technische universität
dortmund

Bachelorarbeit

Zuordnung von Fehlerinjektionsergebnissen zu dynamischen Datenstrukturen

Michael Lenz
23. September 2016

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

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

Zusammenfassung

Fehlerinjektion ist ein experimentelles Werkzeug zur Untersuchung der Zuverlässigkeit von Hard- und Softwaresystemen. Mit dieser Methodik werden insbesondere durch Strahlungseffekte hervorgerufene Hardwarefehler imitiert, um so systematisch die Fehlertoleranz des verwendeten Computersystems – einschließlich verwendeter Software-implementierter Hardwarefehlertoleranzverfahren – zu analysieren und zu bewerten.

Zur Einschätzung des Einflusses einzelner Programmvariablen auf die Fehlertoleranz der untersuchten Software müssen die Fehlerinjektionsergebnisse auf konkrete Datenstrukturen abgebildet werden. Bei Fehlerinjektion mit Werkzeugen, wie dem am diese Arbeit betreuenden Lehrstuhl entwickelten FAIL* [Sch+15], ist dies derzeit nur unter hohem manuellem Aufwand möglich, oder nur relativ grobgranular auf Ebene von globalen und statischen, in der Symboltabelle des Programms gelisteten, Datenstrukturen.

Diese Bachelorarbeit entwickelt Methoden und Werkzeuge zur feingranularen Zuordnung von Fehlerinjektionsergebnissen auf zur Programmlaufzeit nicht notwendigerweise ortsfeste Datenstrukturen, beispielsweise im Stapelspeicher.

Hierzu wird FAIL* um eine Komponente erweitert, welche während eines sogenannten „*golden-run*“ – ein fehlerfreier, deterministisch reproduzierbarer Simulationslauf – zusätzliche Laufzeitinformationen aufzeichnet. Diese Informationen werden anschließend genutzt, um den Datenstrukturen des Zielprogramms Lebenszeit-Intervalle und korrespondierende Speicheradressen zuzuordnen. In Folge wird diese Zuordnung genutzt, um die Ergebnisse der Fehlerinjektionen auf die Datenstrukturen abzubilden. Der Ansatz und die ermittelten Zuordnungen werden anhand von Testprogrammen des Embedded-Betriebssystems eCos[1] mit Hilfe geeigneter Metriken evaluiert.

[1] **E**mbedded **C**onfigurable **O**perating **S**ystem, http://ecos.sourceware.org/

Inhaltsverzeichnis

1. Einleitung **1**
- 1.1. Zielsetzung . . . 2
- 1.2. Abgrenzung . . . 3
- 1.3. Aufbau der Arbeit . . . 3

2. Grundlagen und verwandte Arbeiten **5**
- 2.1. Hardwarefehler . . . 5
  - 2.1.1. Ursachen . . . 5
  - 2.1.2. Fehlerbegriff . . . 6
  - 2.1.3. Fehlertoleranz . . . 6
  - 2.1.4. Injektion . . . 7
- 2.2. Code-Blöcke . . . 9
  - 2.2.1. Sichtbarkeit . . . 9
  - 2.2.2. Lebensdauer . . . 10
- 2.3. Debugging . . . 11
  - 2.3.1. Kontrollfluss . . . 12
- 2.4. FAIL* . . . 12
- 2.5. Verwandte Arbeiten . . . 14
- 2.6. Zusammenfassung . . . 15

3. Analyse und Entwurf **17**
- 3.1. Zuordnung von FI-Ergebnissen . . . 17
  - 3.1.1. Speicherbereiche . . . 18
  - 3.1.2. Lebenszeiten . . . 21
  - 3.1.3. Anteilige FI-Ergebnisse . . . 23
- 3.2. VProf . . . 23
  - 3.2.1. Architektur . . . 24
  - 3.2.2. Debugger . . . 24
  - 3.2.3. Datenbank . . . 27
- 3.3. Zusammenfassung . . . 29

4. Implementierung **31**
4.1. FAIL* . . . 31
4.1.1. Modifikationen der Datenbank . . . 31
4.1.2. Backend . . . 32
4.1.3. Import-Werkzeug . . . 33
4.2. GDB . . . 33
4.2.1. Kommunikation . . . 33
4.2.2. Steuerung des Inferior . . . 34
4.2.3. Erhebung von Beobachtungen . . . 36
4.3. VProf . . . 39
4.3.1. StepObserver . . . 40
4.3.2. Pruning . . . 40
4.3.3. Ermittlung anteiliger FI-Ergebnisse . . . 40
4.3.4. Auswertungsmöglichkeiten . . . 42
4.4. Zusammenfassung . . . 43

5. Evaluation **45**
5.1. Symbole . . . 45
5.1.1. Sichtbarkeit . . . 45
5.1.2. Anteilige Ergebnisse . . . 46
5.1.3. Performanz . . . 47
5.2. N-Stepping . . . 48
5.2.1. Performanzgewinn . . . 48
5.2.2. Aliasing . . . 48
5.3. Zuordenbare Ergebnisse . . . 49

6. Zusammenfassung und Ausblick **51**
6.1. Zusammenfassung [WIP] . . . 51
6.2. Ausblick . . . 52
6.3. Danksagungen . . . 52

Literaturverzeichnis **53**

Abbildungsverzeichnis **57**

A. Anhang **I**

1. Einleitung

Unsere Zivilisation ist ohne Computer, eingebettete Systeme und integrierte Schaltungen mittlerweile nicht mehr denkbar. Um Computer leistungsfähiger, günstiger und nicht zuletzt sicherer zu machen, sind sie Gegenstand aktueller Forschung und werden stets weiterentwickelt.

Das Moore'sche Gesetz [Sch97] besagt, dass sich die Komplexität beziehungsweise die Packungsdichte integrierter Schaltungen etwa alle zwei Jahre verdoppelt. Eine Erhöhung der Dichte ist nur durch eine weitere Miniaturisierung der einzelnen Transistoren zu erzielen und führt in Kombination mit Energiesparmaßnahmen, wie reduzierten Versorgungsspannungen [Int13], zu einer erhöhten Anfälligkeit von Computersystemen gegenüber ionisierender Strahlung [Bor05], [NX06]. Diese kann unterschiedlichste Auswirkungen auf den Ablauf und das Ergebnis eines, auf anfälliger Hardware laufenden, Programms haben, welche bis hin zu Sach- und Personenschäden reichen können.

Zur Illustration der Auswirkungen solcher Fehler sei hier der Fall des Toyota Camry beziehungsweise dessen elektronischen Gaspedals angeführt: Unter gewissen Rahmenumständen und ohne direktes Zutun des Fahrers war es möglich, dass das Fahrzeug selbstständig beschleunigen und im Extremfall einen Unfall verursachen konnte. Diese Umstände traten hinreichend oft auf, um zu mehreren hundert – vgl. J. Yoshida [Yos13] – Sachschadens-, Verletzungs- und Todesfällen zu führen. Im Zuge der gerichtlichen Ermittlungen eines Falls aus dem Jahr 2007 wurde demonstriert, dass schon ein einzelnes gekipptes Bit, ein Einzelbitfehler, im Hauptspeicher als Ursache ausreichend gewesen wäre. Der konkrete Nachweis, dass es sich um ein strahlungsinduzierte Einzelbitfehler gehandelt hat, ist aufgrund der flüchtigen Natur transienter Hardwarefehler und der Nichtexistenz 100-prozentig vollständiger Logfiles des Speicherinhaltes nicht erbringbar. Ungeachtet dieser Tatsache muss die bloße Möglichkeit einer solchen Auswirkung ausgeschlossen werden, weshalb es notwendig ist an Hardwarefehlertoleranz zu forschen.

Das diese Arbeit umgebende DFG-Projekt DanceOS [Sch+11] und insbesondere dessen Teilprojekt FAIL* [Sch+15] – vgl. Kapitel 2.4 – beschäftigt sich mit der Untersuchung und Implementierung von Hardwarefehlertoleranz in Software („*Software-implemented Hardware Fault-Tolerance*“, *SIHFT*): Zur Simulation transienter Hardwarefehler werden Fehler in die Architektur des Systems injiziert,

auf welchem die Ziel-Software läuft, und das sich ergebende Verhalten der Software betrachtet, um daraus Rückschlüsse auf ihre Fehlertoleranz zu ziehen.

Zur Schaffung resp. Erhöhung der Toleranz einer Software gegenüber Fehlern in der unterliegenden Hardware (Härtung), können verschiedene Maßnahmen aus den Bereichen Detektion und Korrektur – vgl. Abschnitt 2.1.3 – ergriffen werden. In einem iterativen Prozess von Fehlerinjektion, Analyse und Härtung kann die Ziel-Software so gegen Hardwarefehler gesichert werden.

Zum gegenwärtigen Zeitpunkt sind die Ergebnisse der Fehlerinjektionen nur grobgranular oder mit viel manuellem Aufwand auf den inneren Aufbau und Zustand der Ziel-Software abbildbar. So ist es lediglich möglich die Ergebnisse räumlich auf bestimmte Speicherbereiche, wie den Stack, oder zeitlich, beispielsweise auf die Laufzeiten einer bestimmten Funktion, einzugrenzen. Aus diesem Grund können Fehlertoleranzmaßnahmen aktuell nur grobgranular eingeführt und untersucht werden. Dies ist suboptimal, weil SIHFT die Fehlertoleranz einer Software auch negativ beeinflussen kann [Mar+12].

1.1. Zielsetzung

Zur Behebung dieses Missstands, entwirft und entwickelt diese Arbeit Methoden und Werkzeuge zur feingranularen Untersuchung der Fehlertoleranz einer Software in Abhängigkeit ihrer internen Datenstrukturen.

Da zum gegenwärtigen Zeitpunkt keine Möglichkeit existiert allen Datenstrukturen einer Ziel-Software ihren Anteil an der Gesamt-Fehleranfälligkeit zuzuordnen, ist es erforderlich eine Abbildung von Fehlerinjektionsergebnissen auf Datenstrukturen zu finden, und nutzbar zu machen. Hierfür ist es notwendig zu wissen, welche Speicherbereiche zu welchen Zeitpunkten „lebendig“ sind, welche Speicherbereiche zu welchen Zeitpunkten Daten der untersuchten Software beinhalten, und um welche Datenstrukturen es sich dabei handelt; Abbildung 1.1 illustriert dies.

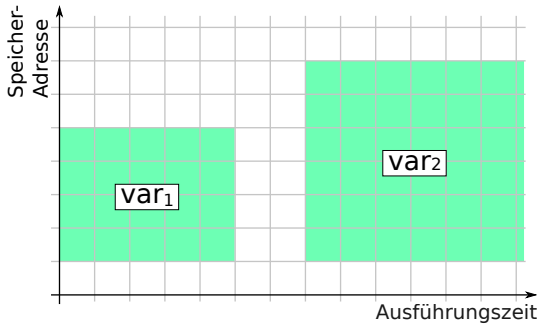


Abbildung 1.1.: Lebendige Speicherbereiche und zugeordnete Variablen

Die für diese Zuordnung notwendigen Informationen finden sich in den vom Tool FAIL* [Sch+15] erhobenen Daten, sowie den Debug-Informationen [10] der Ziel-Software. Zur Erhebung und Verknüpfung dieser Daten werden Werkzeuge entworfen und implementiert, mit denen ermittelt werden kann, welchen Anteil an der Fehleranfälligkeit der Ziel-Software jede einzelne ihrer Datenstrukturen hat. So wird zum Beispiel eine Aufstellung der „n fehleranfälligsten“ Datenstrukturen, Datentypen oder Übersetzungseinheiten möglich.

1.2. Abgrenzung

Der entwickelte Ansatz zur Zuordnung von Fehlerinjektionsergebnissen zu Datenstrukturen ist grundsätzlich für jede Software anwendbar, die mit FAIL* untersucht werden kann.

Als Erklärungsgrundlage dienen im Rahmen dieser Arbeit gleichverteilte Einzelbitfehler im Hauptspeicher. Die entwickelten Methoden und Werkzeuge sind jedoch z.B. auch für Burst- oder Bus-Fehler sowie andere Verteilungen anwendbar. Prinzipiell ist der hier entwickelte Ansatz auch auf implizite Datenstrukturen wie Rücksprungadressen oder auf CPU-Register übertragbar, dies wird hier jedoch nicht verfolgt, da dies den Rahmen dieser Arbeit sprengen würde.

1.3. Aufbau der Arbeit

Die vorliegende Arbeit folgt in ihrem Aufbau dem Beispiel anderer wissenschaftlicher Arbeiten und ist in fünf weitere Kapitel gegliedert:

- *Kapitel 2* erläutert die Grundlagen der Arbeit wie z.B. das Fehlerinjektions- und Analyse-Framework FAIL*, definiert wichtige Begriffe wie „Fehlerraum“ und „Fehlerinjektion“ und beschreibt verwandte Arbeiten.

- *Kapitel 3* analysiert das Kernproblem dieser Arbeit, die Zuordnung von Fehlerinjektionsergebnissen zu dynamischen Datenstrukturen, und entwickelt einen konkreten Lösungsansatz.

- *Kapitel 4* beschreibt die konkrete Umsetzung der Kernkomponenten des Lösungsansatzes. Ferner werden aufgetretene Probleme inklusive deren Behandlung resp. Lösung beschrieben.

- *Kapitel 5* dient der Evaluation des Ansatzes und der Methoden/Werkzeuge anhand der Testprogramme des Embedded-Betriebssystems eCos.

- *Kapitel 6* rekapituliert wesentliche Erkenntnisse und skizziert Zukunftsperspektiven, die diese Arbeit eröffnet.

2. Grundlagen und verwandte Arbeiten

Dieses Kapitel hat zum Ziel die zum Verständnis der Arbeit notwendigen Grundlagen und Beziehungen zu vermitteln. Ferner werden verwandte Arbeiten angeführt und diese Arbeit in Relation zu ihnen dargestellt.

2.1. Hardwarefehler

Der deutsche Begriff „Fehler“ ist – zumindest im Rahmen dieser Arbeit – kontextsensitiv und damit zu unscharf. Dieser Abschnitt dient daher der Abgrenzung der verschiedenen Bedeutungen, der Beschreibung von Fehlerursachen und Fehlertoleranz, sowie der Einführung von Fehlerinjektion als experimentelles Werkzeug zur Untersuchung von Fehlertoleranz.

2.1.1. Ursachen

Hardwarefehler lassen sich in permanente und transiente Fehler untergliedern: Permanenten Hardwarefehlern liegt ein persistenter Defekt zugrunde, dessen Auswirkungen entsprechend permanent existieren. Transiente[1] Fehler dagegen resultieren aus sporadisch und zeitlich begrenzt auftretenden Defekten, die beispielsweise durch ionisierende Strahlung [BSH75], [MW79] hervorgerufen werden, und haben entsprechend nicht-permanenten Auswirkungen. Eine mögliche Auswirkung eines transienten Hardwarefehlers ist eine Änderung des Zustands einer RAM-Speicherzelle, und damit ein Einzelbitfehler im Hauptspeicher. Diese Arbeit beschäftigt sich ausschließlich mit der feingranularen Untersuchung der Auswirkungen transienter Hardwarefehler.

Eine Metrik für die Eintrittshäufigkeit solcher *Single-Event-Upsets* [GWA79] ist die Failure-In-Time (FIT), welche 1 *fault* pro 10^9 h [Muk08] meint. In der Literatur wird für aktuelle DRAM-Technologie eine gemessene FIT-Rate von 0.044 FIT/Mbit [SL12] bis 0.066 FIT/Mbit [Sri+13] angegeben, sodass man pro Mbit Speicher etwa 0.055 FIT/Mbit bzw. $5.5 * 10^{-11}$ *faults* pro Stunde erwarten kann.

[1] von lat. „transire“: vorbeigehen

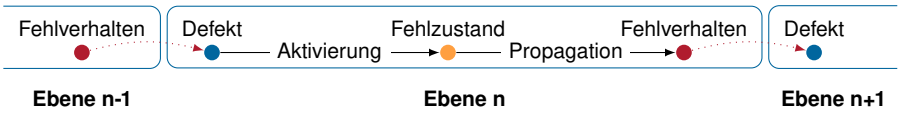


Abbildung 2.1.: Schematische Darstellung der „Fehlerkette“. Entnommen aus [Hof16]

2.1.2. Fehlerbegriff

Hoffmann [Hof16] beschreibt die „Fehlerkette“ von einem ursprünglichen Defekt (*fault*) in der Hardware des zugrunde liegenden Systems, über einen dadurch verfälschten System- und Programmzustand (*error*) zu einem fehlerhaften Verhalten (*failure*) des Systems beziehungsweise der Ziel-Software.

Diese Fehlerkette wird jedoch nicht notwendigerweise vollständig durchschritten, und kann durchbrochen werden, sofern:

- der *fault* „gutartig“ ist, d.h. der veränderte Zustand nie gelesen oder vor einem lesenden Zugriff überschrieben wird.
- der *error* „gutartig“ ist, d.h. der veränderte Zustand zwar gelesen wird, jedoch das Verhalten der Software nicht beeinflusst.
- *Fehlertoleranzmaßnahmen* einen *fault* bzw. *error* detektieren oder korrigieren. Siehe Abschnitt 2.1.3.

Der Übergang von *fault* zu *error* wird „Aktivierung“ genannt, und findet statt, sobald der veränderte Zustand gelesen wird. Der Übergang von *error* zu *failure* dagegen wird „Propagation“ genannt, und findet statt, sobald ein fehlerhaftes Verhalten resultiert. Sofern eine Ziel-Software aus mehreren Komponenten – oder „Ebenen“ – zusammengesetzt ist, die miteinander interagieren, kann das Fehlverhalten einer Komponente als Defekt in eine andere Komponente propagiert werden. Abbildung 2.1 illustriert diesen Sachverhalt.

Der Kürze und begrifflichen Prägnanz wegen verwendet diese Arbeit in Folge, wo nötig, die englischen Begriffe. Sofern nicht anders angeführt, meint der deutsche Begriff „Fehler“ im Kontext dieser Arbeit *fault*. Damit meint zum Beispiel „Fehlertoleranz“, welche in Abschnitt 2.1.3 thematisiert wird, die Toleranz eines Systems gegenüber fehlerverursachenden Defekten.

2.1.3. Fehlertoleranz

Avižienis et al. [ALR01] charakterisieren *Fehlertoleranz* als die Fähigkeit eines Systems bzw. einer Software trotz Vorliegen von *faults* oder *errors* korrekt zu funktionieren. Diese Fähigkeit wird durch *Fehlertoleranzmaßnahmen* aus den Bereichen Erkennung und Korrektur ermöglicht.

