technische universität
dortmund

Bachelorarbeit

Reduzierung der Laufzeitkosten des EDDI-Fehlerdetektionsverfahrens durch selektive Anwendung

Fabian Sieber
21. Juni 2019

Gutachter:
Dr.-Ing. Horst Schirmeier
Prof. Dr.-Ing. Olaf Spinczyk

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

Zusammenfassung

Kosmische Strahlung kann – nicht nur in exotischen Umgebungen wie dem Weltraum – transiente Hardwarefehler auslösen und sich in Berechnungsfehlern oder Bit-Kippern in CPUs oder der Speicherhierarchie niederschlagen. Neben teuren und energiehungrigen Hardwarelösungen wurde dieses Problem in den letzten Jahren auch durch reine Softwarelösungen [4] angegangen, für die das Verfahren Error Detection by Duplicated Instructions (EDDI) [7] als klassisches Beispiel steht. Da EDDI fast alle Programminstruktionen dupliziert und zusätzliche Vergleichsoperationen einfügt, geht das Verfahren mit sehr hohen Laufzeitkosten einher [7], was in manchen Fällen sogar zusätzlich negativ auf die Fehlertoleranzeigenschaften zurückfallen kann [4, 10].

Im Rahmen dieser Bachelorarbeit soll daher EDDI selektiv auf einzelne Programmteile angewandt werden, so dass trotz Reduktion des Rechenaufwands und der Laufzeit eine möglichst hohe Fehlertoleranzwirkung erreicht wird. Dazu sollen verschiedene Detektoren-Selektionskriterien erprobt werden: In eine vollständig mit EDDI abgesicherte Programmvariante sollen Fehler injiziert und festgestellt werden, welche Detektoren die höchsten Erkennungsraten erreichen; weitere mögliche Kriterien sind die Ausführungshäufigkeit der Detektoren oder die mittlere Fehlererkennungs-Latenz. Die Bewertung mit Hilfe von geeignet gewählten Benchmarkprogrammen soll dabei mit dem Fehlerinjektionswerkzeug FAIL* [11, 9] erfolgen, das unter Verwendung der Metrik Extrapolated Absolute Failure Count (EAFC) [10] die durch EDDI erhöhte Programmlaufzeit miteinbezieht.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Eingebettete Systeme finden immer mehr Anwendung im alltäglichen Leben. Angefangen bei Smart-Home-Leuchten, Fitness-Armbändern und Smartphones, bis hin zu stetig intelligenter werdenden Autos. Während Fahrzeuge früher nur einfache Funktionen wie Fensterheber oder auch einen Tempomaten besaßen, besitzen sie heutzutage zum Beispiel Spurhalte- und Abstandshalteassistenten. Um diese Funktionen bereitzustellen wird mehr Hardware benötigt, welche zusätzliches Gewicht für das Fahrzeug, weitere Kosten für die Produktion und einen erhöhten Stromverbrauch bedeuten. Da Verbrennungsfahrzeuge keine konstante Stromquelle besitzen und Elektrofahrzeuge durch den übermäßigen Stromverbrauch solcher Funktionen ihre Reichweite einbüßen, ist dieser Verbrauch gering zu halten. Deshalb wurden die Systeme kleiner und effizienter, sodass sie anfälliger für äußere Einflüsse wie Hitze oder Strahlung wurden. Ein durch diese Einflüsse provoziertes Fehlverhalten ist so gut wie möglich zu verhindern, denn ein solches Verhalten von sicherheitsrelevanten Funktionen kann tödlich enden. Wenn ein Fahrzeug mit Spurhalteassistent nach links auf die Gegenfahrbahn abweicht, kann dies einen seitlich versetzten Frontalzusammenstoß provozieren. Diese sind als besonders gefährlich bekannt und ihre fatalen Folgen zeigen sich in jedem Crashtest. Ebenso gefährlich ist die Fehlfunktion eines Abstandshalte- und Notbrems-Assistenten beim Auffahren eines Lastkraftwagens auf ein Stauende. Diese äußeren Einflüsse treten aufgrund der fehlenden Atmosphäre im Weltall besonders häufig auf, daher befasst sich die Raumfahrt bereits seit vielen Jahren mit diesem Problem. Dort wird das Problem jedoch meistens durch strahlungresistente Hardware oder auch dreifach vorhandene Systeme gelöst. Für die Automobilindustrie ist diese Lösung jedoch nicht praktikabel, da diese Hardware mehr Platz, Strom und Geld kostet, was alleine bei der Stückzahl der Funktionen und Fahrzeuge nicht akzeptierbar ist. Deshalb ist eine Lösung dieses Problems ohne zusätzliche Hardware erwünscht. Das *Error Detection by Duplicated Instructions*-Verfahren [7] gilt als eine durch Software umgesetzte Lösung dieses

Problems, welches 2002 von Oh, Shirvani und McCluskey vorgestellt und getestet wurde. [7] Das EDDI-Verfahren beruht darauf, jeden Berechnungsschritt doppelt auszuführen und die Ergebnisse zu vergleichen. Sollte dann ein transienter Fehler auftreten, wird dieser von dem Programm durch den Vergleich erkannt. Dabei ist es unwahrscheinlich, dass sowohl beide Berechnungen verfälscht werden und besonders unwahrscheinlich, dass beide Berechnungen auf exakt gleiche Art verfälscht werden. Allerdings muss dabei beachtet werden, dass nun doppelt so viele Berechnungen vorhanden sind, die verfälscht werden können, und diese Berechnungen und der Vergleich zusätzliche Ressourcen benötigen.

1.2 Ziele

In dem Paper von Oh, Shirvani und McCluskey wurde für die Bewertung die *Fault-Coverage-Factor*-Metrik genutzt. Da diese jedoch nicht die durch EDDI entstandene zusätzliche Rechenzeit miteinbezieht, wurde die *Extrapolated-Absolute-Failure-Count*-Metrik[10] genutzt und das Verfahren mit ihr neu bewertet. [3] Die EAFC-Metrik wird in dieser Arbeit zur Bewertung genutzt. Dabei wurde jedoch immer das komplette Programm „*eddifiziert*“. Dies bedeutet, dass das EDDI-Verfahren auf das ganze Programm angewandt wurde. Um die Ressourcenkosten zu verringern, könnte nur eine ausgewählte Menge der Berechnungen verdoppelt und mit einem Vergleich versehen werden. Außerdem sorgt die verkürzte Programmlaufzeit dafür, dass die Fehler nicht auftreten können, welche in der vollständig eddifizierten Variante erst nach der Terminierung der verkürzten Variante auftreten würden. Das Ziel dieser Arbeit ist es, eine selektive Variante des Verfahrens zu entwerfen und mehrere Auswahlmöglichkeiten der Detektoren zu erproben. Anhand dieser Auswahlmöglichkeiten soll das Verfahren dann bewertet werden.

1.3 Überblick

Zunächst werden in Kapitel 2 die notwendigen Grundlagen vorgestellt. Besonders relevant für den weiteren Verlauf dieser Abschlussarbeit sind die unterschiedlichen Fehlertypen und der Begriff der Fehlerinjektion. Außerdem wird das EDDI-Verfahren näher erläutert und der LLVM-Compiler vorgestellt. Hinzu kommen die zur Bewertung genutzten Werkzeuge FAIL* und eCos. Danach wird in Kapitel 3 ein Entwurf des selektiven Verfahrens vorgestellt und wie es, wie die bestehende Implementation des EDDI-Verfahrens [3], in LLVM umgesetzt werden kann. Nachdem die Grundlagen und das zentrale Problem vorgestellt wurden, folgt in Kapitel 4 die Vorstellung und Implementierung des LLVM-Passes, welcher das EDDI-Verfahren selektiv auf die Testprogramme anwendet.

In Kapitel 5 werden diese Testprogramme dann vollständig „eddifiziert“ und mit selektiv angewandtem EDDI-Verfahren generiert und mit eCos verknüpft, sodass diese zur Evaluation gemessen werden können.

Kapitel 2

Grundlagen

Dieses Kapitel dient dazu, ein grundlegendes Verständnis über Fehler, Fehlertoleranz und die Prinzipien der Fehlerdetektion zu schaffen. Dazu wird zuerst auf das Auftreten und die unterschiedlichen Arten von Fehlern eingegangen. Darauf folgen die Erläuterung des Begriffes der Fehlertoleranz und die Vorstellung des *Error Detection by Duplicated Instructions*-Verfahrens. Anschließend wird der Begriff der Fehlerinjektion eingeführt und welche Bewertungsmetriken in der Fehlerinjektion genutzt werden. Gegen Ende dieses Kapitels folgen Einführungen in das Fehlerinjektions-Framework FAIL*, welches zum Testen der Erweiterung genutzt wird, und die LLVM-Architektur, welche für die bisherige Implementierung und die Erweiterung in dieser Arbeit genutzt wurde.

2.1 Fehler

Fehler können auf unterschiedliche Arten entstehen, auch wenn die Software und Hardware dazu konzipiert wurden, zuverlässig korrekte Ergebnisse zu liefern. Um Fehler zu erkennen, sollte zuerst klar sein, auf welche Art Fehler überhaupt auftreten können. Fehler sind generell in zwei Bereich zu unterteilen: Softwarefehler und Hardwarefehler.

Unter Softwarefehlern versteht man Fehler, welche nicht erst während der Laufzeit des Programms entstehen, sondern bereits bei der Entwicklung der Software. Das Programm verhält sich normal, bis es die Stelle erreicht, an dem der in der Entwicklung entstandene Programmierfehler Einfluss auf den weiteren Verlauf des Programms hat [3]. Dort kommt es zum Fehlverhalten. Da der Fehler in der Entwicklung entstanden ist, tritt er bei gleichen Bedingungen in der Regel zum gleichen Zeitpunkt auf und verändert den weiteren Verlauf des Programms auf die gleiche Art [3]. Softwarefehler sind auf die zugrundeliegende Software zurückzuführen und werden in dieser Arbeit nicht weiter betrachtet.

Hardwarefehler können durch fehlerhafte Komponenten des Systems entstehen, auf dem das Programm läuft, oder auch durch äußere Einflüsse auf die Hardware. Deshalb lassen diese sich wieder in zwei Kategorien unterteilen. Einerseits gibt es *permanente Hardwa-*

refehler. Diese treten auf, wenn Hardware verwendet wird, die entweder bereits defekt ausgeliefert wurde, oder auch im Betrieb beschädigt wurde. Letzteres kann beispielsweise durch Übertaktung des Prozessors eintreten, wenn dieser dadurch überhitzt. Bei permanenten Hardwarefehlern ist es wie bei Softwarefehlern üblich, dass sie bei wiederholter Ausführung erneut auftreten. Sie können allerdings auch komplett oder nur zeitweise wieder verschwinden. Auch diese Fehler werden hier nicht weiter behandelt.

Andererseits gibt es *transiente Hardwarefehler*. Diese können auch bei fehlerfreier Soft- und Hardware auftreten. Im Gegensatz zu Softwarefehlern und permanenten Hardwarefehlern treten diese ohne Vorzeichen erst während der Laufzeit auf. Deshalb kann dieser bei Wiederholung zu unterschiedlichen Zeitpunkten mit unterschiedlichen Ergebnissen auftreten. Ein Beispiel dafür sind die Auswirkungen von Strahlung auf die Hardware. Beispielsweise kann in der CPU des Systems ein *fault* auftreten, und zwar dass, durch das Treffen eines Teilchens auf die CPU, an einem Transistor ein Bit-Flip aufgetreten ist [3]. Dies bedeutet, dass der Wert eines Bits invertiert wurde, also eine 0 auf 1 gesetzt wurde respektive umgekehrt. Dadurch wird der weitere Verlauf des Programms beeinflusst und dies endet nicht zwingend in einem Absturz des Programms. Ebenso wie durch defekte Hardware auftretende Fehler können diese Fehler bei Wiederholung verschwunden sein oder auch abweichen. Solche zufällig auftretenden Fehler werden *transiente Fehler* [6] genannt. Das Nachstellen dieser Situation ist mit hohen Kosten verbunden und kann erneut viel Zeit benötigen, bevor ein Fehler auftritt. Zusätzlich existiert kein detailliertes CPU-Hardware-Modell für diesen Zweck, weshalb diese Fehler durch Bit-Flips in den CPU-Registern modelliert werden. Im Folgenden wird genauer auf die unterschiedlichen Fehlertypen eingegangen.

2.1.1 Fehlertypen

Um die durch den Fehler auftretenden Folgen für den Programmablauf bewerten und den Fehler beheben zu können, muss zwischen den Fehlertypen genauer unterschieden werden. Zur Verdeutlichung der Unterschiede bietet sich ein Beispiel eines fehlerhaften Programmablaufs an [3]:

Wie in der Erklärung der Softwarefehler beschrieben, läuft das Programm ordnungsgemäß bis zur fehlerhaften Stelle. Der dort auftretende Fehler sorgt für eine inkorrekte Berechnung, ein sogenannter Defekt (*fault*) [6] tritt auf. Wird dieser fehlerhafte Wert im weiteren Programmablauf genutzt, befindet sich das Programm in einem Fehlzustand (*Error*) [6]. In manchen Implementierungen kann durch eine danach stattfindende Berechnung der Fehler wieder korrigiert bzw. aufgehoben werden. Man spricht in diesem Fall von einem gutartigen Fehlzustand (*benign error*)[6]. Sollte der Fehler im weiteren Verlauf nicht aufgehoben werden, kommt es zu einem inkorrekten Ergebnis, welches zu einem Fehlverhalten (*failure*) [6] führt.

2.1.2 Fehlertoleranz

Es kann vorkommen, dass *transiente Fehler* erst nach Jahren der Nutzung auftreten und zu einem Fehlverhalten führen. Dies erschwert die Messung von transienten Fehlern. Fraglich ist dabei, inwiefern sich Systeme gegen diesen Typ von Fehlern schützen lassen. Das angestrebte Ziel ist ein *fehlertolerantes* System. Dies bedeutet, dass das System die Fehler selbst erkennt und, falls möglich, korrigiert oder beispielsweise die betroffene Berechnung erneut durchführt [3]. *Fehlertoleranz* beschreibt also die Fähigkeit eines Systems, trotz fehlerhafter Eingaben, abgeänderter Zwischenergebnisse oder inkorrekter Berechnungen einem Fehlverhalten zu widerstehen. Wie solche Fehler behoben werden, soll im weiteren Verlauf dieser Arbeit nicht genauer betrachtet werden. Wichtiger ist die *Detektion* dieser Fehler, da diese sonst überhaupt nicht behoben werden können. Ein Verfahren, welches die Detektion solcher Fehler ohne veränderte Hardware ermöglicht, heißt *Error Detection by Duplicated Instructions* [7]. *EDDI* ist Hauptbestandteil dieser Arbeit und soll durch die in Kapitel 3 beschriebenen Ideen angewandt werden.

2.2 EDDI

Das Error Detection by Duplicated Instructions-Verfahren, im Folgenden EDDI-Verfahren genannt, ist ein Fehlerdetektionsverfahren, das transiente Fehler erkennt, welche bereits in Kapitel 2.1 vorgestellt wurden.

Um die Funktionsweise des Verfahrens näher erläutern zu können, muss zuerst die Programmstruktur genauer bekannt sein. Jedes Programm enthält mindestens eine Funktion, welche sich wiederum in Basisblöcke unterteilen lässt. Diese Basisblöcke lassen sich erneut in einzelne Instruktionen unterteilen, welche für diesen Anwendungsfall das kleinste Glied der Programmstruktur sind. Basisblöcke (*BB*) sind als zusammenhängende Folge von Instruktionen definiert, die keine Sprünge enthalten. Zusätzlich muss die letzte Instruktion eine sogenannte terminierende Instruktion sein, dies ist entweder eine Branch-Instruktion (br) oder eine Return-Instruktion (ret) [7, 3].

