technische universität
dortmund

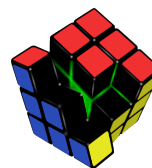
Masterarbeit

FailPanda: Fehlerinjektionsexperimente auf einer eingebetteten ARM-Plattform

Lars Rademacher
19. Dezember 2013

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

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



Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 19. Dezember 2013

Lars Rademacher

Zusammenfassung

Fehlerinjektion (FI) ist ein experimentelles Werkzeug zur Untersuchung der Zuverlässigkeit von unter Gesichtspunkten der Fehlertoleranz entwickelten Hard- und Softwaresystemen bei Fehlerpräsenz. Das FI-Framework *Fail**, welches im Rahmen des Forschungsprojekts „Dependability Aspects in Configurable Embedded Operating Systems“ (DanceOS) entwickelt wird, bietet die Möglichkeit zur parallelen und verteilten Durchführung von FI-Experimenten. Die Architektur des Frameworks sieht die Verwendung von Hardwaresimulatoren als Zielsysteme vor, wobei durch eine generische Zielsystemschnittstelle ebenfalls die Möglichkeit zur Anbindung von Hardwaresystemen besteht.

Ein zentrales Problem bei der Verwendung von Simulatoren ist die eingeschränkte Aussagekraft der gewonnenen Ergebnisse aufgrund der unvermeidbaren Abweichungen vom Verhalten realer Systeme. Es ist daher zu erwarten, dass sich durch die Wiederholung einer Teilmenge der simulativ durchgeführten FI-Experimente auf realer Hardware die Aussagekraft des Gesamtergebnisses steigern lässt. In dieser Masterarbeit werden mögliche Ansätze zur FI auf einer eingebetteten ARM-Plattform (*PandaBoard ES*) auf ihre Eignung für den Einsatz mit *Fail** untersucht und experimentell bewertet. Der Schwerpunkt der Untersuchungen liegt hierbei auf der Verwendung von Debugschnittstellen und der Bewältigung von gegebenen Einschränkungen bezüglich Durchführungsgeschwindigkeit und Hardwarezugriff. Die Bewertung von Effizienz und Aussagekraft erfolgt auf Grundlage von FI-Kampagnen und Microbenchmarks.

Inhaltsverzeichnis

1 Einführung und Motivation 1

2 Grundlagen 5

2.1 Grundbegriffe der Fehlerinjektion . . . 5
2.2 Fehlermodelle . . . 9
2.3 Zuverlässigkeitsmetriken . . . 10
2.4 Fehlerinjektionsverfahren . . . 11
2.4.1 Hardware-basiert . . . 13
2.4.2 Software-basiert . . . 14
2.4.3 Simulator-basiert . . . 14
2.4.4 OCD-basiert . . . 15
2.5 Optimierung von Fehlerinjektionsexperimenten . . . 16
2.6 Das Fehlerinjektions-Framework Fail* . . . 18
2.6.1 Backend-Abstraktion . . . 19
2.6.2 Kampagnenverwaltung . . . 23
2.7 Zusammenfassung . . . 26

3 Analyse und Entwurf 27

3.1 Anforderungsanalyse . . . 27
3.1.1 Zielplattform . . . 29
3.1.2 Systemarchitektur . . . 30
3.2 Gesamtentwurf . . . 31
3.2.1 Pandaboard ES . . . 31
3.2.2 Flyswatter 2 und OpenOCD . . . 33
3.3 Trace-Aufzeichnung . . . 34
3.4 Initialisierung . . . 36
3.5 Trace-Navigation . . . 37
3.5.1 Stand der Kunst . . . 38
3.5.2 Smart-Hopping . . . 40
3.5.3 Checkpointing . . . 46
3.5.4 Äquivalenzklassengewahre Navigation . . . 47
3.6 Fehlerinjektion . . . 48
3.7 Nachuntersuchung . . . 49
3.8 Zusammenfassung . . . 56

4 Realisierung **57**
4.1 Modifikation der Zielplattform . . . 57
4.2 Anbindung von OpenOCD . . . 59
4.3 Integration in Fail* . . . 63
4.4 Zusammenfassung . . . 66

5 Evaluation **67**
5.1 Performanz . . . 67
5.1.1 Schnittstelle . . . 68
5.1.2 Trace-Navigation . . . 70
5.1.3 MMU-Speicherüberwachung . . . 78
5.1.4 Gesamtsystem . . . 80
5.2 Experimentergebnisse . . . 83
5.3 Zusammenfassung . . . 86

6 Ergebnisse und Ausblick **87**
6.1 Ergebnisse . . . 87
6.2 Ausblick . . . 88

Literaturverzeichnis **91**

Abbildungsverzeichnis **97**

Tabellenverzeichnis **99**

1 Einführung und Motivation

Dem Bereich mobiler hochperformanter eingebetteter Computersysteme im Markt der Verbraucherelektronik steht ein Wandel bevor. In diesem Bereich ist eine Entwicklung mit den Optimierungskriterien Performanz, Energieeffizienz und Preis für den Endverbraucher zu erkennen. In den kommenden Jahren wird die konsequente Erhöhung der Strukturdichte zur Steigerung der Performanz sowie die Reduzierung der Versorgungsspannung zur Reduktion des Energieverbrauchs ein erhöhtes Aufkommen an Hardware-Fehlern – beispielsweise Invertierungen von Speicherbits durch Strahlungseinfall – in angesprochenen Systemen hervorrufen [Bor05, DYDS$^+$10, NX06]. Üblicherweise wird einem solchen Aufkommen von Hardware-Fehlern mithilfe von Hardware-Fehlertoleranz begegnet – beispielsweise mittels Speicher mit eingebauter ECC-Einheit. Aufgrund des hohen Preisdrucks im Bereich der Verbraucherelektronik und des verursachten zusätzlichen Energieverbrauchs [GBT04] widerspricht der Einsatz fehlertoleranter Hardware allerdings teilweise den oben genannten Optimierungszielen.

Ein alternativer Ansatz liegt in der Software-Fehlertoleranz. Hierbei werden Systemanteile durch die Einfügung zusätzlichen Codes abgesichert. Beispielsweise kann das Verfahren der *Triple Modular Redundancy* (kurz: TMR) betrachtet werden, welches eine abzusichernde Funktion dreifach ausführt und anschließend das Ergebnis durch einen Mehrheitsentscheid der drei Einzelergebnisse bestimmt [KK07]. Somit kann ein Fehler in einer der drei Ausführungen auftreten, ohne weitere Auswirkungen auf das Funktionsergebnis zu haben. Es ist ebenfalls möglich, Software-Fehlertoleranz gezielt auf bestimmte Variablen anzuwenden, indem etwa bei jedem Zugriff auf die Variable eine Kontrolle einer Prüfsumme durchgeführt wird [BSS13a]. Der Einsatz dieser Verfahren erfordert keine zusätzlichen Hardware-Komponenten und kann zielgerichtet an den Systemanteilen wirken, deren Absicherung zwingend erforderlich ist, wobei an den restlichen Anteilen kein oder ein schwächeres Toleranzverfahren genutzt wird. Durch diese hohe Konfigurierbarkeit können der durch Fehlertoleranz verursachte Mehraufwand und somit Performanz- und Energieeffizienz-Einbußen minimiert werden.

Im Folgenden wird beispielhaft ein mobiles Video-Abspielgerät mit Einsatz von Software-Fehlertoleranz betrachtet [HESM10]. Hierbei können Hardware-Fehler auftreten, die einen Systemabsturz oder eine verfälschte Darstellung der Bildinformation des Videos verursachen können. Durch die hohe Konfigurierbarkeit von Software-Fehlertoleranz ist es möglich, nur die Systemanteile abzusichern, bei denen das fehlerhafte Verhalten für den Nutzer sichtbar wird. So werden Komponenten abgesichert, in denen auftretende Fehler zu einem Absturz führen können und Anteile des Bildspeichers, deren fehlerhafte Darstellung erkennbar ist. Eine große Menge an auftretenden Fehlern kann hingegen toleriert werden, da beispielsweise leichte Farbänderungen im Bild für den Nutzer nicht erkennbar sind. Weil die abzusichernden Komponenten im Regelfall einen kleinen An-

teil am gesamten Systemzustand ausmachen, wird hierdurch ein geringer Mehraufwand erzeugt.

Software-Fehlertoleranz entwickelt sich aufgrund seiner Vorteile auch zu einem angesehenen Verfahren in Bereichen, in denen Aufgrund hoher Anforderungen an die Ausfallsicherheit bislang ausschließlich Hardware-Fehlertoleranz eingesetzt wurde. Beispielhaft ist das Projekt des NASA SpaceCube zu nennen [WZF13]. Hier wird kommerziell erhältliche Hardware außerhalb der Erdatmosphäre verwendet, um hohe Rechenperformanz nutzen zu können. Alternative Plattformen, die Hardware-Fehlertoleranz für den Einsatz in dieser Domäne bieten, sind im Regelfall aufgrund langer Entwicklungszyklen weniger aktuell, weshalb ihre Berechnungsleistung deutlich reduziert ist. Da die starke Strahlung im Einsatzszenario des Space-Cubes ein erhöhtes Fehleraufkommen produziert und eine Wartung der Hardware sehr schwierig ist, erfolgt eine entsprechende Absicherung mittels Software-Fehlertoleranz.

Zur Bemessung der Wirksamkeit von auf ein System angewandten Software-Fehlertoleranzverfahren und zur Identifikation von Systemanteilen, deren Absicherung nötig ist, ist der Einsatz von entsprechenden Messwerkzeugen erforderlich. Ein mögliches Messverfahren ist die sogenannte *Fehlerinjektion*. Hierbei werden systematisch Fehler in ein zu untersuchendes System injiziert und die Auswirkungen der Injektion auf das anschließende Systemverhalten untersucht. Auf Basis dieser Untersuchungen sind statistische Aussagen über die Zuverlässigkeit des Systems möglich. Des Weiteren kann bei genauer Untersuchung der Auswirkungen von Fehlern in verschiedenen Systemanteilen eine spezifische Konfiguration von Software-Fehlertoleranzverfahren entworfen werden. Eine mögliche Realisierung des Verfahrens liegt in der Injektion von Fehlern in Hardware-Simulatoren, wodurch eine günstige Parallelisierung von Injektionsexperimenten und ein leichter Zugriff auf die simulierten Komponenten möglich ist. Im Gegensatz dazu hat eine Injektion in ein echtes Hardwaresystem den Vorteil, dass die Ergebnisse realitätsgetreuer als bei der Verwendung von Simulatoren sind [BGOS12]. Fehlerinjektion in Hardware ist schlecht parallelisierbar, da für jede zusätzliche Instanz des zu untersuchenden Systems zusätzliche Hardware vorhanden sein muss. Dieser Nachteil kann allerdings durch die im Gegensatz zum Simulator im Regelfall deutlich höhere Ausführungsgeschwindigkeit kompensiert werden.

Die Injektion in Hardwaresysteme kann beispielsweise mittels sogenannter *On-Chip Debugger* (kurz: OCD) erfolgen [FGAF06]. Hierbei handelt es sich um in Prozessoren integrierte Hardwarebausteine, die eine Untersuchung des Systemzustands erlauben, um beispielsweise Programmfehler zu erkennen. Zu diesem Zweck wird das Anhalten der Ausführung des Prozessors mithilfe konfigurierbarer Haltebedingungen – sogenannte *Breakpoints* und *Watchpoints* – ermöglicht und es kann der aktuelle Zustand des Systems untersucht und modifiziert werden. Konkret besteht üblicherweise die Möglichkeit zum lesenden und schreibenden Zugriff sowohl auf die Prozessorregister als auch auf den Systemspeicher. Durch die von einer OCD-Komponente bereitgestellte Infrastruktur ist demnach auch eine Injektion von Fehlern in das System möglich.

Ziel dieser Arbeit ist die Entwicklung und Untersuchung eines Fehlerinjektionssystems auf Basis von günstiger sogenannter *Commercial-Off-The-Shelf*-Hardware (kurz: COTS). Durch die Nutzung dieser Hardware ist es für Dritte bei geringem Aufwand

möglich, Forschungsergebnisse nachzuvollziehen und das System zu nutzen und zu erweitern. Es wird daher ein OCD-fähiges Hardware-Entwicklungsboard mit integriertem Prozessor, der üblicherweise in mobiler hochperformanter Verbraucherelektronik verwendet wird, an das bestehende Fehlerinjektions-Framework *Fail** (siehe [SHK+12]) angebunden. Im Kontext dieser Arbeit erhält das zu entwickelnde System den Projektnamen *FailPanda*. Das Framework Fail* bietet bereits die zur Fehlerinjektion nötige Infrastruktur und eine generische Schnittstelle für neue Zielsysteme. Es werden hierzu die zur Fehlerinjektion nötigen Einzelfunktionen implementiert, wobei entstehende Probleme und mögliche Lösungen untersucht werden. Das System wird insbesondere auf Probleme der Performanz untersucht und diese werden mittels geeigneter Verfahren abgeschwächt, sodass die praktische Einsetzbarkeit gezeigt werden kann.

Die Arbeit ist wie folgt gegliedert. Kapitel 2 erläutert zunächst Grundlagen der Fehlerinjektion und stellt das Fehlerinjektions-Framework Fail* vor. Anschließend wird in Kapitel 3 der im Zuge der Arbeit verfolgte Systementwurf auf Basis einer Anforderungsanalyse vorgestellt. Kapitel 4 schildert die Implementierung des Entwurfs und die Integration der entwickelten Verfahren in das Fehlerinjektions-Framework Fail*. Das Gesamtsystem wird in Kapitel 5 hinsichtlich seiner Performanz evaluiert und es wird eine beispielhafte Generierung und Untersuchung von Fehlerinjektionsergebnissen gezeigt. Abschließend werden die Ergebnisse der Arbeit in Kapitel 6 zusammengefasst und es wird ein Ausblick auf mögliche Nachfolgearbeiten und Erweiterungen gegeben.

2 Grundlagen

Dieses Kapitel erläutert die in dieser Arbeit verwendeten Grundlagen nach aktuellem Stand der Forschung. Insbesondere wird auf Grundlagen der Fehlerinjektion eingegangen. In Abschnitt 2.1 werden zunächst die Grundbegriffe der Fehlerinjektion geklärt. Anschließend diskutiert Abschnitt 2.2 Fehlermodelle, die dem Verfahren zugrunde gelegt werden. In Abschnitt 2.3 werden Metriken vorgestellt, die statistische Aussagen über die Zuverlässigkeit eines Systems zulassen. Anschließend zeigt Abschnitt 2.4 einen Vergleich verschiedener Fehlerinjektionsverfahren. In Abschnitt 2.5 wird ein Verfahren zur Optimierung von Fehlerinjektionsexperimenten erläutert und schließlich wird in Abschnitt 2.6 das Fehlerinjektions-Framework Fail* vorgestellt, das in dieser Arbeit verwendet wird.

2.1 Grundbegriffe der Fehlerinjektion

Der deutsche Begriff *Fehler* hat in diesem Zusammenhang mehrere Bedeutungen, die im Folgenden mithilfe der drei englischen Begriffe *Fault*, *Error* und *Failure* erläutert werden [KK07]. Abbildung 2.1 zeigt, dass ein Fault nach einer Latenz in einen Error umgewandelt und dieser wiederum nach einer weiteren Latenz zu einem Failure werden kann. Ein Fault repräsentiert das nicht vorhergesehene Verhalten einer Komponente, wobei dies sowohl eine Software- also auch Hardwarekomponente betreffen kann. Da im Kontext dieser Arbeit ausschließlich Fehler in Hardware-Komponenten untersucht werden, wird hier auf Software-Faults nicht weiter eingegangen. Ein beispielhafter Fault liegt in dem fehlerhaften Verhalten einer Speicherzelle aufgrund von externer hochenergetischer Einstrahlung [DNR02]. Solange dieses Verhalten keine Auswirkung auf den Systemzustand hat, bleibt der Fault ohne Auswirkung. Wird jedoch durch das fehlerhafte Verhalten beispielsweise ein Variablenwert ungewollt verändert, so wird aus einem Fault ein sogenannter Error. Errors können innerhalb des Systems propagieren, indem fehlerhafte Werte weiterverwendet werden. Führt ein Error zu nach außen sichtbarem fehlerhaftem Verhalten, so wird er zu einem Failure. Ein Beispiel für einen Failure ist ein fehlerhaftes Berechnungsergebnis oder ein Systemabsturz. Um im Folgenden eine deutsche Terminologie fortzuführen, wird ein Error als Fehler und ein Failure als Auswirkung eines Fehlers

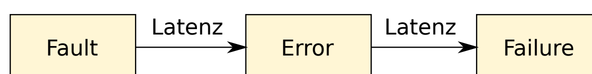


Abbildung 2.1: Darstellung der Umwandlungsreihenfolge von Faults, Errors und Failures nach [BP03]. Es wird gezeigt, dass die Umwandlungsvorgänge nach einer Latenz aktiviert werden.

bezeichnet. Bei Beschreibungen physikalischer Vorgänge wird jedoch der Terminus Fault verwendet.

Fehlerinjektion ist die Simulation eines vorgegebenen Fehlermodells. Es steht im Gegensatz zu dem sogenannten *Life Testing,* wobei ein System im Realszenario beobachtet wird und die Fehler natürlich auftreten [BP03]. Das Verfahren des Life Testings benötigt im Allgemeinen deutlich mehr Zeit zur Aquirierung einer statistisch genügenden Menge von Ergebnissen als die Fehlerinjektion. Allerdings liegt der Vorteil darin, dass auf kein möglicherweise nicht hinreichendes Fehlermodell zurückgegriffen werden muss, da das Fehleraufkommen seinen statistischen Charakter implizit selbst definiert. Die Fehlerinjektion simuliert auf verfahrensabhängige Art und Weise (s. Abschnitt 2.4) das Auftreten von Fehlern, indem direkt auf die entsprechenden Komponenten eingewirkt wird.

Bei diesem Verfahren wird im Allgemeinen ein sogenannter Fehlerraum untersucht, welcher gültige Tupel aus Injektionszeitpunkt und -Ort repräsentiert und diesen eine Reaktion des Systems auf den entsprechenden Fehler zuordnet [BP03]. Die konkrete Ausprägung dieser Werte ist abhängig vom Injektionsverfahren und vom vorausgesetzten Fehlermodell. So wird im Verlauf dieser Ausarbeitung der Injektionszeitpunkt beispielsweise häufig als Position in einer Programmaufzeichnung, im Folgenden *Trace* genannt, definiert. Diese Position wird als *dynamische Instruktion* bezeichnet. Eine dynamische Instruktion ist eine konkrete Ausführung einer sogenannten *statischen Instruktion.* Während statische Instruktionen allein durch ihre Position im Programmcode beschrieben werden können (Instruktionsadresse), sind dynamische Instruktionen demnach die Ausführung dieser Instruktion zu einem bestimmten Zeitpunkt. Der Injektionsort kann beispielsweise ein Bit eines Speicherwortes repräsentieren.

Grundsätzlich wird davon ausgegangen, dass eine vollständige Abdeckung des Fehlerraums aufgrund der potenziell sehr hohen Anzahl nötiger Experimente nicht praktikabel durchführbar ist, weshalb im praktischen Einsatz sogenanntes *Sampling* verwendet wird [BVFK05, BP03]. Mithilfe dieses Verfahrens werden Experimente für eine repräsentative Untermenge des Fehlerraums durchgeführt, indem gleichverteilt randomisiert Experimente gewählt werden, bis eine Experimentmenge erreicht wurde, die mit dem konkreten Verfahren in praktikabler Zeit durchführbar ist[1].

Auf Basis der bekannten Parameter von Ort und Zeit wird die entsprechende Reaktion des Systems bemessen, um den Fehlerraum zu bestimmen. Aus einem solchen Ergebnis können direkt Schlüsse für den Entwurf von Fehlertoleranzverfahren gezogen werden, indem Häufungen von Fehlern gefunden werden [BSS13b]. Durch die Anwendung entsprechender Metriken kann das Fehlerverhalten quantifiziert werden, um einen Vergleich verschiedener Systeme durchführen zu können (s. Abschnitt 2.3).

Eine wichtige Grundannahme bei Fehlerinjektion ist der Determinismus des Zielsystems [SBK10]. Es wird davon ausgegangen, dass das System, solange keine Injektion erfolgt ist, sich in jeder Ausführung exakt gleich verhält. Wird ein System getestet, welches mit externen oder internen Quellen nichtdeterministischer Daten arbeitet (bei-

[1] Eine weitere Einschränkung der Experimentmenge wird durch eine in Abschnitt 2.5 vorgestellte Technik erreicht, welche mehrere Fehlerinjektionsexperimente zu Äquivalenzklassen zusammenfasst, da a priori bekannt ist, dass die Ergebnisse aller Experimente einer solchen Klasse das gleiche Ergebnis produzieren werden.

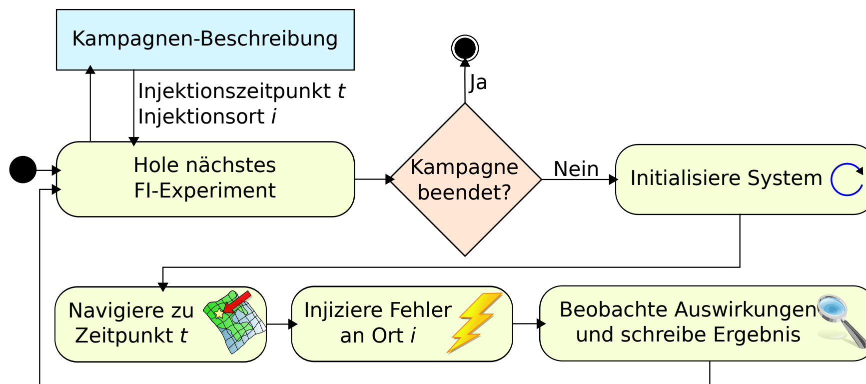


Abbildung 2.2: Schematische Darstellung des Ablaufs einer generischen Fehlerinjektionskampagne. Zur vereinfachten Darstellung wird hier nur von Einzelfehlern ausgegangen.

spielsweise Werte externer Sensoren), so müssen diese Quellen zunächst durch simulierte Quellen ersetzt werden, die typisches Verhalten deterministisch nachahmen [BP03].

Ist im Zielsystem der Determinismus erfolgreich hergestellt worden, so wird zunächst ohne Injektion von Fehlern eine Aufzeichnung eines fehlerfreien Programmablaufs durchgeführt, was als *Golden Run* bezeichnet wird. Wird das Verhalten der Plattform nach einer Fehlerinjektion mit dem Golden Run verglichen, so kann fehlerhaftes Verhalten identifiziert werden [BP03].

Mit einer Aufzeichnung des Programm-Trace ist es möglich, den Fehlerraum zu definieren. Hierbei werden üblicherweise die ausgeführten Instruktionen in der entsprechenden Ausführungsreihenfolge, ebenso wie die während der Programmaufzeichnung durchgeführten Zugriffe auf den Hauptspeicher, aufgezeichnet. Eine Einheit in der zeitlichen Dimension entspricht einer dynamischen Instruktion, welche durch ihre Position im Trace definiert ist. Die von der Zielapplikation verwendeten Ressourcen, wie beispielsweise die gelesenen Speicherwörter, ergeben kumulativ die Menge aller möglicher Injektionsorte. Soll ein Experiment bestimmen, ob das System nach einer Fehlerinjektion ein fehlerhaftes Ergebnis berechnet, so wird ein Vergleich mit dem Ergebnis aus dem Golden Run durchgeführt.

Die Zusammenfassung aller für die Abdeckung eines Fehlerraums nötigen Fehlerinjektionsexperimente wird nachfolgend als *Kampagne* bezeichnet [AVFK01]. Eine Kampagne umfasst eine Menge von einzelnen Fehlerinjektionsexperimenten, die den Fehlerraum in den Dimensionen *Zeit* und *Ort* definieren. Abbildung 2.2 schematisiert den Ablauf einer generischen Kampagne. Hierbei wurde zur einfacheren Darstellung die Einschränkung unternommen, dass in der Kampagne nur zu einem Zeitpunkt während der Ausführung ein Fehler injiziert wird. Der Ablauf einer Kampagne sieht vor, dass solange Parameter für das folgende Experiment von der Kampagne abgerufen werden, bis der Fehlerraum abgedeckt und die Kampagne beendet wurde.

Jedes Experiment hat den grundsätzlichen Aufbau, dass zunächst eine Systeminitialisierung erfolgt, welche von einer Ausführungsphase bis zum Injektionszeitpunkt t gefolgt

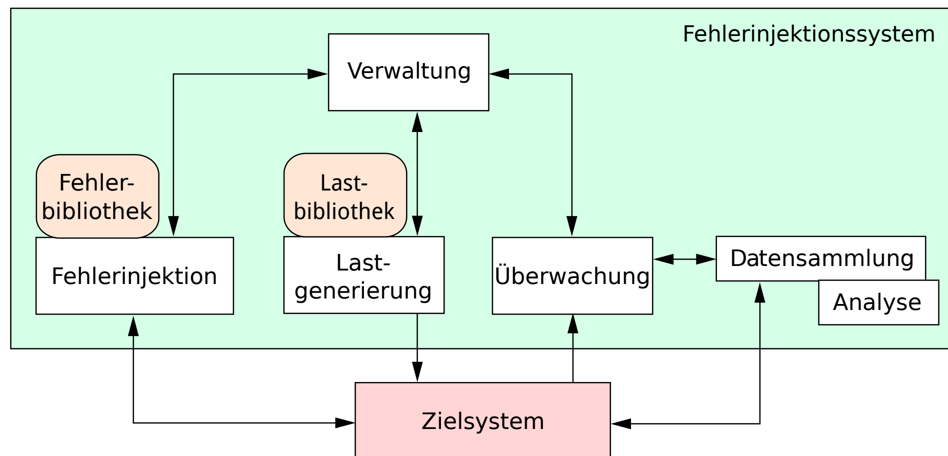


Abbildung 2.3: Allgemeine schematische Darstellung der Kampagnenverwaltung durch ein Fehlerinjektions-Framwork, übersetzt aus [HTI97].

wird. Diese Ausführungsphase wird nachfolgend als *Navigation* zum Injektionszeitpunkt bezeichnet. Anschließend wird am Injektionsort i ein Fehler injiziert und unter fortgesetzter Ausführung das Verhalten des Zielsystems untersucht. Gültige Ergebnisse können unterschiedlichster Form sein und sind im Kontext der jeweiligen Anwendung zu definieren. Beispielsweise können fehlerhafte Speicherzugriffe oder ausgelöste Prozessor-Traps zu möglichen Experimentergebnissen gehören. Das Ergebnis der Nachuntersuchung wird in der vorliegenden Repräsentation des Fehlerraums hinterlegt, um anschließend mit dem nächsten Experiment fortzufahren.

Die notwendige Verwaltung von derartigen Kampagnen wird von Fehlerinjektions-Frameworks bereitgestellt, die eine generische Definition von Experimenten ermöglichen [SBK10]. Derartige Frameworks bieten im Allgemeinen die in Abbildung 2.3 gezeigten Komponenten [HTI97]. Die Komponente der Fehlerinjektion führt zu den definierten Injektionszeitpunkten t eine Einfügung von Fehlern aus einer durch das Fehlermodell definierten Bibliothek in das Zielsystem durch. Gleichzeitig zur Ausführung des Zielsystems wird eine Last generiert, welche beispielsweise die Form der Ausführung eines Benchmarks oder der Übermittlung von definierten Eingaben haben kann. Die Last ist ebenfalls in Form einer Bibliothek vordefiniert. Eine Überwachungskomponente liest den aktuellen Zustand des Zielsystems aus und führt eine Sammlung von relevanten Zustandsdaten aus. Auf Basis der gesammelten Daten wird eine Analyse des Reaktionsverhaltens der Zielplattform auf die injizierten Fehler durchgeführt. Eine zentrale Verwaltung steuert schließlich die einzelnen Komponenten des Fehlerinjektionssystems. In Abschnitt 2.6 wird zur Erläuterung das Framework Fail* detailliert beschrieben, da im Kontext der Arbeit darauf aufgebaut wird.

2.2 Fehlermodelle

Bei der Fehlerinjektion werden Fehler unter einem gegebenen Modell simuliert. Dieses Modell beschreibt die Arten von Fehlern, von deren Auftreten ausgegangen wird und enthält eine statistische Beschreibung ihres Auftretens. Das Modell wird *a priori* definiert, indem Annahmen über die Fehleranfälligkeit der verschiedenen Systemkomponenten und eine statistische Beschreibung der durch die Ausführungsumgebung definierten Fehlerursachen berücksichtigt werden.

Grundsätzlich kann eine Einteilung in die Klassen der Hardware- und Software-Fehler durchgeführt werden. Während Hardware-Fehler beispielsweise durch Einstrahlung kosmischer Teilchen ausgelöst werden können, sind Software-Fehler die Folge fehlerhafter Programmierung [BP03]. Aus der Motivation dieser Arbeit geht hervor, dass im Kontext der Arbeit nur Hardware-Fehler betrachtet werden. Eine weitere Unterteilung in Fehlerklassen berücksichtigt *transiente*, *permanente* und *unregelmäßig aktive* Fehler [KK07]. Transiente Fehler werden auch als *soft errors* bezeichnet, da sie eine vorübergehende Störung des Systems, nicht jedoch den Defekt eines Bauteils beschreiben. Ein typischer transienter Fehler ist ein invertiertes Bit im Speicher, ausgelöst durch Strahlungseinfall [DNR02]. Wird die Speicherzelle nach Auftreten des Fehlers beschrieben, so ist der Fehler eliminiert. Permanente Fehler bezeichnen den Defekt eines Bauteils. So würde beispielsweise ein Bit im Speicher nach Auftreten eines solchen Fehlers aufgrund eines physischen Defekts in der Hardware immer eine logische Null darstellen. Schließlich bezeichnen unregelmäßig aktive Fehler einen dauerhaften Fehlerzustand wie bei den permanenten Fehlern, welcher allerdings nur zeitweise aktiv ist. Ein betroffenes Speicherbit würde beispielsweise zeitweise unregelmäßig im inaktiven Zustand des Fehlers eine korrekte Funktion aufweisen und bei Aktivität des Fehlers dauerhaft einen falschen Wert repräsentieren.

Da Hardware-Fehler je nach Injektionsverfahren auf verschiedenen Ebenen eines Systems wahrgenommen und injiziert werden können, wird im Fehlermodell spezifiziert, auf welcher Ebene die Untersuchung stattfindet. So kann das Fehlermodell beispielsweise Fehler auf Ebene von logischen Werten auf den Übertragungsleitungen oder auf Ebene von physikalischen Vorgängen innerhalb von Halbleitern eines Systems modellieren. In einer abstrakteren Betrachtung ist es möglich, Fehler auf Bit-Ebene in Speicher oder Registern vorzusehen [BP03]. Die Wahl eines Injektionsverfahrens (s. Abschnitt 2.4) definiert implizit die Ebene der möglichen auftretenden Fehler, wodurch wiederum das Fehlermodell eingeschränkt wird. Im Kontext dieser Arbeit werden ausschließlich Fehler auf Bit-Ebene betrachtet, sodass auf Instruktionsebene eine Modifikation der Werte vorgenommen werden kann.

Das Fehlermodell bestimmt direkt die Form der Fehlerinjektion. So wird beispielsweise ein Experiment unter der Annahme von Einzelfehlern das Zielsystem in den Zustand des Fehlerzeitpunktes bringen, einen Fehler an dem untersuchten Fehlerort injizieren und das System weiter ausführen lassen und dabei auf fehlerhaftes Verhalten untersuchen (s. Abb. 2.2). Im Gegensatz dazu müsste unter der Voraussetzung, dass Fehler in sogenannten *Bursts* auftreten, gleichzeitig an mehreren benachbarten Fehlerorten injiziert

werden [HSR95]. Würde das Modell einen zeitlichen Burst vorsehen, so würde ab der ersten Fehlerinjektion das Zielsystem mehrfach an aufeinander folgenden Instruktionen angehalten, um weitere Fehler injiziert werden.

Mathematische Beschreibungen für das Auftreten von Hardware-Fehlern sind mithilfe von statistischen Modellen möglich. Häufig werden zu diesem Zweck *Poisson-Prozesse* eingesetzt [KK07]. Sie beschreiben allgemein die Auftretenswahrscheinlichkeiten von nichtdeterministischen Ereignissen in einem bestimmten Zeitraum. Somit kann mit einer entsprechend parametrisierten Poisson-Verteilung die Wahrscheinlichkeit für das Auftreten von Fehlern modelliert werden. Die Verteilung gibt für einen angegebenen Zeitraum und eine angegebene Anzahl erwarteter Fehler einen Wahrscheinlichkeitswert an. Häufig fällt die Verteilung für steigende Fehlerzahlen sehr stark ab, sodass die Wahrscheinlichkeiten für mehr als einen Fehler im betrachteten Zeitraum mit dem Wert Null angenommen werden. Durch diese Annahme lässt sich begründen, dass Fehlerinjektionskampagnen, die nur Einzelfehler betrachten, eine gute Annäherung an die Realität bieten.

2.3 Zuverlässigkeitsmetriken

Um eine Auswertung der Ergebnisse einer Fehlerinjektions-Kampagne durchführen zu können, existieren Metriken, durch deren Anwendung Aussagen über verschiedene Eigenschaften des Zielsystems getroffen werden können. Durch Verwendung geeigneter Metriken ist der Vergleich zweier Systeme möglich, sodass beispielsweise ein System ohne Einsatz von Fehlertoleranz mit der entsprechenden um eine Fehlertoleranzmaßnahme erweiterten Version verglichen werden kann. Im Folgenden werden einige typische Metriken und deren jeweilige Aussage über das getestete System vorgestellt.

Da Fehlerinjektion üblicherweise eingesetzt wird, um die Zuverlässigkeit (engl.: *Reliability*) eines Systems unter der Annahme des Auftretens von Fehlern zu bestimmen, ist diese Metrik als zentral anzusehen. Die Zuverlässigkeit R ist als

$$R = 1 - P\left\{\text{Failure}\right\}$$

definiert. Das bedeutet, dass die Zuverlässigkeit die Gegenwahrscheinlichkeit zur Wahrscheinlichkeit eines nach außen sichtbaren Fehlers unter einem gegebenen Fehlermodell ist [AAA+90]. Durch diese Definition erweitert sich die Formel nach den zuvor beschriebenen englischen Fehler-Terminologien zu

$$R = 1 - \left(P\left\{\text{Failure}|\text{Fault}\right\} \cdot P\left\{\text{Fault}\right\}\right)$$

[AAA+90]. Die Wahrscheinlichkeit $P\{\text{Fault}\}$ wird durch das Fehlermodell definiert und ist a priori bekannt. $P\{\text{Failure}|\text{Fault}\}$ ist die Wahrscheinlichkeit für das nach außen sichtbare fehlerhafte Verhalten des Systems unter der Voraussetzung, dass ein Fehler nach dem definierten Modell injiziert wurde. Diese Wahrscheinlichkeit ist unter Zuhilfenahme von Fehlerinjektion empirisch zu bestimmen. Zu diesem Zweck wird die Gesamtmenge aller Fehlerinjektionsexperimente durch Division in ein Verhältnis zu der

Menge an Experimenten mit fehlerhaftem Ausgang gesetzt. Es wird demnach der Anteil an Experimenten bemessen, welcher von einem auf dem System operierenden Fehlertoleranzverfahren nicht beseitigt werden konnte. Wird diese Metrik verwendet, um zwei verschiedene Fehlertoleranz-Konfigurationen eines Systems zu vergleichen, so ergeben sich allerdings die nachfolgend erläuterten Probleme.

Bei der Erweiterung einer Applikation um eine Fehlertoleranzmaßnahme zeigt sich der Nebeneffekt, dass zusätzliche Ressourcen verwendet werden. So wird üblicherweise die Laufzeit des Programms verlängert und es werden weitere Speicherbereiche verwendet. Hierdurch vergrößert sich der Fehlerraum, dessen Größe in $P\{\text{Failure}|\text{Fault}\}$ einfließt. Somit kann der Wert von R vergrößert werden, indem ein Verfahren eingebracht wird, welches lediglich eine Vergrößerung des Fehlerraums erwirkt, ohne tatsächlich Fehler abzufangen. Beispielsweise ist ein Verfahren vorstellbar, welches einige `nop`-Instruktionen in einen Benchmark einbringt. Der Vergleich der Werte von R für das Programm ohne und mit eingebrachtem Verfahren würde eine Verbesserung zeigen, obwohl tatsächlich keine Fehlerbehandlung durchgeführt wird.

Es konnte somit gezeigt werden, dass R zwar für absolute Aussagen zur Zuverlässigkeit verwendbar ist, jedoch für den Vergleich von Fehlertoleranzmaßnahmen unbrauchbar ist. Um einen quantitativen Vergleich der Zuverlässigkeit zweier Varianten eines Zielsystems durchzuführen, sind absolute Anzahlen der Failures zu verwenden [BSS13a]. Hierbei kann das Ergebnis nur verbessert werden, indem auftretende Fehler von einer Fehlertoleranzmaßnahme erfolgreich behandelt werden.

Für die statistische Quantisierung der Eigenschaften eines untersuchten Systems existieren noch weitere Metriken, die allerdings im Kontext dieser Arbeit keine Verwendung finden. Daher werden wichtige Metriken an dieser Stelle nur kurz genannt. Die *Verfügbarkeit* (engl.: *Availability*) bezeichnet die Wahrscheinlichkeit dafür, dass ein System zu einem Zeitpunkt t zur Verwendung bereit steht, d.h. korrekt funktioniert und auf Anfragen reagiert [BP03]. Mit der Metrik der Sicherheit (engl.: *Safety*) wird die Wahrscheinlichkeit beschrieben, mit der ein System entweder im Zustand der korrekten Ausführung ist, oder alternativ seine Funktion einstellt, um eine Propagation eines Fehlers durch das Gesamtsystem zu verhindern. Dieses Verhalten wird auch als *fail-stop behaviour* bezeichnet [BSS13b]. Die mittlere Zeit bis zum Auftreten eines Fehlers (engl.: *Mean Time To Failure* (kurz: *MTTF*)) bezeichnet für dauerhaft operierende Systeme die mittlere Zeitspanne, bis zu der ein fehlerhaftes Verhalten des Systems unter Einwirkung von Fehlern erwartet wird.

2.4 Fehlerinjektionsverfahren

Dieser Abschnitt vermittelt einen Überblick über die verschiedenen existierenden Verfahren zur Injektion von Hardware-Fehlern in ein Zielsystem und grenzt diese voneinander ab. Die Verfahren werden anhand folgender Aspekte, angelehnt an die Untersuchungen in [PGLOVE11], verglichen:

Tabelle 2.1: Vergleich der im Kontext dieser Arbeit betrachteten Fehlerinjektionsverfahren.

	Hardware-basiert	Software-basiert	Simulator-basiert	OCD-basiert
Änderung des Zielsystems	keine	hoch	keine	gering
Geschwindigkeit	hoch	hoch	gering	hoch
Parallelisierbarkeit	gering	gering	hoch	gering
Kosten	sehr hoch	gering	gering	hoch
Fehlermodell	grundsätzlich unbeschränkt	nur per Software zugreifbare Komponenten	nur simulierte Komponenten	nur per OCD zugreifbare Komponeten

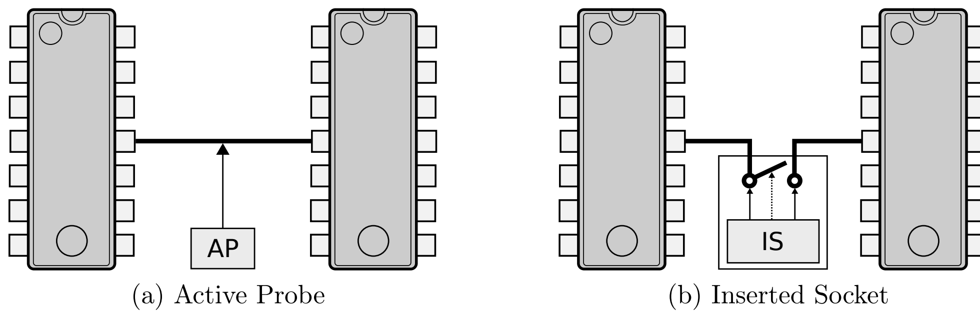
- **Nötige Veränderung des Zielsystems:** Je geringer die nötigen Veränderungen des Zielsystems für die Anwendung des betrachteten Verfahrens sind, umso höher ist die Aussagekraft der Ergebnisse. Ist für das Injektionsverfahren beispielsweise ein Eingriff in den Programmcode nötig, so wird das Verhalten des Zielsystems unter Umständen erheblich verändert, wodurch die Aussagekraft der Ergebnisse sinkt.

- **Geschwindigkeit:** Aufgrund der typischerweise hohen Größe des Fehlerraums müssen Einzelexperimente möglichst schnell durchgeführt werden können, damit eine komplette Abdeckung des Fehlerraums in praktikabler Zeit ermöglicht ist.

- **Parallelisierbarkeit:** Um mögliche Defizite in der Ausführungsgeschwindigkeit zu kompensieren, kann es wichtig sein, mehrere Fehlerinjektionsexperimente parallel auszuführen. Aus diesem Grund wird auch die Möglichkeit zur Parallelisierung der Verfahren untersucht.