Allen Fehlertoleranzmaßnahmen gemein ist, dass sie auf Redundanz basieren und auf Grundlage dieser eine Detektion, sowie möglicherweise eine Korrektur von *faults* und *errors* erlauben [Sch16]. Auf diese Weise kann die zuvor beschriebene Fehlerkette durchbrochen werden.

Wie bereits in der Einleitung angeführt, bewegt sich das umgebende Projekt FAIL* [Sch+15] im Kontext der Software-implementierten Hardwarefehlertoleranz. Fehlertoleranzmaßnahmen in Software haben gegenüber einer Implementierung in Hardware den Vorteil, dass sie unter Nutzung von Wissen über die Anwendung implementiert werden können. Hierdurch ist beispielsweise ein Schützen ausschließlich der kritischsten Datenstrukturen der Ziel-Software möglich. Nachteilig ist jedoch, dass diese Maßnahmen sowohl die Programmlaufzeit verlängern, als auch den Speicherbedarf erhöhen [Mar+12], was zur Folge hat, dass die Angriffsfläche für Hardwarefehler größer wird – vgl. Abschnitt 2.1.1. Festzustellen ist somit, dass ein Einsatz von SIHFT-Maßnahmen dosiert erfolgen muss, um keinen insgesamt nachteiligen Effekt zu erzielen[2] [Sch16].

Eine vergleichsweise einfache, detektierende Maßnahme ist der Vergleich des geschützten Speicherinhalts mit zuvor für diesen zusätzlich eingeführten Paritätsinformationen. Hierdurch wird eine Detektion von Fehlern, und damit eine adäquate Reaktion, wie beispielsweise ein Neustart des Programms, ermöglicht. Eine Fehlerkorrektur ist z.B. mittels Reed-Solomon-Codes [RS60] oder AN-Kodierung [Bro60] möglich.

2.1.4. Injektion

Für eine gegebene Zielsoftware ist das Verhalten bei Vorliegen von Fehlern a priori nicht bekannt und ohne Weiteres nicht vorhersagbar. Um Aussagen über die Fehlertoleranz der untersuchten Software treffen zu können, muss ihr Verhalten bei Vorliegen eines Defekts an jeder möglichen Raum-Zeit-Koordinate – Speicher-Bit und Ausführungszeitpunkt – bekannt sein.

Zur Ermittlung dieser Information, wird für jede dieser Koordinaten eine *Fehlerinjektion* (FI) in einem ansonsten fehlerfreien, deterministisch reproduzierbaren Lauf der Ziel-Software durchgeführt. Nach erfolgter Injektion des Fehlers wird die Ausführung der Software fortgesetzt, und das emergente Verhalten beobachtet und klassifiziert [Sch16]. In Folge werden vier Beispiele solcher Klassen erklärt. Je nach Anforderungen an die Untersuchung kann die Klassifizierung auch deutlich fein- oder grobgranularer vorgenommen werden.

- *OK*: Das FI-Experiment terminierte *failure*-frei, beispielsweise, weil der *fault* korrigiert wurde.

[2] Gemäß Paracelsus [Hoh38] gilt diese Aussage für *alles*.

- *SDC*: Das FI-Experiment führte zu einer „*silent data corruption*“, d.h. einer von der Software unbemerkten Änderung des außen messbaren Verhaltens. Das Verhalten des Toyota Camry, unter Vorliegen von Einzelbitfehlern, welches in der Einleitung angeführt wurde, ist ein Beispiel hierfür.
- *TIMEOUT*: Das FI-Experiment führte zu einer Terminierung des Programmlaufs, aufgrund der Überschreitung einer Laufzeitschranke.
- *TRAP*: Das FI-Experiment löste eine CPU-Trap bzw. CPU-Exception aus.

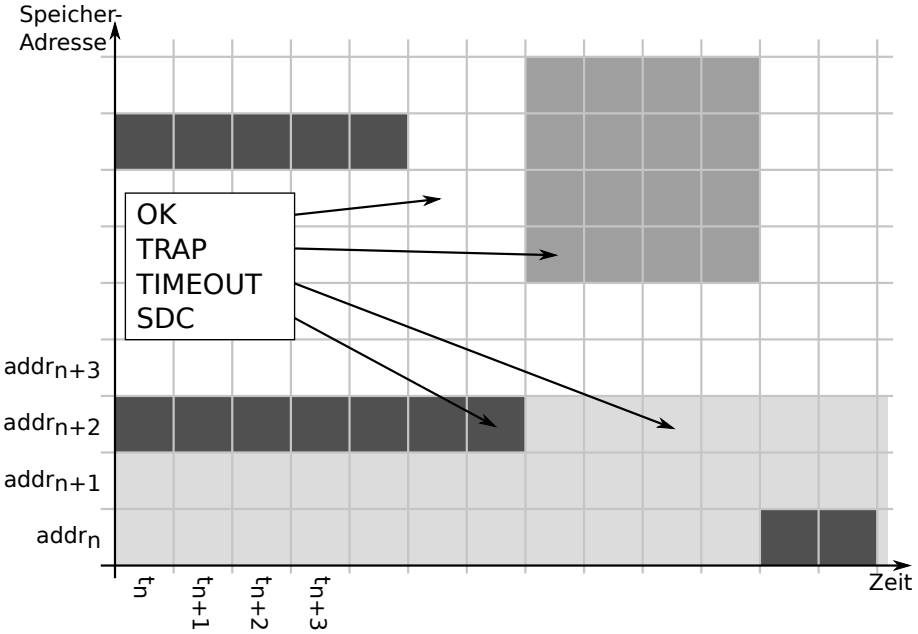


Abbildung 2.2.: Beispielhafter, schematischer Fehlerraumplot. Verhaltensklassen sind annotiert.

Diese Injektion eines Fehlers in einen *golden-run*, sowie das Beobachten und Klassifizieren des resultierenden Verhaltens wird *Fehlerinjektionsexperiment* genannt; die Gesamtheit aller durchzuführenden Experimente bildet eine sogenannte *Fehlerinjektionskampagne* [Sch+15].

Die Kombination der Raum-Zeit-Koordinaten der FI-Kampagne und dem jeweils resultierenden Verhalten wird als *Fehlerraum* bezeichnet, und kann mittels sogenannter *Fehlerraumplots* dargestellt werden [Sch16] [Sch+15] Abbildung 2.2 stellt beispielhaft einen Ausschnitt eines solchen Plots dar; die zu den Farben gehörigen Verhaltensklassen sind annotiert. Die Achsen eines Fehlerraumplots stehen für Ausführungszeitpunkte im *golden-run* (x-Achse) und Hauptspeicher-Adressen

(y-Achse). Die Zeit-Achse ist in dynamischen Instruktionen – Anzahl bis dato ausgeführter Instruktionen – aufgelöst, die Adress-Achse in einzelnen Bits. Die Farbe einer Fehlerraumkoordinate repräsentiert das emergente, äußere Verhalten der Ziel-Software bei Fehlerinjektion in genau diese Koordinate, und damit das Ergebnis des, zur Koordinate gehörigen, FI-Experiments.

Fehlerraumplots werden in Abschnitt 3.1 zur Illustration einer der in dieser Arbeit entwickelten Methoden herangezogen.

2.2. Code-Blöcke

Diese Arbeit betrachtet Code-Blöcke – nachfolgend Blöcke genannt – als logisches Konstrukt. Es wird exemplarisch mit C- bzw. C++-Quellcode gearbeitet, welcher durch Paare geschweifter Klammern untergliedert wird; die getroffenen Aussagen gelten jedoch – möglicherweise mit Einschränkungen – auch für andere Programmiersprachen wie Java oder Python.

Eine Verschachtelung von Code, wie sie beispielsweise in Bedingungs- und Schleifenkonstrukten auftritt, bildet sich entsprechend auch in der Hierarchie der logischen Blöcke ab [Soud]. Der Zusammenhang von Codestruktur und Block-Hierarchie ist exemplarisch in Abbildung 2.3 skizziert: Der gesamte Code eines Programms wird auf den sogenannten globalen Block abgebildet. Unterhalb des globalen Blocks befinden sich die sogenannten statischen Blöcke, von denen jeder zu einer der Übersetzungseinheiten des Programms korrespondiert. Innerhalb der statischen Blöcke finden sich die Blöcke der in der jeweiligen Übersetzungseinheit definierten Funktionen mit entsprechendem Namen, und in diesen wiederum namenlose, anonyme Blöcke von beispielsweise if-Abfragen und Schleifen.

Aus dieser Block-Hierarchie ergeben sich sowohl der Sichtbarkeitsbereich, als auch die Lebensdauer von Variablen; die folgenden beiden Unterabschnitte befassen sich damit.

2.2.1. Sichtbarkeit

Der Sichtbarkeitsbereich einer Variablen, auch *Scope* genannt, beschränkt sich grundsätzlich auf Instruktionen des Blocks, in dem ihre Deklaration erfolgt, sowie auf Instruktionen aller hierarchisch untergeordneten Blöcke.

Eine globale Variable ist entsprechend eine Variable, deren Deklaration im globalen Block, außerhalb aller Funktionen, erfolgt, und damit global sichtbar ist. Eine statische Variable ist nicht unähnlich einer globalen, jedoch mit dem Unterschied, dass das Schlüsselwort „static“ dafür sorgt, dass sie im statischen Block der

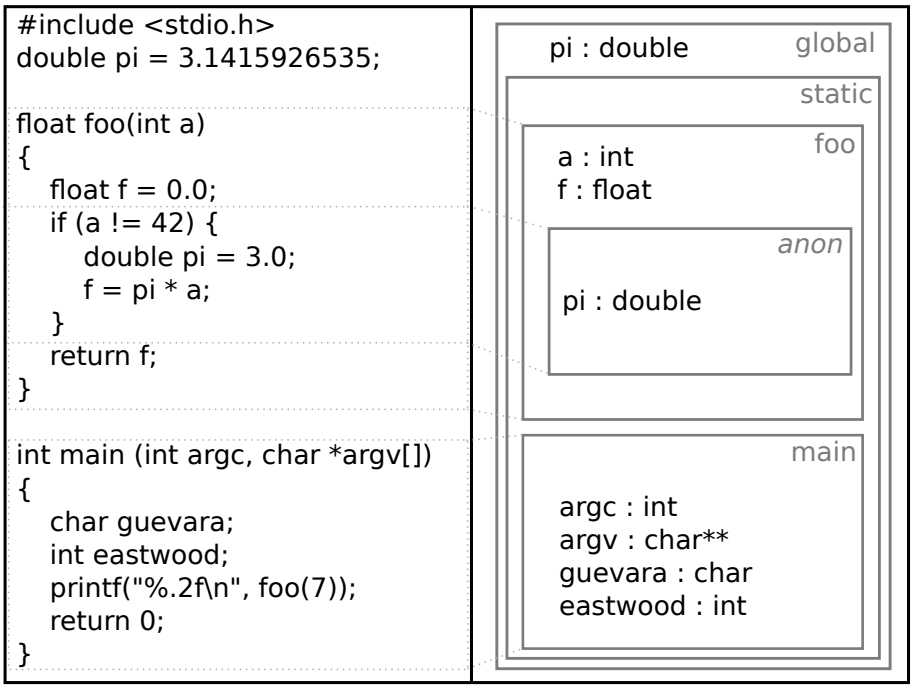


Abbildung 2.3.: Beziehung der Quellcodestruktur zu Blöcken und deren Hierarchie

definierenden Übersetzungseinheit verortet wird, und damit ausschließlich innerhalb dieser sichtbar ist. Lokale Variablen von Funktionen und anonymen Blöcken verhalten sich prinzipiell ähnlich, weisen jedoch die Besonderheit auf, dass sie als Instanzen auf dem Stack liegen; jede dieser Instanzen hat einen eigenständigen, von den anderen Instanzen unabhängigen *Scope*. Variablen aus fremden – d.h. nicht zur ausgeführten Instruktion gehörigen – Block-Hierarchien, beispielsweise aus anderen Übersetzungseinheiten oder fremden Funktionen derselben Übersetzungseinheit, sind entsprechend niemals im *Scope* einer Instruktion.

Anhand Abbildung 2.3 kann der Zusammenhang von Block-Hierarchie und Scope leicht verdeutlicht werden: Die lokale Variable „f“ der Funktion „foo“ ist nur im zugehörigen, gleichnamigen Block, sowie dem untergeordneten, anonymen Block der if-Abfrage sichtbar.

2.2.2. Lebensdauer

Ein Speicherbereich ist „lebendig“ wenn er durch eine Variable belegt wird; die *Lebensdauer* einer Variablen meint die Dauer der Existenz einer Variablen in einem bestimmten Speicherbereich. Wird der Speicher einer Variablen freigegeben, etwa

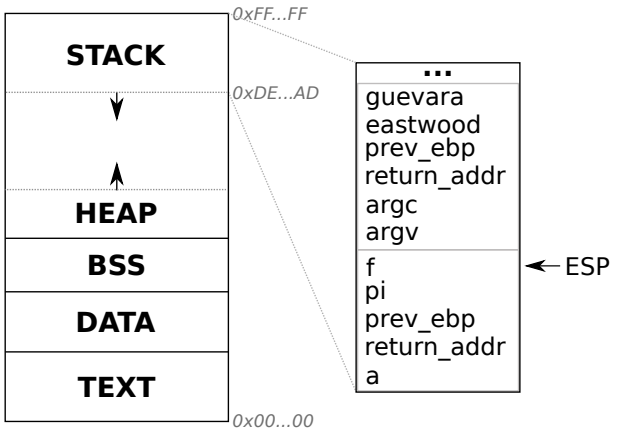


Abbildung 2.4.: Speicherlayout einer von-Neumann-Maschine inkl. Aufbau des Stacks

weil der Stack abgebaut wird, endet die Existenz der Variablen, und damit ihre Lebenszeit.

Globale und statische Variablen liegen während der gesamten Programmlaufzeit ortsfest in den Speicher-Segmenten DATA und BSS, womit ihre Lebensdauer gleich der Programmlaufzeit ist. Lokale Variablen einer Funktion dagegen existieren als Instanzen in einem sogenannten *Stack-Frame* auf dem Stack. *Stack-Frames* dienen der Speicherung des Zustands von Funktionsaufrufen. Aus diesem Grund ist die Lebensdauer lokaler Variablen abhängig von der Existenz-Dauer ihres jeweiligen Stack-Frames.

Eine Bestimmung des Auf- und Abbaus des Stacks, und damit der Lebensdauern und Speicherbereiche lokaler Variablen, ist ohne Laufzeitinformationen nicht möglich. Dies ist eines der Teilprobleme dieser Arbeit.

Daten auf dem Heap werden in dieser Arbeit aufgrund der erschwerten Betrachtung der Inhalte und der zugehörigen Lebensdauern ausgeklammert. Prinzipiell ist eine Untersuchung des Heaps mit den hier vorgestellten Methoden jedoch möglich.

2.3. Debugging

Weist eine Software ein Fehlverhalten auf, ist es vorteilhaft dieses Verhalten näher untersuchen zu können, ohne das Programm modifizieren zu müssen – beispielsweise das Einbauen zusätzlicher Debug-Ausgaben („printf-Debugging“). *Debugger* sind Werkzeuge, die dies ermöglichen, indem sie Mechanismen des Betriebssystems oder der Hardware nutzen, um den Kontrollfluss der untersuchten Software zu steuern und Speicherinhalte zu untersuchen und zu modifizieren.

Debugger, wie der GDB [SS96], stützen sich auf sogenannte *Debug-Informationen*, welche beispielsweise im *DWARF*-Standard [10] spezifiziert sind.

Diese werden für das bloße Ausführen der Software nicht benötigt, und daher üblicherweise im Kompilierungsvorgang nicht in das ausführbare Programm eingefügt. Wenn das Programm jedoch mit einem Debugger untersuchbar sein soll, muss ein entsprechendes Compilerflag verwandt werden, welches den Compiler veranlasst die Debug-Informationen anzufügen. Für Compiler aus der *GNU Compiler Collection* [Sta88] ist das entsprechende Flag „-g“.

Durch Nutzung der Debug-Informationen können Debugger unter anderem das sogenannte *data-value-problem* [TG00] [Böc11] lösen, und somit die aktuelle Speicheradresse einer Variablen und anhand dieser den Inhalt der Variablen bestimmen. Die Fähigkeit zur Bestimmung des, durch eine bestimmte Variable belegten, Speicherbereichs ist essentiell für diese Arbeit.

Da diese Arbeit Gebrauch von den Debug-Informationen macht, ist es notwendig die zu untersuchende Software entsprechend zu kompilieren.

2.3.1. Kontrollfluss

Zur Kontrollflusssteuerung stellen Debugger verschiedene Mechanismen bereit. Für diese Arbeit sind insbesondere das sogenannte *Single-Stepping* und *Breakpoints* relevant:

- *Single-Stepping* ermöglicht es dem Nutzer den Kontrollfluss des Programms instruktions- oder zeilenweise zu durchlaufen. Auf diese Weise ist es möglich sämtliche Vorgänge – anhand des Programmzustands – zu untersuchen, und auf diese Weise Abweichungen vom regulären Ablauf festzustellen.

- *Breakpoints* erlauben es dem Nutzer die Programmausführung bei einer bestimmten Assembler-Instruktion oder Quelltext-Zeile, wie beispielsweise dem Aufruf einer zu untersuchenden Funktion, anzuhalten. So wird ein zeitaufwändiges *Single-Stepping* zu diesem Zeitpunkt vermieden.

2.4. FAIL*

FAIL* – „FAult Injection Leveraged“ – ist ein Framework zur Untersuchung der Fehlertoleranz von Software, welches am diese Arbeit betreuenden Lehrstuhl von Horst Schirmeier entwickelt wurde.

Mit FAIL* ist es möglich Simulator-basiert [BP03] Fehler auf Ebene der Systemarchitektur – CPU-Register, Bus-Systeme, Hauptspeicher, ... – zu injizieren,

und deren Effekt auf eine laufende Software zu untersuchen. So ist es möglich quantitative Aussagen über die Fehlertoleranz einer Software zu treffen.

FAIL* stellt mit Bochs [Law96], Gem5 [Bin+11], OpenOCD [Rat05] und Qemu [Bel05] mehrere, austauschbare *Backends* zur Verfügung, auf denen eine Ziel-Software ausgeführt werden kann. Der „*“ im Projektnamen repräsentiert ebendiese Variabilität bzgl. der Backends.

Da eine detaillierte und tiefgehende Beschreibung von FAIL* den Umfang dieser Arbeit sprengen würde, sei an dieser Stelle für mehr Informationen auf das zugehörige Tool-Paper [Sch+15] verwiesen. Hier wird lediglich der grundsätzliche Ablauf der Untersuchung der Fehlertoleranz einer Ziel-Software näher betrachtet.