Das EDDI-Verfahren unterteilt diese Basisblöcke jedoch weiter, und zwar in sogenannte speicherfreie Basisblöcke (*SBB*). Es kommt also die Bedingung hinzu, dass jeder Basisblock nur eine Store-Instruktion beinhalten darf, welche die vorletzte Instruktion des Basisblocks sein muss. Die letzte Instruktion ist dann eine Branch-Instruktion, die zum nächsten SBB führt. Sollte ein BB diese Bedingung verletzen, wird er in einen SBB und einen BB aufgeteilt, wobei die Branch-Instruktion des ersten SBB zum zweiten BB führt. Sollte der zweite BB diese Bedingung immer noch verletzen, wird dieser erneut aufgeteilt, ansonsten ist er bereits ein SBB [7, 3].

Als nächstes wird jede Instruktion innerhalb eines SBBs dupliziert, dabei wird die neu erzeugte Instruktion als *Schatten(-Instruktion)* bezeichnet. Bei der Duplizierung ist zu be-

Abbildung 2.1: Ablauf von Instruktionen ohne EDDI (links abgebildet) und mit EDDI (rechtsabgebildet) [3]

achten, dass die Schatten unabhängig von den Originalen arbeiten sollen. Sollten beispielsweise Datenabhängigkeiten zwischen zwei originalen Instruktionen bestehen, so werden diese für die Schatten übernommen. Allerdings dürfen währenddessen keine Datenabhängigkeiten zwischen den Originalen und den Schatten entstehen. Dadurch entstehen zwei von einander unabhängige, parallel ablaufende Programmabläufe, die die gleiche Funktionalität abdecken. Wenn beim Lauf des Programmes dann ein Fehler auftritt, existiert diese zweite Berechnung zur Kontrolle. Damit dieser Fehler dann aber auch detektiert wird, muss am Ende des SBBs eine Kontroll-Instruktion (*Check*) hinzugefügt werden. Diese vergleicht die Ergebnisse des Originals und des Schattens und meldet bei Ungleichheit den detektierten Fehler [7, 3].

Ein Beispiel eines solchen Ablaufs ist in Abbildung 2.1 zu sehen. Dabei ist auf der linken Seite der originale Ablauf mit drei Instruktionen (I1-I3) zu sehen. Auf der rechten Seite ist der „eddifizierte“ Ablauf, der neben den drei Instruktionen auch die drei dazugehörigen Schatten (S1-S3) und den Check enthält. Die Instruktion S1 ist also der Schatten zu I1, welcher exakt die gleiche Funktionalität enthält. Dies gilt für S2 und S3 analog. Auch wenn die Instruktionen nacheinander ablaufen, stellt dies die zwei parallel ablaufenden, voneinander unabhängigen Programmabläufe I1 bis I3 und S1 bis S3 dar. Dieses Beispiel ist, wie generell in dieser Arbeit verwendet, eine alternierende Duplizierung der Instruktionen. Eine weitere Möglichkeit der Umsetzung des EDDI-Verfahrens ist eine blockweise Duplizierung der Instruktionen. Der Unterschied besteht darin, dass die Instruktionen nicht direkt innerhalb des SBBs dupliziert werden, sondern dass der SBB dupliziert wird und das Ergebnis des Originals mit Ergebnis des Schatten-SBBs verglichen werden. Im Folgenden wird aufgrund der einfacheren Implementierung, besonders in Hinsicht auf die bestehende Implemenierung, nur die alternierende Duplizierung behandelt.

2.2.1 Laufzeit

Mit der Anwendung des EDDI-Verfahrens geht eine Erhöhung der Laufzeit des Programms einher. Dies ergibt sich bereits daraus, dass jede Instruktion einen Schatten erhält. Hinzu kommt der eingefügte Vergleich, welcher weitere Zeit in Anspruch nimmt. Dadurch ergibt sich grob betrachtet mindestens die doppelte Laufzeit. Welche Risiken dies birgt, wird in

Kapitel 2.3.2 näher erläutert und in Kapitel 3.2 wird genauer darauf eingegangen, inwiefern sich die Laufzeit durch Anwendung des Verfahrens verschlechtert.

2.2.2 Güte der Härtung

Da dieses Verfahren und insbesondere die selektive Anwendung dieses Verfahrens von dem eingegebenen Programm abhängig sind, lässt sich aufgrund der Vielfalt der Programme kein mathematischer Beweis anstellen. Um jedoch trotzdem eine Aussage über die Güte der Härtung des Programms gegen transiente Fehler und die Verringerung der Laufzeitkosten gegenüber der breitflächigen Anwendung des EDDI-Verfahrens treffen zu können, wird stattdessen mit mehreren Testprogrammen eine *Fehlerinjektion* durchgeführt.

2.3 Fehlerinjektion

Der Zeitpunkt des Auftretens von transienten Fehlern ist vorher nicht bestimmbar und kann auch erst nach Jahren der Nutzung eintreten. Daher ist ein einfaches Testen im Zielsystem nicht möglich. Um das Verfahren in annehmbarer Zeit zu testen, simuliert man diese Fehler im Rahmen von *Fehlerinjektionsexperimenten.*

2.3.1 Fehlerinjektionsexperimente

In einem Fehlerinjektionsexperiment wird während der Programmausführung die Hardware simuliert und dann zu einem vorher bestimmten Zeitpunkt ein Fehler injiziert. In diesem Fall bedeutet dies, dass in den Registern der CPU ein Bit-Flip durchgeführt wird. Stattdessen kann der Fehler in der ALU oder auch dem Arbeitsspeicher injiziert werden [3].

Der Ablauf ist wie folgt: Zuerst wird das Programm ohne injizierten Fehler durchlaufen, um einen Vergleichswert zu ermitteln. Dieser Lauf wird als *golden run* bezeichnet. Danach folgen viele *Fehlerinjektionsexperimente*, welches jeweils genau einen *Fehlerinjektionspunkt* besitzt. Ein Fehlerinjektionspunkt besitzt einen Zeitpunkt x und eine Stelle y in den Registern, an denen der Fehler auftreten soll. Das Programm läuft in dem Fehlerinjektionsexperiment normal bis zu dem Zeitpunkt x des zugehörigen Fehlerinjektionspunktes. Das Programm wird pausiert und an der Stelle y wird der Bit-Flip durchgeführt. Danach wird das Programm fortgesetzt und auf dessen Terminierung gewartet. Nachdem dies geschehen ist, wird das Ergebnis mit *golden run* verglichen und zur späteren Auswertung in der Datenbank aufgenommen. Die wiederholte Ausführung von Fehlerinjektionsexperimenten nennt man *Fehlerinjektionskampagne* [3]. Folgende Verhalten kann das Programm dann aufweisen:

1. Der Fehler wird vom Programm erkannt und eine Ausnahmebehandlung gestartet (Trap).

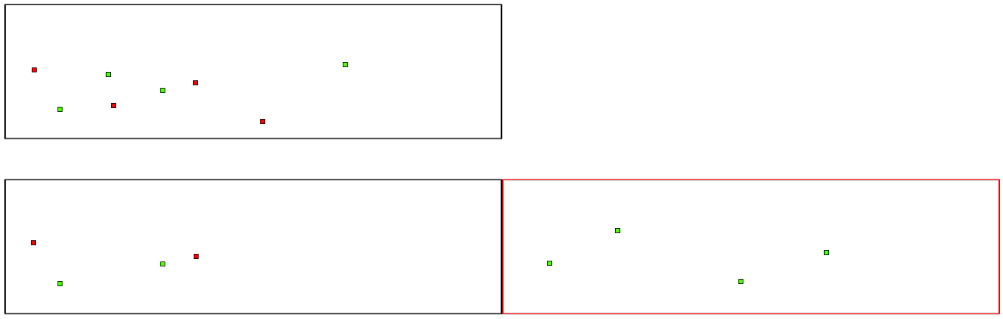

Abbildung 2.2: Beispiel für das Problem der FCF-Metrik [10]

2. Der Bit-Flip hat beispielsweise die Laufvariable einer Schleife verändert, sodass das Programm nie termininiert oder vom System abgebrochen wird (Timeout).

3. Der veränderte Wert findet keine Nutzung mehr und beeinflusst das Ergebnis des Programms nicht (benign error).

4. Das Programm terminiert mit einem fehlerhaften Ergebnis (*Silent Data Corruption/SDC*).

5. Der Fehler wird vom implementierten Härtungsverfahren erkannt (*detected Error*).

Das Auftreten von SDC ist hier der kritische Fall, da somit unentdeckte Fehlberechnungen stattfinden, die es zu verhindern gilt. Der erste und zweite Fall gelten als weniger bedeutend, da diese durch das Programm, respektive dem System, erkannt und behandelt werden. Deshalb gelten die ersten drei Fälle als erfolgreiche Experimente und nur der vierte Fall als fehlgeschlagen. Wenn eine Härtung des Programms durch ein Fehlerdetektionsverfahren, wie EDDI, vorgenommen wurde, existiert ein fünftes Verhalten. Dieses Verhalten wird als erfolgreich angesehen, da der Fehler erkannt wurde.

2.3.2 FCF und EAFC

In Kapitel 2.1.2 wurde der Begriff Fehlertoleranz erläutert. Um die Bewertung dieser Fehlertoleranz vorzunehmen, benötigt es einen Vergleichswert. Eine Möglichkeit ist der *Fault Coverage Factor (FCF):* $1 - \frac{F}{N}$[2]. N beschreibt dabei die Anzahl der durchgeführten Experimente und F die Anzahl der Experimente, die als fehlgeschlagen angesehen werden.

Da die Anwendung des EDDI-Verfahrens mit einer verlängerten Laufzeit des Programms einhergeht, können Fehler auftreten, wenn das originale Programm bereits terminiert hätte. Auch wenn die Fehler dadurch erkannt werden, erhöht dies die Wahrscheinlichkeit, dass überhaupt ein Fehler auftritt [3]. Hinzu kommt ein weiteres Problem, welches in der Abbildung 2.2 dargestellt ist. Gegeben sei ein Programm mit acht gemessenen Ergebnissen, von denen vier in SDC enden und vier unbeeinflusst enden. Mit der FCF-Metrik

ergibt dies folgenden Wert: $c = 1 - \frac{4}{8} = 1 - \frac{1}{2} = \frac{1}{2} = 50\%$. Wenn das Programm nun mit sogenannten *NOPs* gefüllt wird, Instruktionen die nichts tun, sodass die Laufzeit sich verdoppelt, verdoppelt sich dadurch der Fehlerraum. Der rote Bereich soll die eingefügten NOPs repräsentieren, dabei ist es jedoch egal, an welchen Stellen diese eingefügt werden. Zur einfacheren Veranschaulichung wurden sie hier an das Ende gesetzt. Wenn dort ein Fehler injiziert wird, kommt es zu keiner Beeinflussung, da die Instruktionen laut Definition keinen Einfluss auf den Rest haben. Da jedoch weiterhin nur acht Messwerte genommen werden, wird es im idealen Fall dazu kommen, dass nun vier Messwerte aus dem originalen Bereich genommen werden und vier Messwerte aus dem roten Bereich gezogen werden. Mit der FCF-Metrik erhält man nun folgenden Wert: $c = 1 - \frac{2}{8} = 1 - \frac{1}{4} = \frac{3}{4} = 75\%$. Durch die Erhöhung der Laufzeit ergibt sich somit ein besserer Wert, der die Fehlertoleranz des Programms repräsentieren soll, obwohl beide Varianten gleich anfällig für transiente Fehler sind [10].

Deshalb ist es sinnvoll, die Programmlaufzeit in die Bewertung der Fehlertoleranz miteinzubeziehen. Eine Metrik, die dies erfüllt, heißt *Extrapolated Absolute Failure Count (EAFC)*: $F_{\text{extrapolated}} = w \cdot \frac{F_{\text{sampled}}}{N_{\text{sampled}}}$ [10]. N beschreibt weiterhin die Anzahl der durchgeführten Experimente und F die Anzahl der gescheiterten Experimente. Zusätzlich existiert nun der Wert w. Dieser Wert sorgt dafür, dass die gemessenen Werte auf die Größe des Fehlerraums „hochgerechnet“ werden. Sei die Laufzeit des in Abbildung 2.2 dargestellten Beispiels 100 Zeiteinheiten und sei die Anzahl der Register, in die ein Fehler injiziert werden kann, 10. Dadurch ergibt sich $w = 100 \cdot 10 = 1000$. Somit lautet der EAFC-Wert für die erste Variante $F_{\text{extrapolated}} = 1000 \cdot \frac{4}{8} = 1000 \cdot \frac{1}{2} = 500$. In der zweiten Variante ist die Laufzeit doppelt so lang, daher ist auch $w = 200 \cdot 10 = 2000$ doppelt so groß. Der EAFC-Wert für die zweite Variante lautet dann $F_{\text{extrapolated}} = 2000 \cdot \frac{2}{8} = 2000 \cdot \frac{1}{4} = 500$. Wie sich zeigt sind hier beide Werte gleich und nicht durch die Erhöhung der Laufzeit beeinflusst [10].

Hinzu kommt eine anschließende Vereinfachung, die besser deutbar ist: $r = \frac{F_{\text{gehärtet}}}{F_{\text{original}}}$ [10]. Diese vergleicht die geschätzte Anzahl der zur Laufzeit auftretenden Fehler in dem originalen und dem gehärteten Programm. Ein Wert kleiner 1 bedeutet, dass nach der Härtung weniger unentdeckte Fehler auftreten, also eine Verbesserung vorliegt. Ist der Wert größer als 1, bedeutet dies eine Verschlechterung der Fehlertoleranz. Sollte der Wert genau 1 sein, hat die Härtung keinen Einfluss auf die Fehlertoleranz des Programms.

2.3.3 FAIL*

Zur Durchführung der Fehlerinjektionskampagnen wird das Fehlerinjektions-Werkzeug *FAIL** [11] genutzt. Dieses Werkzeug ermöglicht die in Kapitel 2.3.1 beschriebene Simulation der Hardware und Durchführung der Experimente. Um FAIL* nutzen zu können, müssen die

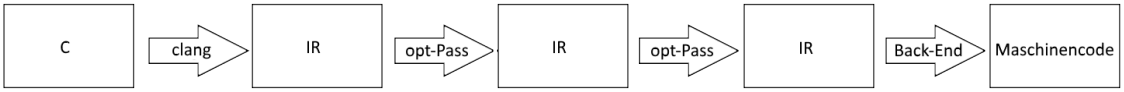

Abbildung 2.3: LLVM-Toolchain [8]

getesten Programme jedoch mit einem Betriebssystem für eingebettete Systeme verknüpft werden. Dazu wird im folgenden Verlauf *eCos* [1] genutzt.

2.4 LLVM

Zur Implementierung des EDDI-Verfahrens wird die Compiler-Architektur *LLVM* (ehemals *Low Level Virtual Machine*) mit dem zugehörigen Front-End für C *(clang)* genutzt. Mit clang wird das C-Programm in die sogenannte *Intermediate Representation (IR/LLVM-IR)* überführt, um es danach mit einem *Optimization pass (Opt-Pass/opt-pass)* modifizieren zu können. Bei dieser Anpassung bleibt der Code jedoch weiterhin in IR vorhanden. Nachdem der in IR-Code durch die Nutzung von Opt-Pässen angepasst wurde, wird er mithilfe eines passendes Back-Ends in Maschinencode der gewünschten Zielplattform (x86, ARM, PowerPC, ..) übersetzt.

