technische universität
dortmund

Masterarbeit

Auswahlstrategien für Zustandsraumtransformationen zur Beschleunigung von Fehlerinjektionsexperimenten

Marcel Sellung
21. Januar 2021

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

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

Zusammenfassung

Die steigende Anzahl durch kosmische Strahlung verursachter *Soft-Errors* vergrößert die Notwendigkeit von Untersuchungen der Zuverlässigkeit von Hard- und Softwaresystemen. Eine geeignete Methodik dafür ist Fehlerinjektion, die durch Werkzeuge wie das am Lehrstuhl entwickelte FAIL* [14], angewendet werden kann. Hierbei wird für jeden Fehlerinjektionspunkt eine Simulation durchgeführt und so der Fehlerraum vollständig erkundet. Dadurch lässt sich die Fehlertoleranz der verwendeten Computersysteme analysieren und bewerten. Die große Anzahl durchzuführender Experimente in Verbindung mit der relativ langsamen Geschwindigkeit eines Simulators, bedingen typischerweise sehr lange Laufzeiten. Der in einer Vorarbeit entwickelte Ansatz Programmabschnitte durch *Skipsections* [19] zu überspringen, zeigt, abhängig von der verwendeten Unterteilung, das Potenzial Fehlerinjektionskampagnen deutlich zu beschleunigen. Dabei wurde noch nicht untersucht, wie diese Unterteilung aussehen muss, um möglichst gute Ergebnisse zu erzielen, sodass Kampagnen teilweise auch verlangsamt wurden [19].

In dieser Masterarbeit soll daher systematisch der Entwurfsraum möglicher Auswahlstrategien für *Skipsections* untersucht werden. Zum Vergleichen und zur vorläufigen Bewertung, soll eine Möglichkeit geschaffen werden, verschiedene Strategien ohne die Durchführung echter Fehlerinjektionskampagnen zu bewerten. Neben einem geeignet parametrisierten Kostenmodell soll dies nur auf Grundlage der *Skipsections* und des Programm-*Traces* geschehen. Die Ergebnisse sollen durch reale Kampagnen an geeignet gewählten Benchmarkprogrammen wie der MiBench- [6] oder PARBOIL-Suite [20] oder Standard-Sortieralgorithmen validiert werden.

Inhaltsverzeichnis

1 Einleitung

Die zunehmende Digitalisierung in allen Bereichen des Lebens bringt neben vielen Vorteilen auch einige Risiken mit, denn die Abhängigkeit unserer Gesellschaft und jedes Einzelnen von Computersystemen nimmt ebenfalls stark zu. Mit SmartHomes oder SmartCities, also der Digitalisierung der eigenen Wohnung und der eigenen Stadt auf kommunaler Ebene, werden weitere wichtige Bereiche des täglichen Lebens von modernen Computern und eingebetteten Systemen durchdrungen, sodass Auswirkungen fehlerhaften Verhaltens häufiger negativen Einfluss auf unser Leben haben. Durch diese Abhängigkeit gewinnt auch die Frage nach Sicherheit und Zuverlässigkeit dieser Systeme eine immer größere Bedeutung. Insbesondere aufgrund der Tatsache, dass Fehler nicht nur Unannehmlichkeiten und monetären Schaden anrichten können, sondern im schlimmsten Fall sogar zum Verlust von Menschenleben führen können.

1.1 Kontext

Zur Vermeidung von Softwarefehlern geben moderne Entwicklungsprozesse eine Vielzahl von Ansätzen vor, die durch Planung, Codegenerierung, Debugging und Testen, deren Auftreten stark verringern und im Idealfall sogar völlig vermeiden sollen. Für Hardwarefehler besteht diese Möglichkeit hingegen nicht.Insbesondere transiente Fehler (einmalig auftretend und danach wieder normal funktionierend), die auch als Soft-Errors bezeichnet werden, können immer und überall auftreten und sind dabei nicht vorhersagbar. Verursacht werden sie in der Regel durch kosmische Strahlung [11]. Durch zunehmend kleinere Strukturbreiten, größere Transistordichte und eine immer geringere Versorgungsspannung [4], verringert sich die notwendige Energie, um ein einzelnes Bit, oder auch mehrere Bits, kippen zu lassen (engl.: *Bit-Flip*), sodass deren Anzahl bei gleichbleibender kosmischer Strahlung steigt [12]. Da das Auftreten nicht vorhersagbar oder zu verhindern ist, muss der Umgang damit abgewogen werden. Der Entwickler muss die Frage beantworten, ob bzw. welche Fehlertoleranzmaßnahmen erforderlich sind. Die vorliegende Arbeit konzentriert sich dabei auf Soft-Errors und in diesem Zusammenhang auftretende Fragen.

In der Praxis können solche Hardwarefehler nämlich sehr teuer werden. Im Februar 2016 verursachte ein Soft-Error den Absturz des Hitomi-Satellits [18], wodurch einen Sachschaden von 286 Mio US$ entstand und eine jahrelange Entwick-

lungszeit zunichte gemacht wurde. In einer besonders strahlungsreichen Region, der südatlantischen Anomalie, wurde durch wiederholte Fehler im „Star-Tracker"-System, die Ausrichtung fehlerhaft ermittelt und eine Kaskade weiterer Fehler verursacht. Durch die fälschlich eingeleiteten Gegenmaßnahmen geriet der Satellit in zu starke Rotation und brach auseinander. Die Beachtung möglicher Soft-Errors während der Entwicklung und ein angemessenes Fehlertoleranzverfahren, hätten dies verhindern können.

Zur Vermeidung solcher katastrophaler Auswirkungen werden in besonders kritischen Systemen häufig entsprechende Gegenmaßnahmen eingesetzt. Dabei unterscheidet man zwischen hardwarebasierter und softwarebasierter Fehlertoleranz. In sensiblen Bereichen wird deshalb Hardware teilweise redundant verbaut. Das ist allerdings nicht nur sehr teuer, sondern es erhöht auch sehr stark den Energieverbrauch, sodass besonders im Bereich der mobilen und eingebetteten Systeme sehr große Nachteile entstehen. Die dann verwendeten Alternativen sind softwareimplementierte Hardwarefehlertoleranzverfahren (SIHFT).

Auch hier werden hierbei Informationen meist durch Redundanz abgesichert, sodass die Laufzeit und der Speicherverbrauch dadurch erhöht werden können. Neben der möglicherweise schlechteren Leistung, besteht dadurch auch die Möglichkeit, dass während der Ausführung zusätzlicher Instruktionen zur Absicherung des Programms, ein Soft-Error auftritt und das Programm im schlimmsten Fall sogar fehleranfälliger wird. Um die Qualität konkreter Maßnahmen abschätzen zu können, ist es wegen der Unvorhersagbarkeit der Soft-Errors kaum möglich, im realen Betrieb zu messen. Stattdessen wird häufig Fehlerinjektion verwendet, um die Zuverlässigkeit und Fehlertoleranz konkreter Programme zu analysieren. Hierbei wird das Programm zu einem festgelegten Zeitpunkt unterbrochen und zur Simulation des Fehlers der Wert eines oder mehrerer Bits gezielt geändert. Anschließend wird das Programm wieder gestartet und das Resultat mit dem erwarteten Ergebnis verglichen. Dabei unterscheidet man zwischen drei Fehlerinjektionsarten, software-, hardware- oder simulatorbasierte, welche in Abschnitt 2.1 genauer betrachtet werden. Als konkretes Fehlerinjektionstool wird in dieser Arbeit FAIL* [14] verwendet, welches im Abschnitt 2.3 genauer beschrieben wird.

1.2 Motivation

In vielen Fällen wäre eine Zuverlässigkeitsuntersuchung von Programmen bereits während der Entwicklung sinnvoll und bei Bedarf auch eine anschließende Verbesserung. Ein mögliches Vorgehen wird als *Fault Tolerance Assessment Cycle* [14] beschrieben. Vielfach wird aber auf Fehlerinjektion zur Untersuchung der Zuverlässigkeit verzichtet. Ein Grund ist der hohe zeitliche Aufwand für Fehlerinjektionskampagnen und die daraus resultierenden hohen Kosten.

Da ein Soft-Error an jeder beliebigen Stelle (Speicher oder CPU) und zu jedem

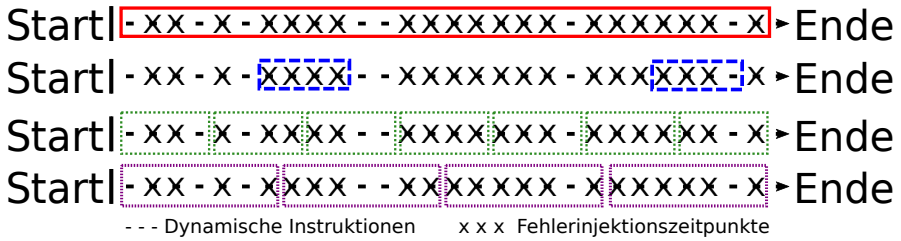

Abbildung 1.1: Verschiedene Möglichkeiten das Programm in überspringbare Abschnitte (Skipsections) einzuteilen

beliebigen Zeitpunkt im Programm auftreten kann, wächst die Anzahl möglicher Fehlerinjektionsexperimente quadratisch mit der Programmgröße. Die Laufzeit jedes einzelnen Experiments wächst dabei ebenfalls mit der Größe des Programms, sodass die Laufzeiten für vollständige Untersuchungen häufig sehr groß werden.

Eine Möglichkeit Fehlerinjektion praktikabler zu machen, wäre es, die Kampagnenlaufzeit zu verkürzen. Eine Methode zu Beschleunigung eines Fehlerinjektionsexperiments ist die Zusammenfassung von Zustandsraumtransformationen, ein Ansatz den Stampa in seiner Masterarbeit [19] beschrieben hat und der im Rahmen dieser Arbeit verbessert werden soll.

Hierbei werden während des Fehlerinjektionsexperiments Teile des Programms nicht ausgeführt, sondern die dadurch resultierenden Veränderungen des Zustands direkt herbeigeführt. Dazu werden für das Zielprogramm während des *Golden Run*, also der einmaligen Ausführung des Programms zum Ermitteln der jeweiligen Injektionspunkte, sogenannte *Skipsections* aufgezeichnet. Hierbei wird für einen Abschnitt aufgezeichnet, welche Register und Speicherbereiche gelesen werden und welche Werte geändert werden. Wird so ein Abschnitt erreicht, kann geprüft werden, ob der aktuelle Zustand in den benutzten Teilen mit der aufgezeichneten Eingabe übereinstimmt. Wenn dies der Fall ist, wird der Zustand, inklusive *Instruction Pointer* und der internen Uhr, geändert, und dadurch die Ausführung ersetzt. Die Entscheidung, wie groß und an welcher Stelle diese überspringbaren Abschnitte sind, beeinflusst dabei sehr stark, ob mit diesem Ansatz ein Zeitgewinn erreicht werden kann. Sind die Abschnitte zu klein, dauert die Unterbrechung und der Kontrollflusswechsel, die Überprüfung der verwendeten Register/Speicherbereiche und die Änderung des Zustands, länger, als die Ausführung der übersprungenen Instruktionen im Simulator und die Experimentlaufzeit verlängert sich. Wählt man die Abschnitte zu groß, können sie nur selten angewendet werden, da zum einen der Zeitpunkt der Fehlerinjektion nicht übersprungen werden kann und entsprechend nicht enthalten sein darf. Zum anderen steigt die Wahrscheinlichkeit, dass der Abschnitt nicht angewendet werden kann, weil sich der Zustand von der Aufzeichnung unterscheidet.

In Abbildung 1.1 wird der Trace eines Programms inklusive alle Zeitpunkte

für die Fehlerinjektion dargestellt, welcher anschließend auf verschiedene Arten in Skipsections unterteilt wurde. In der rot eingezeichneten Variante liegen alle Injektionszeitpunkte in dem vorhandenen Abschnitt, sodass diese Skipsection niemals angewendet werden kann und die Ausführungszeit folglich gar nicht reduziert wird. In der blau eingezeichneten Variante werden große Bereiche des Programms nicht abgedeckt, sodass zumindest für diese Teile keine Beschleunigung möglich ist und der potentielle Laufzeitgewinn nicht vollständig genutzt wird. Die grün eingezeichnete Variante deckt den Trace vollständig ab und außer der Skipsection, in der die Fehlerinjektion liegt, können die anderen Skipsections auch angewendet werden. Die Skipsections sind allerdings sehr kurz, sodass das Anhalten des Simulators, die Überprüfung des Zustands und dessen Änderung, eine längere Laufzeit haben könnten, als die Ausführung der Instruktionen im Simulator. Die lila eingezeichnete Variante stellt eine Unterteilung dar, die am wahrscheinlichsten eine mögliche Laufzeitverbesserung bewirkt.Neben Anzahl und Größe gibt es noch weitere Parameter, die die Güte der gewählten Einteilung in Skipsections bestimmt.

1.3 Zielsetzung

Das Ziel dieser Arbeit gliedert sich in zwei Teile. Es soll der Raum möglicher Strategien zur Unterteilung des Trace in verschiedene Skipsections untersucht und deren Nutzen evaluiert werden. Wie Abschnitt 1.2 gezeigt hat, ist die Unterteilung in konkrete Skipsections maßgeblich entscheidend, wie sehr Zustandsraumtransformationen die Fehlerinjektion beschleunigen können. Die Kriterien nach denen eine Unterteilung stattfindet, können dabei sehr unterschiedlich sein und sollen in Hinblick auf ihre Relevanz für die Verbesserung der Laufzeit untersucht werden. Anhand dieser Untersuchungen soll es in konkreten Fehlerinjektionskampagnen leichter sein, eine möglichst gewinnbringende Unterteilungsstrategie auszuwählen.

Ein anderes Ziel ist die Entwicklung eines Evaluators, der konkrete Unterteilung in Skipsections bezüglich ihres Nutzens abschätzen kann. So können verschiedene Unterteilungen miteinander verglichen werden, ohne die Auswirkungen durch eine reale Fehlerinjektionskampagne messen zu müssen. Das erleichtert die Entwicklung neuer Strategien, kann aber auch den Einsatz bestehender Strategien besser vorbereiten. So kann für verschiedene Parameter, z. B. die Mindestlänge, verglichen werden, wann die Kampagnenlaufzeit vorraussichtlich am stärksten verringert wird.

2 Grundlagen und verwandte Arbeiten

In diesem Kapitel werden die Grundlagen der Fehlerinjektion und deren Ablauf, insbesondere bei Verwendung eines Simulators vorgestellt. Anschließend werden verschiedene Möglichkeiten zur Beschleunigung erläutert, die in anderen Untersuchungen betrachtet und entwickelt wurden. Abschließend wird das im Weiteren verwendete Tool FAIL*, mit Schwerpunkt auf die für das Verständnis dieser Arbeit besonders wichtigen Teilen vorgestellt und der Ansatz der Beschleunigungen durch Zustandsraumtransformationen im Detail präsentiert.

2.1 Fehlerinjektionsarten

Da Soft-Errors im Alltag nicht vorhersehbar sind und auch nur sehr schwer zu detektieren, müssen diese bei der Fehlerinjektion künstlich erzeugt werden. Dafür gibt es im Wesentlichen drei verschiedene Arten, die unterschiedliche Vor- und Nachteile mit sich bringen. Alle haben aber gemein, dass die Anzahl der Fehler so stark erhöht wird, dass eine sinnvolle Beurteilung überhaupt erst möglich wird.

Hardwareimplementierte Fehlerinjektion Hier wird ein Fehler auf physikalischer Ebene durch die direkte Umgebung injiziert. Dazu werde Schwerionenstrahlung oder elektromagnetische Interferenzen erzeugt, die die natürliche Fehlerquelle sehr gut nachahmen und so den Transistorzustand ändern können. Ebenso kann das Herbeiführen von Spannungsabfällen durch Stromversorgungsstörungen einen Soft-Error imitieren [21]. Problematisch dabei ist die Tatsache, dass so eine Fehlerinjektion schwer zu kontrollieren ist und die Wiederholbarkeit von Ergebnissen sehr gering ist, da analog zu natürlich vorkommenden Soft-Errors die konkreten physikalischen Änderungen nur schwer bestimmbar sind und nur anhand des Ergebnisses detektiert werden. Zudem sind diese Verfahren sehr teuer, insbesondere die Bestrahlung.

Ein Ansatz, der diese Nachteile vermindern soll, ist die Fehlerinjektion über Verbindungspins der CPU [9]. Dies reduziert aber Geschwindigkeit der Fehlerinjektion und die möglichen Stellen, an denen Fehler injiziert werden können [14].

Softwareimplementierte Fehlerinjektion Der zentrale Ansatz bei softwareimplementierter Fehlerinjektion (SWIFI) ist, dass durch Hardwarefehler resultierende Ergebnis, auf Softwareebene konsistent zu reproduzieren [21]. Dazu können Software oder Daten vor der Ausführung manipuliert werden, oder während der Laufzeit Debugging Möglichkeiten genutzt werden, z. B. Exceptions. Insbesondere im ersten Fall wird die Erreichbarkeit von Stellen zur Fehlerinjektion stark eingeschränkt, während im zweiten Fall ein verändertes Laufzeitverhalten zur Fehlerinjektion genutzt wird, was die Aussagekraft der Ergebnisse reduziert [14]. Größter Vorteil bei SWIFI sind dabei die deutlich geringeren Kosten.

Simulatorbasierte Fehlerinjektion Die Ziele bei simulatorbasierter Fehlerinjektion sind insbesondere eine Reproduzierbarkeit der Ergebnisse und eine bessere Kontrolle der Fehlerinjektionskampagnen. Die Fehler werden, statt in realer Hardware, in simulierter Hardware injiziert. Dies bietet in Abhängigkeit des konkret gewählten Simulators die Möglichkeit (fast) überall Fehler zu injizieren, was zumindest in kontrollierter Form sonst nicht möglich ist. Die Auswirkungen können dabei sehr genau ermittelt und die Fehlerinjektion detailliert gesteuert werden. Durch deterministische Simulatoren wird auch die Reproduzierbarkeit gewährleistet, was ein enormer Vorteil bei der Beurteilung der Ergebnisse ist. Allerdings besteht bei diesem Ansatz eine große Abhängigkeit vom gewählten Simulator, der die Hardware möglichst detailgetreu simulieren und einen vielseitigen Zugriff ermöglichen muss. Je niedriger das Abstraktionslevel des Simulators dabei ist, um so langsamer ist dabei die Ausführung, was der hauptsächliche Nachteil bei der Nutzung eines Simulators ist [21].

Da für wissenschaftliches Arbeiten die Reproduzierbarkeit der Ergebnisse gegeben seien sollte und eine hohe Kontrolle und große Flexibilität wünschenswert sind, eignet sich nur die simulatorbasierte Fehlerinjektion. Die vergleichsweise langsame Ausführung muss dabei in Kauf genommen werden, oder durch andere Maßnahmen vermindert, um die Handhabung und Nutzbarkeit zu verbessern. Ein sinnvolle Möglichkeit dazu, soll durch die Ergebnisse dieser Arbeit vorgestellt werden, die sich im Weiteren auf simulatorbasierte Fehlerinjektion bezieht, die mit dem in Abschnitt 2.3 beschriebenen Tool FAIL* durchgeführt wird.

2.2 Ablauf der Fehlerinjektion

Die simulatorbasierte Fehlerinjektion bietet, dank des großen Maßes an Kontrolle, vielfältige Möglichkeiten Fehler sehr gezielt zu injizieren. Der normale Ablauf wird dabei in Abbildung 2.1 dargestellt.

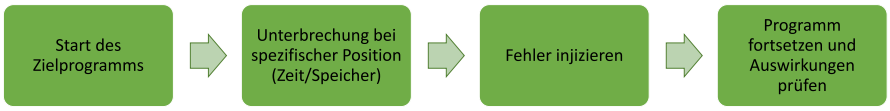

Abbildung 2.1: Typischer Ablauf der Fehlerinjektion: Das Zielprogramm läuft bis zu einem vorher bestimmten Zeitpunkt und injiziert dort gezielt einen Fehler. Anschließend werden die Auswirkungen überprüft.

Das Zielprogramm wird im Simulator gestartet und läuft bis zum Zeitpunkt der Fehlerinjektion normal. Zur Unterbrechung muss ein geeigneter Mechanismus integriert sein, der das Programm, analog zu einem *Breakpoint*, zum richtigen Zeitpunkt unterbrechen kann. Die Fehlerinjektion erfolgt durch lesen, verändern und schreiben von Daten (Single-Bit oder Multi-Bit), dabei hängen die Möglichkeiten, wo überall Fehler injiziert werden können, vom jeweiligen Simulator ab. Typisch ist Fehlerinjektion im RAM oder der CPU, aber auch andere Bereiche, z. B. verschiedene Caches, sind realisierbar. Anschließend wird das Programm fortgesetzt und die möglichen Auswirkungen des injizierten Fehlers geprüft. Typische Ergebnisse sind bspw. folgende:

- *OK:* Der Fehler hatte keine Auswirkung auf das Programmverhalten.
- *Detection:* Der Fehler wurde durch ein entsprechendes Verfahren erkannt und der Benutzer oder das Programm können reagieren.
- *Recovery:* Der Fehler wurde erkannt und behoben. Je nach Tool keine Unterscheidung zu *OK*.
- *Trap:* Das Programm musste wegen einer unerlaubten Operation vorzeitig beendet werden (Speicherverletzung / Division durch Null).
- *Timeout:* Das Programm terminiert nicht bzw. nicht rechtzeitig (Endlosschleife).
- *Silent Data Corruption (SDC):* Das Programm scheint normal zu funktionieren und terminiert, die erzeugten Daten sind aber fehlerhaft.