Die in dieser Arbeit entwickelten Methoden und Werkzeuge bewegen sich im Bereich der Post-mortem-Analyse.

Ablauf

Die Untersuchung der Fehlertoleranz einer Software mittels FAIL*, visualisiert in Abbildung 2.5, ist grundsätzlich ein vierschrittiger Vorgang [Sch+15]:

1. *Definition des FI-Experiments:* Ein Fehlerinjektionsexperiment muss unter Anderem anhand des zugrunde gelegten Fehlermodells angefertigt oder modifiziert werden. Diese Arbeit nutzt das *ecos_kernel_test*-Experiment, welches auch von Borchert et al. [BSS16] und Hellwig [Hel14] verwandt wurde.

2. *Initiale Analyse:* Ein *golden-run* der untersuchten Software wird durchgeführt. Anhand dessen werden unter Anderem Traces des Instruktionsstroms und sämtlicher Speicherzugriffe aufgezeichnet. Aus den erhobenen Informationen ergibt sich eine Partitionierung des Fehlerraums, aus welcher die zu tätigenden Fehlerinjektionsexperimente errechnet werden.

3. *Fehlerinjektionskampagne:* In der Fehlerinjektionskampagne – ein zentraler Server kann mehrere, fehlerinjizierende Clients anleiten – werden die zuvor bestimmten Fehlerinjektionsexperimente durchgeführt und das emergente Verhalten der Software klassifiziert. Diese Fehlerinjektionsergebnisse werden an den Server zurückgeleitet und in der Datenbank gespeichert.

4. *Post-mortem-Analyse:* Aus den, im ersten Analyseschritt und durch Fehlerinjektion generierten, Daten wird Wissen um die Fehlertoleranz der Software extrahiert. Dabei kann es sich um rohe Ergebniszahlen, Fehlerraumplots oder Zuordnungen der Ergebnisse zu Code-Zeilen oder globalen Datenstrukturen handeln. Anhand dieses Wissens kann der Nutzer die Software gezielt gegen das gewählte Fehlermodell härten.

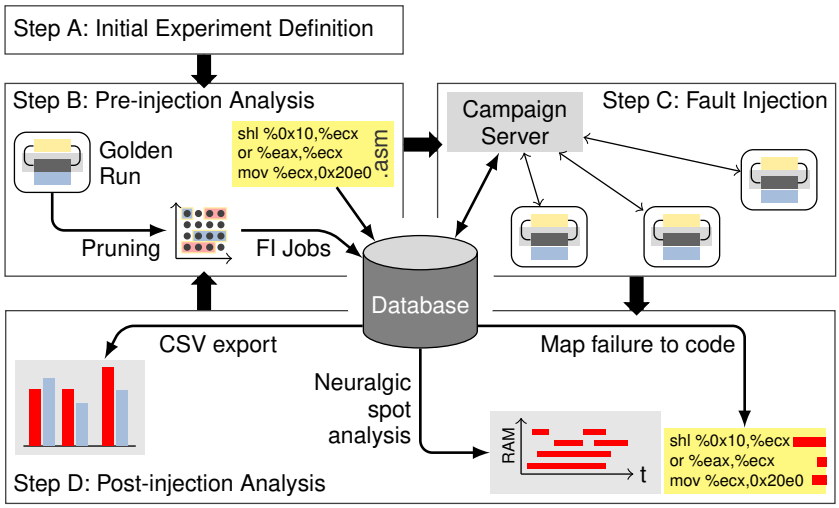


Abbildung 2.5.: Untersuchungsablauf mit FAIL*, entnommen aus [Sch+15]

2.5. Verwandte Arbeiten

Dieser Abschnitt betrachtet themenverwandte Arbeiten, und ordnet diese Arbeit dadurch in den wissenschaftlichen Kontext ein.

Mit FAIL* [Sch+15], [Sch16] besteht zum gegenwärtigen Zeitpunkt die Möglichkeit anteilige Fehlerinjektionsergebnisse für Symbole und insbesondere auch Übersetzungseinheiten und Quellcode-Zeilen der Ziel-Software zu bestimmen. Für eine Bestimmung der anteiligen Fehleranfälligkeit einer Code-Zeile sind eine Nutzung der sogenannten „Rückabbildungsinformationen“ sowie ein vollständiger Instruktionstrace notwendig. Bei Rückabbildungsinformationen handelt es sich um die Rückabbildung von Maschinen-Instruktionen auf hochsprachliche Code-Zeilen; dies ist das sogenannte *code-location-problem*, welches Adrian Böckenkamp in seiner Bachelorarbeit [Böc11] untersuchte. Diese Bachelorarbeit baut auf den durch diese Arbeiten gewonnenen Erkenntnissen und geschaffenen Grundlagen auf.

Als konkreter Anwendungsfall für eine Abbilung von Fehlerinjektionsergebnissen auf Symbole, sei hier die Entwicklung und Untersuchung des Fehlertoleranzverfahrens „*Generic Object Protection*“ angeführt. Hierbei mussten Borchert et al. [BSS16] die Objekte der geschützten Klassen als globale Variablen anlegen, um anteilige FI-Ergebnisse bestimmen und somit eine Aussage über die Auswirkungen der Fehlertoleranzmaßnahme treffen zu können. Die in dieser Arbeit entwickelten Methoden vereinfachen zukünftig derartige Auswertungen und entfernen die Notwendigkeit der Verwendung globaler Variablen.

In seiner Masterarbeit [Hel14] nutzte Richard Hellwig die Libdwarf [14] zum Zugriff auf die Debug-Informationen der untersuchten Software, um Fehlerinjektionsergebnisse auf Quellcode-Zeilen und Datenstrukturen abzubilden. Für nicht optimierten Code erzielte er damit vielversprechende Ergebnisse; unter Compiler-Optimierungen zeigte sein Ansatz jedoch deutliche Schwächen. Diese Arbeit, und insbesondere ihre Evaluation der Zuordnungsgüte unter Optimierung, kann als Vergleichsmaßstab des hier entwickelten Ansatzes dienen.

Abgesehen von der Arbeit von Hellwig [Hel14] konnte keine Literatur gefunden werden, welche die Debug-Informationen der Ziel-Software zur feingranularen Zuordnung von Fehlerinjektionsergebnissen verwendet, obwohl beispielsweise Rademacher [Rad13] sowie Portela-Garcia et al. [Por+11] Fehlerinjektion mittels *On-Chip-Debugger* thematisieren.

2.6. Zusammenfassung

Dieses Kapitel hat die theoretischen und praktischen Grundlagen für diese Arbeit eingeführt. Zentral waren hierbei die Abgrenzung der verschiedenen Bedeutungen des Fehlerbegriffs voneinander, und die Erläuterung ihres Zusammenhangs, sowie natürliche Ursachen für ein Auftreten transienter Hardwarefehler. Auf diesem Fundament aufbauend wurden der Begriff der Fehlertoleranz charakterisiert und mit Fehlerinjektion ein experimentelles Werkzeug zur Überprüfung der Fehlertoleranz einer Software eingeführt. Weiterhin wurden mit einem kurzen Exkurs zum Thema Debugging und Debug-Informationen sowie einer Beschreibung der Fehlertoleranz-Analyse mittels des Frameworks FAIL* die technischen Grundlagen dieser Arbeit beschrieben. Anhand themenverwandter Arbeiten wurde im Anschluss daran diese Arbeit in ihren wissenschaftlichen Kontext eingeordnet. [

3. Analyse und Entwurf

Dieses Kapitel analysiert in den folgenden Abschnitten konzeptionelle und technische Kernaspekte dieser Arbeit, und entwickelt daraus konkrete Lösungsansätze. Diese werden im in dieser Arbeit entwickelten Tool *VProf*[1] zusammengeführt – siehe Abschnitt 3.2.

Wenn nach erfolgter Fehlerinjektionskampagne die Ergebnisse für den Fehlerraum der Software vorliegen, kann anhand der rohen Ergebniszahlen bereits eine grobe Aussage über das Verhalten der Software unter Fehler beziehungsweise über ihre Fehlertoleranz getroffen werden.

Da der Einsatz von Fehlertoleranzmaßnahmen dosiert erfolgen muss – vgl. Abschnitt 2.1.3 – bedarf es zur optimalen Platzierung jedoch feingranularerer Kennzahlen. Grundsätzlich ist eine Einschränkung des Fehlerraums, durch vergleichsweise einfache Datenbankabfragen, in Zeit und Raum möglich. Dies ist allerdings ein Prozess, welcher viel manuelles Eingreifen erfordert und auf die untersuchten Programmstrukturen und verwendeten Fehlertoleranzmechanismen zugeschnitten werden muss.

Aufgrund der für Untersuchtung lokaler Variablen notwendigen Laufzeitinformationen[2] konnten Fehlerinjektionsergebnisse bislang nur, unter Nutzung der Informationen aus der Symboltabelle des Binaries, auf globale und statische Variablen umgelegt werden.

3.1. Zuordnung von FI-Ergebnissen

Zur Zuordnung von Fehlerinjektionsergebnissen zu insbesondere dynamischen Datenstrukturen, ist es erforderlich zu wissen, welche Speicherbereiche in welchen Zeitintervallen von einer Variablen bzw. ihren Instanzen belegt werden.

Ein solches Adress-Zeit-Intervall für eine bestimmte Variable wird im Rahmen dieser Arbeit „Beobachtung“ genannt. Der Raum aller Beobachtungen ist der „Beobachtungsraum“, welcher sich, aufgrund identischer Koordinatensysteme, mit dem Fehlerraum überlagern lässt. Auf diese Weise ermöglichen Fehler- und Beobachtungsraum die Bestimmung der anteiligen Fehlerinjektionsergebnisse einer jeden

[1] Kurzform für „Variable Profiler“
[2] vgl. Kapitel 2.2.2

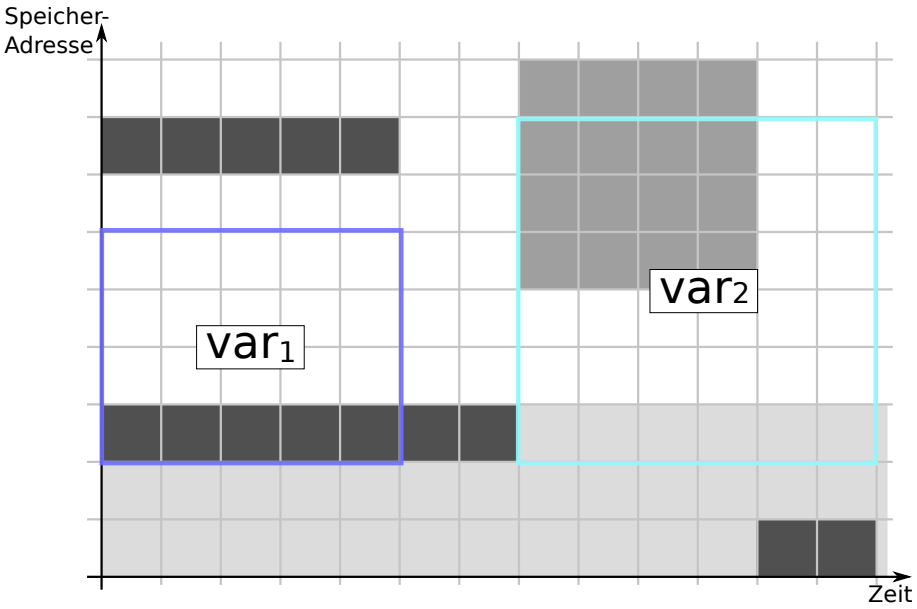


Abbildung 3.1.: Fehlerraumplot mit überlagertem Beobachtungsraum.

Beobachtung. Abbildung 3.1 visualisiert dies anhand der Kombination eines Fehlerraumplots mit den Informationen des Beobachtungsraums. Wie unmittelbar deutlich wird, handelt es sich bei den anteiligen FI-Ergebnissen, um die von einer Beobachtung überlagerten Anteile des Fehlerraums. Mit Hilfe der in Abbildung 2.2 annotierten Verhaltensklassen, kann in diesem Beispiel festgestellt werden, dass 15 Ergebnisse der Klasse *OK* und 5 Ergebnisse der Klasse *SDC* auf die Datenstruktur var_1 entfallen. Für var_2 sind es entsprechend 18x *OK*, 6x *TIMEOUT* und 12x *TRAP*.

In den folgenden Abschnitten werden die Zeit- und Raum-Komponente von Beobachtungen betrachtet, ein Ansatz entwickelt sie zu erheben, und beschrieben wie die anteiligen FI-Ergebnisse für Beobachtungen ermittelt werden können.

3.1.1. Speicherbereiche

Die Bestimmung des Speicherbereichs einer Variablen ist Teil des *data-value-problem* – vgl. Abschnitt 2.3 –, und benötigt, zumindest für lokale Variablen, Laufzeitinformationen. Zur Programmlaufzeit ist es jedoch, beispielsweise mittels eines Debuggers, möglich die Speicheradresse und Größe einer Variablen zu bestimmen. Hierbei ist problematisch, dass gleichnamige Variablen unter Umständen verschattet werden, und nicht unmittelbar sichtbar sind. Weitere Probleme stellen lokale

Variablen anderer *Stack-Frames* und die Existenz mehrerer Stacks in mehrfädigen Programmen dar. Diese Probleme werden im Folgenden betrachtet.

Verschattung

Eine Variable aus einem Code-Block wird „verschattet“, sofern untergeordnete Code-Blöcke – vgl. Abschnitt 2.2 – eine gleichnamige Variable definieren. Die verschattete Variable ist damit für Instruktionen der untergeordneten Blöcke nicht mehr sichtbar, bleibt jedoch lebendig.

In Abbildung 2.3 wird beispielsweise die globale Variable *pi* des Typs *double* von der gleichnamigen Variable aus dem Block der if-Abfrage der Funktion *foo* verschattet. Für eine Instruktion aus dem Block der if-Abfrage kann die Existenz der verschatteten Variablen festgestellt werden, indem die zu ihr gehörige Block-Hierarchie aufwärts, zum globalen Block hin, durchlaufen wird. Wird dabei jeder Block auf in ihm definierte Variablen untersucht, ergibt sich eine vollständige Liste aller Variablen, in der betrachteten Hierarchie.

Somit kann selbst für verschattete Variablen der durch sie belegte Speicherbereich bestimmt werden.

Instruktionsfremde Scopes

Zwar kann, wie im vorigen Abschnitt dargelegt, die komplette Block-Hierarchie einer aktuell ausgeführten Instruktion nach Variablen durchsucht werden, jedoch ist es so nicht möglich Variablen aus instruktionsfremden Scopes zu entdecken. Spätestens an dieser Stelle sind Laufzeitinformationen, in Form des konkreten Stack-Aufbaus, notwendig.

Um beispielsweise die lokalen Variablen einer aufrufenden Funktion betrachten zu können, ist es notwendig den zu dieser Funktion gehörigen Stack-Frame zu untersuchen. Da es sich bei einem *Stack-Frame* um einen Mechanismus zur Speicherung des Zustands eines konkreten Funktionsaufrufs handelt, muss die zugehörige Block-Hierarchie bekannt sein. Anhand dieser können alle zugehörigen Variablen, nach dem im vorigen Abschnitt vorgestellten Verfahren, ermittelt werden.

Da der Stack auf- und abgebaut wird, sind insbesondere auch Informationen um die Stack-Frames selbst vorhanden, welche, beispielsweise von einem Debugger, genutzt werden können, um den Stack zu traversieren. Auf diese Weise können alle aktuell im Speicher befindlichen Variablen gefunden werden.

Das Multi-Stack-Problem

Hellwig beschreibt in seiner Masterarbeit [Hel14] das *Multi-Stack-Problem*, welches auch diese Arbeit betrifft: Programme, welche auf mehreren Threads basieren, wie

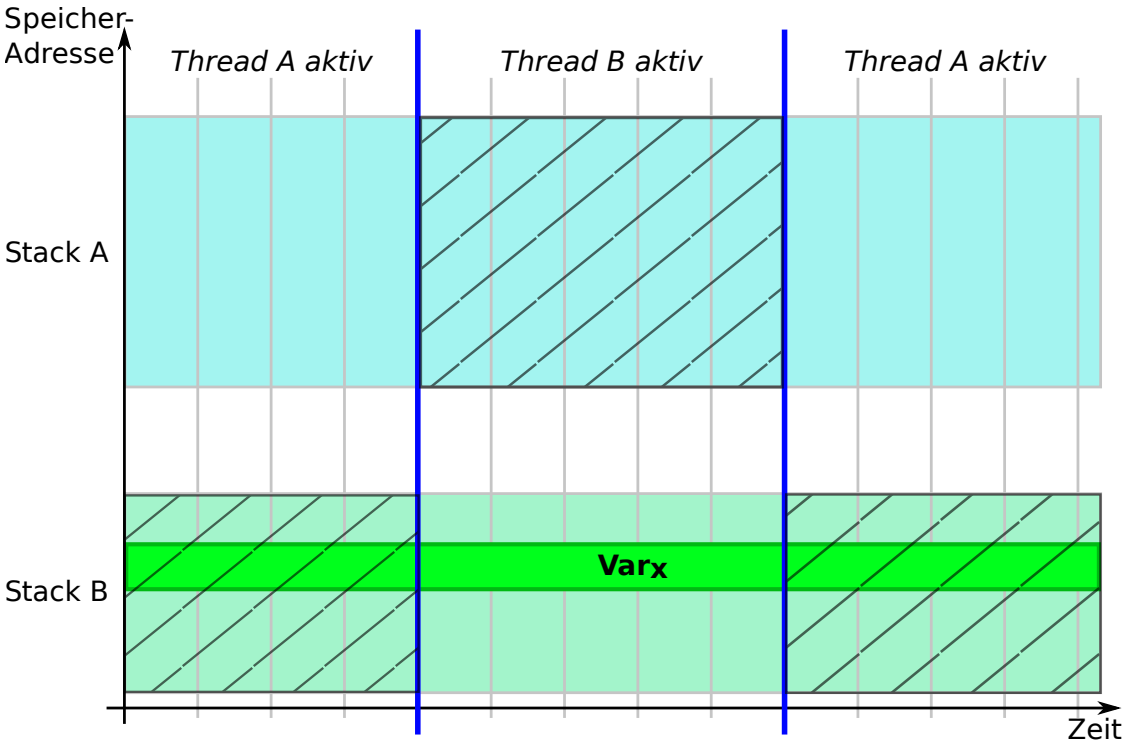


Abbildung 3.2.: Visualisierung des Multi-Stack-Problems.

beispielsweise das hier zur Evaluation verwendete Embedded-Betriebssystem eCos, nutzen für jeden ihrer Threads einen eigenen Stack. Findet ein Wechsel zwischen den Threads statt, wird damit auch der aktive Stack gewechselt. Ohne diesen Stack-Wechsel zu betrachten, und auch inaktive Stacks zu untersuchen, liefert das vorgenannte Verfahren kein vollständiges Bild der Speicherbelegung aller lebendigen Variablen des untersuchten Programms.