Ein Vorteil der IR ist, dass durch die Menge an Front-Ends für unterschiedliche Programmiersprachen die generelle Nutzbarkeit der Opt-Pässe steigt. Zudem nutzt, wie im Codebeispiel 2.2 sichtbar, LLVM unendlich verfügbare *virtuelle Register* [3]. Dies ermöglicht die einfachere Nutzung der Opt-Pässe, da die Register fortlaufend nummeriert werden, sodass beispielsweise Einschübe leicht hinzugefügt werden können. Die Abbildung 2.3 zeigt den Ablauf der Übersetzung und Modifizierung des Codes von der ursprünglichen Programmiersprache bis zum Maschinencode des Zielsystems. Wie zu erkennen, sind mehrere Opt-Pässe auf den in IR vorhandenen Code anwendbar, bevor dieser für das Zielsystem weiter übersetzt wird.

```
0 int main(){
1   int d = 2;
2   d = d + 1;
3   d = d * d;
4   d = d + 1;
5   printf("Done! %i \n",d);
6   return 0;
7 }
```

Listing 2.1: Beispielprogramm in C: bsp.c

```
0 ; Function Attrs: noinline nounwind optnone uwtable
1 define i32 @main() #0 {
```

```
2   %1 = alloca i32, align 4
3   %2 = alloca i32, align 4
4   store i32 0, i32* %1, align 4
5   store i32 2, i32* %2, align 4
6   %3 = load i32, i32* %2, align 4
7   %4 = add nsw i32 %3, 1
8   store i32 %4, i32* %2, align 4
9   %5 = load i32, i32* %2, align 4
10  %6 = load i32, i32* %2, align 4
11  %7 = mul nsw i32 %5, %6
12  store i32 %7, i32* %2, align 4
13  %8 = load i32, i32* %2, align 4
14  %9 = add nsw i32 %8, 1
15  store i32 %9, i32* %2, align 4
16  %10 = load i32, i32* %2, align 4
17  %11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8],
      [11 x i8]* @.str, i32 0, i32 0), i32 %10)
18  ret i32 0
19 }
```

Listing 2.2: Aussehen dieses Programms in IR

2.5 Zusammenfassung

In diesem Kapitel wurden die unterschiedlichen Fehlertypen, die Begriffe Fehlertoleranz, Fehlerdetektion und Fehlerinjektion erklärt und die Funktionsweise des EDDI-Verfahrens erläutert. Außerdem wurden die genutzten Werkzeuge FAIL*, eCos und LLVM vorgestellt, sodass im nachfolgenden Kapitel mithilfe dieses Wissens ein Entwurf für die Verkürzung der Laufzeit des EDDI-Verfahrens entwickelt werden kann.

Kapitel 3

Voraussetzungen und Entwurf

In diesem Kapitel wird zuerst auf eine bestehende Implementierung des EDDI-Verfahrens eingegangen. Anhand dieser Implementierung wird die Problemstellung erläutert und eine möglich Lösung dieses Problems vorgestellt. Zuletzt folgen ein näherer Entwurf dieser Lösung und welche Anpassungen für dessen Umsetzung notwendig sind.

3.1 Bestehende Implementierung

Es besteht bereits eine Implementierung des EDDI-Verfahrens, welche im Rahmen einer Bachelorarbeit von Jan Dirschuweit erstellt und getestet wurde [3]. Diese wurde, wie in Kapitel 2.4 gezeigt, in LLVM in Form eines Opt-Passes umgesetzt. Der Pass sorgt bereits für folgende Bedingungen:

- Die letzte Instruktion eines BBs ist immer ein Sprung oder eine Return-Instruktion.
- Alle BBs wurden in SBBs aufgeteilt.
- Es wurde ein Errorblock eingefügt, welcher im Fehlerfall das Programm terminiert.
- Alle Instruktionen erhalten einen Schatten.
- In jeden SBB wurde vor dem Sprung ein Vergleich des Originals und des Schattens eingefügt.
- Jeder Sprung am Ende eines SBBs wird durch einen bedingten Sprung mit dem Vergleich als Bedingung ersetzt. Dieser Sprung führt zum vorherigen Ziel des unbedingten Sprungs oder zum Errorblock.

Seine Implementierung sorgt für eine alternierende Anwendung des EDDI-Verfahrens auf den Code. Im Folgenden wird ein Verbesserungsansatz vorgestellt.

3.2 Problemanalyse

In der bestehenden Implementierung wird, wie in der zuvor getesteten Variant von Oh, Shirvani und McCluskey [7], jeder einzelne SBB „eddifiziert“. Dies sorgt dafür, dass es durch die verdoppelten Instruktionen und eingefügten Vergleiche zu einem Overhead von bis zu 200% kommt [7]. In Abhängigkeit von dem getesten Programm und dem System, auf dem getestet wird, kann der Overhead geringer ausfallen, da die Schatteninstruktion per Definition bereits unabhängig von den Originalen ablaufen, sodass sich die Berechnungen parallelisieren lassen [7].

Eine Reduzierung dieser zusätzlichen Laufzeitkosten ist somit erwünscht. Eine Möglichkeit dies zu erreichen, ist die selektive Anwendung des EDDI-Verfahrens auf einzelne SBBs. Dabei müssen jedoch folgende Bedingungen weiterhin erfüllt werden:

- Es besteht eine breite Anwendbarkeit auf C-Programme.
- Die Abänderungen des Programms sind nachvollziehbar.
- Das Programm arbeitet korrekt.

Nun stellt sich jedoch die Frage, welche SBBs „eddifiziert“ werden sollen und welche nicht. Es muss also zuvor eine Bewertung der SBBs stattfinden.

3.3 Ansatz

Um die SBBs bewerten zu können, benötigt es jedoch Kriterien, an denen sie bewertet werden können. Diese Kriterien müssen, aufgrund der breiten Anwendbarkeit von EDDI, für eine breite Menge von Programmen herausfindbar sein. Als erstes Kriterium bietet es sich an, zu zählen, wie oft jeder SBB eines Programms ausgeführt wird. Dieses Kriterium kann als negatives Gewicht angesehen werden, da jede Ausführung von einer verlängerten Laufzeit betroffen ist, wenn auf diesen das EDDI-Verfahren angewandt wurde. Da das EDDI-Verfahren dafür sorgen soll, dass transiente Fehler detektiert werden, ist es interessant, wie oft die einzelnen Detektoren anschlagen. Dadurch ergibt sich bereits ein zweites Kriterium.

Weiterhin interessant kann auch die Information sein, welche Detektoren zusammen anschlagen. Sollte eine Menge von Detektoren immer gemeinsam anschlagen, reicht es aus nur einen dieser Detektoren zu selektieren. Hinzu kommt die Latenz der Fehlerdetektion, also die Zeit zwischen dem Eintreten des Fehlers, beziehungsweise der Injektion des Fehlers, und der Detektion der Auswirkungen dieses Fehlers. Letztendlich werden sich viele weitere Kriterien finden lassen, um einen SBB in Hinsicht auf dessen Einfluss auf die Fehlertoleranz und die Laufzeiterhöhung zu bewerten.

3.3.1 Auswahlverfahren der Selektion

Aus diesen Kriterien lassen sich zahlreiche Auswahlverfahren herleiten. Im Folgenden werden die hier verwendeten Auswahlverfahren vorgestellt.

- Die SBBs mit den meisten gefundenen Fehlern werden genommen (*Top N Detektionen*)
- Die SBBs mit den meisten gefundenen Fehlern pro Ausführungen des SBBs werden genommen (*Top N Detektionen/Ausführungen*)

Das Auswahlverfahren *Top N Detektionen* verwendet nur das zweite Kriterium. Anhand der gezählten Detektionen werden die SBBs absteigend sortiert und eine Anzahl der Besten übernommen. Um die Anzahl der Ausführungen miteinzubeziehen, existiert das Verfahren *Top N Detektionen/Ausführungen*. Dieses teilt für jeden SBB die Anzahl der Detektionen durch die Anzahl der Ausführungen, um Letztere mit einem negativen Gewicht zu belegen. Auch dort wird wieder nur eine Anzahl der besten SBBs übernommen. Um auf die breite Anwendbarkeit auf Programme zurückzukommen, ergibt es mehr Sinn, dabei einen prozentualen Anteil der SBBs zu verwenden, anstatt feste Werte zu überprüfen. Denn dann lassen sich die Verfahren auch intern zwischen großen und kleinen Programmen unterscheiden. Indem diese Anzahl variiert wird, lässt sich jedes dieser zwei Verfahren in eine beliebige Anzahl von Verfahren weiter aufteilen. Die Verfahren lassen sich auch insofern variieren, dass entweder nur in selektierten SBBs die Instruktionen einen Schatten erhalten und verglichen werden, oder alle Instruktionen einen Schatten erhalten und nur in selektierten SBBs ein Vergleich eingefügt wird. Eine weitere Abwandlung dieser Verfahren besteht darin, nur SBBs zu selektieren, die mit der jeweiligen Bewertungsmetrik nicht den Wert 0 erhalten, denn dies kann in *Top N Detektionen/Ausführungen* beispielsweise auch auftreten, falls ein Detektor selten anschlägt, der zugehörige SBB aber sehr oft ausgeführt wird.

Hinzu kommen weitere Auswahlverfahren, die aus zeitlichen Gründen nicht implementiert und getestet wurden. Eine Möglichkeit besteht darin, die Latenz der Fehlerdetektion als negatives Kriterium miteinfließen zu lassen. Eine schnellere Detektion von Fehlern, nach ihrem Eintreten, kann dafür sorgen, dass Echtzeit-Schranken von Programmen trotz Verwendung des EDDI-Verfahrens besser eingehalten werden können.

Die Informationen, wie oft ein Detektor angeschlagen hat und mit welchen anderen Detektoren dies geschah, beinhalten eine weitere Information, und zwar welche Fehler durch diesen Detektor abgedeckt werden. Dadurch könnte man wie in Abbildung 3.1 eine Menge von Selektoren bestimmen, die mit einer möglichst geringen Menge von Detektoren und Überschneidungen eine möglichst große Menge von Fehlern überdeckt. Ohne Einfluss weiterer Kriterien sind in diesem Beispiel die grün markierten Pfeile und die orange markierten Pfeile solche Mengen. Dabei könnte in der Menge der orange markierten Pfeile, der erste Detektor mit dem dritten Detektor getauscht werden. Allerdings ist dies eine Frage

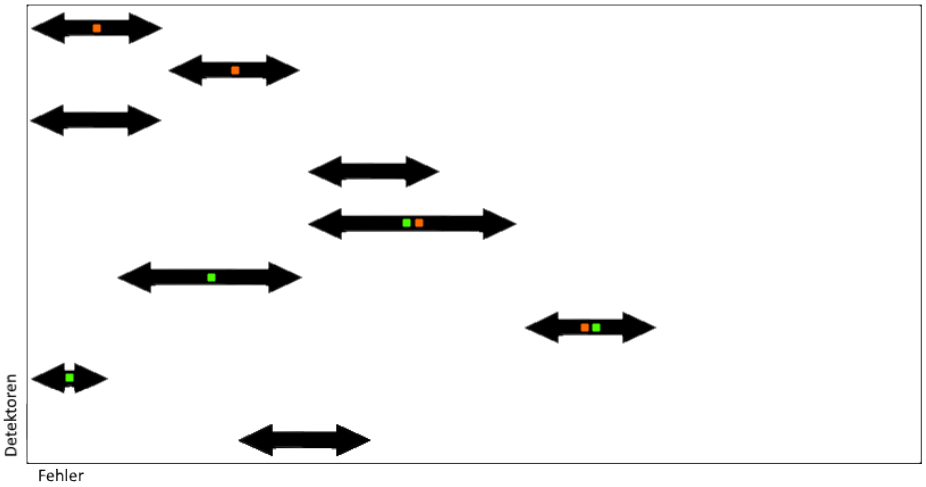

Abbildung 3.1: Selektion mit Fehlerüberdeckung

der Herangehensweise, da hier der erste Detektor zuerst gesehen und in die Menge aufgenommen wurde. Besonders solche gleichwertigen Entscheidungen können über zusätzliche Bewertungskriterien getroffen werden.

Eine weitere Idee besteht darin, mit einem der oben genannten Auswahlverfahren in selektierten Basisblöcken einen Vergleich einzufügen. Dabei erhalten genau die Instruktion einen Schatten, die sich selbst in einem selektierten Basisblock befinden oder Einfluss auf eine Instruktion haben, die sich in einem selektierten Basisblock befindet. Bei selektiver Duplizierung der Instruktionen könnte das Verhalten auftreten, dass die Anzahl der Detektionen aufgrund von Abhängigkeiten zurückgeht. Die Duplizierung aller Instruktionen verhindert dies, kostet jedoch weitere Laufzeit, die eingespart werden könnte.

```
0 r1 = 17                r1:{}
1 r2 = 42                r2:{}
2 r3 = add r1, r2        r3:{r1,r2}
3 r4 = add 1, 1          r4:{}
4 r5 = mul r3, r4        r5:{r3, r4}
```

Listing 3.1: Pseudocode zu einer Folge von Instruktionen mit Abhängigkeiten

Listing 3.1 zeigt drei Intruktionen, die direkte Abhängigkeiten haben. Diese Abhängigkeiten lassen sich jedoch noch durch die transitiven Abhängigkeiten erweitern. *r5* ist von *r3* und *r4* abhängig. Dadurch ergibt sich, dass *r5* auch von *r1* und *r2* abhängig ist, weil *r3* von diesen beiden abhängig ist ($r5 : \{r1, r2, r3, r4\}$). Bildet man nun solche transitiven Ketten, kann für jede Instruktion jeder Vorgänger ermittelt werden, der Einfluss auf sie hat.

3.4 Entwurf

Da für die bisherige Implementierung LLVM genutzt wurde, ist es sinnvoll, weiterhin LLVM zu nutzen. Diese Wahl wurde zuvor getroffen, da LLVM Plattformunabängigkeit bietet und einige Anforderungen bereits durch die Nutzung von LLVM-Klassen erreichbar sind [3].

Zudem wird der bisherige Opt-Pass angepasst, anstatt einen neuen Opt-Pass zu erstellen, da alle vorhandenen Informationen in diesem Pass erneut gebraucht werden und Änderungen des bisherigen Passes mit dem Aufruf des neuen Passes rückgängig gemacht werden müssten. Im Folgenden werden Bedingungen aufgestellt, die für die Anpassung notwendig sind.

3.4.1 Erweiterung um mehrere Modi

Um die zuvor vorgestellten Metriken anwenden zu können, müssen zuvor die zusätzlichen Informationen über das zu härtende Programm gesammelt werden. Einerseits ist die Anzahl der Aufrufe jedes einzelnen SBBs notwendig, andererseits muss die Häufigkeit der detektierten Fehlern bekannt sein. Die Anzahl der Detektionen muss in diesem Fall aufgrund der Art, wie transiente Fehler auftreten, wieder geschätzt werden und wird deshalb mit dem Werkzeug FAIL* ermittelt. Um diese Informationen zu ermitteln und anzuwenden, müssen mehrere Durchläufe stattfinden, die unterschiedliche Aufgaben erledigen. Auch dies wird aufgrund von Redundanz wieder in einem Pass durch Konfigurierbarkeit vereint und nicht einzeln umgesetzt.

Zusätzlich müssen die Basisblöcke eindeutig identifizierbar sein, um diese Daten auswerten zu können. Dies gilt besonders dann, wenn das Programm aus mehreren Dateien besteht, da sonst Fehlinterpretationen auftreten könnten. Wenn für jede Datei die Basisblöcke von 0 aus fortlaufend nummeriert werden, ist nicht genau klar, aus welcher Datei der anschlagende Detektor stammt. Sollten die Basisblöcke dateiübergreifend nummeriert werden, kann, zum Beispiel bei einer Änderung der Reihenfolge der Dateien im Makefile, ein Basisblock „eddifiziert“ werden, der nicht selektiert werden sollte. Währenddessen wurde ein anderer Basisblock nicht „eddifiziert“, obwohl er Fehler detektieren könnte, sodass diese Fehler möglicherweise als SDC enden.