- **Kosten:** Die betrachteten Injektionsmaßnahmen generieren unterschiedliche finanzielle und entwicklungsspezifische Kosten. Je geringer die Kosten für den Einsatz eines Verfahrens sind, umso geringer ist auch die Einstiegshürde für potenzielle Anwender.

- **Fehlermodell:** Verschiedene Fehlerinjektionsverfahren schränken das Fehlermodell bezüglich der fehlersensitiven Komponenten des Zielsystems und der Art von injizierbaren Fehlern ein. Da diese Einschränkung bei der Wahl eines Verfahrens aufgrund des a priori definierten Fehlermodells Verfahren disqualifizieren kann, wird ebenfalls anhand dieses Kriteriums verglichen.



(a) Active Probe (b) Inserted Socket

Abbildung 2.4: Schematische Darstellung der Funktionsweise von Active Probes und Inserted Sockets nach [BP03].

Eine Übersicht zur Einordnung der nachfolgend betrachteten Verfahren wird in Tabelle 2.1 gezeigt. Einzelne Komponenten der Tabelle werden in den jeweiligen Untersuchungen erläutert.

2.4.1 Hardware-basiert

Bei Hardware-basierter Fehlerinjektion handelt es sich grundsätzlich um ein Verfahren, welches eine zusätzliche Hardwarekomponente dazu einsetzt, das ansonsten unveränderte Zielsystem zu stören [HTI97]. Hierbei wird zwischen kontaktbehafteter und kontaktloser Injektion unterschieden. Bei der kontaktbehafteten Fehlerinjektion werden üblicherweise auf Ebene der Ein-/Ausgabe-Kontakte des zu untersuchenden Chips mit sogenannten *Active Probes* oder *Inserted Sockets* fehlerhafte Zustände eingestreut [BP03]. Mittels dieser Schnittstelle können Fehler gezielt an einigen internen Komponenten des Zielsystems injiziert werden und somit sind potenziell Fehlerorte möglich, die per Software nicht zugreifbar sind. Abbildung 2.4 zeigt schematisch den Aufbau bei Einsatz der Verfahren. Active Probes (s. Abb. 2.4a) werden dazu eingesetzt, fehlerhafte Ströme an zugreifbaren Übertragungsleitungen anzulegen, um Veränderungen der logischen Zustände an den Kontakten herbeizuführen. Active Probes können nicht auf den augenblicklichen Zustand des Ein-/Ausgabe-Kontakts reagieren. Inserted Sockets (s. Abb. 2.4b) werden hingegen als Brücke zwischen die Chip-Kontakte und die angeschlossene Hardware geschaltet und können so komplexere logische Veränderungen wie Invertierungen der Signale durchführen.

Im Gegensatz zu den kontaktbehafteten Verfahren simulieren kontaktlose Verfahren die realen Ursachen für Fehler wie beispielsweise Strahlungspartikel oder elektrische Felder. Das zu untersuchende System wird der Fehlerquelle ausgesetzt, welche eine realitätsgetreue Umgebung, wie beispielsweise den Einsatz im Weltraum, simuliert. Hierbei wird die Auftretenswahrscheinlichkeit von Fehlern durch Intensivierung der Bedingungen erhöht, um im Gegensatz zum Life Testing schneller eine statistisch ausreichende Datenmenge generieren zu können [BP03]. Bei dieser Technik kann allerdings weder Ort noch Zeit der auftretenden Fehler exakt gesteuert werden, wodurch sich das Verfahren nur für statistische Aussagen über den Betrieb unter den simulierten Bedingungen

eignet. Es können keine Untersuchungen der Fehleranfälligkeit einzelner Komponenten durchgeführt werden.

Aufgrund der Durchführung auf der echten Hardware ohne ein nötiges Anhalten des Systems ist dieses Verfahren sehr schnell. Die Hardware-Fehlerinjektion setzt keine Modifikation des Programmcodes voraus, wodurch die Ergebnisse eine sehr hohe Aussagekraft haben. Da die parallele Ausführung von Experimenten pro Ausführungseinheit sowohl ein Exemplar der Zielhardware als auch die gesamte Injektions-Infrastruktur erfordert, ist die Parallelisierbarkeit stark beschränkt. Die Kosten des Verfahrens sind durch den Einsatz von spezialisierter Injektions-Hardware als hoch zu betrachten. Hardware-Fehlerinjektion ist für den Einsatz unter Fehlermodellen geeignet, die keine konkrete Vorgabe zu Ort oder Art von möglichen Fehlern treffen, da das Fehlermodell implizit durch die Kombination aus realitätsnaher Umgebung und echter Hardware definiert ist.

2.4.2 Software-basiert

Bei Software-basierter Fehlerinjektion handelt es sich um ein Verfahren, welches eine Fehlerinjektion auf Code-Ebene durchführt [HTI97]. Es lässt sich eine Einteilung in die Klassen der *Injektion zur Kompilierungszeit* und der *Injektion zur Laufzeit* durchführen. Bei der Fehlerinjektion zur Kompilierungszeit werden vor Ausführung des Programms Fehler in den Code eingestreut. Somit lassen sich nur Fehler simulieren, die seit Anfang der Programmausführung bestehen, wodurch das Verfahren stark beschränkt ist. Bei der Injektion zur Laufzeit wird hingegen zusätzlicher Code eingefügt, welcher eine Fehlerinjektion zum spezifizierten Zeitpunkt durchführt. Der Injektionszeitpunkt kann mittels Zeitmessung, Ausnahmebehandlungen oder direkte Injektion der Fehlerinjektions-Routine an die Zielinstruktion erreicht werden. Es kann aber beispielsweise auch ein Verfahren gewählt werden, bei dem ein zusätzlicher Programm-Faden zufällige Fehler einstreut, wenn ein der kontaktlosen Hardware-Fehlerinjektion ähnliches Verfahren simuliert werden soll [HESM10].

Software-Fehlerinjektion ist ein kostengünstiges Verfahren, da auf zusätzliche Hardware verzichtet wird und die Entwicklungszeit für das Injektionsverfahren als gering eingestuft werden kann. Weil das Zielsystem bei diesem Verfahren durch die Modifikation des Programmcodes stark verändert wird, ist die Aussagekraft der Ergebnisse allerdings beschränkt. Die Injektionsorte sind im Gegensatz zur Hardware-Fehlerinjektion auf die per Software zugreifbaren Komponenten beschränkt. Verwendbare Fehlermodelle sehen invertierte Bits in Speicher und Registern vor. Da die Software-Fehlerinjektion auf der unveränderten Zielplattform operiert, ist die Ausführungsgeschwindigkeit wie im Fall der Hardware-Fehlerinjektion hoch und die Parallelisierbarkeit gering.

2.4.3 Simulator-basiert

Die Simulator-basierte Fehlerinjektion basiert auf dem Ansatz, die Hardware-Plattform des Zielsystems durch eine Softwarekomponente nachzubilden und Fehler in die simulierten Komponenten zu injizieren [BP03]. Zu diesem Zweck wird der Programmcode des Simulators modifiziert, nicht jedoch der Code der darin ausgeführten Zielapplikation. Da

ein Simulator grundsätzlich die Hardware-Komponenten wie Speicher und Prozessorregister in Variablen umwandelt und die Ausführung des Prozessors durch Funktionen ersetzt, die mit diesen Zustandskomponenten arbeiten, ist ein voller Zugriff auf alle simulierten Komponenten möglich [JAR+94]. Dies gilt insbesondere, wenn der Simulator quelloffen verfügbar ist, sodass direkte Modifikationen im Simulationscode durchgeführt werden können, um Fehler in die Komponenten zu injizieren.

Grundsätzlich können Hardware-Simulatoren das Verhalten des simulierten Systems auf unterschiedlicher Detailebene nachbilden. Beispielsweise bietet der Simulator Gem5 explizit Prozessormodelle verschiedener Detailgrade für die gleiche Plattform, um dem Nutzer die Wahl zwischen Detailgrad und Ausführungsgeschwindigkeit zu ermöglichen [BBB+11]. Da das Systemmodell allerdings stets von der echten Hardware abstrahiert und insbesondere in durch injizierte Fehler ausgelösten Ausnahmefällen in seinem Verhalten von der Hardware abweichen kann, sind die Ergebnisse der Simulator-basierten Fehlerinjektion weniger realitätsgetreu als die Injektion im korrespondierenden echten System.

Da Hardware-Simulatoren üblicherweise eine deutlich reduzierte Ausführungsgeschwindigkeit gegenüber der simulierten Hardware aufweisen, ist dieses Verfahren als langsam einzustufen. Dieses Geschwindigkeitsdefizit kann jedoch mittels der Möglichkeit zur hoch-parallelen Ausführung abgemildert oder kompensiert werden. Simulator-basierte Fehlerinjektion erfordert keine Modifikationen der Zielsoftware und ist aufgrund der reinen Durchführung in Software kostengünstig im Betrieb. Auch die Implementierung Simulator-basierter Fehlerinjektion ist als günstig anzusehen, da Komponenten im Falle von quelloffenen Simulatoren unmittelbar zugreifbar sind [JAR+94]. Der Detailgrad des simulierten Modells bedingt direkt die Art von injizierbaren Fehlern. So sind teilweise in Simulatoren einige Hardware-Komponenten wie beispielsweise Caches nicht berücksichtigt, weshalb hier keine Injektion durchgeführt werden kann. Andererseits ist es bei sehr detaillierten Modellen möglich, an mehr Stellen zuzugreifen, als es typischerweise mittels Software auf der Plattform möglich.

2.4.4 OCD-basiert

On-Chip Debugger (kurz: OCD) sind eine im Bereich aktueller Mikrocontrollersysteme weit verbreitete Einrichtung, welche es ermöglicht, die Programmausführung von einem externen Hostrechner per Software zu stoppen und Prozessorregister sowie Speicher auszulesen und zu beschreiben [FGAF06]. Zudem bieten typische OCD die Möglichkeit, Breakpoints zu setzen, sodass die Ausführung beim Laden der definierten Instruktion angehalten wird. Komplexere OCD bieten zusätzlich die Möglichkeit, Datenhaltepunkte (sog. *Watchpoints*) anzulegen, die bei einem Speicherzugriff an einer definierten Speicherstelle das System anhalten. Des Weiteren bieten einige OCD komplexere Funktionalitäten, wie beispielsweise die Aufzeichnung eines Programm-Trace [FGAF06] oder die Definition von Nebenbedingungen für Haltepunkte, wie beispielsweise das Anhalten nach einer festen Anzahl an Ausführungen einer Instruktion [SBK10].

OCD wird üblicherweise für die Fehlersuche – das sogenannte *Debugging* – in dem auf der Hardware ausgeführten Programmcode genutzt, was insbesondere im Bereich ein-

gebetteter Hardware aufgrund fehlender Schnittstellen wichtig ist. Das Debugging wird in Software von einem externen Rechner gesteuert, der dem Anwender eine komfortable Schnittstelle bietet.

Aufgrund des Funktionsumfangs von OCD wird eine Fehlerinjektion durch angebundene Steuerungssoftware ermöglicht. Hier kann insbesondere eine Injektion in Speicher und Prozessorregister durchgeführt werden. Die Definition der Injektionszeitpunkte ist durch die Verwendung von Haltepunkten möglich.

Typischerweise wird ein OCD mithilfe eines sogenannten *Debuggers,* einer zusätzlich als Kommunikationsverbindung nötigen Hardware, an einen Hostrechner angebunden. Da die Datenverbindung zwischen Hostrechner und OCD über den Debugger im Allgemeinen eine hohe Datenumlaufzeit aufweist, wird häufig die Steuerung des OCD in einer Spezialhardware – beispielsweise einem FPGA – direkt realisiert [MEEM12, PGLOGVE07, FAF06b]. Wird eine standardisierte OCD-Schnittstelle – wie beispielsweise JTAG – verwendet, so ist das Fehlerinjektionssystem für verschiedene Prozessoren verwendbar [FGAF06].

OCD-basierte Fehlerinjektion weist im Regelfall eine hohe Ausführungsgeschwindigkeit für Experimente auf, was darauf beruht, dass die Hardware den Programmcode in voller Geschwindigkeit ausführen kann. Durch das zeitweise Anhalten des Systems, beispielsweise für die Injektion von fehlerhaften Werten, wird allerdings die Performanz reduziert. Die durch OCD-basierte Fehlerinjektion gewonnenen Ergebnisse weisen eine hohe Realitätstreue auf, da im Allgemeinen keine Modifikation der Hardware oder des darauf laufenden Softwaresystems durchgeführt werden muss, um das Verfahren zu realisieren [FSK98]. Aufgrund der nötigen Verwendung von zusätzlicher Hardware zur Realisierung des Zugriffs auf OCD-Funktionen ist das Verfahren als nicht kostengünstig einzuordnen. Aus diesem Grund ist auch die Parallelisierbarkeit beschränkt, da für jede zur Fehlerinjektion verwendete Systemeinheit tatsächliche Hardware verwendet wird. Realisierbare Fehlermodelle sind auf die per OCD zugreifbaren Komponenten der Hardware beschränkt. Diese decken sich üblicherweise in Großteilen mit den per Software zugreifbaren Systemanteilen.

2.5 Optimierung von Fehlerinjektionsexperimenten

Die Abdeckung eines spezifischen Fehlerraums bedarf im Allgemeinen einer Anzahl an Experimenten, die sich aus der Multiplikation der Länge des Traces mit der Anzahl der Injektionsorte ergibt und demnach schnell stark wächst. Aus diesem Grund ist eine Reduktion der Anzahl durchzuführender Experimente notwendig, damit die Laufzeit einer Kampagne nicht impraktikabel wird. Das sogenannte *Fault-Space Pruning* nach dem *Def-Use*-Prinzip kann bei Fehlerinjektion in Speicher oder Registern eingesetzt werden [BVFK05]. Hierbei werden alle im Trace befindlichen Zugriffe auf ein spezifisches Speicherwort bzw. Register mit der Adresse x betrachtet und wie in Abbildung 2.5 zu Äquivalenzklassen zusammengefasst, die nachfolgend mit $A_x^{(y)}$ bezeichnet werden. Hierbei bezeichnet y einen Index für die Menge aller Äquivalenzklassen innerhalb eines Traces für den Injektionsort x.

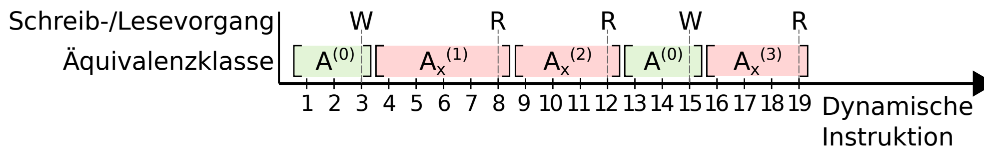


Abbildung 2.5: Schematische Darstellung der Zusammenfassung von Fehlerinjektionsexperimenten zu Äquivalenzklassen für ein Speicherwort oder Register. Die auf dem betrachteten Injektionsort durchgeführten Schreibvorgänge sind mit einem „W“ und die Lesevorgänge mit einem „R“ gekennzeichnet. Die Äquivalenzklassen mit a priori bekanntem Ergebnis sind grün eingezeichnet und die Äquivalenzklassen, für die ein einzelnes Experiment durchgeführt werden muss, sind rot eingezeichnet.

Aufgrund der Betrachtung von Paaren aufeinander folgender Schreib-/Lesevorgänge (definiert durch t_l für die linke und t_r für die rechte dynamische Instruktion) ergeben sich zwei Fälle in Abhängigkeit von dem nachfolgenden Schreib- oder Lesevorgang [BRIM98]:

1. **Lesevorgang:** Ist der nachfolgende Zugriff ein Lesevorgang, wie im Beispiel der dynamischen Instruktionen $t_l = 3$ und $t_r = 8$ in Abbildung 2.5, so werden alle Instruktionen aus dem Intervall $(t_l; t_r]$ zu einer Äquivalenzklasse zusammengefasst (in diesem Fall zur Äquivalenzklasse $A_x^{(1)}$). Für eine solche Äquivalenzklasse wird ein einziges Repräsentativexperiment zu einem beliebigen Injektionszeitpunkt innerhalb der Klasse durchgeführt und im Ergebnisraum für alle Elemente der Klasse eingetragen. Dieser Zusammenschluss ist gültig, da der injizierte Wert zum Zeitpunkt t_r gelesen wird und sich daher erst zu diesem Zeitpunkt auswirken kann.

2. **Schreibvorgang:** Ist der nachfolgende Zugriff ein Schreibvorgang, wie bei den dynamischen Instruktionen $t_l = 12$ und $t_r = 15$, so werden alle Instruktionen, in dem Intervall $(t_l; t_t]$ der Äquivalenzklasse $A^{(0)}$ hinzugefügt, welche alle a priori als fehlerfrei bekannten Experimente beinhaltet. Dieses Vorgehen ist korrekt, da bei Injektion zu einem beliebigen Zeitpunkt in dem Intervall in das betrachtete Speicherwort oder Register der fehlerhafte Wert nicht gelesen, sondern vor dem nächsten Lesevorgang wieder beschrieben wird.

Da die Zusammenfassung von Äquivalenzklassen auf Ebene von Speicherwörtern und Registern durchgeführt wird, wird für jede Äquivalenzklasse $A_x^{(y)} \neq A^{(0)}$ nach dem entsprechenden Fehlermodell eine Menge von Fehlerinjektionsexperimenten durchgeführt, beispielsweise ein Experiment für jedes enthaltene Bit.

Weil die Anzahl durchgeführter Injektionsexperimente durch das Pruning potenziell stark reduziert wird, müssen die Meta-Informationen über angelegte Äquivalenzklassen in die Berechnung der Metriken mit einfließen, indem eine Rücktransformation für jedes Injektionsergebnis in den Fehlerraum mithilfe der Informationen über die geltende Äquivalenzklasse durchgeführt wird.

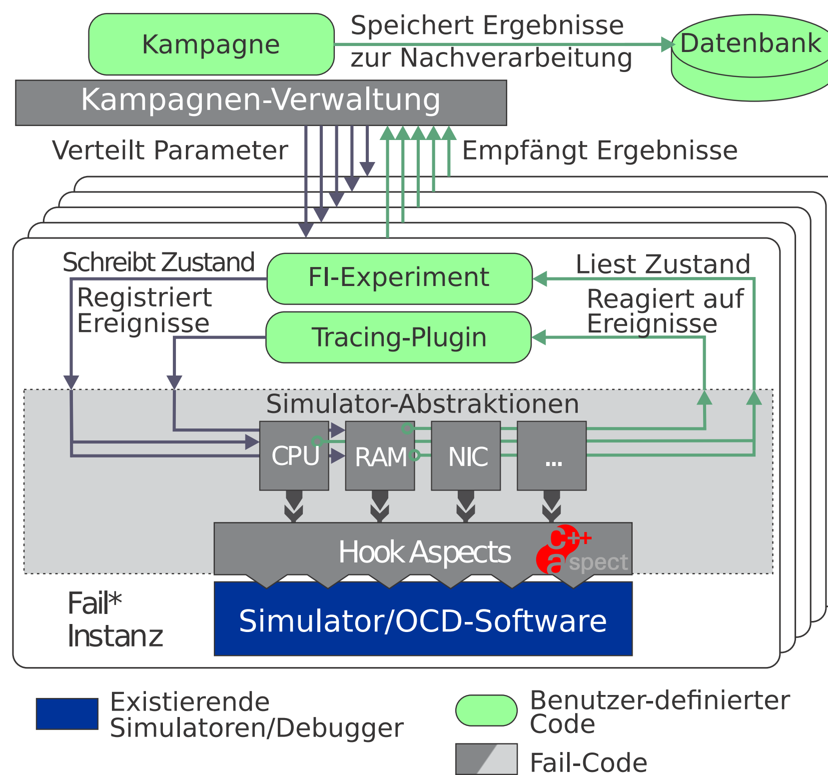


Abbildung 2.6: Schematische Darstellung der Fail*-Architektur für Simulatoren und OCD-angebundene Hardware, modifiziert aus [SHK+12].

2.6 Das Fehlerinjektions-Framework Fail*

Bei Fail* handelt es sich um ein Fehlerinjektionswerkzeug, welches im Kontext des *DanceOS*-Forschungsprojekts entwickelt wird [SHK+12]. Es ist auf Simulator-basierte und OCD-basierte Fehlerinjektion ausgelegt. Die Ausführungsplattform, in die Fail* Fehler injiziert, wird in diesem Kontext als *Backend* bezeichnet. Die Kernkonzepte von Fail* gliedern sich in folgende Aspekte:

- **Parallelität:** Durch die Verwendung einer Client/Server-Architektur wird eine verteilte Durchführung mehrerer Injektionsexperimente zeitgleich ermöglicht. Mit diesem Verfahren lässt sich, insbesondere bei der Verwendung von Simulator-Backends, ein hoher Durchsatz an Experimenten erreichen.

- **Generische Experimente:** Experimenten wird eine generische Kontrollschnittstelle zur Verfügung gestellt, um Einschränkungen bezüglich des Fehlermodells zu minimieren. Es können Ereignisse wie das Erreichen bestimmter Instruktionen oder die Auslösung eines Traps registriert werden und direkte Zustandsänderungen wie das Schreiben in Speicher oder Register durchgeführt werden.

- **Generische Backend-API:** Durch die Nutzung einer generischen Schnittstelle zu Simulatoren oder Hardware-Debuggern lässt sich das System leicht um neue Backends erweitern. Gleichfalls wird dadurch ermöglicht, Experimentcode für verschiedene Backends wiederzuverwenden.

Abbildung 2.6 zeigt die grundlegende Architektur von Fail* gegliedert in ein Modul für die Verwaltung von Kampagnen, eine Datenbank als Parameter- und Ergebnisspeicher und mehrere Module zur Ausführung der Einzelexperimente. Das *Tracing-Plugin* dient einer initialen Aufzeichnung der Programmausführung zur Definition des Fehlerraums und für die im vorherigen Abschnitt beschriebene Reduktion der Experimentmenge.

Der Benutzer entwickelt eine spezialisierte Kampagne, welche eine spezifische Menge an Experimentparametern an die Fail*-Instanzen verteilt. Es werden benutzerdefinierte Experimente auf Basis der empfangenen Parameter durchgeführt, welche die eigentliche Fehlerinjektion in dem spezifischen Backend durchführen. Zu diesem Zweck wird einem Experiment eine Simulator-Abstraktionsschnittstelle zur Verfügung gestellt, welche Zugriff auf im Zielsystem vorhandene Hardware wie Register und Speicher erlaubt.

Die konkrete Implementierung dieser Schnittstelle geschieht üblicherweise mittels sogenannter *Hook-Aspects*, die Code zur Modifikation und Registrierung von Ereignissen in die konkrete Simulatorsoftware oder alternativ die OCD-Steuerungssoftware an den erforderlichen Stellen einsetzt. Mittels der Nutzung von Aspekten lässt sich die nötige Modifikation des Zielcodes minimieren und es ist eine Trennung der Belange möglich, sodass der eingefügte Code separat von der Zielsoftware gekapselt werden kann. Die konkrete Implementierung erfolgt mithilfe der C/C++-Spracherweiterung *AspectC++*, welche aspektorientierte Programmierung ermöglicht [SLU05].

Im Folgenden wird detailliert auf wichtige Konzepte von Fail* eingegangen. Hierzu beschreibt Abschnitt 2.6.1 zunächst die Backend-Abstraktion, welche eine einfache Erweiterung des Verfahrens um neue Simulator- und OCD-basierte Backends ermöglicht. Anschließend wird das verteilte Kampagnen-Management von Fail* in Abschnitt 2.6.2 erläutert.

2.6.1 Backend-Abstraktion

Abbildung 2.7 zeigt eine abstrakte Darstellung der Komponentenstruktur von Fail* in Form eines Klassendiagramms. Die Darstellung ist grundlegend in die zwei Teile der Fail*- und der Backend-Komponenten aufgeteilt. Innerhalb der Fail*-Komponenten wurde weiterhin nach Experiment- und Kern-Komponenten aufgeteilt. Die Anbindung der bislang in Fail* integrierten Simulator-Backends Bochs (x86-Simulation) und Gem5 (im Fail*-Kontext für die Simulation von ARM-Backends verwendet) ist hier beispielhaft demonstriert. Die Abbildung macht zunächst deutlich, dass sowohl Experimente, als auch Backends die generische Schnittstelle des `SimulatorControllers` verwenden. Dieser wird durch konkrete Controller der einzelnen Backends implementiert und bietet Zugriff auf Komponenten wie den `MemoryManager` und die Backend-CPU. Somit sind konkrete Komponenten, wie die Prozessor-Register und der Speicher eines Backends, über eine generische Schnittstelle zugänglich.

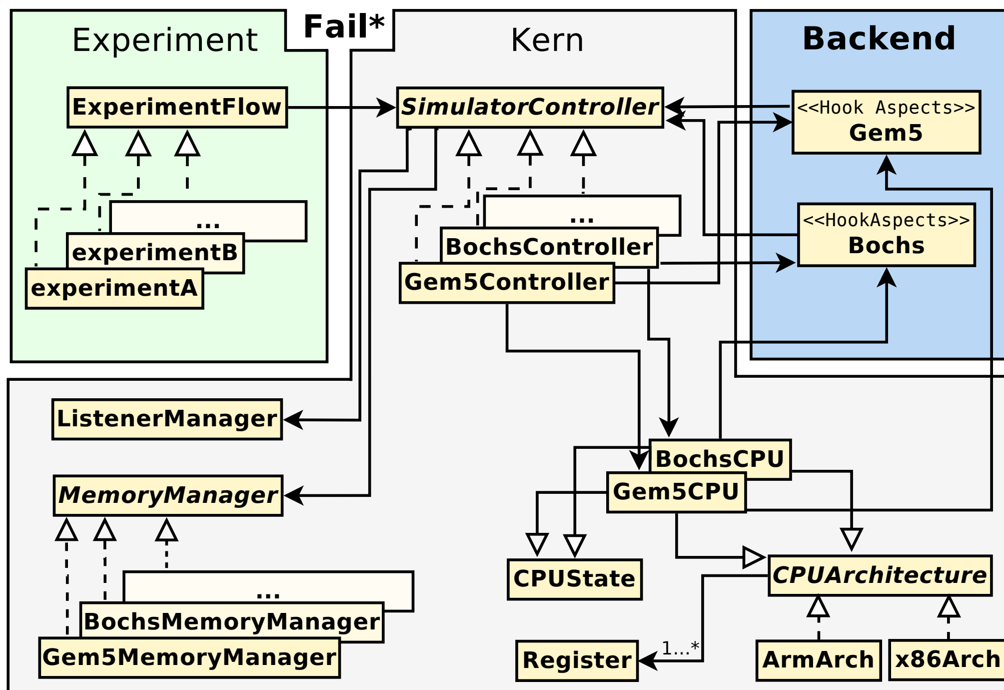


Abbildung 2.7: Abstrakte Darstellung der relevanten Schnittstellen für Experimente und Backends in Fail* als Klassendiagramm. Es wurde eine grobe Modularisierung in Fail*-Komponenten (Kern (grau) und Experiment (grün)), sowie Backend-Komponenten (blau) vorgenommen.

Aufgrund dieser Architektur wird eine Konfigurierbarkeit realisiert, die für ein unverändertes Experiment den einfachen Austausch des zugehörigen Backends ermöglicht. Die konkrete Backend-Komponente wird von den Implementierungen der generischen Schnittstellen angesprochen. Für die Propagation von Informationen des Backends zum Experiment stehen wiederum Schnittstellen im `SimulatorController` zur Verfügung, die eine einfache Anbindung neuer Backends ermöglichen. Im Folgenden werden einzelne Komponenten detaillierter diskutiert.

Der in Abbildung 2.7 aufgeführte `ListenerManager` implementiert eine zentrale Funktionalität in Fail*. In einem Experiment ist es möglich, mittels einer beliebigen Anzahl an aktuell aktiven `EventListenern` – also Komponenten, die eine Benachrichtigung bei Auftreten eines vordefinierten Ereignisses ermöglichen – aktuell relevante Zustandsänderungen auf dem Zielsystem im Experiment zu registrieren. Dieses Vorgehen wird durch den `ListenerManager` verwaltet, indem Schnittstellen für das Anlegen von `EventListenern` und Schnittstellen für die Erzeugung von zugehörigen `Events` angeboten werden. Bei Auftreten eines Ereignisses werden alle registrierten Listener, die der Ereignisklasse entsprechen, auf Übereinstimmung der Ereignisparameter geprüft und gegebenenfalls benachrichtigt. Nach aktuellem Stand sind folgende Ereignisarten definiert:

- **Breakpoint:** Soll die Ausführung einer bestimmten Instruktion (beispielsweise der Eintritt in die untersuchte Funktion) erkannt werden, so lässt sich ein Breakpoint-

```
1  // Empfange Parameter von der Kampagne
2  jc.getParam(par);
3
4  // Lade vorher gespeicherten Zustand: Die Ausführung befindet
5  // sich nun am Eintritt der zu untersuchenden Funktion.
6  simulator.restore("sav/p_entry.sav");
7
8  // Breakpoint auf der zum Injektionszeitpunkt gehörenden
9  // dynamischen Instruktion
10 BPSingleListener ev_fi_instr;
11 ev_fi_instr.setWatchInstructionPointer(ANY_ADDR);
12 ev_fi_instr.setCounter(par.injection_instr());
13 simulator.addListenerAndResume(&ev_fi_instr);
14
15 // Injektion: Einzelner Bitfehler im spezifizierten Register
16 ConcreteCPU cpu = simulator.getCPU(0);
17 Register *r = cpu.getRegister(par.inject_register());
18 regdata_t regval = cpu.getRegisterContent(r);
19 cpu.setRegisterContent(r, regval ^ (1 << par.bitpos()));
20
21 // Nachuntersuchung: Traps, Zeitüberschreitung oder Terminierung?
22 TrapListener ev_trap;
23 simulator.addListener(&ev_trap);
24 // Zeitbedingung: Berechnung in unter 1000 Mikrosekunden
25 TimerListener ev_timeout(1000);
26 simulator.addListener(&ev_timeout);
27 BPSingleListener ev_func_end(ADDR_FUNC_END);
28 simulator.addListener(&ev_func_end);
29 // Warte auf Ereignis
30 BaseListener* ev = simulator.resume();
31 // Speichere Ergebnis im Parameter-Objekt
32 if (ev == &ev_func_end) {
33   unsigned char result;
34   result = simulator.getMemoryManager().getByte(ADDR_RESULT);
35   par.set_resulttype(LOG_NORMAL);
36   par.set_result(result);
37 } else if (ev == &ev_trap) {
38   par.set_resulttype(LOG_TRAP);
39 } else if (ev == &ev_timeout) {
40   par.set_resulttype(LOG_TIMEOUT);
41 }
42
43 // Kommuniziere Ergebnis zurück an Kampagnen-Verwaltung
44 jc.sendResult(par);
```

Abbildung 2.8: Beispielhafter Experimentcode, aktualisiert nach aktueller Schnittstelle aus [SHK+12]. Das Experiment fordert zunächst die Experimentparameter von der Kampagne an. Anschließend wird ein vorher gespeicherter Simulatorzustand geladen, um das Experiment zu initialisieren. Es wird mittels Breakpoint-Listener zum Injektionszeitpunkt navigiert, am Injektionsort ein Bitfehler injiziert und anschließend mittels Trap-, Timer- und auf das Funktionsende konfiguriertem Breakpoint-Listener der Ausgang des Experiments bestimmt.

Ereignis registrieren. Ein Breakpoint ist durch die zu beobachtende Instruktionsadresse definiert. Bei der Registrierung wird automatisch die nötige Modifikation im Backend durchgeführt, um dieses Ereignis abfangen zu können. Es besteht ebenfalls die Möglichkeit, ein Intervall mehrerer Instruktionen zu beobachten, um beispielsweise einen durch Fehlerinjektion bedingten Einsprung in Datenbereiche des Speichers zu erkennen, die keinen Code enthalten.

- **MemoryAccess:** Wird ein durch Adresse, Länge und Art (lesend oder schreibend) definierter Speicherzugriff durchgeführt, so kann dies mittels eines MemoryAccess-Ereignisses angezeigt werden. Es wird somit möglich, beispielsweise einen Speicherzugriff zu erkennen, der aufgrund eines injizierten Fehlers in einen nicht erlaubten Bereich verläuft.

- **Trouble:** Ein sogenanntes Trouble-Ereignis definiert das Auftreten eines unvorhergesehenen Ereignisses auf dem Zielsystem. Konkrete Ausprägungen sind Ereignisse bei einem *Interrupt* oder *Trap*. Diese Ereignisse werden in Experimenten insbesondere für die Erkennung von durch Fehlerinjektion verursachten Traps verwendet.

- **IOPort:** Wird auf der Zielplattform eine Ein-/Ausgabe-Operation mittels Zugriff auf einen *I/O-Port* durchgeführt, so kann ein Ereignis erzeugt werden, um beispielsweise den Inhalt der Datenübertragung im Experiment auszulesen.

- **Jump:** Für die Registrierung von Sprüngen im Programmcode kann ein Jump-Ereignis herangezogen werden.

- **Timer:** Soll beispielsweise bei der Nachuntersuchung eines Fehlerinjektionsexperiments beobachtet werden, ob das Zielsystem innerhalb einer definierten Zeitspanne den Code nicht bis zum Ende ausführt, so kann dies mittels eines Timer-Ereignisses geprüft werden.

- **Guest:** Soll eine direkte Kommunikation von dem auf dem Zielsystem zur Kampagne realisiert werden, so kann dies mittels dieser Ereignisse realisiert werden. Es können Zeichenketten auf mehreren sogenannten *Ports* übertragen werden. Die Funktionalität der Ports dient hierbei der Realisierung mehrerer virtueller Kommunikationskanäle, so dass beispielsweise ein Kanal für Fehlermeldungen und ein Kanal für Status-Meldungen genutzt werden kann.

Zur Veranschaulichung der Verwendung der Ereignisschnittstelle kann der Beispielcode aus Abbildung 2.8 herangezogen werden. Hier wird beispielhaft ein an Abbildung 2.2 angelehntes einfaches Experiment definiert, welches die konkreten Experimentparameter von der Kampagne bezieht (eine detaillierte Betrachtung der Kampagnenverwaltung erfolgt in Abschnitt 2.6.2). Der Injektionszeitpunkt wird mittels eines Listeners für Breakpoint-Events realisiert, welcher bei jeder ausgeführten Instruktion aktiviert wird. Durch den Einsatz eines Zählers ist es somit möglich, eine definierte Anzahl ausgeführter Instruktionen abzuwarten. Der Injektionszeitpunkt ist demnach direkt durch seine Position im Trace definiert. Anschließend wird ein einzelner Bitfehler in ein CPU-Register

injiziert. Zu diesem Zweck wird das Register ausgelesen, das entsprechende Bit invertiert und anschließend wird das Register erneut geschrieben.

Ereignisse sind insbesondere für die Nachuntersuchung von hoher Bedeutung. So ist zu erkennen, dass die konkrete Nachuntersuchung die drei Fälle der korrekten Abarbeitung des Funktionscodes, einer Zeitüberschreitung oder eines ausgelösten Traps unterscheidet. Zu diesem Zweck werden in dem Beispiel `TrapListener`, `TimerListener` und `BPSingleListener` registriert. Nach Signalisierung eines Ereignisses wird geprüft, um welches Ereignis es sich handelt und mit dieser Information wird das Ergebnis des Experiments zurückgegeben.

Die Verwendung einer ereignisbasierten Schnittstelle ist insbesondere wichtig, da die Experimentdurchführung in Fail* mithilfe sogenannter *Koroutinen*, also kooperativer Programmfäden, erfolgt [Con63]. Dieses Verfahren wird verwendet, damit zu jedem Zeitpunkt immer nur entweder die Ausführung des Zielsystems oder der Experimentcode aktiv ist. Hierdurch wird sichergestellt, dass alle relevanten Informationen im Experiment signalisiert werden. Der Ablauf sieht demnach vor, dass das Experiment zunächst nötige Modifikationen durchführt und Ereignis-Listener registriert, um anschließend aktiv den Kontrollfluss an das Zielsystem abzugeben, welches seinerseits wiederum solange läuft, bis ein relevantes Ereignis aufgetreten ist und an Fail* signalisiert wird. In diesem Fall wird ein Koroutinenwechsel ausgelöst, wodurch der Kontrollfluss der Koroutine des Experiments forgeführt wird. Kontrollflusswechsel werden demnach experimentseitig mittels `simulator.resume()` (s. Abbildung 2.8) und Backend-seitig durch Auslösung von Ereignissen aktiviert.

Der aktive Zugriff auf Komponenten des Backends erfolgt durch Verwendung der Schnittstellen des `SimulatorControllers`. Hier wird nach Art der zugegriffenen Komponente eine Unterteilung in `MemoryManager` und abstrakte CPU durchgeführt. Diese Komponenten liefern Meta-Informationen über die verwaltete Komponente, um eine generische Fehlerinjektion zu ermöglichen. So bietet die abstrakte CPU beispielsweise einen Iterator über alle Prozessor-Register und der `MemoryManager` gibt Informationen zu Größe und Ort des genutzten Speichers an. Des Weiteren definiert der `SimulatorController` Schnittstellen, um einen Neustart durchzuführen und einen vorher gespeicherten Systemzustand wiederherzustellen (s. Abb. 2.8).

2.6.2 Kampagnenverwaltung

Aufgrund der in Abbildung 2.6 schematisierten verteilten Architektur von Fail* wird eine zentrale Verwaltung für Kampagnen ausgeführt, welche Zugriff auf eine SQL-Datenbank mit Experiment-Parametern und -Ergebnissen hat. Um die Möglichkeit zu erhalten, mehrere Kampagnen für verschiedene Benchmarks und deren Varianten als Zielapplikationen durchführen zu können, kann eine Datenbank mehrere Kampagnen mittels der Verwendung entsprechender Identifizierer verwalten. Für die Übertragung der Experiment-Parameter kann ein vorkonfigurierter Nachrichtentyp verwendet werden, sodass eine vorgegebene Kampagnenimplementierung vom Benutzer verwendet werden kann. Alternativ können allerdings auch spezialisierte Nachrichtenformate verwendet werden, die jedoch eine Implementierung zugehörigen Kampagnencodes bedingen. Experimen-

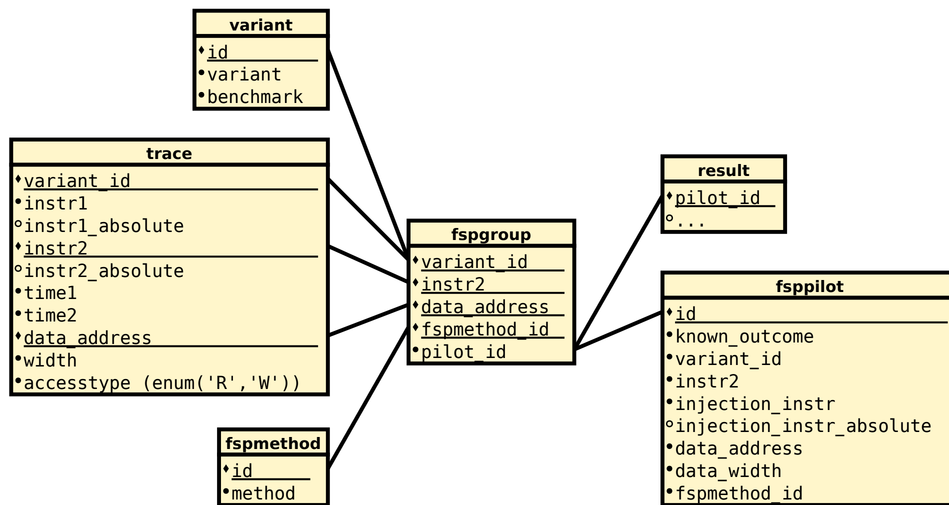


Abbildung 2.9: Darstellung der von Fail* verwendeten Datenbankstruktur.

tergebnisse werden mittels eines weiteren individuell anzulegenden Nachrichtenformat übertragen, dessen Felder direkt in eine Ergebnistabelle geschrieben werden. Die Einzelschritte werden im Folgenden detaillierter betrachtet.

Grundlage eines Experiments bildet der Programm-Trace. Dieser wird initial und vor Durchführung der Kampagne aufgezeichnet. In Abschnitt 2.5 wurde gezeigt, dass eine vollständige Abdeckung des Fehlerraums mit Experimenten im Allgemeinen nicht sinnvoll ist, da sich durch das Def-Use-Schema für die Injektion in Speicherzellen oder Register Äquivalenzklassen mit gleichen Ergebnissen bilden. Um demnach initial die Parameter-Tabelle zu füllen, werden die Äquivalenzklassen mittels des sogenannten Prunings bestimmt. Werden im Folgenden Experiment-Parameter an die Fail*-Instanzen verteilt, so genügt für die Injektion in ein Speicherwort der Adresse x die Übertragung eines innerhalb der zugehörigen Äquivalenzklasse $A_x^{(y)}$ befindlichen Injektionszeitpunktes. Dies gilt, da a priori bekannt ist, dass die Injektionsergebnisse aller Elemente der Klasse gleich sind. Da bei Verwendung von Fail* grundlegend von der Verwendung des Pruning ausgegangen wird, geschieht die Reduktion der Experimentmenge direkt beim Import des Traces in die Datenbank. Somit lässt sich der Speicheraufwand der Datenbank potenziell drastisch reduzieren, ohne notwendige Informationen zu verlieren.