Je nach Anwendungsgebiet sind dabei nicht immer alle Fehlerarten gleich relevant. Ein SDC bei der Videocodierung, der einen Pixelfehler verursacht, vermindert die Qualität des Programms nur sehr gering und ist in der Regel daher vertretbar. Bei anderen Anwendungen wäre SDC unter Umständen besonders schlimm, da dieser oftmals unbemerkt ist und man deshalb nicht reagieren kann. Ein Beispiel wäre die Steuerung eines Röntgengeräts, ein Absturz wäre durch einen Angestellten durch Neustart zu beheben. Eine durch SDC verursachte deutlich höhere Strahlung hingegen könnte gravierende gesundheitliche Schäden für den Patienten bedeuten.

2.3 FAIL*

Die im Rahmen der vorliegenden Arbeit entwickelten Strategien und Verfahren für Zustandsraumtransformationen wurden, ebenso wie der in Abschnitt 2.5 beschriebene und zugrunde liegende Ansatz mit Hilfe von FAIL*[1] implementiert. Daher sind, zum vollständigen Verständnis, Kenntnisse über Aufbau und Funktionsweise von FAIL* erforderlich. Das Tool wurde im Rahmen der Dissertation [15] von Horst Schirmeier entwickelt und seine grundlegende Architektur und Implementierung in [13] beschrieben.

FAIL* bietet mit seinen Plugins vielfältige Möglichkeiten zur Fehlerinjektion und Analyse der Ergebnisse und zusätzlich eine große Flexibilität bezüglich der Backends und Zielprogramme. Es können Fehler im Speicher und in der CPU injiziert werden, vorwiegend Single-Bit-Fehler, aber durch Invertieren eines kompletten Bytes, sind auch Multi-Bit-Fehler möglich, auch wenn nur für ein ganzes, zusammenhängendes Byte. Damit können die häufigsten Fehlermodelle realisiert und der vollständige Fehlerraum untersucht werden. Teile der in Abschnitt 2.4 beschriebenen Verfahren werden durch das integrierte Tool *Prune-Trace* implementiert.

2.3.1 Architektur

Die Grundideen hinter der Architektur von FAIL* sind die Flexibilität bezüglich des *Backends* durch Modularität und eine Verteilung der Experimente für parallele Ausführung. Außerdem ein *API-Design* mit einem ausreichenden Maß an Abstraktion, für die Portierbarkeit von Experimenten und einer einfachen Handhabung bei der Implementierung [13]. Eine Übersicht über den Aufbau wird in Abbildung 2.2 dargestellt.

Das Design von FAIL* lässt sich dabei in zwei Schichten unterteilen, den *Plumbing Layer* und den *Assessment-Cycle Layer*, welcher im Prinzip die Abstraktion des ersteren ist [15]. Der *Plumbing Layer* wiederum abstrahiert dabei selbst von der unterliegenden Hardware durch den *Execution-Environment Abstraction (EEA) Layer*, sodass hier verschiedene *Backends* möglich sind, auch über *Test-Access Ports (TAP)* gesteuerte Hardware. Zur möglichen Parallelisierung der Ausführungen, stellt er die Client/Server-Infrastruktur zur Verfügung. Der *JobServer* kontrolliert die Kampagne über eine Warteschlange, in der die durchzuführenden Experimente, als Parameter für die Fehlerinjektion, gespeichert sind. Diese werden dann an beliebig viele Clients verteilt, die die Experimente parallel durchführen und die Ergebnisse anschließend zurückmelden. Die Clients führen die jeweiligen Einzelexperimente durch und bestehen im Wesentlichen nur aus

[1] Fault Injection Leveraged; das * steht für das austauschbare *Backend*, also FailBochs für eine FAIL-Instanz, die den *Bochs x86* Simulator als *Backend* verwendet.

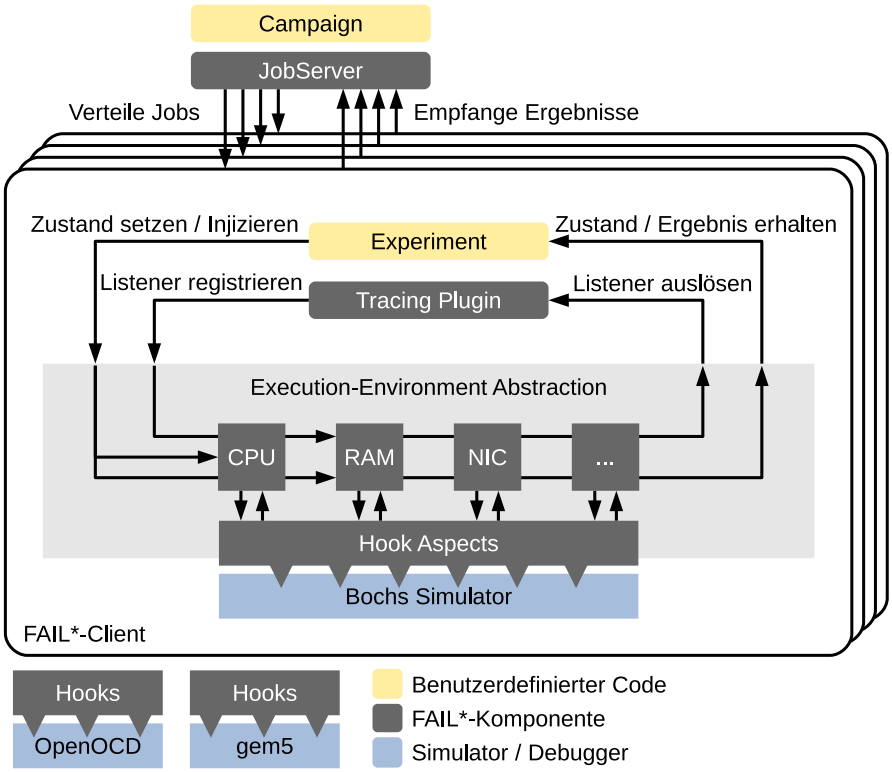

Abbildung 2.2: Architektur des Plumbing Layer von FAIL*. Aus [19] nach [15], Abb. 4.1

.

dem *Backend*, das um einige Komponenten zur Steuerung erweitert wurde. Dabei sind der Simulator und das Experiment parallel laufende Koroutinen, die sich ereignisgesteuert abwechseln. Das heißt, der Experimentkontrollfluss bereitet den Simulator und weitere Plugins vor und gibt die Kontrolle dann explizit an den Simulator ab. Die Vorbereitung erfolgt über eine backendspezifische Variante des `SimulatorControllers`, der über die relevanten Infos des konkret verwendeten Simulators verfügt und über weitere Kontroller auf den kompletten Zustand, den Speicher, die CPU und einzelne Register zugreifen kann. So können z. B. Checkpoints erstellt werden, oder Fehler im Speicher und in den Registern injiziert werden. Das Zurückwechseln zum Experiment bzw. Plugins erfolgt über `Listeners`, die beim `ListenerManager` registriert werden und auf bestimmte Ereignisse im Zielprogramm reagieren.

Listing 2.1: Steuerung des Simulators

```
1 BPSingleListener l_start_symbol(start_address);
2 simulator.addListener(&l_start_symbol);
3 simulator.resume();
4 simulator.save(state_file);
```

Das Zusammenspiel wird exemplarisch in Programm-Ausschnitt 2.1 veranschaulicht. Zur Vorbereitung wird ein `Listener` durch den Client erst initialisiert und anschließend beim Simulator registriert (2). Danach gibt er die Kontrolle explizit wieder an den Simulator ab (3). Erreicht dieser nun den Beginn des Zielprogramms, reagiert der `Listener` und der Kontrollfluss springt zurück zum Client und führt die nächste Zeile aus, in diesem Fall speichert er den Zustand des Simulators (4).

Wichtig beim Konzept der *Listener* ist, dass alle über einen Zähler verfügen, der durch die Basisklasse `BaseListener` zur Verfügung gestellt und bei jeder Aktivierung dekrementiert wird. Ein Kontrollflusswechsel findet nur statt, wenn das entsprechende Ereignis eintritt und der Zähler den Wert 0 erreicht. Die wichtigsten in der Implementierung verwendeten `Listener` sind folgende:

BPSingleListener: Funktioniert analog zu einem *Breakpoint* und wartet auf eine bestimmte statische Instruktion. Wird als Parameter `ANY_ADDR` gewählt, wird der `Listener` bei jeder Instruktion aktiviert und kann dadurch, in Verbindung mit dem Zähler, auf bestimmte dynamische Instruktionen warten und z. B. die Fehlerinjektion durchführen.

BPRangeListener: Funktioniert analog zum `BPSingleListener`, nur kann die statische Adresse in einem definierten Intervall liegen.

MemAccessListener: Wartet auf den Zugriff bestimmter Speicherbereiche, die Unterscheidung zwischen Lese- und Schreibzugriffen ist möglich, aber nicht zwingend.

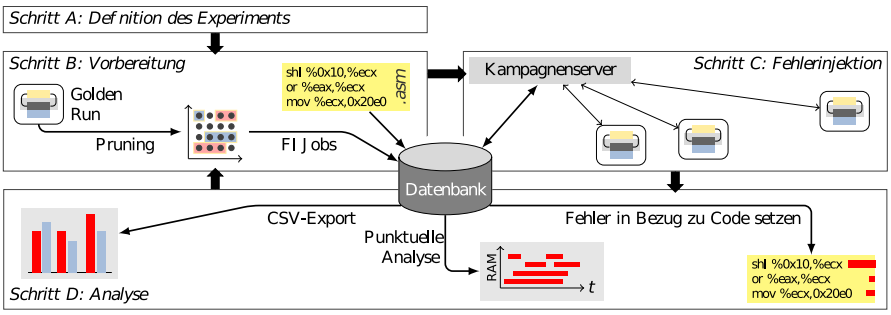

Abbildung 2.3: Fehlertoleranz nach dem Fault Tolerance Assessment Cycle (Aus [19] nach [15], Abb. 4.10). Der Entwickler versucht in einer oder mehreren Iterationen die Widerstandsfähigkeit des Systems gegenüber Fehlern zu verbessern.

TimerListener: Überwacht die interne Uhr des *Backend* und löst bei einem definierten Wert aus.

2.3.2 Fehlerinjektion mit FAIL*

Die Untersuchung und Verbesserung der Widerstandsfähigkeit eines Programms gegenüber (transienter) Fehler erfolgt im Rahmen des *Fault Tolerance Assessment Cycles*, welche in Abbildung 2.3 zusammengefasst ist und einfach oder mehrfach durchlaufen wird [14]. Die Fehlerinjektion wird dabei durch mehrere Tools in der Vor- und Nachbereitung unterstützt, um gezielte Ergebnisse, für etwaige Verbesserungen, schnellstmöglich zu erhalten.

Dabei unterteilt sich der Ablauf in vier verschiedene Phasen:

A. Definition des Experiments Der Anwender definiert ein Experiment und legt damit die Parameter für die Fehlerinjektion fest. Das bestehende Experiment `generic-experiment` ermöglicht die Fehlerinjektion für verschiedene Zielprogramme, kann aber angepasst werden, oder eine komplett eigene Definition erstellt werden. Insbesondere über das zugrunde liegende Fehlermodell wird an dieser Stelle entschieden, also über die Art des Fehlers (transient / permanent), den Umfang des Fehlers (Single-Bit / Multi-Bit = Byte), den Ort des Fehlers (Register / Speicher / Instruktions OP-Code usw.) und das Auftreten des Fehlers (gleichverteilt / beim Lesen / beim Schreiben). Die API bietet hier viele Möglichkeiten, allerdings müssen die Details im Programmcode implementiert werden. Die genannten Variationen wurden auch bereits mit FAIL* implementiert (Vergleich [14], Table 1), weitere Modelle, z. B. Injektion auf *Caches*, sind aber noch möglich. Dabei kann die Auswahl des Fehlermodells sehr entscheidend bei der Bewertung konkreter Fehlertoleranzverfahren seien. Je nach Fehlermodell kann ein konkretes Verfahren

die Fehlertoleranz sowohl verbessern, als auch verschlechtern [16].

B. Vorbereitung Im nächsten Schritt werden die zur vollständigen Abdeckung des definierten Fehlerraums notwendigen Injektionen ermittelt und eine entsprechende Liste der Fehlerinjektionsparameter für den *JobServer* erstellt. Hierzu wird im *Golden Run* ein Experiment ohne Fehlerinjektion durchgeführt und die Instruktionen und Speicherzugriffe im Trace gespeichert. Daraus kann der Fehlerraum abgeleitet und anschließend die in Unterabschnitt 2.4.1 erwähnten Maßnahmen zur Reduktion durchgeführt werden. Die resultierenden Ergebnisse werden in die Datenbank geladen. Dazu werden die Tools `prune-trace` und `import-trace` verwendet.

C. Fehlerinjektion Bei der Durchführung der tatsächlichen Fehlerinjektion generiert der Kampagnenserver *Jobs* für die einzelnen Fail Client Instanzen, die auf beliebigen Systemen laufen können, da die Kommunikation mittels TCP erfolgt. Die Ergebnisse werden anschließend zurück gemeldet und in die Datenbank importiert. Eine genaue Zuordnung der Resultate zum konkreten Fehler (Ort und Zeitpunkt) bleibt dabei bestehen.

D. Analyse Die in der Datenbank vorliegenden Ergebnisse können auf unterschiedliche Art und Weise durch den Benutzer analysiert werden. Eine genaue Zuordnung, zu Quelltextabschnitten und Datenstrukturen ermöglicht die Identifizierung kritischer Abschnitte oder die Evaluierung konkreter Toleranzmechanismen. Zudem sind verschiedene Möglichkeiten zur Visualisierung vorhanden, die eine bessere Verwendung der Ergebnisse ermöglichen.

2.4 Beschleunigung von Fehlerinjektionskampagnen

Wie bereits in Abschnitt 1.2 aufgezeigt, ist das größte Hemmnis bei der Durchführung von Fehlerinjektion, insbesondere bei der Verwendung eines Simulators, der erhebliche Zeitaufwand. Ein Kernanliegen bei der Weiterentwicklung von Fehlerinjektionstools muss entsprechend die Beschleunigung der Fehlerinjektionskampagnen sein. Ohne sinnvolle Verfahren ist Fehlerinjektion kaum bzw. gar nicht anwendbar.

Die Laufzeit der Fehlerinjektion t_{FI} ist das Produkt der durchschnittlichen Laufzeit eines Experiments t_{Exp} multipliziert mit der Anzahl der durchzuführenden Experimente n_{Exp} ($t_{FI} = n_{Exp} \cdot t_{Exp}$). Die durchschnittliche Experimentlaufzeit, also die Anzahl der benötigten CPU-Zyklen, ist vor allem von der Anzahl der Instruktionen abhängig und die Anzahl der Experimente vom Fehlerraum, welcher in Abbildung 2.4 dargestellt wird. Da der Fehlerraum alle möglichen Kombinationen aus aktuellem CPU-Zyklus und vorhandenen Speicherbits n_{Mem} enthält, ist

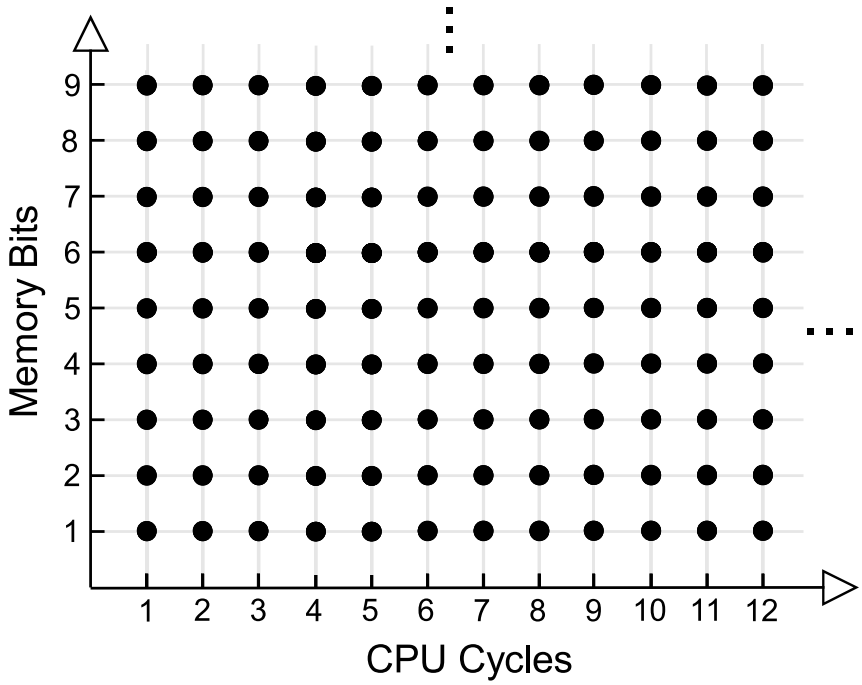

Abbildung 2.4: Fehlerraum Single-Bit Flip Fehlerinjektion. Jeder Punkt stellt eine potentielle Koordinate zur Fehlerinjektion dar (Aus [15], Abb. 1.1).

dieser ebenfalls von der Anzahl der Instruktionen abhängig ($n_{Exp} = t_{Exp} \cdot n_{Mem}$), wobei zur Vereinfachung eine Instruktion mit einem CPU-Zyklus gleichgesetzt wurde, also: $t_{FI} = n_{Mem} \cdot t_{Exp} \cdot t_{Exp} = n_{Mem} \cdot (t_{Exp})^2$. Dies ergibt eine quadratische von der Anzahl der Instruktionen abhängige Laufzeit. Durch immer größere Programme können sowohl n_{Mem}, als auch t_{Exp} sehr große Werte annehmen, sodass die Laufzeit durchaus eher in Tagen als in Stunden gemessen werden kann und eine tatsächliche Durchführung deutlich erschwert.

Um eine deutlich kleinere Laufzeit zu erreichen, können beide Faktoren unabhängig von einander reduziert werden. Die jeweiligen Möglichkeiten werden dabei in Unterabschnitt 2.4.1 und Unterabschnitt 2.4.2 betrachtet. Zusätzlich lässt sich die Laufzeit auch durch bessere Hardware und bei simulatorbasierter Fehlerinjektion, durch parallele Ausführung der jeweiligen Experimente, beschleunigen.

2.4.1 Fehlerraumreduktion

Bei der Fehlerraumreduktion gibt es zwei verschieden Arten. Bei der ersten Art bleiben die Ergebnisse genau die selben, bei der Zweiten wird das Gesamtergebnis nur approximiert.

A. Ohne Genauigkeitsverlust

Es werden nur Experimente nicht ausgeführt, deren Ergebnis man auf andere Art ermitteln kann.

- Injektion nur in tatsächlich verwendetem Speicher:

  Speicherbereiche auf die gar nicht zugegriffen wird, können von der Fehlerinjektion ausgenommen werden. Wegen des Determinus bei der Verwendung eines Simulators, können die relevanten Bereiche im *Golden Run* identifiziert werden.

- Def/use Pruning:

  Beim *Def/use Pruning* werden alle Zeitpunkte zwischen zwei Speicherzugriffen zu sogenannten Äquivalenzklassen zusammengefasst [17]. Dies basiert auf der grundlegenden Tatsache, dass ein Programmablauf erst dann durch einen Hardwarefehler beeinflusst werden kann, wenn auf den entsprechenden Speicherbereich auch wirklich durch eine Instruktion zugegriffen wird.Alle Experimente zwischen zwei Speicherzugriffen haben also das selbe Ergebnis, sodass nur eine tatsächliche Durchführung notwendig ist, um das Ergebnis für alle zu ermitteln.

B. Mit Genauigkeitsverlust - Stichproben

Bei diesem, auch als *Sampling* bezeichnten, Verfahren wird eine gewisse Anzahl aller möglichen Experimente zufällig zur Durchführung ausgewählt [7]. Statistisch gesehen entspricht die Ergebnisverteilung der des vollständigen Fehlerraums annähernd, wenn die Anzahl der Stichproben groß genug ist. Im Einzelfall kann es, auch wenn nur mit geringer Wahrscheinlichkeit, trotzdem zu sehr starken Abweichungen kommen.

2.4.2 Beschleunigung der Einzelexperimente

Durch die hohe Kontrollierbarkeit eines Simulators ergeben sich einige Möglichkeiten, die die grundsätzlich langsamere Ausführung zum Teil zu kompensieren. Die Experimentlaufzeit wird hauptsächlich durch die Hardware, den Simulator und die Anzahl der dynamischen Instruktionen bestimmt. Während die ersten beiden in der Regel festgelegt und nur mit hohem Aufwand und großen Kosten zu verbessern sind, gibt es leicht zu verwendende Ansätze, die die Anzahl der dynamischen Instruktionen verringern.

Dabei unterteilt man, wie in Abbildung 2.5 veranschaulicht, eine Fehlerinjektion in zwei unterschiedliche Phasen. Zunächst muss das Programm möglichst schnell bis zum Zeitpunkt der Fehlerinjektion ausgeführt werden, ohne das der Ablauf genauer betrachtet wird. Diese erste Phase wird als *Fast-Forwarding* bezeichnet. Hier sind noch alle Experimente identisch, da die Fehlerinjektion noch keine Änderung des Programmverhaltens bewirkt haben kann. Anschließend wird

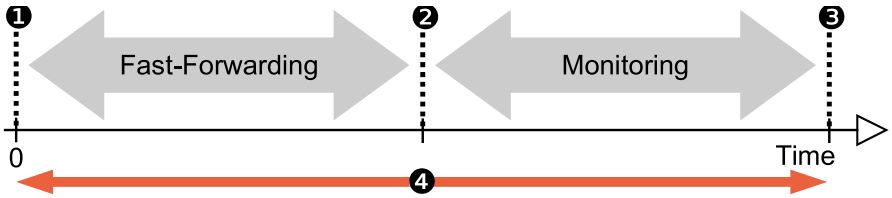

Abbildung 2.5: Unterteilung der Ausführung des Zielprogramms in zwei Bereiche, jeweils vor und nach der Fehlerinjektion (2). Aus [15], Abb. 3.4

die Fehlerinjektion (2) durchgeführt und die zweite Phase erreicht, das sogenannte *Monitoring*. In dieser Phase kann sich der Ablauf grundlegend von anderen Experimenten unterscheiden und die Beobachtung dieses veränderten Verhaltens ist Grundgedanke bei der Fehlerinjektion.