Abbildung 3.2 verdeutlicht das Problem: Dargestellt sind die Aktivitätszeiten zweier Threads sowie die zugehörigen Stacks. Für Thread respektive Stack B ist ebenfalls eine Variable Var_x eingezeichnet, welche für die gesamte dargestellte Zeit lebendig ist und ortsfest – beispielsweise im untersten Stack-Frame auf Stack B – im Speicher liegt. Wird ein Thread- bzw. Stack-Wechsel nicht berücksichtigt, und lediglich der aktive Stack betrachtet, werden keine Beobachtungen in den schraffiert dargestellten Flächen erhoben. In Bezug auf Var_x bedeutet dies, dass im dargestellten Ausschnitt lediglich etwa ein Drittel der tatsächlichen Lebenszeit beobachtet wird. Dies bedeutet in Folge, dass für sie nur etwa ein Drittel ihrer anteiligen Fehlerinjektionsergebnisse ermittelt wird.

Die von Hellwig angeführten Verfahren zur Lösung des Problems [Hel14] sind hier prinzipiell anwendbar. Dies wurde aus Zeitgründen jedoch nicht verfolgt.

3.1.2. Lebenszeiten

Zur Bestimmung der Lebenszeiten insbesondere lokaler Variablen sind ebenfalls Laufzeitinformationen notwendig, weil diese eng mit der Bestimmung der belegten Speicherbereiche verknüpft sind. Für eine vollständige und korrekte Bestimmung aller Lebenszeiten, muss für jede ausgeführte Instruktion des Programms bestimmt werden, welche Variablen lebendig sind.

Der intuitive Ansatz hierfür ist daher eine instruktionsweise Ausführung des Ziel-Programms; in den folgenden Abschnitten wird er zu den Verfahren "N-Stepping“ und „Breakpoint-Hopping“ erweitert, sowie das aus ihnen erwachsende Problem der Fragmentierung des Beobachtungsraums betrachtet.

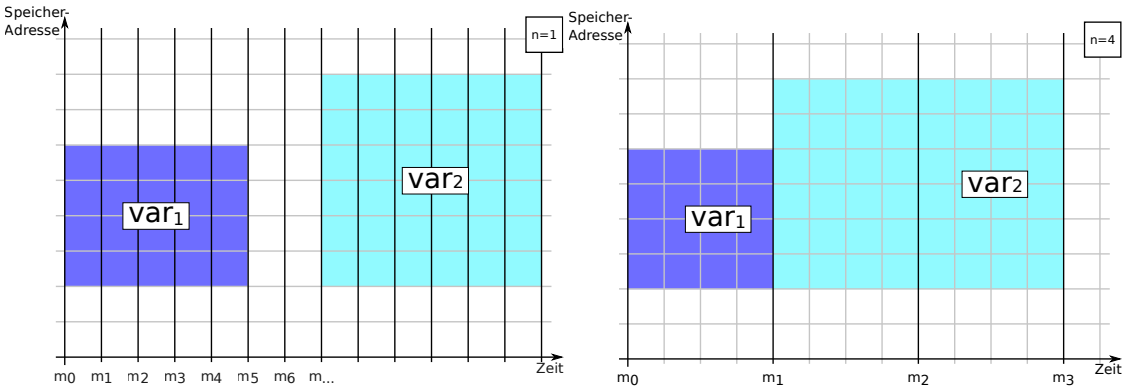
Als durchzuführende Referenzimplementierung wird das N-Stepping gewählt.

N-Stepping

Bei N-Stepping handelt es sich um eine direkte Erweiterung des intuitiven Ansatzes der instruktionsweisen Ausführung der Ziel-Software. Das „N“ steht dabei für eine spezifizierbare Schrittweite, mit welcher die Software durchlaufen wird, und bestimmt in Folge die Frequenz mit der Beobachtungen aller Variablen vorgenommen werden.

Für N=1 handelt es sich damit um den eingangs beschriebenen Ansatz. Durch größere Schrittweiten ist, unter Verzicht auf Korrektheit und Vollständigkeit, die Geschwindigkeit der Erhebung des Beobachtungsraums steigerbar. Dies ist darin begründet, dass so insgesamt weniger Beobachtungen erhoben werden müssen, und es sich bei dem Beobachtungsvorgang um den größten Kostenfaktor – Laufzeit – handelt. Analog zum Shannon'schen Abtasttheorem[Sha49] tritt für größere Schrittweiten jedoch unter Umständen für lokale Variablen *Aliasing* auf: Je nach Lage des Endes eines Lebenszeit-Intervalls zum Beobachtungszeitpunkt, ist es möglich, dass dieses Intervall zu groß oder zu klein bemessen wird, oder gar gänzlich verschwindet. In Folge werden dadurch unter Umständen Fehlerinjektionsergebnisse nicht oder nicht korrekt auf die Datenstrukturen des Programms abgebildet.

Abbildung 3.3 stellt den Aliasing-Effekt einer Schrittweite > 1 auf die erhobenen Beobachtungen im Vergleich zur korrekten Erhebung dar. Die Achsen entsprechen denen eines Fehlerraumplots, Beobachtungszeitpunkte sind mit m_x beschriftet. Entgegen der korrekten Erhebung mit Schrittweite 1, wird bei einer Schrittweite von 4 Instruktionen var_1 um 4 Einheiten zu klein und var_2 um 12 Einheiten zu groß bemessen.



(a) Schrittweite=1, Realität **(b)** Schrittweite=4

Abbildung 3.3.: Effekt unterschiedlicher Schrittweiten auf den Beobachtungsraum.

Anhand der Schrittweite ist somit ein Kompromiss zwischen Geschwindigkeit und Genauigkeit möglich. Eine Messung des Geschwindigkeitsgewinns, und ein Nachweis des Aliasing-Effekts findet sich in Abschnitt 5.2.2.

Breakpoint-Hopping

Eine weitere Beschleunigung des N-Stepping-Verfahrens lässt sich durch „Breakpoint-Hopping“ erzielen. Hierbei werden im Vorfeld der Erhebung des Beobachtungsraums Breakpoints über die Ziel-Software verteilt, beispielsweise auf alle Funktionsaufrufe. Sobald ein Breakpoint erreicht wird, wird eine Beobachtung aller zu diesem Zeitpunkt existenten Variablen vorgenommen. Die Ermittlung des Beobachtungsintervalls gestaltet sich bei diesem Ansatz jedoch schwieriger als beim N-Stepping, was an der nicht vorhersagbaren Anzahl ausgeführter Instruktionen zwischen zwei Breakpoints liegt.

Eine Lösung dieses Problems ist, unter Nutzung des GDB, über den von FAIL* erhobenen Instruktionstrace möglich: Der GDB hält für jeden Breakpoint einen sogenannten „Hit-Count“ vor, welcher zählt, wie oft dieser Breakpoint bis zum aktuellen Zeitpunkt ausgelöst wurde. In Kombination mit der Instruktionsadresse des Breakpoints kann so der Zeitpunkt der Beobachtung eindeutig bestimmt werden; der Zeitpunkt ist gleich dem x-ten – x = Hit Count – Auftreten der Breakpoint-Instruktion im Instruktionsstrom. Das beobachtete Zeit-Intervall ergibt sich in Folge mit dem vorherigen Auslösen eines Breakpoints. Dieser Ansatz wird nicht implementiert.

Aus der Verwandtschaft zum N-Stepping folgt, dass dieser Ansatz ebenfalls für Aliasing anfällig ist.

Fragmentierung des Beobachtungsraums

Beiden Ansätzen ist gemein, dass sie möglicherweise mehrere, in Raum oder Zeit zusammenhängende, Beobachtungen für eine Variable liefern, und so den Beobachtungsraum fragmentieren. Dieser Effekt tritt besonders prominent bei N-Stepping mit einer Schrittweite von einer Instruktion für globale Variablen zu Tage: Obwohl eine einzige Beobachtung mit Zeit-Intervall $[0, t_{Programm-Ende}]$ ausreichend wäre, werden viele Beobachtungen mit Dauer 1 erhoben. Es ist vor dem Hintergrund der weiteren Verarbeitung – insbesondere der Bestimmung der anteiligen FI-Ergebnisse – angeraten „Pruning“ zu betreiben, d.h. in Raum und oder Zeit zusammenhänge Beobachtungen einer Variablen zusammenzufassen.

3.1.3. Anteilige FI-Ergebnisse

Zur Ermittlung der anteiligen Fehlerinjektionsergebnisse aller Datenstrukturen einer Ziel-Software, muss der Fehlerraum auf die erhobenen Beobachtungen eingeschränkt werden. Eine räumliche Einschränkung auf einen bestimmten Speicherbereich ist beispielsweise für die Ermittlung der Fehleranfälligkeit globaler und statischer Variablen notwendig und wird unter anderem von Borchert et al. [BSS16] vorgenommen. Eine Einschränkung in der Zeit ist ebenfalls möglich, und bedarf lediglich kleiner Modifikationen an der von ihnen verwandten SQL-Abfrage.

3.2. VProf

Die zuvor beschriebenen Methoden zur Erhebung des Beobachtungsraums einer Ziel-Software, sowie der anteiligen Fehlerinjektionsergebnisse, werden im Tool *VProf* zusammengeführt, welches den letzten Analyseschritt von FAIL* optional erweitert.

Unter Nutzung eines Debuggers steuert VProf den Kontrollfluss eines *golden-run* der Software, erhebt – im Rahmen dieser Arbeit mittels N-Stepping – den Beobachtungsraum, und ermittelt die anteiligen Fehlerinjektionsergebnisse der Variablen. Um die Auswertung der erhobenen Daten zu vereinfachen und eine Kombination mit anderern über die Ziel-Softwre ermittelten Daten zu ermöglichen, werden die von VProf erhobenen Daten in der Datenbank von FAIL* abgelegt.

Die nachfolgenden Abschnitte thematisieren die Architektur von VProf, die Nutzung eines Debuggers zum Zugriff auf die Debug-Informationen und zur Steuerung eines *golden-run*, sowie das entworfene Datenbanklayout.

3.2.1. Architektur

Die Architektur von VProf wird nachfolgend anhand des zugehörigen Klassendiagramms – dargestellt in Abbildung 3.4 – betrachtet. In den weiteren Unterabschnitten wird die Interaktion mit Debugger und Datenbank beschrieben.

Die Funktionalität der einzelnen Klassen wird nun im Einzelnen beschrieben:

- *VProf*: Dies ist die namensgebende Komponente des entworfenen Werkzeugs, und fasst alle notwendigen Mechanismen zusammen. Mittels eines *Observers* wird der Beobachtungsraum erhoben. Für jede erhobene Beobachtung einer Variablen werden ihre anteiligen FI-Ergebnisse ermittelt. Variablen, Beobachtungen und anteilige Ergebnisse werden in der FAIL*-Datenbank – siehe Abschnitt 3.2.3 – gespeichert.

- *Observer*: Diese abstrakte Klasse stellt eine einheitliche Schnittstelle zur Nutzung konkreter, abgeleiteter Observer, wie dem *StepObserver*, dar. Ein konkreter Observer implementiert einen Mechanismus zur Erhebung des Beobachtungsraums der Ziel-Software; im Falle des StepObserver ist dies das N-Stepping[3].

- *Variable*: Diese Klasse ist ein Container für die Kenndaten einer Variablen der Ziel-Software. Eine Variable ist durch ihren Namen und Typ nicht eindeutig identifiziert – vgl. „Verschattung“ in Abschnitt 3.1.1. Werden jedoch noch die zugehörige Quellcode-Datei und die Zeile ihrer Definition hinzugenommen, ist eine Variable eindeutig identifizierbar.

- *Observation*: Diese Klasse repräsentiert eine konkrete Beobachtung einer Variablen, also die Start- und Endzeit, sowie die Basisadresse und Größe des durch sie belegten Speicherbereichs.

- *GDB*: Diese Klasse ist die Kommunikationsschnittstelle zu einem GDB-Prozess, und wird in Abschnitt 3.2.2 eingehender beschrieben. Der GDB-Schnittstelle ist die Klasse *Breakpoint* beigeordnet, welche lediglich ein Container für relevante Informationen zu Breakpoints ist.

3.2.2. Debugger

Aufgrund der von Hellwig [Hel14] gemachten Erfahrungen – vgl. Abschnitt 2.5 – mit direkter Nutzung der Libdwarf [14] in Kombination mit Compiler-Optimierungen, verwendet diese Arbeit einen Debugger zum Zugriff auf die Debug-Informationen

[3] N-Stepping wird in Abschnitt 3.1.2 eingeführt

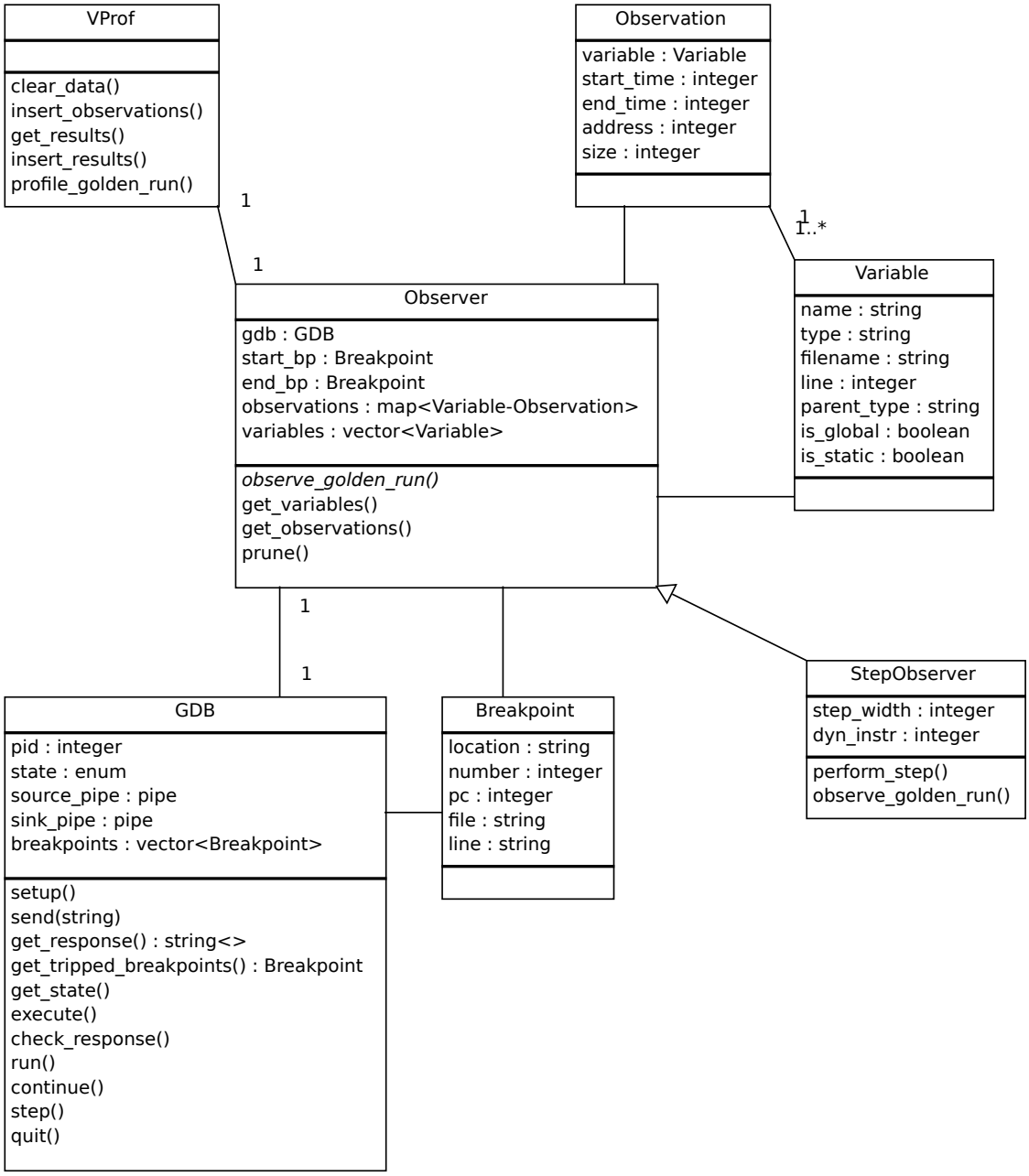


Abbildung 3.4.: Klassendiagramm von VProf.

der Ziel-Software. Durch Nutzung eines Debuggers ist es darüber hinaus möglich den Kontrollfluss der Software zu steuern, und selbst bei Compiler-Optimierung die von Variablen belegten Speicherbeiche zu bestimmen.

Vor dem Hintergrund, dass alle FAIL*-Backends[4] eine Schnittstelle für den GDB bieten, und dieser alle notwendigen Fähigkeiten vorweist, verwendet diese Arbeit den GDB [SS96] als Debugger.

Zur Nutzung des GDB aus VProf heraus, muss eine bidirektionale Kommunikation zwischen VProf-Prozess und GDB-Prozess möglich sein. Eine derartige Kommunikationsschnittstelle wird „Front End“ [Soua] genannt, der nachfolgende Unterabschnitt befasst sich damit.

GDB Front End

Mit der *Libgdb* [Pro] und der *libGDB* [Soue] existieren beziehungsweise existierten zwei Bibliotheken zur Nutzung des GDB; beide werden jedoch seit Ende der 1990er Jahre nicht mehr gepflegt, und waren zu Beginn dieser Arbeit nicht nutzbar.

Darüber hinaus existiert das *GDB machine interface* [Soub], welches die Definition eines maschinenlesbaren Ein- und Ausgabeformats zur Kommunikation mit dem GDB darstellt. Mit der *libmigdb*[5] ist zwar eine Bibliothek – für die Sprachen C und C++ – dafür verfügbar, jedoch war diese zu Beginn dieser Arbeit seit 6 Jahren ungepflegt, und ebenfalls nicht nutzbar.

Vor diesem Hintergrund wurde die Entscheidung getroffen eine Schnittstelle zu implementieren, welche eine textbasierte Kommunikation, analog einer interaktiven Sitzung, mit einer GDB-Instanz ermöglicht. Sofern sich in einer zukünftigen Version des GDB das Ausgabeformat von Kommandos ändert, bedarf es daher Anpassungen an Code, welcher ebendiese Kommandos nutzt. Um den eventuellen Anpassungsaufwand minimal zu halten, werden in dieser Arbeit reguläre Ausdrücke zum Auswerten der Ausgaben des GDB verwandt.