3.4.2 Profiling-Lauf

Der erste Modus ist der *Profiling-Lauf*. Dieser soll in Erfahrung bringen, wie oft jeder SBB in einem Programm ausgeführt wird. Dazu wird das Programm normal „eddifiziert“ und anstatt das Ergebnis des Vergleichs für einen bedingten Sprung zu nutzen, wird eine Zählfunktion aufgerufen. Dadurch wird der in Abbildung 3.2 gezeigte *Errorblock* jedoch nicht mehr genutzt, sodass er weggelassen werden kann. Diese Funktion wird somit, wie in Abbildung 3.3 erkennbar, bei jeder Ausführung dieses Basisblocks aufgerufen, bevor der

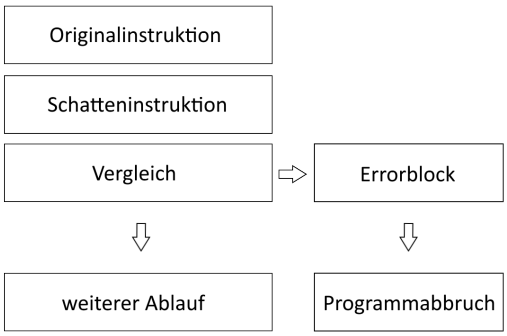

Abbildung 3.2: Ablauf eines Programms im normalen Lauf

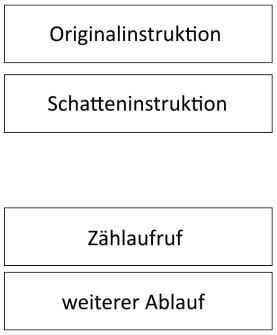

Abbildung 3.3: Ablauf eines Programms im Profiling-Lauf

Sprung zum nächsten Basisblock stattfindet. Dieser Zählmethode wird eine *FileID* und eine *BasicBlockID* übergeben, um den Basisblock, welcher zu diesem Zeitpunkt ausgeführt wird, eindeutig identifizieren zu können.

Zum Schluss wird das so gesammelte Wissen über die Basisblöcke ausgegeben, um für den *selektiven Lauf* genutzt werden zu können.

3.4.3 Injektions-Lauf

Der zweite Lauf ist der *Injektions-Lauf*. Dieser soll herausfinden, welche Detektoren anschlagen und wie oft sie dies tun. Auch in dieser Konfiguration wird das Programm normal „eddifiziert“, jedoch mit einer leichten Strukturveränderung, wie Abbildung 3.4 zeigt. Anstatt das Programm abzubrechen, wird mithilfe der FileID und der BasicBlockID der detektierende SBB identifiziert und im Errorblock durch Aufruf einer Zählfunktion ausgegeben, welche von der Zählfunktion des Profiling-Laufs abweicht. Danach wird der reguläre Programmablauf wieder aufgenommen.

Die Nutzung des Errorblocks ist, im Gegensatz zum Profiling-Lauf, wieder notwendig. Denn ein einfacher Aufruf der Zählfunktion ist eine Instruktion, zu der Fehlerinjektionspunkte existieren, welche im Programmablauf erreichbar sind. Dadurch können in der

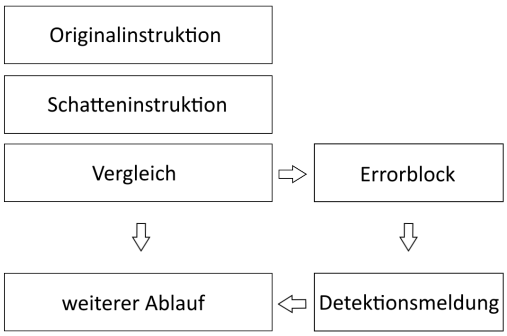

Abbildung 3.4: Ablauf eines Programms im Injektions-Lauf

Fehlerinjektionskampagne Detektoren ausgegeben werden, die keinen Fehler im Programmablauf gefunden haben, sondern durch eine fehlerhaft an die Zählfunktion übergebene Variable hervorgerufen wurden. Deshalb wird dieser Aufruf in einen eigenen SBB verschoben, welcher durch den bedingten Sprung erreichbar ist. Da in einem Fehlerinjektionsexperiment, wie in Kapitel 2.3.1 vorgestellt, nur an genau einer Stelle ein Fehler injiziert wird, ist es nun nicht mehr möglich, dass der Aufruf der Zählfunktion beeinflusst wird. Denn der Sprung in den Errorblock findet nur statt, wenn bereits ein Fehler aufgetreten ist.

Die Detektionen sind nach der Fehlerinjektionskampagne in der Datenbank in den Ergebnissen abgelegt und können für die Verwendung im selektiven Lauf aufbereitet werden.

3.4.4 Selektiver Lauf

Der selektive Lauf besitzt die gleiche Struktur wie die normale Variante der bestehenden Implementierung (Abbildung 3.5). Dies ist bereits sinvoll, weil dieser Lauf die endgültige Variante darstellt, die auf dem Zielsystem als Ersatz für die normal gehärtete Variante angewandt werden soll. Diese Konfiguration entspricht fast vollständig der bestehenden Implementierung. Der Unterschied besteht darin, dass vor Duplikation der Instruktionen, dem Einfügen des Vergleichs und dem Ersetzen des Sprungs überprüft wird, ob der aktuell bearbeitete Basisblock in der Menge der selektierten Basisblöcke ist. Die Menge dieser Basisblöcke wurde zuvor mithilfe der Informationen aus dem Profiling- und Injektions-Lauf und der ausgewählten Bewertungsmetrik bestimmt.

3.5 Zusammenfassung

In diesem Kapitel wurde die Idee zur Verbesserung des Original-Verfahrens näher erläutert und ein Entwurf der Umsetzung vorgestellt. Die Implementierung soll weiterhin als Opt-Pass stattfinden, um die Vorteile des Werkzeugs LLVM, wie in der bestehenden Im-

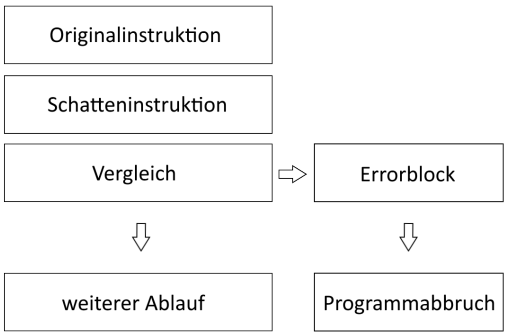

Abbildung 3.5: Ablauf eines Programms im selektiven Lauf

plementierung, beizubehalten. Das nächste Kapitel befasst sich mit der Implementierung des verbesserten Verfahrens.

Kapitel 4

Implementierung

Das vorherige Kapitel zeigte einen Entwurf zur Verbesserung des EDDI-Verfahrens. Im Folgenden wird die Umsetzung dieses Entwurfs vorgestellt und wie sie angewandt werden kann.

4.1 Ursprüngliche Implementierung

Die bestehende Implementierung wurde in C++ in Form eines LLVM-Passes geschrieben, welcher 355 Zeilen Code umfasst [3]. Dieser Pass setzt die in Kapitel 3.1 vorgestellten Bedingungen um, indem er folgende vier Schritte vollführt:

- Aufteilung der BBs in SBBs
- Duplizieren der Instruktionen
- Einfügen der Fehlerfunktion
- Einfügen der Vergleichsinstruktion

4.2 Erweiterungen

Da der LLVM-Pass nun mehrere Modi umfassen muss, steigt die Größe des Passes auf 533 Zeilen Code. Hinzu kommen weitere 59 Zeilen Code, die zum Auslesen und Festhalten der eindeutigen FileIDs dienen. Außerdem übernehmen diese im selektiven Lauf das Einlesen der zu härtenden SBBs, anhand der FileID und BasicBlockID. Die Unterschiede zum originalen Pass befinden sich größtenteils in dem Teil des Codes, welcher sich mit dem Einfügen der Vergleichsinstruktion und der Fehlerfunktion befasst. Die Aufteilung der BBs in SBBs ist komplett übernommen, da dort keine Anpassungen notwendig waren. Das Duplizieren der Instruktionen ist nur insofern abgeändert, dass im selektiven Lauf ausgewählt werden kann, ob alle Instruktionen verdoppelt werden sollen oder nur die Instruktionen der selektierten SBBs dupliziert werden sollen.

Um zwischen den unterschiedlichen Modi wechseln zu können, wurde außerdem ein zusätzlicher Parameter eingeführt, welcher beim Aufruf des Passes übergeben werden kann. In Abhängigkeit von diesem Parameter werden globale Variablen gesetzt, welche zur Interpretation der Konfiguration zur Laufzeit des Passes dienen. Dadurch werden genau die Anpassungen am zu härtenden Programm getroffen, welche in dieser Konfiguration vorgenommen werden sollen.

4.2.1 Profiling-Lauf

Der Profiling-Lauf benötigt keinen Vergleich der Instruktionen und keinen Errorblock, daher wird dieser Teil des Codes übersprungen. Hier wird eine Zählfunktion benötigt, welche die Laufvariablen für die FileID und die BasicBlockID verwendet und sich die Anzahl der Aufrufe merkt. Dazu wird in jedem SBB, vor dem Sprung zum nächsten SBB, ein Aufruf dieser Funktion eingefügt. (Listing 4.1)

```
for (auto &F : M) {
  [...]
  for (auto &B : F) {
    [...]
    for (auto &I : B) {
      if (auto *op = dyn_cast<BranchInst>(&I)) {
        [...]
        IRBuilder<> errorBuilder(op);
        [...]
        if (configPro && configCount && !errorBlockCreated) {
          std::vector<Value *> ArgsV;
          ArgsV.push_back(errorBuilder.getInt32(i)); //BasicBlockID
          ArgsV.push_back(errorBuilder.getInt32(fileID)); //FileID
          errorBuilder.CreateCall(errorFunction, ArgsV);
          errorBlockCreated = true;
        }
      }
    }
  }
}
```

Listing 4.1: Einfügen der Zählfunktion

Die somit gewonnene Information wird in zwei verschachtelten Arrays festgehalten, dessen erste Tiefe die FileID darstellt und dessen zweite Tiefe die BasicBlockID repräsentiert. Bevor das gehärtete Programm terminiert, wird eine zusätzliche Funktion aufgerufen, die die FileID, die BasicBlockID und die Anzahl der Aufrufe in einen zweiten Ausgabekanal schreibt. Dieser Ausgabekanal kann während der Ausführung mit FAIL* abgehört und mitgeschrieben werden. FAIL* wurde dazu um diesen zweiten Ausgabekanal erweitert, da

dieser auch im Injektions-Lauf wichtig wird. Das Resultat der Anwendung dieser Konfiguration auf ein Beispielprogramm ist in Listing A.1 zu sehen.

```
0 .split.split.split:                                     ; preds = %.split.split
1   %7 = add nsw i32 %5, 1
2   %8 = add nsw i32 %6, 1
3   store i32 %7, i32* %3, align 4
4   store i32 %8, i32* %4, align 4
5   call void @eddiCounter(i32 4, i32 1)
6   br label %.split.split.split.split
```

Listing 4.2: Aussehen eines SBBs im Profiling-Lauf

4.2.2 Injektions-Lauf

Im Injektions-Lauf soll wie im normalen Lauf ein Sprung in einen Errorblock erfolgen, danach soll jedoch wieder der Programmablauf aufgenommen werden, anstatt das Programm zu terminieren. Damit die Erkennung der SBBs und der Rücksprung zum Programmablauf eindeutig sind, erhält jeder SBB einen eigenen Errorblock (Listing 4.3). Dieser Errorblock erhält auch die Kennung des SBBs in Form von dessen FileID und BasicBlockID. Zusätzlich erhält der im Errorblock erfolgende Funktionsaufruf diese auch als Parameter.

Da dieses Codestück für jeden Basisblock einer Funktion ausgeführt wird, muss sicher gestellt werden, dass dies nicht erneut auf einen Errorblock angewandt wird. Ansonsten würde eine Endlosschleife entstehen, die für jeden Errorblock einen weiteren Errorblock erstellt. Dies geschieht über das Einfügen der Errorblöcke in ein *std::set* (Zeile 18) und die Überprüfung, ob der aktuelle Basisblock in dieser Menge ist (Zeile 1), bevor die Variable gesetzt wird, die über die Erstellung eines Errorblocks entscheidet (Zeile 3).

```
0 for (auto &B : F) {
1   if (!configCount && errorblockSet.find(&B) == errorblockSet.end() ||
      configCount) {
2     if (configPro) {
3       errorBlockCreated = false;
4     }
5     i += 1;
6     }
7     for (auto &I : B) {
8       if (configPro && !configCount && errorBlockCreated == false) {
9         errorBlock = BasicBlock::Create(M.getContext(), errorName + std::
      to_string(fileID) + "|" + std::to_string(i), &F, &(F.back()));
10        errorBuilder.SetInsertPoint(errorBlock);
11        std::vector<Value *> ArgsV;
12        ArgsV.push_back(errorBuilder.getInt32(i));
13        ArgsV.push_back(errorBuilder.getInt32(fileID));
14        errorBuilder.CreateCall(errorFunction, ArgsV);
15        errorBuilder.CreateBr(op->getSuccessor(0));
```

```
16          errorBlockCreated = true;
17          errorblockSet.emplace(errorBlock);
18        }
19        [...]
20      }
21    }
22  }
```

Listing 4.3: Erstellung der Errorblocks

Da die eindeutige Identifizierung des Basisblocks im jeweiligen Errorblock enthalten ist, muss bei dem Aufruf des Errorblocks keine Anpassung getroffen werden.

In Listing 4.4 ist die Funktion zu sehen, welche im jeweiligen Errorblock mit BasicBlockID und FileID aufgerufen wird. Diese schreibt die übergebenen Parameter in den zuvor erwähnten, zweiten Ausgabekanal, welcher während der Fehlerinjektionskampagne abgehört und in die Ergebnisse der Experimente übertragen wird.

```
0  extern "C"
1  void eddiCounter(int id, int fileID) {
2    char source[2000];
3    sprintf(source, "%ld-%ld/", fileID, id);
4    int len = strlen(source);
5    int i;
6    for (i = 0; i < len; i++) {
7      asm volatile ("outb %%al,%%dx": : "d" (0x2f8), "a" (source[i]));
8    }
9  }
```

Listing 4.4: Funktion zur Detektionsmeldung

Somit können die anschlagenden Detektoren aus der Datenbank entnommen werden. Das Resultat der Anwendung dieser Konfiguration auf ein Beispielprogramm ist in Listing A.2 zu sehen.

```
0  .split.split.split:                                    ; preds = %.split.split,
     %"errorBlock1|3"
1    %7 = add nsw i32 %5, 1
2    %8 = add nsw i32 %6, 1
3    store i32 %7, i32* %3, align 4
4    store i32 %8, i32* %4, align 4
5    %StoreICI = icmp ne i32 %8, %7
6    br i1 %StoreICI, label %"errorBlock1|4", label %.split.split.split.split
7
8  "errorBlock1|4":                                       ; preds = %
       .split.split.split
9    call void @eddiCounter(i32 4, i32 1)
10   br label %.split.split.split.split
```

Listing 4.5: Aussehen eines SBBs und des Errorblocks im Injektions-Lauf

4.2.3 Selektiver Lauf

Der selektive Lauf entspricht, wie im Entwurf vorgestellt, dem normalen Lauf. Allerdings werden nur ausgewählte Basisblöcke „eddifiziert“, was durch eine einfache If-Abfrage umsetzbar ist (Listing 4.6). Weitere Anpassungen sind nicht notwendig.

```
0  if (!(configPro && configCount)) {
1    if (configSelect) { //select-run
2      if (selectedBasicBlocks.find(intString) != std::string::npos) {
3        Value *cI = errorBuilder.CreateICmpNE(storeValue, shadowStore, "
       StoreICI");
4        Value *br = errorBuilder.CreateCondBr(cI, errorBlock, op->getSuccessor
       (0));
5        op->eraseFromParent();
6      }
7    }
8    else { //normal-/error-run
9      Value *cI = errorBuilder.CreateICmpNE(storeValue, shadowStore, "StoreICI
       ");
10     Value *br = errorBuilder.CreateCondBr(cI, errorBlock, op->getSuccessor
       (0));
11     op->eraseFromParent();
12   }
13 }
```

Listing 4.6: Ersetzen der Sprünge durch bedingte Sprünge

Das Resultat der Anwendung dieser Konfiguration auf ein Beispielprogramm ist in Listing A.3 zu sehen. Dieser Lauf benötigt jedoch weitere Vorbereitung.

```
0  .split.split.split:                                ; preds = %.split.split
1    %5 = add nsw i32 %3, 1
2    %6 = add nsw i32 %4, 1
3    store i32 %5, i32* %2, align 4
4    store i32 %6, i32* %2, align 4
5    %StoreICI = icmp ne i32 %6, %5
6    br i1 %StoreICI, label %errorBlock, label %.split.split.split.split
7
8  errorBlock:                                        ; preds = [...]
9    call void @eddiErrorHandler()
10   unreachable
```

Listing 4.7: Aussehen eines SBBs und des Errorblocks im selektiven Lauf

4.3 Selektions-Tool

Die Informationen aus dem Profiling-Lauf und dem Injektions-Lauf müssen noch aufbereitet werden, um im selektiven Lauf eine Liste an SBBs zu haben, die „eddifiziert“ werden

sollen. Um dies umzusetzen, wurde in 180 Zeilen Code ein C++-Programm erstellt, welches diese Informationen einliest. Anhand der eingegebenen Kommandozeilen-Parameter werden die Basisblöcke dann sortiert, ausgewählt und in eine Datei geschrieben, die der selektive Lauf einliest. Die in Kapitel 3.2.1 vorgestellten Auswahlverfahren sind als unterschiedliche Vergleichsoperatoren umgesetzt, die zur Sortierung der Basisblöcke verwendet werden. Außerdem ist die Anzahl der zu selektierten Basisblöcke als fester Wert oder auch als prozentualer Anteil angebbar.

