergebnisse minimal. Eine Ursache für die Abweichungen konnte nicht gefunden werden. Eine Abweichung bei zwei Ergebnissen bei 100000 durchgeführten Experimenten ist jedoch statistisch nicht relevant, sodass die Ergebnisse dennoch benutzbar sind.

5.3 Aussagekraft der Ergebnisse

Auch wenn die vorgestellten Ergebnisse teilweise recht eindeutig sind, muss man diese aufgrund der Rahmenbedingungen kritisch betrachten. Die zeitliche Verteilung, mit der Neutronen in der Atmosphäre auf die CPUs treffen, kann, wie in dieser Arbeit abgebildet, als gleichverteilt angenommen werden. Diese Gleichverteilung trifft jedoch sicher nicht auf die dadurch ausgelösten Fehler zu. Es gibt Komponenten innerhalb der CPU, die aus baulichen Gründen besser abgeschirmt sind als andere, und somit weniger fehleranfällig sind.

Die benutzte CPU ist eine Microcontroller-CPU, die mit sehr wenigen Ressourcen auskommt, dafür nur einen kleinen Leistungsumfang abdeckt. Es ist fraglich, ob die in dieser Arbeit ermittelten Ergebnisse zum Beispiel auch auf moderne Serverprozessoren übertragbar sind.

Ein weiterer Kritikpunkt ist die relativ geringe Anzahl an Benchmarks, aus denen die Schlussfolgerungen gezogen wurden. Leider ist der Aufwand, neue Benchmarks auf eine Plattform zu portieren, welche nur aus der CPU und einem kleinem Speicher besteht, teilweise relativ hoch. Außerdem gibt es nur wenige Benchmarks, die aufgrund der beschränkten Ressourcen geeignet und frei verfügbar sind.

5.4 Zusammenfassung

Im ersten Abschnitt des Kapitels wurde die ursprüngliche Fragestellung nach dem Gütevergleich verschiedener Fehlermodelle beantwortet. Im Vergleich zu dem Gate-Modell lassen die Ergebnisse von Fehlerinjektionskampagnen mit dem Flip-Flop-, Register- und Register-Byte-Modell dieselben Schlüsse zu, wenn die EAF-Metrik benutzt wird. Außerdem weisen Kampagnen, die mit FAIL∗ ausgeführt wurden, überwiegend ähnliche Ergebnisse auf. Bei FAIL∗ besteht im Zusammenhang mit der EAF-Metrik jedoch das Problem, dass dieses andere Benchmarklaufzeiten simuliert als die FPGA-Plattform. Die Vergleichbarkeit der Fehlermodelle wurde sowohl qualitativ beschrieben und auch mittels Zusammenhangsmaßen analysiert. Anhand der Rangfolge der Benchmarks sowie zweier gehärteter Benchmarks wurde gezeigt, dass die Fault-Coverage-Metrik sich nicht dazu eignet, Benchmarks mit unterschiedlicher Laufzeit miteinander zu vergleichen.

Im zweiten Abschnitt wurde die entwickelte Fehlerinjektionsplattform evaluiert. Der Verbrauch der FPGA-Ressourcen steigt bei zunehmender Anzahl von Partitionen erwartungsgemäß an. Die Laufzeit der Fehlerkampagnen ist gegenüber alternativen, Simulator-basierten Plattformen um mehrere Größenordnungen schneller. Außerdem wurde festgestellt, dass die Experimente gut reproduzierbar sind, wenn auch nicht hundertprozentig. Abschließend wurden einige Einschränkungen bezüglich der Aussagekraft der Ergebnisse festgestellt. Das Überwinden dieser Einschränkungen sprengt jedoch den Rahmen dieser Arbeit und könnte daher Grundlage für weitere Arbeiten sein.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine Fehlerinjektionsplattform auf Basis eines ARM Cortex-M0-Prozessors entworfen und implementiert, welche für allgemeine Fehlerinjektionsexperimente verwendbar ist. Die Plattform ist in der Lage, transiente CPU-Fehler auf Gate-, Flip-Flop, Register (Bit und Byte) sowie Register-On-Write-Ebene zu injizieren. Dadurch ist es möglich, Vergleiche zwischen diesen Modellen durchzuführen und die Güte dieser Fehlermodelle zu bewerten. Ebenfalls können entwickelte SIHFT-Techniken umfassend getestet werden, um zu verhindern, dass diese nur mit dem bei der Entwicklung benutzten Fehlermodell gut wirken. Die auf diese Plattform portierten Benchmarks wurden zusätzlich auf die Fail∗-Plattform portiert, um diese in den Vergleich mit einzubeziehen. Die entwickelte Injektionsplattform ist auf einem FPGA lauffähig und gegenüber anderen, Simulator-basierten Alternativen sehr performant in der Durchführung der Experimente und einfach zu benutzen.