Diese GDB-Schnittstelle wird in der Klasse *GDB* – siehe Abbildung 3.4 – gekapselt. Bei Erzeugung eines GDB-Objekts werden alle notwendigen Initialisierungsschritte, inklusive der Erzeugung eines GDB-Prozesses und dem Einrichten der bidirektionalen Kommunikation, automatisch ausgeführt.

Aufgrund der quasi-interaktiven Nutzung des GDB, können die Standard-Kanäle für Ein- und Ausgabe – *stdin* und *stdout* – des GDB-Prozesses genutzt werden. Auf diese wird mittels zweier *pipes* zugegriffen, von denen jede eine Kommunikationsrichtung ermöglicht: Die „source_pipe“ dient dem Lesen der Ausgaben des GDB auf dem Standard-Ausgabe-Kanal, die “sink_pipe“ entsprechend dem Senden von Eingaben an den GDB über dessen Standard-Eingabe-Kanal.

[4] FAIL*s Backends werden Abschnitt 2.4 angeführt

[5] libmigdb – `https://sourceforge.net/projects/libmigdb/`

3.2.3. Datenbank

Das zur Speicherung der, den Beobachtungsraum ausmachenden, Daten notwendige Datenbankschema ergibt sich aus der Tatsache, dass zu einer Variablen mehrere Beobachtungen existieren können, und für jede Beobachtung ihre anteiligen Fehlerinjektionsergebnisse ermittelt werden.

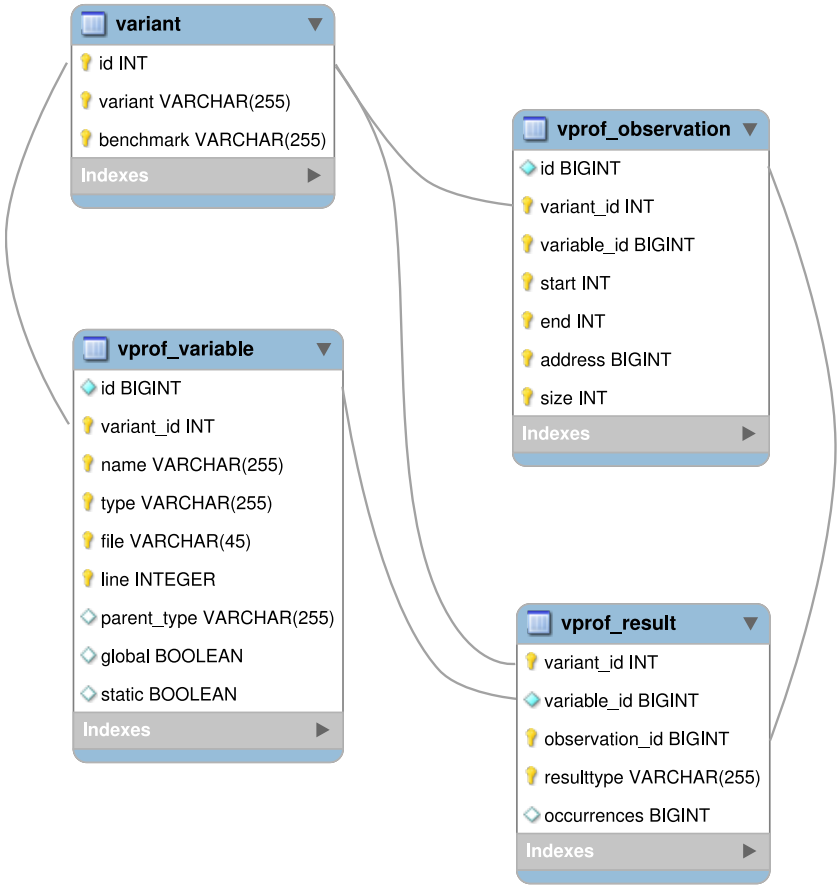


Abbildung 3.5.: Datenbankschema für VProf.

Die für VProf relevanten Tabellen sind in Abbildung 3.5 dargestellt. Die Tabelle *variant* beschreibt eine Ziel-Software und ist bereits Bestandteil der FAIL*-Datenbank, und ihre ID – *variant_id* – dient daher als Identifikator der untersuchten Software. Die *„vprof_“*-Tabellen wurden neu modelliert, und werden folgend beschrieben:

- *vprof_variable*: Diese Tabelle dient der Speicherung aller Variablen der Ziel-Software. Wie eingangs dargelegt, ist eine Variable der Ziel-Software eindeu-

tig durch ihren Namen, Typ, sowie den Ort ihrer Definition – Übersetzungseinheit und -Zeile – identifiziert. Diese Daten können daher als Primärschlüssel verwandt werden; als zusätzlicher Identifikator dient eine eindeutige ID. Zusätzliche Informationen wie ein eventueller Eltern-Datentyp – für Bestandteile komplexer Datenstrukturen – und Flags, ob es sich um eine statische oder globale Variable handelt, werden ebenfalls erhoben und gespeichert.

- *vprof_observation*: Diese Tabelle dient der Speicherung der Beobachtungen von Variablen. Für eine Variable, welche über ihre ID aus *vprof_variable* referenziert wird, wird gespeichert in welchem Zeit-Intervall sie welchen Speicher-Bereich belegt. Dies wird über die Start- und End-Zeit, sowie die Basis-Adresse und Größe der Beobachtung modelliert. Da keine zwei Variablen in exakt demselben Raum-Zeit-Intervall existieren können, handelt es sich bei diesen Informationen um eindeutige Identifikatoren, weshalb sie als Primärschlüssel verwandt werden können. Wie auch für Variablen, wird für Beobachtungen eine eindeutige ID als alternativer Primärschlüssel eingeführt.

- *vprof_result*: Diese Tabelle dient der Speicherung der anteiligen Fehlerinjektionsergebnisse von Beobachtungen. Für jede Verhaltensklasse aus dem anteiligen Fehlerraum der Beobachtung wird ihr Flächenanteil gespeichert. Verhaltensklasse und Beobachtung – referenziert über ihre ID aus der Tabelle *vprof_observation* – bilden damit den Primärschlüssel dieser Tabelle. Um eine Aggregation der Fehlerinjektionsergebnisse nach Variablen, und insbesondere das Filtern nach Informationen von Variablen – wie beispielsweise Datentyp oder Übersetzungseinheit – zu vereinfachen, wird die zur Beobachtung gehörige Variable – referenziert über ihre ID aus der Tabelle *vprof_variable* – als zusätzliches Datum gespeichert.

In Kombination mit bereits von FAIL* erhobenen Informationen, wie dem Instruktionstrace oder den Rückabbildungsinformationen, ermöglicht die entworfene Datenbankstruktur annähernd beliebig komplexe Auswertungen der Fehlertoleranz der Ziel-Software in Bezug auf ihre Datenstrukturen. Beispielsweise kann damit die Auswirkung lokaler Variablen eines bestimmten Datentyps während der Ausführung bestimmter Funktionen oder Instruktionen auf die Fehlertoleranz der Software untersucht werden.

Mit den in der Datenbank gespeicherten Informationen des Beobachtungsraums könnten darüber hinaus Fehlerraumplots, ähnlich zu Abbildung 3.1, annotiert werden. Dies wird im Rahmen dieser Arbeit jedoch nicht verfolgt.

3.3. Zusammenfassung

Dieses Kapitel entwickelte einen grundsätzlichen Lösungsansatz für die Zuordnung von Fehlerinjektionsergebnissen zu dynamischen Datenstrukturen, welcher auf den Debug-Informationen der Ziel-Software und Laufzeit-Informationen des *golden-run* basiert. Auf diese Weise werden Lebenszeit-Intervalle und zugehörige Speicherbelegungen aller Variablen der Ziel-Software erhoben. Die Informationen dieser einzelnen Beobachtungen können in Folge genutzt werden, um den Fehlerraum der untersuchten Software einzuschränken, und so beobachtungsanteilige Fehlerinjektionsergebnisse zu ermitteln. Dieser Lösungsansatz wird durch die entworfene Software *VProf* umgesetzt, welche den GDB zur Steuerung des Kontrollflusses und Untersuchung des Speicherinhalts eines *golden-run* der Software verwendet. Die Implementierung der GDB-Schnittstelle, der Erweiterungen des GDB und insbesondere die Implementierung von VProf selbst werden im nachfolgenden Kapitel beschrieben.

4. Implementierung

Dieses Kapitel beschreibt die Implementierung von VProf, und damit die Umsetzung der im vorigen Kapitel entworfenen Methoden zur Erhebung des Beobachtungsraums und der anteiligen Fehlerinjektionsergebnisse.

Zunächst werden in Abschnitt 4.1 notwendige Arbeiten an existenten Komponenten von FAIL* beschrieben. Im Anschluss daran wird in Abschnitt 4.2 die Implementierung der GDB-Schnittstelle thematisiert. Abschnitt 4.3 bildet, mit der Beschreibung der Implementierung von VProf, den Abschluss dieses Kapitels.

Im Zuge der Implementierung traten diverse Probleme auf, welche an geeigneten Stellen beschrieben werden.

Die Implementierung von VProf und der GDB-Schnittstelle umfasst etwa 1.500 Zeilen C++-Code, die implementierten Erweiterungen des GDB – vgl. Abschnitt 4.2 – umfassen etwa 250 Zeilen Python-Code.

4.1. FAIL*

Zu den durchgeführten, notwendigen Arbeiten an existierenden Komponenten von FAIL* zählen die Erweiterung der Datenbank um die in Abschnitt 3.2.3 modellierten Tabellen, sowie Modifikationen am in dieser Arbeit verwendeten FAIL*-Backend und an einem Daten-Import-Werkzeug. Diese werden in den folgenden Unterabschnitten separat betrachtet.

4.1.1. Modifikationen der Datenbank

Die entworfenen Tabellen – vprof_variable, vprof_observation und vprof_result – wurden implementiert, der verwendete SQL-Code befindet sich als Listing A.1 im Anhang dieser Arbeit.

Der Entwurf sieht eine automatische Wahrung der referenziellen Integrität der Daten mittels der Referenzierung von Fremdschlüsseln und Kaskadierung von Löschbefehlen vor. Dies wird jedoch nur von MySQLs Datenbank-Engine „InnoDB“ unterstützt, nicht jedoch von „MyISAM“, welches in der FAIL*-Datenbank verwandt wird. Aus diesem Grund muss die Datenintegrität manuell gewahrt werden.

Eine weitere Modifikation an der Datenbank war bezüglich des Instruktionstraces notwendig; sie wird in Abschnitt 4.1.3 beschrieben.

4.1.2. Backend

Diese Arbeit beschränkt sich auf das Backend *FAILBochs* – eine Erweiterung des Simulators Bochs [Law96] – , da es das am weitesten entwickelte ist; der Ansatz zur Zuordnung von Fehlerinjektionsergebnissen auf die Datenstrukturen einer untersuchten Software ist jedoch auf die übrigen Backends – siehe hierzu Abschnitt 2.4 – übertragbar.

Damit ein Zugriff mit dem GDB auf Bochs resp. FAILBochs möglich ist, muss dessen sogenannter *GDB Stub*, eine minimale Kontroll-Schnittstelle für den GDB, aktiviert werden. Zur Aktivierung dessen ist es notwendig FAILBochs im Übersetzungsvorgang entsprechend zu konfigurieren – Konfigurationsflag: „–enable-gdb-stub“ –, und ihn schließlich in der Konfigurationsdatei des Backends einzuschalten, wenn er benötigt wird.

Damit der GDB Stub optional aktiviert werden kann, wurde FAIL*s Build-System, CMake, um eine entsprechende Option erweitert.

Alternativer Kontrollfluss mit aktivem GDB Stub

Die Simulation des Gastsystems erfolgt in Bochs in einer kontinuierlich durchlaufenen CPU-Schleife. Um die Ausführungszeit der Fehlerinjektionsexperimenten zu reduzieren, wurde FAILBochs um einen zur Laufzeit nutzbaren Save/Restore-Mechanismus – die Fähigkeiten zum Einfrieren und Laden des Systemzustands – erweitert, welcher in diese Schleife eingreift.

Ist der GDB Stub aktiv, wird diese CPU-Schleife jedoch nicht kontinuierlich durchlaufen, sondern nur in Einzeldurchgängen, welche von einer vorgelagerten Kontrollschleife, die sogenannte „debug_loop“, verwaltet werden. Aufgrund der Implementierung des Laufzeit-Save/Restore-Mechanismus, führt der geänderte Kontrollfluss dazu, dass FAILBochs beziehungsweise dessen GDB Stub nach dem Wiederherstellen des Systemzustands in eine Endlosschleife gelangt. Die Lösung dieses Problems ist eine Anpassung des Kontrollflusses des GDB Stubs analog zu den Modifikationen an der CPU-Schleife.

FAILBochs verfügt über den GDB Stub hinaus noch über einen eigenen Debugger, welcher eine vom GDB Stub verschiedene Kontrollschleife hat. Das im Rahmen dieser Arbeit beobachtete und behobene Problem betrifft mit hoher Wahrscheinlichkeit auch den Bochs-eigenen Debugger, sodass eine entsprechende Modifikation auch dort notwendig wird, wenn dieser genutzt werden soll. Dies wurde jedoch nicht weiter verfolgt.

4.1.3. Import-Werkzeug

Maschineninstruktionen haben nicht notwendigerweise eine Ausführungsdauer von einer Zeiteinheit, sondern unter Umständen eine deutlich höhere, weshalb die tatsächliche Zeitdauer zur Berechnung der Fehlerraumgröße verwandt werden muss. Insbesondere ist diese damit bei der Bestimmung der anteiligen FI-Ergebnisse einer von VProf erhobenen Beobachtung notwendig.

Die x86-Instruktion *hlt* ist ein Beispiel einer solchen Instruktion: Diese Instruktion hält die CPU an und wird beispielsweise in Leerlaufprozessen verwandt, wenn auf externe Ereignisse wie Schreib- oder Lesezugriffe gewartet werden muss. In ebendieser Funktion wird *hlt* im, zur Evaluation verwandten, eCos-Testprogramm *thread1* verwandt, weshalb dies bei der Bestimmung anteiliger FI-Ergebnisse von VProf berücksichtigt werden muss.

Aus diesem Grund mussten das Import-Werkzeug *FulltraceImporter* und die Datenbank-Tabelle *fulltrace* dahingehend erweitert werden, dass sie die Start- und End-Zeit einer ausgeführten Instruktion erheben bzw. speichern. In Folge verfügt der von FAIL* ermittelte, vollständige Instruktionstrace des golden-run nun über die tatsächliche Laufzeit der einzelnen ausgeführten Instruktionen.

4.2. GDB

Dieser Abschnitt beschreibt die Implementierung der entworfenen GDB-Schnittstelle und der, zur Automatisierung der Beobachtung von Variablen dienenden – in Python geschriebenen – Erweiterungen des GDB.

4.2.1. Kommunikation

Wie im Entwurf bereits beschrieben wurde, erfolgt die Kommunikation zwischen GDB-Schnittstelle und GDB-Prozess mittels zweier *pipes*, welche über den Systemaufruf *pipe()* erzeugt werden. Eine *pipe* ist die Bündelung zweier Dateideskriptoren, der erste dient als „Lese-Ende“ und der zweite als „Schreib-Ende“.

Die im Entwurf ebenfalls beschriebene Verwendung dieser *pipes* als Standard-Eingabe respektive Standard-Ausgabe für den GDB-Kindprozess, ist durch Nutzung des Systemaufrufs *dup2()* möglich. Dieser kann Dateideskriptoren duplizieren und überschreiben. Der in Abbildung 4.1 dargestellte Ausschnitt aus dem entsprechenden Quelltext der implementierten GDB-Schnittstelle illustriert dies für den Standard-Ausgabe-Kanal aus Sicht des, mittels *fork()* erzeugten, Kindprozesses: Der Aufruf von *dup2* in Zeile 2 überschreibt den Dateideskriptor des Standard-Ausgabe-Kanals – *STDOUT_FILENO* – mit dem beschreibbaren Dateideskriptor

der im Vorfeld erzeugten *pipe_from_gdb*. Für die andere Kommunikationsrichtung wird analog verfahren.

```
1    // STDOUT is write-end of pipe_from_gdb
2    if (dup2(pipe_from_gdb[1], STDOUT_FILENO) == -1) {
3        std::cerr << "cannot setup pipe from GDB!\n";
4        exit(EXIT_FAILURE);
5    }
```

Abbildung 4.1.: Nutzung des Systemaufrufs *dup2* zur Modifikation des Standard-Ausgabe-Kanals.

Nach Einrichtung der *pipes* sind keine weiteren Anpassungen am Kindprozess notwendig, und dieser kann mittels Systemaufruf aus der *exec*-Familie durch einen GDB-Prozess ausgetauscht werden. Weil dieser die Dateideskriptoren vom verdrängten Prozess erbt, ist mittels der angelegten *pipes* nun eine textbasierte Kommunikation mit dem GDB möglich: Daten die beispielsweise mittels *fputs()* in die „sink_pipe“ der GDB-Schnittstelle geschrieben werden, gehen als Eingabe an den GDB, für welchen sie sich wie direkte, interaktive Nutzer-Eingaben darstellen. Ausgaben die der GDB erzeugt sind entsprechend über die „source_pipe“ lesbar.

Weil es sich aus Sicht des GDB um eine interaktive Sitzung handelt, wird dessen Eingabe-Prompt – *(gdb)* – als Marker herangezogen, welcher die Bereitschaft zum Empfang eines Kommandos bzw. das Ende aller Ausgaben des vorigen Kommandos signalisiert. In Folge ist eine Interaktion wie in einer Interaktiven Sitzung möglich: GDB-Kommandos werden als String gesendet und der GDB-Prozess gibt eine möglicherweise mehrzeilige Antwort zurück.

Die Antwort des GDB wird auf spezielle Strings hin überprüft, welche etwa Zustandstransitionen der Software oder Fehler signalisieren; der folgende Abschnitt elaboriert dies.

4.2.2. Steuerung des Inferior

Eine mittels GDB ausgeführte Software wird als „Inferior“ bezeichnet, und kann die grundlegenden Zustände *Running*, *Stopped* und *Exited* annehmen. Da die Ausführung des Inferior durch den Debugger gesteuert wird – insbesondere auch durch Breakpoints –, muss das Front End eine Abstraktion des Zustands des Inferior haben. Zustandstransitionen des Inferior werden im GDB als „Events“ behandelt. Für jede Klasse von Events, können sogenannte „Event-Handler“ registriert werden, welche anhand der Python-API [Souc] des GDBs implementiert werden können.