4.4 Anwendung

Da der Pass nun implementiert ist, muss er vor der Nutzung nur noch kompiliert werden. Nachdem dies erfolgt ist kann das in Listing 2.1 gezeigte C-Programm mit dem Aufruf **clang-6.0 -S -emit-llvm bsp.c** in die Intermediate Representation überführt werden. Dadurch erhält man den in Listing 2.2 dargestellten IR-Code. Dieser Code wird dann wiederum durch den Aufruf von **opt-6.0 -S -load /PATH/libEddiPass.so -eddi bsp.ll -o bsp-eddi.ll -var MODE** angepasst. Dabei ist PATH durch den Pfad zu ersetzen, an dem der Pass liegt, und MODE durch den gewünschten Modus zu ersetzen.

- **normal** härtet das Programm wie die bisherige Implementierung komplett.
- **pro** trifft die Anpassungen für den Profiling-Lauf.
- **error** trifft die Anpassungen für den Injektions-Lauf.
- **select** härtet das Programm nur in selektierten SBBs (Vergleich und Duplizierung).
- **selectV2** setzt den Vergleich nur in selektierten SBBs, dupliziert aber alle Instruktionen.

Danach muss die Fehlerfunktion, beziehungsweise die Zählfunktion, verlinkt werden. Dies geschieht durch folgenden Aufruf: **llvm-link-6.0 -S LINKFILE.ll bsp-eddi.ll -o bsp-error.ll**. Folgende Dateien werden je nach Modus verlinkt:

- **eddiErrorHandler.ll** wird im normalen und selektiven Lauf verwendet.
- **eddiCounter.ll** wird im Profiling-Lauf verwendet.
- **eddiErrorHandlerFB.ll** wird im Injektions-Lauf verwendet.

Zum Schluss wird der IR-Code mit dem Aufruf **clang-6.0 bsp-error.ll -o bsp -lm** in Maschinencode übersetzt. Diese Schritte sind auch in einem Aufruf vereinbar, sodass die Einbindung in einem Makefile anstelle des normalen Compilers möglich ist. Der Aufruf lautet **clang-6.0 -c -Xclang -load -Xclang /PATH/libEddiPass.so -mllvm -var -mllvm MODE "$@"**. Allerdings muss in dem Makefile dann auch die zu verlinkende

Datei angegeben werden. Das Selektions-Tool wird wie folgt aufgerufen: **./selectTool SELEKTION ANZAHL %**. Dabei ist ANZAHL durch den gewünschten Anteil zu ersetzen und SELEKTION durch eine der folgenden Optionen zu ersetzen.

- **top** wird für die Top N Detektionen verwendet.
- **ausf** wird für die Top N Detektionen/Ausführungen verwendet.

Der letzte Parameter kann weggelassen werden, um von einem prozentuellen Anteil der Basisblöcke zu einer festen Anzahl zu wechseln.

4.5 Einschränkungen

Die bisherige Implementierung hat bereits das Problem, dass nicht alle Instruktionen verdoppelt werden können. *Structs* stellen ein Problem dar, da für sie kein einfacher Vergleich stattfinden kann. Der Inhalt des structs müsste einzeln überprüft werden, was in der Aufteilung in weitere SBBs endet, da jeder SBB nur einen Vergleich und bedingten Sprung enthalten darf [3]. Dies erzeugt weiteren Overhead, der aufgrund des Einflusses der Laufzeit erneut in Betracht gezogen werden muss. Außerdem ist es möglich, diese Strukturen beliebig oft zu verschachteln, sodass weitere Sonderfälle entstehen [3].

Zusätzlich ist die Übergabe von Zeigern in einem Funktionsaufruf nicht abgedeckt, falls die Funktion den Wert abändert, auf den der Zeiger verweist. Die Kopie wird dabei nicht verändert, sodass bei einer weiteren Verwendung des Originals und des Schattens immer ein Detektor anschlägt, auch wenn kein Fehler injiziert wurde [3]. Um ein solches Programm trotzdem härten zu können, könnte die hier verwendete Selektion von Basisblöcken auch dazu genutzt werden, nur Basisblöck zu „eddifizieren“, die keine Funktionsaufrufe mit Zeigern besitzen. Dies ist allerdings nicht umgesetzt worden.

4.6 Zusammenfassung

Die Implementierung des Verbesserungsentwurfs des EDDI-Verfahrens wurde in diesem Kapitel vorgestellt. Es wurde auf die bestehende Implementierung eingegangen und die drei benötigten Konfigurationen wurden umgesetzt. Außerdem wurde die Anwendung des Passes erläutert, welcher im folgenden Kapitel verwendet wird, um herauszufinden, ob eine Selektion der Detektoren zu einer Verbesserung der Laufzeit und Fehlertoleranz führt.

Kapitel 5

Evaluation

Dieses Kapitel befasst sich mit den Ergebnissen der Fehlerinjektionskampagnen. Die dort gemessenen Programme werden dazu vorher vorgestellt und die notwendigen Anpassungen beschrieben. Danach wird der Ablauf der Fehlerinjektionskampagnen in FAIL* erläutert. Als Letztes werden die gemessenen Ergebnisse, mit der Berechnung des EAFC-Wertes [10] und dem Vergleich dieser Werte, ausgewertet.

5.1 Benchmarks

Um die selektive Anwendung des EDDI-Verfahrens mit der breichtflächigen Anwendung des EDDI-Verfahrens zu vergleichen, wurden Testprogramme aus der MiBench-Suite genutzt [5]. Programme aus dieser Datenbank wurden bereits zur Erprobung der bestehenden Implementierung genutzt [3]. Die Programme wurden angepasst, sodass sie mit eCos verlinkt und für FAIL* vorbereitet werden konnten. Beispielsweise wurden ihre Eingaben gekürzt und im Code hinterlegt, anstatt separat als Datei vorzuliegen. Diese Änderung ist notwendig, da eCos kein Dateisystem unterstützt und somit die Eingaben sonst nicht mit den genutzten Bibliotheksfunktionen eingelesen werden könnten. Hinzu kommen selbst geschriebene Programme, die während dem Entwicklungsprozess genutzt wurden, um die Funktionalität des Opt-Passes zu überprüfen. Dazu gehört Fibonacci, welches bereits zur Erprobung der bestehenden Implementierung eigens implementiert wurde [3].

Die Messung der Testprogramme erfolgt mit FAIL* [11] und dem eCos-Betriebssystem [1]. Die Tabelle 5.1 zeigt nur eine Auswahl an Benchmarks der MiBench-Suite. Diese stellt weitere Benchmarks zur Verfügung, welche jedoch nicht so kompiliert und mit eCos verlinkt werden konnten, dass sie mit FAIL* korrekt arbeiten.

Benchmark	Ursprung	Funktion
basicmath	MiBench	Berechnung verschiedener Winkel und Wurzeln mit math-Bibliothek
blackscholes	MiBench	Bewertung von Finanzoptionen nach dem *Black-Scholes-Modell*
bubblesort	MiBench	Sortierung von Zeichen nach Bubblesort-Vorgehensweise
bitcount	MiBench	
fibonacci	eigene	Berechnung der 100000sten Fibonacci-Zahl
simple	eigene	Einfache Rechenanweisungen, 10000-fach wiederholt

Tabelle 5.1: Die verwendeten Testprogramme

5.2 FAIL*

Die Durchführung der Fehlerinjektionskampagnen erfolgt mit FAIL* [11]. FAIL* ist ein Werkzeug mit vielseitigen Möglichkeiten der Durchführung von Fehlerinjektionskampagnen. Dabei lassen sich die Kampagnen über den gesamten Fehlerraum durchführen, oder auch mit Hilfe von Sampling über einen Teil des Fehlerraums. Außerdem bietet es Tools zur Analyse der gemessenen Daten an. Der Ablauf einer Fehlerinjektionskampagne erfolgt in vier Schritten [11, 3]:

1. Die Erstellung des *golden run*.
2. Das Finden und Festhalten der Fehlerinjektionspunkte.
3. Die Auswahl der Fehlerinjektionspunkte, für die jeweils ein Fehlerinjektionsexperiment durchgeführt werden soll.
4. Die Ausführung der Fehlerinjektionsexperimente mit Festhalten der Ergebnisse und Zusatzinformationen in der Fehlerstatistik.

Zur Durchführung des *golden run* werden zwei Dateien benötigt, die .elf und die .iso des Programms, welches getestet werden soll. Der Name dieser Dateien muss angegeben werden, damit FAIL* zu dem passenden Programm den *golden run* starten kann [3]. Dieser wird daraufhin ausgeführt und dabei ein *trace* aufgezeichnet, welcher den Ablauf der ausgeführten Instruktionen beinhaltet. Daraus kann FAIL* dann Fehlerinjektionspunkte ableiten, die bei einer Fehlerinjektion zu einem Fehlzustand führen können [3]. Als Nächstes erfolgt die Auswahl der Fehlerinjektionspunkte, für die ein Fehlerinjektionsexperiment durchgeführt werden soll. Entweder werden sämtliche Punkte ausgewählt, oder ein Sampling vorgenommen [3]. Da eine Durchführung über den gesamten Fehlerraum sehr lang dauern kann, wird hier ein Sampling vorgenommen. Zum Schluss erfolgt die Durchführung der eigentlichen Kampagne, indem die einzelnen Fehlerinjektionsexperimente durchgeführt werden und die Ergebnisse in einer zuvor angegebenen Datenbank hinterlegt werden [3].

Da FAIL* aktuell die Injektion eines einzelnen Bitfehlers nicht unterstützt, werden stattdessen 8 Bit zeitgleich invertiert [3].

5.3 Ergebnisse

Im Folgenden werden in Tabelle 5.2 die Ergebnisse der Fehlerinjektionskampagnen vorgestellt. Diese Tabelle zeigt die berechneten EAFC-Werte mit der zugehörigen Standardabweichung (EAFC-SE) zu blackscholes. Außerdem ist die gemessene Laufzeit aufgelistet und eine Relation zu der originalen Anwendung des EDDI-Verfahrens angegeben. Diese sagt aus, um wie viel Prozent die gezeigte Variante schneller ist, als die original „eddifizierte“ Variante. Die Messergebnisse werden in die fünf Kategorien unterteilt, welche bereits in Kapitel 2.3.1 vorgestellt wurden: *ok, timeout, trap, detected, SDC*. Aufgrund der Anzahl an Varianten, und der daraus resultierenden Menge von Messergebnissen, wird nur der SDC-Wert gezeigt. Denn die anderen vier Kategorien stellen die erfolgreichen Experimente dar, bei denen der Fehler detektiert wurde oder keinen Einfluss auf das Programm hatte. Des Weiteren sind die Ergebnisse der anderen Testprogramme aufgrund der Menge der Daten im Anhang zu finden.

Somit lässt sich mit diesem Wert eine Aussage über die Verbesserung der Fehlertoleranz treffen. Darauf aufbauend kann dann die Ausgangsfrage, ob das EDDI-Verfahren sich durch selektive Anwendung verbessern lässt, beantwortet werden.

5.3.1 Auswertung

Zunächst gilt es zu sagen, dass die Ergebnisse der meisten Benchmarks nur bedingt Aussagekraft besitzen. Dies liegt daran, dass die EAFC-Werte für diese Benchmarks durch Anwendung des EDDI-Verfahrens bereits schlechter werden als die EAFC-Werte der ursprünglichen Programme. Zurückzuführen ist dies auf die deutlich erhöhte Laufzeit durch Anwendung des EDDI-Verfahrens. Dies ist jedoch nicht der Fall für blackscholes, weshalb dieses Testprogramm im weiteren Verlauf näher betrachtet wird, als die Anderen.

In den Tabellen ist zu sehen, dass die EAFC-Werte der Varianten sich deutlich unterscheiden können und diese auch abhängig vom Testprogramm unterschiedlich gut abschneiden. Beispielsweise sind die Varianten, welche Detektoren generell weglassen, wenn sie durch das Auswahlverfahren den Wert 0 erhalten, für bitcount, blackscholes und bubblesort besser als die original „eddifizierte“ Variante, während sie für basicmath, fibonacci und simple schlechter sind.

In Abbildung 5.1 sieht man die addierten EAFC-Werte der Varianten über alle Benchmarks, unterteilt nach dem Auswahlverfahren der Selektoren. Dort ist bereits zu erkennen, dass das Auswahlverfahren *Top N Detektionen/Ausführungen* (aus) generell bessere EAFC-Werte liefert, als das Auswahlverfahren *Top N Detektionen* (top). Für die entsprechenden Abwandlungen, die alle Instruktionen duplizieren und nur die Vergleiche weglassen, (dAus und dTop) gilt dieses Verhalten auch. Lediglich basicmath verhält sich im Gegensatz zu den anderen Benchmarks dort anders, da die Fehlerinjektionskampagnen zu *dTop20%* und *dTop80%* so gut abgeschnitten haben, dass aufgrund der Programmgröße im Balkendia-