Die Kommunikation zwischen Kampagnenverwaltung und Fail*-Instanzen wird mittels Google Protocol Buffers (Protobuf) [Goo13] durchgeführt. Es handelt sich hierbei um ein leichtgewichtiges Datenübertragungsformat [GDK11], welches sich insbesondere dazu eignet, Nachrichten in eine automatisch aus dem Nachrichtenformat generierte Datenbanktabelle zu transferieren. In Fail* werden Protobuf-Nachrichten für die Kontrollkommunikation zwischen Client und Server (Parameter-Anfrage/-Antwort) und für die Übertragung der Parameter und der Experimentergebnisse verwendet. Da das Ergebnis-Nachrichtenformat individuell für eine Kampagne angepasst werden kann, wird die Ergebnistabelle automatisch aus diesem Format generiert.

Die in Fail* verwendete Struktur von Datenbanktabellen wird in Abbildung 2.9 schematisiert. Die Aufgaben der verschiedenen Tabellen werden im Folgenden kurz erläutert:

- Die Tabelle *variant* wird verwendet, um mehrere Benchmark-Varianten und verschiedene darin eingebrachte Fehlertoleranzverfahren zu untersuchen. Alle verwendeten Tupel von *variant* und *benchmark* erhalten durch diese Tabelle einen eindeutigen Identifizierer und bei der anschließenden Untersuchung können die Ergebnisse in Relation zu der jeweiligen Variante gesetzt werden. Die Spalten *variant* und *benchmark* enthalten jeweils textuelle Beschreibungen.

- Für jede Variante werden in der Tabelle *trace* die durch Fault-Space-Pruning bestimmten Äquivalenzklassen eingetragen, welche durch ihre Anfangs- und Endpunkte definiert sind. Hierbei ist zu beachten, dass nach Notation aus Abschnitt 2.5 der Anfangspunkt dem Zeitpunkt t_l+1 entspricht, da der vorangegangene Speicherzugriff nicht zur Äquivalenzklasse gehört. Die Äquivalenzklassen werden sowohl durch Trace-Positionen (*instr1*, *instr2*), als auch durch Zeiten in der Ausführung (*time1*, *time2*) repräsentiert. Der Unterschied liegt hier darin, dass unterschiedliche Instruktionen im Allgemeinen unterschiedliche Ausführungsdauern aufweisen. Es ist daher durch den Einsatz der Definition mit (*time1*, *time2*) möglich, Ergebniswerte in eine zeitliche Relation bezüglich der Gesamtausführung des Zielsystems zu setzen, wobei die Werte (*instr1*, *instr2*) nur Aufschluss über die Anzahl der betroffenen Instruktionen geben. Mithilfe der Einträge für Instruktionsadressen der unteren und oberen Begrenzung der Äquivalenzklasse wird es zusätzlich ermöglicht, zu prüfen ob sich das Zielsystem zum Injektionszeitpunkt tatsächlich an der richtigen Instruktion befindet. Weiterhin beschreibt jeder Eintrag der Tabelle *trace* die Art des Speicherzugriffs, definiert durch Adresse (*data_ address*), Breite (*width*) und Zugriffsart (*accesstype*).

- In der Tabelle *fsppilot* werden Einträge der Tabelle *trace* zu sogenannten Experiment-Piloten zusammengefasst. Hier werden die Parameter für alle einer Kampagne zugehörigen Experimente angelegt. Ein Pilot bietet die Möglichkeit, Pruning auch auf anderen Ebenen als dem in dieser Arbeit beschriebenen Def-Use-Schema durchzuführen. Dies wird über den Eintrag *fspmethod_ id* angezeigt. Da Mengen von Experimenten existieren, für die bereits a priori das Ergebnis des Experiments bekannt ist, werden sie durch den Wahrheitswert *known_ outcome* markiert. Ein Pilot enthält die Parameter *injection_ instr, data_ address* und *data_ width* unter denen das zugehörige Experiment gesteuert wird.

- Zur Realisierung verschiedener Pruning-Methoden ordnet die Tabelle *fspmethod* jedem Eintrag aus *fsppilot* eine textuelle Beschreibung der Art der Zusammenfassung von Experimenten zu.

- Zu jedem Eintrag der Tabelle *fsppilot* werden je nach Fehlermodell ein oder mehrere Einträge in der Tabelle *result* angelegt. Im Diagramm ist angedeutet, dass nur die Verbindung zur Tabelle *fsppilot* mithilfe des entsprechenden Identifizierers fest

definiert ist. Alle weiteren Einträge der Ergebnistabelle sind experimentabhängig. Die Definition der Tabellenspalten wird automatisch aus dem für eine Kampagne definierten Ergebnisnachrichtenformat generiert.

- Die Tabelle *fspgroup* bietet schließlich die Möglichkeit zur Verknüpfung der verschiedenen Tabellen, um im Anschluss an die Durchführung einer Kampagne entsprechende Nachuntersuchungen auf den Daten durchführen zu können. Sie ordnet den Identifizierern *variant_ id*, *instr2*, *data_ address* und *fspmethod_ id* einem Eintrag aus *fsppilot* und damit einem Eintrag der Ergebnistabelle zu.

Durch diese Datenbank-Architektur ist eine Erweiterbarkeit hinsichtlich verschiedener Ergebnistypen und Pruning-Methoden möglich. Es können in einer Kampagne verschiedene Benchmark- und Fehlertoleranz-Varianten untersucht werden. Außerdem wird eine anschließende Auswertung der Ergebnisse durch den Zusammenschluss der generierten Daten möglich.

2.7 Zusammenfassung

In diesem Kapitel wurden die Grundlagen dieser Arbeit erläutert. Es wurde zunächst auf grundlegende Begrifflichkeiten in Bezug auf Fehlerinjektion eingegangen, auf denen im Folgenden aufgebaut wird. Es wurde gezeigt, dass Hardware-Fehler durch verschiedene Modelle beschreibbar sind, wobei das im Kontext der Arbeit entworfene Fehlerinjektionssystem keine Einschränkung auf ein konkretes Modell vorsieht. Des Weiteren wurden Zuverlässigkeitsmetriken vorgestellt, welche den Vergleich von verschiedenen Fehlertoleranzverfahren auf Basis der Ergebnisse von Fehlerinjektion erlauben. Es wurden die Software-, Hardware-, Simulator- und OCD-basierten Fehlerinjektionsverfahren einander vergleichend gegenübergestellt, wobei im Folgenden auf den Erkenntnissen bezüglich der OCD-basierten Verfahren aufgebaut wird. Bei der Entwicklung von FailPanda wird die in diesem Kapitel vorgestellte Optimierung von Fehlerinjektionsexperimenten angewandt, um die durchzuführende Experimentmenge zu reduzieren. Schließlich wurde in diesem Kapitel detailliert auf das Fehlerinjektions-Framework Fail* eingegangen. Die Anbindung eines eingebetteten Systems an dieses Framework zur Durchführung von OCD-basierter Fehlerinjektion wird in den nachfolgenden Kapiteln erläutert.

3 Analyse und Entwurf

Dieses Kapitel beschreibt konkrete Entwurfsentscheidungen bei der Entwicklung eines Fehlerinjektionssystems auf Basis von COTS-Hardware. Die Entwurfsentscheidungen werden auf Basis einer Anforderungsanalyse begründet und es werden Alternativen diskutiert. Im vorangegangenen Kapitel, insbesondere in den Abschnitten 2.4.4 und 2.6, wurden bereits wichtige Anforderungen an die Anbindung eines Hardwaresystems an ein Fehlerinjektionssystem, schwerpunktmäßig in Bezug auf Fail*, beschrieben. Auf dieser Grundlage werden in Abschnitt 3.1 zunächst wichtige Anforderungen an das System hinsichtlich Funktion und Performanz identifiziert. Darauf basierend wird in einem Top-Down Verfahren in Abschnitt 3.2 zunächst der Entwurf der konkreten Gesamtlösung erläutert, um anschließend in den Abschnitten 3.3 bis 3.7 auf den Entwurf der Einzelfunktionalitäten einzugehen.

3.1 Anforderungsanalyse

Die im vorherigen Kapitel beschriebenen Funktionen, insbesondere bezüglich OCD-basierter Fehlerinjektion, werden in diesem Kapitel als funktionale Grundanforderungen interpretiert, sodass ein auf diesen Anforderungen basierender Entwurf des Gesamtsystems durchgeführt werden kann. Abbildung 3.1 zeigt die nach den Phasen Trace-Aufzeichnung, Initialisierung, Trace-Navigation, Fehlerinjektion und Nachuntersuchung getrennten verschiedenen Aufgaben eines Fehlerinjektionssystems. Im Folgenden werden die Anforderungen an diese Einzelfunktionen in der aufgezeichneten Reihenfolge analysiert.

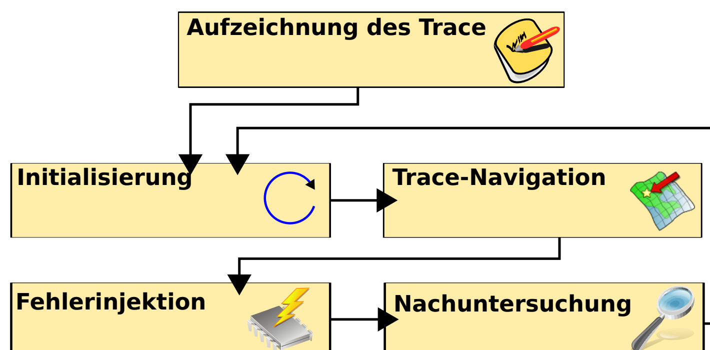


Abbildung 3.1: Schematische Aufteilung von Fehlerinjektionsexperimenten in 5 Phasen und ihre Ablaufreihenfolge für die Durchführung einer Kampagne.

Die minimale funktionale Anforderung an das Tracing ist die Aufzeichnung der Anzahl ausgeführter dynamischer Instruktionen. Mit dieser Information lässt sich die zeitliche Dimension des Fehlerraums definieren. Somit kann im Experiment mittels sogenannter *Single-Steps* zum Injektionszeitpunkt navigiert werden. Ein tatsächliches Tracing, also die Aufzeichnung der ausgeführten dynamischen Instruktionen und der Ausführungsreihenfolge, ermöglicht im Gegensatz dazu bei der Experimentnavigation die Verwendung von Breakpoints, um zum nächsten Auftreten einer Instruktion zu „springen“. Der Vorteil dieser Sprünge gegenüber dem Single-Stepping liegt darin, dass das Zielsystem alle „übersprungenen“ Instruktionen in voller Ausführungsgeschwindigkeit abarbeiten kann, wodurch sich ein erheblicher Performanzgewinn ergibt (s. Abschnitt 3.5).

Durch zusätzliche Aufzeichnung von durchgeführten Speicherzugriffen können einerseits Speicherzugriffe als zusätzliche Sprungziele verwendet werden und andererseits kann die Gesamtperformanz gesteigert werden, indem eine Reduktion der Experimentmenge durchgeführt wird (s. Abschnitt 2.5). bei Verwendung eines Fehlermodells, das Register-Fehler vorsieht, sind ebenfalls die Registerzugriffe erforderlich. Diese können allerdings nach dem Tracing in Form einer statischen Analyse des Trace gewonnen werden.

Da eine Trace-Aufzeichnung nur einmal vor der Ausführung einer Kampagne durchgeführt werden muss, sind die Anforderungen an Performanz nicht sehr streng. Dennoch ist die potenziell große Länge generierter Traces zu beachten, aufgrund derer die gesamte Durchführungszeit des Tracings stark ansteigen kann. Das Verfahren muss demnach für Aufzeichnungen üblicher Länge eine praktikable Laufzeit aufweisen. Beispielsweise wäre es noch tolerierbar, wenn das Tracing eine Verarbeitungszeit in der Größenordnung eines Tages aufwiese. Um in den folgenden Systemphasen möglichst hohe Performanz zu erreichen, sollte das Tracing funktional vollständig in der hier beschriebenen Art sein. Es müssen also die ausgeführten Instruktionen und Speicherzugriffe aufgezeichnet werden können.

Die Systeminitialisierung dient der Herstellung eines definierten fehlerfreien Systemzustands zum Experimentbeginn, sodass das Verhalten der Hardware bis zum Zeitpunkt der Fehlerinjektion immer identisch ist. Weiterhin kann die Initialisierung der Navigation des Zielsystems zu der für die Untersuchung relevanten Funktion dienen, indem beispielsweise ein Sicherungszustand eingespielt wird [BP03]. Da die Initialisierung zu Beginn jedes Experiments ausgeführt wird, ist sie als kritisch für die Performanz des Gesamtsystems anzusehen, weshalb sie diesbezüglich zu optimieren ist.

Die Trace-Navigation hat sich als kritisches Element in der Performanz eines Fehlerinjektionsexperiments herausgestellt. Messungen an dem Gesamtsystem, welches in Abschnitt 3.2 vorgestellt wird, haben ergeben, dass beispielsweise das sogenannte Single-Stepping – also das Anhalten nach jeder ausgeführten Instruktion – eine Navigationszeit von

$$t_{nav}^{single}\ (n_{target}) = n_{target} \cdot 69,99ms \tag{3.1}$$

aufweist. Hierbei beschreibt n_{target} die Anzahl nötiger Navigationsschritte bis zur Zielinstruktion. Bei einer als gering einzuordnenden Größenordnung von beispielsweise durchschnittlich $n_{target} = 10^6$ wird t_{nav}^{single} zu etwa $19,4$ Stunden Navigationszeit für ein einziges Experiment, wodurch das Gesamtsystem nicht praktisch einsetzbar ist. Demnach müs-

sen geeignete Navigationsstrategien für eine Minimierung der Kosten gefunden werden. Funktional muss die Trace-Navigation die Möglichkeit bieten, die Ausführung zu jeder im Trace aufgezeichneten Instruktion fortzusetzen.

Für die Ausführung der eigentlichen Injektion eines Fehlers muss Zugang zu der zu untersuchenden Komponente bestehen. Es ist üblicherweise erforderlich, sowohl schreibende als auch lesende Zugriffe durchführen zu können. Somit können Werte ausgelesen und der Inhalt teilweise modifiziert wieder zurückgeschrieben werden. Da sich das Fehlermodell dieser Arbeit auf Fehler im Speicher und in den CPU-Registern beschränkt, ist kein weiterer Zugriff für die Injektion von Fehlern notwendig. Da innerhalb eines Experiments nur eine einzelne beziehungsweise wenige Injektionen erfolgen, ist der Vorgang als weniger performanzkritisch anzusehen.

Der letzte Vorgang innerhalb eines Fehlerinjektionsexperiments besteht in der Untersuchung der durch die Fehlerinjektion verursachten Abweichungen vom normalen Verhalten, welches im Golden Run aufgezeichnet wurde. Da die Ausführung bis zum Ende der untersuchten Funktion erfolgen muss, ist dieser Teil des Experiments performanzkritisch. Daher sollten alle Ereignisse, die es zu untersuchen gilt, während der aktiven Ausführung der Hardware auftreten können. Das heißt, dass es nicht praktikabel ist, eine Ausführung der nachfolgenden Einzelinstruktionen mit jeweiliger Zustandsuntersuchung im Single-Stepping-Verfahren durchzuführen. Zur Untersuchung der Nachwirkungen muss erkannt werden können, ob

- eine erfolgreiche Terminierung,
- ein unerlaubter Speicherzugriff oder
- ein Trap

aufgetreten ist. Im Fall einer erfolgreichen Terminierung ist es zusätzlich nötig, das Berechnungsergebnis zu untersuchen, um mögliche Berechnungsfehler zu erkennen.

Nach diesen Definitionen von Anforderungen zur Ermöglichung von Fehlerinjektion wird im Folgenden nach den Teilbereichen der spezifischen Zielplattform (s. Abschnitt 3.1.1) und der Systemarchitektur (s. Abschnitt 3.1.2) aufgeteilt auf weitere spezifische Anforderungen eingegangen, welche sich grundsätzlich stellen oder spezielle Performanzoptimierungen begünstigen.

3.1.1 Zielplattform

An die Zielplattform werden einige Anforderungen gestellt, die im Folgenden erörtert werden. Zunächst wird grundlegend davon ausgegangen, dass die Fehlerinjektionsexperimente insbesondere für den Bereich mobiler hochperformanter eingebetteter Systeme, wie beispielsweise in Prozessoren aktueller Smartphones, durchgeführt werden (s. Abschnitt 1). Mit dieser Wahl wird gleichzeitig berücksichtigt, dass eine möglichst hohe Ausführungsgeschwindigkeit erreicht werden kann, um einen hohen Performanzgewinn gegenüber einer Simulator-basierten Lösung zu erlangen. Aktuelle Vertreter aus dem Bereich der Smartphone-Prozessoren arbeiten mit sehr hohen Taktraten, beispielsweise

nutzt das Samsung Galaxy S4 einen Prozessor mit einer maximalen Taktfrequenz von 1,9 GHz auf bis zu vier Prozessorkernen [Sam13].

Von einem solchen Prozessor wird gleichzeitig eine weitere Anforderung erfüllt: Das zu wählende System soll eine Multiprozessorarchitektur aufweisen, damit Teile der Fehlerinjektionsfunktionalität vom Hostrechner auf einen Zielprozessorkern ausgelagert werden können, während ein anderer Kern als eigentliches Zielsystem verwendet wird. Des Weiteren ist eine Beschränkung auf COTS-Hardware gefordert, damit die Ergebnisse dieser Arbeit einfach reproduzierbar sind.

Grundsätzlich wird davon ausgegangen, dass die Zielhardware einen OCD beinhaltet und diesen per JTAG-Schnittstelle zugänglich macht. Aus diesem Grund ist der Einsatz eines sogenannten Entwicklungsboards sinnvoll, da hier beispielsweise im Vergleich zum Einsatz eines Smartphones typischerweise Debugger-Schnittstellen integriert sind. Die Kommunikation zwischen Hostrechner und Zielsystem soll für diesen Entwurf allerdings nicht von vornherein auf die ausschließliche Verwendung der OCD-Schnittstelle beschränkt werden. Insbesondere bei Integration von Fehlerinjektionslogik auf Prozessoren des Zielsystems ist daher ein Angebot an alternativen Schnittstellen vorteilhaft, um möglicherweise eine schnellere Datenübertragung realisieren zu können.

Um Komplikationen bei der Entwicklung des Fehlerinjektionssystems zu reduzieren, kann es von Vorteil sein, eine Hardware-Plattform zu verwenden, die eine breite Unterstützung erfährt. So ist es wichtig, dass beispielsweise bei Verwendung der JTAG-Schnittstelle Debugger mit entsprechender Unterstützung existieren, welche sich ebenfalls in die Kategorie COTS eingliedern, da beispielsweise die Verwendung eines sehr teuren Debuggers nach der Zielsetzung dieser Arbeit (siehe Abschnitt 1) nicht zielführend ist. Weiterhin ist es sinnvoll, eine Plattform mit Unterstützung durch quelloffene OCD-Steuerungssoftware zu wählen, damit diese mit geringem Entwicklungsaufwand genutzt und erweitert werden kann.

3.1.2 Systemarchitektur

Anforderungen an die Architektur des Gesamtsystems bestehen hauptsächlich in der Kostenminimierung der Lösung und in der Maximierung der Performanz. Die Schnittstelle zum On-Chip Debugger stellt einen maßgeblichen Faktor in der Gesamtperformanz dar. So wurden bereits in artverwandten Arbeiten kostenintensive Debugger wie der Lauterbach t32 verwendet [YRLG03], dessen Einsatz die Performanz des Gesamtsystems stark verbessern kann. Im Kontext dieser Arbeit liegt der Schwerpunkt allerdings auf der Verwendung kostengünstiger Hardware. Durch die Verwendung von Entwicklungsboards mit einem großen Angebot an Schnittstellen ist es möglich, bei der Entwicklung auch weitere Verbindungen von Host zu Zielsystem zu verwenden. So kann beispielsweise Ethernet zur Datenübertragung verwendet werden, um in Verbindung mit einer Implementierung von Fehlerinjektionslogik auf einem Prozessorkern ein performantes Gesamtsystem zu realisieren.

Ein weiterer Teil der Systemarchitektur wird durch die Software zur Steuerung der OCD-Einheit ausgemacht. Anforderungen an diese Software bestehen primär darin, dass die gewählte Kombination aus Prozessor und Debugger unterstützt werden. Weiterhin

ist es notwendig, dass entweder eine detaillierte Schnittstelle zur Verfügung gestellt wird, die alle hier entworfenen Funktionalitäten ermöglicht oder dass die Software quelloffen ist, sodass entsprechende Funktionalitäten direkt in den Code des Injektionssystems eingebunden werden können. Die Software sollte im Idealfall alle prozessorspezifischen OCD-Funktionen unterstützen.

3.2 Gesamtentwurf

Der Lösungsentwurf für das in dieser Arbeit entwickelte Fehlerinjektionssystem sieht die Verwendung eines Entwicklungsboards mit einem Multikern-Prozessor der ARM-Architektur als Zielsystem vor. Es wird weiterhin zur Steuerung der OCD-Komponente ein von diesem Prozessor unterstützter JTAG-Debugger verwendet, welcher über eine quelloffene OCD-Steuerungssoftware an Fail* angebunden wird. Abschnitt 3.2.1 erläutert zunächst den Entwurf bezüglich der verwendeten Zielplattform und Abschnitt 3.2.2 geht anschließend auf die Systemarchitektur ein.

3.2.1 Pandaboard ES

Als Zielhardware wurde das *Pandaboard ES* ausgewählt [pan11]. Hierbei handelt es sich um ein Entwicklungsboard, welches einen typischen Smartphone-Prozessor, den *Texas Instruments OMAP4460*, beinhaltet. Der Prozessor wird beispielsweise von dem Smartphone Google Galaxy Nexus verwendet [Wik13]. Der OMAP4460 enthält zwei hochperformante ARM Cortex-A9-Prozessorkerne mit einer maximalen Taktfrequenz von 1,2 GHz und zwei ARM Cortex-M3-Mikroprozessoren. Im POP-Verfahren ist 1GB LPDDR2 DRAM in den Prozessor integriert. Zusätzlich sind weitere Komponenten zur performanten Medienverarbeitung verbaut, auf die in dieser Arbeit nicht weiter eingegangen wird. Das Pandaboard wird zu einem Preis von etwa 180 € angeboten und liegt damit im Bereich kostengünstiger Entwicklungssysteme. Im Vergleich zu ähnlichen hochperformanten Entwicklungsboards aus dem Bereich der mobilen eingebetteten Systeme mit hohem Unterstützungsgrad – wie beispielsweise dem Beagleboard [bea13] – ist die vom Pandaboard gebotene Performanz deutlich höher.

Der Cortex-A-Prozessor entspricht der ARM architecture v7 (kurz: ARMv7). Durch die Architektur sind Programmiermodell, Instruktionssatz, Systemkonfiguration, Ausnahmebehandlung und Speichermodell vorgegeben, sodass Entwicklungen innerhalb einer Architektur übertragbar sind [ARM09]. Ebenfalls sind zwei Cortex-M3-Prozessoren verfügbar, die ebenfalls ARMv7 entsprechen. Im Detail unterscheiden sich die Architekturen der Prozessoren dennoch, was allerdings an dieser Stelle nicht relevant ist [ARM09].

Die grundlegende Idee bei der Integration der unterschiedlichen Prozessoren auf einem Chip liegt darin, dass die Cortex-A-Prozessoren im praktischen Betrieb für rechenintensive Prozesse eingesetzt werden, bei geringem Performanzbedarf allerdings die laufende Software schnell auf die Cortex-M3-Prozessorkerne migriert werden kann. Durch dieses Verfahren können zeitweise hohe Performanzanforderungen unter im Mittel niedrigen Energiekosten erfüllt werden [ARM13a].

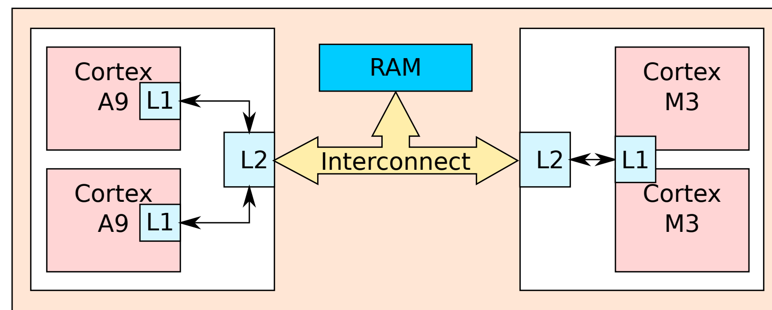


Abbildung 3.2: Schematische Darstellung der Prozessorarchitektur des OMAP4460 in Bezug auf den Speicherzugriff (abstrahiert aus [Tex12]). Dargestellt sind Prozessor-Kerne (rot), Cache-Hierarchie (hellblau), Interconnect (gelb) und Hauptspeicher (dunkelblau).

Um den Vorgang der Migration schnell durchführen zu können, haben alle Prozessorkerne Zugriff auf den gleichen Speicher, wie Abbildung 3.2 zeigt. Hier ist ebenfalls ersichtlich, dass die Prozessoren aufgeteilt nach Typ paarweise in sogenannten Subsystemen angeordnet sind. Beide Prozessor-Subsysteme kommunizieren über einen sogenannten *Interconnect* mit dem Hauptspeicher. Innerhalb der Subsysteme sind Caches untergebracht, sodass das Cortex-A- und das Cortex-M-Subsystem keinen gemeinsamen Cache haben. Während auch die Cortex-A9-Kerne jeweils einen eigenen Level-1-Cache besitzen, wird dieser im Cortex-M-Subsystem von beiden Prozessoren geteilt. Es wird in beiden Subsystemen jeweils ein gemeinsamer Level-2-Cache verwendet. Zusätzlich zu den aufgezeichneten Komponenten enthält jedes Prozessor-Subsystem unter anderem eine MMU zur Verwaltung von virtuellem Speicher.

Konkret wird der OMAP4460 im Kontext dieser Arbeit so verwendet, dass einer der beiden Cortex-A9-Kerne das eigentliche Zielsystem darstellt, während der andere Prozessorkern ignoriert wird. Zusätzlich wird ein Cortex-M3-Kern für die Speicherinjektion ohne Cache-Invalidierung verwendet, was durch die in Abbildung 3.2 gezeigte Architektur ermöglicht wird, da die Caches lokal in den Subsystemen verwaltet werden und keine Invalidierung über den Interconnect propagiert wird.

Das Entwicklungsboard bietet einen JTAG-Anschluss für entsprechende Debugger. Es wird ebenfalls ein sogenannter *Trace Port* angeboten, mit dessen Hilfe das Auslesen von Ausführungsaufzeichnungen mithilfe spezieller Tracing-Komponenten möglich ist [Tex12]. Die Verwendung eines Trace Port wurde im Kontext der Arbeit aufgrund der zu kostenintensiven notwendigen Hardware jedoch verzichtet. Zusätzlich ist es möglich, Systemausgaben direkt über eine integrierte serielle Schnittstelle durchzuführen und eine schnelle Datenübertragung mit über eine angebotene Ethernet-Schnittstelle durchzuführen. In dieser Arbeit wird ausschließlich mit der JTAG-Schnittstelle gearbeitet. Andere Schnittstellen können zwar von der zu untersuchenden Applikation verwendet werden, sind jedoch nicht Teil der Fehlerinjektionsarchitektur.

Der Cortex-A9 Prozessor ermöglicht das Anlegen von sechs Breakpoints und vier Watchpoints in Hardware. Zusätzliche Breakpoints können prinzipiell auch als Software-Breakpoint eingesetzt werden, indem die entsprechende Instruktion im Speicher durch

eine spezielle BKPT-Instruktion ersetzt wird [ARM11a]. Im Kontext der Arbeit wird allerdings auf die Verwendung verzichtet, da die Anzahl von sechs Breakpoints für die volle Funktionalität des entwickelten Systems ausreicht.

Die Breakpoints können in verschiedenen Modi verwendet werden. Üblicherweise wird ein sogenannter *Instruction Address Match* verwendet. Hierbei wird ein Breakpoint aktiviert, wenn der aktuelle Wert des *Program Counters* – also des Registers, welches die Adresse der aktuell ausgeführten Instruktion beinhaltet – gleich dem im Breakpoint definierten Vergleichswert ist. Alternativ ist auch ein *Instruction Address Mismatch* möglich. Hierbei wird genau dann ein Debug-Ereignis erzeugt, wenn die Adresse der aktuell verarbeiteten Instruktion nicht dem Vergleichswert entspricht.

Watchpoints können einen Adressbereich der Breite von 4 Byte beobachten. Hierbei kann der Typ des Zugriffs als Lesen, Schreiben oder Zugriff spezifiziert werden. Im letzten Fall ist der Watchpoint sowohl für lesende als auch für schreibende Zugriffe sensitiv. Mithilfe von sogenanntem *Address Range Masking* ist es zusätzlich möglich, bis zu acht der weniger signifikanten Bits einer Adresse von dem Adressvergleich auszuschließen, wodurch größere Bereiche untersucht werden können [ARM11a]. Die Architektur bietet zusätzlich zu den genannten OCD-Funktionen die Funktion des sogenannten *Performance Monitoring*, welches beispielsweise das Auslesen eines Prozessorzyklenzählers ermöglicht [Tex12].

Der typischerweise auf dem Pandaboard verwendete Bootcode führt eine grundlegende Initialisierung durch, die das Booten eines Linux-Systems erlaubt. Es wird hierbei nur ein Cortex-A9-Kern aktiviert, ohne das Caching oder die MMU einzuschalten. Alle über diese Initialisierung hinausgehenden grundlegenden Vorbereitungen für FailPanda werden in einem spezialisierten Bootcode vor Ausführung der eigentlichen Zielplattform durchgeführt. Eine detaillierte Beschreibung dieses Vorgehens wird in Abschnitt 4.1 durchgeführt.

3.2.2 Flyswatter 2 und OpenOCD

Zur Anbindung des Pandaboards an einen Hostrechner wird in dieser Arbeit der *Flyswatter2* der Firma *TinCan Tools* verwendet [Tin13]. Hierbei handelt es sich um einen ARM- und MIPS-Debugger, welcher für das Debugging des Pandaboards und ähnlicher Entwicklungsboards eingesetzt werden kann. Bei einem Preis von etwa 80 € ist der Debugger als kostengünstig einzuordnen. Er wird mittels der USB2.0-Schnittstelle an einen Rechner angeschlossen, wodurch eine mit maximal 30 MHz taktbare JTAG-Schnittstelle zur Verfügung steht. Im Einsatz in der nachfolgend beschriebenen Schnittstellen-Software ist allerdings ein Betrieb nur bei bis zu 6 MHz möglich.

Sowohl der Flyswatter2 als auch das Pandaboard werden von der freien Software *OpenOCD* unterstützt, welche eine Softwareschnittstelle zur Verwendung von OCD-Debuggern darstellt [HR06, OHEB13]. Die Software ist stark konfigurierbar und bietet die Möglichkeit, verschiedene Kombinationen aus Debugger und Hardware zu verwenden. Aus diesem Grund ist die in dieser Arbeit entwickelte Anbindung von OpenOCD an Fail* für andere Backends weiterverwendbar, wobei die notwendigen Modifikationen als gering angenommen werden. Da es sich um eine quelloffene Software handelt, ist es

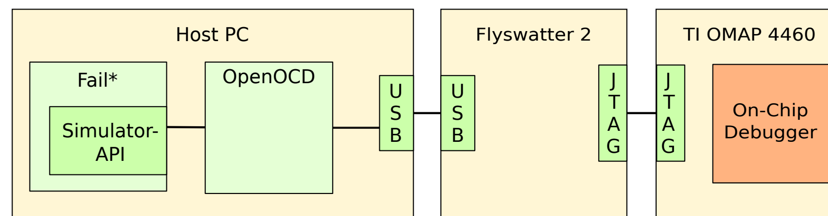


Abbildung 3.3: Schematische Darstellung der Gesamtsystemarchitektur.

möglich, spezialisierte Anpassungen durchzuführen, um die besonderen Anforderungen eines Fehlerinjektionssystems erfüllen zu können. In dieser Arbeit wurde OpenOCD in der Version 7.0 verwendet. Eine Einschränkung der Software liegt darin, dass zwar auf beide Cortex-M3 Kerne, jedoch nur auf einen der Cortex-A9 Prozessorkerne zugegriffen werden kann. Da in dieser Arbeit allerdings keine Fehlerinjektion in Multikern-Systeme erfolgen soll, bleibt die Einschränkung ohne Folgen.

Die Architektur des entworfenen Gesamtsystems ist in Abbildung 3.3 dargestellt. Der Grafik kann entnommen werden, dass auf dem Hostrechner Fail* mit Anbindung an OpenOCD ausgeführt wird. Die Software verbindet den Rechner mit der externen Komponente des JTAG-Debuggers Flyswatter2, welcher wiederum per JTAG die Verbindung zum On-Chip Debugger des OMAP4460 herstellt.

3.3 Trace-Aufzeichnung

Für die Aufzeichnung von Programm-Traces eignen sich insbesondere speziell dafür vorgesehene Komponenten. Der Cortex-A9 bietet zu diesem Zweck eine sogenannte *Program Trace Macrocell* (kurz: PTM), welche eine Aufzeichnung eines reinen Instruktions-Trace bei voller Ausführungsgeschwindigkeit durchführen kann [pan11]. Es besteht die Möglichkeit, den so generierten Trace in einem Pufferspeicher in der CPU zwischenzuspeichern und blockweise auszulesen oder instruktionsweise mittels speziellen Debuggern abzurufen. Da OpenOCD zu diesem Zeitpunkt keine Unterstützung zur Verwendung der PTM bietet und da diese auch kein Tracing von Speicherzugriffen erlaubt, konnte die PTM im Kontext der Arbeit nicht eingesetzt werden. Im Bereich teurer Hardware stehen zum Auslesen der PTM weitere Debugger wie der ARM DSTREAM [ARM13b] zur Verfügung, die den Trace sehr schnell in einen internen Puffer auslesen. Wird ein derartiger Debugger mit entsprechender Steuerungssoftware eingesetzt, so lässt sich das Tracing, wie auch weitere OCD-Operationen, deutlich beschleunigen. Bei Wahl der Systemkomponenten wurde allerdings aus Kostengründen darauf verzichtet.

Zum Anlegen eines einfachen Instruktions-Trace wird bei Fail* üblicherweise zunächst unter Zuhilfenahme eines Breakpoints zum Eintritt in die zu untersuchende Funktion navigiert. Ab diesem Punkt wird per Single-Stepping jeweils eine Instruktion ausgeführt, wobei der aktuelle Program Counter im Trace abgespeichert wird. Dieses Verfahren wurde auch im Zuge dieser Arbeit durchgeführt. Wie in Abschnitt 3.1 bereits erwähnt wurde, benötigt die hier verwendete Hardwarekombination eine Zeit von 69,99 ms zur

Durchführung eines Single-Steps. Wird beispielsweise ein Trace der Länge $n = 10^6$ aufgezeichnet, so benötigt das System für diesen Vorgang etwa 19,4 h. Da ein Trace nur einmalig aufgezeichnet werden muss, ist diese hohe Bearbeitungszeit für kurze Traces dennoch praktisch anwendbar.

Zusätzlich zu dem einfachen Instruktions-Trace ist für die Reduktion der Experimentmenge (s. Abschnitt 2.5) die Aufzeichnung von Speicherzugriffen erforderlich. Da Maschineninstruktionen typischerweise Speicherzugriffe auf Grundlage aktueller Registerwerte durchführen, kann eine Offline-Analyse des Programmcodes keine Lösung für das Problem der Erkennung von Speicherzugriffen liefern. Stattdessen wird die aktuelle Instruktion während des Tracings untersucht [HKS+13]. Zunächst wird untersucht, ob die aktuelle Instruktion einen Speicherzugriff durchführt und wenn das der Fall ist, ob dieser lesend oder schreibend ist. Es ist anzumerken, dass der ARM-Instruktionssatz keine speicherindirekte Adressierung bietet. Die Speicheradresse des aktuellen Zugriffs kann demnach ausschließlich aus den aktuellen Inhalten der Prozessor-Register und eventuell in der Instruktion kodierter Konstanten berechnet werden. Da die Komplexität der vom ARM-Instruktionssatz ermöglichten Speicherzugriffe stark begrenzt ist, ist eine Kalkulation der zugegriffenen Speicheradresse weitestgehend trivial. Es kann beispielsweise in einer Instruktion nicht lesender und schreibender Zugriff vereint werden und der zugegriffene Speicherbereich ist immer zusammenhängend [ARM05].

Zur Berechnung der aktuell zugegriffenen Speicheradresse ist demnach zunächst zu dekodieren, welche Registerwerte an der Berechnung beteiligt sind. Diese werden per Debugger von der Zielhardware ausgelesen. In einem nächsten Schritt kann aus diesen Werten die konkrete Zugriffsadresse berechnet werden. Da die Breite des Speicherzugriffs in dem Instruktions-Opcode kodiert ist, kann diese ebenfalls per Dekodierung ausgelesen werden.

Nach dem gleichen Prinzip ist es zusätzlich möglich, alle zugegriffenen Register einer Instruktion zu bestimmen. Hierbei ist lediglich der Opcode zu analysieren, welcher Ein- und Ausgabe-Register direkt kodiert. Diese Dekodierung kann allerdings offline durchgeführt werden, da im Instruktions-Trace bereits die Adressen der Instruktionen und in der entsprechenden Binär-Datei die zugehörigen Opcodes definiert sind. Aus diesem Grund ist die Aufzeichnung von Registerzugriffen keine Anforderung an das Tracing.

Um die aufgezeichneten Trace-Ereignisse in einen zeitlichen Bezug setzen zu können, wird im Kontext von Fail* nach Möglichkeit auch ein Zähler der vergangenen Prozessorzyklen ausgelesen. Dieser zeitliche Bezug ist insbesondere für die auf eine Kampagnendurchführung folgende Auswertung wichtig, da nicht alle Instruktionen die gleiche Zeit für die Ausführung benötigen. Beispielsweise kann ein Ein-/Ausgabe-Vorgang deutlich länger als die Durchführung einer Addition dauern, weshalb eine Anwendung der vorgestellten Metriken auf Basis von Instruktionsintervallen nicht sinnvoll ist. Ein Instruktionszähler wird vom Pandaboard direkt über das Performance Monitoring bereitgestellt und daher in FailPanda integriert. Dieser Zähler hat die Eigenschaft, dass er beim Anhalten durch OCD-Ereignisse wie Breakpoints ebenfalls angehalten wird, sodass der zeitliche Bezug durch die langen Haltezeiten nicht verloren geht. Problematisch bei der Verwendung des Zählers ist allerdings, dass dieser bei Verwendung von OCD-

Funktionen wie dem Schreiben in den Systemspeicher weiter läuft. Somit werden die Werte verfälscht und weisen nur noch eine beschränkte Aussagekraft auf.

Prinzipiell besteht alternativ zu der hier vorgestellten Strategie die Möglichkeit, den Programm-Trace mittels eines Simulator-Backends aufzuzeichnen. Fail* unterstützt in aktueller Version nur Gem5 als ARM-Backend. Abschnitt 5.1 wird zeigen, dass das Tracing bei FailGem5 um Größenordnungen schneller abläuft, als das Tracing von Fail-Panda. Der Ansatz ist allerdings problematisch, da grundsätzlich nicht von gleichem Verhalten von Simulator und Hardware ausgegangen werden kann. Da im Kontext dieser Arbeit keine Untersuchung der Gleichheit beider Verfahren der Trace-Aufzeichnung durchgeführt wurde, kann dieser Ansatz nicht weiter verfolgt werden.

3.4 Initialisierung

Die Experimentinitialisierung wurde in den bisherigen Beispielen üblicherweise mittels einer Zustandssicherung am Eintrittspunkt der zu untersuchenden Funktion durchgeführt. Diese Zustandssicherung kann zu Beginn eines neuen Experiments wieder eingespielt werden, um einen definierten Startpunkt für Experimente zu garantieren. Zu dem zu sichernden Systemzustand gehört

- der gesamte zugreifbare Speicherinhalt,
- die Werte aller Register sowie
- die Konfiguration von internen Komponenten wie etwa der MMU [PBKL95].

Da die Aufzeichnung und das Wiederherstellen eines solchen Zustands berechnungs- und speicheraufwendig ist und insbesondere bei der Übertragung vom Hostrechner an die Zielplattform zu Beginn jedes Experiments erhebliche Performanzverluste bedeuten würde, sollte dieses Verfahren optimiert oder nicht verwendet werden. Eine Reduktion des gesicherten Speicherinhalts kann durchgeführt werden, indem nur die Bereiche in einen Checkpoint einfließen, die seit der Initialisierung geschrieben wurden [PBKL95]. Die Übertragungszeit könnte stark reduziert werden, indem Checkpoints in einem abgesicherten Speicherbereich auf der Zielhardware gesichert und entsprechend von dem Zielsystem selbst geladen werden könnten. Somit würde die Übertragung über die Host-Target-Schnittstelle entfallen. Das Verfahren wurde im Kontext der Arbeit aufgrund des beschränkten Zeitrahmens jedoch nicht implementiert.