Für das entwickelte GDB Front End ist es notwendig auf das Anhalten und Beenden des Inferior zu reagieren, weshalb ein Stop- und ein Exit-Handler implemen-

```
1  #!/usr/bin/python
2  # this is needed to detect the gdbstub halting after a restore
3  # and for breakpoints, of course
4  import gdb
5
6  def stop_handler(event):
7      if (isinstance(event, gdb.SignalEvent)):
8          print("GDB_INFERIOR_STOPPED")
9          print(event.stop_signal)
10
11     elif (isinstance(event, gdb.BreakpointEvent)):
12         print("GDB_INFERIOR_BREAKPOINT")
13         for breakpoint in event.breakpoints:
14             print("BREAKPOINT #%d:%d (%s)" % (breakpoint.number,
                   breakpoint.hit_count, breakpoint.location))
15
16     elif (isinstance(event, gdb.StopEvent)):
17         print("GDB_INFERIOR_STOPPED")
18         print("generic StopEvent")
19
20     else:
21         print("GDB_EXTENSION_ERROR")
22         print(event)
23
24 # register event handlers
25 gdb.events.stop.connect(stop_handler)
```

Abbildung 4.2.: Quellcode des implementierten Stop-Handlers.

tiert wurden. Als Beispiel für die Implementierung ist in Listing 4.2 der Code des Stop-Handlers dargestellt. Wie daraus erkenntlich wird, sind Event-Handler nicht sonderlich komplex: Es handelt sich dabei um Funktionen, die ein Event-Objekt als Parameter übergeben bekommen, und entsprechend des Events reagieren. Der implementierte Stop-Handler reagiert auf Signal-, Stop- und Breakpoint-Events in der Regel mit Ausgabe eines fest definierten Strings. Im Fehlerfall wird von allen implementierten Erweiterungen des GDBs der String „GDB_EXTENSION_ERROR“ ausgegeben. Für den Fall, dass ein Breakpoint ausgelöst wird, werden zusätzlich dessen Informationen ausgegeben; besonders hervorzuheben sind hierbei, auch im Hinblick auf eine Implementierung des Breakpoint-Hopping, die Instruktionsadresse und der Hit-Count des Breakpoints.

Vor dem Hintergrund, dass es bei einem *golden-run* einen betrachteten, interessanten Abschnitt geben kann, kann durch eine derartige Behandlung von Breakpoints gewährleistet werden, dass eine Erhebung des Beobachtungsraums nur im interessanten Abschnitt erfolgt. Im Falle der, zur Evaluation genutzten Kernel-Testprogramme von eCos, liegt dieser interessante Abschnitt zwischen den Aufrufen der Funktionen „cyg_start“ und „cyg_test_exit“.

4.2.3. Erhebung von Beobachtungen

Die in Abschnitt 3.1.1 beschriebene Vorgehensweise zur Beobachtung aller Variablen wird, analog zur Behandlung von Zustandstransitionen des Programms, anhand der Python-API implementiert.

Diese API bietet die Möglichkeit eigene GDB-Kommandos zu implementieren und stellt Abstraktionen für Stack-Frames, Blöcke und Variablen in Form von Objekten zur Verfügung.

Für ein gegebenes *gdb.Frame*-Objekt – die Repräsentation eines Stack-Frames – können über die Methode „block“ der zugehörige Block und über die Methode „older“ der unterliegende Stack-Frame ermittelt werden. Allein anhand des aktuellen Stack-Frames, welcher über einen Aufruf der GDB-Methode „newest_frame“ verfügbar ist, kann somit der gesamte Stack traversiert werden.

Blöcke werden von der Python-API als *gdb.Block*-Objekt modelliert, und stellen über das Attribut „superblock“ – ein gdb.Block – die Möglichkeit zur Verfügung ihre Block-Hierarchie bis hin zum globalen Block zu traversieren. Darüber hinaus liefert ein gdb.Block unter Iteration alle Variablen, in Form eines *gdb.Symbol*-Objekts, zurück, welche im zugehörigen Code-Block definiert wurden. Sofern es sich bei dem Block um den globalen oder einen statischen handelt, handelt es sich bei den Variablen entsprechend um globale oder statische Variablen. Diese Informationen sind über Attribute der Klasse gdb.Block verfügbar.

Für ein *gdb.Symbol*-Objekt sind alle in Abschnitt 3.2.1 beschriebenen, erforderlichen Informationen zu der zugehörigen Variablen, über Attribute des Objekts und Methoden der Python-API, verfügbar.

Diese Möglichkeiten wurden in einem Plugin zur Erhebung des Beobachtungsraums der aktuellen Instruktion zusammengefasst, welches das neue Kommando „observe-variables“ zur Verfügung stellt. Abbildung 4.3 stellt in einem redigierten Auszug des zugehörigen Quellcodes das Traversieren einer Block-Hierarchie inklusive Ermittlung aller in dieser Hierarchie definierten Variablen dar.

Komplexe Datenstrukturen

Für ein gegebenes *gdb.Symbol*-Objekt bzw. dessen Datentyp – *gdb.Type* – kann festgestellt werden, ob es sich um eine komplexe Datenstruktur handelt und aus welchen Komponenten sie zusammengesetzt ist. Bei den Komponenten, welche über ein Attribut des Typ-Objekts zugreifbar sind, handelt es sich jedoch nicht um *gdb.Symbol*-Objekte, sondern lediglich um Attributsname und Datentyp.

Aus diesem Grund ist eine Untersuchung komplexer Datenstrukturen, mit dem derzeitigen Implementierungsstand, lediglich bis zu einer Untersuchungstiefe von

```
1  #!/usr/bin/python
2
3  [...]
4
5    # return *all* symbols in the block hierarchy as a list
6    def get_symbols(self):
7      for symbol in self.block:
8        # skip non-args and non-variables
9
10       # get the symbol's name, address and size
11       name = symbol.print_name
12       address = self.get_address(name)
13       size = self.get_size(name)
14       datatype = symbol.type
15       is_global = self.block.is_global
16       is_static = self.block.is_static
17
18       # skip address-less (e.g. register-residing) and size-less symbols
19
20       # create a container object and store it in asymbol list
21
22       # get the symbol's children (if any) accordingly
23
24     # traverse upwards in the block hierarchy
25     if self.block.superblock != None:
26       return self.symbols + BlockSyms(self.block.superblock).get_symbols()
27     else:
28       return self.symbols
29
30 [...]
```

Abbildung 4.3.: Iteration eines gdb.Block zur Gewinnung der zugehörigen Variablen.

1 möglich, d.h. eine Untersuchung der Komponenten von Komponenten – und beliebig tiefer verschachtelt – ist zur Zeit nicht möglich.

Es ist zu vermuten, dass eine beliebig tiefe Untersuchung über Indirektionen möglich ist. Da komplexen Datenstrukturen jedoch bei der Umfangsdefinition ausgeklammert wurden, und eine entsprechende Untersuchung der Möglichkeiten den Rahmen dieser Arbeit sprengen würde, wurde dies nicht weiter verfolgt.

Ermittlung des Speicherbereichs

Zur Ermittlung des Speicherbereichs einer Variablen können sowohl ein GDB-Kommando, als auch eine API-Funktion benutzt werden. Für eine Variable, welche durch ein *gdb.Symbol* repräsentiert ist, wird die API-Funktionalität genutzt, für alle anderen Variablen – insbesondere die Komponenten komplexer Datenstrukturen – das GDB-Kommando „p &var“.

Eine Überlappung von Speicherbereichen ist möglich, sofern es sich um Komponenten einer komplexen Datenstruktur oder Variablen auf einem Stack handelt.

GDB Bug 20120

Im Zuge der Implementierung wurde ein Bug in GDB oder dessen Python-API gefunden: Es existiert, aus Sicht der API, nicht ein einziger globaler Block für ein gegebenes Programm, sondern ein globaler Block *pro Übersetzungseinheit*. Dies hat zur Folge, dass das implementierte Plugin für Instruktionen aus einer Übersetzungseinheit A nur globale Variablen sehen kann, welche in dieser definiert wurden, nicht aber globale Variablen aus anderen Übersetzungseinheiten.

Dieses Verhalten wurde am 19.05.2016 als Fehler in der Python-API gemeldet, und ist unter Nummer 20120 im GDB-Bugzilla[1] zu finden. Zum gegenwärtigen Zeitpunkt hat der Bug-Report noch immer den Status „UNCONFIRMED“ und ist keinem GDB-Maintainer zugewiesen.

Aufgrund der Tatsache, dass ein Bugfix seitens der GDB-Entwickler für den Zeitraum dieser Arbeit nicht in Aussicht stand, wurde folgender Workaround in das Pruning von Beobachtungen eingebaut: Sämtliche Beobachtungen einer globalen oder statischen Variablen werden durch eine einzige Beobachtung ersetzt, deren Zeit-Intervall die komplette Programmlaufzeit überspannt. Dies ist möglich, da diese Variablen ortsfest im Speicher liegen, und die gesamte Programmlaufzeit lebendig sind – vgl. Abschnitt 3.1.2.

Dieser Workaround hat allerdings den Nachteil, dass Symbole aus Übersetzungseinheiten, deren Code nicht im „interessanten Abschnitt“ ausgeführt wird, von *observe-variables* nie beobachtet werden können. Damit kann VProf zum gegenwärtigen Zeitpunkt niemals *alle* globalen und statischen Variablen sehen und anteilige Fehlerinjektionsergebnisse zuordnen.

Unter Nutzung der Informationen aus der Symbol-Tabelle ist hierfür jedoch auch prinzipiell Abhilfe möglich. Da allerdings insbesondere dynamische Datenstrukturen, also lokale Variablen, im Fokus dieser Arbeit stehen, wurde dies nicht weiterverfolgt.

Probleme der Implementierung [OK]

Die Implementierung von „observe-variables“ ist zum gegenwärtigen Zeitpunkt von einem oder mehreren gravierenden Fehlern betroffen, wegen derer eine vergleichende Evaluation mit der Arbeit von Hellwig [Hel14] entfallen muss. Dies liegt darin begründet, dass die Speicherbereiche von lokalen Variablen und Funktionsargumenten nicht korrekt erhoben werden, und in Folge die anteiligen FI-Ergebnisse nicht korrekt ermittelt werden können.

[1] Im Zuge der Implementierung entdeckter Bug im GDB: `https://sourceware.org/bugzilla/show_bug.cgi?id=20120`

Die ermittelten Adressen von lokalen Variablen und Argumenten verändern sich von Instruktion zu Instruktion, und Funktionsargumente sind, entgegen der Theorie und der lokal durchgeführten Praxis, nicht mehr sichtbar, sobald eine weitere Funktion aufgerufen wird. Dieser Adressdrift ist unabhängig von der Ermittlungsmethode der Adresse; beide Methoden – vgl. Abschnitt 4.2.3 – liefern dasselbe Ergebnis.

name	type	start	end	o.address	file	line
data	CYG_ADDRWORD	18500	18500	0x112B38	tests/thread1.cxx	87
data	CYG_ADDRWORD	18501	18502	0x112B3C	tests/thread1.cxx	87
data	CYG_ADDRWORD	18503	18506	0x112B34	tests/thread1.cxx	87
data	CYG_ADDRWORD	18507	18507	0x112B28	tests/thread1.cxx	87
data	CYG_ADDRWORD	18508	18508	0x112B24	tests/thread1.cxx	87
data	CYG_ADDRWORD	18515	18515	0x112B24	tests/thread1.cxx	87
data	CYG_ADDRWORD	18516	18518	0x112B34	tests/thread1.cxx	87
data	CYG_ADDRWORD	18519	18519	0x112B28	tests/thread1.cxx	87
data	CYG_ADDRWORD	18520	18520	0x112B24	tests/thread1.cxx	87

Abbildung 4.4.: Manifestation des Adressdrifts.

Abbildung 4.4 zeigt einen Auszug der Beobachtungen eines Funktionsarguments: Dargestellt sind Name und Datentyp, Start- und Endzeit der Beobachtung, die ermittelte Basisadresse sowie Dateiname und Zeile der Definition. Die Ausführungszeit der umgebenden Funktion erstreckt sich über die dynamischen Instruktionen 18.500 bis einschließlich 18.520; im Intervall $[18.509, 18.514]$ findet ein Funktionsaufruf statt. Eine direkte Korrelation von Adressdrift und beispielsweise der Anzahl ausgeführter Instruktionen ist nicht erkennbar. Während der Ausführung der aufgerufenen Funktion existiert keine Beobachtung des Arguments.

Die beschriebenen Effekte treten nicht auf, wenn die zu Grunde liegenden Methoden manuell in einer interaktiven, lokalen GDB-Sitzung durchgeführt werden. Es ist somit möglich, dass der Fehler in der Implementierung des Plugins, in der Python-API, und/oder in Bochs bzw. dessen GDB Stub liegt. Die für diese Bachelorarbeit zur Verfügung stehende Zeit reicht jedoch nicht aus, um den zu Grunde liegenden Fehler zu lokalisieren und zu beheben.

4.3. VProf

Die Implementierung von VProf folgt im Wesentlichen dem entworfenen Klassendiagramm. Große Teile der notwendigen Verfahrensschritte sind bereits durch die hier implementierte GDB-Schnittstelle und die enwickelten GDB-Erweiterungen

abgedeckt. VProf dient der Automatisierung der Erhebung des Beobachtungsraums mit Hilfe des GDB-Plugins.

4.3.1. StepObserver

Mit dem StepObserver wurde das N-Stepping-Verfahren zur Erhebung des Beobachtungsraums implementiert. Die hierzu verwendete Methode *observe_golden_run* des Observer-Interfaces benutzt die GDB-Schnittstelle, um den interessanten Abschnitt des zu untersuchenden *golden-run* mittels Breakpoints einzugrenzen. Zwischen dem Auslösen des Start- und End-Breakpoints wird die Ziel-Software mit Hilfe des GDB in einer spezifizierbaren Schrittweite durchlaufen.

Nach jedem gemachten Schritt wird mittels des GDB-Plugins – über die Methode *observe* der Basisklasse – eine Beobachtung der existenten Variablen vorgenommen; die Ausgabe des GDB wird mittels eines, auf die Ausgabe des Plugins zugeschnittenen, regulären Ausdrucks geparst. Mit den so extrahierten Informationen über die Variablen und den beobachteten Zeitraum werden Objekte des Typs *Variable* und *Observation* angelegt, und in internen Datenstrukturen gespeichert.

4.3.2. Pruning

Um die Fragmentierung des Beobachtungsraums zu minimieren – vgl. Abschnitt 3.1.2 – ist das Pruning aller Beobachtungen als Methode der Basisklasse *Observer* implementiert. Diese macht sich die implementierte *join*-Methode des Klasse *Observation* zu nutze, um zeitlich zusammenhängende Beobachtungen desselben Speicherbereichs zusammenzufassen. Dies resultiert wie bereits beschrieben in einer Reduktion der Anzahl an Beobachtungen und in Folge in einer reduzierten Zahl notwendiger Anfragen zur Ermittlung anteiliger FI-Ergebnisse. Diese Anfrage wird im nachfolgenden Abschnitt beschrieben.

4.3.3. Ermittlung anteiliger FI-Ergebnisse

Die Ermittlung der anteiligen Fehlerinjektionsergebnisse einer Beobachtung ist in der Methode *get_results* der Klasse VProf implementiert. Sie basiert im Wesentlichen auf einer Datenbankabfrage, mit welcher der Fehlerraum auf das Raum- und Zeit-Intervall der fraglichen Beobachtung eingeschränkt wird. Die Abfrage in Abbildung 4.5 dargestellt, die ihr zu Grunde liegende Struktur in Abbildung 4.6. Die Tabelle *variant* identifiziert die untersuchte Ziel-Software; die zugehörige *variant_id* wird in den übrigen Tabellen als Identifikator und in Folge als Kriterium für die SQL-Joins verwandt. Der vollständige Instruktionstrace der Ziel-Software,

```
1  SELECT r.resulttype, SUM( (f2.time2-f1.time1) * t.width) as occurrences
2  FROM trace t
3  INNER JOIN fulltrace f1
4      ON f1.variant_id = t.variant_id
5      AND f1.instr = GREATEST(t.instr1, OBSERVATION.START)
6  INNER JOIN fulltrace f2
7      ON f2.variant_id = t.variant_id
8      AND f2.instr = LEAST(t.instr2, OBSERVATION.END)
9  INNER JOIN fspgroup g
10     ON t.variant_id = g.variant_id
11     AND t.instr2 = g.instr2
12     AND t.data_address = g.data_address
13 INNER JOIN result_EcosKernelTestProtoMsg r
14     ON g.pilot_id = r.pilot_id
15 INNER JOIN variant v
16     ON v.id = f.variant_id
17 WHERE v.variant = VARIANT
18     AND v.benchmark = BENCHMARK
19     AND NOT (t.instr2 < OBSERVATION.START OR t.instr1 > OBSERVATION.END)
20     AND g.data_address BETWEEN OBSERVATION.ADDRESS AND (OBSERVATION.ADDRESS +
           OBSERVATION.SIZE -1)
21     AND 1=1
22 GROUP BY r.resulttype
23 ORDER BY r.resulttype;
```

Abbildung 4.5.: Datenbankabfrage zur Ermittlung der anteiligen FI-Ergebnisse einer Beobachtung

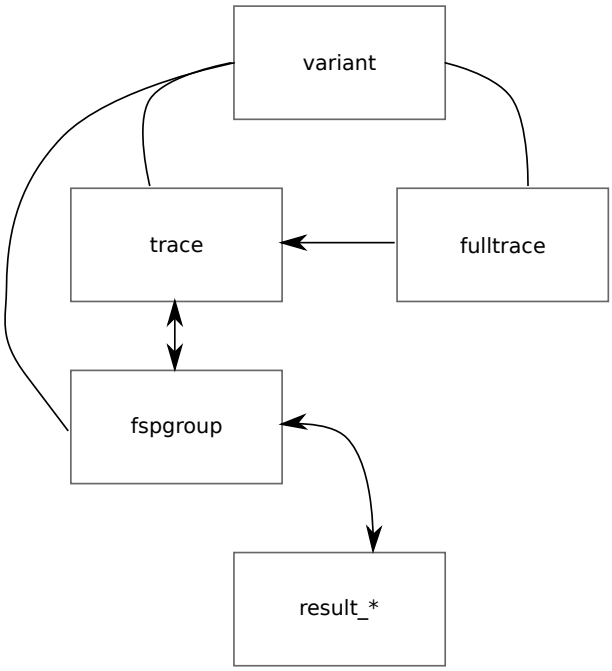


Abbildung 4.6.: Datenbankschema für VProf.

welcher in der Tabelle *fulltrace* gespeichert wird, dient als zeitliches Filterkriterium für Einträge des Speicherzugriffstraces, welcher in der Tabelle *trace* gespeichert wird. Darüber hinaus wird aus der zu einer Instruktion gehörigen Laufzeit – vgl. Abschnitt 4.1.3 – die Zeit-Basis für die anteiligen FI-Ergebnisse gebildet – siehe Zeilen 5 und 8 sowie 19 der Abbildung 4.5. Aus dem Speicherzugriffstrace werden sogenannte Fehlerraumpruning-Gruppen – Tabelle *fspgroup* – gebildet, aus welchen die für eine Software durchzuführenden FI-Experimente hervorgehen. Die Ergebnisse eines FI-Experiments werden in der *result*-Tabelle gespeichert und sind über einen Join mit der Tabelle *fspgroup* zugreifbar.