Da die Beschleunigung der jeweiligen Einzelexperimente keine Veränderung der Ergebnisse bewirkt, kann diese unabhängig von der Reduktion des Fehlerraums eingesetzt werden. Allerdings besitzen die Verfahren einen gewissen Overhead, z. B. Speicherverbrauch, oder Ausführungszeit zur Vorbereitung, sodass sich die Anwendung bestimmter Methoden nicht immer lohnt, zumal der Nutzen, durch die Verringerung der Anzahl der Experimente, ebenfalls geringer wird. Sei t_{Prep} die Anzahl der CPU-Zyklen, die die Vorbereitung des Verfahrens braucht, t_{Gain} der durchschnittliche Gewinn je Experiment und n_{count} die Anzahl der durchgeführten Experimente, so sollte für einen sinnvollen Einsatz des Verfahrens gelten: $t_{Prep} < t_{Gain} \cdot n_{count}$. Durch eine Reduktion des Fehlerraums wird n_{count} vermindert, sodass das Verhältnis zwischen Kosten und Nutzen schlechter wird.

2.4.2.1 Vor der Fehlerinjektion (Checkpoints)

Eine vielfach angewendete Methode zur Beschleunigung in der *Fast-Forwarding* sind Checkpoints. Hierzu werden, unabhängig von den jeweiligen Experimenten, Zwischenzustände als Checkpoints gespeichert, üblicherweise gleichmäßig verteilt [1]. Abbildung 2.6 zeigt die konkrete Beschleunigung bei einer Fehlerinjektion. Statt von Beginn des Programms (1) startet dieses Experiment ab Checkpoint C_3 und überspringt alle vorherigen dynamischen Instruktionen.

Die Erstellung und Speicherung dieser Checkpoints kostet Laufzeit und Speicherplatz, sodass, wie in Unterabschnitt 2.4.2 der Nutzen, also die Anzahl der jeweils übersprungenen Instruktionen, groß genug sein muss, diese Kosten aufzuwiegen. Da die Fehlerinjektion eine Veränderung gegenüber den in den Checkpoints gespeicherten Zuständen bedeutet, kann diese Methode in der *Monitoring* Phase nicht mehr angewendet werden.

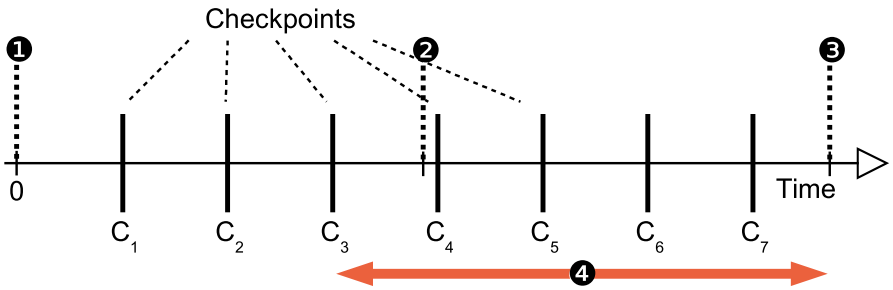

Abbildung 2.6: Beschleunigung der Fast Forwarding Phase durch Checkpoints. Der letzte Checkpoint vor der Injektion (2), hier C_3 wird geladen und das Programm erst ab diesem Zeitpunkt ausgeführt. Aus [15], Abb. 3.4

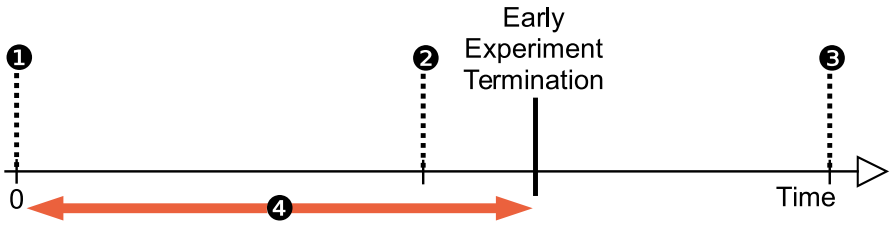

Abbildung 2.7: Beschleunigung der Monitoring Phase durch Abbrechen des Experiments, sobald das Ergebnis durch andere Experimente vorhergesagt werden kann. Aus [15], Abb. 3.4

2.4.2.2 Nach der Fehlerinjektion (Ergebnisvorhersage)

Unabhängig von der Beschleunigung vor der Fehlerinjektion kann auch danach in der *Monitoring*-Phase die Laufzeit verkürzt werden. Basis dieser Methode ist die Idee, dass bestimmte Experimente vor ihrem regulären Ende terminiert werden können, falls das Ergebnis vorhersehbar ist. Wie in Abbildung 2.7 gezeigt, kann die Ausführung der Instruktionen am Ende weggelassen werden, sobald das Resultat vorherzusehen ist.

Dafür gibt es verschiedene Ansätze, das Ergebnis vor dem regulären Ende zu bestimmen. Die Unterscheide bestehen jeweils darin, was und womit verglichen wird.

- **Vergleich mit anderen Experimenten:**

  Berrojo u. a. vergleichen den Zustand in quadratisch wachsenden Abständen ebenfalls mit dem *Golden Run*, versucht hier aber Zustände zu finden, die sich nur in einem Bit vom *Golden Run* unterscheiden [1]. Das Experiment wird dann als fehleräquivalent zu dem Experiment betrachtet, bei dem der Fehler genau in diesem unterschiedlichen Bit injiziert wird. Ist dessen Ergebnis bereits bekannt, kann das Experiment beendet und das Ergebnis

übernommen werden. Falls nicht, führt man das Experiment zu Ende und übernimmt das Resultat ebenfalls für das gefundene fehleräquivalente Experiment, welches folglich nicht mehr ausgeführt werden muss.

- **Vergleich mit relevanten Teilen des Zustands:**

  Li und Tans *SmartInjector* funktioniert ähnlich, konzentriert sich bei der Auswertung auf die Auswirkungen auf *ISA*-Level und dort auf unbemerkte Datenfehler bzw. fehlerfreie Experimente. Dafür wird allerdings nicht der komplette Zustand verglichen, sondern heuristisch ausgewählte Teile, z. B. Ausgaben oder Branch-Instruktionen [8].

- **Bildung von „*Gangs*“:**

  Vergleichbar zum *SmartInjector* versuchen Chen u. a. mit *GangES* Experimente zusammenzufassen, die einen gleichen Zwischenzustand erreichen [3]. Nur ein Experiment wird dann bis zum Ende fortgeführt und das Ergebnis für alle anderen Experimente in der „*Gang*“ angenommen.

2.5 Zustandsraumtransformationen

Die bisherigen Ansätze aus Unterabschnitt 2.4.2 beschleunigen die Experimente dabei nur vor oder nach der Fehlerinjektion. Ein Ansatz der in beiden Teilen angewendet werden kann, fehlt an dieser Stelle noch. Hier versucht Stampa anzusetzen. In seiner Masterarbeit untersucht er den Ansatz von Zustandsraumtransformationen zur Beschleunigung verschiedener Fehlerinjektionsexperimente und das grundlegende Potenzial für Fehlerinjektionskampagnen [19]. Dazu werden sogenannte *Skipsections* aufgezeichnet, die für einen Teil des Programmablaufs die Ausführung der dynamischen Instruktionen im Simulator durch Änderungen des Systemzustands ersetzten.

2.5.1 Idee

Die Idee setzt bei der in Unterunterabschnitt 2.4.2.1 beschriebenen Verwendung von *Checkpoints* an und versucht das Verfahren zu verbessern, indem die Einschränkungen umgangen werden. Statt jeweils den kompletten Zustand des Simulators zu speichern, wird nur der für den jeweiligen Abschnitt relevante Teil gespeichert. Das reduziert deutlich den Speicherbedarf und ermöglicht eine Anwendung auch nach der Fehlerinjektion. In Abbildung 2.8 wird diese skizziert. Für den grün eingezeichneten überspringbaren Abschnitt sind nur die Werte als Eingabe relevant, die, ohne vorheriges Überschreiben, im Abschnitt gelesen werden, hier gelb hervorgehoben. Als Ausgabe werden die Werte gespeichert, die im Abschnitt verändert werden, hier türkis.

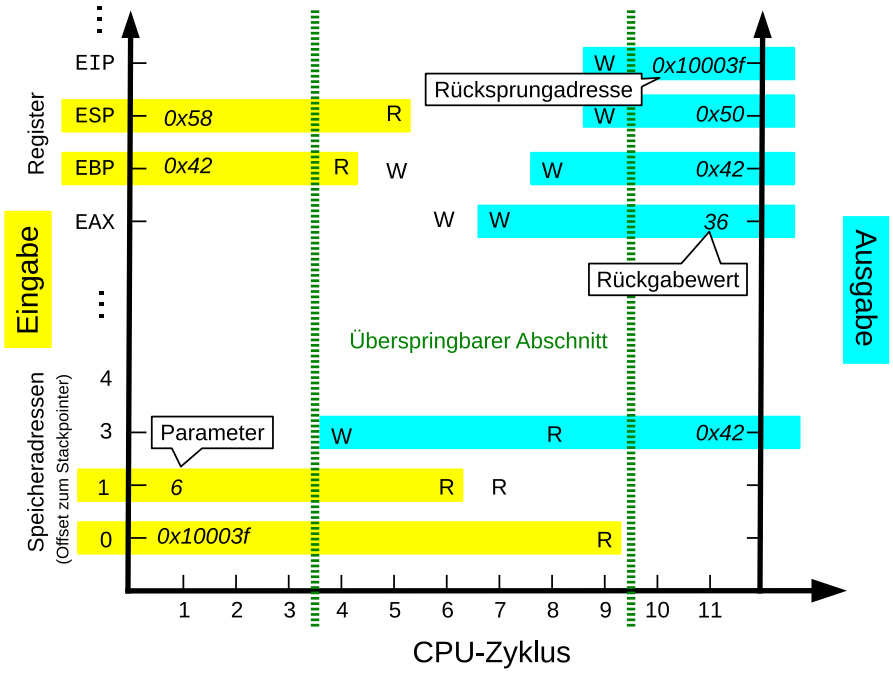

Abbildung 2.8: Gelesene Werte (gelb) werden als Eingabe und geschriebene Werte (türkis) als Ausgabe gespeichert (Aus [19], Abb. 3.3).

Wird die statische Adresse des Startpunkts eines solchen Abschnitts erreicht, also die erste zu überspringende Instruktion, werden die als Eingabe gespeicherten Werte mit den tatsächlichen Werten verglichen. Im obigen Beispiel würde beim Erreichen der vierten dynamischen Instruktion diese nicht direkt ausgeführt, sondern die Aufzeichnung der Eingabe mit den tatsächlichen Speicherwerten und Registern verglichen. Stimmen sie überein, wird der Zustand des Simulators entsprechend der Ausgabe geändert. Zusätzlich werden die Systemzeit und der *Instruction Pointer* entsprechend angepasst, sodass die Ausführung der Instruktionen übersprungen wird, ohne dass sie Änderungen für das Systemverhalten ergeben.

Wird der durch die Injektion direkt oder indirekt fehlerhafte Teil des Zustands durch die im vorliegenden Abschnitt enthaltenen Instruktionen nicht gelesen, so kann die Zustandsraumtransformation auch nach der Fehlerinjektion stattfinden. Auch eine zeitliche Verzögerung, bspw. durch ein Fehlerkorrekturverfahren, stört die Anwendung nicht, da diese sich an den statischen Instruktionsadressen orientiert und nicht an bestimmten CPU-Zyklen.

2.5.2 Ablauf und Implementierung

Die Implementierung des Verfahrens gliedert sich in zwei Bereiche. Zunächst müssen die überspringbaren Abschnitte inklusive aller benötigten Informationen er-

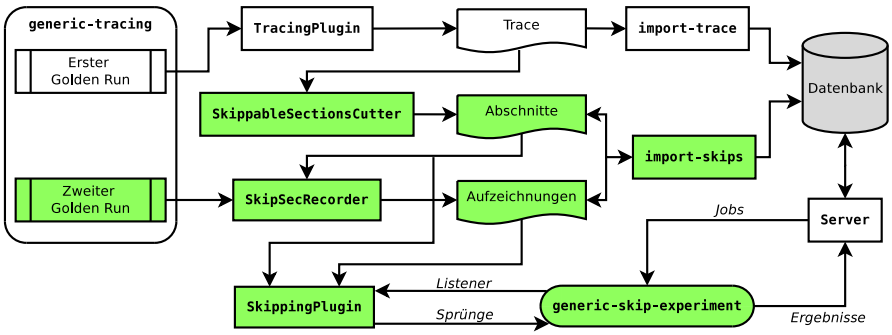

Abbildung 2.9: Die notwendigen Komponenten für die Anwendung von Skipsections und ihre Einbettung in FAIL*. Aus [19], Abb. 4.1

stellt werden. Später müssen die möglichen Transformationen während der jeweiligen Einzelexperimente überprüft und durchgeführt werden. Abbildung 2.9 zeigt eine Übersicht der dazu verwendeten Komponenten und ihre Einbettung in FAIL*.

Erstellen der Skipsections Zur Erstellung der Skipsections wird das Tracing erweitert und zwei zusätzliche Plugins verwendet, die die Unterteilung und das Aufzeichnen in unterschiedlichen Schritten vornehmen.

- Unterteilung des Traces - `SkippableSectionsCutter`: Nach der Aufzeichnung des Traces im *Golden Run*, wird dieser nach vorher festgelegten Kriterien in überspringbare Abschnitte unterteilt. Dazu wird der Trace dem `SkippableSectionsCutter` übergeben. Zusätzlich wird das *Binary* übergeben und disassembliert, um zu verhindern, das I/O-Instruktionen übersprungen werden (s. u.). Sollte eine I/O-Instruktion in einem möglichen Abschnitt auftauchen, wird dieser verworfen und ein neuer nach der Instruktion gesucht. Das Ergebnis wird als Protobuf-Datei gespeichert und dem `SkipSecRecorder` übergeben.
- Erstellen der Aufzeichnungen - `SkipSecRecorder`: Zur Aufzeichnung wird ein zweiter *Golden Run* durchgeführt und das `SkipSecRecorder` Plugin aktiviert. Wird der Beginn einer Skipsection erreicht, werden alle Register gesichert und bis zum Ende auch alle Speicherzugriffe, die Anzahl dynamischer Instruktionen und die benötigte Zeit aufgezeichnet und als Protobuf-Datei gespeichert.
- Unterteilung: Um eine geeignete Unterteilung zu finden, wird beim Einlesen des Traces ein Histogramm erstellt, das die bisher verwendeten Instruktionen enthält. Zusätzlich wird die Verwendung von Speicher approximiert. Anhand dieser Daten erfolgt die Unterteilung unter Zuhilfenahme einer vorher festgelegten Mindestgröße. Sobald der Abschnitt groß genug ist, wird ein möglichst sinnvolles Ende gesucht. Da direkt

aufeinander folgende Skipsections keinen Kontrollflusswechsel zum Simulator benötigen, sind sie besonders effektiv. Deswegen wird ein Endpunkt gesucht, der gleichzeitig gut als Startpunkt der nächsten Skipsection dienen kann. Ein guter Start-/Endpunkt sollte deshalb eine nicht sehr häufig vorkommende statische Adresse besitzen und die mit dem aktuellen Zustand zu vergleichende Ein- und Ausgabe möglichst geringhalten.

- Anbindung an die Datenbank: Unterteilung und Aufzeichnung wird durch das Tool `import-skips` in die Datenbank übernommen. Dazu werden die beiden erzeugten Protobuf-Dateien gelesen und in das erweiterte Datenbank-Schema aufgenommen. Dabei werden die Register gefiltert und nur die für den Abschnitt relevanten Register übernommen.
- Einschränkungen: Bei der Interaktion mit der seriellen Schnittstelle oder *Memory Mapped I/O* Peripheriegeräten ist nicht nur der letzte Wert für den Zustand relevant, sondern auch alle Änderungen. Daher dürfen Instruktionen, die mit der seriellen Schnittstelle kommunizieren, nicht übersprungen werden. Bei den Peripheriegeräten könnte selbst das nicht ausreichen, da die Zeit zwischen den Zugriffen verkürzt wird, was zu einem geänderten Verhalten führen kann. Daher ist der Ansatz ggf. nicht anwendbar.

Anwendung der Skipsections Das `SkippingPlugin` läuft parallel zur eigentlichen Fehlerinjektion und versucht, wann immer möglich, die Ausführungszeit durch Sprünge zu verkürzen. Dazu muss jede Skipsection das Erreichen der statischen Adresse ihrer Startinstruktion überwachen und registriert hierfür ein `BPSingleListener`, auf dessen Auslösen gewartet wird. Vor der Durchführung eines Sprungs, wird geprüft, ob dieser durch keinen anderen `Listener` des Experiments blockiert wird und die Aufzeichnung der Eingabe mit dem Systemzustand übereinstimmt. Der Sprung wird dann entweder durchgeführt, oder die Simulation normal fortgeführt.

- Blockierung der Sprünge: Durch einen Sprung darf die Ausführung eines Experiment-`Listener` auf keinen Fall verhindert werden. Experiment-`Listener` dienen zur Steuerung, z. B. zur Erkennung eines Timeouts mit einem `TimerListener` oder zur Identifizierung des Fehlerinjektionszeitpunkts durch einen `BPSingleListener(ANY_ADDR)` mit entsprechendem *Counter*. Über den Befehl `blockSkippingForListener()` können solche `Listener` daher registriert werden und das Überspringen verhindert werden.
- Optimierung der Laufzeit: Laufzeitvorteile können an verschiedenen Stellen erreicht werden. Nach der Anwendung einer Skipsection kann

BENCHMARK	TICKS	EXPERIMENTE	$T_{Original}$	$T_{Sprünge}$	REDUKTION
bitcount	$4,07 \cdot 10^7$	$5,91 \cdot 10^4$	39m 12s	41m 25s	$-5,65\%$
blowfish	$1,13 \cdot 10^7$	$2,41 \cdot 10^5$	12m 13s	8m 26s	$30,97\%$
dijkstrant	$2,03 \cdot 10^7$	$9,16 \cdot 10^5$	2h 36m	1h 41m	$35,27\%$
qsort	$4,19 \cdot 10^7$	$7,65 \cdot 10^5$	1h 52m	1h 38m	$12,5\%$
rijndael	$1,14 \cdot 10^7$	$3,76 \cdot 10^5$	22m 33s	9m 56s	$55,95\%$

Tabelle 2.1: Änderung der Laufzeit durch den Einsatz von Skipsections (Aus [19], Tab 5.3).

die Überprüfung, ob ein weiterer Sprung möglich ist, direkt erfolgen, ohne, dass der Kontrollfluss zum Simulator und zurück wechseln muss. Skipsections, die so häufig angewendet wurden, wie sie im *Golden Run* auftraten, können deaktiviert werden, also der `Listener` entfernt werden. Eine zusätzliche Anwendung, die durch die Injektion erst möglich wird, ist allgemein nicht zu erwarten, ein weiteres Auftauchen der statischen Instruktion aber schon, sodass weitere unnötige Kontrollflusswechsel und Überprüfungen gespart werden können.

2.5.3 Evaluation

Stampa beschreibt in seiner Masterarbeit [19] die grundsätzliche Durchführbarkeit von Fehlerinjektionsexperimenten mit Beschleunigung durch Skipsections (*Proof of Concept*). Dabei war das Ziel, Fehlerinjektionskampagnen zu beschleunigen, ohne die Ergebnisse zu verändern (Ergebnistreue). Dies konnte mit der Durchführung einiger Beispielkampagnen bestätigt werden, die Ergebnisse blieben mit und ohne überspringbare Abschnitte dieselben.

Wie Tabelle 2.1 zeigt, ist auch das zweite Ziel, eine Beschleunigung der Kampagne grundsätzlich möglich, allerdings nicht bei jedem Benchmark. Im Beispiel Bitcount verlangsamte sich die Ausführung, während sie sich in allen anderen Fällen verbesserte. Der enorme Unterschied zwischen 55,98% Gewinn und 5,65% Verlust zeigt das mögliche Potenzial, was in dieser Arbeit genauer untersucht werden soll.

Der erzeugte Overhead durch die Speicherung der Skipsections betrug nur zwischen 0,1% und 11% des ohnehin gespeicherten Trace-Files und fällt bei den verwendeten Aufzeichnungen nicht ins Gewicht. Das liegt aber auch daran, dass nur sehr wenige SkipSections aufgezeichnet wurden und kann unter Umständen ein größerer Faktor werden.

Ein wichtiges Detail für weitere Untersuchungen ist nämlich die verwendete Mindestlänge, welche für die Auswertung mit dem sehr großen Wert von 10^6 dynamischen Instruktionen, festgelegt wurde. Stampa erläutert die Festlegung auf diesen hohen Wert damit, dass so möglichst wenig Listenerblockierungen und *Mis-*

matches auftreten und das Filtern und Importieren in die Datenbank schneller geht [19]. Wie bereits in Abschnitt 1.2 gezeigt, bedeutet dies aber auch, dass viele Experimente die jeweiligen Skipsection nicht nutzen können. Das heißt dann auch, dass viel möglicher Laufzeitgewinn nicht ausgenutzt wird.