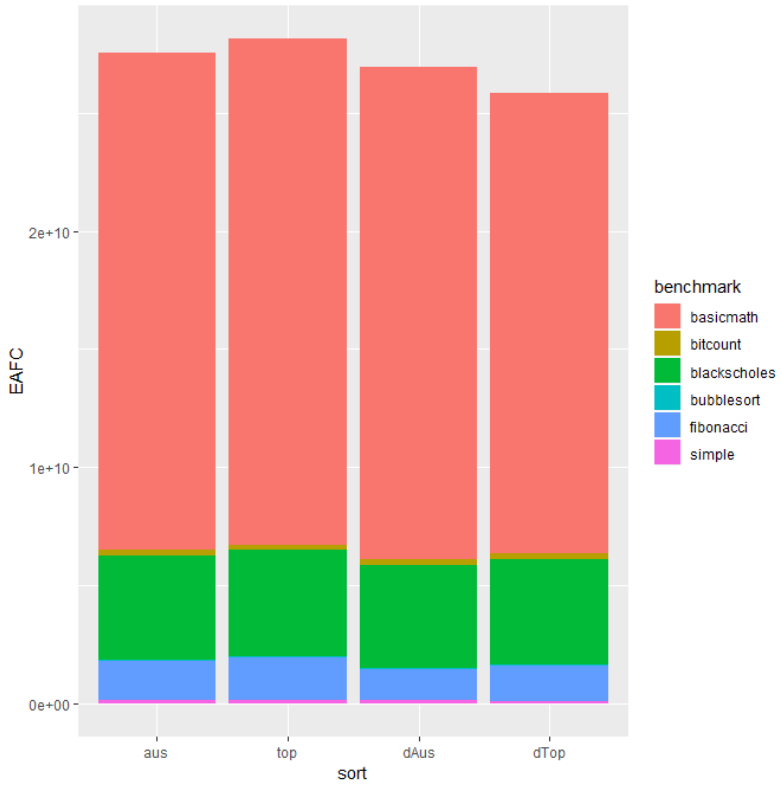

Abbildung 5.1: EAFC-Werte aller Benchmarks zum Vergleich der Strategien

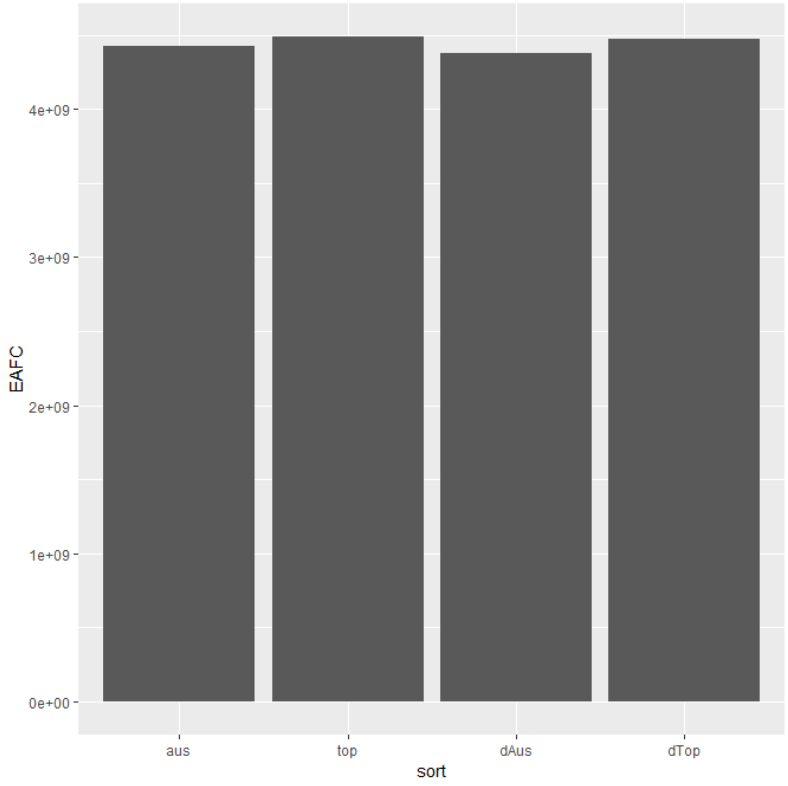

Abbildung 5.2: EAFC-Werte für blackscholes zum Vergleich der Strategien

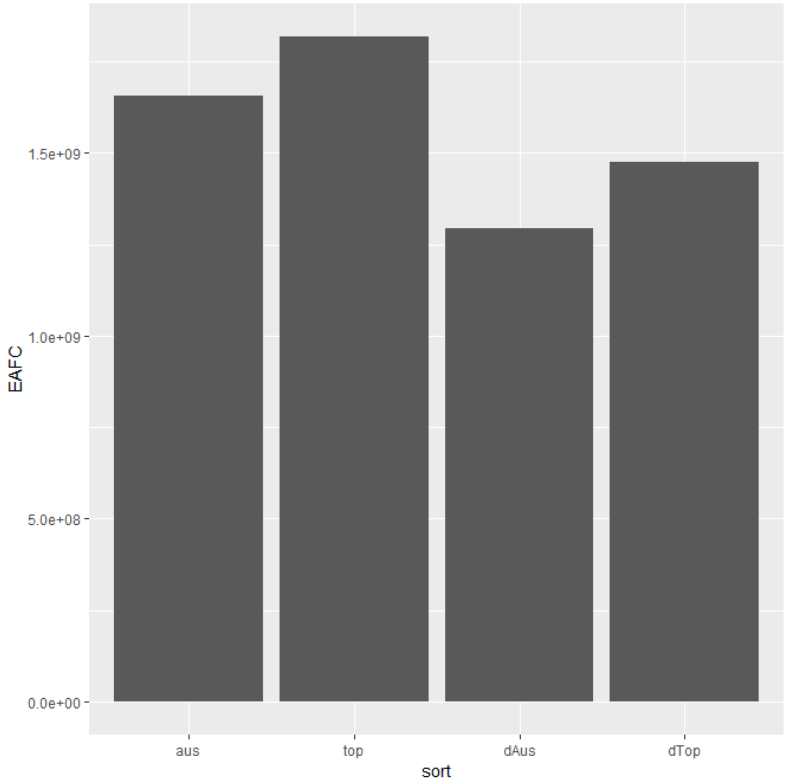

Abbildung 5.3: EAFC-Werte für fibonacci zum Vergleich der Strategien

gramm 5.1 auf den ersten Blick nicht zu erkennen ist, dass das Auswahlverfahren *Top N Detektionen/Ausführungen* ein generell besserer Ansatz ist als das Auswahlverfahren *Top N Detektionen*. Schaut man sich die Werte der Benchmarks einzeln an, schneidet *Top N Detektionen/Ausführungen*, wie in Abbildung 5.2 für blackscholes gezeigt, klar erkennbar ein wenig besser ab als *Top N Detektionen*. Schaut man sich nun das gleiche Diagramm zu fibonacci an (Abbildung 5.3), erkennt man dieses Verhalten erneut. Allerdings ist dort auch erkennbar, dass das Duplizieren aller Instruktionen (dAus und dTop) dort bessere Ergebnisse erzielt als das selektive Duplizieren der Instruktionen (aus und top). Zurückzuführen ist dies auf die Programmstruktur, welche immer wieder die gleichen Variablen nutzt, um die nächste Fibonacci-Zahl zu berechnen, und auf die Abhängigkeiten der Instruktionen innerhalb dieser Funktion, die die nächste Fibonacci-Zahl berechnet. Schaut man sich dazu Tabelle A.1 an, erkennt man, dass das ursprüngliche Programm ungefähr 85% schneller ist, als das „eddifizierte“ Programm. Während beispielsweise *aus40%* 28% schneller ist und *dAus40%* nur 13% schneller ist, liefert *dAus40%* dennoch wider Erwarten einen besseren EAFC-Wert. Aufgrund dieses Verhaltens wird vermutlich besonders hier das bereits vorgestellte Auswahlverfahren, welches Ketten von voneinander abhängigen Instruktionen bildet, bessere Ergebnisse erzielen können.

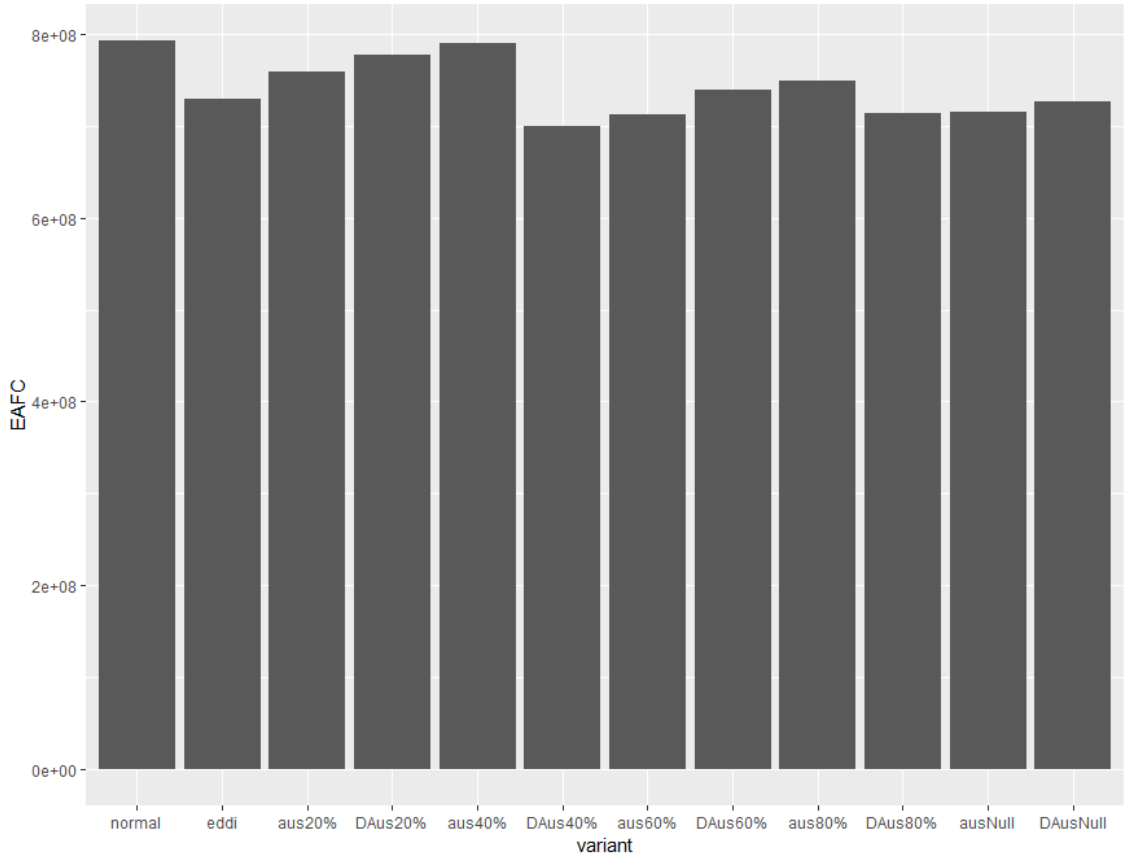

Abbildung 5.4: EAFC-Werte für blackscholes mit Top N Detektionen/Ausführungen

Für einige Testprogramme nimmt die Laufzeit durch Anwendung des EDDI-Verfahrens so stark zu, dass wie bereits erwähnt der EAFC-Wert schlechter wird. Beispielsweise ist auch das ursprüngliche Programm zu bubblesort ungefähr 36% schneller als das „eddifizierte“ Programm. Dadurch ergibt sich, dass für diese Testprogramme die Varianten mit 20%, 40% oder auch 60% generell besser abschneiden, da sie näher an der Laufzeit des ursprünglichen Programms sind.

Daher wird als Nächstes das Testprogramm blackscholes betrachtet. In Abbildung 5.4 ist zu sehen, dass *aus100%* und *dAus100%* weggelassen wurden, da diese denselben Code erzeugen, der bei der originalen Anwendung des EDDI-Verfahrens entsteht. Ihre Laufzeiten und ihre EAFC-Werte sollten also ungefähr den Werten der Variante *eddi* entsprechen. Alle dort abgebildeten Varianten besitzen weiterhin einen besseren EAFC-Wert als das ursprüngliche Programm. Somit kann also auf Kosten der Fehlertoleranz entschieden werden, beispielsweise *aus20%* zu verwenden, da diese Variante ungefähr 3% schneller als das vollkommen „eddifizierte“ Programm ist, während das ursprüngliche Programm nur ungefähr 6% schneller ist. Es kann also in diesem Fall mit geringen Einbußen der Fehlertoleranz die Verschlechterung der Laufzeit, welche durch die Anwendung des EDDI-Verfahrens aufgetreten ist, halbiert werden.

Zusätzlich ist zu erkennen, dass die Varianten *dAus40%, aus60%, dAus60%, dAus80%, ausNull* und *dAusNull* für blackscholes bessere EAFC-Werte liefern, als die Variante *eddi*, welche die originale Anwendung des EDDI-Verfahrens repräsentiert. Dies zeigt also, dass nicht nur eine Reduzierung der Laufzeitkosten durch selektive Anwendung des EDDI-Verfahrens möglich ist, sondern sich dabei die Fehlertoleranz des Programms sogar noch verbessern lässt. Verbindet man diese Erkenntnisse nun auch noch mit Tabelle 5.2, sieht man, dass *dAus40%* und *aus60%* mit $1,22\%$ und $1,18\%$ Laufzeitverbesserung gegenüber *eddi* auch ungefähr die gleiche Laufzeit besitzen, während *dAus40%* mit weniger ausgewählten Detektoren knapp besser abschneidet. Dies weist erneut darauf hin, dass, wie für das Testprogramm fibonacci, vermutlich eine weitere Verbesserung durch die Wahl eines Auswahlverfahrens möglich ist, welches Abhängigkeiten zwischen Instruktionen miteinbezieht. Die größte Laufzeitverbesserung dieser Varianten würde die Variante *ausNull* mit ungefähr 2% bieten, während sie einen nur gering schlechteren EAFC-Wert als die Variante *dAus40%* besitzt. Weiterhin ist dieser allerdings immer noch besser als der EAFC-Wert der Variante *eddi*.

5.4 Zusammenfassung

Nach Abschluss der Bewertung der vorgestellten Auswahlverfahren der Selektoren zeigt sich, dass eine Verbesserung der Laufzeit und der Fehlertoleranz in Abhängigkeit von dem gewählten Auswahlverfahren möglich ist. Außerdem wurde deutlich, dass das Auswahlverfahren *Top N Detektionen/Ausführungen* im Durchschnitt bessere Ergebnisse erzielt als das Auswahlverfahren *Top N Detektionen*. Dies kann allerdings auch anders ausfallen, wie das Testprogramm basicmath gezeigt hat.

Je nach Programm, welches gegenüber transienten Fehlern gehärtet werden soll, kann das Resultat einer selektiven Anwendung des EDDI-Verfahrens auch schlechter werden, wie das Testprogramm bubblesort für mehrere Varianten gezeigt hat. Außerdem kann bei Programmen, die selber bereits fehlertolerant sind und durch Anwendung des EDDI-Verfahrens schlechtere EAFC-Werte erzielen, oder auch allgemein bei allen Programmen, mit der selektiven Anwendung ein Kompromiss zwischen gar keiner Anwendung des Verfahrens und der kompletten Härtung mit EDDI in Bezug auf Fehlertoleranz und Laufzeiterhöhung geschlossen werden.

benchmark	variant	EAFC	EAFC-SE	Laufzeit	Verringerung
blackscholes	eddi	730008263.2470	33649120.29983504	1348283	0,00%
blackscholes	normal	793225933.0710	33942306.65563412	1262257	6,38%
blackscholes	aus20%	759660836.0400	33709908.320513636	1309120	2,90%
blackscholes	aus40%	790794231.4920	34348471.69780503	1322381	1,92%
blackscholes	aus60%	712956843.9440	33131975.724638876	1332389	1,18%
blackscholes	aus80%	749912110.5450	34070303.623211734	1347602	0,05%
blackscholes	aus100%	694149415.1455	33073018.978164278	1347602	0,05%
blackscholes	ausNull	715899434.9115	32998785.658636153	1320772	2,04%
blackscholes	top20%	751521901.4000	33609081.16220589	1314101	2,54%
blackscholes	top40%	737486606.3685	33452515.680135593	1324796	1,74%
blackscholes	top60%	773598115.9080	34169754.27593668	1333348	1,11%
blackscholes	top80%	748741939.1400	34017139.93309533	1343346	0,37%
blackscholes	top100%	751834962.1105	34103359.49383813	1347602	0,05%
blackscholes	topNull	728036231.1705	33235307.103878446	1324534	1,76%
blackscholes	DAus20%	778084648.4815	34315101.26203685	1323924	1,81%
blackscholes	DAus40%	699847193.7150	33065394.687106486	1331897	1,22%
blackscholes	DAus60%	739068821.4555	33631237.89880571	1338063	0,76%
blackscholes	DAus80%	713518134.0635	33321870.93580421	1344292	0,30%
blackscholes	DAus100%	717451445.3600	33505559.73797367	1347574	0,05%
blackscholes	DAusNull	727453111.8760	33477081.17520345	1331503	1,24%
blackscholes	DTop20%	734861969.4030	33600743.50192102	1326054	1,65%
blackscholes	DTop40%	756093742.8660	34026339.83798344	1331956	1,21%
blackscholes	DTop60%	714369863.4120	33197640.465934068	1338063	0,76%
blackscholes	DTop80%	724995637.5605	33526814.9942826	1344292	0,30%
blackscholes	DTop100%	771308390.3200	34440155.47861765	1347574	0,05%
blackscholes	DTopNull	773156972.3080	34255776.82939699	1331503	1,24%

Tabelle 5.2: Ergebnisse der Fehlerinjektionskampagnen zu blackscholes mit jeweils 2000 Samples. Zu sehen ist der EAFC-Wert und die zugehörige Standardabweichung (EAFC-SE). Hinzu kommt die Laufzeit und um wie viel % die jeweilige Variante schneller ist als die Variante mit dem original angewandtem EDDI-Verfahren

Kapitel 6

Zusammenfassung

Die selektive Anwendung des EDDI-Verfahrens wurde in dieser Abschlussarbeit beschrieben, erläutert und zum Schluss bewertet. Dazu wurden zunächst grundlegende Begriffe wie die unterschiedlichen Fehlertypen, Fehlertoleranz und das EDDI-Verfahren selbst erklärt [7]. Hinzu kamen weitere Grundlagen, wie der Begriff der Fehlerinjektion und die verwendeten Werkzeuge, die eine Basis für die Bewertung geschaffen haben. Daraufhin wurden die FCF-Metrik und ihre Schwächen beschrieben, welche zu der Vorstellung der EAFC-Metrik führten, welche die Laufzeit der Programme miteinbezieht [3]. Zur Bewertung wurde die EAFC-Metrik verwendet und eine Relation zwischen den gemessenen Laufzeiten gebildet.