Die aus dieser Abfrage resultierenden, anteiligen Ergebniszahlen werden im Anschluss daran – innerhalb der Methode *insert_results* derselben Klasse – in die Datenbank eingefügt, und stehen in Folge zur Auswertung zur Verfügung.

4.3.4. Auswertungsmöglichkeiten

Die von VProf erhobenen Daten über die Ziel-Software können in Kombination mit den bisher ermittelbaren Daten genutzt werden um annähernd beliebig komplexe Filter für anteilige Fehlerinjektionsergebnisse zu kreieren.

Das Code-Listing in Abbildung 4.7 stellt mit der Bestimmung der anteiligen FI-Ergebnisse für alle Variablen einer Ziel-Software, eine vergleichsweise einfache Abfrage dar.

```
1  [...]
2
3  $MYSQL << EOT
4      SELECT v.name, v.file, v.line,
5             r.resulttype, SUM(r.occurrences) as occurrences
6      FROM vprof_variable v
7      INNER JOIN vprof_result r
8          ON r.variable_id=v.id
9      INNER JOIN variant x
10         ON x.id=v.variant_id
11     WHERE x.variant="$VARIANT"
12         AND x.benchmark="$BENCHMARK"
13     GROUP BY v.name, v.file, v.line, r.resulttype
14     ORDER BY v.name, v.file, v.line, r.resulttype
15 EOT
```

Abbildung 4.7.: Auszug eines Skripts zur Ermittlung anteiliger FI-Ergebnisse aller beobachteten Datenstrukturen. Dargestellt ist die durchgeführte Datenbankabfrage.

4.4. Zusammenfassung

Dieses Kapitel beschrieb die Umsetzung des in Kapitel 3 entworfenen Ansatzes zur Ermittlung anteiliger Fehlerinjektionsergebnisse. Mit der GDB-Schnittstelle wurde effektiv eine Schnittstelle in einen *golden-run* einer untersuchten Software geschaffen, welche es erlaubt mittels des implementierten GDB-Plugins Beobachtungen von Variablen vorzunehmen. Diese Möglichkeiten werden von VProf genutzt, um mit einer Erhebungsstrategie wie dem N-Stepping, den Beobachtungsraum der untersuchten Software zu ermitteln. Anhand einer Überlagerung von Beobachtungs- und Fehlerraum können anteilige Fehlerinjektionsergebnisse für einzelne Beobachtungen erhoben werden, und für einzelne Datenstrukturen daraus aggregiert werden. Darüber hinaus ermöglicht die Speicherung der erhobenen Daten in der FAIL*-Datenbank eine Nutzung auch in anderen Werkzeugen.

5. Evaluation

Dieses Kapitel evaluiert den Lösungsansatz dieser Arbeit anhand ausgewählter Testprogramme des Embedded-Betriebssystems eCos. Zwecks Vergleichbarkeit mit den Daten der Arbeiten von Hellwig [Hel14] und Borchert et al. [BSS16], wurden die Tests *thread1* und *mutex1* gewählt.

Die bisher, durch ein Skript namens „symbol_occurrences.sh“ (SymOcc), für Symbole erhobenen, anteiligen Fehlerinjektionsergebnisse, sowie die Performanz dieses Verfahrens werden in Abschnitt 5.1 als Evaluationsgrundlage für VProf herangezogen.

In Abschnitt 5.2 wird das N-Stepping-Verfahren auf Performanzgewinn und Aliasing-Einfluss für verschiedene Schrittweiten untersucht.

Abschnitt 5.3 beschreibt den Ansatz einer vergleichenden Evaluation des Ansatzes dieser Arbeit mit dem Ansatz von Hellwig [Hel14].

5.1. Symbole

Das Erheben der für Symbole anteiligen FI-Ergebnisse erfolgte bisher mittels SymOcc, sodass die auf Symbole umlegbaren FI-Ergebnisse als Muss-Kriterium genutzt werden können. Ferner werden die Auswirkungen des GDB-Bugs untersucht und die Performanz hinsichtlich Geschwindigkeit von VProf und SymOcc gegenübergestellt.

5.1.1. Sichtbarkeit

Wie in Abschnitt 4.2.3 ausgeführt, ist die Implementierung des Beobachtungs-Plugins von einem Bug in GDB beziehungsweise dessen Python-API betroffen, welcher zu einer Einschränkung der Sichtbarkeit von Symbolen für VProf führt. Diese Auswirkung kann mit Hilfe der Ergebnisse von SymOcc quantifiziert werden. Tabelle 5.1 stellt VProf und SymOcc bezüglich der Anzahl der vom jeweiligen Verfahren zuordenbaren Symbole gegenüber, wobei SymOcc als Grundwahrheit dient, da es auf der Symboltabelle der Ziel-Software basiert.

Wie aus diesen Daten ersichtlich wird, sieht VProf im Mittel über alle acht evaluierten Testprogramme nur circa 51.4 % ($\sigma = 9.9\,\%$) der tatsächlich existen-

Testprogramm	$\#_{VProf}$	$\#_{SymOcc}$	%
thread1 (O0)	34	83	40.96%
thread1 (O1, O2, O3)	31	73	42.46%
mutex1 (O0)	51	88	57.95%
mutex1 (O1, O2, O3)	48	78	61.54%

Tabelle 5.1.: Quantifizierung für VProf sichtbarer Symbole.

ten Symbole. Die Varianz des Stichprobenmittels liegt in der Mittelung über zwei unterschiedliche Ziel-Softwares, und deren unterschiedlicher Quellcodestruktur, begründet.

5.1.2. Anteilige Ergebnisse

Für jede von VProf beobachtete globale und statische Variable wurden die anteiligen Fehlerinjektionsergebnisse mit den durch SymOcc erhobenen Zahlen verglichen. Notwendigerweise müssen beide Verfahren dieselben anteiligen Ergebniszahlen ermitteln.

Zur Automatisierung dieses Vergleichs für alle betrachteten Testprogramme wurde ein Shell-Skript geschrieben, welches als Listing A.3 im Anhang dieser Arbeit zu finden ist, und wie folgt arbeitet: Für jedes von VProf gesehene Symbol werden die anteiligen Ergebniszahlen jeder Verhaltensklasse mit den korrespondierenden, durch SymOcc ermittelten, Zahlen verglichen. Im Falle von Ungleichheit wird eine entsprechende Meldung ausgegeben, sodass im Idealfall keine Ausgabe erfolgt.

In der Realität erzeugt dieses Skript jedoch für alle acht Testprogramme Ausgaben für aufgetretene Ungleichheiten der mit beiden Verfahren ermittelten Ergebniszahlen. Listing A.2, welches sich ebenfalls im Anhang dieser Arbeit findet, beinhaltet eine beispielhafte Ausgabe dieses Vergleichs für das Testprogramm thread1, ohne Compiler-Optimierungen. Wie daraus ersichtlich wird, handelt es sich bei jeder ausgegebenen Ungleichheit der Ergebniszahlen ausschließlich um Teilstrukturen komplexer Variablen – beispielsweise Dividend und Divisor des *rtc_resolution*-Objekts –, für die VProf anteilige Zahlen bestimmen kann, SymOcc jedoch nicht.

Folglich erhebt VProf die korrekten, anteiligen Fehlerinjektionsergebnisse für die Symbole der Ziel-Software.

Testprogramm	$t_{GR,N=1}$	$t_{Erg.}$	t_{VProf}	t_{SymOcc}
thread1 (O0)	140.26s	14.76s	155.02s	1,006.17s
thread1 (O1)	109.93s	5.83s	115.76s	254.08s
thread1 (O2)	114.46s	4.93s	119.39s	247.31s
thread1 (O3)	83.65s	2.86s	86.51s	279.46s
mutex1 (O0)	243.93s	34.29s	278.22s	1,371.95s
mutex1 (O1)	175.19s	7.87s	183.06s	260.80s
mutex1 (O2)	185.62s	6.56s	192.18s	355.86s
mutex1 (O3)	124.83s	6.03s	130.86s	324.58s

Tabelle 5.2.: Erhebungsdauer, Vergleich VProf ↔ SymOcc.

5.1.3. Performanz

Zur Evaluation der Performanz von VProf dienen die zur Beobachtung aller Variablen notwendige Laufzeit und die Laufzeit der Ermittlung aller anteiligen Fehlerinjektionsergebnisse für alle getätigten Beobachtungen. Die zur Ermittlung der anteiligen FI-Ergebnisse von Symbolen, mittels SQL-Abfrage, notwendige Zeit wird durch die Netzwerklatenz zum Datenbankserver dominiert, und ist mit unter $10\,ms$ vernachlässigbar klein. Als Vergleichsgrundlage dient die Laufzeit von SymOcc; aufgrund der Größenordnung der Werte, ist hierbei die Netzwerklatenz, wie bereits erwähnt, vernachlässigbar.

Die Tabelle 5.2 stellt die, für alle acht Testprogramme ermittelten, Laufzeiten von VProf und SymOcc gegenüber:

Bei $t_{GR,N=1}$ handelt es sich um die Laufzeit des golden-run, in welchem mittels N-Stepping und einer Schrittweite von N=1 Beobachtungen aller Variablen getätigt wurden. $t_{Erg.}$ ist die Zeit, welche zur Ermittlung der anteiligen Ergebnisse aller erhobenen Beobachtungen, notwendig ist. Die Summe dieser Teillaufzeiten ist somit die relevante, zu vergleichende Laufzeit von VProf, und unter t_{VProf} in der Tabelle zu finden. Bei t_{SymOcc} handelt es sich entsprechend um die Zeit, welche SymOcc zur Ermittlung der für Symbole anteiligen FI-Ergebnisse benötigt.

Es zeigt sich, dass VProf lediglich zwischen 15 und 70 Prozent der Laufzeit von SymOcc benötigt. Somit kann festgestellt werden, dass VProf deutlich performanter bezüglich der Laufzeit ist.

Testprogramm	$\#_{Instr}$	$t_{GR,N=1}$	$t_{GR,N=2}$	$t_{GR,N=4}$
thread1 (O0)	30609	140.26s	92.29s *(-34.2 %)*	65.47s *(-53.3 %)*
thread1 (O1)	12638	109.93s	73.57s *(-33.1 %)*	54.62s *(-50.3 %)*
thread1 (O2)	12664	114.46s	77.51s *(-32.3 %)*	56.58s *(-50.6 %)*
thread1 (O3)	6844	83.65s	62.63s *(-25.1 %)*	49.38s *(-41.0 %)*
mutex1 (O0)	58717	243.93s	143.32s *(-41.3 %)*	96.25s *(-60.5 %)*
mutex1 (O1)	22914	175.19s	104.10s *(-40.6 %)*	71.11s *(-59.4 %)*
mutex1 (O2)	23192	185.62s	110.58s *(-40.4 %)*	78.11s *(-57.9 %)*
mutex1 (O3)	13510	124.83s	80.19s *(-35.8 %)*	59.16s *(-52.6 %)*
Mittelwerte:			-35.3 % ($\sigma = 5.5\,\%$)	-53.2 % ($\sigma = 6.3\,\%$)

Tabelle 5.3.: Dauer eines mit VProf untersuchten golden-run in Abhängigkeit der Schrittweite des N-Stepping-Verfahrens.

5.2. N-Stepping

Zur Evaluation des N-Stepping-Verfahrens werden in Abschnitt 5.2.1 der Performanzgewinn durch höhere Schrittweiten, und in Abschnitt 5.2.2 die Auswirkungen des Aliasing-Problems betrachtet.

5.2.1. Performanzgewinn

Zur Ermittlung des Performanzgewinns durch höhere Schrittweiten, wurde für jedes der untersuchten Testprogramme die Dauer der Beobachtung eines golden-run unter Verwendung von N-Stepping mit Schrittweiten $N \in \{1, 2, 4\}$ gemessen.

Wie aus Tabelle 5.3 deutlich wird, resultiert für jedes so untersuchte Testprogramm aus der Erhöhung der Schrittweite eine deutliche Steigerung der Geschwindigkeit. Die prozentualen Angaben beziehen sich auf die Dauer des *golden-run* bei Untersuchung mit einer Schrittweite von N=1.

5.2.2. Aliasing

Zum Nachweis des Aliasing-Effekts bei Schrittweiten größer 1, wurden das Testprogramm thread1, ohne Compiler-Optimierungen, mit Schrittweiten $N \in \{1, 2, 4\}$ beobachtet. Für jede der Schrittweiten wurden die für die beobachteten Variablen anteiligen FI-Ergebnisse ermittelt, um eine Untersuchung des Phänomens zu ermöglichen.

Variable	$\vec{r_{N=1}}$	$\vec{r_{N=2}}$	$\vec{r_{N=4}}$
stack_base	$\begin{pmatrix} 3358 \\ 15 \\ 157 \\ 214 \end{pmatrix}$	$\begin{pmatrix} 3430 \\ 20 \\ 128 \\ 198 \end{pmatrix}$	$\begin{pmatrix} 3648 \\ 30 \\ 60 \\ 102 \end{pmatrix}$
Cyg_Thread::thread_list	$\begin{pmatrix} 14893156 \\ 0 \\ 548992 \\ 60 \end{pmatrix}$	$\begin{pmatrix} 14893156 \\ 0 \\ 548992 \\ 60 \end{pmatrix}$	$\begin{pmatrix} 14893156 \\ 0 \\ 548992 \\ 60 \end{pmatrix}$

Tabelle 5.4.: Nachweis des Aliasing-Effekts bei lokalen Variablen.

In Tabelle 5.4 ist der Einfluss der Schrittweite auf die anteiligen Fehlerinjektionsergebnisse dargestellt. Beispielhaft wurden die lokale Variable *stack_base* und die globale Variable *Cyg_Thread::thread_list* aus der eCos-Übersetzungseinheit *thread.cxx* gewählt. Für die drei Schrittweiten ist jeweils ein Vektor eingetragen, welcher folgendes Format hat: $\vec{r} = (\#_{OK}, \#_{SDC}, \#_{TIMEOUT}, \#_{TRAP})^T$
Die einzelnen Komponenten stellen somit die Ergebniszahlen der korrespondierenden Verhaltensklasse der Ziel-Software dar.

Wie in Abschnitt 3.1.2 beschrieben, sind nur lokale Variablen von Aliasing betroffen, was in Tabelle 5.4 an der globalen Variable deutlich wird. Tatsächlich trifft diese Beobachtung auf alle globalen und statischen Variablen zu. Die lokale Variable hingegen zeigt schon für eine kleine Erhöhung der Schrittweite deutliches Aliasing: So führt etwa eine Vervierfachung der Schrittweite zu einer Verdopplung der anteiligen Ergebnisse der Verhaltensklasse SDC, und in etwa zu einer Halbierung der anteiligen Ergebnisse der Klasse TRAP.

Ein genereller Trend für die Entwicklung des Aliasing-Einflusses unter steigenden Schrittweiten kann nicht angegeben werden, da dieser Einfluss ausschließlich an der relativen Lage von Lebenszeit-Intervallen hängt. Somit beeinflusst für eine gegebene Schrittweite bereits ein Versatz von einer Instruktion die ermittelten Ergebnisse massiv.

5.3. Zuordenbare Ergebnisse

Eine qualitative Beurteilung der Zuordnungsgüte, anhand der von Hellwig [Hel14] verwendeten Metrik des Prozentsatzes zuordenbarer Fehler, insbesondere im Hinblick auf das Verhalten unter Compiler-Optimierungen, war vorgesehen, konnte

jedoch zum Zeitpunkt der Erstellung der Ausarbeitung, aufgrund der in Abschnitt 4.2.3 beschriebenen Probleme, nicht durchgeführt werden.

Sofern korrekte Daten über den Beobachtungsraum vorliegen, ist ein solcher Vergleich jedoch trivial: Die Speicherbereiche und anteiligen FI-Ergebnisse der Stacks müssen festgestellt werden, sodass sich anhand dieser der Prozentsatz auf Datenstrukturen zuordenbarer Fehler des jeweiligen Stacks ermitteln lässt. Unter Verwendung verschiedener Optimierungslevel wäre dadurch ein direkter Vergleich mit dem Verhalten der Zuordnungsgüte unter Compiler-Optimierungen möglich.

6. Zusammenfassung und Ausblick

Dieses Kapitel dient der Rekapitulation der wesentlichen Beiträge dieser Arbeit sowie als Ausblick auf Weiterentwicklung und zukünftigen Nutzen der entwickelten Methoden und nicht zuletzt des Tools VProf.

6.1. Zusammenfassung [WIP]

Diese Arbeit entwickelte einen theoretischen Ansatz zur Zuordnung von Fehlerinjektionsergebnissen zu insbesondere dynamischen Datenstrukturen und setze diese im Werkzeug VProf um. Es gibt zum gegenwärtigen Zeitpunkt, abgesehen vom Forschungsprototypen von Hellwig [Hel14], keine existierenden Werkzeuge, die dies ermöglichen. Neben der Betrachtung von Symbolen und lokalen Variablen sind sowohl der Ansatz, als auch das Werkzeug zumindest theoretisch dahingehend erweiterbar, dass der gesamte Speicher einer untersuchten Software betrachtet und analysiert werden kann. Der Vergleich mit existierenden Möglichkeiten zur Zuordnung von FI-Ergebnissen auf globale und statische Variablen zeigt, dass der vorgestellte Ansatz dieselben Ergebnisse liefert, und dafür weniger Zeit benötigt, obwohl Informationen auch über lokale Variablen und Funktionsargumente erhoben werden. Vorbehaltlich einer zukünftig fehlerfrei funktionierenden Implementierung – vgl. Abschnitt 4.2.3 – scheint es daher ratsam VProf statt der bisherigen Möglichkeiten zur feingranularen Fehlertoleranzanalyse heranzuziehen.

Im Zuge der Entwicklung von VProf wurden darüber hinaus infrastrukturelle Komponenten geschaffen und Bestandteile des zugrunde liegenden Frameworks FAIL* erweitert. Insbesondere hervorzuheben sind das Erheben der tatsächlichen Zeitdauer ausgeführter Instruktionen, die Nutzbarmachung des GDB Stubs in FAILBochs sowie die Implementierung einer quasi-interaktiven GDB-Schnittstelle für C++.