2.6 Zusammenfassung

In diesem Kapitel wurde gezeigt, dass Fehlerinjektion in vielfältiger Form angewendet wird und es gute Gründe gibt, Fehlerinjektionskampagnen zu beschleunigen. Der aktuelle Stand der Wissenschaft weist dabei eine Lücken zwischen Verfahren, die vor und nach der Fehlerinjektion beschleunigen, auf. Diese soll durch den Ansatz der Zustandsraumtransformationen geschlossen werden, deren Potenzial weder genau untersucht, noch vollständig ausgereizt wird. Die Probleme und Möglichkeiten zur Optimierung werden in Kapitel 3 genauer betrachtet.

3 Problemanalyse

In diesem Kapitel werden die Anforderungen und das bisherige Vorgehen genauer betrachtet, um mögliches Verbesserungspotenzial zu ermitteln. Anschließend werden die Möglichkeiten bei der Auswahl von Skipsections untersucht und mögliche Probleme bei der Bewertung der zugrunde liegenden Strategien behandelt, anhand derer in Kapitel 4 ein Lösungsansatz entwickelt wird.

3.1 Bewertungskriterien

Fehlerinjektion, die den kompletten Fehlerraum abdeckt, ist auch durch die in Unterabschnitt 2.4.1 beschriebenen Methoden, wie dem *Def/use Pruning*, häufig nur mit beachtlichem Zeit- und Ressourcenbedarf durchzuführen, da für jedes gelesene Bit ein Experiment durchgeführt werden muss. Je nach verwendeten Benchmarks ergeben sich so sehr leicht Laufzeiten im Bereich von Stunden bis Tagen. Für die im Rahmen dieser Arbeit verwendeten Benchmarks, welche in Abschnitt 6.2 kurz vorgestellt werden, zeigt Tabelle 3.1 die Bandbreite.

Auch bei in der Praxis verwendeten Benchmarks übersteigt die Anzahl der durchzuführenden Experimente leicht die Millionenmarke, während die Laufzeit jedes einzelnen Experiments ebenfalls mitwächst, sodass die Machbarkeit, auch mit Parallelisierung, nur noch begrenzt gegeben ist. Die Bedeutung von Verfahren zur Beschleunigung einzelner Experimente bleibt also entsprechend groß und der Ansatz der Zustandsraumtransformationen versucht an dieser Stelle, eine sinnvolle Ergänzung zu den in Unterabschnitt 2.4.2 beschriebenen Verfahren zu liefern. Das mögliche Potenzial wurde dabei aber noch nicht untersucht und variiert mit der Einteilung in Skipsections, welche hier genauer untersucht werden soll.

Dabei ergeben sich folgende Anforderungen für Verfahren zur Beschleunigung von Einzelexperimenten:

`Benchmark`	`Instruktionen`	`Gelesene Bytes`	`Experimente`
sha	16.712	25.170	201.360
dijkstrant	$5,05 \cdot 10^6$	$5,44 \cdot 10^6$	$4,35 \cdot 10^7$
histo	$1,089 \cdot 10^8$	$5,30 \cdot 10^7$	$4,24 \cdot 10^8$

Tabelle 3.1: Übersicht über verwendete Benchmarks, Werte mit `dump-trace` ermittelt.

Effektivität: Primärziel ist die Beschleunigung der Fehlerinjektionskampagnen durch Reduzierung der Laufzeit jedes einzelnen Experiments. Die Güte verschiedener Strategien zur Unterteilung wird also hauptsächlich an der zu erwartenden Laufzeit gemessen und diese ist zu optimieren.

Overhead: Die Speicherung der Skipsections benötigt Kapazität, sodass ein gewisser Overhead nicht zu vermeiden ist. Im Vergleich zum Trace ist der benötigte Speicherplatz alledings minimal, sodass auch mehrere Skipsections zur vollständigen Überdeckung des Traces nicht sonderlich ins Gewicht fallen. Da der Trace aber auch mehrfach überdeckt werden kann, ist der Anzahl an Skipsections keine Grenze gesetzt, sodass durch deren schiere Anzahl der Overhead zu groß werden kann. Dies ist unbedingt zu vermeiden.

Ergebnistreue: Die Ergebnise Einzelexperimente und damit auch der vollständigen Kampagne dürfen durch die Anwendung von Skipsections nicht verfälscht werden.

Generalität: Um einen größtmöglichen Nutzen zu haben, sollen Zielprogramme und Backends flexibel sein. Dies ergibt sich im Wesentlichen aus der Flexibilität von FAIL*, die Anpassung an das jeweilige Backend ist nur minimal.

Nutzerfreundlichkeit: Der Mehraufwand für den Anwender sollte minimal sein, sowohl bezüglich des zeitlichen Aufwands, als auch bezüglich der Bedienbarkeit. Dafür bietet sich eine Nutzung über Kommandozeilenparameter an.

3.2 Bisheriges Vorgehen

Um eine Idee für Vor- und Nachteile möglicher Strategien zu bekommen und die notwendigen Stellschrauben zur Verbesserung zu identifizieren, betrachten wir zunächst den genauen Ablauf der Unterteilung, Aufzeichnung und Anwendung von Skipsections, wie sie durch Stampa umgesetzt wurde [19]. Dabei liegt das Augenmerk darauf, welche Aspekte der Programmierung einen großen Einfluss auf das Gesamtergebnis haben. Hierbei ist anzumerken, dass die Arbeit von Stampa sich nur ganz am Rande mit der Unterteilung beschäftigt hat und das Ziel nur der Nachweis war, dass mit dem Ansatz eine Beschleunigung möglich ist.

3.2.1 Unterteilung

Die Unterteilung wird durch den `SkippableSectionsCutter` vorgenommen, der als Parameter das Trace-File und das ELF-Binary des jeweiligen Benchmarks erhält. Den genauen Ablauf betrachten wir im Folgenden schrittweise:

(1) Das Binary wird disassembliert, um nicht überspringbare Bereiche, z. B. Instruktionen zur seriellen Ein- und Ausgabe, zu identifizieren.

(2) Der Trace wird aus der Datei eingelesen und dabei gleichzeitig ein Histogramm der statischen Instruktionsadressen erstellt.

(3) Der Trace wird rückwärts durchlaufen und der Speicherverbrauch zu jedem Zeitpunkt abgeschätzt. Dabei wird der aktuelle Speicherverbrauch durch Lesen eines unbenutzten Bytes erhöht und durch Schreiben eines zuvor gelesenen Bytes reduziert.

(4) Bei der anschließenden Unterteilung wird über den Trace iteriert und immer dann eine neue Skipsection gestartet, wenn die aktuelle dynamische Instruktion nicht bereits in einer möglichen Skipsection ist. Wird eine I/O-Instruktion erreicht, wird die aktuelle Skipsection verworfen und weiter gesucht.

(5) Wurde die Mindestlänge erreicht, wird, bis zur Maximallänge, ein besseres Ende gesucht, wobei als Gütekriterium die minimale Summe von Speicherbedarf und Vorkommen der aktuellen statischen Instruktion (mit dem Faktor 100 multipliziert) verwendet wird. Anschließend startet direkt die nächste Skipsection.

Die Unterteilung wird also anhand der drei Parameter Größe der Ein-und Ausgabe, Häufigkeit der statischen Adresse der letzten Instruktion und Mindest- bzw. Maximallänge ausgewählt. Wobei die Maximallänge abhängig von der Mindestlänge ist, hier das Doppelte. Der Gedanke dahinter ist, dass bei geringer Ein- und Ausgabegröße, sowohl der Vergleich, als auch die Transformation schneller durchzuführen sind [19]. Allerdings funktioniert die Abschätzung nur bei einer Skipsection gut, bei allen weiteren, wird sie immer ungenauer.

Abbildung 3.1 veranschaulicht dies an einem Beispiel. Das Schreiben der Bytes eins und zwei bei der sechsten Instruktion verringert die Abschätzung der Speichernutzung. Unterteilt man allerdings zwischen dem Schreib- und Lesezugriff, wie in diesem Fall durch die orangene Linie dargestellt, so hat der Schreibzugriff keinen Einfluss mehr auf die tatsächliche Eingabe einer möglichen Skipsection. In beiden Abschnitten sind die Bytes enthalten. Analog gilt dies für aufeinanderfolgende Lesezugriffe ohne einen Schreibzugriff dazwischen. Hier hat nur der Erste Einfluss auf die Abschätzung, tatsächlich wäre aber jeder, je nach Unterteilung, ein folgender Zugriff ebenfalls relevant. In diesem Beispiel würden die Speicherzugriffe bei der zweiten Instruktion, keinen Einfluss mehr haben, da die gelesenen Bytes bereits berücksichtigt werden. Durch die Unterteilung zwischen zwei Zugriffen (lila) sind diese nur in einem der Abschnitte bereits enthalten und der Lesezugriff bei der zweiten Instruktion erhöht die Speichernutzung.

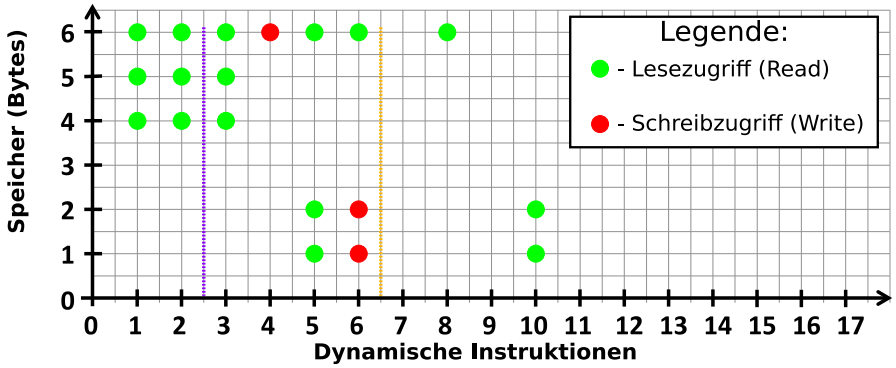

Abbildung 3.1: Ablauf von Speicherzugriffen und Einteilung in Skipsections zur Verdeutlichung möglicher Ungenauigkeiten.

Eine häufig im Programmablauf wiederkehrende statische Adresse als Beginn einer Skipsection wäre sehr problematisch, da der *Listener* auf die Start-Adresse sehr häufig einen Kontrollflusswechsel und eine Überprüfung auslösen würde, ohne dass der Beginn einer Skipsection erreicht wurde, sodass kein Vorteil durch einen Sprung diesen Aufwand ausgleichen kann.

Zwei direkt aufeinander folgende Skipsections haben den Vorteil, dass zwischen beiden kein Kontextwechsel zurück zum Simulator notwendig ist. Daraus ergibt sich, dass der gemeinsame Start- und Endpunkt einer Skipsection bei einer möglichst seltenen, statischen Instruktion liegen sollte. Allerdings stimmt dies nur, wenn die nächste Skipsection auch tatsächlich verwendet wird. Wird diese, wegen des Erreichens einer I/O-Instruktion oder des Ende des Traces, verworfen, so würde der Endpunkt unnötig beeinflusst und der verbliebene Faktor der Ein- und Ausgabegröße weniger stark berücksichtigt. Der Start der nächsten Skipsection hingegen ist direkt nach der I/O-Instruktion, obwohl hier unter Umständen eine sehr häufig vorkommende Instruktion ausgeführt wird.

3.2.2 Aufzeichnung

Das Anfertigen passender Aufzeichnungen geschieht während eines zweiten *Golden Run*, durch das `SkipSecRecorder` Plugin, welches zusätzlich die Skipsection als Datei übergeben bekommt.

(1) Ein `MemAccessListener` und ein `BPSingleListener` werden jeweils mit `ANY_ADDR` initialisiert und beim Simulator registriert.

(2) Bei einem Speicherzugriff wird geprüft, ob aktuell eine Skipsection aufgezeichnet wird. In diesem Fall werden, bei erstmaligem Lesen, Wert und Speicheradresse der Eingabe hinzugefügt bzw. beim Schreiben, der Wert und die Speicheradresse, der Ausgabe hinzugefügt.

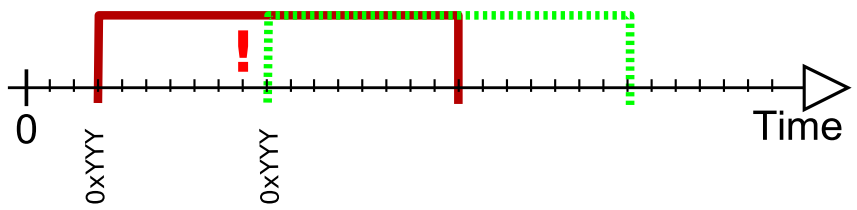

Abbildung 3.2: Das erste Auftreten der statischen Adresse kann bereits vor und außerhalb einer Skipsection sein, sodass falsch aufgezeichnet wird.

(3) Bei jedem Erreichen einer Instruktion wird ebenfalls unterschieden, ob sich diese in einer aufzunehmenden Skipsection befindet. Falls ja, prüft sie, ob die Anzahl der übersprungenen Instruktionen der Länge der Skipsection entsprechen und beendet in diesem Fall die Aufzeichnung.

(4) Außerhalb einer Skipsection wird der *Instruction Pointer* mit der Start-Adresse der nächsten aufzuzeichnenden Skipsection verglichen und ggf. eine neue Aufzeichnung gestartet.

(5) Beim Öffnen und Schließen einer Skipsection werden dabei alle Register ebenfalls gespeichert. Die Filterung, welche davon Teil der Ein- und Ausgabe sind, erfolgt später beim Importieren in die Datenbank.

Diese Implementierung hat leider einige Nachteile. Dadurch, dass auch außerhalb einer SkipSec der `MemAccessListener` aktiv ist, wird der Kontrollfluss unnötig häufig gewechselt. Auch könnte zwischen zwei Skipsections ein Listener auf die nächste Startadresse verwendet werden, statt bei jeder Instruktion zu prüfen, ob diese erreicht wurde. Viel schlimmer ist allerdings, dass unter Umständen auch die falsche Aufzeichnung erstellt werden kann. Sei die in Abbildung 3.2 grün dargestelle Skipsection zur Aufzeichnung ausgewählt, ihre statische Adresse aber schon eher im Trace vorkommend. Im dargestellten Fall liegt das vorherige Auftreten nicht in einer anderen Skipsection, z. B. durch eine nicht überspringbare Instruktion und einen zu kurzen Abstand davor. Dann würde der `SkipSecRecorder` ab da die Aufzeichnung beginnen und die rote Aufzeichnung erstellen.

Durch die „falsche“ Skipsection, würde nicht nur die Laufzeit schlechter, sondern auch die Ergebnistreue könnte nicht mehr gegeben sein, da die unterschiedliche serielle Ausgabe als SDC gewertet wird. Es müssen für das Aufzeichnen also weitere Informationen vorliegen, um die richtige statische Adresse auswählen zu können.

3.2.3 Skipping

Die Anwendung der Skipsections zur Beschleunigung der Fehlerinjektion erfolgt durch das `SkippingPlugin`, wobei sich der Ablauf in die Vorbereitung und die

Anwendung gliedert.

Vorbereitung Das Plugin lädt die SkipSecs[1] und SkipSecRecs[2]. Dabei wird für jede Skipsection ein *Listener* auf die Startadresse (statische Adresse) erzeugt. Die Aufzeichnungen, auch als *Recordings* bezeichnet, werden der jeweiligen Skipsection hinzugefügt, also alle Skipsections mit gleicher Startadresse zusammengefasst. Die Information über die Länge der jeweiligen Skipsection befindet sich dann nur noch im Recording. Um Skipsections später deaktivieren zu können, wird eine *Counter* festgelegt, der doppelt so hoch ist, wie die das Vorkommen der jeweiligen Aufzeichnungen im *Golden Run*. Alle *Listener* werden dann im Simulator registriert.

Anwendung Wenn ein *Listener* auslöst, wird geprüft, zu welcher Skipsection dieser passt und anschließend für alle Aufzeichnungen geprüft: Wurde die Aufzeichnung noch nicht zu häufig genutzt, wird sie durch keinen *Listener* des Experiments blockiert und passt die Eingabe zum aktuellen Systemzustand. Falls ja, ändere den Zustand und prüfe vor der Rückkehr zum Simulator, ob eine mögliche Skipsection direkt anschließt. Zusätzlich wird gezählt, welche Aufzeichnung benutzt, oder fälschlich überprüft wurde, um Skipsctions möglicherweise zu deaktivieren. Die Zustandsänderungen beinhalten dabei auch die *Counter* der *Listener*, die Systemzeit und den *Instruction Pointer*.

Allgemein können Skipsections nicht den kompletten Trace abdecken, sodass ein Teil der Instruktionen auch im Simulator ausgeführt wird. In dieser Zeit ist für jede Skipsection ein *Listener* auf die statische Startadresse aktiv, was bei eine langen Trace und viele Skipsections dazu führen kann, dass viele Skipsections auslösen, ohne dass die jeweilige Aufzeichnung passt. Das würde einen erheblichen Laufzeit-Nachteil mit sich bringen. Durch sehr lange Skipsections, wie sie bei der Auswertung verwendet wurden, tritt dieses Problem nur sehr begrenzt auf. Für eine Optimierung der Unterteilung muss die Anzahl der aktiven Skipsections irgendwie eingeschränkt werden.

3.2.4 Zusammenfassung

Aus diesen Untersuchungen ergeben sich einige Ideen für den Entwurf, die in der folgenden Auflistung genauer beschrieben werden.

(1) Eine geringe Eingabe (bzw. Ausgabe) und eine seltene Startadresse sind gute Kriterien für die Unterteilung. Deren Bestimmung muss aber zusammen mit der jeweiligen Einteilung vorgenommen werden.

[1] Die Datei oder Datenstruktur, die die Skipsections enthält.
[2] Die Datei oder Datenstruktur, die die Aufzeichnungen enthält.

(2) Zum Aufzeichnen einer Skipsection müssen genaue Informationen über deren Zeitpunkt vorliegen, also entweder das wievielte Auftreten der statischen Startadresse der tatsächliche Beginn ist, oder der Zeitpunkt, also die wievielte dynamische Instruktion.

(3) Zu viele aktive statische Adressen erhöhen die Laufzeit, deshalb muss ein Mechanismus verwendet werden, der nicht alle möglichen Skipsections nutzt, sondern nur einen relevanten Teil aktiviert.

3.3 Strategische Platzierung von Skipsections

Das Ziel der Platzierung verschiedener Skipsections ist, die durchschnittliche Laufzeit so weit wie möglich zu verringern. Daraus ergibt sich die Frage, bei welcher Verteilung die Laufzeit den möglichst geringsten Wert annimmt.

3.3.1 Ideale Strategie eines Einzelexperiments

Die Ausführung von Instruktionen im Simulator ist um ein Vielfaches langsamer als die Ausführung auf echter Hardware. Ziel bei der Platzierung von Skipsections muss also sein, so viele Simulatorinstruktionen wie möglich zu überspringen und trotzdem an der richtigen Stelle den Fehler injizieren zu können. Abbildung 3.3 zeigt die Platzierung von Skipsections über den zeitlichen Verlauf eines Experiments. Von dem Beginn des Zielprogramms (1), über die Injektion des Fehlers (2), bis zum Ende des Programms. Die Bereiche ohne Unterteilung der Zeitachse, jeweils durch "[...]" gekennzeichnet, können dabei beliebig lang sein und die unterschiedlichen Einteilungen ergeben sich durch die verwendete Hardware und durch die ausgeführten Benchmarks.

In der ersten Variante (rot), startet direkt zu Beginn eine Skipsection und endet unmittelbar vor der Fehlerinjektion. Anschließend startet sofort die zweite Skipsection und geht bis zum Ende des Programms. Die Anzahl ausgeführter Simulatorinstruktionen ist so gering wie möglich und die Einteilung folglich optimal, falls alle Skipsections abgespielt werden können. Auch wenn der injizierte Fehler keine Auswirkungen hat und wieder verschwindet, so ist anzunehmen, dass er zumindest über mehrere Instruktionen noch Teil des verwendeten Zustands ist.

Die zweite Variante (oliv) wäre in diesem Fall ideal. Die der Fehlerinjektion folgenden Instruktionen sind nicht Teil der Skipsection, sodass die Verwendung des fehlerhaften Systemteils nicht mehr enthalten ist und die Skipsection auch tatsächlich abgespielt werden kann. Alle überspringbaren Simulatorinstruktionen werden übersprungen, die Strategie ist also optimal.