Eine weitere Möglichkeit zur Initialisierung eines Experiments liegt im Neustart der Hardware. Da eine Grundannahme dieser Arbeit ist, dass jegliche Quelle von Nichtdeterminismus eliminiert wurde, kann davon ausgegangen werden, dass das Verhalten der Hardware somit immer identisch ist. Nach einem Neustart muss auf den Eintritt in die zu untersuchende Funktion gewartet werden, was typischerweise mittels eines Breakpoints realisiert werden kann. Die Realisierung eines Neustarts kann mithilfe von OpenOCD auf drei verschiedene Arten erfolgen. So ist es möglich, das System direkt nach erfolgtem Neustart anzuhalten, erst nach der Initialisierung anzuhalten oder das System nach dem Neustart im Zustand der Programmausführung zu belassen.

Da das Zielsystem vor Ausführung der zu untersuchenden Funktion angehalten werden muss, ist es sinnvoll, unmittelbar nach Initialisierung des Systems zu halten. Es hat sich allerdings herausgestellt, dass die Verwendung der Funktionalitäten zum Anhalten des Systems nach Neustart und nach Initialisierung in einen nichtdeterministisch auftretenden Fehlerzustand führen. In diesem Zustand kann der Debugger nicht mehr auf das Zielsystem zugreifen und der Fehler kann nur durch Trennung des Zielsystems von der Versorgungsquelle überwunden werden. Aus diesem Grund wird im Systementwurf die Neuinitialisierung mithilfe des Neustarts im ausführenden Zustand durchgeführt.

Um dennoch ein rechtzeitiges Anhalten garantieren zu können, wird im Bootcode von FailPanda vor dem Ausführen der `main()`-Funktion der Zielapplikation eine im folgenden als *Sicherheitsschleife* bezeichnete unendliche Schleife eingefügt. Wird demnach das System neu gestartet und anschließend angehalten, so befindet sich die Ausführung garantiert vor oder innerhalb der Sicherheitsschleife. Die Schleife kann durch Schreiben des *Program Counter*-Registers mittels des OCD-Debuggers überwunden werden. Nach diesem Vorgehen befindet sich das System im Haltezustand vor Eintritt in die `main()`-Funktion, sodass die Ausführung eines Experiments beginnen kann. Beginnt der zu untersuchende Code allerdings noch nicht an dieser Stelle, so muss mittels der nachfolgend untersuchten Methoden zunächst eine Trace-Navigation durchgeführt werden.

3.5 Trace-Navigation

Eine wichtige Komponente eines Fehlerinjektionsexperiments besteht in der Navigation der Programmausführung zur dynamischen Instruktion, zu deren Ausführungszeitpunkt die Fehlerinjektion stattfinden soll. Bei einer dynamischen Instruktion handelt es sich um die k-te Ausführung der Instruktion mit der Speicheradresse x. Bei der Trace-Navigation gilt es demnach nicht nur, zur Zielinstruktion (definiert durch ihren Speicherort), sondern zu ihrer k-ten Ausführung zu navigieren.

Wie bereits gezeigt wurde, ist ein Einzelschrittdurchlauf bei der in dieser Arbeit untersuchten Hardware nicht praktikabel. Bei Verwendung der Break- und Watchpoints wird eine volle Ausführungsgeschwindigkeit ermöglicht, bis die Haltebedingung eintritt. In diesem Fall wird das Ereignis über den Debugger an das Hostsystem benachrichtigt. Da die Ausführungsgeschwindigkeit der Hardware deutlich höher ist, als bei einem Einzelschrittdurchlauf mithilfe des Debuggers, wird eine Lösung mittels der durch die OCD-Infrastruktur angebotenen Haltebedingungen (Breakpoints und Watchpoints) gesucht.

Die OCD-Komponente des Pandaboards bietet nicht die Möglichkeit, konditionale Haltepunkte anzulegen, mit denen beispielsweise ein automatisches Halten nach dem k-ten Auftreten der Zielinstruktion möglich wäre [FAF06a]. Aus diesem Grund ist es im Allgemeinen nötig, bei der Navigation zur Zielinstruktion das System mehrfach anzuhalten, um so beispielsweise so oft im Breakpoint der Zielinstruktion die Ausführung fortzusetzen, bis k Ausführungen erreicht wurden.

Im nachfolgenden Abschnitt 3.5.1 werden zunächst verschiedene Verfahren zur Navigation zu einer dynamischen Instruktion vorgestellt, um den aktuellen Stand der For-

```
1 for (i = 0; i < 64; ++i) {
2     for (j = 0; j < 64; ++j) {
3         sum += array[i+j];
4     }
5 }
```

```
 8: b    30 <func+0x30> ; (outer loop entry)
 c: add  r12, r0, r3    ; i+j
10: ldrb r12, [r12, r2] ; array[...]
14: add  r2, r2, #1     ; ++j
18: cmp  r2, #64        ; j < 64
1c: add  r1, r1, r12    ; sum += ...
20: bne  c <func+0xc>   ; (inner loop back edge)
24: add  r3, r3, #1     ; ++i
28: cmp  r3, #64        ; i < 64
2c: beq  38 <func+0x38> ; (outer loop exit)
30: mov  r2, #0         ; j = 0
34: b    c <func+0xc>   ; (outer loop back edge)
```

Abbildung 3.4: Beispielcode für die Untersuchung verschiedener Verfahren zur Trace-Navigation. Es ist sowohl der C-Code als auch der mithilfe des gcc übersetzte ARM-Assemblercode dargestellt. Die Stelle im C-Code und im Assemblercode an der das aktuelle Datum aus dem Feld ausgelesen wird, ist zur Übersichtlichkeit grau hinterlegt.

schung darzustellen. In Abschnitt 3.5.2 wird anschließend ein im Kontext dieser Arbeit neu entwickeltes Navigationsverfahren vorgestellt, welches ein deutliche Performanzsteigerung bringen kann. Über die grundsätzlichen Verfahren zur Trace-Navigation hinaus wird in Abschnitt 3.5.3 Checkpointing als generische Erweiterung zur Deckelung von Navigationskosten vorgestellt. Abschließend zeigt Abschnitt 3.5.4 mit der äquivalenzklassengewahren Trace-Navigation eine weitere Optimierung, welche die durch Äquivalenzklassenbildung gewonnenen Freiheitsgrade bei der Trace-Navigation ausnutzt, um eine Kostenminimierung zu erreichen.

3.5.1 Stand der Kunst

Da die Trace-Navigation einen kritischen Faktor in der Performanz des Gesamtsystems darstellt, wurde im Zuge der Entwicklung besonderes Augenmerk auf die Entwicklung von Algorithmen zur Bestimmung kostengünstiger Trace-Pfade gelegt. Abbildung 3.4 zeigt den C-Code eines kurzen Beispielprogramms und den zugehörigen übersetzten ARM-Assemblercode. Anhand dieses Kurzbeispiels sollen verschiedene Arten der Trace-Navigation zunächst vergleichend dargestellt werden. Das Programm führt eine Aufsummierung von Werten in einem im Speicher lokalisierten Feld in Form einer doppelt verschachtelten Schleife durch. Abbildung 3.5 zeigt die von den nachfolgend diskutierten Verfahren berechneten Pfade durch den Trace. Die mit den Ziffern 1-3 beschrifteten Pfade basieren ausschließlich auf der Verwendung von Breakpoints. Bei dem Verfahren,

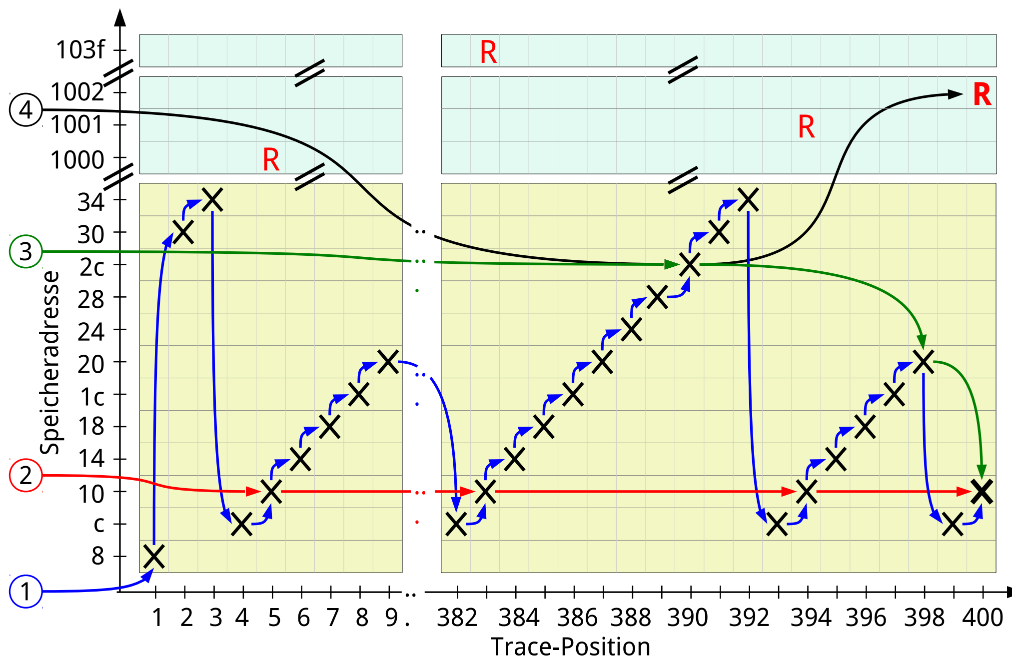


Abbildung 3.5: Schematische Darstellung eines Programm-Traces. Die Koordinaten repräsentieren Trace-Position (x-Achse) und Speicheradresse (y-Achse) einer ausgeführten Instruktion (X) bzw. eines Lese- (R) oder Schreibvorgangs (W). Die mit 1 bis 4 markierten Navigationen durch den Trace stellen die Verfahren Single-Stepping (1), Simple-Hopping(2), Smart-Hopping ohne Watchpoints (3) und Smart-Hopping mit Watchpoints (4) dar.

dessen Lösung mit der Ziffer 4 bezeichnet ist, wird zusätzlich auf Watchpoints zurückgegriffen.

Watchpoints können eine große Steigerung der Performanz bedeuten, wenn das zu bearbeitende Programm lange Schleifen beinhaltet, die auf wechselnden Speicherzellen (Iteration über Felder) arbeiten. Hierbei kann der Watchpoint ein Quereinstieg in eine bestimmte Schleifeniteration darstellen. Im Beispiel soll zur 400. dynamischen Instruktion navigiert werden. Dies ist die in Abbildung 3.4 grau hinterlegte Instruktion mit der Adresse `0x10` in ihrer 65. Ausführung; in Abbildung 3.5 ist die Zielinstruktion mit einem fettgedruckten „X" markiert.

Pfad 1 stellt das sogenannte *Single-Stepping* dar. Hierbei wird immer genau bis zur nachfolgend ausgeführten Instruktion gesprungen, wodurch der Pfad maximale Länge – in diesem Fall 400 Sprünge – aufweist. Bei Pfad 2 handelt es sich um ein Verfahren, welches im Kontext dieser Arbeit als *Simple-Hopping* bezeichnet wird. Dieses Verfahren nutzt einen Breakpoint, der auf die Zielinstruktion konfiguriert wird. Es wird solange die Ausführung fortgesetzt, bis die n-te Ausführung erreicht wurde, im Beispiel werden demnach 65 Sprünge benötigt. Simple-Hopping ist grundsätzlich auch um die Verwendung von Watchpoints erweiterbar, allerdings ist dies nur möglich, wenn die Ziel-

instruktion einen Speicherzugriff ausführt. Im Folgenden wird auf die Betrachtung von Simple-Hopping mittels Watchpoints verzichtet. Das mit den Pfaden 3 und 4 gezeigte Verfahren *Smart-Hopping* wird in Abschnitt 3.5.2 detailliert erläutert. Es nutzt eine beliebige Kombination von Breakpoints, um von jeder Ausgangsinstruktion den Sprung maximal möglicher Länge zu nehmen. So reduziert sich die Navigationslänge von Pfad 3 auf die Länge 3. Im Fall von Pfad 4 sind zusätzlich auch Watchpoints als Sprungziele erlaubt, sodass sich die Pfadlänge weiter auf 2 reduzieren lässt.

In Fail* wird üblicherweise Single-Stepping zur Trace-Navigation verwendet. Bei der Verwendung von Simulatoren als Backend ergibt sich kein Performanzvorteil durch die Verwendung von Break- oder Watchpoints, da sich durch ihren Einsatz die Ausführungszeit des Codes nicht wesentlich verbessert, wenn Break- und Watchpoints überhaupt in der modellierten CPU verfügbar sind. Aus diesem Grund ist die Beschränkung auf Single-Steps valide. Single-Stepping ist mit der Verwendung der OCD-Funktionalität des Pandaboards nicht praktisch einsetzbar, da die Anzahl an nötigen Sprüngen in Multiplikation mit den Kosten für einen Single-Step zu hohe Performanzverluste bedeutet (s. Gleichung 3.1).

In vielen artverwandten Arbeiten wird das Verfahren der Nutzung eines einzelnen Breakpoints (Simple-Hopping) vorgeschlagen [SZR03, FGAF06, PGLOGVE07]. Wie 3.5 zeigt, hat dieses Verfahren allerdings insbesondere bei Traces mit vielen Schleifendurchläufen Defizite. Soll innerhalb einer Schleifeniteration gehalten werden, so ist es nötig, in jedem vorherigen Schleifendurchlauf zu halten. Die Problematik des hohen Performanzverlusts durch häufig nötiges Anhalten ist auch in der wissenschaftlichen Literatur bekannt [RR99].

3.5.2 Smart-Hopping

Das im Kontext dieser Arbeit neu entwickelte Verfahren Smart-Hopping berechnet sogenannte *Sprungketten* für die Trace-Navigation unter Zuhilfenahme von Break- und Watchpoints. Die Grundannahme ist, dass mittels beliebig komplexer Kombinationen von Sprüngen ein kostenoptimaler Pfad über die Knoten eines Trace gefunden werden kann. Als Knoten N_i wird hierbei die Menge der Trace-Ereignisse $\{e_j | e_j \in N_i\}$ an einer Trace-Position i bezeichnet. Als Trace-Ereignisse werden sowohl aufgezeichnete Instruktionen als auch Speicherzugriffe bezeichnet und diese sind eindeutig bezeichnet durch ein Tupel $e_j = (a_j, t_j)$. Hierbei bezeichnet a_j die Adresse der ausgeführten Instruktion bzw. des zugegriffenen Speichers und t_j stellt den Ereignistyp dar. Erlaubte Ereignistypen sind: Instruktionsausführung, lesender Zugriff und schreibender Zugriff. Das Verfahren betrachtet nur Adressbereiche der Länge 1, sodass zu Beginn alle Trace-Ereignisse größerer Adressbreite auf Ereignisse der Breite 1 aufgeteilt werden.

Für das Smart-Hopping wird der Terminus der *Sichtbarkeit* von Trace-Ereignissen definiert. Die Sichtbarkeit definiert für jede Trace-Position eine Menge von Trace-Ereignissen, zu denen direkt mittels eines Sprungs navigiert werden kann. Alle nicht sichtbaren Ereignisse werden von einem Trace-Ereignis verdeckt, welches zwischen der Ausgangsposition und dem verdeckten Ereignis liegt und die gleichen Parameter aufweist. Abbildung 3.6 zeigt eine schematische Darstellung dieser Menge der Sichtbarkeit.

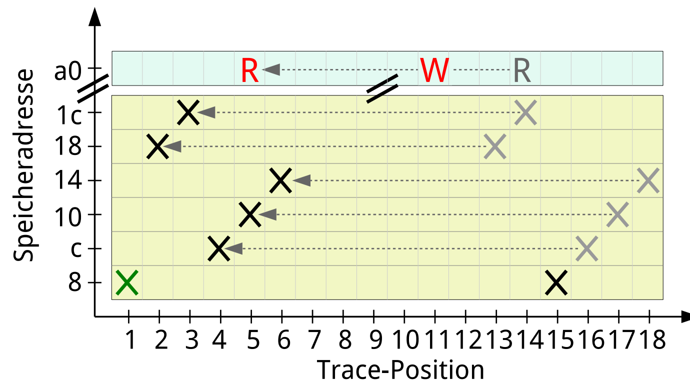


Abbildung 3.6: Schematische Darstellung der durch die Sichtbarkeit definierten Menge an Trace-Ereignissen, zu denen direkt navigiert werden kann. Das Ausgangs-Ereignis ist als grünes „X“ eingetragen. Sichtbare Instruktionsausführungen sind als schwarze „X“ dargestellt und sichtbare Speicherzugriffe werden durch rote „R“ oder „W“ symbolisiert. Nicht sichtbare Ereignisse sind grau dargestellt und ein zugehöriger Pfeil zeigt das verdeckende Ereignis.

Die Grundidee von Smart-Hopping sieht vor, dass eine Sprungkette von Beginn des Traces bis zur Position i genau dann minimale Länge aufweist, wenn in jeder Iteration des Verfahrens, welche den nächsten Navigationsschritt von der aktuellen Ausgangsposition j aus bestimmt, der weitestmögliche Sprung gewählt wird. Dieses Verfahren ist optimal, da durch die Wahl eines weitestmöglichen Schritts die maximale Sichtbarkeit nie geringer ist, als durch einen weniger weiten Schritt. Aufgrund der Linearität des Traces und der eingeschränkten Navigationsrichtung (immer in Richtung fortschreitender Trace-Positionen) gibt es hier keine Sonderfälle, die ein komplexeres Verfahren begründen könnten.

Um das entwickelte Verfahren praktisch einsetzbar zu machen, ist eine algorithmische Beschreibung erforderlich. Hierbei ist zu beachten, dass sowohl Berechnungszeit als auch Speicherverbrauch des Verfahrens zu minimieren sind. Da die Länge von zu verarbeitenden Traces mitunter sehr groß werden kann, ist es wichtig, dass nicht der gesamte Trace für jede zu berechnende Lösung durchsucht werden muss. Ein mögliches Verfahren zur Berechnung der kürzesten Sprungketten liegt in der Umformung des Traces in einen Graphen, dessen Knoten die einzelnen Trace-Ereignisse und dessen Kanten die Sichtbarkeit repräsentieren. Die optimale Lösung für das Problem der Sprungkettensuche könnte beispielsweise mittels Anwendung des Kürzeste-Wege-Algorithmus von Dijkstra bestimmt werden [Dij59]. Da allerdings in diesem Fall die Bedingung verletzt würde, dass nicht der gesamte Trace im Speicher repräsentiert sein sollte, wird der Speicherverbrauch des Verfahrens für lange Traces so hoch, dass die in üblichen Rechnern vorhandenen Ressourcen schnell erschöpft sind.

Im Zuge dieser Arbeit wurde das in Algorithmus 3.1 beschriebene Verfahren entwickelt, welches auf Grundlage des Ergebnisses für die Trace-Position $p-1$, einer Datenstruk-

Algorithmus 3.1 Beschreibung des Smart-Hopping Algorithmus in Pseudocode. Der Algorithmus berechnet eine optimale Sprungsequenz für alle Trace-Positionen, wobei das vorherigen Ergebnis wiederverwendet wird.

```
 1: type : enum ACCESSTYPE { execute, read, write }
 2: type : class ACCESS { INT address, ACCESSTYPE type }
 3: var cur_solution : list of tuples (INT trace_pos, ACCESS a)
 4: var access_last_seen : map ACCESS → INT trace_pos
 5: var cur_trace_events : list of type ACCESS
 6:
 7: cur_trace_pos ← 0
 8: while not at trace end do
 9:    read trace events at cur_trace_pos into cur_trace_events
10:    if any ACCESS a ∈ cur_trace_events not in access_last_seen then
11:       clear cur_solution, add (cur_trace_pos, a) to it
12:    else
13:       new_hop ←
14:        {x|x ∈ cur_trace_events ∧ access_last_seen[x] minimal }
15:       last_seen ← access_last_seen[new_hop]
16:       while length of cur_solution is > 1 do
17:          (pos_a, a) ← rightmost entry in cur_solution
18:          (pos_{a−1}, a − 1) ← 2nd to rightmost entry in cur_solution
19:          if last_seen ≤ pos_{a−1} then
20:             remove (pos_a, a) from cur_solution
21:          else
22:             break
23:          end if
24:       end while
25:       add (cur_trace_pos, new_hop) to cur_solution
26:    end if
27:    print cur_solution
28:    for all ACCESS x ∈ cur_trace_events do
29:       access_last_seen[x] ← cur_trace_pos
30:    end for
31:    cur_trace_pos ← cur_trace_pos + 1
32: end while
```

tur, die das letzte Auftreten jedes bislang aufgetretenen Trace-Ereignisses beinhaltet (im Folgenden als `access_last_seen` bezeichnet) und der Trace-Ereignisse der Trace-Position p eine Sprungkette mit minimalen Kosten für die Navigation zu Trace-Position p berechnet.

Die Funktionsweise des Algorithmus kann am Beispiel von Abbildung 3.7 erläutert werden. Das Beispiel hat keinen praktischen Bezug, sondern dient lediglich der Beschreibung einer Iteration des Algorithmus. Aus Gründen der Übersicht werden hier keine Speicherzugriffe durchgeführt. Zu jedem Berechnungsschritt ist der gesamte Trace in horizontaler Richtung ablesbar, wobei die der Lösung zugehörigen Trace-Positionen schwarz

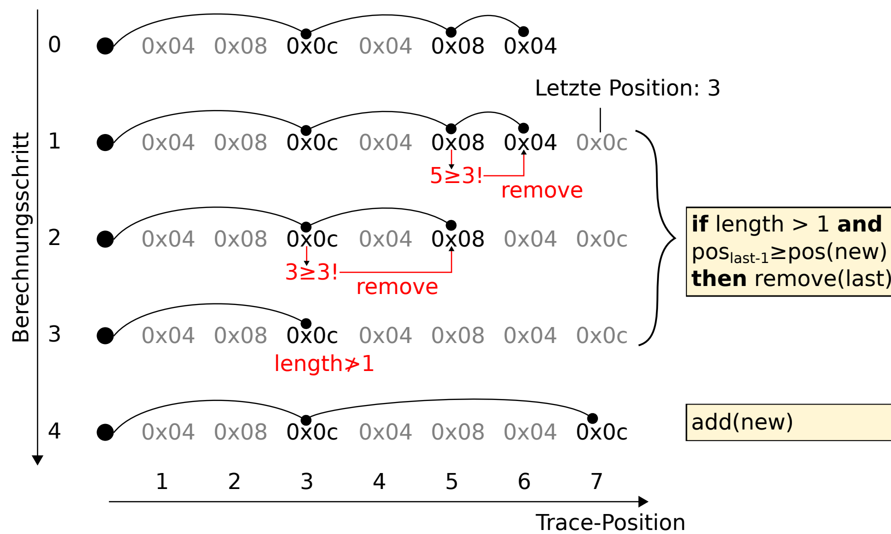


Abbildung 3.7: Beispielhafte Durchführung eines Iterationsschritts des Smart-Hopping-Algorithmus.

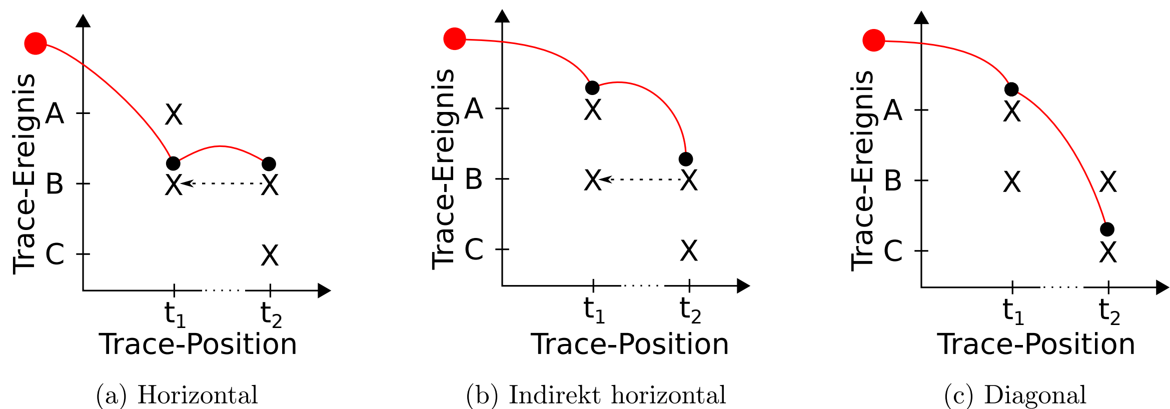
und alle restlichen Instruktionen grau gedruckt sind. Zu Beginn (Berechnungsschritt 0) ist bereits eine Lösung der Länge 3 für die vorherige Trace-Position angegeben. In einem nächsten Schritt wird die neue Zielinstruktion (mit der Adresse `0x0c`) betrachtet. Aus der Datenstruktur `access_last_seen` wird zunächst die Information extrahiert, dass die Instruktion `0x0c` zuletzt an Trace-Position 3 aufgetreten ist. Im Folgenden wird die Lösung der letzten Position schrittweise abgebaut, bis die Bedingung „Position des Vorgängers in der Lösung ist größer oder gleich 3" nicht mehr zutrifft. Somit wurde die vorherige Lösung soweit abgebaut, bis die neue Instruktion noch in der Sichtbarkeit des letzten Knotens liegt. Anschließend wird die neue Instruktion an die bestehende Sprungkette angehängt, sodass die neue optimale Lösung berechnet ist.

Für nachfolgende Berechnungen wird anschließend nach Algorithmus 3.1 noch die Datenstruktur `access_last_seen` für alle Trace-Ereignisse der aktuellen Position aktualisiert. Sind im Gegensatz zu dem gezeigten Beispiel mehrere Trace-Ereignisse an der aktuellen Trace-Position vorhanden, so wählt der Algorithmus vor der Reduktion des alten Ergebnisses das Ereignis, dessen Wert in `access_last_seen` minimal ist, da somit die Länge des neuen Sprungs maximiert wird.

Erweiterung durch Kostenmodell

Bei der Ausführungsfortsetzung im Haltezustand eines Break- oder Watchpoints ergibt sich bei dem dieser Arbeit zugrunde liegenden System folgende Einschränkung: Wird die Ausführung ohne weitere Vorkehrungen fortgesetzt, so wird das System unmittelbar im gleichen Break- oder Watchpoint erneut anhalten, demnach würde keine Instruktion ausgeführt.

Um dieses Problem zu umgehen, wird eine Ausführungsfortsetzung im Break- oder Watchpoint in OpenOCD durchgeführt, indem zunächst ein Single-Step ausgeführt wird.



(a) Horizontal (b) Indirekt horizontal (c) Diagonal

Abbildung 3.8: Schematisierung von horizontalen, indirekt horizontalen und diagonalen Sprüngen zwischen zwei Haltebedingungen an den Trace-Positionen t_1 und t_2.

Single-Steps sind so implementiert, dass aktuell aktive Break- oder Watchpoints übersprungen werden können[1]. Durch die zusätzlich nötige Verwendung eines Single-Steps bei der Fortsetzung der Ausführung im Haltezustand ergibt sich eine Haltezeit von etwa 135 ms. Die Verwendung von Single-Stepping ist jedoch nicht notwendig, wenn im Haltezustand der aktuelle Break- oder Watchpoint entfernt wird, da dieser die Ausführung demnach nicht augenblicklich erneut anhalten kann. Hierdurch reduziert sich die Haltezeit auf etwa 67 ms. In Abschnitt 5.1.1 werden genaue Kosten für die verschiedenen Arten von Sprüngen evaluiert.

Aufgrund dieses deutlichen Unterschiedes der Haltezeiten ist es notwendig, Sprünge zwischen zwei OCD-Haltebedingungen zu klassifizieren. Zunächst wird an dieser Stelle der Begriff der dynamischen Instruktion zu dem Terminus der *Trace-Position* verallgemeinert. Diese Verallgemeinerung ist notwendig, da zu einem Zeitpunkt im Trace zwar immer eine Instruktion ausgeführt wird, allerdings auch potenziell Speicherzugriffe durchgeführt werden. Beides kann in der Trace-Navigation als Sprungziel verwendet werden. Zu diesem Zweck werden die gleichzeitig an einer Trace-Position auftretenden Instruktionen und Speicherzugriffe allgemein als *Trace-Ereignisse* beschrieben.

Es wird weiterhin eine Klassifizierung von Sprüngen mittels Break- und Watchpoints in die Klassen der *horizontalen* und *diagonalen* Sprünge eingeführt. Die Bezeichnungen sind geometrisch an Abbildung 3.8 angelehnt. Ein horizontaler Sprung entspricht dem Übergang zwischen zwei Haltezuständen, wobei die für den Übergang gewählte Haltebedingung an beiden Positionen (vor und nach dem Sprung) aktiviert wird. Dies kann *direkt* geschehen, indem in beiden Fällen die gleiche Haltebedingung – beispielsweise ein Breakpoint – aktiviert wird (s. Abbildung 3.8a) oder *indirekt*, indem die Haltebedin-

[1]Die konkrete Implementierung eines Single-Steps basiert auf der Nutzung eines Breakpoints, welcher aktiv wird, sobald nicht die aktuelle Instruktion ausgeführt wird (sogenannter *Instruction Address Mismatch* [ARM11a]). Ist im aktuellen Zustand ein Breakpoint oder Watchpoint aktiv, so wird dieser zunächst entfernt und nach der Ausführung des Single-Steps wieder aktiviert.

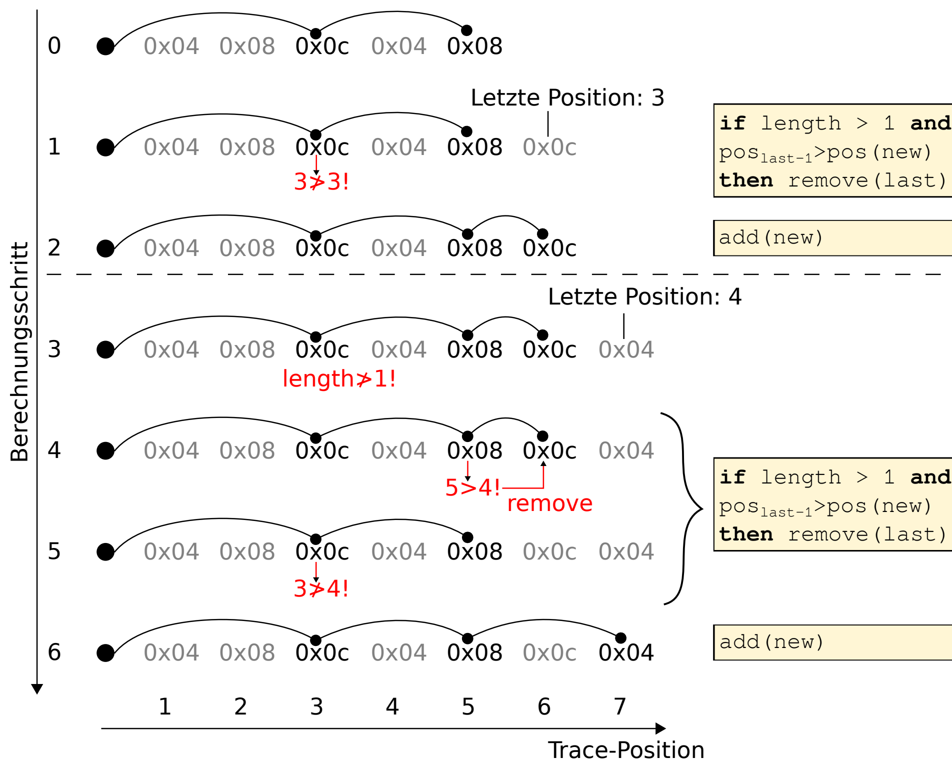


Abbildung 3.9: Beispielhafte Durchführung des Smart-Hopping-Algorithmus unter Berücksichtigung von Sprungkosten.

gung des Sprungziels auch bei der Sprungquelle aktiviert wird (s. Abbildung 3.8b). Von einem diagonalen Sprung wird hingegen gesprochen, wenn die neue Haltebedingung im alten Haltezustand nicht aktiviert wird (s. Abbildung 3.8c). Diese Arbeit geht von einem approximativen Kostenmodell des Verhältnisses 2:1 aus. Es wird demnach im Folgenden davon ausgegangen, dass ein horizontaler Sprung exakt die doppelten Kosten eines diagonalen Sprungs hat, um die entwickelten Verfahren zu vereinfachen.

Um demnach eine optimale Lösung berechnen zu können, müssen diese Kosten in der Berechnung berücksichtigt werden. Aufgrund des approximativen 2:1-Verhältnisses von Kosten horizontaler zu diagonalen Sprüngen wird die Lösung nie schlechter, wenn ein horizontaler Sprung durch zwei diagonale Sprünge ersetzt wird. Mittels dieser Grundannahme wird eine algorithmische Erweiterung eingeführt, welche horizontale Sprünge vermeidet, soweit dies möglich ist und dafür in Kauf nimmt, dass potenziell ein horizontaler Sprung durch zwei diagonale Sprünge ersetzt wird. Die nötige Änderung am Algorithmus liegt lediglich darin, dass die Bedingung in Zeile 19 von Algorithmus 3.1 von $\text{last_seen} \leq \text{pos}_{a-1}$ in $\text{last_seen} < \text{pos}_{a-1}$ umgewandelt wird. Diese Änderung erlaubt keine Entfernung eines Sprungs, wenn daher durch das Hinzufügen der neuen Instruktion ein horizontaler Sprung erzeugt werden würde. Durch die Änderung kann maximal ein zusätzlicher Sprung entstehen, weil im Fall von $\text{last_seen} = \text{pos}_{a-1}$ im nachfolgenden Schritt die Abbruchbedingung in jedem Fall erfüllt ist.

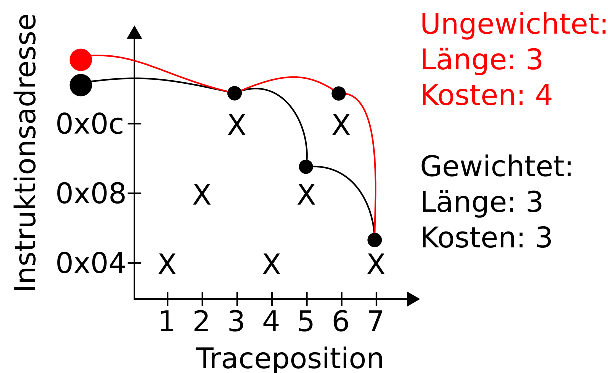


Abbildung 3.10: Sprung-Graph der in Abbildung 3.9 bestimmten kürzesten Pfade. In rot ist die Variante ohne Gewichtung, in schwarz die Variante mit Gewichtung eingezeichnet.

Abbildung 3.9 zeigt ein zu Abbildung 3.7 analoges Beispiel bei Anwendung des modifizierten Smart-Hopping-Algorithmus. Um die Vorteile der Modifikation erkennbar zu machen, ist es in diesem Beispiel nötig, zwei Iterationen des Verfahrens zu betrachten. In Berechnungsschritt 1 wird eine neu hinzuzufügende Instruktion betrachtet. Im Schritt der Reduktion des vorherigen Ergebnisses wird sofort abgebrochen, da das Verfahren anderenfalls einen horizontalen Sprung bestimmen würde. Die neue Lösung hat demnach die Länge 3 und auch Kosten von 3. Im nächsten Iterationsschritt wird der Vorteil des Verfahrens deutlich. Es wird erneut eine Reduktionsphase für die vorherige Lösung durchgeführt, wobei der letzte Sprung entfernt wird. Durch diesen Schritt hat die Lösung wieder die gleiche Länge, wie die Lösung der Basis-Version. Anschließend wird der neue Sprung angehängt, wodurch eine rein diagonale Lösung berechnet wird. Abbildung 3.10 zeigt die Lösungen von Smart-Hopping in der einfachen und in der um Gewichtungen erweiterten Version für die beiden in Abbildung 3.9 und 3.10 angegebenen Beispiele. Hier ist zu erkennen, dass die Lösung des gewichteten Verfahrens geringere Kosten, als die Lösung des ungewichteten Verfahrens aufweist.

3.5.3 Checkpointing

Die Evaluation der Navigationsverfahren in Abschnitt 5.1.2 wird zeigen, dass die entwickelten Verfahren zwar für viele Traces gut funktionieren, allerdings in Sonderfällen weiterhin hohe Kosten verursachen können. Für derartige Fälle ist es sinnvoll, die Navigationskosten deckeln zu können. Eine Realisierung dieser Anforderung könnte durch Checkpoints realisiert werden.

Checkpoints wurden bereits in Abschnitt 3.4 vorgestellt. Das Laden eines Checkpoints über die Debug-Schnittstelle verursacht zwar ebenfalls nicht vernachlässigbare Kosten, allerdings können die Kosten der regulären Trace-Navigation diese deutlich überschreiten. Im Fall einer Überschreitung eines Schwellwerts wird demnach ein neuer Checkpoint angelegt, der als Grundlage für nachfolgende Sprungketten verwendet wird, solange die Kosten der Sprungkette oberhalb des Grenzwertes liegen.

Beim Checkpointing muss beachtet werden, dass jede Zustandssicherung im Hintergrundspeicher des Hostrechners abgelegt werden muss und somit zu zusätzlichen Kosten führt. Das Verfahren muss demnach sicherstellen, dass die Anzahl an zu sichernden Checkpoints minimiert wird. Dies kann dadurch erreicht werden, dass der Schwellwert deutlich höher als die Kosten für das Laden eines Checkpoints angelegt wird. In diesem Fall werden neue Sprungketten zunächst auf Basis eines Checkpoints angelegt, bis die Gesamtkosten wiederum den Schwellwert übersteigen. Durch die Erhöhung des Grenzwertes erhöhen sich allerdings die erwarteten Kosten, sodass es sich hierbei um ein Optimierungsproblem mit gegenläufigen Bedingungen handelt. Es muss ein an die konkret vorhandenen Ressourcen angepasster Kompromiss zwischen hohem Speicherverbrauch und hohen Experimentlaufzeiten gefunden werden. Da im Kontext dieser Arbeit, wie in Abschnitt 3.4 bereits erläutert wurde, auf die Implementierung von Checkpointing für das Zielsystem verzichtet wurde, ist auch die hier vorgestellte Kostendeckelung nicht realisiert worden, jedoch wird im Folgenden ein theoretischer Entwurf des Verfahrens durchgeführt und in Abschnitt 5.1.2 die dadurch mögliche Kostenreduktion evaluiert.

In Bezug auf die Anwendung im Kontext von Smart-Hopping ist eine Realisierung von Checkpointing beispielsweise wie folgt möglich. Die Aufzeichnung von Checkpoints könnte während des Tracings durchgeführt werden, da hier jede Trace-Position abgelaufen wird und demnach an den gewählten Stellen auch das Anlegen von Checkpoints erfolgen kann. Zu diesem Zweck ist notwendig, dass parallel zum Tracing bereits der Smart-Hopping-Algorithmus zur Berechnung der jeweiligen Navigationskosten ausgeführt wird. Überschreiten aktuelle Kosten einen Schwellwert *CP-Threshold*, so wird ein Checkpoint angelegt. Das Kostenmodell von Smart-Hopping wird so erweitert, dass die aktuelle Lösung aus dem Laden eines Checkpoints besteht. Diese Lösung kann genau wie in der Basis-Variante wieder abgebaut und erweitert werden.

Bei dem Abbau einer Sprungkette ergibt sich allerdings potenziell das Problem, dass über den Checkpoint hinweg abgebaut wird und die dadurch bestimmte neue Lösung wieder Kosten knapp unter dem CP-Threshold aufweist. Es ist demnach möglich, dass eine große Anzahl an Checkpoints generiert wird, wenn die aktuellen Navigationskosten um den Wert des CP-Threshold schwanken – also häufig Checkpoints angelegt und für zukünftige Lösungen wieder verworfen werden. Um dieses Problem zu umgehen, wurde ein zweiter Schwellwert, der *Rollback-Threshold*, eingeführt. Dieser Grenzwert gibt an, wie viele Sprünge über den Checkpoint hinweg abgebaut werden müssen, damit der Vorgang erlaubt ist. Bleibt der Rollback-Threshold unterschritten, so wird die Lösung nur bis zum Checkpoint abgebaut. Der Schwellwert beeinflusst direkt die Anzahl generierter Systemabbilder.

3.5.4 Äquivalenzklassengewahre Navigation

Wie in Abschnitt 2.5 gezeigt wurde, existieren im Allgemeinen Mengen von zusammenhängenden Instruktionen, für die bei Injektion an einen Injektionsort x (Speicherstelle oder Register) nur ein Experiment durchgeführt werden muss, da x erst am Ende dieser Menge gelesen wird. Aus dieser Beobachtung folgt, dass alle Injektionsorte innerhalb einer solchen Äquivalenzklasse gleichwertig sind.