6.2. Ausblick

Die in Abschnitt 4.2.3 angeführten, aufgetretenen Probleme der Implementierung und/oder der Umgebung müssen gefunden und behoben werden, damit VProf uneingeschränkt genutzt und vor allem weiterentwickelt werden kann.

VProf stellt ein wertvolles Werkzeug für künftige Forschung an SIHFT dar. Hervorzuheben sind insbesondere die Möglichkeiten zur:

- Untersuchung weiterer Datenstrukturen des Stacks wie beispielsweise Rücksprungadressen.[1]
- Untersuchung beliebig komplex verschachtelter Datenstrukturen, und Datenstrukturen auf dem Heap.
- Implementierung und Evaluation des Breakpoint-Hopping-Ansatzes im Hinblick auf Software mit vergleichsweise langer Ausführungszeit.
- Übertragen des Ansatzes auf die anderen FAIL*-Backends.
- Nutzung der erhobenen Informationen zur Annotation von Fehlerraumplots zwecks Verbesserung der Visualisierung von Fehlertoleranz.
- Nutzung der Informationen des Beobachtungsraums zur gezielten Fehlerinjektion in Datenstrukturen, also eine Art Datenstruktur-basiertes Fehlerraum-Pruning.

Eine Veröffentlichung von VProf und der entwickelten GDB-Schnittstelle unter geeigneten Open-Source-Lizenzen ist vorgesehen, sobald sie einen ausgereiften Zustand erreicht haben.

6.3. Danksagungen

Mein Dank gilt insbesondere meiner Freundin Nele und meinem guten Freund Patrick, die mich stets tatkräftig und geduldig unterstützt haben. Darüber gilt mein Dank selbstverständlich auch meinen Eltern, ohne deren Unterstützung mir das Studium nicht möglich gewesen wäre.

Nicht zuletzt möchte ich meinen Betreuern, Horst Schirmeier und Olaf Spinczyk, für Rat, Tat und vertrauensvolle Zusammenarbeit danken.

[1] Dies wäre gerade im Hinblick auf beispielsweise die von Borchert et al. [BSS13] entwickelte „Return-address protection“ interessant, und würde zukünftige derartige Untersuchungen erleichtern.

Literatur

[10] *DWARF Debugging Information Format Version 4*. DWARF Standards Committee. Juni 2010. URL: http://www.dwarfstd.org/doc/DWARF4.pdf.

[14] *libdwarf - elftoolchain*. Apr. 2014. URL: http://sourceforge.net/apps/trac/elftoolchain/wiki/libdwarf.

[ALR01] A. Avižienis, J.-C. Laprie und B. Randell. *Fundamental concepts of dependability*. Techn. Ber. UCLA CSD Report 0100. Computer Science Department, University of California, Los Angeles, USA, 2001.

[Bel05] F. Bellard. "QEMU, a fast and portable dynamic translator". In: 2005, S. 41–46.

[Bin+11] N. Binkert u. a. "The Gem5 Simulator". In: 39.2 (Aug. 2011), S. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[Böc11] A. Böckenkamp. *Untersuchung des Code-Location-Problems in C/C++-Programmen für eingebettete x86- und ARM-Architekturen*. Emil-Figge-Straße 50, 44227 Dortmund, Germany, 2011.

[Bor05] S. Y. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation". In: 25.6 (2005), S. 10–16. ISSN: 0272-1732.

[BP03] A. Benso und P. Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*. Frontiers in electronic testing. Boston, Dordrecht, London, 2003. ISBN: 1-4020-7589-8.

[Bro60] D. T. Brown. "Error Detecting and Correcting Binary Codes for Arithmetic Operations". In: *IRE Transactions on Electronic Computers* EC-9.3 (Sep. 1960), S. 333–337. ISSN: 0367-9950. DOI: 10.1109/TEC.1960.5219855.

[BSH75] D. Binder, E. Smith und A. Holman. "Satellite Anomalies from Galactic Cosmic Rays". In: 22.6 (Dez. 1975), S. 2675–2680. ISSN: 0018-9499. DOI: 10.1109/TNS.1975.4328188.

[BSS13] C. Borchert, H. Schirmeier und O. Spinczyk. "Return-Address Protection in C/C++ Code by Dependability Aspects". In: *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*. (Koblenz, Germany). Lecture Notes in Informatics. German Society of Informatics, Sep. 2013.

[BSS16] C. Borchert, H. Schirmeier und O. Spinczyk. "Generic Soft-Error Detection and Correction for Concurrent Data Structures". In: *IEEE Transactions on Dependable and Secure Computing* PP.99 (2016). ISSN: 1545-5971. DOI: 10.1109/TDSC.2015.2427832.

[GWA79] C. S. Guenzer, E. A. Wolicki und R. G. Allas. "Single Event Upset of Dynamic Rams by Neutrons and Protons". In: *IEEE Transactions on Nuclear Science* 26 (Dez. 1979), S. 5048–5052. DOI: 10.1109/TNS.1979.4330270.

[Hel14] R. Hellwig. *Automatisierte Analyse von Fehlerinjektionsexperimenten mit Fail**. Masterarbeit. Emil-Figge-Straße 50, 44227 Dortmund, Germany, 2014.

[Hof16] M. Hoffmann. "Konstruktive Zuverlässigkeit: Eine Methodik für zuverlässige Systemsoftware auf unzuverlässiger Hardware". Diss. Erlangen: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2016. URL: https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/7038.

[Hoh38] P. T. A. B. von Hohenheim genannt Paracelsus. *Septem Defensiones, Kapitel: Die dritte Defension wegen des Schreibens der neuen Rezepte.* online; Abruf 02.09.2016. zeno.org, 1538. URL: http://www.zeno.org/Philosophie/M/Paracelsus/Septem+Defensiones/Die+dritte+Defension+wegen+des+Schreibens+der+neuen+Rezepte.

[Int13] International Roadmap Committee. "International Technology Roadmap for Semiconductors, 2013 edn. (Executive Summary)". In: *Semiconductor Industry Association* (2013).

[Law96] K. P. Lawton. "Bochs: A Portable PC Emulator for Unix/X". In: *Linux Journal* 1996.29es (1996), S. 7.

[Mar+12] A. Martínez-Álvarez u. a. "Compiler-Directed Soft Error Mitigation for Embedded Systems". In: 9.2 (März 2012), S. 159–172. ISSN: 1545–5971. DOI: 10.1109/TDSC.2011.54.

[Muk08] S. Mukherjee. *Architecture design for soft errors*. Burlington, MA: Elsevier, 2008. URL: https://cds.cern.ch/record/1251606.

[MW79] T. C. May und M. H. Woods. “Alpha-particle-induced soft errors in dynamic memories”. In: *IEEE Transactions on Electron Devices* 26.1 (Jan. 1979), S. 2–9. ISSN: 0018-9383. DOI: 10.1109/T-ED.1979.19370.

[NX06] V. Narayanan und Y. Xie. “Reliability concerns in embedded system designs”. In: 39.1 (2006), S. 118–120. ISSN: 0018-9162.

[Por+11] M. Portela-Garcia u. a. “Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures”. In: *IEEE Transactions on Dependable and Secure Computing* 8.2 (2011), S. 308–314. ISSN: 1545-5971. DOI: http://doi.ieeecomputersociety.org/10.1109/TDSC.2010.50.

[Pro] G. Project. *Libgdb*. online; Abruf 19.09.2016. URL: https://www.gnu.org/software/gdb/papers/libgdb/libgdb.html.

[Rad13] L. Rademacher. “FailPanda: Fehlerinjektionsexperimente auf einer eingebetteten ARM-Plattform”. 2013.

[Rat05] D. Rath. “OpenOCD: Open On-Chip Debugging”. Diploma Thesis. FH Augsburg, Juli 2005.

[RS60] I. Reed und G. Solomon. “Polynomial codes over certain finite fields”. In: *Journal of the Society of Industrial and Applied Mathematics* 8.2 (Juni 1960), 300–304. URL: http://www.jstor.org/pss/2098968.

[Sch+11] H. Schirmeier u. a. “DanceOS: Towards Dependability Aspects in Configurable Embedded Operating Systems”. In: Hrsg. von A. Orailoglu. Heraklion, Greece, Jan. 2011, S. 21–26.

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

[Sch16] H. Schirmeier. “Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance”. Dissertation. Technische Universität Dortmund, Juli 2016. DOI: 10.17877/DE290R-17222.

[Sch97] R. R. Schaller. “Moore’s Law: Past, Present, and Future”. In: *IEEE Spectr.* 34.6 (Juni 1997), S. 52–59. ISSN: 0018-9235. DOI: 10.1109/6.591665. URL: http://dx.doi.org/10.1109/6.591665.

[Sha49] C. E. Shannon. “Communication in the Presence of Noise”. In: *Proceedings of the IRE* 37.1 (Jan. 1949), S. 10–21. ISSN: 0096-8390. DOI: 10.1109/JRPROC.1949.232969.

[SL12] V. Sridharan und D. Liberty. “A study of DRAM failures in the field”. In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12, Salt Lake City, UT, USA - November 11 - 15, 2012*. 2012, S. 76. DOI: 10.1109/SC.2012.13. URL: http://dx.doi.org/10.1109/SC.2012.13.

[Soua] Sourceware. *GDB Front Ends*. online; Abruf 19.09.2016. URL: https://sourceware.org/gdb/wiki/GDB%20Front%20Ends.

[Soub] Sourceware. *GDB machine interface*. online; Abruf 19.09.2016. URL: https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html.

[Souc] Sourceware. *GDB Python-API*. online; Abruf 18.09.2016. URL: https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html.

[Soud] Sourceware. *GDB Python-API, Blocks In Python*. online; Abruf 18.09.2016. URL: https://sourceware.org/gdb/onlinedocs/gdb/Blocks-In-Python.html.

[Soue] Sourceware. *libGDB*. online; Abruf 19.09.2016. URL: https://sourceware.org/gdb/papers/libgdb2/libgdb.html%5C#SEC9.

[Sri+13] V. Sridharan u. a. “Feng shui of supercomputer memory: positional effects in DRAM and SRAM faults”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC’13, Denver, CO, USA - November 17 - 21, 2013*. 2013, 22:1–22:11. DOI: 10.1145/2503210.2503257. URL: http://doi.acm.org/10.1145/2503210.2503257.

[SS96] R. M. Stallman und C. Support. *Debugging with GDB : The GNU source-level debugger, GDB version 4.16*. Boston, MA: Free software foundation, 1996. ISBN: 1-88211-409-4. URL: http://opac.inria.fr/record=b1104446.

[Sta88] R. M. Stallman. “Using and porting the GNU compiler collection”. In: (1988).

[TG00] C. Tice und S. L. Graham. *Key instructions : solving the code location problem for optimized code*. Techn. Ber. 164. Palo Alto: Compaq. Systems research center (Palo Alto, CA US), 2000. URL: http://opac.inria.fr/record=b1044035.

[Yos13] J. Yoshida. “Toyota Case: Single Bit Flip That Killed”. In: (Okt. 2013). online; Abruf 29.08.2016. URL: http://www.eetimes.com/document.asp?doc_id=1319903&print=yes.

Abbildungsverzeichnis

1.1. Lebendige Speicherbereiche und zugeordnete Variablen 2

2.1. Schematische Darstellung der „Fehlerkette“. Entnommen aus [Hof16] 6
2.2. Beispielhafter, schematischer Fehlerraumplot. Verhaltensklassen sind annotiert. 8
2.3. Beziehung der Quellcodestruktur zu Blöcken und deren Hierarchie . 10
2.4. Speicherlayout einer von-Neumann-Maschine inkl. Aufbau des Stacks 11
2.5. Untersuchungsablauf mit FAIL*, entnommen aus [Sch+15] 14

3.1. Fehlerraumplot mit überlagertem Beobachtungsraum. 18
3.2. Visualisierung des Multi-Stack-Problems. 20
3.3. Effekt unterschiedlicher Schrittweiten auf den Beobachtungsraum. . 22
3.4. Klassendiagramm von VProf. 25
3.5. Datenbankschema für VProf. 27

4.1. Nutzung des Systemaufrufs *dup2* zur Modifikation des Standard-Ausgabe-Kanals. 34
4.2. Quellcode des implementierten Stop-Handlers. 35
4.3. Iteration eines gdb.Block zur Gewinnung der zugehörigen Variablen. 37
4.4. Manifestation des Adressdrifts. 39
4.5. Datenbankabfrage zur Ermittlung der anteiligen FI-Ergebnisse einer Beobachtung 41
4.6. Datenbankschema für VProf. 41
4.7. Auszug eines Skripts zur Ermittlung anteiliger FI-Ergebnisse aller beobachteten Datenstrukturen. Dargestellt ist die durchgeführte Datenbankabfrage. 42

A.1. SQL-Code der entworfenen Tabellen für VProf. II
A.2. Vergleich von VProf und SymOcc anhand des Testprogramms thread1, ohne Compiler-Optimierungen III
A.3. Skript zum Vergleich der von VProf und SymOcc ermittelten, anteiligen Fehlerinjektionsergebnisse für die Symbole eines Testprogramms. IV

A. Anhang

```
1  -- MySQL needs InnoDB as table-enging for FK constraints to work!
2  --       MyISAM simply ignores these constraints.
3  --       InnoDB cannot use a FK from a MyISAM-table
4  --   => :-(
5
6  CREATE TABLE IF NOT EXISTS vprof_variable (
7       id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
8       variant_id INT(11) NOT NULL,
9       name VARCHAR(255) NOT NULL,
10      type VARCHAR(255) NOT NULL,
11      file VARCHAR(255) NOT NULL,
12      line INT(10) NOT NULL,
13      parent_type VARCHAR(255),
14      global BOOLEAN,
15      static BOOLEAN,
16
17      PRIMARY KEY (id),
18      UNIQUE KEY variable (name, type, file, line, variant_id),
19      INDEX ind_variant_id (variant_id),
20      FOREIGN KEY (variant_id)
21          REFERENCES variant (id)
22          ON DELETE CASCADE
23 );
24
25 CREATE TABLE IF NOT EXISTS vprof_observation (
26      id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
27      variant_id INT(11) NOT NULL,
28      variable_id BIGINT UNSIGNED NOT NULL,
29      start INT UNSIGNED  NOT NULL,
30      end INT UNSIGNED NOT NULL,
31      address BIGINT UNSIGNED NOT NULL,
32      size BIGINT UNSIGNED NOT NULL,
33
34      PRIMARY KEY (id),
35      -- this does not prevent space-time-intervals from overlapping!
36      UNIQUE KEY (variable_id, variant_id, start, end, address, size),
37      INDEX ind_variant_id (variant_id),
38      FOREIGN KEY (variant_id)
39          REFERENCES variant (id)
40          ON DELETE CASCADE,
41      INDEX ind_variable_id (variable_id),
42      FOREIGN KEY (variable_id)
43          REFERENCES vprof_variable (id)
44          ON DELETE CASCADE
45 );
46
47 -- this table uses a string for the resulttype,
48 -- because this doesn't depend on a specific result-table
49 CREATE TABLE IF NOT EXISTS vprof_result (
50      variant_id INT(11) NOT NULL,
51      variable_id BIGINT UNSIGNED NOT NULL,
52      observation_id BIGINT UNSIGNED NOT NULL,
53      resulttype VARCHAR(255) NOT NULL,
54      occurrences BIGINT(11) UNSIGNED NOT NULL,
55
56      PRIMARY KEY (variable_id, observation_id, resulttype)
57 );
```

Abbildung A.1.: SQL-Code der entworfenen Tabellen für VProf.

```
1  Cyg_Interrupt::interrupt_disable_spinlock.lock, OK: 15442208 (VProf),
       0 (SymOcc)
2  cyg_kapi_check_structure_sizes.dummy, OK: 15442208 (VProf), 0 (SymOcc)
3  Cyg_RealTimeClock::rtc.interrupt, OK: 47552314 (VProf), 0 (SymOcc)
4  Cyg_RealTimeClock::rtc.interrupt, TIMEOUT: 24788330 (VProf), 0
       (SymOcc)
5  Cyg_RealTimeClock::rtc.interrupt, TRAP: 35754812 (VProf), 0 (SymOcc)
6  Cyg_RealTimeClock::rtc.rtc, OK: 124458294 (VProf), 0 (SymOcc)
7  Cyg_RealTimeClock::rtc.rtc, TIMEOUT: 26178726 (VProf), 0 (SymOcc)
8  Cyg_RealTimeClock::rtc.rtc, TRAP: 50111684 (VProf), 0 (SymOcc)
9  Cyg_Scheduler::scheduler.scheduler, OK: 492091628 (VProf), 0 (SymOcc)
10 Cyg_Scheduler::scheduler.scheduler, TIMEOUT: 11356391 (VProf), 0
       (SymOcc)
11 Cyg_Scheduler::scheduler.scheduler, TRAP: 6144845 (VProf), 0 (SymOcc)
12 rtc_resolution.dividend, OK: 15442208 (VProf), 0 (SymOcc)
13 rtc_resolution.divisor, OK: 15442208 (VProf), 0 (SymOcc)
```

Abbildung A.2.: Vergleich von VProf und SymOcc anhand des Testprogramms thread1, ohne Compiler-Optimierungen

```
1  #!/bin/bash
2  set -e
3
4  if [ ! $# -eq 1 ]; then
5      echo "usage: $0 BENCHMARK" >&2
6      echo "This opens and compares BENCHMARK.vprof to BENCHMARK.symocc"
7      exit 1
8  fi
9  BENCHMARK=$1
10
11 # extract symbol name, resulttype and occurrences from VProf's output
12 VP_RES=`awk 'NR>1 { printf "%s,%s,%d\n", $1,$4,$5 }' $BENCHMARK.vprof`
13 for res in $VP_RES; do
14     SYM=$( echo $res | awk -F "," ' { print $1 }' )
15     TYPE=$( echo $res | awk -F "," ' { print $2 }' )
16     OCC=$( echo $res | awk -F "," ' { print $3 }' )
17
18     # occurrences as surveyed by symbol_occurrences.sh
19     SO_OCC=$( awk -v SYM="$SYM" -v TYPE="$TYPE" '$3==SYM && $5==TYPE
           { print $6}' $BENCHMARK.symocc )
20     # default to 0, if SymOcc has no results for $SYM
21     SO_OCC=${SO_OCC:=0}
22
23     # on mismatch of VProf and SymOcc, report
24     if [ $OCC -ne $SO_OCC ]; then
25         echo "$SYM, $TYPE: $OCC (VProf), $SO_OCC (SymOcc)"
26     fi
27 done
```

Abbildung A.3.: Skript zum Vergleich der von VProf und SymOcc ermittelten, anteiligen Fehlerinjektionsergebnisse für die Symbole eines Testprogramms.

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