Allerdings kann es vorkommen, dass im weiteren Verlauf der Ausführung der fehlerhafte Teil nochmal verwendet wird, oder eine nicht überspringbare I/O-Instruktion, die Anwendung einer Skipsection verhindert. Dies wird in der dritten

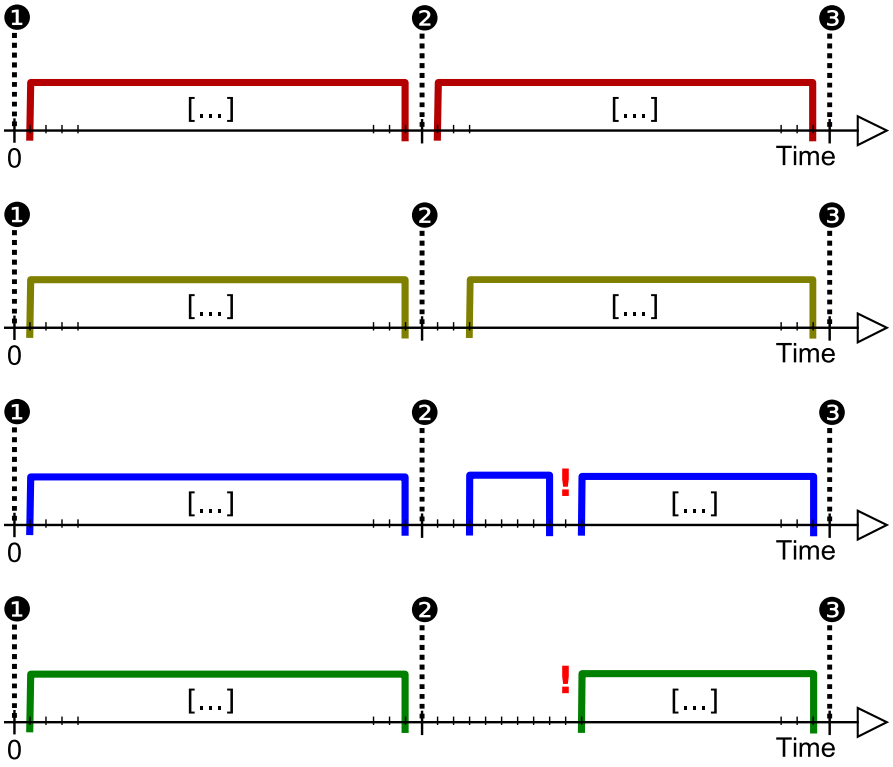

Abbildung 3.3: Optimale Einteilung von Skipsections in Abhängigkeit von Hardware und Benchmark

Variante (blau) veranschaulicht. Die durch das rote Ausrufezeichen markierte Instruktion darf nicht übersprungen werden. Deshalb wird der zu überdeckende Bereich nach der Fehlerinjektion in zwei Skipsections aufgeteilt, zwischen denen eine (oder mehrere) Instruktion(en) im Simulator durchgeführt werden müssen. Die Einteilung erfüllt damit noch immer das Kriterium der maximalen Überdeckung von Simulatorinstruktionen und kann also ebenfalls optimal sein.

Allerdings werden hier in der zweiten Skipsection nur sehr wenige Instruktionen übersprungen. Je nach Hardware und Simulator ist daher anzunehmen, dass die Ausführung von einigen wenigen Instruktionen im Simulator deutlich schneller ist als die Unterbrechung mit Kontrollflusswechsel. Die in Abschnitt 5.2 beschriebenen Messungen bestätigen den verhältnismäßig hohen Zeitaufwand eines Kontrollflusswechsels. In diesem Fall wäre die letzte Variante (grün) optimal, da hier nur die Bereiche überdeckt werden, die eine Beschleunigung bringen, diese allerdings so vollständig wie möglich.

Das Beispiel zeigt dabei deutlich, wie abhängig eine optimale Strategie, selbst für ein einzelnes Experiment, vom Benchmark und der Ausführungsgeschwindigkeit des Simulators ist. Für eine Vielzahl von Experimenten wird die Platzierung noch deutlich komplizierter, zumal weitere Aspekte zu berücksichtigen sind. Ein teurer Kontrollflusswechsel passiert immer dann, wenn die statische Adresse einer Skipsection erreicht wird, also auch, wenn die Skipsection nicht anwendbar ist. Die Wahrscheinlichkeit dafür steigt dabei nicht nur mit Anzahl und Häufigkeit der Startadressen der Skipsections, sondern auch mit der Anzahl im Simulator ausgeführten Instruktionen.

3.3.2 Varianz der Ergebnisse

Da eine große Anzahl von Experimenten für verschiedenste Fehlerinjektionszeitpunkte optimiert werden soll, bleibt die Kernidee, möglichst wenig Kontrollflusswechsel und Simulatorinstruktionen auszuführen, um von Beginn des Programms zu Fehlerinjektion und danach zum Ende zu gelangen. Eine so detaillierte Überlegung kann dabei natürlich nicht für jeden einzelnen Zeitpunkt gemacht werden, zumal deren Aussagekraft durch die Wechselwirkung mit weiteren Skipsections für andere Experimente nicht sonderlich hoch wäre. Sinnvoll wäre ein Ansatz, der für Sprünge *zu jedem Zeitpunkt* durchschnittlich möglichst wenig Simulatorinstruktion bedarf.

Dabei kann ein falscher Ansatz nicht nur sehr viel Potenzial verschwenden, es kann auch passieren, dass die Laufzeit erheblich schlechter wird.

In Abbildung 3.4 wird dies an einem besonders anschaulichen Beispiel dargestellt. Die Abschätzungen für den, auf die ersten 300.000 Instruktionen limitierten, Benchmark *bfs* zeigt einen besonders großen Unterschied für verschiedene Strategien, welche später in Abschnitt 4.5 ausführlich beschrieben werden. Eine Verhältnismäßig kleine Mindestlänge für Skipsections von 4.000 Instruktionen be-

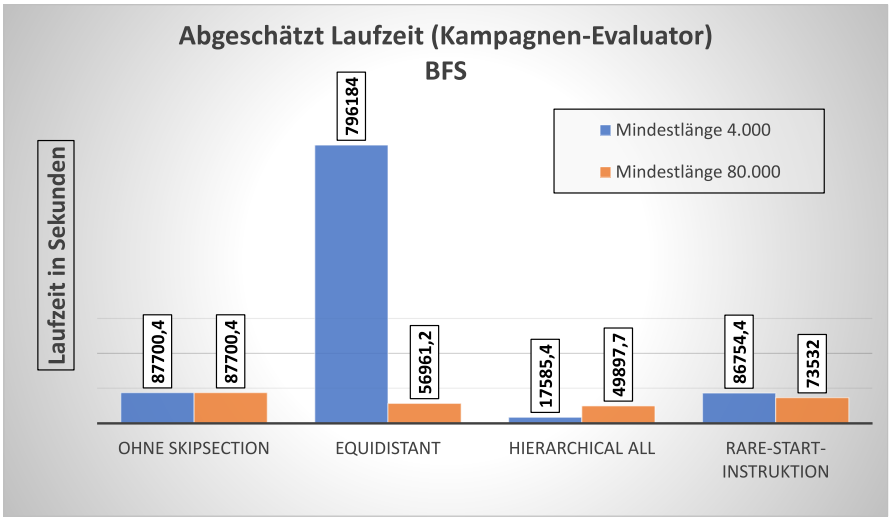

Abbildung 3.4: Abgeschätzte Laufzeiten verschiedener Unterteilungsstrategien auf einem abgekürzten Benchmark als Beispiel (BFS - Limitierung auf die ersten 300.000 Instruktionen).

wirkt für die *Equidistant*-Srategie eine Verschlechterung um fast das zehnfache, während die *Hierarchical-All*-Strategie die Laufzeit um den Faktor 5 reduziert. Erhöht man allerdings die Mindestlänge hingegen auf 80.000 Instruktionen, so zeigt sich, dass die *Equidistant*-Strategie die Laufzeit auch verbessern kann, während die *Hierarchical-All*-Strategie nur noch einen deutlich kleineren Vorteil bietet. Die dritte Strategie hingegen weist eine nur sehr leicht verbesserte Laufzeit auf, die auch weniger von der Länge der Skipsection abhängt.

Die Berechnung der Laufzeit erfolgt dabei bereits unter Berücksichtigung einer in Abschnitt 4.3 näher erläuterten Optimierung, die verhindert, dass viele Skipsections gleichzeitig aktiv sind und damit die Anzahl der Kontrollflusswechsel durch wiederholt auftauchende Startadressen erheblich reduziert. Der ursprüngliche Ansatz aus [19] sah dies nur für den Fall vor, dass die Skipsection bereits vielfach falsch geprüft wurden.

3.4 Test- und Analysierbarkeit von Auswahlstrategien

Die Anzahl möglicher Strategien für eine Unterteilung ist dabei nahezu unbegrenzt, weil jeder Ansatz zur Unterteilung noch viele Parameter hat, deren Werte je nach Benchmark unterschiedliche Ausprägungen annehmen müssen, um die Laufzeit möglichst gut zu verbessern. So würde in Unterabschnitt 3.3.2 deutlich gezeigt, dass die Kennzahl der Mindestlänge für jede Strategie neu untersucht

werden muss und zudem offensichtlich auch von der Anzahl der dynamischen Instruktionen, also vom jeweiligen Benchmark abhängig ist.

Andere Strategien suchen auf unterschiedliche Art und Weise besonders geeignete Start- und Endpunkte. Das Intervall, in dem gesucht wird, ist dabei, analog zur Mindestgröße, für die Laufzeit entscheidend. Ein weiterer Punkt ist der Grad der Überdeckung des Traces, der zwischen nur wenigen, ausgewählten Bereichen und einer mehrfachen Überdeckung durch überlappende Skipsections, variieren kann.

Diese enormen Variations- und Kombinationsmöglichkeiten bei der Auswahl geeigneter Skipsections, in Verbindung mit der langen Laufzeit von Fehlerinjektionskampagnen, macht einen Vergleich vieler verschiedener Parametrisierungen extrem aufwändig und im Rahmen einer Masterarbeit für reale Fehlerinjektionskampagnen nahezu unmöglich. Ein Kernziel der Arbeit besteht daher darin, einen sinnvollen Evaluator zu entwickeln, der die Laufzeit von Kampagnen abschätzt und so eine Beurteilung verschiedener Ansätze ermöglicht. Da die Ergebnisse verschiedener Strategien auch von der Struktur des verwendeten Benchmarks abhängig ist, sollten alle Untersuchungen mit mehreren Benchmarks durchgeführt werden. Dabei besteht allerdings das Porblem, dass diese sehr unterschiedlich lang sind. Das machts es schwierig, die Mindestlänge konstant zu lassen und nur einen anderen Parameter zu ändern, aber trotzdem mit verschiedenen Benchmarks zu arbeiten.

3.5 Zusammenfassung

Das bisherige Vorgehen zur Einteilung in Skipsections und deren Anwendung bietet noch einige Stellschrauben zur Verbesserung. Aus ihnen würden wichtige Entwurfsideen abgeleitet, die in Kapitel 4 zu wichtigen Entscheidungen bezüglich des *Skippings* allgemein führen. Da es nicht sinnvoll möglich ist, für jedes Experiment eine optimale Unterteilung zu bekommen, muss zudem mit verschiedenen Ansätzen probiert werden, wann eine Unterteilung im Durchschnitt möglichst große Laufzeitgewinne liefert. Die unterschiedlichen Strategien dafür werden ebenfalls in Abschnitt 4.5 vorgestellt.

4 Entwurf

In diesem Kapitel soll das überarbeitete Konzept der überspringbaren Abschnitte (Skipsections) entworfen und dabei die Möglichkeit integriert werden, dass verschiedene Strategien verwendet und bewertet werden können. Es stellt den Kern der vorliegenden Arbeit dar und ist die Grundlage der in Kapitel 5 präsentierten Implementierung.

4.1 Überblick

Die Verwendung von Skipsections zur Beschleunigung von Fehlerinjektionskampagnen gliedert sich in zwei separate Schritte. Die Unterteilung und Erstellung der Skipsections ist der erste Schritt, der im Rahmen dieser Arbeit zum Vergleich vielfach mit verschiedenen Strategien wiederholt wird. Die Anwendung einer konkreten Unterteilung bei einer realen Kampagne erfolgt anschließend in einem weiteren Schritt. Beide Schritte sollen dabei mit möglichst wenig Änderungen an bestehenden Komponenten, in die Architektur von FAIL* eingebettet werden. Dies kann durch die Anpassung der Experimente `generic-tracing` und `generic-skip-experiment` umgesetzt werden, welche die typischen Anwendungsfälle der Fehlerinjektion mit FAIL* abdecken. Das `generic-tracing` Experiment dient dabei zur Vorbereitung der Fehlerinjektion, indem die Instruktionen und Speicherzugriffe während eines *Golden Run* aufgezeichnet werden. Aus der entstehenden Trace-Datei werden dann die Zeitpunkte der Fehlerinjektion ermittelt. Das Experiment `generic-experiment` führt die Fehlerinjektion anschließend aus.

Die notwendigen Ergänzungen werden in Abbildung 4.1 dargestellt. Zusätzlich zum `TracingPlugin` – welches für die Anwendung in diesem Kontext erweitert wurde – verfügt das Experiment über einen `Cutter` und einen `Evaluator`. Beide verwenden dabei mindestens eine `CuttingStrategy`, sind aber auch in der Lage für viele Strategien gleichzeitig eine Unterteilung bzw. Laufzeitabschätzung zu erstellen. Das Zwischenschalten einer weiteren Klasse zwischen Experiment und konkrete Strategie soll bei der Verwendung mehrerer Strategien das wiederholte Laden der Trace-Datei verhindern und zudem die Möglichkeit bieten, diesen nur teilweise zu verwenden. So lassen sich Strategien auf verschiedenen Benchmarks mit gleicher Länge untersuchen, was die Vergleichbarkeit der Ergebnisse erhöht. Das `SkippingPlugin` des `generic-skip-experiment` verwendet die jeweilige Strategie nur zum Laden der benötigten Komponenten, die Benutzung

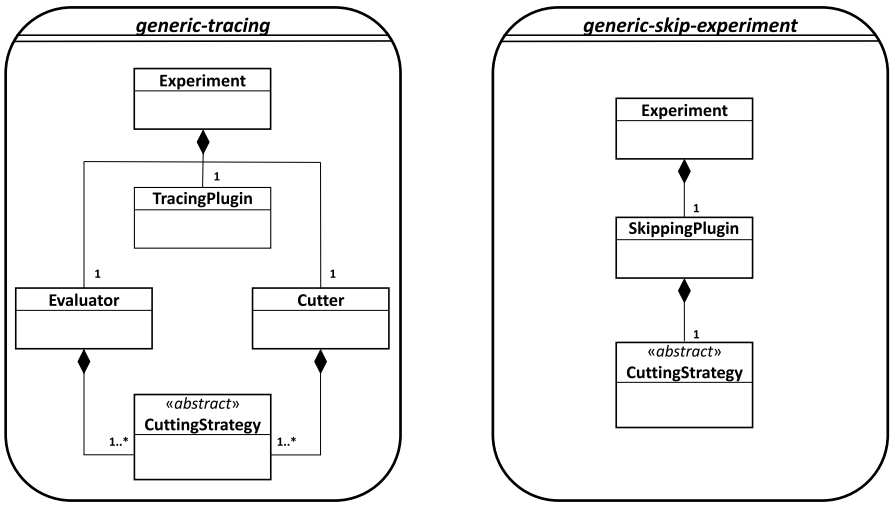

Abbildung 4.1: Unterteilung des Ablaufs in die Abschnitte `generic-tracing` zur Erstellung der Skipsections und `generic-skip-experiment` zu deren Anwendung.

bleibt aber im Wesentlichen gleich.

Daraus resultiert ein etwas veränderter Ablauf, der in Abbildung 4.2 visualisiert wird. Das Experiment steuert die jeweiligen Komponenten und startet einen *Golden Run* mit aktiviertem `TracingPlugin` zur Erstellung des Traces. Dabei wird zusätzlich eine erweiterte Variante[1] erzeugt, die im weiteren Verlauf von den neuen Komponenten verwendet wird.

Der `Cutter` lädt die Datei in einen Vektor in den *RAM* und übergibt diesen an die verwendeten Strategien. Je nach konkreter Strategie werden unterschiedliche Kriterien zur Unterteilung in Skipsections angewendet. Durch das Laden des kompletten Traces in den *RAM*, kann mit wenig Aufwand beliebig auf diesem vor- und zurückgegangen werden. Das erleichtert die Benutzung, kann bei langen Traces aber große Teile des verfügbaren Speichers belegen. Anschließend werden unter Verwendung des Traces Aufzeichnungen zu den jeweiligen Skipsections erstellt, die die Ein- und Ausgabe, so wie später benötigte Informationen zu den übersprungenen Instruktionen, enthalten. Außerdem wird ein Skipgraph erstellt, der die in Unterabschnitt 4.3.1 genauer beschriebenen, optimalen Pfade enthält.

Der `Evaluator` berechnet aus diesen Dateien und dem Trace anschließend eine abgeschätzte Laufzeit, den zusätzlichen Speicherverbrauch und die Anzahl der erstellten Skipsections und speichert alle Informationen zur Analyse der Strategien als CSV-Datei ab. Die Funktionsweise und Idee dahinter werden in Abschnitt 4.4 genauer betrachtet.

[1]Der ursprüngliche Trace wird nur zur Kompabilität mit anderen Anwendungen erzeugt, im Folgenden meint Trace immer die erweiterte Version.

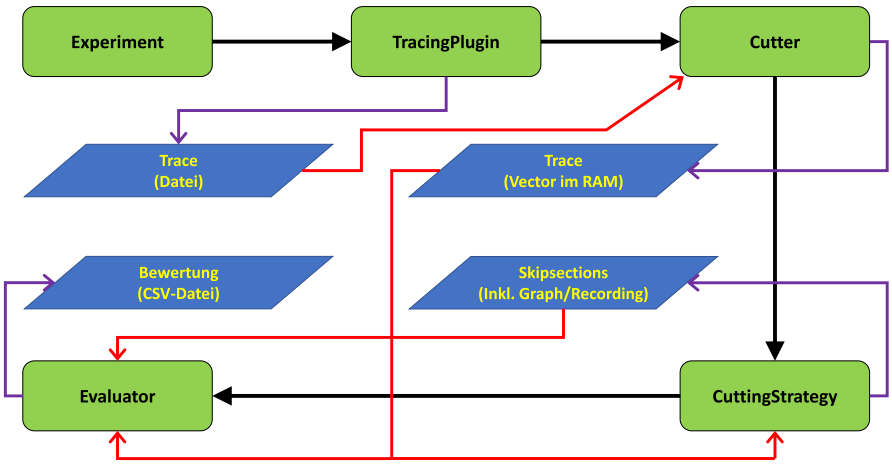

Abbildung 4.2: Ablauf der Erstellung und Bewertung von Skipsections mit den jeweiligen Ein- und Ausgaben.

4.2 Erweiterung des Tracings

Zur Unterteilung des Traces sind genauere Erkenntnisse über die jeweiligen Instruktionen notwendig, insbesondere um zu verhindern, dass I/O-Instruktionen übersprungen werden. Zudem benutzt ein Teil der Strategien weitere Informationen, die aus dem bisherigen Trace nicht direkt erkennbar sind, wie die Verwendung von Registern. Dafür wird das ELF-Binary übergeben und disassembliert[2]. Die dabei zusätzlich gewonnenen Informationen wurden bisher aber nicht gespeichert, sondern anschließend verworfen, sodass wichtige Informationen für die Aufzeichnung und Anwendung verloren gingen.

Deshalb soll das Disassemblieren bereits frühzeitig erfolgen und der Trace um die später noch relevanten Angaben erweitert werden. Dazu wird der Trace um den Inhalt verwendeter Speicherbereiche, den Namen der Instruktionen und die genutzten Register inklusive alle dazugehöriger Informationen ergänzt. Das hat auch den Vorteil, dass so die Registerinhalte zur Unterteilung verwendet werden können. Welche Register genutzt werden, lässt sich aus dem disassemblierten Binary ablesen, deren Werte aber nur bei der tatsächlichen Ausführung des Programms ermittelt werden.

Da der Trace in dieser Form alle Informationen enthält, die zur Erstellung der Aufzeichnungen der jeweiligen Skipsections notwendig sind, kann diese ebenfalls auf dem Trace erfolgen, ohne dass das Programm ein zusätzliches mal ausgeführt werden muss. Anhand der aus der Disassemblierung gewonnen Erkenntnisse kann die Aufzeichnung sogar genauer erfolgen, da bekannt ist, welche Teile eines Registers gelesen werden. Bei FAIL* sind alle Register 64 Bit breit , auch wenn diese

[2]Zur Disassemblierung wird in dieser Arbeit die Capstone Bibliothek verwendet.

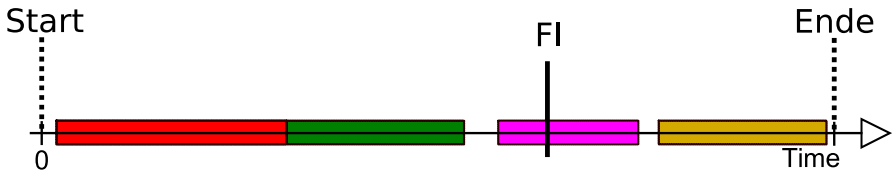

Abbildung 4.3: Mit den bekannten Werten Startpunkt und Länge (jeweils dynamische Instruktionen) kann immer genau entschieden werden, welche Skipsection aktiv sein muss.

bei 32-Bit Systemen nicht vollständig genutzt werden. Manche Instruktionen lesen und ändern aber nur kleine Teile des Registers, z. B. nur das jeweilige *high*- oder *low*-Register.

Ist die Information im Trace und dadurch in der Aufzeichnung der jeweiligen Skipsection enthalten, so kann die Änderung in einem nicht verwendeten Registerteil, die Anwendung der Skipsection nicht blockieren.

Der in Abschnitt 4.4 genauer beschrieben Kampagnen-Evaluator nutzt diese Informationen noch nicht, könnte damit aber auch zur Abschätzung bei anderen Fehlermodellen verwendet werden, bspw. die Fehlerinjektion in Register.

4.3 Skipping: Prüfen und Anwenden von Skipsections

Das Skipping selbst, also während der Durchführung einer Kampagne prüfen, ob eine Skipsection schalten kann und entsprechend die Transformation des Zustandsraums durchführen, soll ebenfalls durch ein `SkippingPlugin` erfolgen. Der bedeutenste Unterschied ist dabei, dass die Skipsections um einen Startzeitpunkt, entsprechend ihrer dynamischen Instruktion ergänzt werden. Dadurch hat jede Skipsection nur noch eine passende Aufzeichnung. Zwei Skipsections, die zum gleichen Zeitpunkt starten und gleich lang sind, müssen zwangsläufig identisch sein. Bis zur Fehlerinjektion und mit Einschränkung auch danach, lässt sich also genau ermitteln, welche Skipsections in welcher Reihenfolge auslösen müssten. Dies ermöglicht es, leicht zu erkennen, welche Skipsections nicht bzw. erst nach der Fehlerinjektion gebraucht werden, sodass nur noch ein relevanter Teil zu Beginn als *Listener* registriert wird. Genau betrachtet muss nur genau eine Skipsection aktiv sein und sogar der *Listener* für die Fehlerinjektion kann erst später registriert werden. Da der Startzeitpunkt und die Länge jeder Skipsection bekannt ist, kann neben der Reihenfolge sehr leicht bestimmt werden, welche vor der Fehlerinjektion liegen, welche danach und welche sie überdecken.