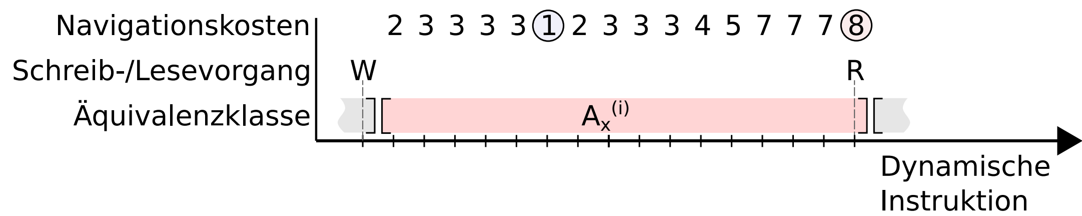


Abbildung 3.11: Beispielhaftes Schema einer Äquivalenzklasse mit Angabe der Navigationskosten zu jedem durch Pruning bestimmten möglichen repräsentativen Injektionszeitpunkt. Der üblicherweise in Fail* gewählte Zeitpunkt ist rot markiert und der in diesem Fall optimale Zeitpunkt wurde blau markiert.

Bei der Trace-Navigation hat sich allerdings gezeigt, dass die Navigation zu verschiedenen Instruktionen unterschiedliche Kosten verursacht. Somit kann der Freiheitsgrad der Wahl eines Injektionsortes innerhalb von Äquivalenzklassen dazu genutzt werden, die Navigationskosten weiter zu reduzieren, indem der Injektionsort immer dem Kostenminimum der entsprechenden Äquivalenzklasse entspricht.

Abbildung 3.11 schematisiert das Verfahren. Hier wird eine Äquivalenzklasse mit zu den zugehörigen Injektionszeitpunkten und den jeweiligen Navigationskosten gezeigt. Fail* verwendet bislang immer die am rechten Rand einer Äquivalenzklasse liegende Instruktion. In dem gezeigten Beispiel ist diese in rot eingezeichnet. Das Kostenminimum ist in blau eingezeichnet und würde durch die vorgestellte Erweiterung gewählt werden. Dieses Verfahren ist grundsätzlich auf alle Navigationsarten anwendbar, welche die Kosten für verschiedene Navigationspfade berechnen.

3.6 Fehlerinjektion

Die Grundanforderungen an Fehlerinjektion bestehen darin, dass lesender und schreibender Zugriff auf Speicher und Prozessorregister besteht. OpenOCD unterstützt diese Funktionalitäten direkt für das Pandaboard, sodass sie problemlos umgesetzt werden können. Wird in einem System mit Speicher-Caches ein Fehlermodell vorausgesetzt, welches zwar Fehler im Speicher aber nicht in den zugehörigen Caches vorsieht, so kann es zusätzlich nötig sein, Fehler in den Speicher zu injizieren, ohne eine Cache-Invalidierung zu provozieren. Abbildung 3.2 zeigt die Speicherhierarchie des OMAP4460. Hier ist zu erkennen, dass die Speicheranbindung der CPU-Subsysteme für die Cortex-A9-Kerne und für die Cortex-M3-Kerne nach einer jeweils eigenen Cache-Hierarchie mittels eines Interconnects realisiert ist. Es hat sich gezeigt, dass bei schreibenden Speicherzugriffen aus dem M3-Subsystem keine Invalidierung der Caches im A9-Subsystem durchgeführt wird. Somit ist es möglich, die Architektur des OMAP4460 für Speicherinjektion ohne Cache-Invalidierung im Zielsystem (Cortex-A9) auszunutzen, indem mittels OpenOCD ein schreibender Speicherzugriff via einem der Cortex-M3-Kerne durchgeführt wird. Zu

diesem Zweck muss dieser Prozessorkern allerdings im FailPanda-Bootcode initialisiert und anschließend angehalten werden.

3.7 Nachuntersuchung

In der Nachuntersuchung muss zunächst eine Erkennung der erfolgreichen Terminierung der Zielsoftware möglich sein. Dies ist unter Zuhilfenahme eines Breakpoints auf der Austrittsinstruktion der Funktion möglich. Diese Lösung ist sehr performant, da die Nutzung von Breakpoints eine volle Ausführungsgeschwindigkeit bis zum Auslösen des Ereignisses gewährleistet. Die Verwendung von Breakpoints stellt keine besondere Anforderung an die Entwicklung in dieser Arbeit, da die Funktionalität von OpenOCD grundsätzlich unterstützt wird.

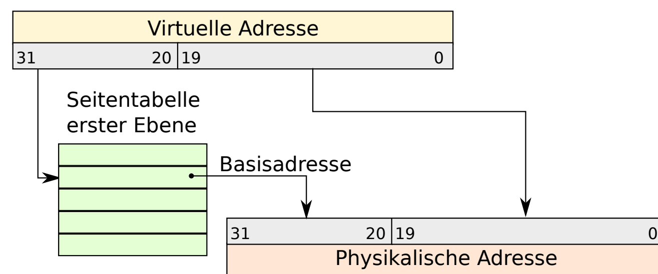
Im Falle einer erfolgreichen Terminierung gilt es, das von der Zielsoftware berechnete Ergebnis zu untersuchen. Dieses Ergebnis kann üblicherweise entweder im Speicher oder in den CPU-Registern ausgelesen werden. Die Funktionalität zum Lesen dieser Bereiche ist im hier entwickelten System bereits vorhanden. Die Performanz des Auslesens ist nicht kritisch, da der Vorgang in jedem Experiment maximal einmal durchgeführt werden muss.

Eine weitere Möglichkeit zur Betrachtung von Ergebnissen liegt in der Auswertung der Ausgabe der Zielapplikation über eine serielle Schnittstelle. Da das Pandaboard über eine serielle Schnittstelle verfügt, ist die Verwendung möglich, im Kontext dieser Arbeit wird allerdings darauf nicht zurückgegriffen, da die Möglichkeit der Ergebnisprüfung durch Auslesen des Speichers ausreichend ist.

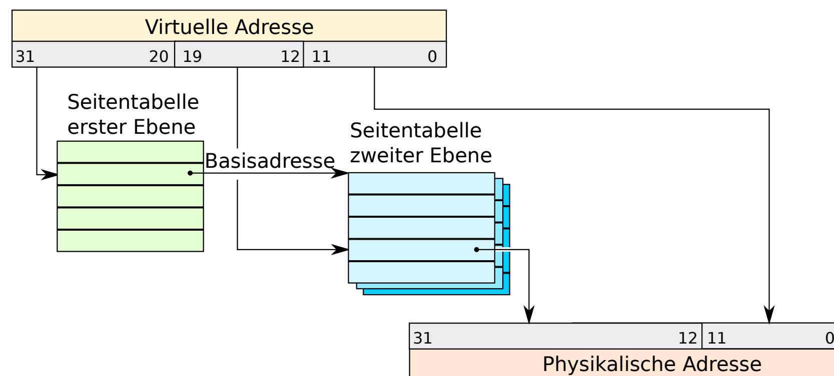
Die Erkennung von Traps und Interrupts wird durchgeführt, indem Breakpoints in den entsprechenden Behandlungsroutinen eingefügt werden. Zu diesem Zweck werden die nötigen Behandlungsroutinen im FailPanda-Bootcode ohne Funktion initialisiert. Es ist somit keine Auflösung, aber das Erkennen von Traps möglich. Mithilfe eines Breakpoints, der eine Instruktion überwacht, in die alle Behandlungsroutinen einen Sprung ausführen, kann ein Trap-Ereignis an Fail* signalisiert werden. Da im Kontext dieser Arbeit bei der Nachuntersuchung der Typ des Traps zunächst nicht relevant ist, reicht die Verwendung eines Hardware-Breakpoints. Es wäre allerdings auch möglich, Software-Breakpoints zu verwenden, um weniger Hardware-Breakpoints dauerhaft zu belegen. Insbesondere wäre es dadurch möglich, alle Trap-Typen zu unterscheiden.

Die Erkennung von Speicherzugriffen außerhalb des erlaubten Bereichs ist bei dem vorgestellten OMAP4460 zunächst schwierig, da nur vier Watchpoints zur Verfügung stehen. Jeder dieser Watchpoints kann einen Speicherbereich der Breite 4 Byte kontrollieren (s. Abschnitt 3.2.1). Auch unter Anwendung des Address Range Masking ist keine umfassende Überwachung des 32-Bit-Adressraums möglich.

Ein alternativer Ansatz zur Beobachtung von Speicherzugriffen auf konfigurierbaren Speicherbereichen liegt in der Verwendung der Memory Management Unit (kurz: MMU) des Systems [BH05]. Eine MMU dient der virtuellen Speicherverwaltung für Prozesse in einem Betriebssystem [Tan07]. Hierbei hat jeder Prozess grundsätzlich Zugriff auf einen virtuellen Speicher, welcher den gesamten Adressraum abdeckt. Die MMU führt



(a) Abbildung einer Section



(b) Abbildung einer Page

Abbildung 3.12: Schematische Darstellung der Abbildung einer virtuellen Adresse auf eine Section (s. 3.12a) oder auf eine Page (s. 3.12b) beim OMAP4460 nach [Tex12].

eine Übersetzung von virtuellen Adressen auf physikalische Adressen durch. Zu diesem Zweck wird der physikalische Speicher in sogenannte Kacheln und der virtuelle Speicher in sogenannte Seiten eingeteilt, sodass Kacheln und Seiten identische Größe aufweisen. Eine Seite kann entweder nicht verknüpft oder einer Kachel zugeordnet sein. Die Verknüpfung wird vom Betriebssystem berechnet und in einer für jeden Prozess angelegten sogenannten Seitentabelle eingetragen. Eine Seitentabelle ist ein Objekt im Speicher, welches direkt von der MMU ausgelesen wird. Wird vom gegenwärtig aktiven Prozess auf eine virtuelle Adresse zugegriffen, der in der Seitentabelle keine Kachel zugeordnet ist, so löst die MMU eine Ausnahme (im Fall von ARMv7 ist dies ein sogenannter *Data Abort*) aus, damit das Betriebssystem die Verknüpfung zu einer Kachel herstellen kann. Anschließend wird die Ausführung an der Instruktion, die den Data Abort ausgelöst hat, erneut begonnen. Dieses Verfahren kann dazu ausgenutzt werden, Speicherzugriffe zu erkennen, wenn sie im Adressbereich einer nicht verknüpften Seite durchgeführt werden. Eine nötige Grundannahme ist jedoch, dass der Experimentcode die MMU nicht verwendet.

In der konkreten Hardware wird eine hierarchische Struktur von Seitentabellen verwendet, um einen möglichst geringen Speicherverbrauch durch die angelegten Tabellen zu verursachen, wobei es maximal zwei Hierarchie-Ebenen gibt. ARMv7 definiert somit

zwei grundsätzliche Arten sogenannter *Deskriptoren*, welche mögliche Einträge in Seitentabellen definieren. Die Deskriptoren erster Ebene verwalten einen Speicherbereich der Größe von 1 MB und können einer der folgenden Typen sein [ARM11a]:

- **Fault:** Wird dieser Eintrag bei dem Versuch der Adressübersetzung gefunden, so wird ein Data Abort ausgelöst, in dessen Behandlungsroutine ein entsprechender Eintrag hinzugefügt werden kann, damit anschließend ein erfolgreicher Speicherzugriff ausgeführt werden kann. Dieser Typ von Einträgen wird demnach in FailPanda dazu verwendet, große 1 MB-Bereiche des Speichers auf Zugriffe zu überwachen.

- **Section:** Wird ein gesamter 1 MB-Bereich des Speichers benötigt, so kann bereits auf der ersten Ebene eine sogenannte Section angelegt werden. Der Deskriptor enthält die Basisadresse der zugehörigen Speicherkachel. Abbildung 3.12a zeigt schematisch die Übersetzung einer virtuellen Adresse in eine physikalische Adresse im Fall einer Section. Es wird demnach mit den oberen 12 Bit der Adresse der entsprechende Deskriptor in der Seitentabelle adressiert. Dieser stellt bei einem Section-Deskriptor eine an 1 MB-Adressen ausgerichtete Basisadresse dar, auf welche der Wert der unteren 20 Bit der virtuellen Adresse aufaddiert wird. In FailPanda wird die Verwendung von Sections bevorzugt, da mit wenig Konfigurationsaufwand ein großer Speicherbereich untersucht werden kann.

- **Page:** Wird eine feingranulare Verwaltung des virtuellen Speichers benötigt, so wird an dieser Stelle die Basisadresse einer Seitentabelle der zweiten Ebene eingetragen. Im Kontext der Arbeit wird nach Möglichkeit auf Verwendung von Seiten verzichtet, da ihre Verwendung einen potenziell hohen Konfigurationsaufwand bedeutet. Wird beispielsweise der Bereich, den eine Section abdeckt, mithilfe von Pages beobachtet, so müssen 256 Deskriptoren auf der zweiten Ebene geschrieben werden. Die gleiche Aufgabe kann durch einen einzigen Section-Deskriptor bearbeitet werden.

- **Supersection:** Bei Supersections handelt es sich um Deskriptoren, die einen 16 MB großen Bereich im virtuellen Adressraum auf einen entsprechend großen Bereich im physikalischen Speicher abbilden. Da die Seitentabelle erster Ebene allerdings so adressiert wird, dass ein 16 MB-Bereich sich über 16 Einträge erstreckt, müssen tatsächlich 16 identische Einträge in der Tabelle vorhanden sein. Supersections bieten einen Vorteil bei der Verwendung eines TLB, dessen Funktion im Folgenden noch kurz erläutert wird.

Wird mittels eines Page-Deskriptors eine Seitentabelle zweiter Ebene adressiert, so kann diese wiederum folgende Typen von Deskriptoren enthalten [ARM11a]:

- **Fault:** Dieser Eintrag hat die gleiche Funktion wie das Pendant auf erster Tabellen-Ebene. Es wird ein Data Abort zur Behandlung der fehlenden Abbildung ausgelöst.

- **Small Page:** Diese Deskriptoren definieren die Abbildung eines 4 kB großen Speicherbereichs. Abbildung 3.12b zeigt die zweischrittige Übersetzung einer virtuellen

Adresse in die zugehörige physikalische Adresse mithilfe von hierarchischen Seitentabellen. Im Fall von FailPanda werden diese Seiten eingesetzt, wenn die Grenzen des zu beobachtenden Bereichs innerhalb einer Section verlaufen. In diesem Fall wird die entsprechende Section mit Seiten abgedeckt, wobei der zu beobachtende Anteil mit Fault-Deskriptoren gefüllt wird.

- **Large Page:** Large Pages sind ähnlich wie Supersections. Sie decken einen 64 kB großen Speicherbereich ab. Auch hier liegt der Vorteil bei der Verwendung nur in der Optimierung der TLB-Operation, weshalb Large Pages nicht weiter beachtet werden.

Mittels entsprechenden Fault- sowie Section- und Page-Einträgen auf erster Ebene und Fault- oder Small Page-Einträgen auf zweiter Ebene lässt sich die Beobachtung von Zugriffen auf definierten Speicherbereichen realisieren. Die Granularität liegt hierbei zunächst bei 4 kB, weshalb eine nachträgliche Untersuchung ergeben muss, ob der Zugriff tatsächlich außerhalb des erlaubten Bereichs erfolgt ist.

Der sogenannte Translation Lookaside Buffer (kurz: TLB) ist ein Baustein, der dafür sorgt, dass die MMU nicht für die Auflösung jedes Speicherzugriffs weitere Speicherzugriffe auf die Seitentabelle durchführen muss, da unter dieser Voraussetzung die Performanz stark reduziert werden würde. Der TLB ist ein spezialisierter Cache, welcher Einträge aus der Seitentabelle enthält [ARM11b]. Damit bei Veränderungen in den Seitentabellen die entsprechende Information im TLB aktualisiert werden kann, werden spezielle Funktionen zur Invalidierung angeboten. So kann beim Schreiben eines Seiten-Deskriptors der zugehörige Deskriptor invalidiert werden, bei einem Komplettwechsel der Seitentabelle muss jedoch eine volle Invalidierung des TLB erfolgen. Durch die Verwendung der gezielten Invalidierung eines einzelnen Eintrages bleibt demnach die Funktion aller restlichen Einträge erhalten, wodurch die Performanzsteigerung durch den TLB aufrechterhalten werden kann. Die Invalidierungsfunktionen werden mittels des Schreibens spezieller Koprozessor-Register aktiviert. Bei den nachfolgend beschriebenen Operationen in Bezug auf die Seitentabelle ist stets eine Invalidierung entsprechender TLB-Einträge durchzuführen, wenn eine bereits abgebildete Seite von einer Seitentabelle entfernt wird.

Die grundsätzliche Idee bei Verwendung der MMU sieht vor, dass ein lineares MMU-Mapping realisiert wird. Das bedeutet, dass alle virtuellen Adressen auf die gleiche physikalische Adresse abgebildet werden. Somit kann der Speicher verwendet werden, als sei die MMU nicht eingeschaltet. Alle Seiten, die einem zu überprüfenden Bereich zugeordnet sind, werden auf entweder erster oder zweiter Ebene mit einem Fault-Deskriptor versehen. Somit ist sichergestellt, dass bei einem nicht erlaubten Speicherzugriff ein Data Abort ausgelöst wird, dessen Ausführung mithilfe eines Breakpoints erkannt und gestoppt wird. Eine Behandlungsroutine für einen Data Abort erhält über spezielle Register Informationen über Adresse des Zugriffs, die Zugriffsart (lesend oder schreibend) und die auslösende Instruktion. Die Routine modifiziert demnach die Seitentabelle, sodass der Zugriff erfolgen kann und führt einen Sprung zurück zu der auslösenden Instruktion aus.

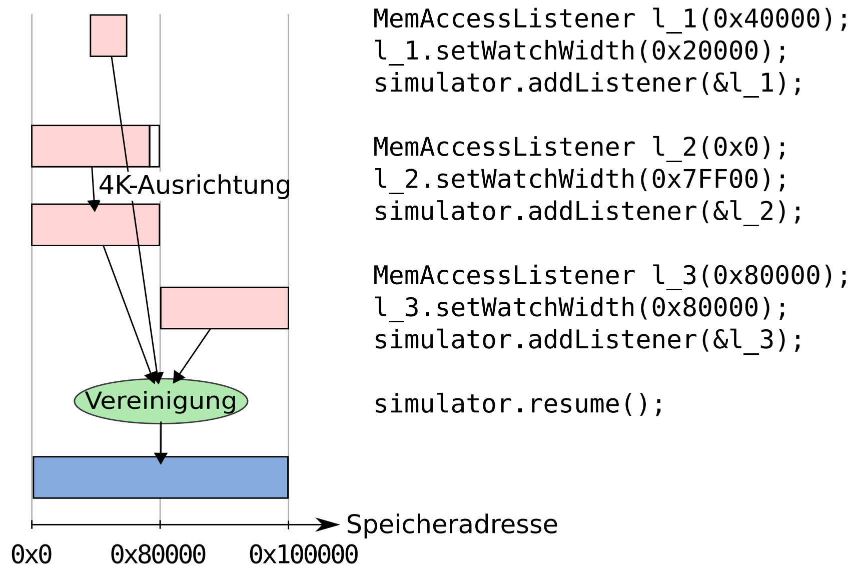


Abbildung 3.13: Schematische Darstellung der Zusammenfassung mehrerer beobachteter Speicherbereiche für die Konfigurierung der MMU.

Das entwickelte Verfahren sieht das nachfolgend erläuterte Vorgehen vor, um sowohl das Anlegen als auch das Entfernen eines zu beobachtenden Speicherbereichs zu ermöglichen. Wird ein neuer `MemoryAccessListener` über die Schnittstelle des `SimulatorControllers` hinzugefügt, so wird zunächst geprüft, ob die Länge des zu überwachenden Bereichs kleiner oder gleich 4 Byte ist. Ist dies der Fall, so wird zusätzlich geprüft, ob noch ein freier Watchpoint im System verfügbar ist. Gegebenenfalls wird eine Beobachtung durch einen Watchpoint ausgeführt. In allen anderen Fällen wird die Prüfung auf Speicherzugriffe mithilfe der MMU durchgeführt. Zu diesem Zweck nutzt FailPanda eine Liste der aktuell mittels MMU überwachten Bereiche. Beim Hinzufügen oder Entfernen eines Bereichs über den `SimulatorController` wird zunächst nur diese Liste aktualisiert. Da dieser Vorgang vor der Fortsetzung des Zielsystems mehrfach ausgelöst werden kann, lassen sich somit mehrere nötige Konfigurationen der MMU vereinen.

Zu diesem Zweck müssen die hinzugefügten Bereiche zunächst auf Seitengröße (also 4 kB-Adressen) ausgerichtet und erweitert werden, da eine detailliertere Speicherbetrachtung mittels der MMU nicht möglich ist. Es ist potenziell möglich, dass überlappende Bereiche überwacht werden müssen. Abbildung 3.13 zeigt beispielhaft einen solchen Fall, in dem l_1 von l_2 überlagert wird. Für die Konfiguration der MMU ist die Anzahl der Listener nicht relevant, und so werden derartige überlappende Speicherbereiche zunächst vereint. Anschließend ist es notwendig, aneinander angrenzende Bereiche ebenfalls zu vereinen, um Bereiche maximaler Größe zu untersuchen, wodurch ein potenziell geringerer Konfigurationsaufwand entsteht, wenn Sections verwendet werden können. Abbildung 3.13 zeigt schematisch die Zusammenfassung mehrerer zu beobachtender Speicherbereiche. In rot sind zunächst die neu hinzugefügten Speicherbereiche in Kombination mit dem zugehörigen Experimentcode gezeigt. Im Fall des Listeners l_2 wird zusätzlich eine Ausrichtung an 4 kB-Adressen durchgeführt. Bei Aufruf der Funktion `simulator.resume()`

wird die Vereinigung durchgeführt, wobei sowohl Überlagerungen von Bereichen, als auch direkte Nachbarschaften erkannt werden. Der letztendlich zu konfigurierende Bereich ist blau markiert und in diesem Fall zusammenhängend.

In diesem Beispiel ist der Vorteil des Vorgehens deutlich zu erkennen. Für die Beobachtung aller einzelner Speicherbereiche wäre es nötig, die 256 zu einer Section gehörenden Seiten zu konfigurieren, durch die Zusammenfassung kann der Gesamtbereich allerdings mithilfe einer Section abgedeckt werden. Das Entfernen von beobachteten Speicherbereichen funktioniert analog. Nach Entfernung entsprechender Bereiche aus der Liste mit allen Beobachtungsanforderungen wird eine Differenzbildung zur aktuellen MMU-Konfiguration durchgeführt und entsprechende Bereiche werden wieder dem linearen Mapping zugeführt.

Ist die aktuell zu schreibende MMU-Konfiguration ermittelt, so wird diese im Speicher der Zielhardware geschrieben. Der Speicherort der Seitentabellen wird von dem spezialisierten Bootcode von FailPanda definiert. Die Startkonfiguration nach dem Systemstart stellt eine vollständige lineare Abbildung dar. Es wird initial so viel Speicher belegt, dass eine vollständige Konfiguration des gesamten Adressraums mit Seiten durchgeführt werden kann. Dies bedeutet eine Speicherbelegung von $4096 \cdot 4\,\text{Byte} = 4\,\text{kB}$ für die Tabelle erster Ebene und $4096 \cdot 256 \cdot 4\,\text{Byte} = 1\,\text{MB}$ für alle Tabellen zweiter Ebene. Das Schreiben der Tabellen erfolgt über die reguläre Schnittstelle zum Schreiben im Zielspeicher. Wie Untersuchungen gezeigt haben, ist allerdings das Schreiben zusammenhängender Speicherbereiche deutlich performanter, als die Durchführung mehrerer Einzelschreibvorgänge (s. Abschnitt 5.1.1). Daher ist klar, dass Schreibvorgänge für performantes Verhalten zwingendermaßen weitestmöglich zusammengefasst werden müssen. Nach dem Schreiben der Einträge in der Seitentabelle werden noch entsprechende TLB-Einträge invalidiert.

Hält das Gesamtsystem in einer Data Abort Behandlungsroutine, so sind folgende Informationen über den entsprechenden Speicherzugriff bekannt:

- Die Adresse des Zugriffs aus dem Koprozessorregister `DFAR,`
- die Art des Zugriffs (lesend oder schreibend) aus dem Koprozessorregister `DFSR` und
- die zugehörige Instruktionsadresse aus dem Abort Link-Register.

Für eine erfolgreiche Verarbeitung des Ereignisses eines Zugriffs ist es allerdings nötig, zusätzlich eine Information über die Länge des Speicherzugriffs zu erhalten. Zu diesem Zweck wird eine Dekodierung der verursachenden Instruktion wie in Abschnitt 3.3 durchgeführt, wofür zunächst die auslösende Instruktion aus dem Speicher geladen und anschließend die zur Dekodierung nötigen Register ausgelesen werden. Durch die Instruktionsdekodierung werden alle Informationen des Speicherzugriffs exakt berechnet und können anschließend an die Schnittstelle des `SimulatorControllers` weitergegeben werden.

Greift das Zielsystem auf einen durch die MMU beobachteten Speicherbereich zu, so wird ein Data Abort ausgelöst, welcher von FailPanda erkannt wird. Anschließend muss,

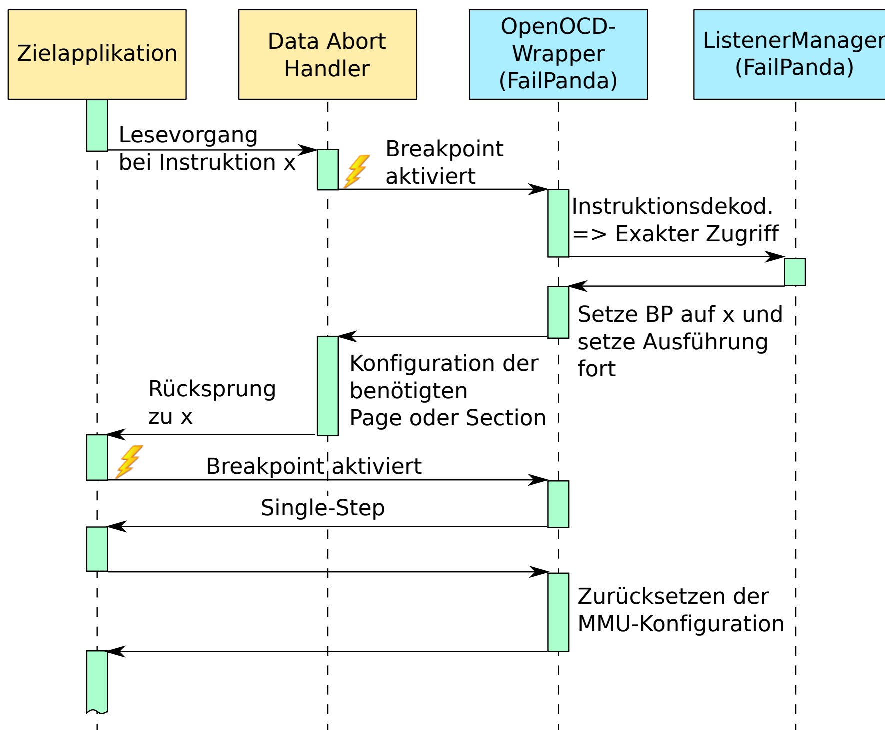


Abbildung 3.14: Sequenzdiagramm zur Erkennung eines mithilfe der MMU beobachteten Speicherzugriffs. In gelb sind die Softwarekomponenten auf dem Pandaboard markiert und in grün die Komponenten von FailPanda. Obwohl zwei unabhängige Systeme beschrieben werden, ist immer nur ein Kontrollfluss aktiv. Dies schematisiert den jeweiligen Wartevorgang auf ein Ereignis von der Gegenseite.

damit das Zielsystem weiter ausgeführt werden kann, der Speicherzugriff erlaubt werden. Abbildung 3.14 schematisiert das zu diesem Zweck entwickelte Verfahren in Form eines Sequenzdiagramms. Zunächst fügt die spezialisierte Abort-Behandlungsroutine von FailPanda einen entsprechenden Eintrag in die Seitentabelle ein. Damit allerdings nach diesem Speicherzugriff weitere relevante Speicherzugriffe erkannt werden können, muss dieser Eintrag nach Ausführung der Instruktion direkt wieder entfernt werden. Zu diesem Zweck wird in FailPanda ein Breakpoint auf die entsprechende Instruktion eingetragen. Bei Aktivierung dieses Breakpoints nach dem Auslösen eines Speicherzugriffs-Ereignisses der MMU wird ein Single-Step ausgeführt, damit die Ausführung sich auf der nachfolgenden Instruktion befindet. An dieser Stelle wird die Änderung an der Seitentabelle wieder rückgängig gemacht, um erneute Speicherzugriffe in der entsprechenden Seite zu erkennen. Da der entsprechende Eintrag an dieser Stelle auch im TLB vorhanden sein sollte, muss dieser entsprechend invalidiert werden. Der gesamte Vorgang der Behandlung eines Data Aborts kann somit einen kritischen Faktor in der Performanz des Gesamtsystems darstellen, da hier in zwei Breakpoints gehalten, ein Single-Step, ein Speicherzugriff und eine Invalidierung des TLB ausgeführt werden müssen. Zusätzlich muss für die nötige

Instruktionsdekodierung ein weiterer Speicherzugriff durchgeführt und mehrere Register ausgelesen werden.

Die Behandlungsroutine selbst benötigt Zugriff auf einige Bereiche im Speicher, insbesondere auf die zu modifizierende Seitentabelle. Da auch diese Speicherbereiche potenziell durch die MMU maskiert sein können, ist zu Beginn der Behandlung des Traps eine Abschaltung der Funktionalität der MMU durchzuführen. Am Ende der Ausführung der Behandlungsroutine kann die MMU wieder in den aktiven Zustand zurückversetzt werden.

Es ist zu beachten, dass ein Zielsystem potenziell aufgrund eines injizierten Fehlers schreibende Speicherzugriffe auf der Seitentabelle durchführen kann. Gleiches gilt insbesondere auch für die definierten Trap-Behandlungsroutinen. Diese speziellen Speicherbereiche, die von FailPanda genutzt werden, sollten demnach geschützt werden, indem eine MMU-Konfiguration nur lesende Zugriffe im user-mode erlaubt, in welchem das Zielsystem ausgeführt wird. Im Kontext dieser Arbeit wurde eine derartige Absicherung allerdings nicht realisiert.

Der Aufwand zur Behandlung eines MMU-Ereignisses wird insbesondere dann kritisch, wenn ein aktiv von der Applikation genutzter Speicherbereich durch die Grobgranularität der MMU-Überwachung innerhalb der Überwachung liegt, obwohl Speicherzugriffe in diesem Bereich nicht relevant für die Nachuntersuchung sind. Im Folgenden wird dieses Problem als *falsch-positives Ergebnis* bei der Speicherüberwachung mithilfe der MMU bezeichnet. In diesem Fall ist es potenziell möglich, dass die MMU-Überwachung sehr häufig aktiv wird, obwohl die generierten Informationen nicht für das Experiment von Bedeutung sind, wodurch erhebliche Performanzdefizite entstehen können [BH05].

3.8 Zusammenfassung

In diesem Kapitel wurden zunächst grundlegende Anforderungen an den Entwurf eines OCD-basierten Fehlerinjektionssystems vorgestellt. Auf Basis dieser Anforderungen wurde ein Entwurf durchgeführt, dessen Implementierung im nachfolgenden Kapitel erläutert wird. Grundsätzlich baut der Entwurf auf der Nutzung des Pandaboard ES auf. Dieses Entwicklungsboard wird mithilfe des Debuggers Flyswatter2 und der OCD-Schnittstellen-Software OpenOCD an einen Hostrechner angebunden. Der detaillierte Entwurf der Funktionalitäten sieht zunächst die Trace-Aufzeichnung zur Generierung der für die folgenden Funktionen nötigen Datenbasis vor. Die während eines Experiments verwendeten Funktionen wurden in die vier Phasen der Systeminitialisierung, der Trace-Navigation, der eigentlichen Injektion von Fehlern und der Nachuntersuchung unterteilt. Der Entwurf ist insbesondere bezüglich einer Minimierung der Experiment-Zeit optimiert. So wurde beispielsweise die als kritischer Laufzeitfaktor erkannte Trace-Navigation detailliert betrachtet und es konnte ein Verfahren zur Generierung von Sprungketten kürzester Länge auf Basis von Break- und Watchpoints entwickelt werden. Auf Basis der Entwicklung eines Kostenmodells für verschiedene Arten von Sprüngen auf der verwendeten Hardware war es möglich, das Verfahren weiterhin in Hinsicht auf eine Minimierung der Laufzeit zu optimieren.

4 Realisierung

Dieses Kapitel erläutert die konkrete Realisierung der in Kapitel 3 entworfenen Funktionen und der Architektur von FailPanda sowie der Einbindung des Systems in Fail*. Abschnitt 4.1 zeigt zunächst die zur Realisierung der entworfenen Funktionen von FailPanda und der dazu nötigen Modifikationen der Zielplattform. Anschließend thematisiert Abschnitt 4.2 die Anbindung der quelloffenen OCD-Steuerung OpenOCD. Schließlich zeigt Abschnitt 4.3 die Integration der Funktionalität in Fail* durch Implementierung der generischen Backend-Schnittstelle.

4.1 Modifikation der Zielplattform

Der in Kapitel 3 vorgestellte Entwurf des Systems macht es notwendig, dass einige grundsätzliche Modifikationen des Zielsystems durchgeführt werden. Diese Änderungen basieren auf den Entwurfsentscheidungen zur Erkennung von ausgelösten Traps, sowie zur Erkennung von Speicherzugriffen außerhalb des für die Applikation erlaubten Bereichs. Zur Realisierung der Funktionalitäten wird ein vorgefertigter FailPanda-Bootcode bereitgestellt, welcher vor Untersuchung der Zielplattform mit der Zielapplikation zusammengefügt wird. Somit wird eine ausführbare Datei erstellt, die von dem Bootloader des Pandaboards geladen werden kann. Abbildung 4.1 schematisiert das Vorgehen, wobei die hinzugefügten Komponenten im Folgenden erörtert werden.

Für die Erkennung von ausgelösten Traps werden entsprechende Behandlungsroutinen mit dem FailPanda-Code bereitgestellt. Bis auf den Data Abort, welcher eine Behandlung eines MMU-Übersetzungsfehlers durchführt, führen alle weiteren Behandlungsroutinen einen Sprung in eine Funktion durch, welche lediglich das System anhält und keine eigentliche Behandlung durchführt. Durch dieses Verfahren kann mithilfe eines Breakpoints eine Erkennung aller potenziell auftretenden Traps im System erfolgen.

Die Behandlungsroutine für den Data Abort dient der Erkennung von Speicherzugriffen in beobachteten Speicherbereichen und ist nach dem Entwurf in Abschnitt 3.7 implementiert. Hier wird ebenfalls ein Breakpoint an geeigneter Stelle eingefügt. Damit FailPanda die Adressen der entsprechenden Behandlungsroutinen kennt, werden diese als ELF-Symbole aus der generierten Zielapplikation ausgelesen, da die Adressen potenziell durch Veränderungen des Codes abgewandelt werden können.

Für die Herstellung eines definierten Ausgangszustands wird ein spezieller Bootcode bereitgestellt, welcher nach dem Systemstart durch die Bootloader *x-loader* und *u-boot* ausgeführt wird. Diese Bootloader werden üblicherweise auf dem Pandaboard zum Laden eines entsprechenden Betriebssystems verwendet. Da das Pandaboard keinen Speicher für Bootloader bereithält, sind die genannten Bootloader mit der Zielapplikation zusam-

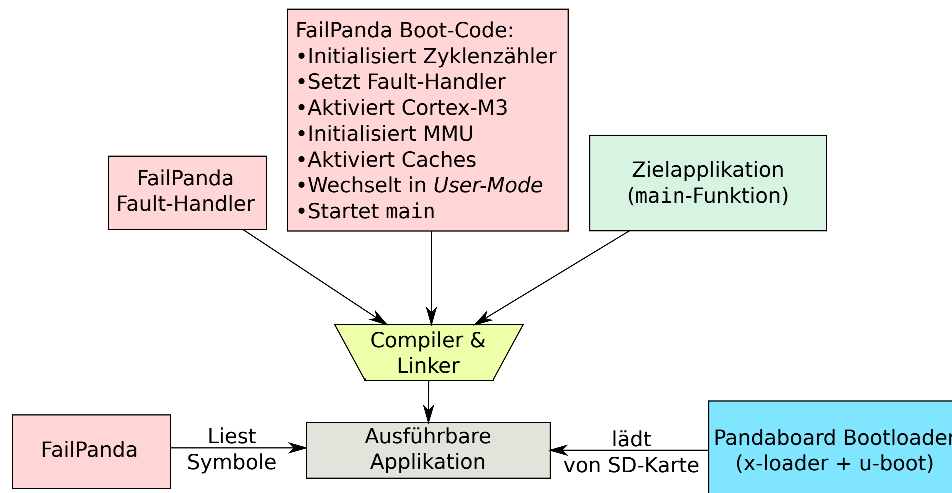


Abbildung 4.1: Realisierung der Einbindung des zu untersuchenden Applikationscodes (grün) in den vordefinierten FailPanda-Code zum Vorkonfigurieren der Hardware und Behandlung von auftretenden Faults (rot) mithilfe des Compilers und Linkers (gelb) in eine ausführbare Datei. Diese wird vom Bootloader des Pandaboards (blau) von der SD-Karte geladen.

men auf einer SD-Karte abgelegt [eli13]. Der zunächst durch einen minimalen auf dem Pandaboard enthaltenen Bootloader geladene x-loader ist lediglich für das Laden von u-boot zuständig. Dieser initialisiert unter anderem die serielle Schnittstelle des Pandaboards und bietet Zugriff auf das Dateisystem der SD-Karte. Der Bootloader wurde für FailPanda so konfiguriert, dass unmittelbar nach dem Ladevorgang die bereitgestellte Zielapplikation in den Speicher geladen und gestartet wird.

Innerhalb dieser Applikation wird zunächst der Initialisierungscode von FailPanda ausgeführt. Die Intialisierungsfunktion ist in C implementiert und wird in einem Linker-Skript als Einsprungspunkt definiert und dadurch initial ausgeführt. Zu Beginn aktiviert der Bootcode den Zyklenzähler des Pandaboards, um insbesondere beim Tracing einen zeitlichen Bezug zur Ausführung der verschiedenen Instruktionen erhalten zu können. Die Fault-Behandlungsroutinen sind in ARM-Assemblercode implementiert und werden durch Setzen des Koprozessor-Registers *Vector Base Address* auf die Adresse der ersten Behandlungsroutine definiert [ARM11a].

Da nach Neustart nur ein Cortex-A9-Prozessorkern Code ausführt, wird an dieser Stelle noch ein Cortex-M3-Prozessor aktiviert, um die Funktionalität der Speicherfehlerinjektion ohne Invalidierung des Cache zu ermöglichen. Da die Cortex-M3-Kerne nicht ohne aktivierte MMU operieren können, wird zunächst die MMU des Cortex-M3 mit einer linearen Übersetzungstabelle initialisiert. Anschließend wird der Kern über eine Initialisierungsroutine in den laufenden Zustand versetzt.

Darauf folgend wird die MMU des Cortex-A9, also des Zielprozessors, initialisiert. Die von den Seitentabellen belegten Speicherbereiche werden in diesem Code als globale Variablen deklariert, sodass entsprechender Speicherplatz reserviert ist. Der Initialisierungscode konfiguriert die Seitentabelle erster Ebene zunächst linear und setzt initi-

al die Adresse der Seitentabelle in dem Koprozessor-Register *Translation Table Base* [ARM11a]. Anschließend werden die Daten- und Instruktions-Caches des Cortex-A9 aktiviert. Schließlich wird ein Wechsel in den nicht-privilegierten *User-Mode* vorgenommen und es wird ein Sprung in die Funktion `main()` der Zielapplikation durchgeführt.

4.2 Anbindung von OpenOCD

OpenOCD bietet Unterstützung für eine Vielzahl an Debuggern – hauptsächlich der JTAG-Schnittstelle – und Zielsystemen [OHEB13]. Die konkrete Konfiguration wird mittels vorgefertigter Skripte aktiviert, welche die Anwendung zur Laufzeit konfigurieren. Diese Konfigurierbarkeit wurde bei der Implementierung der Anbindung bewusst beibehalten, um etwaige weitere Zielsysteme und Debugger einfach anbinden zu können. Zu diesem Zweck wird lediglich die konkrete Konfiguration für den Flyswatter2 und das Pandaboard ES vordefiniert geladen und kann unter geringem Aufwand ausgetauscht werden.

OpenOCD stellt 3 Schnittstellen für verschiedenen Einsatz zur Verfügung. Dies sind

- die Tcl-Schnittstelle, welche eine definierte Skriptsprache in entsprechende OCD-Kommandos umsetzt,
- eine Telnet-Schnittstelle, die das Debuggen in einer Kommandozeile ermöglicht und
- die GDB-Schnittstelle, mit der sich eine Instanz des *GNU Debuggers* (kurz: *GDB*) [GNU13b] verbinden kann, um eine komfortable Debugging-Schnittstelle bereitzustellen.