Die ermittelten Ergebnisse bezüglich der Vergleichbarkeit von Fehlerinjektionsexperimenten auf verschiedenen Ebenen sind teilweise recht eindeutig, aber auch teils etwas unklar. Ein starker Zusammenhang ist zwischen den Fehlermodellen feststellbar, die auf der FPGA-Plattform in Gates, Flip-Flops und CPU-Register injizieren. Die CPU-Registerfehlerinjektion von Fail∗ passt überwiegend auch in das Schema, jedoch gibt es dort mit dem *delta-quicksort*-Benchmark einen deutlichen Ausreißer. Die Ergebnisse der Experimente mit dem Register-On-Write-Fehlermodell korrelieren hingegen nur wenig mit denen der anderen Modelle, sodass dieses Modell nicht geeignet zu sein scheint, um CPU-Fehler zu simulieren.

Möchte man die Fehlertoleranz mehrerer Programme vergleichen, ist die Metrik ein sehr wichtiger Faktor. Besonders gut wird dies anhand der vorgestellten Delta-Benchmarks deutlich, bei denen die gehärteten Varianten laut der Fault-Coverage wesentlich besser abschneiden. Nach der EAF-Metrik ist ein gehärteter Benchmark aufgrund des Laufzeitzuwachs durch die Härtungsmechanismen jedoch sogar schlechter als seine ungehärtete Variante.

Weitergehend wäre es interessant, den Einfluss der CPU-Mikroarchitektur auf die Vergleichbarkeit der Fehlermodelle zu untersuchen. Hierfür muss eine andere CPU auf diese Plattform portiert werden. Anbieten würde sich sowohl eine sehr minimalistische CPU wie die ZPU oder eine sehr komplexe CPU, wie der LEON3.

Um die abstrakteren Fehlermodelle zu verbessern, müsste die Propagierung von Fehlern der detailgenauen Modelle auf die Saboteure betrachtet werden, die durch die abstrakteren Modelle bereitgestellt werden. Auch ist eine Untersuchung besonders kritische Fehlerpunkte denkbar. Im Kontext von SIHFT könnten dadurch Fehlertoleranzmechanismen entwickelt werden, die diese kritischen Punkte besonders absichern.

Literaturverzeichnis

[1] Cha, Hungse ; Rudnick, Elizabeth M. ; Patel, Janak H. ; Iyer, Ravishankar K. ; Choi, Gwan S.: A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults. In: *IEEE Trans. Computers* 45 (1996), Nr. 11, 1248–1256. http://dx.doi.org/10.1109/12.544481. – DOI 10.1109/12.544481

[2] Iyer, Ravishankar K. ; Rossetti, David J.: A Measurement-Based Model for Workload Dependence of CPU Errors. In: *Computers, IEEE Transactions on* C-35 (1986), June, Nr. 6, S. 511–519. http://dx.doi.org/10.1109/TC.1986.5009428. – DOI 10.1109/TC.1986.5009428. – ISSN 0018–9340

[3] Baumann, Robert: Soft Error Rate Overview and Technology Trends. In: *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, 2002

[4] Koopman, Phil: *A Case Study of Toyota Unintended Acceleration and Software Safety*. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf. Version: 2014

[5] Yeh, Yen-Chih: Triple-triple redundant 777 primary flight computer. In: *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE* Bd. 1, 1996, S. 293–307 vol.1