Dabei hat sich gezeigt, dass die selektive Anwendung des EDDI-Verfahrens zwar immer für eine Verringerung der Programmlaufzeit gesorgt hat, diese allerdings nicht zwingend zu einer Verbesserung der Fehlertoleranz geführt hat. Die Programme konnten, besonders wenn sie durch das EDDI-Verfahren schlechtere Fehlertoleranz aufwiesen, fehlertoleranter sein, jedoch nur in einigen Varianten, während andere Varianten widerum schlechtere Fehlertoleranz aufwiesen, als das originale EDDI-Verfahren. Ein Beispiel für den letzteren Fall trat bei dem Testprogramm bubblesort auf.

Es ist nur schwer möglich eine allgemeine Aussage über das Potenzial der selektiven Anwendung des EDDI-Verfahrens zu treffen, wenn nur sechs Benchmarks getestet wurden und zwei (beziehungsweise vier) Auswahlverfahren angewandt wurden. Allerdings deuten diese Ergebnisse bereits an, dass eine generelle Verbesserung gegenüber dem originalen EDDI-Verfahren von Oh, Shirvani und McCluskey [7] möglich ist. Dies erfolgt jedoch unter der Bedingung, dass für jedes Programm die Fehlerinjektionskampagnen mit unterschiedlichen Varianten durchgeführt und bewertet werden und darauf basierend eine Wahl für das jeweilige Programm getroffen wird.

6.1 Ausblick

Aufgrund der zunehmenden Anzahl von eingebetteten Systemen in unserem Alltag und dem bestehenden Kostendruck in der Industrie wird die Softwarehärtung auch in Zukunft noch eine gravierende Rolle haben. Auch wenn das EDDI-Verfahren als Detektionsverfahren konzipiert wurde, lässt es sich um einen Teil erweitern, der die Korrektur des aufgetretenen Fehlers übernimmt. Außerdem könnte eine blockweise Duplizierung der Instruktionen, die von Oh, Shirvani und McCluskey ebenfalls vorgestellt [7], hier aber nicht verwendet wurde, für eine weitere Verbesserung sorgen, da die angesprochenen Abhängigkeiten zwischen Instruktionen so zumindest zum Teil abgedeckt wären.

Hinzu kommen weiterhin vorhandene Lücken im Code, der in dieser Arbeit vorgestellt und erweitert wurde. Beispielsweise werden Structs nicht dupliziert und überprüft, Berechnungen von Funktionen aus Standardbibliotheken nicht abgedeckt und Fehler nicht erkannt, die beim Aufruf einer Funktion mit Übergabe eines Zeigers als Parameter entstehen, weil der Zeiger nicht so dupliziert wird, dass auch der gezeigte Wert einen Schatten erhalten würde und durch den Schatten-Zeiger beeinflusst werden würde.

Des Weiteren wurden in Kapitel 3 weitere Auswahlverfahren der Selektoren vorgestellt, die aufgrund von mangelnder Zeit nicht implementiert und bewertet werden konnten. Diese und weitere Auswahlverfahren, die sich finden lassen, könnten möglicherweise zu einer noch besseren Fehlertoleranz führen und vielleicht sogar für eine große Menge von Programmen bessere Ergebnisse erzielen, als die originale Anwendung des EDDI-Verfahrens, anstatt für jedes Programm unterschiedliche Ergebnisse vorzuweisen.

Anhang A

Weitere Informationen

```
; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  call void @eddiCounter(i32 1, i32 1)
  br label %.split

.split:                                           ; preds = %0
  store i32 2, i32* %3, align 4
  store i32 2, i32* %4, align 4
  call void @eddiCounter(i32 2, i32 1)
  br label %.split.split

.split.split:                                     ; preds = %.split
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  call void @eddiCounter(i32 3, i32 1)
  br label %.split.split.split

.split.split.split:                               ; preds = %.split.split
  %7 = add nsw i32 %5, 1
  %8 = add nsw i32 %6, 1
  store i32 %7, i32* %3, align 4
  store i32 %8, i32* %4, align 4
  call void @eddiCounter(i32 4, i32 1)
  br label %.split.split.split.split

.split.split.split.split:                         ; preds = %
    .split.split.split
```

```
32   %9 = load i32, i32* %3, align 4
33   %10 = load i32, i32* %4, align 4
34   call void @eddiCounter(i32 5, i32 1)
35   br label %.split.split.split.split.split
36
37 .split.split.split.split.split:                  ; preds = %
       .split.split.split.split
38   %11 = load i32, i32* %3, align 4
39   %12 = load i32, i32* %4, align 4
40   call void @eddiCounter(i32 6, i32 1)
41   br label %.split.split.split.split.split.split
42
43 .split.split.split.split.split.split:            ; preds = %
       .split.split.split.split.split
44   %13 = mul nsw i32 %9, %11
45   %14 = mul nsw i32 %10, %12
46   store i32 %13, i32* %3, align 4
47   store i32 %14, i32* %4, align 4
48   call void @eddiCounter(i32 7, i32 1)
49   br label %.split.split.split.split.split.split.split
50
51 .split.split.split.split.split.split.split:       ; preds = %
       .split.split.split.split.split.split
52   %15 = load i32, i32* %3, align 4
53   %16 = load i32, i32* %4, align 4
54   call void @eddiCounter(i32 8, i32 1)
55   br label %.split.split.split.split.split.split.split.split
56
57 .split.split.split.split.split.split.split.split: ; preds = %
       .split.split.split.split.split.split.split
58   %17 = add nsw i32 %15, 1
59   %18 = add nsw i32 %16, 1
60   store i32 %17, i32* %3, align 4
61   store i32 %18, i32* %4, align 4
62   call void @eddiCounter(i32 9, i32 1)
63   br label %.split.split.split.split.split.split.split.split.split
64
65 .split.split.split.split.split.split.split.split.split: ; preds = %
       .split.split.split.split.split.split.split.split
66   %19 = load i32, i32* %3, align 4
67   %20 = load i32, i32* %4, align 4
68   call void @eddiCounter(i32 10, i32 1)
69   br label %.split.split.split.split.split.split.split.split.split.split
70
71 .split.split.split.split.split.split.split.split.split.split: ; preds = %
       .split.split.split.split.split.split.split.split.split
72   %21 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8],
       [11 x i8]* @.str, i32 0, i32 0), i32 %20)
```

```
73    call void @eddiCounter(i32 11, i32 1)
74    br label %
        .split.split.split.split.split.split.split.split.split.split.split
75
76  .split.split.split.split.split.split.split.split.split.split.split: ; preds
        = %.split.split.split.split.split.split.split.split.split.split
77    ret i32 0
78  }
```

Listing A.1: Aussehen eines Programms im Profiling-Lauf

```
0   ; Function Attrs: noinline nounwind optnone uwtable
1   define i32 @main() #0 {
2     %1 = alloca i32, align 4
3     %2 = alloca i32, align 4
4     %3 = alloca i32, align 4
5     %4 = alloca i32, align 4
6     store i32 0, i32* %1, align 4
7     store i32 0, i32* %2, align 4
8     br i1 false, label %"errorBlock1|1", label %.split
9
10  .split:                                           ; preds = %0, %"
        errorBlock1|1"
11    store i32 2, i32* %3, align 4
12    store i32 2, i32* %4, align 4
13    br i1 false, label %"errorBlock1|2", label %.split.split
14
15  .split.split:                                     ; preds = %.split, %"
        errorBlock1|2"
16    %5 = load i32, i32* %3, align 4
17    %6 = load i32, i32* %4, align 4
18    %IntCI = icmp ne i32 %6, %5
19    br i1 %IntCI, label %"errorBlock1|3", label %.split.split.split
20
21  .split.split.split:                               ; preds = %.split.split,
        %"errorBlock1|3"
22    %7 = add nsw i32 %5, 1
23    %8 = add nsw i32 %6, 1
24    store i32 %7, i32* %3, align 4
25    store i32 %8, i32* %4, align 4
26    %StoreICI = icmp ne i32 %8, %7
27    br i1 %StoreICI, label %"errorBlock1|4", label %.split.split.split.split
28
29  .split.split.split.split:                         ; preds = %
        .split.split.split, %"errorBlock1|4"
30    %9 = load i32, i32* %3, align 4
31    %10 = load i32, i32* %4, align 4
32    %IntCI1 = icmp ne i32 %10, %9
```

```
33   br i1 %IntCI1, label %"errorBlock1|5", label %
       .split.split.split.split.split
34
35 .split.split.split.split.split:                    ; preds = %
       .split.split.split.split , %"errorBlock1|5"
36   %11 = load i32, i32* %3, align 4
37   %12 = load i32, i32* %4, align 4
38   %IntCI2 = icmp ne i32 %12, %11
39   br i1 %IntCI2, label %"errorBlock1|6", label %
       .split.split.split.split.split.split
40
41 .split.split.split.split.split.split:               ; preds = %
       .split.split.split.split.split , %"errorBlock1|6"
42   %13 = mul nsw i32 %9, %11
43   %14 = mul nsw i32 %10, %12
44   store i32 %13, i32* %3, align 4
45   store i32 %14, i32* %4, align 4
46   %StoreICI3 = icmp ne i32 %14, %13
47   br i1 %StoreICI3, label %"errorBlock1|7", label %
       .split.split.split.split.split.split.split
48
49 .split.split.split.split.split.split.split:         ; preds = %
       .split.split.split.split.split.split , %"errorBlock1|7"
50   %15 = load i32, i32* %3, align 4
51   %16 = load i32, i32* %4, align 4
52   %IntCI4 = icmp ne i32 %16, %15
53   br i1 %IntCI4, label %"errorBlock1|8", label %
       .split.split.split.split.split.split.split.split
54
55 .split.split.split.split.split.split.split.split: ; preds = %
       .split.split.split.split.split.split.split , %"errorBlock1|8"
56   %17 = add nsw i32 %15, 1
57   %18 = add nsw i32 %16, 1
58   store i32 %17, i32* %3, align 4
59   store i32 %18, i32* %4, align 4
60   %StoreICI5 = icmp ne i32 %18, %17
61   br i1 %StoreICI5, label %"errorBlock1|9", label %
       .split.split.split.split.split.split.split.split.split
62
63 .split.split.split.split.split.split.split.split.split: ; preds = %
       .split.split.split.split.split.split.split.split , %"errorBlock1|9"
64   %19 = load i32, i32* %3, align 4
65   %20 = load i32, i32* %4, align 4
66   %IntCI6 = icmp ne i32 %20, %19
67   br i1 %IntCI6, label %"errorBlock1|10", label %
       .split.split.split.split.split.split.split.split.split.split
68
```

```
.split.split.split.split.split.split.split.split.split.split: ; preds = %
    .split.split.split.split.split.split.split.split.split.split, %"errorBlock1|10"
  %21 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8],
    [11 x i8]* @.str, i32 0, i32 0), i32 %20)
  br label %
    .split.split.split.split.split.split.split.split.split.split.split

"errorBlock1|1":                                  ; preds = %0
  call void @eddiCounter(i32 1, i32 1)
  br label %.split

"errorBlock1|2":                                  ; preds = %.split
  call void @eddiCounter(i32 2, i32 1)
  br label %.split.split

"errorBlock1|3":                                  ; preds = %.split.split
  call void @eddiCounter(i32 3, i32 1)
  br label %.split.split.split

"errorBlock1|4":                                  ; preds = %
    .split.split.split
  call void @eddiCounter(i32 4, i32 1)
  br label %.split.split.split.split

"errorBlock1|5":                                  ; preds = %
    .split.split.split.split
  call void @eddiCounter(i32 5, i32 1)
  br label %.split.split.split.split.split

"errorBlock1|6":                                  ; preds = %
    .split.split.split.split.split
  call void @eddiCounter(i32 6, i32 1)
  br label %.split.split.split.split.split.split

"errorBlock1|7":                                  ; preds = %
    .split.split.split.split.split.split
  call void @eddiCounter(i32 7, i32 1)
  br label %.split.split.split.split.split.split.split

"errorBlock1|8":                                  ; preds = %
    .split.split.split.split.split.split.split
  call void @eddiCounter(i32 8, i32 1)
  br label %.split.split.split.split.split.split.split.split

"errorBlock1|9":                                  ; preds = %
    .split.split.split.split.split.split.split.split
  call void @eddiCounter(i32 9, i32 1)
  br label %.split.split.split.split.split.split.split.split.split
```

```
108
109 "errorBlock1|10":                                ; preds = %
      .split.split.split.split.split.split.split.split.split
110   call void @eddiCounter(i32 10, i32 1)
111   br label %.split.split.split.split.split.split.split.split.split.split
112
113 "errorBlock1|11":                                ; No predecessors!
114   call void @eddiCounter(i32 11, i32 1)
115   br label %
      .split.split.split.split.split.split.split.split.split.split.split
116
117 .split.split.split.split.split.split.split.split.split.split.split: ; preds
      = %"errorBlock1|11", %
      .split.split.split.split.split.split.split.split.split.split
118   ret i32 0
119 }
```