Abbildung 4.3 zeigt eine Unterteilung des Traces in vier verschiedene Skipsections. Es reicht, einen Listener nur für die dynamische Adresse der ersten Skipsection

(rot) anzulegen, da diese die kleinste dynamische Startinstruktion hat und folglich als erstes auslösen muss. Nach deren Anwendung würde erst ein Listener auf die folgende Skipsection (grün) registriert werden. Der Wert für den *Instruction-Couter* des *Listeners* berechnet sich dabei aus ihrer dynamischen Startadresse minus Startadresse und Länge der vorherigen Skipsection. Folgen, wie in diesem Beispiel, beide direkt aufeinander, so kann die nächste Skipsection, auch ohne das Registrieren eines *Listeners* und Kontrollflusswechsel zurück zum Simulator, direkt angewendet werden. Da der Zeitpunkt der Fehlerinjektion innerhalb der folgenden Skipsection (rosa) liegt, wird für diese niemals ein Listener registriert. Nach der Fehlerinjektion würde schließlich nur noch die letzte Skipsection (ocker) zu erwarten sein, sodass für diese ein Listener registriert wird. Da der dynamische Zeitpunkt durch die Fehlerinjektion oder Mechanismen zur Erkennung und Behebung von Fehlern, sich ändern kann, wird nach der Fehlerinjektion die statische Startadresse der folgenden Skipsections verwendet. Der *Listener* für die Fehlerinjektion kann dabei analog zu den Skipsections, ebenfalls erst später gesetzt werden. Dies kann aber je nach Implementierung etwas mehr Aufwand bedeuten, da der Zugriff und die Steuerung des *Listeners* und sein *Parent* zu beachten sind.

Dadurch, dass also nicht mehr alle aktiv sind, sondern nur noch eine bzw. ein Teil der Skipsections, ist die Anzahl vorhandener Skipsections nicht mehr so wichtig, da nicht mehr automatisch mehr Kontrollflusswechsel erfolgen. Mit potenziell beliebig vielen Skipsections sind weitere Strategien denkbar, die, durch deutlich mehr Skipsections, flexibler auf den jeweiligen Zeitpunkt der Fehlerinjektion angepasst werden können. So können z. B. überlappende Strategien sinnvoll sein, die zum einen einen großen Vorteil durch sehr lange Skipsections haben und trotzdem für jede mögliche Fehlerinjektion, nach deutlich kürzerer Zeit, den Beginn einer Skipsection erreichen.

Theoretisch könnte man so eine optimale Unterteilung finden, die für jede Fehlerinjektion nur genau eine Simulatorinstruktion ausführt. Dafür legt man für jeden Zeitpunkt eine Skipsection an, die vom Start bis genau dort hin geht und jeweils eine weitere Skipsection, die von jedem Zeitpunkt bis zum Ende des Traces geht. In der Praxis ist dies natürlich nicht möglich. Zum einen müssen die Skipsections noch immer eine Mindestgröße haben, zum anderen würden durch nicht überspringbare I/O-Instruktionen noch viele weitere Skipsections hinzu kommen. Damit würde der Aufwand zur Erstellung und der Overhead zur Speicherung, den erwarteten Gewinn rasch übersteigen. Die konkreten Ideen, zur Überlappung bei Unterteilungsstrategien, werden in Abschnitt 4.5 und Abschnitt 4.6 genauer untersucht.

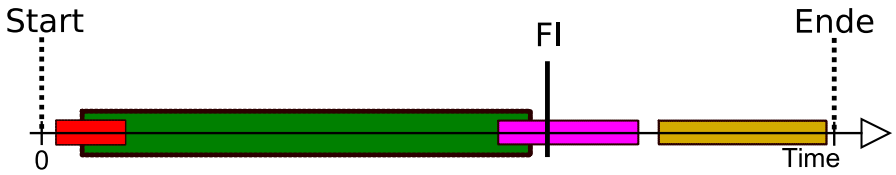

Abbildung 4.4: Da die rote und grüne Skipsection sich überlappen, würde das Anwenden der ersten, recht kurzen, das Anwenden der zweiten und deutlich längeren Skipsection verhindern, was den Laufzeitgewinn verringert.

4.3.1 Einschränkung verwendeter Skipsection: Shortest-Path

Die Ergänzung der möglichen Strategien auf Überlappung, erschwert allerdings die Auswahl der besten Skipsections. Um die Laufzeit möglichst gering zu halten, kann nicht immer davon ausgegangen werden, dass die nächste mögliche Skipsection auch angewendet werden soll. Zwar würde diese einen Laufzeitvorteil bringen, kann aber verhindern, dass eine andere Skipsection, die möglicherweise deutlich besser ist, angewendet wird. Abbildung 4.4 veranschaulicht dies an einem einfachen Beispiel:

Die erste Skipsection (rot) überspringt den Startzeitpunkt der deutlich längeren und damit auch besseren, zweiten Skipsection. Der Unterschied ist in diesem Beispiel dabei recht einfach zu erkennen, in der Praxis aber deutlich komplizierter. Für eine spätere Fehlerinjektion, die nicht innerhalb der dritten Skipsection (rosa) liegt, müsste zur Abwägung des optimalen Pfades, die Länge dieser Skipsection ebenfalls mitberechnet werden. Diese Überlegung lässt sich endlos fortsetzen, weil auch diese Skipsection eine andere wieder überlagern kann. Zudem sind nicht immer alle Skipsections so unterschiedlich lang, wie in diesem Beispiel.

Zur Bestimmung des optimalen Pfades wird daher ein „*Kürzeste Wege Algorithmus*“ für gewichtete Graphen nach Dijkstra verwendet [5], der einmalig beim Erstellen und Aufzeichnen der Skipsections angewendet wird und einen Skipgraphen erstellt. Wie in Abbildung 4.5 dargestellt, wird dafür eine Knotenmenge erstellt, die Start-Instruktion und End-Instruktion des Algorithmus und jeder Skipsection enthält. Als Bezeichner wird dabei jeweils die dynamische Instruktion verwendet, um Dopplungen in der Menge zu vermeiden. Für die Kantengewichte wird eine Adjazenzliste erstellt, die von jedem Knoten, also vorkommender dynamischer Instruktion, eine Kante zum nächsten Knoten, also der nächst höheren dynamischen Instruktion erstellt. Deren Kantengewicht ist die Anzahl der dynamischen Instruktionen, zwischen beiden Knoten. Zusätzlich wird für jede Skipsection eine Kante von deren Start zum Ende eingefügt, deren Kantengewicht sich aus den Kosten eines Kontrollflusswechsels und der Größe der Ein- und Ausgabe ergibt.

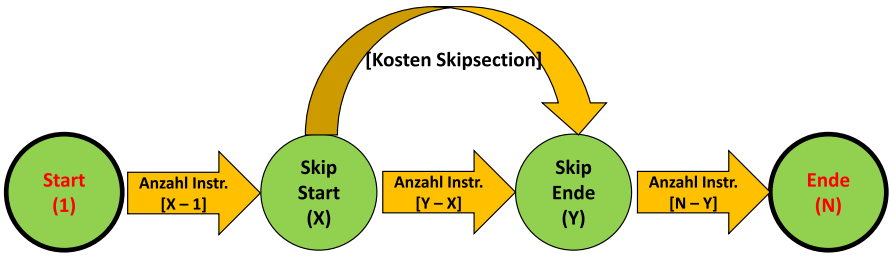

Abbildung 4.5: Zur Erstellung des Skipgraph werden die Anzahl der auszuführenden Instruktionen oder die Kosten der Anwendung einer Skipsection, als Kantengewicht verwendet.

Um diese Größen sinnvoll in Relation setzen zu können, müssen, wie in Abschnitt 5.2 genauer beschrieben, entsprechende Faktoren experimentell bestimmt und zur Berechnung des jeweiligen Kantengewichts verwendet werden. Anschließend wird eine vollständige Pfadsuche durchgeführt. Als Ergebnis erhält man für jede Kombination aus Start- und Endknoten eine Liste der verwendeten Skipsections in ihrer Reihenfolge.

Später genutzt und deswegen auch nur gespeichert und geladen, werden die Pfade, auf denen Skipsections verwendet werden und indirekt dadurch auch die enthaltenen Knoten. Durch die vollständige Pfadermittlung, lässt sich dies für jede mögliche Fehlerinjektion nutzen, indem man den Pfad vom Start zum Zeitpunkt der Fehlerinjektion nimmt und dann den Pfad von der, der Fehlerinjektion folgenden, Instruktion zum Ende. Da der Zeitpunkt der Fehlerinjektion allgemein nicht Teil des Graphen ist, werden die Skipsections verwendet, die auf dem Pfad vom Start zum letzten Knoten vor der Fehlerinjektion bzw. von der ersten Instruktion nach der Fehlerinjektion zum Ende, vorliegen.

4.3.2 Vor der Fehlerinjektion

Die Unterteilung in Skipsections vor und nach der Fehlerinjektion bietet teilweise weitere Vorteile für die Skipsections vor der Fehlerinjektion. Da Timeout- und End-Marker-Listener erst nach der Fehlerinjektion registriert werden und eine Überdeckung der Fehlerinjektion durch die Pfadbestimmung ausgeschlossen ist, kann die Überprüfung, ob ein Experiment-*Listener* das Auslösen blockiert, weggelassen werden. Auch ist sichergestellt, dass der Zustand sich nicht vom *Golden Run* unterscheidet. Die Überprüfung der Eingabe kann also ebenfalls entfallen. Beides spart Laufzeit und vergrößert dadurch den Nutzen der Skipsections. Allerdings sind diese Laufzeiten jeweils deutlich geringer als ein Kontrollflusswechsel oder Simulatorinstruktionen, sodass bei Bedarf die Überprüfung als *Sanity-Check* weiterhin enthalten bleiben kann.

4.3.3 Nach der Fehlerinjektion

Auch nach der Fehlerinjektion kann die Reihenfolge weiterhin als Grundlage dienen, nur einen Teil der verwendeten Skipsections zu aktivieren bspw. indem nur die jeweils als nächste erwartete Skipsection einen *Listener* registriert. Bei einem Großteil der Experimente kann angenommen werden, dass sich der Programmablauf nicht sehr stark verändert, vgl. Annahme in Unterabschnitt 4.4.2. Dementsprechend würde in vielen Fällen die Reihenfolge, in der die Skipsections auftreten, unverändert bleiben.

Allerdings wird dies natürlich nicht immer der Fall sein, sodass teilweise eine Skipsection nicht schalten kann. Wird also die statische Startadresse einer Skipsection erreicht, welche im *Golden Run* der tatsächliche Beginn wäre und der Vergleich der Eingabe scheitert als Konsequenz des injizierten Fehlers, so würde diese dennoch nicht aktiviert werden. Die im weiteren Ablauf erwarteten Skipsections wären dann unter Umständen nicht mehr die optimalen, sodass sich die Laufzeitverbesserung verringert. Viel problematischer ist jedoch die Tatsache, dass das Plugin nicht weiß, ob gerade der Beginn einer Skipsection erreicht wurde, oder nur ein anderes Vorkommen der statischen Adresse. Daher kann nicht genau bestimmt werden, wann die nächste Skipsection aktiviert werden muss. Würde weiterhin nur auf die erste statische Startadresse gewartet, so würde keine einzige Skipsection mehr angewendet werden. Um zu verhindern, dass mögliche Skipsections nicht angewendet werden, weil man vergeblich auf eine andere wartet, gibt es fünf verschiedene Möglichkeiten. Wichtig ist, dass in keinem Fall einfach alle aufgezeichneten Skipsections aktiviert werden. Die Wahrscheinlichkeit, dass eine Skipsection die vor der Fehlerinjektion aufgenommen wurde, danach durch eine zufällige Änderung wieder *matcht* und sich ein neue optimale Reihenfolge verwendeter Skipsections ergibt, ist sehr gering. Folgende Alternativen wären denkbar:

(1) Für jede Skipsection wird ein Zähler initialisiert, der jedesmal, wenn eine Überprüfung stattfindet, erhöht wird. Passiert dies zu oft, wird angenommen, dass zu viele dynamische Instruktionen vergangen sind und die Skipsection zum passenden Zeitpunkt nur nicht anwendbar war. Deswegen wird sie verworfen und die nächste Skipsection wird stattdessen aktiviert.

(2) Für die ersten beiden Skipsections wird ein *Listener* registriert. Matcht die erste Skipsection, wird diese ganz normal angewendet, anschließend verworfen und die nächste inaktive Skipsection registriert ihren *Listener*. Sollte aber die zweite Skipsection matchen, so werden beide nach der Anwendung verworfen, da davon auszugehen ist, dass die erste wegen der Fehlerinjektion nicht angewendet werden konnte, der Zeitpunkt dafür aber schon längst vorbei ist. Anschließend werden die nächsten beiden Skipsections aktiviert.

(3) Es werden alle Skipsections aktiviert, deren Auftreten auf dem optimalen Pfad nach der Fehlerinjektion erwartet wird. Dadurch soll verhindert werden,

eine mögliche Skipsection zu verpassen, weil man zu lange auf ihre Vorgänger gewartet hat. Ähnlich zur vorherigen Idee, sollte nach der Anwendung einer Skipsection jede vorherige deaktiviert werden.

(4) Es werden alle Skipsections aktiviert, deren dynamische Startadresse nach der Fehlerinjektion liegt. Dadurch, dass eine Skipsection nicht angewendet wird, kann ggf. eine andere genutzt werden, die auf dem optimalen Pad überdeckt worden wäre. Auch hier sollten, bei der Anwendung einer Skipsection, alle anderen deaktiviert werden, deren dynamische Startadresse kleiner ist.

(5) Es wird für jede Skipsection die nach der FI erwartet wird, ein *Listener* auf den dynamischen Startzeitpunkt registriert. Anschließend wird geprüft, ob die statische Adresse übereinstimmt. Ist dies der Fall, wird davon ausgegangen, das die Fehlerinjektion (noch) keinen zeitlichen Einfluss hatte und die Skipsection wird angewendet oder verworfen. Stimmt die statische Adresse mit dem Beginn der Skipsection nicht überein, wird ein *Listener* auf die statische Startadresse der Skipsection hinzugefügt. Sobald die statische Adresse erreicht wird, wird geprüft, ob die Skipsection hier startet. Den zeitlichen Unterschied kann man für die anderen Skipsection zum Startzeitpunkt hinzurechnen.

Alle Varianten treffen dabei eine Abwägung bezüglich der Anzahl der aktiven Skipsections. Je mehr Skipsections aktiv sind, um so größer ist die Wahrscheinlichkeit, dass jede dieser Skipsections auch zum richtigen Zeitpunkt aktiv ist und angewendet werden kann, falls der injizierte Fehler keinen Einfluss auf die Eingabe hatte. Der Nachteil ist natürlich, je mehr Skipsections aktiv sind, desto mehr statische Adressen lösen einen Kontrollflusswechsel aus. Da dieser erheblich länger dauert, als eine Simulatorinstruktion, sollten sie nur sehr selten geschehen, idealerweise einmal pro Skipsection. Welcher Ansatz jeweils am besten funktioniert, hängt mutmaßlich sehr stark vom verwendeten Benchmark, dem Zeitpunkt der Fehlerinjektion und den Skipsections ab, sodass hier keine Bewertung getroffen wird. Eine genauere Untersuchung würde aber in weitergehenden Arbeiten möglicherweise Sinn ergeben.

Wichtig ist, dass der Ansatz aus der ersten Variante, irgendwie auch in den restlichen umgesetzt wird und eine statische Startadresse einer Skipsection irgendwann deaktiviert wird. Zur Veranschaulichung zeigt Abbildung 4.6 das Histogramm für den Benchmark Dijkstra. Es werden 2423 verschiedene statische Instruktionen insgesamt über fünf Millionen mal ausgeführt, also im Schnitt jede etwas mehr als 2.000 mal. Die Verteilung ist aber nicht gleichmäßig, sodass einige (15) nur einmal ausgeführt werden, während die häufigsten drei jeweils 427.442 mal ausgeführt werden. Bei einem frühen Fehlerinjektionszeitpunkt, in Verbindung mit einer entsprechend extrem häufig vorkommenden Startadresse, können selbst verhältnismäßig kurze Abschnitte so viele Kontrollflusswechsel auslösen, dass der

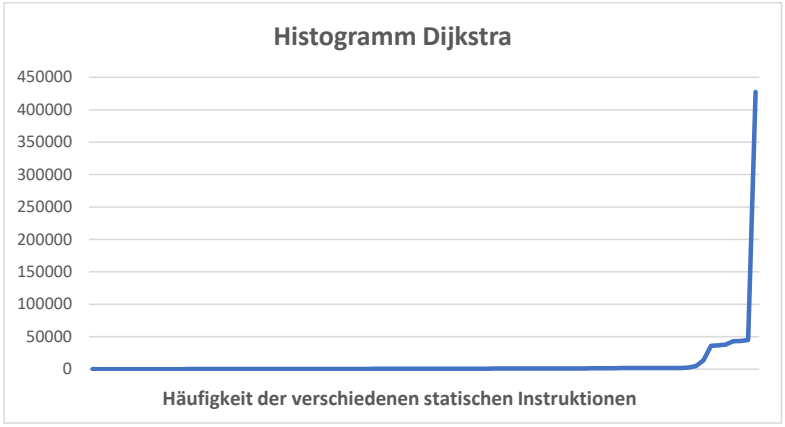

Abbildung 4.6: Häufigkeitsverteilung über alle statischen Instruktionen. Sehr wenige, sehr häufig vorkommende Instruktionen haben den größten Anteil an der Gesamtverteilung

Gewinn durch die Anwendung der Skipsection nicht mehr ausreicht, dies zu kompensieren. Im schlimmsten Fall kann keine Skipsection mehr angewendet werden und wir würden 427.442 mal unnötig eine Skipsection überprüfen. Im *Worst-Case* sogar noch mehr, wenn das Programm durch den Fehler in eine Endlosschleife gerät.

Ab einer bestimmten Grenze müssen Skipsections also immer deaktiviert werden, wie auch immer das jeweils genau gemacht wird. Bei der wievielten fehlerhaften Überprüfung das geschehen sollte, ist ebenso fraglich, wie die Variante zur Aktivierung der Skipsections. Egal bei welchem Wert man die Überprüfung abbricht, wenn die $X+1$-te Überprüfung erfolgreich wäre, ist die Einteilung für das konkrete Experiment sehr schlecht. Durch die große Häufigkeit ist es nicht unwahrscheinlich, dass eine der Skipsections bei einer dieser sehr häufigen Instruktionen startet. Würde diese bspw. beim sechsten Auftreten der statischen Adresse matchen, die Überprüfung aber nach fünf vergeblichen Überprüfungen abbrechen, würde man für nur einen weiteren Vergleich sehr viele Simulatorinstruktionen überspringen können. Gut abschätzbar ist dieser Zeitpunkt wegen des injizierten Fehlers aber nicht.

4.4 Kampagnen-Evaluator

Bei der Entwicklung neuer Strategien und der Anpassung verschiedener Parameter für diese, müssen die Ergebnisse miteinander verglichen werden. Hierfür jedes Mal eine vollständige Fehlerinjektionskampagne zu starten, wäre zeitlich sehr aufwändig, insbesondere, da es unter Umständen sein kann, dass sich bei unpassenden Benchmarks oder schlecht gewählten Parametern die Laufzeit im Vergleich zur normalen Kampagne sogar erhöht. Die Laufzeit mit einem Evaluator abzuschätzen, würde die Entwicklung neuer Strategien deutlich vereinfachen.

4.4.1 Abschätzung der Laufzeit

Der Kampagnen-Evaluator schätzt die Laufzeit für eine Strategie nur mit Hilfe des Traces und der Skipsections ab. Dazu wird berechnet, wie viele Simulatorinstruktionen, Kontrollflusswechsel, Vergleiche der Eingabe und Änderungen der Ausgabe zu erwarten sind. Diese Werte werden jeweils mit experimentell zu bestimmenden Faktoren für die Laufzeit multipliziert. Dabei wird angenommen, dass auch nach der Fehlerinjektion alle Skipsections angewendet werden können.

Die Anzahl der Simulatorinstruktionen ergibt sich dabei aus der Anzahl der dynamischen Instruktionen im Trace und der durch Skipsections übersprungenen Instruktionen. Dieser Wert wird mit der Anzahl der durchzuführenden Experimente multipliziert. Ähnliches gilt für die Ein- und Ausgabe, hier wird jeweils die Anzahl der aufgezeichneten Werte jeder angewendeten Skipsection als Grundlage verwendet. Für Kontrollflusswechsel muss zur Anzahl der angewendeten Skipsection noch die Anzahl der zu erwartenden fehlerhaften Überprüfungen gezählt werden. Dazu wird geprüft, wie häufig die statische Startadresse der Skipsection zwischen Aktivierung und Anwendung im Trace vorkommt.

4.4.2 Basisannahme der Umsetzung

Das Problem ist, dass Fehlerinjektionskampagnen durchgeführt werden, weil die Auswirkungen des injizierten Fehlers nicht bekannt sind. Um dessen Einfluss beurteilen zu können, müsste der komplette Systemzustand im Evaluator mitgeführt werden, inklusive des injizierten Fehlers und dessen Auswirkungen, was der Durchführung einer Fehlerinjektion entspricht. Um eine Abschätzung der Laufzeit mit deutlich geringerem Aufwand zu erreichen, wird der Evaluator unter der Annahme entwickelt, dass sich die Laufzeit und das Verhalten durch die Fehlerinjektion nur unwesentlich ändert.