Diese Schnittstellen wurden jedoch als nicht zweckdienlich identifiziert, da in allen Fällen ein Funktionsaufruf, wie beispielsweise das Anlegen eines Breakpoints, zunächst in eine entsprechende mit der Schnittstelle kompatible Repräsentation umgewandelt werden müsste um anschließend die Reaktion der Schnittstelle wieder zurück umzuwandeln. Da dieser Vorgang als potenzielles Problem bezüglich der Performanz identifiziert wurde, erfolgt die Verwendung von OpenOCD als Bibliothek.

Üblicherweise werden einfache Grundfunktionen innerhalb der Schnittstellensoftware mithilfe von Skripten aktiviert, welche Kommandos von einer der vorgesehenen Schnittstellen (Tcl, Telnet oder GDB) interpretieren. Die gleichen Grundfunktionen können allerdings auch direkt in einem OpenOCD-Kompilat, welches als Bibliothek gebaut wurde, aufgerufen werden. Die Software wird mithilfe von GNU autoconf [GNU13a] aus dem bereitgestellten Quellcode erzeugt. Dabei wird sie auch als in andere Projekte einbindbare Bibliothek erstellt.

Bei der Verwendung dieser Bibliothek sind folgende Einschränkungen zu beachten. Da einige Funktionen nur innerhalb von Kompilationseinheiten verwendet werden, sind diese als `static` deklariert und daher nicht außerhalb der Kompilationseinheit zugreifbar. Ist die Ausführung dieser Funktionen jedoch bei Nutzung der Bibliothek erforderlich, so

muss diese Deklaration geändert werden. Es wurde bei der Implementierung dennoch darauf geachtet, nur minimale Änderungen am OpenOCD-Code vorzunehmen, damit dieser unter geringem Aufwand durch Code neuerer Versionen ersetzt werden kann.

Eine weitere Einschränkung liegt darin, dass eine eigene Initialisierung von OpenOCD programmiert werden muss, da diese nicht in der Bibliothek enthalten ist. Sie kann allerdings leicht in Anlehnung an die im Quellcode vorhandene Initialisierung implementiert werden, welche im Normalfall mithilfe des Linkers zu der Bibliothek gelinkt wird, wodurch das ausführbare OpenOCD-Programm erzeugt wird.

Die im Kern bei der Anbindung von OpenOCD verwendeten Bibliotheksfunktionen und bereitgestellten C-structs werden im Folgenden erläutert:

- `struct target`: Diese Struktur beschreibt einen Zielprozessor, auf den mittels OpenOCD zugegriffen wird. Zu Systemstart wird für jeden zugreifbaren Prozessorkern eine solche Struktur erstellt. Eine Referenz auf die entsprechende Struktur kann mithilfe der Funktion `get_target()` durch Angabe eines eindeutigen textuellen Identifizierers – im Fall des ersten Cortex-A9 Kerns: „omap4460.cpu" – angefordert werden. Bei allen OpenOCD-Funktionen, die auf einen Zielprozessor zugreifen, muss diese Referenz angegeben werden. Auf diese Weise ist es möglich, in einem Programmfluss auf mehrere Prozessorkerne zuzugreifen.

- `target_poll()`: Mit diesem Aufruf wird der aktuelle Systemzustand ausgelesen. Hierbei wird geprüft, ob sich das System zum aktuellen Zeitpunkt im Zustand der Programmausführung oder des Haltens befindet. Soll ein Haltezustand abgewartet werden, so muss demnach solange diese Funktion aufgerufen werden, bis sich der abgerufene Zustand in einen Haltezustand ändert.

- `target_halt()`: Das Zielsystem muss vor Durchführung möglicher Modifikationen wie dem Schreiben in Speicher oder dem Setzen von Haltebedingungen angehalten werden. Dies geschieht im Kontext von FailPanda üblicherweise durch die Aktivierung von Haltebedingungen. Nach einem Neustart befindet sich das System allerdings zunächst in dem Zustand der Programmausführung. Um den Prozessor initial anzuhalten, wird die Funktion `target_halt()` verwendet.

- `target_resume()`: Um die Ausführung des Zielprozessors fortzusetzen, wird diese Funktion aufgerufen.

- `target_step()`: Die OpenOCD-Schnittstelle bietet mit dieser Funktion die Möglichkeit einen Einzelschritt auszuführen.

- `breakpoint_add()`/`breakpoint_remove()`: Unter Angabe der entsprechenden Instruktionsadresse kann ein Breakpoint hinzugefügt bzw. entfernt werden.

- `watchpoint_add()`/`watchpoint_remove()`: Ähnlich wie bei Breakpoints wird mittels dieser Funktionen das Anlegen und Entfernen von Watchpoints ermöglicht. Hierzu müssen Speicheradresse, Breite des zu beobachtenden Bereichs und Art des Zugriffs übergeben werden.

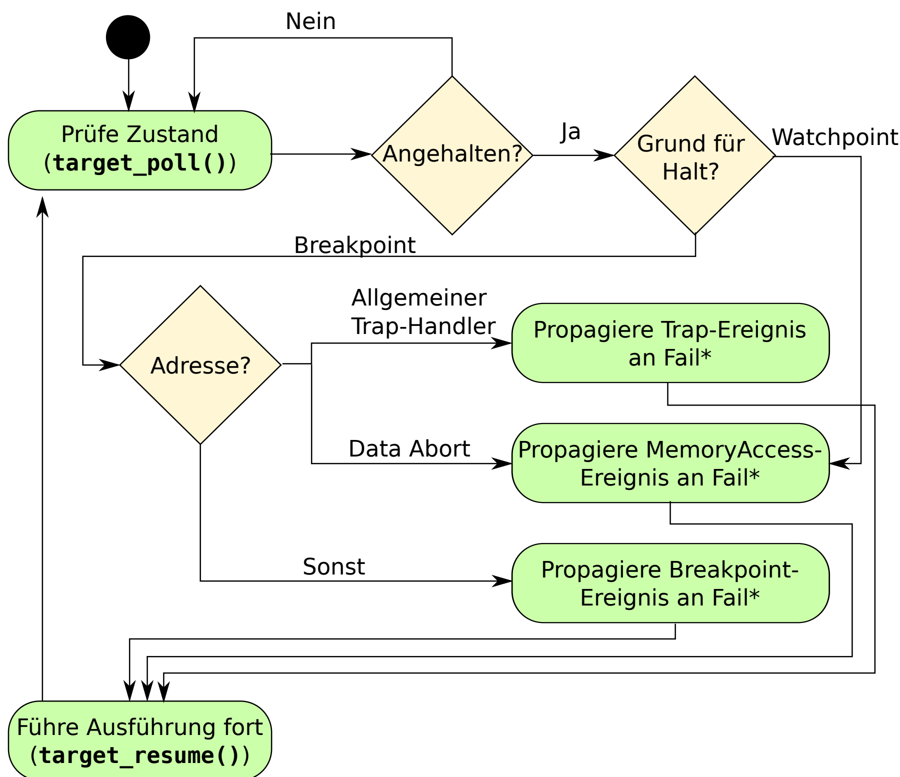


Abbildung 4.2: Schema der Abläufe der Hauptschleife des OpenOCD-Wrappers in FailPanda.

- `target_read_memory()`/`target_write_memory()`: Diese Schnittstelle bietet die Möglichkeit, lesend oder schreibend auf den Speicher des Zielsystems zuzugreifen. Hierbei ist die Länge des Zugriffs und ein entsprechender Puffer zum Auslesen oder Zurückschreiben der Daten anzugeben.

- `struct reg`: Eine derartige Struktur wird zur Initialisierung für jedes Prozessorregister angelegt. Eine Referenz auf diese Strukturen kann aus einem entsprechenden Feld von `struct target` abgerufen werden. Somit sind mithilfe von innerhalb der Struktur definierten `set()`- und `get()`-Funktionen Zugriffe auf das entsprechende Register möglich.

- `Jim_Eval(cmd_ctx->interp, "reset")`: Der Neustart des Zielsystems ist eine Funktionalität, die nicht über eine Bibliotheksfunktion zugreifbar ist. Somit ist dies ein Sonderfall und es wird ein Kommando über den integrierten Interpreter der Skriptsprache *Jim-Tcl* durchgeführt. Der hier dargestellte Aufruf startet das gesamte Zielsystem neu.

Mithilfe der hier beschriebenen Schnittstellenfunktionen sind die wesentliche Teile von FailPanda realisiert worden.

Aufgrund der grundsätzlichen Konzeption von Fail* ist es notwendig, dass das Zielsystem einen Ausführungszyklus aufweist, welcher als Koroutine die Ausführung unter-

brechen kann, um im Fall eines Ereignisses den Kontrollfluss an die Fail*-Komponenten übergeben zu können. Die OpenOCD-Bibliothek bietet diese Funktionalität nicht, weshalb ein sogenannter *Wrapper* implementiert wurde. Die Aufgabe des Wrappers besteht einerseits darin, einfache Aufrufe, wie das Setzen eines Breakpoints in entsprechende Aufrufe an die Bibliothek zu übersetzen, andererseits wird hier ebenfalls eine Hauptschleife implementiert, welche die Aufgabe hat, auf Ereignisse des OCD zu reagieren und in entsprechende Fail*-Ereignisse umzuwandeln.

Abbildung 4.2 zeigt ein Schema der Abläufe innerhalb dieser Schleife. Zu dem Zweck der Erkennung von OCD-Ereignissen wird in der Schleife periodisch der Prozessor-Status mittels `target_poll()` ausgelesen. Befindet sich der Prozessor im Haltezustand, so wird die Ursache untersucht. Bei einem aktivierten Breakpoint oder Watchpoint wird ein entsprechendes Ereignis an Fail* übergeben, wobei ein Koroutinenwechsel in das Experiment durchgeführt wird. Da OpenOCD allerdings keine Information darüber liefert, welcher Break- oder Watchpoint den Haltezustand eingeleitet hat, muss dies an dieser Stelle herausgefunden werden. Zu diesem Zweck wird für Breakpoints lediglich das *Program Counter* Register des Prozessors ausgelesen. Bei Watchpoints kann, solange nur ein Watchpoint gesetzt wurde, das einzige Element der Watchpoint-Liste des `struct target` ausgelesen und entsprechende Information an Fail* weitergegeben werden. In Fällen, in denen mehrere Watchpoints gesetzt sind, muss allerdings eine Instruktionsdekodierung (s. Abschnitt 3.3) erfolgen, um Informationen über den aktuellen Speicherzugriff zu erhalten.

Bei einem aktivierten Breakpoint wird vor der Übergabe an Fail* zunächst geprüft, ob es sich um einen Breakpoint in der generischen Trap-Behandlungsroutine oder in der Abort-Behandlungsroutine für Seitenfehler handelt. In diesen Fällen wird eine entsprechende Sonderbehandlung durchgeführt und ein Trap-Ereignis oder ein Speicherzugriffs-Ereignis an Fail* kommuniziert.

Nach der Behandlung eines Halte-Ereignisses wird das Experiment die Ausführung fortsetzen, wodurch ein Koroutinenwechsel in die Schleife des Wrappers an die Stelle zurückkehrt, von der aus der letzte Koroutinenwechsel durchgeführt wurde. Da nun die Ausführung des Backends fortgesetzt werden soll, wird nach der Behandlung von Halte-Ereignissen die Funktion `target_resume()` aufgerufen und anschließend wird zum Beginn der Schleife zurückgekehrt.

Die grundsätzliche Vorgehensweise bei der Konfiguration der MMU zur Erkennung von Speicherzugriffen wurde in Abschnitt 3.7 beschrieben. Bei der konkreten Umsetzung wird wie folgt vorgegangen. Es wird intern eine Repräsentation der aktuellen MMU-Konfiguration gehalten, damit diese nicht kostenintensiv ausgelesen werden muss. Diese Repräsentation besteht aus einer vollständigen Beschreibung aller Deskriptoren erster Ebene, die jeweils einen Typ besitzen, welcher definiert, ob es sich um einen Section- oder einen Page-Deskriptor handelt. Im Fall eines Section-Deskriptors ist zusätzlich gespeichert, ob die Section beobachtet wird und im Fall einer Page enthält der Deskriptor zusätzlich eine Referenz auf ein 256-Einträge beinhaltendes Feld von Wahrheitswerten, die aussagen, ob eine entsprechende Seite beobachtet wird. Diese Struktur wird nach jedem Neustart zurückgesetzt, so dass alle Deskriptoren unbeobachtete Section-Deskriptoren

sind, was der Konfiguration der MMU beim Neustart entspricht. Bei jeder Änderung der MMU-Konfiguration wird diese Repräsentation entsprechend modifiziert.

Mithilfe der aktualisierten Repräsentation des Zustands der MMU-Konfiguration kann bei Hinzufügen oder Entfernen beobachteter Bereiche gezielt identifiziert werden, welche Deskriptoren zu verändern sind, da sie bislang noch nicht beobachtet werden. Es wird demnach nach Unterschieden der aktuellen und der neu gewünschten Konfiguration gesucht, indem ein Abgleich mit der lokalen Repräsentation der MMU-Konfiguration durchgeführt wird. Soll eine Seite oder Section durch die MMU beobachtet werden, so wird der entsprechende Deskriptor in einen Fault-Deskriptor umgewandelt. Da der Deskriptor potenziell im TLB vorhanden sein könnte, wird hier anschließend eine Invalidierung vorgenommen, indem entsprechende Koprozessor-Register geschrieben werden.

4.3 Integration in Fail*

In Abschnitt 2.6 wurden bereits die grundlegenden Schnittstellen von Fail* erläutert. An dieser Stelle wird nun konkret auf die Implementierung der Konfigurierbarkeit mithilfe des eingesetzten Build-Systems und auf die damit implementierte Integration der Fehlerinjektion für das Pandaboard eingegangen. Im Projekt Fail* wird das Werkzeug *CMake* [Kit13] zur Durchführung der Kompilierung verwendet. Hierbei handelt es sich um ein quelloffenes Build-System, welches die Erstellung eines Projekts auf verschiedenen Plattformen unabhängig vom zu verwendenden Compiler erlaubt. So ist es beispielsweise möglich, ein mit CMake konfiguriertes Projekt auf einem Linux-System und ohne Modifikationen auf einem Windows-System zu erstellen. Zu diesem Zweck generiert CMake die zur Erstellung nötige Infrastruktur – im Beispiel eines Linux-Systems wird eine Struktur von Makefiles erzeugt.

Da Fail* typischerweise in Linux ausgeführt wird, ist die Funktionalität der systemübergreifenden Erzeugung allerdings nicht primär relevant. CMake ermöglicht ebenfalls die Definition von Optionen, die vor der Generierung vom Benutzer gewählt werden und entsprechend die Erstellung beeinflussen. Im Kontext von Fail* wird der Mechanismus zur statischen Konfiguration des Frameworks verwendet. Somit können verschiedene Varianten von Fail* generiert werden, die koexistieren können, da die generierten Komponenten vom Quellcode getrennt werden.

Es ist möglich, mittels dieser Konfigurierbarkeit das zu verwendende Backend, die verwendeten Ereignistypen und die Experimente auf diese zu wählen. Auf Grundlage der Auswahl werden Unterprojekte aktiviert, die entsprechend einer CMake-Definition Quelldateien in die Erstellung mit einbinden. Innerhalb von Quellcode-Dateien ist die Maskierung von Teilen des Codes beispielsweise unter Zuhilfenahme von Präprozessor-Direktiven steuerbar, die durch die Konfiguration gesetzt werden. Mittels dieser Werkzeuge ist eine komplexe Konfigurierbarkeit möglich.

Die grundsätzliche Varianten-Auswahl geschieht in Fail* durch Auswahl einer CMake-Konfigurationsdatei, welche für jede Backend-Variante angelegt wird. Hier werden spezifische Quellcode-Dateien definiert, die zu einer Bibliothek gebaut und anschließend zur Kernapplikation gebunden werden. Zur Integration der Unterstützung des Panda-

boards ist es notwendig, OpenOCD zu erstellen. Die OCD-Schnittstellen-Software nutzt allerdings zur Erstellung das Werkzeug *GNU autoconf* (s. Abschnitt 4.1). In CMake ist die Erstellung eines mithilfe eines externen Build-Systems zu erstellenden Projekts möglich, weshalb sich kein Konflikt ergibt. Im Generierungsprozess von CMake wird die Erstellung mittels autoconf aktiviert und die resultierende Bibliothek anschließend zum restlichen System gelinkt.

Die Anbindung von Backends geschieht in Fail* mithilfe der in Abschnitt 2.6 erläuterten generischen Schnittstelle. Die Implementierung der Schnittstelle erfolgt bei Simulator-Backends in der Hauptsache durch Anwendung von aspektorientierter Programmierung mittels AspectC++, da hierbei an entsprechend relevanten Stellen der Simulation durch Verwendung von Aspekten zusätzlicher Programmcode eingewoben werden kann, welcher die relevanten Informationen an die Fail*-Schnittstelle propagiert [SLU05]. Durch dieses Vorgehen kann der Simulatorcode weitestgehend unmodifiziert verwendet werden, was beispielsweise eine Aktualisierung auf neue Versionen erleichtert. Der Simulator Gem5 wurde allerdings ohne Verwendung von AspectC++ durch direkte Modifikation im Quellcode angebunden.

Im Fall von OpenOCD stehen genügende Schnittstellen zur Verfügung. Somit wurde, wie bereits im vorherigen Abschnitt dargestellt wurde, auf die Anwendung von AspectC++ verzichtet, um stattdessen einen Wrapper für die Funktionalität zu implementieren. Dieser Wrapper bietet bereits eine gute Abbildung auf entsprechende Funktionen der zu implementierenden Backend-Schnittstelle. Somit sind lediglich Aufrufe der Wrapper-Funktionen und entsprechende Signalisierungen von Ereignissen im Wrapper zur Einbindung in Fail* nötig.

Wie bereits im vorherigen Kapitel erläutert wurde, sind nicht alle in Abschnitt 2.6.1 definierten Backend-Funktionen, die Fail* unterstützt, im Kontext dieser Arbeit implementiert worden. Insbesondere wurden die folgenden Funktionen nicht realisiert:

- Zugriff auf Ausgaben der seriellen Schnittstelle,
- Checkpointing (im Kontext von Fail* *Save/Restore* genannt),
- Erkennung von aufgetretenen Interrupts,
- Gastsystem-Ereignisse,
- IOPort-Ereignisse und
- Sprung-Ereignisse.

Der Ausschluss der Funktionen aus der Implementierung wurde beschlossen, da sie für die Ausführung von einfachen Fehlerinjektionskampagnen zunächst nicht zwingend notwendig sind.

In der bisherigen Implementierung von Fail* wurde die Trace-Navigation in den Backends stets mithilfe von Single-Steps implementiert. Aus diesem Grund war die Verwendung der durch Smart-Hopping berechneten Sprungketten nicht ohne Modifikationen des Frameworks möglich. Da die Architektur von Fail* vorsieht, dass ein Server Parameter

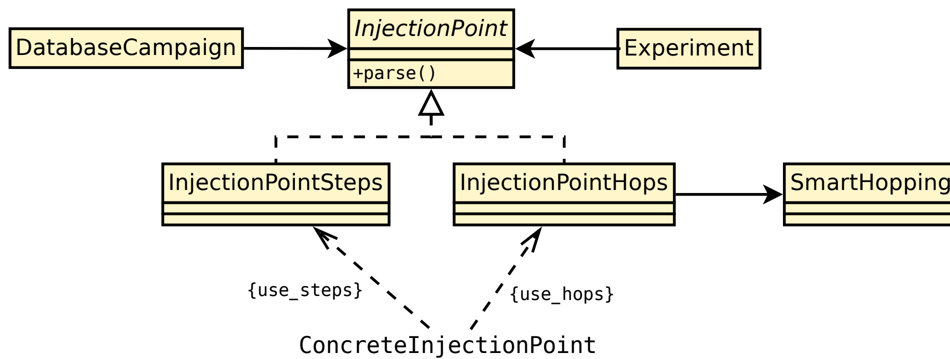


Abbildung 4.3: Klassendiagramm zur Beschreibung der Konfigurierbarkeit eines abstrakten `InjectionPoint`.

an mehrere Clients übergibt, ist es sinnvoll, die Berechnung von Sprungketten direkt auf der Serverkomponente durchzuführen und berechnete Sprungketten mit den weiteren Experiment-Parametern an die Clients zu übergeben.

Um eine generische Lösung zu verfolgen, wurde hier eine Abstraktion des Injektionszeitpunktes namens `InjectionPoint` in Fail* integriert. Abbildung 4.3 zeigt eine Darstellung der Zusammenhänge bezüglich der Verwendung des abstrakten `InjectionPoint`. Serverseitig wird von der `DatabaseCampaign` ein `InjectionPoint` angelegt und es werden die zwei Trace-Positionen, die Anfang und Ende einer Äquivalenzklasse definieren, als Initialisierung an diesen abstrakten Injektionszeitpunkt übergeben. Intern wird die Umwandlung in die Information durchgeführt, welche von der aktuell konfigurierten Navigationsart benötigt wird. Dies wird durch die Implementierung verschiedener konkreter `InjectionPoints` realisiert, deren Nutzung über eine CMake-Option konfiguriert wird. So berechnet ein `InjectionPointHops` beispielsweise eine Sprungkette und speichert diese. Ein `InjectionPointSteps` sichert intern lediglich die Trace-Position des rechten Rands der Äquivalenzklasse.

Der `InjectionPoint` bietet zusätzlich eine Schnittstelle, um die Navigationsinformationen in die Protobuf-Nachricht einzufügen, die die Experiment-Parameter enthält. Zu diesem Zweck wird die Nachrichtendefinition der Protobuf-Nachricht für Kampagnenparameter ebenfalls mithilfe von CMake konfiguriert. Der Client nutzt eine `InjectionPoint`-Schnittstelle, die es ermöglicht, den konkreten Injektionszeitpunkt aus einer Protobuf-Nachricht zu extrahieren.

Im Experimentcode kann eine Navigationsschnittstelle nach Übergabe eines `InjectionPoint`-Objekts die konkrete Implementierung der Trace-Navigation durchführen. Auch dieser Code ist konfigurierbar. Durch diese Implementierung ist sowohl der Kampagnen-Programmcode als auch der Programmcode von Experimenten für verschiedene Arten der Trace-Navigation wiederverwendbar.

Zur Integration von Smart-Hopping in die generische `DatabaseCampaign` war es notwendig, die Reihenfolge der Berechnung von Experiment-Parametern abzuwandeln. Bis zur Integration von Smart-Hopping gab es keine Vorteile durch die Verwendung einer bestimmten Reihenfolge der Abarbeitung von Experimenten. Aufgrund der inkrementel-

len Implementierung von Smart-Hopping ist es jedoch notwendig, dass zu berechnende Sprungketten in aufsteigender Reihenfolge der Trace-Position verarbeitet werden. Aus diesem Grund werden die Experimentparameter bei Verwendung des Verfahrens aufsteigend nach der Trace-Position des linken Rands der Äquivalenzklassen sortiert. Das Verfahren berechnet somit immer soweit neue Sprungketten und deren Kosten, bis für alle in der aktuell bearbeiteten Äquivalenzklasse befindlichen Trace-Positionen ein Ergebnis verfügbar ist. Es wird dann die Sprungkette mit den geringsten Kosten gewählt und weiter verfahren.

Aufgrund der Sortierung können alle Lösungen für Positionen vor dem linken Rand der aktuell bearbeiteten Äquivalenzklasse verworfen werden, wodurch auch der Speicherverbrauch gering gehalten werden kann.

4.4 Zusammenfassung

In diesem Kapitel wurde das konkrete Vorgehen bei der Anbindung des Pandaboard ES an das Fehlerinjektions-Framework Fail* beschrieben. Hier wurden zunächst grundsätzlich drei verschiedene Ebenen der Implementierung identifiziert. Es wurde gezeigt, dass zusätzlicher Programmcode auf dem Zielsystem eingebracht werden musste, um Funktionalitäten wie die MMU-Konfiguration zur Beobachtung von Speicherzugriffen in bestimmten Bereichen realisieren zu können. Zur Verwendung der von dem Pandaboard gebotenen OCD-Funktionen, die in Kapitel 3 in dem Entwurf des Gesamtsystems berücksichtigt wurden, wurde hier die Entwicklung eines Wrappers für die OCD-Schnittstellen-Software OpenOCD skizziert. Dieser Wrapper bietet eine Schnittstelle, durch die die Integration in das Fehlerinjektions-Framework Fail* leicht möglich war. Abschließend wurde die Integration des OpenOCD-Wrappers in Fail* durchgeführt. Hierbei konnte gezeigt werden, dass die Anbindung des neuen Backends nur geringe Modifikationen des Frameworks bedingte. Des Weiteren wurde deutlich, wie durch geeignete Konfiguration des Build-Systems CMake eine Einbindung ermöglicht wurde. Im folgenden Kapitel wird das implementierte System bezüglich seiner Performanz und der generierbaren Ergebnisse evaluiert.

5 Evaluation

Dieses Kapitel stellt eine Evaluation von FailPanda unter den Untersuchungskriterien der Performanz und der Ergebnisqualität dar. Da die Untersuchungen dieser Arbeit sich im Schwerpunkt auf die Realisierbarkeit von OCD-Fehlerinjektion mittels COTS-Hardware beziehen, wird schwerpunktmäßig die Einsetzbarkeit des Verfahrens analysiert. Hierbei können insbesondere Defizite der Performanz und der Ergebnisgenerierung problematisch sein, weshalb eine detaillierte Betrachtung dieser Aspekte durchgeführt wird.

Die Evaluation bezüglich Performanz erfolgt im folgenden Abschnitt 5.1. Anschließend werden in Abschnitt 5.2 Ergebnisse einer mit FailPanda durchgeführten Kampagne vorgestellt und diskutiert.

5.1 Performanz

Wie bereits die Anforderungen in Kapitel 3 deutlich machten, ist die Performanz der Einzelkomponenten von FailPanda entscheidend dafür, ob eine Reihe an Experimenten in einer praktisch vertretbaren Zeit durchgeführt werden kann. Ein Kernproblem von Fehlerinjektion liegt in der üblicherweise sehr hohen Anzahl durchzuführender Experimente, weshalb die Laufzeitoptimierung ein primäres Ziel der Entwicklung dieser Arbeit darstellt.

Die in diesem Kapitel dargestellten Messungen sind in der Regel Mittelwerte über Messreihen von 100 Werten. Somit können Lastschwankungen des Testsystems ausgeglichen werden und es kann eine Standardabweichung σ angegeben werden, um die Streuung der Werte erkennbar zu machen. Messungen zur Performanz wurden, wenn nicht anders dargestellt, auf einem Linux-PC mit Intel Core i7-2630QM (4×2 GHz, durch Intel Turbo Boost bis zu 2,9 GHz) und 8 GB Hauptspeicher durchgeführt. Als Betriebssystem wird Debian GNU/Linux 7.1 (wheezy) mit Linux-Kernel SMP Debian 3.2.46-1 x86_64 GNU/Linux verwendet. Für die Erstellung der Komponenten des Hostrechners wird der gcc in der Version 4.6.3 verwendet und für die Erstellung der Zielapplikationen wird der gcc als Cross-Compiler (Sourcery G++ Lite 2010.09-51) in der Version 4.5.1 eingesetzt. Zur Laufzeitmessung wird die Linux-Funktion `gettimeofday` aus dem System-Header *sys/time.h* verwendet [lin13]. Die feinste darstellbare Auflösung liegt hierbei im Bereich einer Mikrosekunde, die tatsächliche Auflösung kann allerdings abweichen. Zur Messung des Speicherverbrauchs von Prozessen wird die Linux-Funktion *readproc* verwendet [man13]. Mittels dieser Funktion aus dem System-Header *proc/readproc.h* ist es möglich, Betriebssystem-Prozessinformationen, insbesondere den aktuellen Speicherverbrauch, auszulesen. Aufgrund der Betriebssystem-Sicht kann hier jedoch nicht genauer als auf Seiten-Granularität (Seitengröße: 4 kB) gemessen werden.

Tabelle 5.1: Verarbeitungszeit der von FailPanda implementierten Einzelfunktionen.

Operation	**Mittel**	σ
Neustart	1032 ms	3,43 ms
Register lesen	5,980 ms	0,0122 ms
Register schreiben	25,98 ms	0,0297 ms
Speicher lesen (4 B)	11,90 ms	0,161 ms
Speicher schreiben (4 B)	11,93 ms	0,0984 ms
Single-Step	69,99 ms	2,17 ms

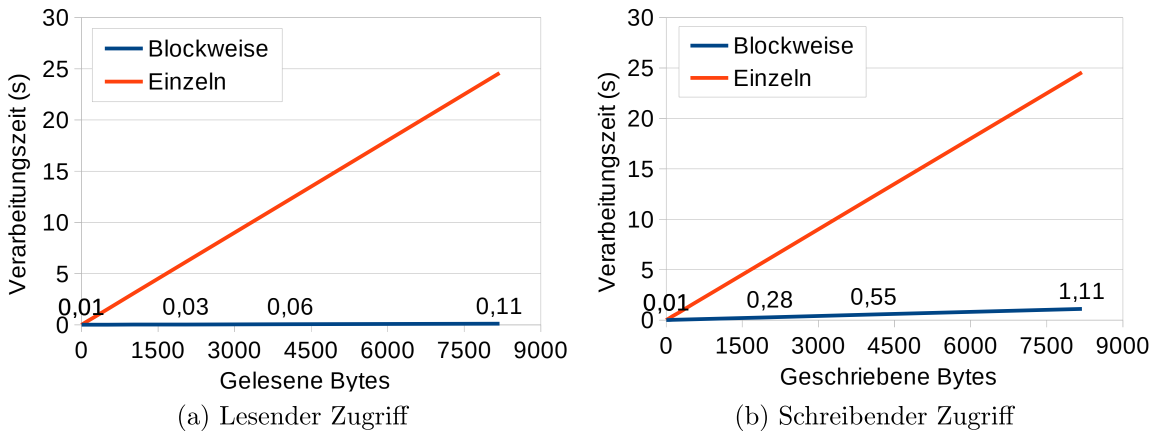
Im Folgenden werden in Abschnitt 5.1.1 zunächst grundlegende Einzelfunktionen der OCD-Schnittstelle gemessen. Hierbei handelt es sich um die elementaren Bausteine zum Aufbau der komplexeren Funktionen. Anschließend zeigt die Untersuchung in Abschnitt 5.1.2 die Performanz verschiedener Verfahren zur Trace-Navigation, die als Kernproblem der Arbeit intensiv untersucht wurde. Abschnitt 5.1.3 stellt die Untersuchung zur Verwendung der MMU zur Erkennung von Speicherzugriffen dar. Abschließend wird in Abschnitt 5.1.4 eine Evaluation der Performanz des Gesamtsystems durchgeführt.

5.1.1 Schnittstelle

Zu Beginn der Evaluation der Systemperformanz werden zunächst grundlegende Funktionen der OCD-Schnittstelle evaluiert. Diese Schnittstelle ist ein Zusammenschluss von OpenOCD, Flyswatter2 und dem OCD des Pandaboards. Tabelle 5.1 zeigt die mittleren Werte und die Standardabweichung der Ausführung einiger grundlegender Funktionen. Bei dem Wert für den Neustart des Systems ist allerdings zu beachten, dass dieser vom Zeitpunkt des Auslösens eines Neustarts bis zum Halten vor Eintritt in die `main`-Funktion gemessen wurde und demnach auch die Ausführungszeit des FailPanda-Bootcodes berücksichtigt. Alle restlichen betrachteten Funktionen werden direkt auf Schnittstellenfunktionen von OpenOCD abgebildet.

Die Tabelle zeigt, dass die Systeminitialisierung (Neustart), welche in jedem Experiment durchgeführt werden muss und daher die Untergrenze der Experimentkosten darstellt, eine Ausführungszeit von mehr als 1 s aufweist. Es wird demnach bei Betrachtung dieses Werts bereits deutlich, dass eine Kampagne mit einer Anzahl von beispielsweise 10^6 Experimenten nicht praktikabel durchgeführt werden kann, da bereits nur für die Initialisierung der Experimente eine Zeit von etwa 286 h benötigt würde.

Da beispielsweise beim Tracing häufige Zugriffe auf Registerwerte des Prozessors durchgeführt werden, um in einer Instruktionsdekodierung Speicherzugriffe zu erkennen, werden die Zugriffszeiten hier ebenfalls betrachtet. Es zeigt sich, dass das Lesen eines Registers nur etwa ein Fünftel der Zeit für das Schreiben benötigt. Es wird somit klar, dass die Operation des seltenen Ereignisses des Schreibens eines Registers – dies Tritt nur bei der Fehlerinjektion in Registern auf – zwar zeitintensiv ist, sich jedoch aufgrund seiner geringen Häufigkeit kaum auf die Gesamtzeit auswirken kann.



(a) Lesender Zugriff

(b) Schreibender Zugriff

Abbildung 5.1: Verarbeitungszeit von Speicherzugriffen für verschiedene Anzahlen von gelesenen und geschriebenen Bytes. Zur Differenzierung einzelner und blockweiser Zugriffe werden beide Arten getrennt dargestellt, wobei Einzelzugriffe in Gruppen der Größe von 4 B durchgeführt wurden. Zur besseren Lesbarkeit sind die deutlich geringeren Zugriffszeiten blockweiser Zugriffe zusätzlich abschnittsweise in das Diagramm eingetragen.

Speicherzugriffe werden im Kontext von FailPanda für Fehlerinjektion, die Konfiguration der MMU und für die Überprüfung des berechneten Ergebnisses genutzt. In letzteren beiden Fällen ist es auch möglich, dass auf große Speicherbereiche zugegriffen wird. Tabelle 5.1 zeigt zunächst, dass der Zugriff auf einzelne Speicherwörter der Länge 4 B mit unter 12 ms sowohl schreibend als auch lesend nur einen geringen zusätzlichen Aufwand verursacht.

Da jedoch auch größere Speicherbereiche zugegriffen werden, zeigt Abbildung 5.1 eine Betrachtung von Speicherzugriffen auf Bereichen verschiedener Größen. Hierbei geschieht eine getrennte Betrachtung von Zugriffen in einem Block oder mehrerer Einzelzugriffe der Breite 4 B. Abbildung 5.1a stellt hierbei die Verarbeitungszeit für lesende und Abbildung 5.1b für schreibende Zugriffe dar. Es wird zunächst deutlich, dass der blockweise Zugriff in beiden Fällen deutlich schneller ist als die Zusammenfassung zu mehreren Einzelzugriffen. Des Weiteren zeigen die Diagramme, dass blockweise lesende Zugriffe etwa um einen Faktor 10 schneller als schreibende Zugriffe ablaufen. Schließlich ist ebenfalls zu erkennen, dass beispielsweise für das Lesen eines Blocks von 8 kB eine Zeit von 110 ms benötigt wird. Diese Zeit ist nur etwa das Zehnfache der in Tabelle 5.1 angegebenen Zeit für das Lesen eines Blocks der Größe 4 B. Demnach ist der eigentliche Lesevorgang nicht der dominierende Faktor bei Ausführung eines Lesevorgangs über die OCD-Schnittstelle. Es wird also klar, dass Speicherzugriffe nach Möglichkeit immer zusammenhängend durchgeführt werden sollten, um eine Minimierung des Verarbeitungsaufwands zu erreichen.

Die Zeit für die Durchführung eines Single-Steps beträgt im Mittel etwa 69,99 ms und definiert damit insbesondere die für das Tracing benötigte Verarbeitungszeit. Eine Be-

Tabelle 5.2: Mittlere Kosten und Standardabweichungen bei der Ausführung von horizontalen und diagonalen Sprüngen, gemessen am Gesamtsystem. Der jeweilige bemessene Übergang ist abgekürzt mit BP für Breakpoints und WP für Watchpoints als Haltebedingungen an Anfang und Ende des Sprungs.

	Horizontal		**Diagonal**			
	BP→BP	**WP→WP**	**BP→BP**	**BP→WP**	**WP→BP**	**WP→WP**
Mittel	135,6 ms	135,6 ms	67,7 ms	67,7 ms	67,7 ms	65,7 ms
σ	0,1 ms	0,1 ms	0,08 ms	0,09 ms	0,09 ms	0,08 ms

trachtung des Tracing erfolgt noch im Gesamtkontext der Durchführung einer Kampagne (s. Abschnitt 5.1.4).

Single-Steps werden zusätzlich bei der Trace-Navigation eingesetzt, wenn horizontale Sprünge durchgeführt werden. Wie bereits in Abschnitt 3.5 gezeigt wurde, sind daher die Kosten eines horizontalen Sprungs deutlich höher als die Kosten eines diagonalen Sprungs. Tabelle 5.2 zeigt Messergebnisse der verschiedenen Arten von möglichen Sprüngen. Hierbei wird zunächst deutlich, dass der Typ der Sprungziele (Break- oder Watchpoint) größtenteils ohne Auswirkung auf die Kosten ist. Lediglich diagonale Sprünge von einem Watchpoint auf einen anderen Watchpoint benötigen im Mittel etwa 2 ms weniger Zeit als andere diagonale Sprünge. Es wird vermutet, dass der Grund für diesen Umstand in der Implementierung des Anlegens und Entfernens von Break- und Watchpoints liegt. Auf eine weitere Untersuchung wird hier allerdings verzichtet, da der Grund für die leicht unterschiedliche Laufzeit für die Erstellung eines Kostenmodells nicht relevant ist. Das Verhältnis der Kosten von horizontalen zu diagonalen Sprüngen entspricht etwa dem Faktor 2. Auf Basis dieser Relation wurden spezielle Verfahren zur Trace-Navigation entwickelt, die im Folgenden evaluiert werden.

5.1.2 Trace-Navigation

In Abschnitt 3.5 wurden verschiedene Verfahren zur Trace-Navigation auf eingebetteter Hardware diskutiert, welche im Folgenden detailliert im Hinblick auf die durch die Navigation verursachte Verzögerung und auf ihren Berechnungsaufwand untersucht werden. Zur Evaluation werden hierbei Benchmarks aus der *MiBench*-Sammlung verwendet [GRE+01]. Diese Benchmarks sind speziell für die Untersuchung eingebetteter Systeme ausgelegt und berechnen demnach typische Probleme aus diesem Bereich. Jeder Benchmark ist als *small* und *large* Variante verfügbar, wobei die *large* Variante die gleichen Berechnungen auf größeren Datensätzen durchführt und demnach länger läuft, als die *small* Variante. Die konkret eingesetzten Benchmarks werden im Folgenden kurz vorgestellt. Aus dem Bereich *automotive* wurden folgende Benchmarks verwendet:

- **BASICMATH:** Dieser Benchmark berechnet grundlegende mathematische Probleme. Konkret werden kubische Gleichungen gelöst, ganzzahlige Quadratwurzeln gebildet und Winkelwertkonvertierungen zwischen Grad und Radiant durchgeführt.

- **BITCOUNT:** BITCOUNT führt das Zählen von gesetzten Bits in einer Variable aus. Hierzu werden sieben verschiedene Implementierungen des Problems nacheinander durchlaufen.

- **QSORT:** Es wird der qsort-Algorithmus der *C-Stdlib* [Cod13] auf einer Menge von Wörtern, die als Zeichenkette repräsentiert sind, aufgerufen.

- **SUSAN:** SUSAN führt typische Algorithmen aus dem Bereich der Bildverarbeitung aus – beispielsweise eine Kanten- und Eckerkennung. Diese Verarbeitungen werden auf der Basis von Grauwertbildern durchgeführt.

Aus dem *network*-Bereich werden weiterhin folgende Benchmarks verwendet:

- **DIJKSTRA:** Auf der Grundlage einer Adjazenzmatrix werden kürzeste Pfade zwischen zwei Knoten mithilfe des Kürzeste-Wege-Algorithmus von Dijkstra berechnet.

- **PATRICIA:** Dieser Benchmark baut einen sogenannten *Patricia Trie* aufgrund einer gegebenen Menge von IPv4-Adressen auf. Mithilfe dieser Struktur ist eine schnelle Berechnung einer nötigen Ablehnung einer Verbindung möglich (sogenanntes *Address Masking*).

Diese Wahl dieser Benchmarks aus der Gesamtmenge der MiBench-Sammlung weist Applikationen mit unterschiedlichen Arten von Programmabläufen auf, sodass etwaige Schwachstellen der verschiedenen Navigationsverfahren erkannt werden können. Die Benchmarks wurden für diese Untersuchung für die ARMv5 kompiliert. Mithilfe der Ausführung in Gem5 im *System Call* Modus wurden die zur Untersuchung nötigen Traces erzeugt. Durch dieses Verfahren ist es möglich, die nötigen Trace-Informationen schnell zu generieren, wobei die Ergebnisse dennoch nah an denen der Ausführung auf der Zielplattform bleiben.