Listing A.2: Aussehen eines Programms im Injektions-Lauf

```
0  ; Function Attrs: noinline nounwind optnone uwtable
1  define i32 @main() #0 {
2    %1 = alloca i32, align 4
3    %2 = alloca i32, align 4
4    store i32 0, i32* %1, align 4
5    br label %.split
6
7  .split:                                           ; preds = %0
8    store i32 2, i32* %2, align 4
9    store i32 2, i32* %2, align 4
10   br i1 false, label %errorBlock, label %.split.split
11
12 .split.split:                                     ; preds = %.split
13   %3 = load i32, i32* %2, align 4
14   %4 = load i32, i32* %2, align 4
15   %IntCI = icmp ne i32 %4, %3
16   br i1 %IntCI, label %errorBlock, label %.split.split.split
17
18 .split.split.split:                               ; preds = %.split.split
19   %5 = add nsw i32 %3, 1
20   %6 = add nsw i32 %4, 1
21   store i32 %5, i32* %2, align 4
22   store i32 %6, i32* %2, align 4
23   %StoreICI = icmp ne i32 %6, %5
24   br i1 %StoreICI, label %errorBlock, label %.split.split.split.split
25
26 .split.split.split.split:                         ; preds = %
      .split.split.split
27   %7 = load i32, i32* %2, align 4
```

```
28    br label %.split.split.split.split.split
29
30  .split.split.split.split.split:                   ; preds = %
      .split.split.split.split
31    %8 = load i32, i32* %2, align 4
32    br label %.split.split.split.split.split.split
33
34  .split.split.split.split.split.split:             ; preds = %
      .split.split.split.split.split
35    %9 = mul nsw i32 %7, %8
36    %10 = mul nsw i32 %7, %8
37    store i32 %9, i32* %2, align 4
38    store i32 %10, i32* %2, align 4
39    %StoreICI1 = icmp ne i32 %10, %9
40    br i1 %StoreICI1, label %errorBlock, label %
      .split.split.split.split.split.split.split
41
42  .split.split.split.split.split.split.split:       ; preds = %
      .split.split.split.split.split.split
43    %11 = load i32, i32* %2, align 4
44    %12 = load i32, i32* %2, align 4
45    %IntCI2 = icmp ne i32 %12, %11
46    br i1 %IntCI2, label %errorBlock, label %
      .split.split.split.split.split.split.split.split
47
48  .split.split.split.split.split.split.split.split: ; preds = %
      .split.split.split.split.split.split.split
49    %13 = add nsw i32 %12, 1
50    store i32 %13, i32* %2, align 4
51    br label %.split.split.split.split.split.split.split.split.split
52
53  .split.split.split.split.split.split.split.split.split: ; preds = %
      .split.split.split.split.split.split.split.split
54    %14 = load i32, i32* %2, align 4
55    br label %.split.split.split.split.split.split.split.split.split.split
56
57  .split.split.split.split.split.split.split.split.split.split: ; preds = %
      .split.split.split.split.split.split.split.split.split
58    %15 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8],
      [11 x i8]* @.str, i32 0, i32 0), i32 %14)
59    br label %
      .split.split.split.split.split.split.split.split.split.split.split
60
61  errorBlock:                                       ; preds = %
      .split.split.split.split.split.split.split, %
      .split.split.split.split.split.split, %.split.split.split, %.split.split,
      %.split
62    call void @eddiErrorHandler()
```

benchmark	variant	EAFC	EAFC-SE	Laufzeit	Verringerung
fibonacci	eddi	230647090.9050	21910028.431103464	9215175	0,00%
fibonacci	normal	57509709.6600	5158915.403905633	1407714	84,72%
fibonacci	aus20%	181361972.6320	14451391.615084838	4911203	46,71%
fibonacci	aus40%	221454594.9700	18049374.34590262	6612682	28,24%
fibonacci	aus60%	279560786.0850	22194240.992059775	8114161	11,95%
fibonacci	aus80%	346827689.4680	26035117.42576998	9215161	0,00%
fibonacci	aus100%	219663896.1000	21410188.20266184	9215175	0,00%
fibonacci	ausNull	408529134.8160	28032164.636706896	9215149	0,00%
fibonacci	top20%	235613051.3230	17341440.322005577	5811694	36,93%
fibonacci	top40%	254957504.3015	20544270.971817367	7513670	18,46%
fibonacci	top60%	397890355.2635	27702981.567639966	9215149	0,00%
fibonacci	top80%	344699912.2320	25962195.819153536	9215161	0,00%
fibonacci	top100%	224057174.0220	21611846.504473418	9215175	0,00%
fibonacci	topNull	361718504.7850	26537304.839443628	9215149	0,00%
fibonacci	DAus20%	137087416.4400	15216402.383229893	7213182	21,72%
fibonacci	DAus40%	183353922.2070	18461744.152298193	8014168	13,03%
fibonacci	DAus60%	264464528.8950	22806148.80884763	8814661	4,35%
fibonacci	DAus80%	243829109.0400	22491843.006934717	9215164	0,00%
fibonacci	DAus100%	224057174.0220	21611846.504473418	9215175	0,00%
fibonacci	DAusNull	241630321.5150	22396027.56990074	9215154	0,00%
fibonacci	DTop20%	205800009.0960	18983875.689155776	7613673	17,38%
fibonacci	DTop40%	227317829.3475	20968771.737267826	8414661	8,69%
fibonacci	DTop60%	276776550.0990	23867878.29253046	9215154	0,00%
fibonacci	DTop80%	239435791.7600	22300088.59863511	9215164	0,00%
fibonacci	DTop100%	252613480.5150	22869058.31257842	9215175	0,00%
fibonacci	DTopNull	272383271.5260	23690324.436693665	9215154	0,00%

Tabelle A.1: Ergebnisse der Fehlerinjektionskampagnen zu fibonacci mit jeweils 2000 Samples.

```
63    unreachable
64
65  .split.split.split.split.split.split.split.split.split.split.split: ; preds
      = %.split.split.split.split.split.split.split.split.split.split
66    ret i32 0
67  }
```

Listing A.3: Aussehen eines Programms im selektiven Lauf

benchmark	variant	EAFC	EAFC-SE	Laufzeit	Verringerung
basicmath	eddi	3369324477.6360	175241419.88253972	17863545	0,00%
basicmath	normal	3001024666.7680	150412367.9963224	13560348	24,09%
basicmath	aus20%	2984895116.4080	160141177.4248255	15661293	12,33%
basicmath	aus40%	3654615788.0750	184169201.100481	17562087	1,69%
basicmath	aus60%	3598843208.2785	184043617.10028887	17853705	0,06%
basicmath	aus80%	3563430987.7260	180560305.66761655	17856871	0,04%
basicmath	aus100%	3758092686.5940	183091376.0429432	17863545	0,00%
basicmath	ausNull	3506595519.8625	178006721.28349793	16979190	4,95%
basicmath	top20%	3548122740.9600	181787195.1909937	17680195	1,03%
basicmath	top40%	3566380752.1755	180379344.1689865	17702251	0,90%
basicmath	top60%	3440964246.2400	176296956.0932428	17856757	0,04%
basicmath	top80%	3660852272.2200	184148621.3409118	17860017	0,02%
basicmath	top100%	3585306815.9460	179696786.2848567	17863545	0,00%
basicmath	topNull	3627493955.3615	182802434.64598122	17680117	1,03%
basicmath	DAus20%	3186355374.9150	167963201.83960018	16864586	5,59%
basicmath	DAus40%	3491078740.2870	181229970.18545324	17812957	0,28%
basicmath	DAus60%	3305562167.8500	175951982.6419793	17856934	0,04%
basicmath	DAus80%	3764199420.9715	186977263.7205607	17860235	0,02%
basicmath	DAus100%	3423320062.2135	176378211.42885986	17863545	0,00%
basicmath	DAusNull	3656830711.3860	183281588.0362658	17774840	0,50%
basicmath	DTop20%	2963041030.5405	166643167.8346989	17774876	0,50%
basicmath	DTop40%	3465866842.6200	180262670.05624953	17791307	0,40%
basicmath	DTop60%	3338617789.5285	176673459.11118037	17856934	0,04%
basicmath	DTop80%	3082845816.5820	172286360.45251048	17860235	0,02%
basicmath	DTop100%	3272132425.3965	173155176.9950578	17863545	0,00%
basicmath	DTopNull	3381466952.9985	177561759.91615233	17774840	0,50%

Tabelle A.2: Ergebnisse der Fehlerinjektionskampagnen zu basicmath mit jeweils 2000 Samples.

benchmark	variant	EAFC	EAFC-SE	Laufzeit	Verringerung
bubblesort	eddi	9280546.8915	364784.4980604421	108706	0,00%
bubblesort	normal	6079021.2560	233570.88400632975	69080	36,45%
bubblesort	aus20%	7999786.9830	296387.43045609834	88493	18,59%
bubblesort	aus40%	9935585.7125	349745.37995129696	102143	6,04%
bubblesort	aus60%	9714847.3045	348776.38139110146	102774	5,46%
bubblesort	aus80%	9716927.7660	354568.26181238994	105356	3,08%
bubblesort	aus100%	9584204.8675	368736.38857054035	108706	0,00%
bubblesort	ausNull	8130037.7600	309916.037017725	93808	13,70%
bubblesort	top20%	8801622.0810	312881.0677668868	91571	15,76%
bubblesort	top40%	9797351.4765	348277.3701306802	102143	6,04%
bubblesort	top60%	9523678.5740	346643.2601637995	102774	5,46%
bubblesort	top80%	9485572.3430	351884.38490602653	105356	3,08%
bubblesort	top100%	9982755.9610	373672.66235002945	108706	0,00%
bubblesort	topNull	7796579.1800	305627.8806205868	93808	13,70%
bubblesort	DAus20%	8806357.8135	335257.2136930287	97931	9,91%
bubblesort	DAus40%	9287051.5080	357777.7964775067	105436	3,01%
bubblesort	DAus60%	9159857.9325	357140.73705294373	105904	2,58%
bubblesort	DAus80%	9817201.3585	368902.9296482398	107484	1,12%
bubblesort	DAus100%	9830926.9730	371825.0920827321	108706	0,00%
bubblesort	DAusNull	8192312.6220	331434.6996763184	99955	8,05%
bubblesort	DTop20%	8332923.9790	332031.4884370169	99309	8,64%
bubblesort	DTop40%	9526598.4715	360784.74580684816	105436	3,01%
bubblesort	DTop60%	9659486.5470	363446.9341015682	105904	2,58%
bubblesort	DTop80%	9329156.9315	362767.77119930997	107484	1,12%
bubblesort	DTop100%	9128717.9035	362744.75164693303	108706	0,00%
bubblesort	DTopNull	8927520.1650	341211.67928450886	99955	8,05%

Tabelle A.3: Ergebnisse der Fehlerinjektionskampagnen zu bubblesort mit jeweils 2000 Samples.

benchmark	variant	EAFC	EAFC-SE	Laufzeit	Verringerung
simple	eddi	21126349.3375	2809234.0069880886	897250	0,00%
simple	normal	5232014.5680	300003.8607688149	193599	78,42%
simple	aus20%	12860396.8680	1917332.6158731917	476757	46,86%
simple	aus40%	19400621.4070	2425390.5689381533	676769	24,57%
simple	aus60%	20812267.2960	2559570.4441863457	796769	11,20%
simple	aus80%	19974488.0400	2733716.433356375	897242	0,00%
simple	aus100%	22662811.1075	2906601.832588042	897250	0,00%
simple	ausNull	22891847.9700	2910657.4354062616	897232	0,00%
simple	top20%	13444960.3620	1959421.6042963385	476757	46,86%
simple	top40%	19087708.1585	2406372.054551032	676769	24,57%
simple	top60%	22438225.6785	2654239.4843489756	796769	11,20%
simple	top80%	19206238.5000	2682005.247518767	897242	0,00%
simple	top100%	22278695.6650	2882606.616140583	897250	0,00%
simple	topNull	22510317.1705	2887043.83712842	897232	0,00%
simple	DAus20%	21327330.9260	2541971.134527975	666762	25,69%
simple	DAus40%	20112975.7845	2535626.721646495	816773	8,97%
simple	DAus60%	18718369.6805	2641092.5309316833	877235	2,23%
simple	DAus80%	14980680.1170	2375324.9257943933	897243	0,00%
simple	DAus100%	26503965.5325	3135179.79612765	897250	0,00%
simple	DAusNull	26120434.7540	3113253.664810196	897235	0,00%
simple	DTop20%	18818233.1700	2392704.6156016057	666762	25,69%
simple	DTop40%	16815766.6395	2324462.292780305	816773	8,97%
simple	DTop60%	19100377.2250	2667222.5249643726	877235	2,23%
simple	DTop80%	21126600.1650	2809267.36027273	897243	0,00%
simple	DTop100%	23431041.9925	2953932.6665917877	897250	0,00%
simple	DTopNull	13060217.3770	2220688.818743929	897235	0,00%

Tabelle A.4: Ergebnisse der Fehlerinjektionskampagnen zu simple mit jeweils 2000 Samples.

benchmark	variant	EAFC	EAFC-SE	Laufzeit	Verringerung
bitcount	eddi	39862686.4615	1055029.2326644768	136864	0,00%
bitcount	normal	34228137.2160	849569.2765419815	113465	17,10%
bitcount	aus20%	36381709.2670	919638.7695273171	123596	9,69%
bitcount	aus40%	36011574.4800	932804.1840586287	126040	7,91%
bitcount	aus60%	38884582.7160	972989.9526500057	130390	4,73%
bitcount	aus80%	39789715.4150	1019167.8428356559	131944	3,59%
bitcount	aus100%	39216996.5100	1051945.9094218963	136764	0,07%
bitcount	ausNull	36130217.4375	925434.0068257579	124700	8,89%
bitcount	top20%	38155506.3800	924400.5344485592	123724	9,60%
bitcount	top40%	38573328.8025	939239.7050180461	126010	7,93%
bitcount	top60%	37921225.9370	970319.9315920253	130390	4,73%
bitcount	top80%	37719730.2200	1011783.6509817536	131944	3,59%
bitcount	top100%	39934380.5925	1054755.2111511186	136764	0,07%
bitcount	topNull	37550364.7125	929208.0750801763	124700	8,89%
bitcount	DAus20%	37703613.3800	995835.7210441073	129598	5,31%
bitcount	DAus40%	38075085.2575	1005647.1139535744	130990	4,29%
bitcount	DAus60%	39992530.4315	1030647.4341673575	133390	2,54%
bitcount	DAus80%	38700114.6140	1031671.1315516566	134184	1,96%
bitcount	DAus100%	40699590.2805	1057475.401258366	136764	0,07%
bitcount	DAusNull	36692108.6790	997564.8265694802	130570	4,60%
bitcount	DTop20%	38743279.1460	999472.3542100918	129604	5,30%
bitcount	DTop40%	39702082.6995	1010732.3945060327	130940	4,33%
bitcount	DTop60%	40644329.5305	1032618.2639482121	133390	2,54%
bitcount	DTop80%	38606409.7360	1031298.7059106533	134184	1,96%
bitcount	DTop100%	38308310.0055	1048024.2430192156	136764	0,07%
bitcount	DTopNull	37737856.5100	1001880.9425769504	130570	4,60%

Tabelle A.5: Ergebnisse der Fehlerinjektionskampagnen zu bitcount mit jeweils 2000 Samples.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Literaturverzeichnis

[1] *eCos Homepage*. URL: http://ecos.sourceware.org/. Zugegriffen am: 17.05.2019.

[2] Bouricius, W. G., W. C. Carter und P. R. Schneider: *Reliability modeling techniques for self-repairing computer systems*. ACM '69 Proceedings of the 1969 24th national conference, Seiten 295–309, August 1969. ACM Press.

[3] Dirschuweit, J.: *Quantitative Bewertung des EDDI-Fehlerdetektionsverfahrens unter Berücksichtigung der Programmlaufzeit*. Bachelorarbeit, Technische Universität Dortmund, Sept. 2018.

[4] Goloubeva, O., M. Rebaudengo, M. S. Reorda und M. Violante: *Software-Implemented Hardware Fault Tolerance*. Springer, New York, NY, USA, 2006.

[5] Guthaus, Matthew R. u. a.: *MiBench: A Free, Commercially Representative Embedded Benchmark Suite*. Proceedings of the IEEE International Workshop on Workload Characterization (WWC '01), Seiten 3–14, 2001. IEEE Computer Society Press. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.15.

[6] Hoffmann, M.: *Konstruktive Zuverlässigkeit: Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware*. Dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016. URL: https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/7038.

[7] Oh, N., P. P. Shirvani und E. J. McCluskey: *Error detection by duplicated instructions in super-scalar processors*. IEEE Transactions on Reliability, 51(1):63–75, März 2002. DOI: 10.1109/24.994913. URL: https://doi.org/10.1109/24.994913.

[8] Sampson, A.: *LLVM for Grad Students*, 2015. URL: https://www.cs.cornell.edu/ asampson/blog/llvm.html. Zugegriffen am: 15.05.2019.

[9] Schirmeier, H.: *Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance*. Dissertation, Technische Universität Dortmund, 2016.

[10] Schirmeier, H., C. Borchert und O. Spinczyk: *Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors*. Proceedings of the 45th

IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15), Seiten 319–330, Juni 2015. IEEE Press. DOI: 10.1109/DSN.2015.44.

[11] Schirmeier, H., M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann und O. Spinczyk: *FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance.* Proceedings of the 11th European Dependable Computing Conference (EDCC '15), Seiten 245–255, Sept. 2015. IEEE Press. DOI: 10.1109/EDCC.2015.28.

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