Diese Annahme gründet sich auf der Beobachtung verschiedener Fehlerinjektionskampagnen, bei denen die Mehrheit der Experimente ohne Auswirkungen blieb. Tabelle 4.1 zeigt die Ergebnisse einer vollständigen Fehlerinjektionskampagne für den Benchmark *bin_sem3*, einen Benchmark aus den *eCos-Kernel-Test*

Fehlermodell	Variant	Benchmark	Result	Occurrences
RAM Gleichverteilt	mlqueue_baseline	bin_sem3	OK	2.779.122.696.545
RAM Gleichverteilt	mlqueue_baseline	bin_sem3	ERROR	377.544.865.695
RAM Read / Write	mlqueue_baseline	bin_sem3	OK	121.790
RAM Read / Write	mlqueue_baseline	bin_sem3	ERROR	35.018

Tabelle 4.1: Ergebnisse einer vollständigen Fehlerinjektionskampagne. Aus [16], Tab 2.1.

Benchmarks, die bereits in anderen Arbeiten vollständig mit FAIL* untersucht wurden [2].

Die fehlerhaften Ergebnisse Trap, Timeout und SDC wurden unter der Bezeichnung „ERROR“ zusammengefasst. Die Anzahl fehlerfreier Ergebnisse war dabei mehr als siebenmal so groß, wie die Anzahl der Experimente mit einem veränderten Ausgang. Der zweite Eintrag zeigt die Ergebnisse unter Verwendung eines anderen Fehlermodells, bei dem angenommen wurde, dass ein Fehler nicht gleichverteilt auftritt, sondern nur bei einem tatsächlichen Speicherzugriff. Dies entspricht damit eher den wirklich durchgeführten Experimenten, da für dieses Fehlermodell, die gleichen Ergebnisse genommen werden und die Gewichtung durch die Äquivalenzklassen ignoriert wird [16]. Die Anzahl der Ergebnisse ohne Fehler ist hier immer noch dreieinhalbmal so groß, wie die Anzahl der Ergebnisse mit Fehler. Allerdings musste der Teil der fehlerfreien Experimente nicht ausgeführt werden, bei denen der Fehler vor einem Schreibzugriff injiziert worden wäre, sodass das Verhältnis noch etwas schlechter sein wird.

Allerdings ist die Annahme auch in vielen Fehlerfällen noch verwendbar. Bei einer SDC sollte der Ablauf dem des *Golden Run* in etwa entsprechen, nur dass einer oder mehrere der Ausgabewerte verändert sind. In diesem Fall ähnelt der Programmablauf noch immer sehr dem *Golden Run* und nur eine Skipsection die wirklich ein fehlerhaftes Datenelement in der Eingabe hat, kann nicht angewendet werden.

Tritt ein Trap auf, so endet die Ausführung vorzeitig und die Abschätzung für dieses Experiment kann stark von der tatsächlichen Laufzeit abweichen. Gleiches gilt aber auch für die Fehlerinjektion ohne Skipsections, sodass die Kampagnenlaufzeit in beiden Fällen ähnlich falsch berechnet wird.

Einzig bei einem Timeout ist der Unterschied zwischen berechneter und realer Zeit bei der Verwendung von Skipsections deutlich höher. Ein Timeout wird meist durch eine Endlosschleife verursacht. Das heißt, die Laufzeit zur Ausführung von Simulatorinstruktionen ist mit und ohne Verwendung von Skipsections die gleiche und jeder Kontrollflusswechsel zur Überprüfung einer Skipsection verlängert die Laufzeit. Das gilt auch für den Fall, dass die Skipsection anwendbar ist und einen Teil der Instruktionen der Endlosschleife überspringt. Dies würde nur bedeuten,

dass mehr Durchläufe bis zum Timeout stattfinden können.

4.4.3 Limitierung der Genauigkeit

Die Annahme, dass sich das Verhalten und dadurch auch die Laufzeit nach der Fehlerinjektion nicht wesentlich verändert, ist natürlich nur sehr begrenzt richtig. Das Ziel ist , auf die komplette Kampagne bezogen, die Laufzeit so gut abzuschätzen, dass Strategien sinnvoll miteinander verglichen werden können, also die erwarteten Kontrollflusswechsel, ausgeführten Simulatorinstruktionen und der Mehraufwand für das Prüfen und Ändern des Maschinenzustands, möglichst gut abgeschätzt werden kann. Insbesondere die Überlegungen aus Unterabschnitt 4.3.3, welche Skipsections wann nach der Fehlerinjektion ihren *Listener* registrieren, kann nicht sinnvoll einbezogen werden. Die limitierenden Aspekte sind dabei hauptsächlich folgende:

(1) Der in Unterabschnitt 4.3.3 diskutierte Unterschied nach der Fehlerinjektion nur die als nächstes erwartete Skipsection oder alle zu aktivieren, kann nicht sinnvoll abgeschätzt werden. Der Vorteil dabei, alle Skipsections zu aktivieren, ist, dass eine nicht oder zu spät angewendete Skipsection keine anderen Skipsections blockieren kann. Da der Evaluator von einem unveränderten Verhalten ausgeht, wird angenommen, dass auch dann alle zum richtigen Zeitpunkt schalten, wenn nur jeweils die nächste aktiv ist. Der mögliche Vorteil kann also nicht erkannt werden. Der Nachteil, dass die statischen Startadressen häufiger auftauchen, kann hingegen simuliert werden.Die abgeschätzte Laufzeit muss also schlechter werden.

(2) Wie ebenfalls in Unterabschnitt 4.3.3 beschrieben, muss eine Deaktivierung möglich sein, auch wenn keine der Skipsections angewendet wird. Dafür wird in dieser Arbeit, genau wie in der zugrunde liegenden Arbeit [19], ein Zähler verwendet, der nach einer festgelegten Anzahl von Überprüfungen einer Skipsection diese deaktiviert. Dadurch kann es aber passieren, dass eine Skipsection deaktiviert wird, obwohl sie im späteren Verlauf noch angewendet werden könnte. Muss ein Teil der Instruktionen z. B. immer im Simulator ausgeführt werden, weil I/O-Instruktionen mögliche Skipsection blockieren, kann die Startadresse in diesem Teil zu häufig auftreten. Da unter der grundsätzlichen Annahme für den Evaluator, solch ein Zähler gar nicht benötigt würde, weil alle Skipsections wie geplant angewendet werden könnten, kann der Grenzwert zur Deaktivierung beliebig hoch sein und es muss davon ausgegangen werden, dass er zumindest ausreichend hoch ist, um das Deaktivieren einer noch anwendbaren Skipsection zu verhindern.

(3) Manche Fehlertoleranzverfahren widersprechen der Annahme, dass sich der

Programmablauf nicht ändert, vollständig. Ein mögliches Korrekturverfahren wäre Folgendes:

- Die Daten werden geladen und eine Prüfsumme gebildet.
- Die eigentliche Arbeit wird auf den Daten durchgeführt, z. B. eine Sortierung.
- Eine erneute Prüfsumme der Daten wird gebildet und mit der ursprünglichen Prüfsumme verglichen.
- Unterscheiden sich beide Werte, werden die Daten neu geladen und die Arbeit erneut durchgeführt.

Die Laufzeit wäre damit annähernd doppelt so lang und das Verhalten spätestens nach der Detektierung des Fehlers völlig verändert. Ist die Berechnung sehr aufwändig, kann das Bilden der Prüfsumme sogar regelmäßig wiederholt und Zwischenergebnisse gespeichert werden. In diesem Fall würde sich das Verhalten, unter Umständen sogar deutlich vor dem normalen Ende, sehr stark vom *Golden Run* unterscheiden.

(4) Manche Strategien zielen neben der Beschleunigung durch eine Skipsection, auch darauf ab, dass die Anwendung durch die Fehlerinjektion möglichst selten verhindert wird. Dazu kann z. B. der Startzeitpunkt so gewählt werden, dass die Eingabemenge möglichst gering ist und damit die Wahrscheinlichkeit, dass der Fehler Auswirkungen auf den aktuellen Abschnitt hat, möglichst klein ist. Dieser Vorteil kann ebenfalls nicht erkannt werden.

Zusammenfassend heißt das, dass der Laufzeitgewinn von Strategien systematisch überschätzt wird. Strategien die also nur leichte Verbesserung im Vergleich zu einer Ausführung ohne Skipsections erreichen, sollten genauer untersucht werden. Auch alle Ansätze, die versuchen nicht simulierte Einschränkungen möglichst gut zu umgehen, können auch trotz schlechterer Werte in der Evaluation mehr Laufzeitgewinn erzielen, als andere Strategien.

4.5 Auswahlstrategien

Die Unterteilung des Traces in verschiedene Skipsections entscheidet maßgeblich, welche Verbesserung oder Verschlechterung die Anwendung von Skipsections bringt. Dabei sollte ein Verfahren möglichst allgemein einen großen Laufzeitgewinn erreichen und nicht nur für einen bestimmten Benchmark. Dafür werden in den folgenden Abschnitten eine Reihe von Strategien entworfen, die die in der Problemanalyse identifizierten Anforderungen zu erfüllen versuchen.

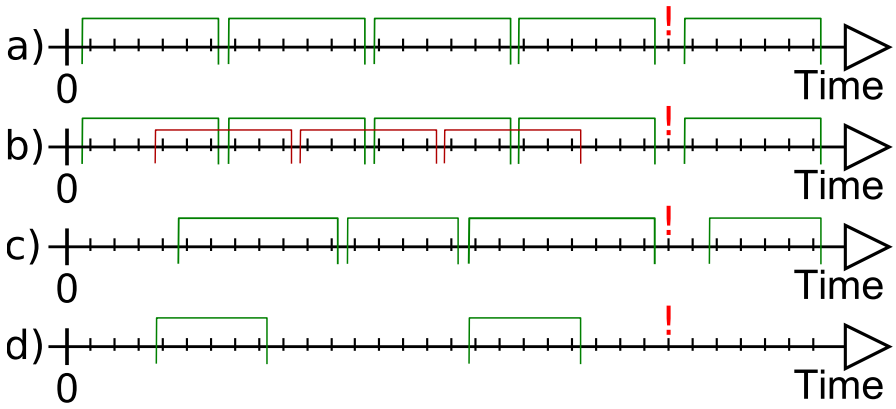

Abbildung 4.7: Die Unterteilung eines Beispiel-Trace mit a) *Equidistant*, b) *Overlapping*, c) *MinInputSet* und d) *RareStart*.

4.5.1 Equidistant

Verwendete Parameter: Mindestlänge

Die Unterteilung des Trace erfolgt in gleich große und aufeinanderfolgende Skipsections, die Mindestlänge ist also automatisch auch die Maximallänge. Soweit keine I/O-Instruktion das verhindert sollte der komplette Trace abgedeckt werden und verschiedene Skipsections sich nicht überschneiden. Abbildung 4.7 a) zeigt ein einfaches Beispiel mit der Mindestlänge sechs und einer I/O-Instruktion (rotes Ausrufezeichen).

Der Vorteil dieser Strategie ist, dass die Programmierung und Anwendung recht einfach ist. Andere Verfahren nutzen weitere Informationen zur Unterteilung, müssen für diese aber auch einen größeren Aufwand betreiben. Dieser Overhead sollte sich durch eine verbesserte Laufzeit rechtfertigen. Die *Equidistant*-Strategie dient damit auch als Referenz für die anderen Strategien.

4.5.2 Overlapping

Verwendete Parameter: Mindestlänge und Versatz

Analog zur vorherigen Strategie ist die Mindestlänge auch die Maximallänge und alle Skipsections haben die gleiche Größe. Allerdings werden sie hier überlappend festgelegt, sodass der Trace teilweise mehrfach überdeckt ist. Als Versatz wird Anzahl der Instruktionen bezeichnet, nach der die nächste Skipsection innerhalb der aktuellen Skipsection startet. Angegeben wird dieser Wert als Anteil der Mindestlänge. Abbildung 4.7 b) zeigt den Unterschied zur *Equidistant*-Strategie. Als Versatz wurde dabei die Hälfte der Mindestlänge gewählt. Sobald diese erreicht wurde, fängt eine neue Skipsection (hier rot) an.

Die Idee dahinter ist, dass lange Skipsections grundsätzlich mehr Gewinn bringen, weil mehr Instruktionen übersprungen werden. Je länger eine Skipsection

aber ist, um so mehr Fehlerinjektionszeitpunkte befinden sich innerhalb dieser und können die Skipsection entsprechend nicht nutzen. Dadurch, dass schon nach der Hälfte der Skipsection die nächste startet, müssen für viele Injektionszeitpunkte deutlich weniger Simulatorinstruktionen ausgeführt werden, bis der Beginn einer Skipsection erreicht wird. Bei der Wahl eines kleineren Wertes werden noch weitere Skipsections der gleichen Länge erzeugt und der Trace noch häufiger überdeckt und noch eher eine mögliche Skipsection erreicht. Allerdings muss aufgepasst werden, dass die Anzahl erzeugter Skipsections nicht zu groß wird.

4.5.3 MinInputSet

Verwendete Parameter: Mindestlänge und Maximallänge, möglich auch eine Maximalgröße der Eingabe

Bei diesem Ansatz wird versucht, die Eingabemenge einer Skipsection möglichst gering zu halten. Dafür wird der Trace rückwärts durchlaufen und die Eingabemenge durch Speichernutzung und verwendete Register berechnet. Sobald die Mindestlänge erreicht wurde, wird der Trace bis zur Maximallänge weiter durchlaufen. Die Instruktion, bei der die Eingabemenge am geringsten ist, wird als Startpunkt festgelegt und ab dort wieder nach einer möglichst geringen Eingabemenge gesucht. Wurde eine Maximalgröße festgelegt, wird dabei nur eine Skipsection erzeugt, wenn die Größe der Eingabe unter diesem Wert liegt.

Abbildung 4.7 c) zeigt eine mögliche Unterteilung für die Mindestlänge fünf und die Maximallänge zehn. Da vom Ende aus nach Skipsections gesucht wird, kann es passieren, dass einige Instruktionen zu Beginn in keiner Skipsection liegen.

4.5.4 RareStartInstruction

Verwendete Parameter: Mindestlänge

Bei dieser Strategie sollen vor allem die unnötigen Kontrollflusswechsel nach der Fehlerinjektion vermieden werden, indem Skipsections nur bei Instruktionen starten, deren statische Adresse selten im Trace vorkommt. Wird solch eine Instruktion erreicht, beginnt eine Skipsection fester Länge, falls sie durch keine I/O-Instruktion überdeckt wird. Anschließend wird nach der nächsten möglichen Skipsection gesucht.

Die Idee dabei ist, dass die Kosten für einen Kontrollflusswechsel um ein Vielfaches höher sind, als die Kosten für den Vergleich einer Eingabe, oder die Ausführung von Simulatorinstruktionen. Die resultierenden Skipsections sind dabei nicht zusammenhängend und über den kompletten Trace verteilt, wie Abbildung 4.7 d) skizziert. Der Nachteil ist, dass unter Umständen der größte Teil der Simulatorinstruktionen nicht überdeckt wird und folglich nur sehr begrenzte Laufzeitgewinne erzielt werden können.

4.5.4.1 Varianten der RareStartInstruction-Strategie

Ein entscheidender Parameter ist die Frage, was eine seltene Start-Instruktion überhaupt ist. Ab welcher Häufigkeit eine Instruktion nicht mehr als Start verwendet werden sollte, ist dabei nicht sehr leicht zu beantworten und sehr abhängig vom Benchmark. Bei einem sehr kleinen Grenzwert werden nur sehr wenige Skipsections erstellt, häufig nur eine oder sogar gar keine, wenn an den wenigen möglichen Stellen eine I/O Instruktion dies verhindert. Die verwendeten Skipsections bringen zwar einen verhältnismäßig sehr großen Nutzen, decken aber nur kleine Teile des Trace ab und können entsprechend wenig Laufzeitgewinn generieren. Im Rahmen dieser Arbeit werden folgende Ansätze betrachtet:

Durchschnitt Bei dieser Variante der Strategie, wird jedesmal eine neue Skipsection gestartet, wenn die statische Instruktion seltener als der Durchschnitt vorkommt. Innerhalb einer Skipsection wird dabei nicht nach weiteren Startpunkten geprüft.

Feste Werte Es werden nur die besten N Instruktionen zur Erstellung von Skipsections verwendet. Dabei wird N vom Anwender übergeben und sollte in Abhängigkeit von Länge des Traces und Mindestlänge der Skipsections gewählt werden, damit der Trace möglichst vollständig überdeckt wird.

Proportional zur Anzahl der Instruktionen Statt eines festen Wertes, übergibt der Anwender einen relativen Wert, sodass z. B. die besten 10% aller möglichen Instruktionen verwendet werden.

Überlappend/Beste zuerst Unabhängig von den möglichen Startadressen, kann die Suche nach Skipsections unterschiedlich den Trace prüfen. Bei der überlappenden Variante, wird jedes mal eine Skipsection erzeugt, wenn eine mögliche Startadresse erzeugt wird. In der Variante aus Unterabschnitt 4.5.4, werden die Skipsections nur nacheinander erzeugt. So kann es passieren, dass eine mögliche Startadresse erreicht wird, die verhältnismäßig noch recht häufig auftritt und eine deutlich seltenere Startadresse deshalb nicht mehr verwendet werden kann. Die Variante Beste zuerst, geht die möglichen Startadressen nacheinander durch und prüft, ob damit in noch nicht überdeckten Bereichen eine Skipsection erzeugt werden kann.

4.5.5 Random

Verwendete Parameter: Mindestlänge, Maximallänge, Abstand zwischen dem Beginn verschiedener Skipsections

Bei der *Random*-Strategie wird, wie der Name schon andeutet, die Einteilung zufällig vorgenommen. Bestimmte Parameter müssen dabei natürlich noch immer durch den Programmierer oder Nutzer festgelegt werden, die Ergebnisse sollten

aber eine große Varianz von unterschiedlichen Skipsections ergeben. In der konkreten Umsetzung im Rahmen dieser Arbeit wurde dabei nur die Mindestlänge als Parameter übergeben und alle weiteren Entscheidungen zufällig, aber mit unterschiedlichen Wahrscheinlichkeiten getroffen. Der Beginn der ersten Skipsection, sowie später aller weiteren, wird durch eine Zufallszahl bestimmt, wenn der Tracedurchlauf oder eine Skipsection starten. Die folgendende Skipsection kann dabei noch überlappend in der aktuellen starten, aber auch mit einigem Abstand erst später. Die Länge ist immer ein zufälliger Offset zur Mindestlänge. Eine weitere Untersuchung der Rahmenbedingungen, wie bei den vorherigen Strategien, erfolgt nicht.

Diese Strategie dient in erster Linie nur zur Beurteilung anderer Ansätze und zum Rechtfertigen des dortigen Mehraufwands. Bleiben die Ergebnisse bei dieser Strategie in einem ähnlichen Bereich wie andere optimierte Ansätze, so scheinen die unter Umständen sehr aufwändigen Berechnungen keinen bedeutenden Effekt zu haben. In dem Fall muss geprüft werden, ob der betriebene Aufwand sich lohnt. Zudem bietet die Strategie eine gute Möglichkeit alle Bereiche der Implementierung auf Programmierfehler zu testen, weil hier Sonderfälle auftreten, die andere Unterteilungen verhindern würden.

4.6 Querschneidende Belange

Querschneidende Belange sind solche, die zusätzlich zu jeder zugrunde liegenden Strategie abgewogen werden müssen und allgemein auf Basis von jeder angewendet werden können.

4.6.1 Mindestgröße

Ein wichtiges Kriterium bei der Unterteilung ist die Länge der jeweiligen Skipsection. Dabei kann der Begriff Mindestlänge auch irreführend sein, da viele Strategien eine konstante Länge haben bzw. die Maximallänge ein Vielfaches der Mindestlänge ist.

In einer optimalen Einteilung würde als wirkliche Mindestlänge der Wert ausreichen, ab dem die Ungleichung $n_{SimInstr} \cdot t_{SimInstr} > t_{BP} + n_{Eingabe} \cdot t_{Eingabe} + n_{Ausgabe} \cdot t_{Ausgabe}$, (Mit n_X gleich Anzahl von X und t_X gleich Dauer für X) erfüllt ist.

In der Praxis könen aber auch viele Überprüfungen ohne die Anwendung einer Skipsection ausgeführt werden. Die entsprechenden Mehrkosten müssen durch die verwendeten Skipsections ebenfalls ausgeglichen werden und sind außerdem stark von deren Anzahl abhängig. Allgemein wird wohl zu erwarten sein, dass dieser Wert auch von der Länge des verwendeten Benchmarks abhängt.

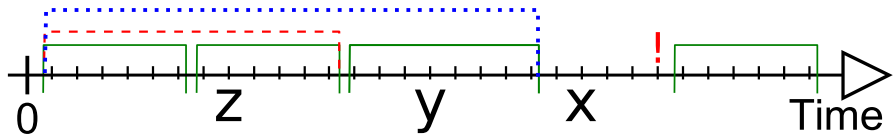

Abbildung 4.8: Bestehende Skipsections werden hierarchisch zu längeren Skipsections zusammengefasst.

4.6.2 Hierarchie

Der Ansatz Skipsections hierarchisch zu verbinden, versucht den Vorteil sehr langer Skipsections mit der Flexibilität kürzerer Skipsection zu verbinden. Dazu werden Skipsections um die jeweils nachfolgenden verlängert, sodass zu Beginn der ursprünglichen Skipsection mehrere unterschiedlich lange Skipsections starten. Je nach Zeitpunkt der Fehlerinjektion kann dann so weit wie möglich gesprungen werden.