[6] Goloubeva, Olga ; Rebaudengo, Maurizio ; Reorda, Matteo S. ; Violante, Massimo: *Software-Implemented Hardware Fault Tolerance*. Springer, 2006. – I–XI, 1–224 S. – ISBN 978–0–387–26060–0

[7] Nightingale, Edmund B. ; Douceur, John R. ; Orgovan, Vince: Cycles, Cells and Platters: An Empirical Analysisof Hardware Failures on a Million Consumer PCs. In: *Proceedings of the Sixth Conference on Computer Systems*. New York, NY, USA : ACM, 2011 (EuroSys '11). – ISBN 978–1–4503–0634–8, 343–356

[8] Sieh, Volkmar ; Tschache, O. ; Balbach, F.: VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions. In: *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, 1997. – ISSN 0731–3071, S. 32–36

[9] Guan, Qiang ; DeBardeleben, N. ; Blanchard, S. ; Fu, Song: F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014. – ISSN 1530–2075, S. 1245–1254

[10] Yim, Keun S. ; Kalbarczyk, Zbigniew ; Iyer, Ravishankar K.: Measurement-based analysis of fault and error sensitivities of dynamic memory. In: *DSN*, IEEE, 2010. – ISBN 978–1–4244–7501–8, 431-436

[11] Kuvaiskii, Dmitrii ; Fetzer, Christof: Δ-encoding: Practical Encoded Processing. In: *Proceedings of The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*, IEEE Computer Society, June 2015

[12] Schirmeier, Horst ; Borchert, Christoph ; Spinczyk, Olaf: Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors. In: *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, IEEE Computer Society Press, Juni 2015, S. 319–330

[13] Leveugle, R. ; Calvez, A. ; Maistri, P. ; Vanhauwaert, P.: Statistical fault injection: Quantified error and confidence. In: *2009 Design, Automation Test in Europe Conference Exhibition*, 2009. – ISSN 1530–1591, S. 502–506

[14] Sanda, Pia N. ; Kellington, Jeffrey W. ; Kudva, P. ; Kalla, R. ; McBeth, R. B. ; Ackaret, J. ; Lockwood, R. ; Schumann, J. ; Jones, C. R.: Soft-error resilience of the IBM POWER6 processor. In: *IBM Journal of Research and Development* 52 (2008), May, Nr. 3, S. 275–284. `http://dx.doi.org/10.1147/rd.523.0275`. – DOI 10.1147/rd.523.0275. – ISSN 0018–8646

[15] Cho, Hyungmin ; Mirkhani, S. ; Cher, Chen-Yong ; Abraham, J.A. ; Mitra, S.: Quantitative evaluation of soft error injection techniques for robust system design. In: *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, 2013. – ISSN 0738–100X, S. 1–10

[16] Gaisler, Aeroflex: *LEON3 Processor*. `http://www.gaisler.com/index.php/products/processors/leon3`

[17] Schirmeier, Horst ; Hoffmann, Martin ; Dietrich, Christian ; Lenz, Michael ; Lohmann, Daniel ; Spinczyk, Olaf: FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In: *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, IEEE Computer Society Press, September 2015, S. 245–255

[18] Mentor Graphics: *ModelSim*. `https://www.mentor.com/products/fv/modelsim/`

[19] Williams, Stephen: *Icarus Verilog*. `http://iverilog.icarus.com/`

[20] Saggese, G. P. ; Vetteth, A. ; Kalbarczyk, Z. ; Iyer, Ravishankar: Microprocessor sensitivity to failures: control vs. execution and combinational vs.

sequential logic. In: *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, 2005, S. 760–769

[21] Kuon, I. ; Rose, J.: Measuring the Gap Between FPGAs and ASICs. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26 (2007), Feb, Nr. 2, S. 203–215. http://dx.doi.org/10.1109/TCAD.2006.884574. – DOI 10.1109/TCAD.2006.884574. – ISSN 0278–0070

[22] Synopsys Inc.: *Design Compiler 2010*. http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx

[23] Wolf, Clifford: *Yosys Open SYnthesis Suite*. http://www.clifford.at/yosys/, 2016

[24] Wikipedia User Tibor89: *Full-adder logic diagram*. https://en.wikipedia.org/wiki/Adder_(electronics), 2006 (accessed Nov 8, 2015)