Um die in Abschnitt 3.5 vorgestellten Verfahren zur Berechnung von Trace-Navigationspfaden zu bewerten, wird ein einfaches Kostenmodell zugrunde gelegt. Es wurde bereits erläutert, dass ein sogenannter horizontaler Sprung doppelte Kosten im Vergleich zu einem vertikalen Sprung verursacht. Dementsprechend werden Kosten im Folgenden in einer abstrakten Einheit beschrieben, welche einer Verarbeitungszeit von etwa 67 ms entspricht, wodurch die verschiedenen Verfahren vergleichbar sind und die Werte übersichtlich bleiben. Es fallen bei der Navigation allerdings noch zusätzliche Kosten für die Ausführung der zwischen den Haltebedingungen liegenden Instruktionen an, die hier nicht betrachtet werden, da sie als gering angenommen werden und unabhängig vom Verfahren bei jeder Trace-Navigation anfallen.

Tabelle 5.3 beschreibt zunächst die grundlegenden Eigenschaften der für die Evaluierung generierten Traces. Hierbei wird die Anzahl dynamischer Instruktionen – also die Länge des Traces – und die Anzahl an Speicherzugriffen für jeden Trace angegeben. Somit können Rückschlüsse über den Einfluss des Speicherzugriffsverhaltens auf die Anwendbarkeit der verschiedenen Verfahren gezogen werden. Bei Betrachtung der Werte wird deutlich, dass sich die Längen der Traces verschiedener Benchmarks teilweise deutlich

Tabelle 5.3: Eigenschaften der MiBench-Benchmarks der Kategorien *automotive* und *network* mit einer zusätzlichen Angabe der maximalen und mittleren Kosten sowie der Standardabweichung der Kosten unter Verwendung von Simple-Hopping.

					Simple-Hopping		
	Benchmark	**Eingabe**	**Dyn. Instr.**	**Speicherzugriffe**	**Max**	**Mittel**	σ
automotive	BASICMATH	large	$3{,}37 \times 10^9$	$1{,}05 \times 10^9$	$1{,}76 \times 10^7$	$3{,}62 \times 10^6$	$4{,}00 \times 10^6$
		small	$2{,}66 \times 10^8$	$5{,}00 \times 10^7$	$2{,}92 \times 10^6$	$7{,}21 \times 10^5$	$7{,}55 \times 10^5$
	BITCOUNT	large	$6{,}26 \times 10^8$	$4{,}28 \times 10^7$	$6{,}75 \times 10^7$	$2{,}30 \times 10^7$	$2{,}08 \times 10^7$
		small	$4{,}18 \times 10^7$	$2{,}86 \times 10^6$	$4{,}50 \times 10^6$	$1{,}53 \times 10^6$	$1{,}38 \times 10^6$
	QSORT	large	$4{,}59 \times 10^8$	$1{,}50 \times 10^8$	$3{,}14 \times 10^6$	$6{,}78 \times 10^5$	$7{,}06 \times 10^5$
		small	$1{,}78 \times 10^7$	$7{,}12 \times 10^6$	$4{,}63 \times 10^5$	$6{,}73 \times 10^4$	$8{,}85 \times 10^4$
	SUSAN	large	$3{,}92 \times 10^8$	$1{,}52 \times 10^8$	$9{,}95 \times 10^6$	$2{,}54 \times 10^6$	$2{,}93 \times 10^6$
		small	$2{,}42 \times 10^7$	$8{,}94 \times 10^6$	$6{,}50 \times 10^5$	$1{,}74 \times 10^5$	$1{,}93 \times 10^5$
network	DIJKSTRA	large	$2{,}03 \times 10^8$	$6{,}94 \times 10^7$	$1{,}85 \times 10^7$	$6{,}76 \times 10^6$	$5{,}11 \times 10^6$
		small	$4{,}59 \times 10^7$	$1{,}60 \times 10^7$	$3{,}67 \times 10^6$	$1{,}18 \times 10^6$	$1{,}04 \times 10^6$
	PATRICIA	large	$5{,}81 \times 10^8$	$2{,}25 \times 10^8$	$5{,}10 \times 10^6$	$4{,}47 \times 10^5$	$6{,}13 \times 10^5$
		small	$9{,}45 \times 10^7$	$3{,}65 \times 10^7$	$7{,}84 \times 10^5$	$6{,}84 \times 10^4$	$9{,}30 \times 10^4$

unterscheiden. Um dennoch Ergebnisse gut vergleichen zu können, wird im Folgenden schwerpunktmäßig mit Durchschnittswerten über alle Ergebnisse argumentiert.

Die Tabelle zeigt die maximalen und mittleren Kosten bei Anwendung von Simple-Hopping und die Standardabweichung der Werte. Diese Metriken beziehen sich auf einen Datensatz bei dem für jede dynamische Instruktion ein Ergebnis generiert wurde. Dies entspricht im Allgemeinen nicht der Realität, da hier durch die Reduktion der Experimentmenge nur eine Untermenge dieser Navigationsziele tatsächlich verwendet werden. Dieser Umstand wird im Folgenden noch betrachtet, hier jedoch zunächst zur allgemeinen Betrachtung der Verfahren ignoriert.

Es zeigt sich, dass die Verwendung von Simple-Hopping im Kontext von FailPanda nicht tragbar wäre, da alle Benchmarks hohe durchschnittliche und maximale Kosten aufweisen. So würde die Navigation bei qsort-small – hierbei ergaben sich die besten Ergebnisse für Simple-Hopping – durchschnittlich eine Zeit von etwa 1,25 h benötigen.

Bei Betrachtung der Standardabweichung fällt auf, dass diese in allen Fällen in der Größenordnung des Mittelwerts liegt, wodurch die Werte sehr weit streuen. Demnach sind hier teilweise Experimente deutlich günstiger als andere Experimente.

Smart-Hopping

Tabelle 5.4 beschreibt in Anlehnung an die vorher betrachtete Tabelle 5.3 die Ergebnisse bei Einsatz des Smart-Hopping-Verfahrens. Hierbei werden die zwei Varianten der ausschließlichen Nutzung von Breakpoints und der Nutzung von Break- und Watchpoints

Tabelle 5.4: Ergebnisse der Berechnung von Sprungsequenzen mit dem Smart-Hopping-Algorithmus für MiBench-Benchmarks der Kategorien *automotive* und *network*.

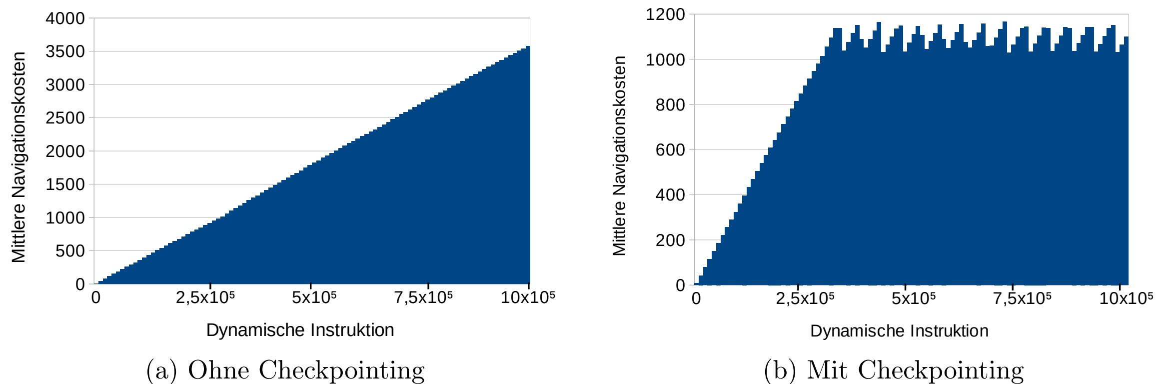
	Benchmark	Eingabe	Smart-Hopping (BP)			Smart-Hopping (BP+WP)		
			Max	**Mittel**	σ	**Max**	**Mittel**	σ
automotive	BASICMATH	large	12 889	215,2	1246,2	8367	114,5	702,9
		small	267	10,5	12,3	177	4,7	7,3
	BITCOUNT	large	301 372	150 200,6	86 956,6	301 372	124 995,0	92 278,2
		small	20 122	10 051,3	5807,5	20 122	8374,9	6170,8
	QSORT	large	59 562	3203,7	4838,8	5	3,1	0,7
		small	6318	708,9	879,8	8	2,7	0,7
	SUSAN	large	28 849	116,0	689,5	40	2,9	1,0
		small	1883	55,3	57,5	40	2,8	0,8
network	DIJKSTRA	large	534	179,8	96,3	9	4,6	1,0
		small	534	153,3	103,2	7	3,8	1,1
	PATRICIA	large	70	9,4	2,7	8	2,7	0,7
		small	70	7,9	2,5	8	2,6	0,7

getrennt betrachtet. Somit lässt sich die durch zusätzlichen Einsatz von Watchpoints gewonnene Kostenreduzierung identifizieren.

Es wird hier deutlich, dass der Einsatz von Smart-Hopping auf Basis von Breakpoints im Vergleich zum Simple-Hopping in jedem Fall eine Reduzierung der Navigationskosten um mehrere Größenordnungen bringt. Im Einzelfall ist diese Reduktion allerdings zur praktikablen Anwendung im Kontext von FailPanda nicht ausreichend. So sind die durchschnittlichen Navigationskosten bei dem Benchmark BITCOUNT in der *large*-Variante noch etwa doppelt so hoch, wie bei dem zuvor betrachteten QSORT-*small* bei Anwendung von Simple-Hopping. Es ergibt sich hierbei demnach eine durchschnittliche Navigationszeit von etwa 2,32 h, womit eine praktikable Anwendung unmöglich ist.

Es zeigt sich, dass bereits einige Benchmarks sehr niedrige Navigationskosten aufweisen, so liegt die durchschnittliche Navigationszeit bei dem PATRICIA-Benchmark beispielsweise in beiden Varianten deutlich unterhalb einer Sekunde. Einige Benchmarks weisen noch eine erhöhte Navigationszeit auf, was in fast allen Fällen durch den zusätzlichen Einsatz von Watchpoints eliminiert werden kann.

Beim Einsatz von Watchpoints konnten die Kosten beim BITCOUNT-Benchmark in beiden Varianten um lediglich etwa 16 % reduziert werden, sodass die Navigation in FailPanda in diesem Fall weiterhin in einem praktischen Zeitrahmen nicht möglich ist. Die Kosten aller restlichen Benchmarks konnten hier allerdings weiter reduziert werden, sodass mit Ausnahme der *large*-Variante von BASICMATH alle Benchmarks durchschnittliche Navigationszeiten von unter einer Sekunde aufweisen. Bei der genannten Ausnahme wird zwar im Durchschnitt eine Zeit von etwa 7,6 s zur Navigation benötigt, allerdings ist auch dieser Wert noch als tolerierbar zu bewerten.



(a) Ohne Checkpointing (b) Mit Checkpointing

Abbildung 5.2: Mittlere Navigationskosten (Bin-Größe: 10^4) für den Benchmark BITCOUNT reduziert auf die ersten 10^6 dynamischen Instruktionen. Abbildung 5.2a zeigt die Kosten ohne Einsatz von Checkpointing, in Abbildung 5.2b wurden die Kosten bei Checkpointing mit einer Kostendeckelung bei $\textit{CP-Threshold} = 1200$ berechnet.

Bei Betrachtung des Traces des BITCOUNT-Benchmarks fällt auf, dass dieser zu einem großen Teil in der Ausführung von Schleifen besteht. Da in einer Schleifenausführung üblicherweise sehr häufig nacheinander die gleichen Instruktionen ausgeführt werden, sind diese als schlecht für die Anwendung von Navigationsverfahren mit Verwendung von Breakpoints anzusehen. Werden innerhalb einer solchen sehr langen Schleife Speicherzugriffe durchgeführt, wobei sich die Adresse des Zugriffs in den verschiedenen Iterationen unterscheidet, so kann mithilfe des Einsatzes von Watchpoints ein Quereinsprung in eine der Iterationen erfolgen, wodurch die Navigation wieder schnell abläuft. Im Fall von BITCOUNT hat sich allerdings gezeigt, dass die sehr langen Schleifen auch keine Speicherzugriffe durchführen, wodurch dieser Benchmark unter Verwendung der hier entwickelten Verfahren sehr hohe Navigationskosten zeigt.

Die Kosten von Single-Stepping sind nicht weiter aufgeführt, entsprechen allerdings direkt der Anzahl dynamischer Instruktionen bis zur Zielinstruktion und damit im Durchschnitt der Hälfte dieses Wertes, weshalb das Verfahren grundsätzlich ausgeschlossen wurde.

Checkpointing

In Abschnitt 3.5.3 wurde Checkpointing als Erweiterung für die Trace-Navigation vorgeschlagen, um eine Deckelung von Kosten in Fällen hoher auftretender Navigationskosten zu erwirken, wie beispielsweise im Fall des BITCOUNT-Benchmarks. Das Verfahren wurde zwar im Kontext der Arbeit nicht für FailPanda implementiert, dennoch soll im Folgenden der mögliche Gewinn durch den Einsatz dargestellt werden.

Abbildung 5.2 zeigt für das Beispiel von BITCOUNT-*small* die Kosten für die Trace-Navigation mittels durch Smart-Hopping berechneter Sprungketten gemittelt über Bins der Größe 10^4 dynamischer Instruktionen. Die Grafik stellt hierbei nur die ersten 10^6

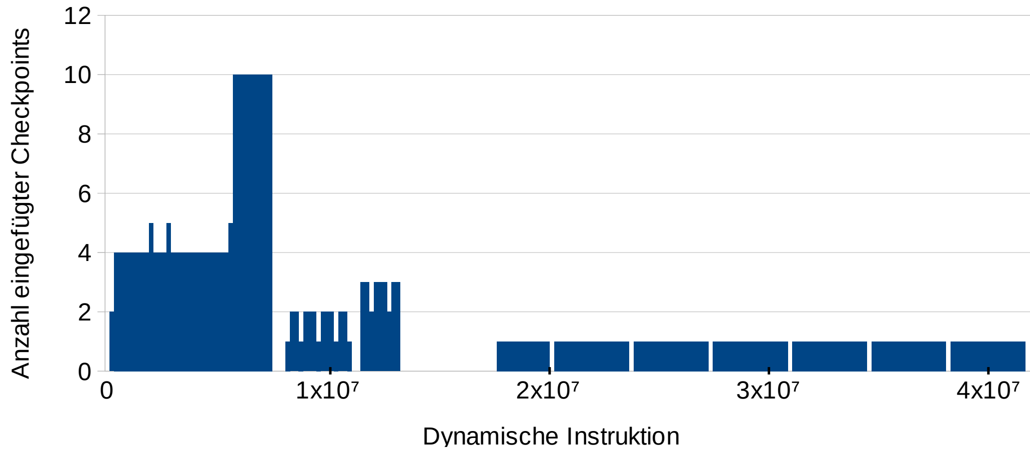


Abbildung 5.3: Anzahl produzierter Checkpoints (Bin-Größe: 2×10^5) bei der Ausführung von Smart-Hopping auf dem Benchmark BITCOUNT in der *small*-Variante mit einer Kostendeckelung bei *CP-Threshold* = 1200.

dynamischen Instruktionen dar, um das durch die Deckelung verursachte Muster sichtbar zu machen.

In Abbildung 5.2a werden zunächst die ursprünglichen Kosten ohne Verwendung von Checkpointing dargestellt. Hier ist ein lineares Kostenwachstum mit steigender Trace-Position zu erkennen. In Abbildung 5.2b sind die Kosten bei Einsatz von Checkpointing zur Deckelung der Kosten bei dem Wert *CP-Threshold* = 1200 zu sehen. Dieser Wert wurde gewählt, um zu zeigen, dass mit dem Einsatz einer geringen Anzahl an Checkpoints eine hohe Reduzierung der Kosten möglich ist. Die Kosten für das Laden eines Checkpoints via der OCD-Schnittstelle wurden hier mit 1000 abgeschätzt, da das Schreiben der für BITCOUNT relevanten Speicherbereiche (etwa 0,5 MB) in FailPanda etwa diese Kosten produziert. Der *Rollback-Threshold* wurde so angelegt, dass mindestens 50 Instruktionen über einen Checkpoint hinweg abgebaut werden müssen, damit dieser entfernt werden darf. Es wird deutlich, dass die ursprüngliche näherungsweise lineare Kostenfunktion durch das Checkpointing bei Erreichen des Deckelungswertes abgeschnitten wird, sodass die Kosten immer darunter liegen.

Somit lässt sich die durchschnittliche Navigationszeit in jedem Fall nach oben abschätzen, es entsteht allerdings ein zusätzlicher Aufwand zur Speicherung der Systemabbilder. Abbildung 5.3 zeigt dem Beispiel zugehörig die Anzahl verwendeter Checkpoints über die Verarbeitung des gesamten Traces ($4{,}18 \times 10^7$ dynamische Instruktionen) als Histogramm mit einer Bin-Größe von 2×10^5 dynamischen Instruktionen. Die Grafik zeigt, dass die Anzahl generierter Checkpoints klein gehalten wird, indem in den verschiedenen Abschnitten des Traces jeweils die aktuell nötige Anzahl an Checkpoints angelegt wird. Im Anfangsbereich des Traces sind sichtbar mehr Checkpoints erforderlich als beispielsweise in der zweiten Hälfte. Insgesamt werden 367 Checkpoints angelegt, was als für den praktischen Gebrauch annehmbar zu bewerten ist, da sich in Multiplikation mit der Checkpoint-Größe von etwa 0,5 MB eine gesamte Hintergrundspeicherbelastung von etwa 183,5 MB ergibt. Es lässt sich demnach schließen, dass Checkpointing eine geeig-

Tabelle 5.5: Verwendung der durch Pruning gewonnenen Freiheitsgrade durch äquivalenzklassengewahre Trace-Navigation. Es werden die durchschnittlichen Kosten bei Durchführung einer Kampagne für die *small*-Varianten der Benchmarks angegeben.

	Maximum		**Mittel**		
Benchmark	Rechter Rand	Optimiert	Rechter Rand	Optimiert	Gewinn
A-BASICMATH	172	171	4,304	3,886	9,72 %
A-BITCOUNT	20 111	20 102	5052,31	4899,87	3,017 %
A-QSORT	4	4	2,471	1,728	30,06 %
A-SUSAN	18	4	2,679	1,773	33,81 %
N-DIJKSTRA	7	7	3,677	2,640	28,19 %
N-PATRICIA	5	5	2,587	2,149	16,93 %

nete Erweiterung für die Trace-Navigation sein kann, wenn problematische Traces wie der des BITCOUNT-Benchmarks zu verarbeiten sind. Für alle anderen hier betrachteten Traces stellt das Checkpointing allerdings aufgrund der angenommenen hohen Kosten für das Einspielen eines Checkpoints keine Verbesserung dar.

Äquivalenzklassengewahre Trace-Navigation

In Abschnitt 3.5.4 wurde das Verfahren der äquivalenzklassengewahren Trace-Navigation vorgestellt, dem die Reduktion der Experimentmenge (s. Abschnitt 2.5) durch das sogenannte Def-Use-Pruning zugrunde liegt. Tabelle 5.5 stellt die durchschnittlichen und maximalen Kosten für die Ausführung einer Kampagne dar. Hierbei berücksichtigt das Fehlermodell nur Speicherfehler, sodass das Pruning auf Basis der Speicherzugriffe erfolgt. Die Werte wurden nur für die *small*-Varianten der Benchmarks bestimmt, womit allerdings bereits der mögliche Gewinn durch das Verfahren offensichtlich wird.

Das zur Bestimmung der Sprungketten verwendete Verfahren ist Smart-Hopping mit Verwendung von Breakpoints und Watchpoints. Die Kosten werden jeweils für die Verwendung der Instruktion am rechten Rand einer Äquivalenzklasse (übliches Vorgehen bei Fail*) den Kosten der Sprungkette mit minimalen Kosten innerhalb einer Äquivalenzklasse gegenübergestellt. Für eine einfachere Vergleichbarkeit der Werte ist zusätzlich der prozentuale Gewinn in Bezug auf die mittleren Kosten angegeben.

Es zeigt sich, dass auch die äquivalenzklassengewahre Trace-Navigation im Fall des BITCOUNT-Benchmarks keine Kostenreduktion auf ein praktisch nutzbares Niveau bringt. Hierbei werden die Kosten lediglich um etwa 3 % gesenkt. In den anderen Fällen ist der Gewinn jedoch teilweise deutlich höher. So reduzieren sich die Kosten bei den Benchmarks QSORT, SUSAN und DIJKSTRA jeweils etwa um 30 %. Bei den restlichen Benchmarks ist der Gewinn zwar geringer, es kann dennoch eine deutliche Ersparnis erkannt werden. Das Verfahren ist demnach in der Lage, eine deutliche Steigerung der Performanz zu erwirken, wenn die Werte der Sprungkosten für alle Trace-Positionen vorhanden sind.

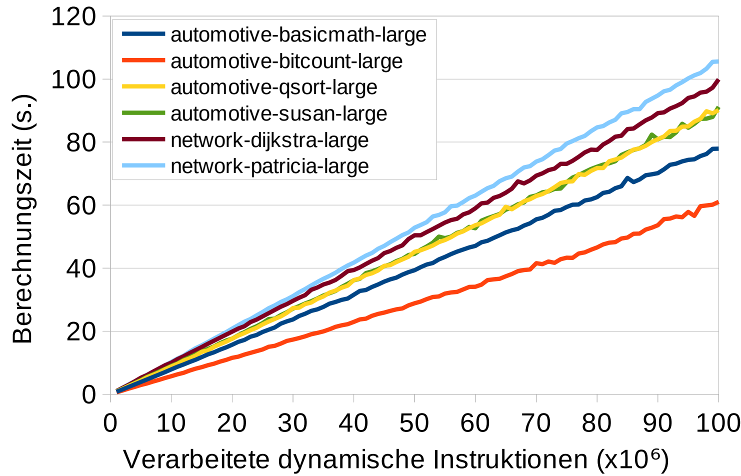


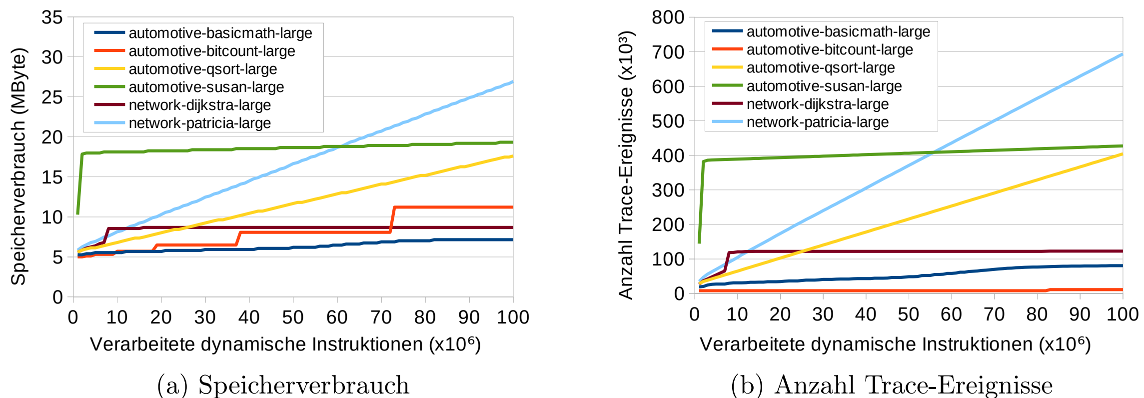
Abbildung 5.4: Berechnungszeit von Smart-Hopping aufgetragen gegen die Größe des Eingabe-Trace. Gemessen bei Einsatz des Kostenmodells und der Verwendung von Watchpoints. Es wurden jeweils für die x ersten dynamischen Instruktionen der large-Varianten der bereits vorgestellten Traces vollständige Lösungen berechnet.

Performanz des Smart-Hopping-Algorithmus

Abschließend wird der durch Smart-Hopping begründete Berechnungs- und Speicheraufwand evaluiert. In dem in Abschnitt 3.5.2 beschriebenen algorithmischen Entwurf wurde insbesondere vorausgesetzt, dass der Speicherverbrauch des Verfahrens nicht linear mit der Eingabedaten-Länge (Trace-Länge) wachsen darf. Außerdem wurde durch den Entwurf als auf einem Datenstrom operierendes Verfahren auch die Erreichung geringer Laufzeiten begünstigt.

Abbildung 5.4 zeigt konkrete Berechnungszeiten für die Berechnung der Sprungketten für die ersten 10^8 dynamischen Instruktionen der jeweiligen *large*-Varianten der bekannten Benchmarks. Die Berechnungen wurden auf einem Rechnersystem mit Intel Xeon CPU E5345 (2,33 GHz) durchgeführt. Es zeigt sich für alle Benchmarks ein lineares Verhältnis von Berechnungszeit zur Eingabelänge. Des Weiteren sind die angegebenen Trace-Längen aus Sicht des Kontextes von FailPanda sehr groß, sodass die Berechnungszeit von maximal etwa 100 s als praktikabel anzusehen ist.

Abbildung 5.5 zeigt den von Smart-Hopping verursachten Speicherverbrauch. Abbildung 5.5a zeigt den Speicherverbrauch in Anlehnung an die Untersuchung der Berechnungszeit für die Berechnung der Sprungketten zu den ersten 10^8 dynamischen Instruktionen für die *large*-Varianten aller Benchmarks. Da die abgebildeten Graphen ohne weitere Information zu den einzelnen Traces keinen Aufschluss über die dominierende Ursache des Speicherverbrauchs geben, ist zusätzlich in Abbildung 5.5b die zur Untersuchung zugehörige Anzahl an unterschiedlichen Trace-Ereignissen dargestellt. Diese Informationsquelle wurde gewählt, da der in Abschnitt 3.5.2 entworfene Algorithmus in der Hauptsache eine Datenstruktur mit einem Eintrag für jedes bereits aufgetretene



(a) Speicherverbrauch (b) Anzahl Trace-Ereignisse

Abbildung 5.5: Speicherverbrauch von Smart-Hopping in Relation zur Größe des Eingabe-Trace (s. Abb. 5.5a). Die Messung erfolgte bei Einsatz des Kostenmodells und der Verwendung von Watchpoints. Es wurden jeweils für die x ersten dynamischen Instruktionen der *large*-Varianten der bereits vorgestellten Traces vollständige Lösungen berechnet. Zur Erkennung der dominierenden Ursache für den Speicherverbrauch zeigt Abb. 5.5b die Anzahl verschiedener Trace-Ereignisse bei der jeweiligen Berechnung.

Trace-Ereignis und die letzte Lösung bereithält, wodurch ein potenziell hoher Speicherverbrauch erzeugt wird.

Im Vergleich von Abbildung 5.5a mit 5.5b ist eine klare Korrelation der Daten zu erkennen, weshalb darauf geschlossen werden kann, dass der Speicherverbrauch durch die Anzahl der Trace-Ereignisse und damit durch die Größe von Code und zugegriffenem Speicher dominiert wird. Lediglich der Speicherverbrauch des BITCOUNT-Benchmark weicht deutlich davon ab. Da dieser allerdings im Vergleich zu den anderen Traces während der gesamten Laufzeit die geringste Anzahl an unterschiedlichen Trace-Ereignissen aufweist und auch, wie die bisherigen Untersuchungen zeigten, große Ergebnis-Sprungketten erzeugt, ist in diesem Fall als dominierender Faktor die Größe des aktuellen Ergebnisses anzunehmen.

Insgesamt lässt sich feststellen, dass sich der Speicherverbrauch für alle Benchmarks in einem praktisch vertretbaren Rahmen bewegt. Somit kann davon ausgegangen werden, dass weder Berechnungs- noch Speicheraufwand des Verfahrens im praktischen Einsatz problematisch sein können.

5.1.3 MMU-Speicherüberwachung

Im Folgenden wird die MMU-Speicherüberwachung evaluiert, welche insbesondere in der Nachuntersuchung für die Erkennung von Speicherzugriffen in nicht erlaubten Bereichen verwendet wird. Hierbei wird einerseits der durch die nötige Konfiguration der MMU verursachte Mehraufwand und andererseits der durch falsch-positive Ergebnisse in der Ausführung verursachte Mehraufwand evaluiert.

Konfigurationsaufwand

Bei der Verwendung der MMU zur Erkennung von unzulässigen Speicherzugriffen in der Nachuntersuchung ist zunächst zu beachten, dass typischerweise ein großer Speicherbereich beobachtet wird. Dies gilt insbesondere bei FailPanda, da hier ein 32-Bit Adressraum zur Verfügung steht und die Zielapplikationen üblicherweise nur auf einen kleinen Teil dieses Speichers zugreifen. Aus diesem Grund werden an dieser Stelle nur MMU-Konfigurationen evaluiert, die einen Großteil des Adressraums abdecken.

In Abschnitt 3.7 wurde erläutert, dass die Beobachtung ganzer Sections deutlich performanter als die Verwendung von Seiten ist. Daher wird hier zunächst der Zeitaufwand für die Konfiguration einer kompletten Abdeckung des Speichers und demnach der Konfiguration aller 4096 Section-Deskriptoren gemessen. Es hat sich ergeben, dass diese Konfiguration eine Zeit von etwa 2,248 s benötigt.

Im Vegleich dazu wurde eine Messung vorgenommen, bei der acht gleich große Speicherbereiche (jeweils 512 MB) beobachtet wurden, wobei diese jeweils um die Größe einer Seite reduziert wurden. Es wurden demnach Lücken der Größe von 4 kB zwischen den großen beobachteten Bereichen gelassen. Daher werden pro beobachtetem Bereich 511 Sections und 256 Seiten verwendet. Die gemessene Konfigurationszeit beträgt in diesem Fall etwa 2,554 s. Hier zeigt sich, dass der Konfigurationsaufwand aufgrund der geeigneten Zusammenfassung zu blockweisen Schreibzugriffen nur gering gesteigert wird.

In einer letzten Untersuchung soll der schlechteste Fall für die Konfiguration der MMU gezeigt werden. Hier wird eine komplette Abdeckung des Speichers durchgeführt, welche ausschließlich auf der Verwendung von Page-Deskriptoren basiert. In diesem Fall ergab sich ein Zeitaufwand von etwa 672 s. Dieser sehr hohe Aufwand ist dadurch begründet, dass alle Elemente aller Seiten-Tabellen geschrieben werden müssen. Hierbei überwiegt demnach die über die OCD-Schnittstelle übertragene Datenmenge. Da eine derartige Konfiguration allerdings in der Praxis üblicherweise nicht auftritt, ist das Ergebnis nicht kritisch.

Üblicherweise kann der Konfigurationsaufwand etwa mit der beispielhaften Verwendung der acht Speicherbereiche abgeschätzt werden. Der sich hierbei ergebende Aufwand von etwa 2,554 s liegt in einem Bereich, welcher als praktisch anwendbar eingeordnet werden kann.

Zusätzlicher Aufwand durch falsch-positive Ergebnisse

Für die Evaluation des durch falsch-positive Ergebnisse verursachten Mehraufwands wurde die Zeit der Ausführung einer kurzen Applikation gemessen, die 454 Speicherzugriffe in einem Bereich durchführt, welcher innerhalb einer durch Konfiguration der MMU beobachteten Speicher-Seite liegt. Der Speicherbereich, der durch die MMU beobachtet werden soll, lag hierbei am Ende einer Speicherseite, während die Applikation nur Speicherzugriffe am Anfang der Seite durchführte. Somit wurde häufig der MMU-Ereignis-Mechanismus aktiviert, obwohl ein Speicherzugriff erfolgte, welcher nicht untersucht werden sollte. Es hat sich gezeigt, dass für die 454 Speicherzugriffe ein Mehraufwand von

Tabelle 5.6: Mittlere Verarbeitungszeit der vier Experimentphasen einer Beispielkampagne und zusätzliche Aufsummierung zur mittleren für ein Experiment benötigten Gesamtzeit. Zu allen Werten ist zusätzlich die zugehörige Standardabweichung σ angegeben, die die Streuung der Werte um den Mittelwert quantifiziert.

Phase	**Mittel**	σ
Initialisierung	1034 ms	25,0 ms
Trace-Navigation	221,3 ms	72,9 ms
Injektion	23,9 ms	0,000 08 ms
Nachuntersuchung	3055 ms	293 ms
Gesamt	4333,7 ms	300,9 ms

etwa 525,1 s verursacht wurde. Es wird demnach im Mittel pro auftretendem falsch-positiven Ergebnis ein zusätzlicher Zeitaufwand von etwa 1,16 s produziert.

Es ist hierbei zu beachten, dass ein sehr kurzes Beispielprogramm untersucht wurde. Entsprechend würden in länger laufenden Zielapplikationen potenziell wesentlich mehr falsch-positive Ergebnisse produziert, wenn ein Teil des aktiv genutzten Speichers untersucht würde. Es kann demnach gefolgert werden, dass bei aktueller Implementierung strengstens darauf zu achten ist, dass kein von der Zielapplikation verwendeter Speicherbereich (durch Grobgranulariät des Verfahrens) untersucht wird. Anderenfalls ist eine Kampagnen-Durchführung nicht in praktikabler Zeit durchführbar.

5.1.4 Gesamtsystem

Zur Bemessung der Performanz von FailPanda in Ausführung einer gesamten Kampagne wurde der Benchmark DIJKSTRA aus der MiBench-Sammlung verwendet. Für eine schnelle Verarbeitung wurde die *small*-Eingabe des Benchmarks weiter reduziert und direkt in die generierte Binär-Datei kodiert, sodass kein Auslesen einer Datei mehr nötig ist. Durch diese Modifikationen ist es möglich, die Trace-Länge auf 84 495 Instruktionen mit 33 314 Speicherzugriffen (21 531 lesende und 11 783 schreibende Zugriffe) zu reduzieren.

Die Aufzeichnung des Traces durch FailPanda benötigte etwa eine Zeit von 171,68 min. Die verwendeten Sprungketten zur Trace-Navigation wurden mit Smart-Hopping unter Verwendung von Break- und Watchpoints berechnet. Durch Anwendung der äquivalenzklassengewahren Trace-Navigation konnten mittlere Navigationskosten von 3,019 auf 1,973 reduziert werden. Die Kampagne sah ein Fehlermodell von 8-bit Burst-Fehlern in Bytes im Hauptspeicher vor. Diese Fehler sind durch die Invertierung eines Byte zu implementieren, wodurch die Menge durchzuführender Experimente im Vergleich zur Implementierung von Einzelbit-Fehlern auf ein Achtel reduziert wurde. Die Annahme dieses Fehlermodells wurde bereits in anderen Arbeiten zugrunde gelegt [BSS13b]. Nach einer Durchführung des Pruning auf Basis der aufgezeichneten Speicherzugriffe blieb eine Menge von 84 123 durchzuführenden Experimenten übrig.

Um die Performanz des Gesamtsystems zu evaluieren, wurde jedes Experiment in die bekannten vier Phasen der Initialisierung, Trace-Navigation, Injektion und Nachuntersuchung aufgeteilt. Die jeweiligen Zeiten wurden gemessen und als Experimentergebnis mit in die Datenbank übertragen, um eine einfache Auswertung zu ermöglichen. Tabelle 5.6 zeigt mittlere Verarbeitungszeiten und deren Standardabweichung σ für die vier Phasen. Da die Initialisierung einen konstanten Wert in der Experimentausführung ausmacht, ist dieser Wert auch in dieser Betrachtung unverändert gleich dem in Abschnitt 5.1.1 dargestellten. Die Trace-Navigation konnte mithilfe der vorgestellten Optimierungen auf einen geringen Wert reduziert werden, sodass diese nur etwa 5 % der gesamten Verarbeitungszeit ausmacht. Es zeigt sich bei Betrachtung der Standardabweichung, dass die erwartete Abweichung mit etwa 33 % eine moderate Schwankung der Kosten darstellt. Die Differenz zu dem theoretisch bestimmten Wert einer mittleren Navigationszeit ist darauf zurückzuführen, dass der Benchmark einige Ausgaben auf der seriellen Schnittstelle durchführt, wodurch die Ausführungsgeschwindigkeit deutlich reduziert und daher die Zeit für die eigentlichen Sprünge nicht mehr vernachlässigbar ist. Dieser zusätzliche Aufwand ist allerdings unabhängig von der verwendeten Navigationsstrategie.

Da bei der Injektion ein Byte des Speichers gelesen, invertiert und wieder geschrieben werden muss, wird etwa die doppelte Zeit eines einzelnen Speicherzugriffs benötigt. Dennoch fällt sie mit etwa 0,6 % nur minimal ins Gewicht. Der größte Anteil der Experimentzeit wird von der Nachuntersuchung eingenommen. Dies ist insbesondere der Fall, da die MMU-Konfiguration, wie Abschnitt 5.1.3 zeigte, einen hohen Zeitaufwand verursacht. Im Fall einer Zeitüberschreitung wird die Nachuntersuchung zudem um die Differenz zwischen normaler Ausführungszeit und der durch die Zeitbegrenzung mit 3 s definierten maximalen Ausführungszeit vergrößert.

Werden die Gesamtwerte betrachtet, so kann die Kampagnenausführung als noch praktisch durchführbar eingestuft werden. Für die Ausführung der gesamten Kampagne wurde eine Zeit von etwa 101,3 h benötigt. Es besteht allerdings Optimierungsbedarf bei der Konfiguration der MMU, da diese einen Großteil der Experimentausführung ausmacht.

Vergleich zu FailGem5

Um an dieser Stelle ein Fazit zur Performanz des entwickelten Systems im Vergleich zur Performanz von Simulator-basierter Fehlerinjektion zu ziehen, ist eine Betrachtung der Verarbeitungszeit unter unterschiedlichen Experimentlängen notwendig. Abbildung 5.6 schematisiert einen qualitativen Vergleich der Kosten bei Verwendung von FailPanda und dem Simulator-basierten Verfahren FailGem5. Es werden hier keine konkreten Werte verwendet, da diese für FailGem5 teilweise nicht verfügbar sind. Es wird jedoch davon ausgegangen, dass die Fixkosten für die Durchführung eines Experiments bei FailPanda zunächst oberhalb derer von FailGem5 liegen. Durch die höhere Ausführungsgeschwindigkeit der Hardware übersteigen allerdings ab einer Experimentlänge x die durchschnittlichen Experimentkosten von FailGem5 diejenigen von FailPanda. Aus diesem Grund ist FailPanda bei der Untersuchung sehr lang laufender Funktionen im Vorteil gegenüber der Simulator-basierten Variante. Im Fall von sehr lang laufenden Experimenten ist zwar das langsame Tracing ein Problem von FailPanda, allerdings sind auch Szenarien denk-

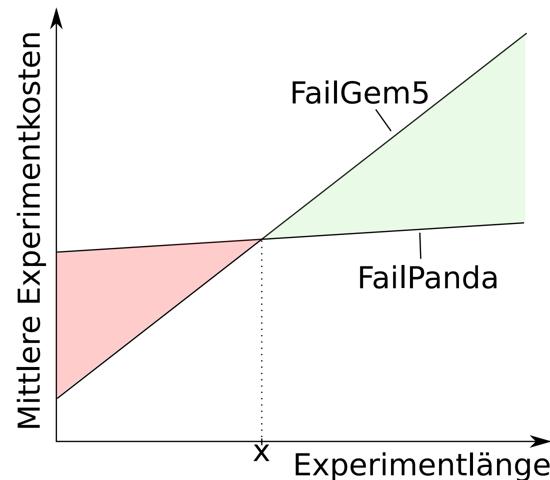


Abbildung 5.6: Schematische Darstellung der durchschnittlichen Experimentkosten im Vergleich zwischen FailPanda und FailGem5. Es werden qualitativ die höheren Fixkosten von FailPanda und der geringere Kostenwachstum bei steigender Experimentlänge dargestellt.

bar, in denen nur in einem kurzen Bereich zu Beginn des Experiments Fehler injiziert werden und anschließend die Applikation lange läuft, bis das Ergebnis berechnet wurde. In einem solchen Fall könnte die hohe Ausführungsgeschwindigkeit der Hardware bei geringen Kosten für das Tracing eine insgesamt hohe Performanz begründen.

Es wurden Messungen der Performanz einzelner Funktionen des auf dem Simulator Gem5 basierenden Fehlerinjektionssystems FailGem5 durchgeführt, um einen quantitativen Vergleich von FailPanda und FailGem5 durchführen zu können. Das Tracing verläuft bei Gem5 in einer Geschwindigkeit, die in der Größenordnung der Geschwindigkeit reiner Code-Ausführung des Simulators liegt. Im Mittel wird für das Tracing einer Instruktion eine Zeit von etwa 0,0231 ms benötigt, wobei ebenfalls die Erkennung von Speicherzugriffen durchgeführt wird. Dies entspricht einer um einen Faktor von etwa 3000 schnelleren Lösung im Vergleich zur Verwendung von FailPanda. Für die Ausführung einer Instruktion wird im Mittel eine Zeit von etwa 0,008 28 ms benötigt. Vergleichsweise liegt die Ausführungsgeschwindigkeit des Pandaboards, welches mit einer maximalen Taktrate von 1,2 GHz betrieben werden kann, in der Größenordnung von Nanosekunden[1]. Das Pandaboard weist demnach eine deutlich höhere Ausführungsgeschwindigkeit im Gegensatz zum Simulator auf.

Grundsätzlich ist bei Verwendung von FailGem5 zur Systeminitialisierung vorgesehen, dass ein Checkpoint geladen wird, welcher das System zu Beginn der zu untersuchenden Funktion repräsentiert. Der Vorgang der Generierung eines Checkpoints benötigt bei einem kurzen Benchmark eine Zeit von etwa 1,87 s. Da dieser Vorgang allerdings nur einmal vor der Ausführung einer Kampagne durchgeführt werden muss, fällt er nicht

[1] Für die Ausführung einer Instruktion wird potenziell mehr als ein Takt benötigt, weshalb eine Umrechnung der Taktrate in die Ausführungsgeschwindigkeit einer Instruktion im Allgemeinen nicht möglich ist.

weiter ins Gewicht. Das Laden eines solchen Checkpoints benötigt etwa eine Zeit von 1,23 s und liegt damit nahe der Initialisierungszeit von FailPanda.

FailGem5 zeigt nur einen geringen Mehraufwand bezüglich der Aktivierung von Break- und Watchpoints. Speicher- und Registerzugriffe fallen im Vergleich zur Zeit für die Instruktionsausführung ebenfalls nicht ins Gewicht. Auf die Angabe von exakten Werten wird hier allerdings verzichtet.

5.2 Experimentergebnisse

In Abschnitt 5.1.4 wurden Performanzmessungen anhand der Ausführung einer Kampagne auf Basis des stark reduzierten DIJKSTRA-Benchmarks vorgestellt. In diesem Abschnitt werden die durch Ausführung der Kampagne generierten Ergebnisse diskutiert.

Das Fehlermodell sieht Bitinvertierungen im Speicher vor, wobei immer alle acht Bits eines Speicher-Bytes invertiert werden. Für die Nachuntersuchung wurde eine Klassifizierung von Fehler-Auswirkungen vorgenommen, die eine Einteilung in die Klassen fehlerfrei, fehlerhaftes Ergebnis, Trap, Zeitüberschreitung und unerlaubter Speicherzugriff durchführt. Hierbei ist zu beachten, dass der modifizierte DIJKSTRA-Benchmark Ergebnisse in einen speziellen Speicherbereich schreibt, welcher nach jedem erfolgreich terminierten Experiment ausgelesen und mit dem fehlerfreien Ergebnis des Golden Run abgeglichen werden kann. Eine Untersuchung der ebenfalls durch den Benchmark produzierten seriellen Ausgabe wurde nicht durchgeführt, da für FailPanda das Auslesen der seriellen Schnittstelle nicht implementiert wurde. Bei Traps wird keine Unterscheidung verschiedener Arten durchgeführt. Ein durch die MMU ausgelöster Trap ist allerdings aus der erkennbaren Menge ausgeschlossen. Die Zeitüberschreitung wurde zu einem Wert von 3 s gewählt. Zu den unerlaubten Speicherzugriffen wurden alle Speicherbereiche außerhalb des Programmcodes (.text-Segment) der verwendeten Daten (.data- und .bss-Segment), des Stacks und der speicherabgebildeten seriellen Ausgabe gezählt. Damit keine falsch-positiven MMU-Ereignisse bei fehlerfreier Ausführung erfolgten, wurden die beobachteten Speicherbereiche so angelegt, dass sie an der nächsten 4 kB-Adresse ausgerichtet sind. Hierdurch werden zwar potenziell fehlerhafte Speicherzugriffe als fehlerfrei identifiziert, es wird allerdings davon ausgegangen, dass die Ergebnis-Verfälschung nur gering ist.

Wie in Abschnitt 2.3 erläutert wurde, können verschiedene Fehlertoleranz-Verfahren, die auf einen Benchmark angewendet wurden, mittels der absoluten Anzahl aufgetretener Fehler verglichen werden. Zu diesem Zweck ist es zunächst notwendig, die Ergebnisse der durch Def-Use-Pruning gewonnenen Äquivalenzklassen auf den gesamten Fehlerraum zurück zu projizieren. Dies wurde durchgeführt, indem die result-Tabelle der Datenbank mithilfe der Tabelle fspgroup mit der trace-Tabelle verknüpft wurde. Hierdurch wird für jedes Ergebnis die Breite der zugehörigen Äquivalenzklasse bestimmt. So können die Breiten der Äquivalenzklassen, deren Ergebnis einer Klasse zugehören, aufsummiert werden, um absolute Fehlerzahlen zu bestimmen.

In dieser Untersuchung wurde für die Definition der Breite der Äquivalenzklassen die Anzahl zugehöriger Instruktionen verwendet, da der Zyklenzähler des Pandaboards

Tabelle 5.7: Absolute Fehlerzahlen als Ergebnis einer beispielhaften Fehlerinjektionskampagne auf Basis des DIJKSTRA-Benchmarks. Es sind zusätzlich die Anteile an der Gesamtmenge angegeben.

Typ	**Anzahl**	**Anteil Gesamtmenge**
Fehlerfrei	130 076 001	80,94 %
Fehlerhaftes Ergebnis	2 166 675	1,348 %
Trap	4 485 077	2,790 %
Zeitüberschreitung	163 215	0,102 %
Unerlaubter Speicherzugriff	23 820 328	14,82 %
Gesamt	160 711 296	100 %

durch die Verwendung von OCD-Funktionen gestört wird und daher potenziell das Ergebnis verfälscht. Es wird angenommen, dass die Anzahl der in einer Äquivalenzklasse enthaltenen Instruktionen eine bessere Annäherung an die reale zeitliche Breite der Klasse bietet.

Tabelle 5.7 zeigt die berechneten absoluten Fehlerzahlen des Kampagnen-Ergebnisses. Zur einfacheren Betrachtung der Ergebnisse sind ebenfalls prozentuale Anteile an der durch den Fehlerraum definierten Gesamtmenge an Tupeln aus Injektionsort und -zeit angegeben. Es wird deutlich, dass der Hauptteil der Experimente (80,94 %) fehlerfrei verlief. Demnach führt in etwa einem Fünftel des Fehlerraums ein auftretender 8-Bit-Burst-Fehler zu einem nach außen sichtbarem fehlerhaften Verhalten des Systems. Dieses fehlerhafte Verhalten wird dominiert durch unerlaubte Speicherzugriffe (14,82 % der Gesamtmenge). Traps treten in etwa 2,790 % aller Fälle auf. Fehlerhafte Ergebnisse machen einen Anteil von 1,348 % aus. Der deutlich am wenigsten auftretende Experimentausgang ist mit 0,102 % eine Zeitüberschreitung nach Fehlerinjektion.

Eine Auswertung des Ergebnisses kann ebenfalls auf Basis einer grafischen Darstellung des Fehlerraums und der eingezeichneten Nachuntersuchungsergebnisse durchgeführt werden. Abbildung 5.7 zeigt eine solche Darstellung für einen Ausschnitt des Fehlerraums. Hier können Systemanteile identifiziert werden, deren Absicherung als besonders wichtig angesehen wird. In einem Detailausschnitt wird eine globale Variable des DIJKSTRA-Benchmarks betrachtet (`QITEM *qHead`). Hierbei handelt es sich um einen Listenkopf, welcher auf den Beginn einer Liste verweist, die eine dynamisch veränderte aktuelle Lösung des Verfahrens repräsentiert. Diese Variable stellt demnach ein wichtiges Element des Verfahrens dar.

Bei der Betrachtung ist zu beachten, dass der Prozessor im *little-endian*-Modus betrieben wurde. Daher sind weniger signifikante Anteile der in der Variable gesicherten Adresse an den niedrigeren Speicherstellen zu finden. Es ist zu erkennen, dass bei der Fehlerinjektion in dem am wenigsten signifikanten Byte der 4 B großen Variable in der Folge ein Trap ausgelöst wird. Dies ist damit zu begründen, dass durch die 8-Bit-Invertierung der Adresse eine Konvertierung in eine ungerade Adresse durchgeführt wird. Bei einem Speicherzugriff wird demnach ein Alignment Trap ausgelöst, da ein Speicherzugriff der Länge 4 nur an einer 4 B-ausgerichteten Adresse durchgeführt werden darf. Bei Injektion

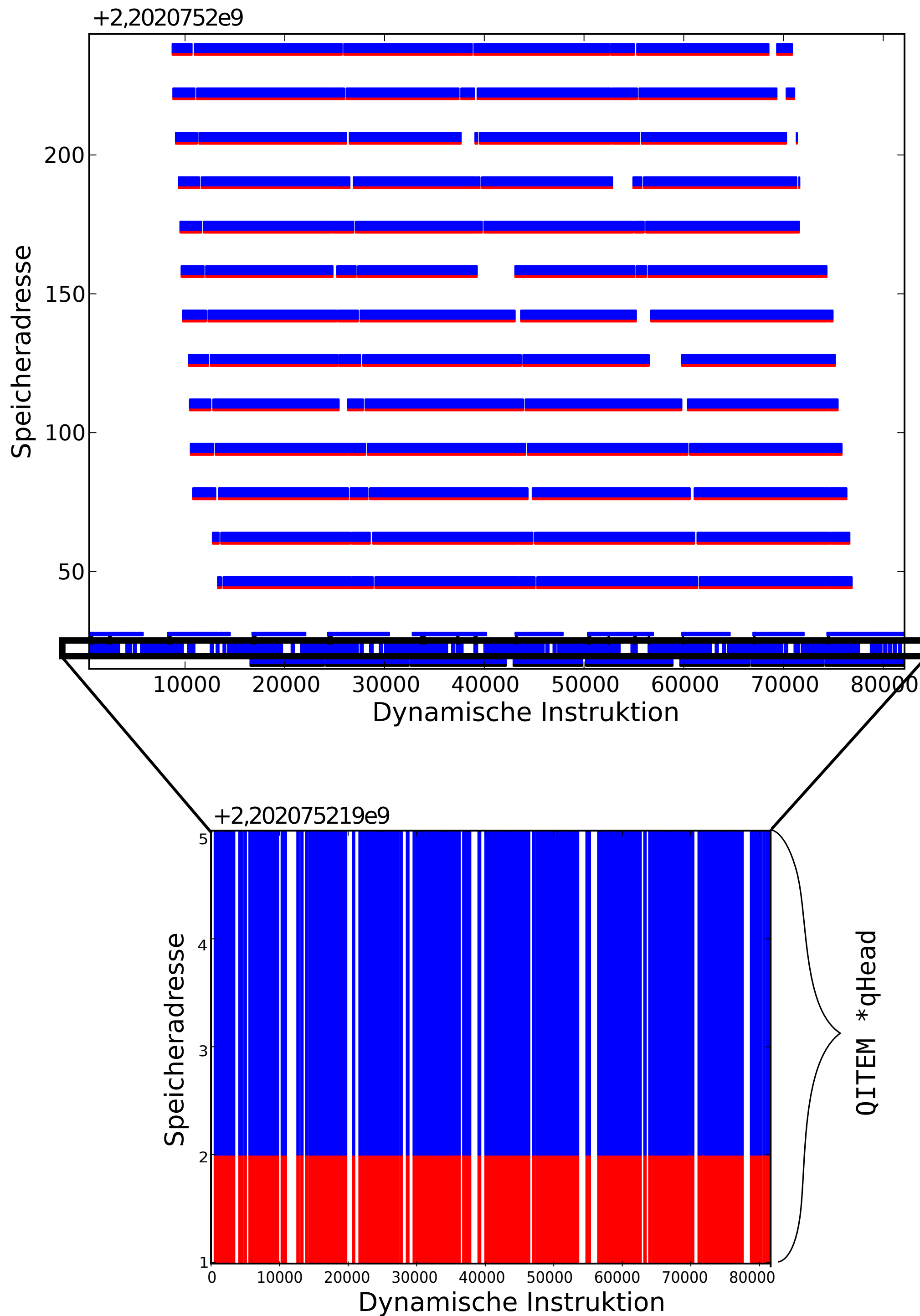


Abbildung 5.7: Ausschnitt aus einer grafischen Darstellung des Fehlerinjektionsergebnisses einer auf dem DIJKSTRA-Benchmark basierenden Kampagne. Weiße Bereiche sind fehlerfrei, rote Bereiche zeigen einen Trap, blaue Bereiche zeigen Speicherzugriffe außerhalb des erlaubten Bereichs, schwarze Bereiche sind fehlerhafte Ergebnisse und grüne Bereiche zeigen Zeitüberschreitungen. Ein Teilausschnitt zeigt die Ergebnisse für eine Variable der Breite von 4 Byte.

in den drei restlichen Byte der Adresse wird der Wert immer so stark abgewandelt, dass ein Speicherzugriff außerhalb des erlaubten Bereichs erfolgt.

Beide Fehlerarten treten in einem Großteil der Gesamtlaufzeit des Benchmarks auf. Es zeigt sich daher, dass die Fehlerfreiheit des Wertes der Variable für die korrekte Durchführung des DIJKSTRA-Benchmarks sehr wichtig ist. Aufgrund einer derartigen Untersuchung der verschiedenen vom Benchmark verwendeten Speicheranteile kann eine maßgeschneiderte Konfigurierung mittels Fehlertoleranzmaßnahmen erfolgen. Im Fall von `qHead` wäre somit beispielsweise eine Maßnahme anzuwenden, die vor jedem Zugriff den korrekten Wert der Variable prüft.

5.3 Zusammenfassung

In diesem Kapitel wurde eine Evaluation der Performanz von FailPanda durchgeführt und es wurde ein beispielhaftes Ergebnis einer Fehlerinjektionskampagne vorgestellt und untersucht. Der Vergleich mit FailGem5 hat gezeigt, dass bezüglich der Performanz Defizite im Bereich des Tracings und der Nachuntersuchung bestehen. Es wurden im Kontext der Arbeit bereits alternative Entwurfskonzepte vorgestellt, bei deren Einsatz eine deutliche Erhöhung der Performanz in diesen Bereichen erwartet wird. Diese Alternativen wurden allerdings aufgrund von Beschränkungen durch die Hardware oder aufgrund einer potenziellen Überschreitung des zeitlichen Rahmens einer Masterarbeit nicht realisiert. Es konnte gezeigt werden, dass die entwickelten Verfahren zur Trace-Navigation eine Verbesserung der Performanz um einige Größenordnungen ermöglichen. Somit konnte dieser als performanzkritisch identifizierte Teil des Gesamtverfahrens für einen Großteil der untersuchten Benchmarks auf einen Mehraufwand reduziert werden, der einer praktischen Nutzung nicht entgegen spricht. Die Demonstration einer beispielhaften Ergebnisuntersuchung einer mittels FailPanda durchgeführten Fehlerinjektionskampagne zeigte schließlich die praktische Einsetzbarkeit des Systems.

6 Ergebnisse und Ausblick

Die grundsätzliche Zielsetzung dieser Arbeit sah vor, die Realisierbarkeit eines OCD-Fehlerinjektionssystems auf Basis von eingebetteter COTS-Hardware durchzuführen. Hierbei sollten Vorteile gegenüber der Verwendung eines auf Nutzung von Hardware-Simulatoren basierenden Systems gezeigt werden. Zu diesem Zweck wurde ein entsprechendes Hardware-System an das Fehlerinjektions-Framework Fail* angebunden und untersucht. Auf Basis der in Abschnitt 5 vorgestellten Evaluation des Systems kann auf die praktische Einsetzbarkeit geschlossen werden. Es wurden jedoch auch Probleme des Systems identifiziert, die hier ebenfalls zusammengefasst werden sollen.

Im Folgenden werden in Abschnitt 6.1 zunächst die Ergebnisse der Arbeit vorgestellt. Anschließend zeigt Abschnitt 6.2 einen Ausblick auf mögliche Weiterführungen der Entwicklung und Untersuchungen zur Verbesserung des Systems.

6.1 Ergebnisse

Grundsätzlich wurde in dieser Arbeit gezeigt, dass die Verwendung von COTS-Hardware zur Realisierung eines Fehlerinjektions-Frameworks möglich ist. Es wurden zwar nicht alle von Fail* angebotenen Schnittstellenfunktionen implementiert, jedoch ist mit den implementierten Schnittstellen die vollständige Durchführung einer Kampagne möglich. Abschnitt 5.2 hat beispielhaft die Ergebnisse einer durchgeführten Kampagne gezeigt, auf deren Grundlage eine Bewertung des Systems und eine Verbesserung der Fehlertoleranz mit entsprechenden Software-Maßnahmen durchgeführt werden kann.

Da bereits die Anbindung eines Simulator-Backends zur Simulation von ARM-Prozessoren an Fail* existiert, sollte gezeigt werden, welche Vorteile sich durch die zusätzliche Anbindung eines entsprechenden Hardwaresystems bieten. Zunächst ist offensichtlich, dass die Ergebnisse von Fehlerinjektion auf dem Hardwaresystem potenziell realitätsgetreuer als die des Simulators sind, da das Verhalten des Simulators von dem der Hardware stellenweise abweichen kann. Des Weiteren wird durch die Verwendung von FailPanda der zu untersuchende Programmcode deutlich schneller als von einem Simulator ausgeführt. Vorteile der Hardwarelösung bezüglich Performanz werden demnach deutlich, wenn die Zielapplikation lange ausgeführt wird und daher einen langen Trace aufweist, sodass sowohl die Trace-Navigation als auch die Nachuntersuchung deutlich performanter als auf einem Simulator-Backend durchgeführt werden kann.

Diese Vorteile müssen jedoch stets unter dem Nachteil der geringen Parallelisierbarkeit von OCD-Fehlerinjektion im Gegensatz zu Simulator-basierter Fehlerinjektion gesehen werden. So ist es möglich, Performanznachteile durch den Einsatz zusätzlicher Simulationsrechner zu kompensieren. Um eine gesamte Verbesserung der Performanz zu erwirken,

wäre es demnach beispielsweise möglich, eine hybride Lösung zu verfolgen, welche lang laufende Experimente mithilfe von FailPanda und restliche Experimente mit FailGem5 durchführt.

Die im Kontext dieser Arbeit entwickelten Funktionalitäten weisen teilweise Defizite in der Performanz auf. Eine kritische Komponente ist hierbei die Verwendung der MMU zur Erkennung von Speicherzugriffen. Durch sie wird aufgrund des hohen Kommunikationsaufwands die Zeit der Experimentausführung deutlich erhöht. Abschnitt 5.1.3 hat gezeigt, dass durch die Behandlung jedes von der MMU ausgelösten Ereignisses auf dem Hostrechner Defizite in der Performanz zu verzeichnen sind. Insgesamt ist das entwickelte System dennoch praktisch einsetzbar. Es weist an den genannten Stellen lediglich Optimierungspotenzial auf.

6.2 Ausblick

Bei der Analyse des im Kontext dieser Arbeit entwickelten OCD-Fehlerinjektionssystems FailPanda wurden einige Erkenntnisse erlangt, die einen entsprechenden Ausblick auf mögliche darauf aufbauende Arbeiten ermöglichen. So kann beispielsweise eine weitere Ausnutzung der Multikern-Architektur des Pandaboards der Steigerung der Gesamtperformanz dienen. In Kapitel 3 wurden bereits derartige Optimierungen vorgestellt, welche allerdings aus zeitlichen Gründen nicht umgesetzt wurden.

Da die Checkpointing-Funktionalität im Kontext dieser Arbeit nicht implementiert wurde, eine Implementierung mittels OCD-Schnittstelle allerdings aufgrund der Daten aus Abschnitt 5.1.1 als zeitintensiv angenommen wird, ist eine Implementierung unter Nutzung eines Cortex-M3-Prozessorkerns vorteilhaft. Da der ausschlaggebende Faktor hier in der Absicherung und Wiederherstellung des Applikationsspeichers besteht, würde eine Implementierung dieser Funktionalität auf dem Pandaboard ausreichen, während die Registerwerte weiterhin über die JTAG-Schnittstelle abgerufen und zurückgeschrieben werden könnten. Um eine Datenübertragung zu verhindern, lassen sich die Daten des Checkpoints direkt im Festspeicher der Zielhardware absichern. Zu diesem Zweck muss ein spezieller Speicherbereich bereitgestellt werden. Eine deutliche Steigerung der Performanz im Vergleich zur Verwendung der OCD-Schnittstelle ist zu erwarten, da das Lesen der SD-Karte und das Schreiben im Speicher von einem Prozessorkern um Größenordnungen schneller durchgeführt werden kann als das Schreiben per OCD-Schnittstelle.

Eine weitere zu untersuchende Alternative liegt in der Übertragung von Checkpoints über die Ethernet-Schnittstelle und entsprechender Anwendung durch einen Cortex-M3-Prozessor. Der Vorteil dieses Verfahrens liegt darin, dass Checkpoints nicht auf den Zielsystem abgesichert werden müssen. Das Verfahren wäre insbesondere im Einsatz flexibler.

Die Programmierung der MMU zur Erkennung von Speicherzugriffen weist hohe Performanzdefizite auf. Das Verfahren hat zwei grundsätzliche Probleme. Einerseits ist das Schreiben der Seitentabelle kostenintensiv (s. Abschnitt 5.1.3) und andererseits begründet die grobe Granularität des Verfahrens eine große Zahl an falsch-positiven Ergebnissen, deren Bearbeitung in der im Kontext der Arbeit entwickelten Lösung auf dem

Hostrechner erfolgt. Es kann daher darauf geschlossen werden, dass eine Implementierung der gesamten Verwaltung der MMU auf der Zielhardware eine deutliche Steigerung der Performanz ermöglicht. Bei dem so modifizierten Verfahren wird demnach nur noch ein Ereignis an den Hostrechner gemeldet, wenn auf einen tatsächlich untersuchten Speicherbereich zugegriffen wird. Der Hostrechner muss zu diesem Zweck lediglich die zu überwachenden Speicherbereiche, definiert als Intervalle, an das System übertragen, wodurch der Aufwand deutlich reduzierbar ist.

Abschnitt 5.1.4 hat gezeigt, dass das Tracing auf dem Hostrechner zwar für kurz laufende Zielapplikationen praktisch einsetzbar ist, jedoch im Vergleich zum Tracing mittels Simulator um Größenordnungen langsamer ist. Hier besteht das Potenzial, die auf der Hardware vorhandene Tracing-Unterstützung zu nutzen, um einen Instruktions-Trace bei voller Ausführungsgeschwindigkeit aufzuzeichnen. Um Speicherzugriffe zu erkennen, wäre weiterhin die Verwendung der MMU ebenfalls ausschließlich auf dem Hardwaresystem zu implementieren. Sie könnte somit für das Tracing von Speicherzugriffen verwendet werden.

Um eine bessere Unterstützung für das Tracing zu erhalten, ist es ebenfalls möglich, auf kostenintensivere Debugger wie den ARM DSTREAM zurückzugreifen. Es wird angenommen, dass durch die Verwendung derartiger Debugger auch die Ausführung sonstiger OCD-Operationen deutlich performanter möglich ist. Ein derartiger Ansatz widerspricht jedoch der grundsätzlichen einer Voraussetzung dieser Arbeit, nämlich der Verwendung günstiger Hardware.

Bei Verwendung eines Fehlermodells, welches Speicherfehler, aber keine Fehler in den Prozessor-Caches vorsieht, kann mittels FailPanda eine Injektion in den RAM getätigt werden, ohne den zugehörigen Cache-Inhalt zu invalidieren. Dieses Verfahren führt demnach viele Fehlerinjektionen durch, welche durch die Verwendung von Caches maskiert werden und sich daher nicht auswirken können. Ein alternativer Ansatz könnte verfolgt werden, indem mithilfe eines Cache-Simulators diejenigen Experimente identifiziert werden würden, bei denen der Cache die Injektion überdeckt. Diese Experimente sind a priori als fehlerfrei zu markieren. Bei Einsatz eines derartigen Verfahrens wird erwartet, dass eine deutliche Reduktion der nötigen Experiment-Menge möglich ist.

Auf Grundlage der Entwicklungen dieser Arbeit wäre es möglich, eine Identifikation derjenigen Experimente durchzuführen, deren Ergebnisse auf Simulator und Hardware abweichen. Stimmen die Ergebnisse weitestgehend überein, so kann die Aussagekraft von Simulator-basierter Fehlerinjektion mit dieser Untersuchung untermauert werden. Mithilfe der Untersuchung könnte es jedoch auch möglich sein, Regeln für abweichende Ergebnisse zu extrahieren. So könnte für einen spezifischen Simulator herausgefunden werden, dass Fehlerinjektionsexperimente beispielsweise in Elementen einer Gleitkomma-Einheit nicht immer korrekte Ergebnisse erzielen. Darauf basierend wäre der Aufbau einer hybride Lösung aus Simulator- und OCD-basierter Fehlerinjektion möglich, wobei Experimente der abweichenden Klassen auf der Hardware durchgeführt werden, wohingegen Experimente der Klassen, deren Ergebnisse nicht abweichen, auf Basis von Simulator-basierter Fehlerinjektion generiert werden könnten. Somit wäre eine Stärkung der Aussagekraft und eine Erhöhung der Performanz des Gesamtsystems möglich. In dem

Beispiel wären daher alle Experimente, die Fehler in der Gleitkomma-Einheit injizieren, mithilfe der OCD-basierten Fehlerinjektion durchzuführen.

Literaturverzeichnis

[AAA+90] Arlat, J. ; Aguera, M. ; Amat, L. ; Crouzet, Y. ; Fabre, J. C. ; Laprie, J. C. ; Martins, E. ; Powell, D.: Fault injection for dependability validation: a methodology and some applications. In: *Software Engineering, IEEE Transactions on* 16 (1990), Nr. 2, S. 166–182. – ISSN 0098–5589

[ARM05] ARM: *ARM Architecture Reference Manual*, 2005

[ARM09] ARM: *Architectures, Processors, and Devices - Development Article*, 2009

[ARM11a] ARM: *ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition*, 2011

[ARM11b] ARM: *Cortex-A Series Programmer's Guide*. Version: 2.0, 2011

[ARM13a] ARM: *big.LITTLE Processing*. Version: 2013. `http://www.arm.com/products/processors/technologies/biglittleprocessing.php`, Abruf: 18.12.2013

[ARM13b] ARM: *DSTREAM*. Version: 2013. `www.arm.com/dstream`, Abruf: 16.12.2013

[AVFK01] Aidemark, J. ; Vinter, J. ; Folkesson, P. ; Karlsson, J.: GOOFI: generic object-oriented fault injection tool. In: *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, 2001, S. 83–88

[BBB+11] Binkert, N. ; Beckmann, B. ; Black, G. ; Reinhardt, S. K. ; Saidi, A. ; Basu, A. ; Hestness, J. ; Hower, D. R. ; Krishna, T. ; Sardashti, S. ; Sen, R. ; Sewell, K. ; Shoaib, M. ; Vaish, N. ; Hill, M. D. ; Wood, D. A.: The Gem5 Simulator. In: *SIGARCH Comput. Archit. News* 39 (2011), aug, Nr. 2, S. 1–7. – ISSN 0163–5964

[bea13] beagleboard.org: *BeagleBoard*. Version: 2013. `http://www.beagleboard.org/Products/BeagleBoard`, Abruf: 23.11.2013

[BGOS12] Butko, A. ; Garibotti, R. ; Ost, L. ; Sassatelli, G.: Accuracy evaluation of GEM5 simulator system. In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, 2012, S. 1–7

[BH05] Berg, E. ; Hagersten, E.: Fast Data-locality Profiling of Native Execution. In: *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA : ACM, 2005 (SIGMETRICS '05). – ISBN 1–59593–022–1, 169-180

[Bor05] Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. In: *Micro, IEEE* 25 (2005), Nr. 6, S. 10–16. – ISSN 0272–1732

[BP03] Benso, A. ; Prinetto, P.: *Fault injection techniques and tools for embedded systems reliability evaluation*. Boston, Dordrecht, London : Kluwer academic publ. cop., 2003 (Frontiers in electronic testing). `http://opac.inria.fr/record=b1100851`. – ISBN 1–4020–7589–8

[BRIM98] Benso, A. ; Rebaudengo, M. ; Impagliazzo, L. ; Marmo, P.: Fault-list collapsing for fault-injection experiments. In: *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*, 1998. – ISSN 0149–144X, S. 383–388

[BSS13a] Borchert, C. ; Schirmeier, H. ; Spinczyk, O.: Generative Software-based Memory Error Detection and Correction for Operating System Data Structures. In: *Proceedings of the 43nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, IEEE Computer Society Press, jun 2013. – ISBN 978–1–4673–6471–3

[BSS13b] Borchert, C. ; Schirmeier, H. ; Spinczyk, O.: 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)*, German Society of Informatics, sep 2013 (Lecture Notes in Informatics)

[BVFK05] Barbosa, R. ; Vinter, J. ; Folkesson, P. ; Karlsson, J.: Assembly-level pre-injection analysis for improving fault injection efficiency. In: *Dependable Computing-EDCC 5*. Springer, 2005, S. 246–262

[Cod13] CodeCogs: *Stdlib.h - C*. Version: 2013. `http://www.codecogs.com/library/computing/c/stdlib.h/index.php`, Abruf: 09.12.2013

[Con63] Conway, M. E.: Design of a Separable Transition-diagram Compiler. In: *Commun. ACM* 6 (1963), jul, Nr. 7, S. 396–408. – ISSN 0001–0782

[Dij59] Dijkstra, E. W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik* 1 (1959), S. –269

[DNR02] Dupont, E. ; Nicolaidis, M. ; Rohr, P.: Embedded robustness IPs for transient-error-free ICs. In: *Design Test of Computers, IEEE* 19 (2002), Nr. 3, S. 54–68. – ISSN 0740–7475

[DYDS+10] Duranton, M. ; Yehia, S. ; De Sutter, B. ; De Bosschere, K. ; Cohen, A. ; Falsafi, B. ; Gaydadjiev, G. ; Katevenis, M. ; Maebe, J. ; Munk, H. ; Navarro, N. ; Ramirez, A. ; Temam, O. ; Valero, M.: *The Hipeac Vision, 2010*. 2010

[eli13] elinux.org: *Panda How to MLO & u-boot*. Version: 2013. `http://elinux.org/Panda_How_to_MLO_%26_u-boot`, Abruf: 05.12.2013

[FAF06a] Fidalgo, A. V. ; Alves, G. R. ; Ferreira, J. M.: Real time fault injection using a modified debugging infrastructure. In: *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, 2006, S. 6

[FAF06b] Fidalgo, A. V. ; Alves, G. R. ; Ferreira, J. M.: Real Time Fault Injection Using Enhanced OCD – A Performance Analysis. In: *Defect and Fault Tolerance in VLSI Systems, 2006. DFT '06. 21st IEEE International Symposium on*, 2006, S. 254–264

[FGAF06] Fidalgo, A. V. ; Gericota, M. G. ; Alves, G. R. ; Ferreira, J. M.: Using NEXUS compliant debuggers for real time fault injection on microprocessors. In: *SBCCI*, 2006, S. 214–219

[FSK98] Folkesson, P. ; Svensson, S. ; Karlsson, J.: A comparison of simulation based and scan chain implemented fault injection. In: *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, 1998. – ISSN 0731–3071, S. 284–293

[GBT04] Ghosh, S. ; Basu, S. ; Touba, N. A.: Reducing power consumption in memory ECC checkers. In: *Test Conference, 2004. Proceedings. ITC 2004. International*, 2004, S. 1322–1331

[GDK11] Gligoric, N. ; Dejanovic, I. ; Krco, S.: Performance evaluation of compact binary XML representation for constrained devices. In: *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, 2011, S. 1–5

[GNU13a] GNU Project: *Autoconf - GNU Project - Free Software Foundation (FSF)*. Version: 2013. `http://www.gnu.org/software/autoconf/`, Abruf: 05.12.2013

[GNU13b] GNU Project: *GDB: The GNU Project Debugger*. Version: 2013. `https://www.gnu.org/software/gdb/`, Abruf: 16.12.2013

[Goo13] Google: *Protocol Buffers - Google's data interchange format*. Version: 2013. `http://code.google.com/p/protobuf/`, Abruf: 20.11.2013

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

[HESM10] Heinig, A. ; Engel, M. ; Schmoll, F. ; Marwedel, P.: Improving transient memory fault resilience of an H.264 decoder. In: *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*, 2010, S. 121–130

[HKS+13] Heinig, A. ; Korb, I. ; Schmoll, F. ; Marwedel, P. ; Engel, M.: Fast and Low-Cost Instruction-Aware Fault Injection. In: *Proc. of SOBRES 2013*, 2013

[HR06] Högl, H. ; Rath, D.: Open On-Chip Debugger – OpenOCD –. (2006)

[HSR95] Han, S. ; Shin, K. G. ; Rosenberg, H. A.: DOCTOR: an integrated software fault injection environment for distributed real-time systems. In: *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, 1995, S. 204–213

[HTI97] Hsueh, M.-C. ; Tsai, T. K. ; Iyer, R. K.: Fault injection techniques and tools. In: *Computer* 30 (1997), Nr. 4, S. 75–82. – ISSN 0018–9162

[JAR+94] Jenn, E. ; Arlat, J. ; Rimen, M. ; Ohlsson, J. ; Karlsson, J.: Fault injection into VHDL models: the MEFISTO tool. In: *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, 1994, S. 66–75

[Kit13] Kitware: *CMake - Cross Platform Make*. Version: 2013. `http://www.cmake.org/`, Abruf: 05.12.2013

[KK07] Koren, I. ; Krishna, C. M.: *Fault-Tolerant Systems*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007. – ISBN 0120885255, 9780080492681

[lin13] linux.die.net: *gettimeofday(2) - Linux man page*. Version: 2013. `http://linux.die.net/man/2/gettimeofday`, Abruf: 02.12.2013

[man13] manned.org: *READPROC(3) - Linux Programmer's Manual*. Version: 2013. `http://manned.org/readproc.3`, Abruf: 02.12.2013

[MEEM12] Mohammadi, A. ; Ebrahimi, M. ; Ejlali, A. ; Miremadi, S. G.: SCFIT: A FPGA-based fault injection technique for SEU fault model. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012. – ISSN 1530–1591, S. 586–589

[NX06] Narayanan, V. ; Xie, Y.: Reliability concerns in embedded system designs. In: *Computer* 39 (2006), Nr. 1, S. 118–120. – ISSN 0018–9162

[OHEB13] Oliver, S. ; Harboe, O. ; Ellis, D. ; Brownell, D.: *Open On-Chip Debugger: OpenOCD User's Guide for release 0.7.0*, Mai 2013

[pan11] pandaboard.org: *OMAP4460 Pandaboard ES System Reference Manual*. Rev 0.1, Sept 2011

[PBKL95] Plank, J. S. ; Beck, M. ; Kingsley, G. ; Li, K.: Libckpt: Transparent Checkpointing under Unix. In: *Usenix Winter Technical Conference*, 1995, S. 213–223

[PGLOGVE07] Portela-García, M. ; López-Ongil, C. ; Garcia-Valderas, M. ; Entrena, L.: A Rapid Fault Injection Approach for Measuring SEU Sensitivity in Complex Processors. In: *On-Line Testing Symposium, 2007. IOLTS 07. 13th IEEE International*, 2007, S. 101–106

[PGLOVE11] Portela-García, M. ; López-Ongil, C. ; Valderas, M. G. ; Entrena, L.: Fault Injection in Modern Microprocessors Using On-Chip Debugging Infrastructures. In: *Dependable and Secure Computing, IEEE Transactions on* 8 (2011), Nr. 2, S. 308–314. – ISSN 1545–5971

[RR99] Rebaudengo, M. ; Reorda, M. S.: Evaluating the fault tolerance capabilities of embedded systems via BDM. In: *VLSI Test Symposium, 1999. Proceedings. 17th IEEE*, 1999. – ISSN 1093–0167, S. 452–457

[Sam13] Samsung: *GALAXY S4 LTE - Technische Daten*. Version: 2013. `http://www.samsung.com/de/consumer/mobile-device/mobilephones/smartphones/GT-I9505ZKADBT-spec`, Abruf: 08.12.2013

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

[SHK+12] Schirmeier, H. ; Hoffmann, M. ; Kapitza, R. ; Lohmann, D. ; Spinczyk, O.: FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In: Mühl, G. (Hrsg.) ; Richling, J. (Hrsg.) ; Herkersdorf, A. (Hrsg.): *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, German Society of Informatics, 2012, S. 201–210

[SLU05] Spinczyk, O. ; Lohmann, D. ; Urban, M.: Advances in AOP with AspectC++. In: *New Trends in Software Methodologies, Tools and Techniques (SoMeT'05)* (2005), Nr. 129, S. 33–53

[SZR03] Santos, L. ; Zenha Rela, M.: Constraints on the Use of Boundary-Scan for Fault Injection. In: Lemos, R. (Hrsg.) ; Weber, T. S. (Hrsg.) ; Camargo, J. B. (Hrsg.): *Dependable Computing* Bd. 2847. Springer Berlin Heidelberg, 2003, Kapitel Lecture Notes in Computer Science, S. 39–55

[Tan07] Tanenbaum, A. S.: *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA : Prentice Hall Press, 2007. – ISBN 9780136006633

[Tex12] Texas Instruments: *Technical Reference Manual - OMAP4460 Multimedia Device - Silicon Revision 1.x*, 2012. – Version S

[Tin13] TinCan Tools: *TinCan Tools - Flyswatter2*. Version: 2013. `http://www.tincantools.com/JTAG/Flyswatter2.html`, Abruf: 23.11.2013

[Wik13] Wikipedia: *Wikipedia - Galaxy Nexus*. Version: 2013. `http://de.wikipedia.org/wiki/Galaxy_Nexus`, Abruf: 08.12.2013

[WZF13] Walters, J. P. ; Zick, K. M. ; French, M.: A practical characterization of a NASA SpaceCube application through fault emulation and laser testing. In: *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, 2013. – ISSN 1530–0889, S. 1–8

[YRLG03] Yuste, P. ; Ruiz, J. C. ; Lemus, L. ; Gil, P.: Non-intrusive Software-Implemented Fault Injection in Embedded Systems. In: Lemos, R. (Hrsg.) ; Weber, T. S. (Hrsg.) ; Camargo, J. B. (Hrsg.): *Dependable Computing* Bd. 2847. Springer Berlin Heidelberg, 2003, Kapitel Lecture Notes in Computer Science, S. 23–38

Abbildungsverzeichnis

2.1 Umwandlungsreihenfolge von Faults, Errors und Failures 5
2.2 Schematische Darstellung des Ablaufs einer Fehlerinjektionskampagne . . 7
2.3 Kampagnen-Verwaltung durch ein Fehlerinjektions-Framwork 8
2.4 Active Probes und Inserted Sockets 13
2.5 Zusammenfassung von FI-Experimenten in Äquivalenzklassen 17
2.6 Schematische Darstellung der Fail*-Architektur 18
2.7 Abstrakte Darstellung der Fail*-Komponenten 20
2.8 Beispielhafter Fail*-Experimentcode 21
2.9 Darstellung der Fail* Datenbank-Struktur 24

3.1 Aufteilung von Fehlerinjektion in 5 Phasen 27
3.2 Schematische Darstellung der Speicherhierarchie des OMAP 4460 32
3.3 Gesamtsystem-Architektur 34
3.4 Beispielcode für den Vergleich von Trace-Navigations-Verfahren 38
3.5 Trace-Navigation mit Single-Stepping, Simple-Hopping und Smart-Hopping 39
3.6 Schematische Darstellung der Sichtbarkeit 41
3.7 Beispielhafte Durchführung von Smart-Hopping 43
3.8 Schematisierung eines indirekter horizontalen Sprungs 44
3.9 Smart-Hopping mit Berücksichtigung von Sprungkosten 45
3.10 Sprunggraphen von einfachem und gewichtetem Smart-Hopping 46
3.11 Beispielhafte Navigationskosten-Verteilung innerhalb einer Äquivalenzklasse 48
3.12 Abbildung einer virtuellen Adresse auf Sections und Pages 50
3.13 MMU-Konfigurierung: Zusammenfassung mehrerer Anforderungen 53
3.14 Sequenzdiagramm zur Erkennung von Speicherzugriffen mithilfe der MMU 55

4.1 Einbindung des Applikations-Codes in den vordefinierten FailPanda-Code 58
4.2 Hauptschleife des OpenOCD-Wrappers in FailPanda 61
4.3 Klassendiagramm zum abstrakten `InjectionPoint` 65

5.1 Verarbeitungszeit von Speicherzugriffen 69
5.2 Vergleich der Navigationskosten ohne und mit Einsatz von Checkpointing 74
5.3 Anzahl vom Checkpointing erstellter Sicherungen 75
5.4 Berechnungszeit von Smart-Hopping 77
5.5 Speicherverbrauch von Smart-Hopping 78
5.6 Qualitativer Vergleich der Kosten von FailPanda und FailGem5 82
5.7 Grafische Darstellung eines beispielhaften Fehlerinjektions-Ergebnisses . . 85

Tabellenverzeichnis

2.1 Vergleich verschiedener Fehlerinjektions-Verfahren 12

5.1 Verarbeitungszeit OCD-Grundfunktionen 68
5.2 Kosten für horizontale und vertikale Sprünge 70
5.3 Eigenschaften der MiBench Benchmarks 72
5.4 Vergleich der Ergebnisse von Simple-Hopping und Smart-Hopping 73
5.5 Kostenoptimierung durch äquivalenzklassengewahre Trace-Navigation . . 76
5.6 Mittlere Verarbeitungszeit der Experimentphasen einer Beispielkampagne 80
5.7 Absolute Fehlerzahlen einer beispielhaften Fehlerinjektions-Kampagne . . 84