[25] OpenCores.org: *OR1200 OpenRISC Processor*. http://opencores.org/or1k/OR1200_OpenRISC_Processor,

[26] OpenCores.org: *OpenRISC 1000 Architecture Manual*. https://github.com/openrisc/doc/blob/master/openrisc-arch-1.1-rev0.pdf, 2014

[27] OpenCores.org: *GCC port for OpenRISC 1000*. https://github.com/openrisc/or1k-gcc,

[28] OpenCores.org: *openMSP430*. http://opencores.org/project,openmsp430,

[29] Texas Instruments Inc.: *MSP430x1xx Family User's Guide*. http://www.ti.com/lit/ug/slau049f/slau049f.pdf, 2006

[30] Texas Instruments Inc.: *GCC - Open Source Compiler for MSP430 Microcontrollers*. http://www.ti.com/tool/msp430-gcc-opensource,

[31] OpenCores.org: *Amber ARM-compatible core*. http://opencores.org/project,amber,

[32] Arriëns, H.J. L.: *MB-Lite+ User Guide*. http://ens.ewi.tudelft.nl/~huib/free/MB-Lite_Plus_UG_v12.1.2.pdf,

[33] Informatik Lehrstuhl 12 Arbeitsgruppe Eingebettete Systemsoftware TU Dortmund: *Laufzeitplattform für anwendungsspezifische verteilte Architekturen*. http://ess.cs.tu-dortmund.de/DE/Research/Projects/LavA/index.html,

[34] Harboe Øyvind: *The Zylin ZPU*. https://github.com/zylin/zpu,

[35] Clarke, Peter: *European Space Agency launches free Sparc-like core.* http://www.eetimes.com/document.asp?doc_id=1214267,

[36] Aeroflex Gaisler: *LEON3FT-RTAX Fault-tolerant Processor.* http://gaisler.com/index.php/products/components/leon3ft-rtax,

[37] ARM Ltd.: *Cortex-M0 Processor.* http://www.arm.com/products/processors/cortex-m/cortex-m0.php,

[38] Xilinx (Hrsg.): *7 Series FPGAs Configurable Logic Block.* : Xilinx, 11 2014. (UG474) . – v1.7

[39] Wikipedia User jjbeard: *Taktflankengesteuertes D-Flipflop nach dem Master-Slave-Prinzip.* https://de.wikipedia.org/wiki/Flipflop#/media/File:D-Type_Flip-flop_Diagram.svg, 2006 (accessed Mar 10, 2016)

[40] Patavalis, Nick: *picocom - Minimal dumb-terminal emulator.* https://github.com/npat-efault/picocom,

[41] Marsaglia, George: Random Number Generators. In: *Journal of Modern Applied Statistical Methods*, 2003

[42] Guthaus, M. R. ; Ringenberg, J. S. ; Ernst, D. ; Austin, T. M. ; Mudge, T. ; Brown, R. B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop.* Washington, DC, USA : IEEE Computer Society, 2001 (WWC '01). – ISBN 0–7803–7315–4, 3–14

[43] Sellung, Marcel: *Vergleich von Hardwarefehlermodellen bei Fehlerinjektions-Experimenten.* 2015

[44] Tukey, J.W.: *Exploratory Data Analysis.* Addison-Wesley Publishing Company, 1977 (Addison-Wesley series in behavioral science). https://books.google.co.uk/books?id=UT9dAAAAIAAJ. – ISBN 9780201076165

[45] Gilpin, Andrew R.: Table for Conversion of Kendall's Tau to Spearman's Rho Within the Context of Measures of Magnitude of Effect for Meta-Analysis. In: *Educational and Psychological Measurement* 53 (1993), Nr. 1, 87–92. http://dx.doi.org/10.1177/0013164493053001007. – DOI 10.1177/0013164493053001007

[46] Gibbons, Jean D.: *Nonparametric Methods for Quantitative Analysis.* 1980

Abbildungsverzeichnis

Eidesstattliche Versicherung

_____ _____

Name, Vorname Matr.-Nr.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem Titel

_____

_____

_____

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____ _____

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -)

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin“) zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

_____ _____

Ort, Datum Unterschrift