Abbildung 4.8 soll dies an einem kurzen Beispiel verdeutlichen. Bei der ursprünglichen Unterteilung wurden vier verschiedene Skipsections (grün) erstellt. Zusätzlich wird für jede Skipsection geprüft, ob weitere direkt folgen. In diesem Beispiel wird für die erste Skipsection eine weitere erstellt, die die Nachfolgende enthält (rot) und eine weitere (blau), die alle drei aufeinander folgenden Skipsections umfasst. Liegt der Zeitpunkt der Fehlerinjektion außerhalb der verbundenen Skipsections, z. B. zum Zeitpunkt x, kann die größtmögliche Skipsection angewendet werden. Bei anderen Fehlerinjektionszeitpunkten, hier y und z, wird einer der kleineren Skipsections verwendet.

Der Kern hinter dem Verfahren, Teilmengen der Skipsections einer Basisstrategie zu neuen Skipsections zusammenzufassen, kann auf jede Strategie angewendet werden, die mindestens zwei aufeinanderfolgende Skipsections hat. Da MinInputSet und Equidistant diese Vorgabe soweit möglich immer erfüllen, eignen diese sich besonders gut für den hierarchischen Ansatz. Für zwei aufeinander folgende Skipsections wird eine neue Skipsection erzeugt, die zum gleichen Zeitpunkt wie die erste startet, aber so lang wie beide zusammen ist.

Bei einem Trace, der nicht durch I/O-Instruktionen unterbrochen wird, kann die Anzahl der Skipsections sehr schnell stark wachsen und nicht mehr skalieren, wenn alle möglichen Kombinationen erstellt werden. Dieser Ansatz, im Weiteren als *Hierarchical All* bezeichnet, kann also nicht immer verwendet werden, sodass auch Varianten betrachtet werden müssen, die nur einen Teil der Kombinationen verwenden. Drei verschiedene Ansätze werden im Rahmen dieser Arbeit untersucht.

Hierarchical All Alle möglichen Kombinationen von Skipsections werden erstellt. Das bietet den größten Laufzeitgewinn eines hierarchischen Ansatzes, ist aber in der Praxis nicht mehr unbedingt anwendbar, da viel zu viele Skipsec-

tions erzeugt werden. Wird ein Trace in n aufeinander folgende Skipsections der Länge x unterteilt, so würden zu Beginn n verschiedene Skipsections starten (der Länge $1x$, $2x$, $3x$, ..., nx), mit Beginn der nächsten Skipsection entsprechend noch $n-1$ verschiedene. Dadurch ergibt sich für die Gesamtanzahl mit der Gaußschen Summenformel: $\frac{1}{2} \cdot (n^2 + n)$.

Abbildung 4.9 a) zeigt ein einfaches Beispiel, bei dem die ursprünglichen vier Skipsections um sechs neue ergänzt werden. Zur Übersicht sind die Skipsections in verschiedene Level gegliedert, wobei auf Level n alle Skipsections zusammengefasst werden, die aus n Skipsections der Basisstrategie erzeugt werden. Hier also drei neue Skipsections auf Level 2, zwei auf Level 3 und eine auf Level 4, also gesamt: $\frac{1}{2} \cdot (n^2 + n) = \frac{1}{2} \cdot (4^2 + 4) = \frac{1}{2} \cdot 20 = 10$

Hierarchical Level Um eine quadratisch wachsende Anzahl von neuen Skipsections zu vermeiden, werden in diesem Ansatz nur noch Skipsections miteinander kombiniert, wenn sie sich auf dem gleichen Level befinden. Das heißt, für alle ursprünglichen Skipsections wird jede Kombination geprüft und bezogen auf unser Beispiel würden nur noch Skipsections der Länge $2^l \cdot x$ erstellt werden.

In Abbildung 4.9 b) wird der Unterschied dargestellt. Auf Level 1 befinden sich alle Skipsections der Basisstrategie, die jeweils zu zweit zu einer neuen Skipsection auf Level 2 kombiniert werden. Diese werden dann aber nicht mehr mit den Skipsections der Basisklasse verglichen, sondern nur noch untereinander kombiniert. Dadurch werden auf Level 3 keine neuen Skipsections erzeugt.

Der Unterschied zu *Hierarchical All* wächst dabei mit der Anzahl der Skipsections, da immer nur $log_2(n)$ der möglichen n Level verwendet werden, die Anzahl liegt also im Bereich von $O(n \cdot log_2(n))$ statt bei $O(n^2)$.

Hierarchical Bin Bei der Kombination auf dem gleichen Level, kann jede Skipsection noch immer sowohl mit ihrem Nachfolger, als auch ihrem Vorgänger, eine neue Skipsection der Länge $2x$ bilden. Dies wird hier noch weiter eingeschränkt, indem alle Skipsections nur noch paarweise verglichen werden, also keine Überlappung auf dem nächsten Level.

Abbildung 4.9 c) verdeutlicht den Unterschied zum vorherigen Ansatz. Auf Level 2 werden nur noch zwei statt drei Skipsections erzeugt, da die zweite Skipsection der Basisstrategie nicht mehr mit der Dritten verglichen wird. Das reduziert die maximale Anzahl neuer Skipsections noch weiter, sodass auf Level x nur noch maximal $\frac{1}{x} \cdot n$ Skipsections erzeugt werden können, also gesamt $2n - 1$, was in $O(n)$ liegt.

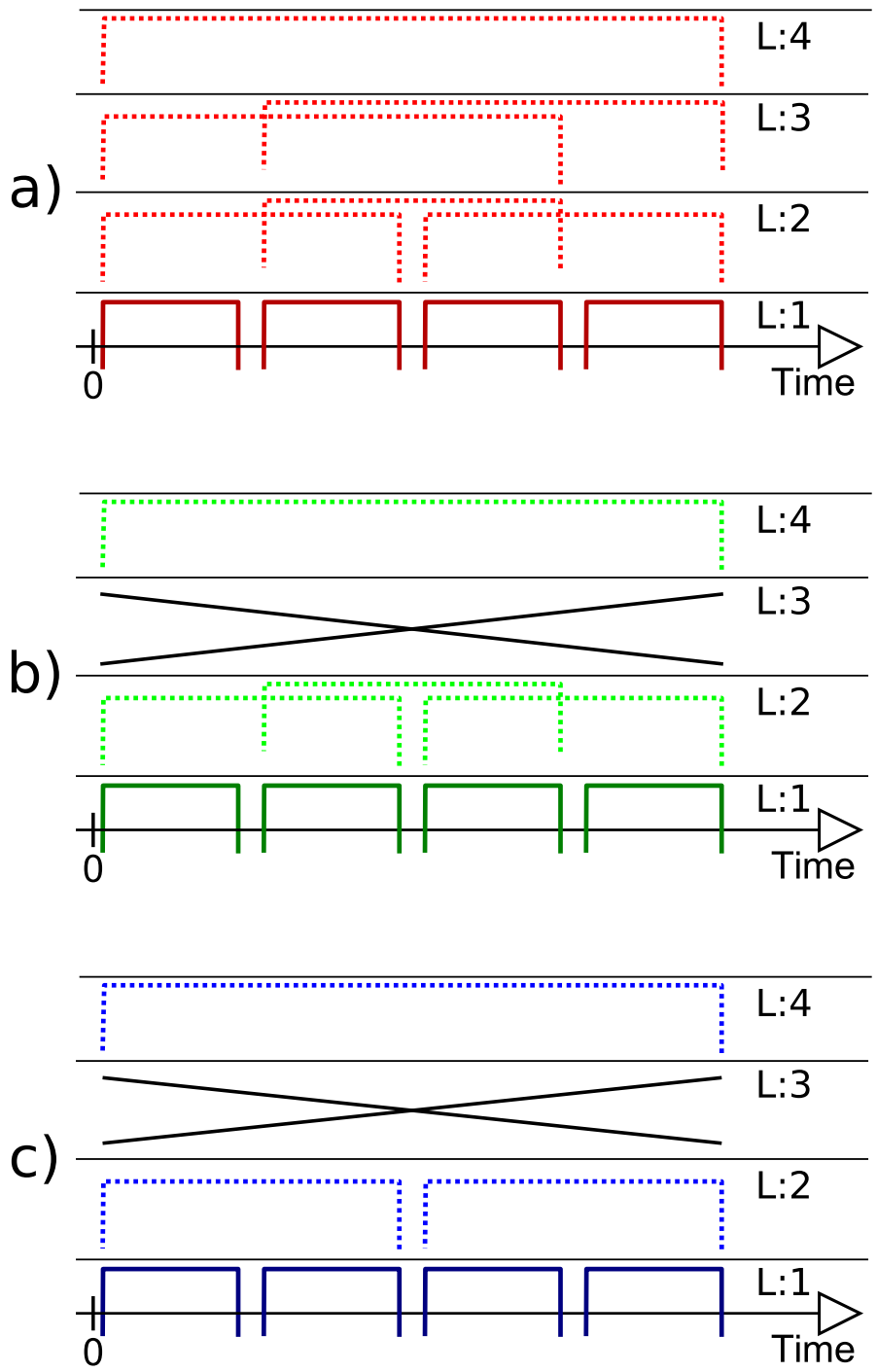

Abbildung 4.9: Verschiedene Ansätze wie stark Skipsections hierarchisch zusammengefasst werden im Vergleich.

4.6.3 Re-Use-Check

Der *Re-Use-Check* ist kein Ansatz zur Unterteilung, sondern verändert nur die Aufzeichnung bestehender Skipsections. Dabei wird die Ausgabe reduziert, indem für jeden Wert geprüft wird, ob dieser auch nach der Skipsection nochmal verwendet wird.

Dabei orientiert der Ansatz sich an der Verwendung eines Stacks. Einmal geschriebene Speicherbereiche können im Verlauf des Programms ihre Bedeutung verlieren, sobald der Stack abgebaut wird. Während der Aufzeichnung einer Skipsection ist allerdings nicht abzusehen, ob geschriebene Werte später nochmal gelesen werden oder nur innerhalb der Skipsection verwendet werden. Dies wird nachträglich mit Hilfe des Traces geprüft. Dieser Ansatz kann also die Ausgabe für jede Skipsection verringern und auf alle Basisstrategien angewendet werden.

Bezüglich der Laufzeit kann durch dieses Verfahren kein Nachteil entstehen. Da die Ausgabe weniger Werte enthält, muss nur ein kleinerer Teil des Systemzustands geändert werden. Es besteht allerdings die Möglichkeit, dass das Kriterium der Ergebnistreue verletzt wird. In sehr seltenen aber nicht auszuschließenden Fällen kann das Programmverhalten durch den injizierten Fehler so verändert werden, dass eine eigentlich nicht mehr benutzte Speicherstelle gelesen wird. Der nicht geänderte Wert kann dann einen Einfluss auf das Ergebnis haben.

Den gleichen Effekt haben auch manche Fehlerkorrekturverfahren. Werden diese verwendet, muss der Ansatz grundsätzlich weggelassen werden. Ein Beispiel wäre folgendes Vorgehen:

(1) Für alle (wichtigen) Daten wird eine Prüfsumme und eine Kopie gespeichert.

(2) Bei der Verwendung wird die Prüfsumme erneut gebildet und mit der gespeicherten Prüfsumme verglichen.

(3) Stimmen berechnete und gespeicherte Prüfsumme nicht überein, liegt ein Fehler vor und die Kopie wird verwendet.

(4) Unter Umständen wird vorher die Prüfsumme der Kopie mit der Prüfsumme der Daten und der gespeicherten Prüfsumme verglichen.

Bei der Verwendung dieses Verfahrens, würden die Werte der Kopien im *Golden Run* niemals gelesen werden, sodass deren Werte aus der Aufzeichnung gelöscht werden. Das komplette Fehlerkorrekturverfahren könnte dadurch dann nicht mehr sinnvoll angewendet werden.

4.7 Zusammenfassung

In diesem Kapitel wurde die Idee zur Verbesserung des ursprünglichen Verfahrens näher erläutert und ein Entwurf der Umsetzung vorgestellt. Das größte bisher

ungenutzte Potenzial liegt dabei in der Optimierung der Unterteilungsstrategien, die bisher noch nicht näher betrachtet wurden. Das nächste Kapitel befasst sich mit der Implementierung des geänderten Verfahrens und der neuen Strategien.

5 Implementierung

In diesem Kapitel werden die Details der Implementierung gezeigt, die sich nicht direkt aus dem Entwurf ergeben.

5.1 Strategien

Alle Strategien sind in der `skipsecrecorder`-Plugin integriert und wurden in über 3300 Zeilen C++ Quellcode implementiert[1].

Die Unterteilung des Traces in verschiedene Skipsections erfolgt durch unterschiedliche Strategien, die als Eingabe einen Vektor mit *TraceEvents* übergeben bekommen. Neben der konkreten Unterteilung durch die zentrale Methode `cut()` werden hier aber weitere Funktionen bereitgestellt, die unabhängig von der jeweiligen Umsetzung der Unterteilung sind.

Die Implementierung erfolgt dabei vor allem unter Verwendung von Vererbung, um die unterschiedlichen querschneidenden Belange aus Abschnitt 4.6 bei Bedarf jeder Strategie zur Verfügung zu stellen.

In Abbildung 5.1 wird die Klassenhierarchie veranschaulicht. Basisklasse ist für alle die `CuttingStrategy`, die gleichbleibende Funktionen zur Verfügung stellt, wie das Anlegen der Aufzeichnungen oder das Speichern der Skipsections. Weitere Belange werden dann jeweils durch eine neue Klasse implementiert, wie die Hierarchie oder der *Reuse-Check*.

Die unterschiedlichen Strategien erben dann jeweilis von der `CuttingStrategy` oder einer davon abgeleiteten Klasse. Die Verwendung erfolgt wie in Listing 5.1 gezeigt. Die Strategie erbt von `ReuseStrategy` und dadurch nur indirekt von `CuttingStrategy`. Zur Aufzeichnung wird dann die abgeleitete Methode verwendet (Zeile 2). Diese wiederum verwendet die ursprüngliche Methode zum Aufzeichnen (Zeile 8) und führt anschließend die zusätzlichen Arbeitsschritte durch (Zeile 9).

Listing 5.1: Integration eines querschneidenden Belangs, hier: *Reuse*.

```
1 bool CuttingStrategy_XYZ::record(std::vector<Trace_Event> *trace) {
2     bool result = ReuseStrategy::record(trace);
3     createSkipGraph();
```

[1] Effektive Quellcodezeilen (Ohne Leerzeilen und Kommentare) mit *cloc* ermittelt: http://cloc.sourceforge.net.

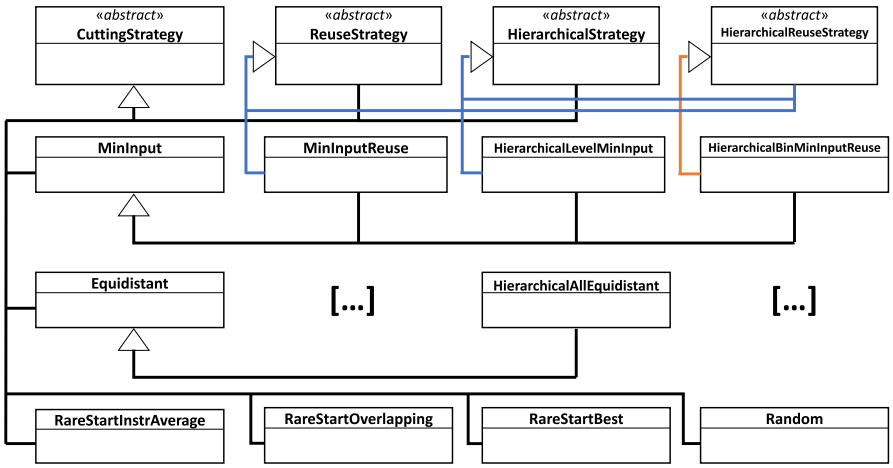

Abbildung 5.1: Klassenhierarchie der Strategie-Klassen.

```
4       saveToFile_recs();
5       return result;
6   }
7   bool ReuseStrategy::record(vector<Trace_Event> *trace) {
8       if(CuttingStrategy::record(trace)) {
9           uint64_t erg = checkRecs(trace);
10          return true;
11      }
12      return false;
13  }
```

Eine weitere Besonderheit zur Verringerung der Laufzeit ist die Möglichkeit bereits erstellte Skipsections als Grundlage für andere Strategien zu verwenden. Die `HierarchicalBinEquidistantReuse`-Strategie[2] kann nicht nur die Funktionalität anderer Klassen übernehmen, sondern die durch die Basisklasse erstellten Skipsections benutzen. Dazu muss nur die Basisstrategie als Parameter übergeben werden.

Listing 5.2: Skipsections der Basisstrategie laden.

```
1   uint64_t Hier_B_Equi_Reuse::cut(std::vector<Trace_Event> *trace) {
2       if(hasStrategyForSkips) {
3           // Lade bereits erstellte Skipsections einer anderen
                Strategie
4           skipsecs = vector<SkipSec>(*strategy->getSkipSecs());
5       } else {
6           // Weiterer Programmcode zur Ausfuehrung des Cuttings
7       }
8       addHierarchicalSkipsBin();
9       saveToFile_cuts();
```

[2]Bei der Programmierung als *Hier_B_Reuse* abgekürzt.

```
10      return skipsecs.size();
11  }
```

In diesem Fall muss die Methode `cut()` nicht mehr ausgeführt werden, was je nach Ansatz sehr aufwändige Berechnungen zur Folge hätte. Das Ergebnis wird anschließend weiter angepasst, in diesem Beispiel durch das Hinzufügen weiterer hierarchischer Skipsections.

Eine weitere wichtige Anpassung die zur gleichzeitigen Verwendung mehrerer Strategien gebraucht wird, ist die Anpassung des Dateipfads zur Speicherung. Hier wird die als Parameter übergebene Mindestlänge und die ID der Strategie mit integriert, sodass alle gleichzeitg erzeugten Skipsections auch gespeichert werden können, also statt z. B. *bfs.skip bfs_80000_2.skip*.

5.2 Zeitmessung und Gewichtung

Zur Gewichtung bei der Erstellung des *SkipGraph* und bei der Berechnung der Laufzeit im Kampagnen-Evaluator muss die Ausführungszeit einer Simulatorinstruktion, eines Kontrollflusswechsels, des Prüfens einer Eingabe und des Änderns einer Ausgabe bestimmt werden. Dazu wurde ein neues Experiment, `time-test`, implementiert, welches die Zeitmessung durchführt.

Dabei wurde vereinfacht angenommen, dass die Laufzeit für alle Simulatorinstruktionen gleich lang ist und analog für die Ein- und Ausgabe, dass auch hier nicht relevant ist, welcher Wert genau betrachtet wird.

Simulatorinstruktion Dafür würde die Ausführungszeit des Programms ohne Änderungen oder weitere Plugins gemessen und anschließend durch die Anzahl der dynamischen Instruktionen geteilt.

Kontrollflusswechsel Analog zur Messung bei den Simulatorinstruktionen, wurde das Programm ohne Fehlerinjektion durchgeführt, zusätzlich aber ein neues Plugin registriert. Das `BPSingleStep`-Plugin verwendet dabei einen *Listener* auf jede Instruktion, sodass jedes Mal ein Kontrollflusswechsel stattfindet. Anschließend wird die Kontrolle direkt wieder an den Simulator abgegeben. Die gemessene Laufzeit enthält damit genau so viele Kontrollflusswechsel wie Simulatorinstruktionen. Die Differenz zur vorherigen Messung kann durch diese Anzahl geteilt werden, um die Dauer eines einzelnen Kontrollflusswechsels zu ermitteln.

Ein- und Ausgabe Für die letzte Messung wird ein zweites Plugin verwendet, das eine beliebige Strategie nutzt, um bereits erzeugte Skipsections zu laden. Analog zum späteren Skipping wird zum Startzeitpunkt einer beliebigen Fehlerinjektion gesprungen und die Zeitmessung begonnen. Dabei wird einmal die benötigte Zeit für den Vergleich der Eingabe und einmal für die

Änderung des Zustands durch die Ausgabe gemessen. Um Ungenauigkeiten bei der sehr kurzen Laufzeit zu verhindern, wurde statt der Dauer einer vollständigen Transformation, die Zeit für 1000 Wiederholungen gemessen.

Alle Messungen wurden mehrfach wiederholt und auf mehreren Benchmarks durchgeführt, ohne das sich große Abweichungen ergaben. Bei der Implementierung wurde der jeweilige Mittelwert verwendet. Grundsätzlich kann hier aber eine große Abhängigkeit von der Hardware bestehen, sodass ggf. die Messungen für andere Hardware wiederholt werden sollten.

5.3 Steuerung durch Kommandozeilen Parameter

Zur einfachen Nutzung in realen Fehlerinjektionskampagnen kann das Unterteilen und Anwenden über Kommandozeilen-Parameter gesteuert werden

generic-tracing Alle Schritte zur Vorbereitung der realen Fehlerinjektionsexperimente sind im Experiment `generic-tracing` gekoppelt. Das beinhaltet die Unterteilung des Traces in Skipsections, das Anfertigen dazugehöriger Aufzeichnungen und die Abschätzung der Laufzeit. Einzelne Arbeitsschritte können dabei durch entsprechende Parameter angepasst bzw. ganz abgeschaltet werden.

- Abschalten des Tracings: `--skipsec-creation-mode`

  Werden später weitere Strategien ergänzt, wäre ein erneutes Erstellen des Traces unnötig, da sich hier keine Änderungen ergeben, entsprechend kann dies durch den Parameter abgeschaltet und der bereits gespeicherte Trace verwendet werden.

- Abschalten der Unterteilung: `--no-skip-creation`

  Die Abschätzung der benötigten Laufzeit kann auch für bestehende Skipsections erfolgen, sodass das erneute Unterteilen des Traces entfallen kann. Insbesondere nach der Änderung der Implementierung oder der Anpassung verwendeten Werte für die Laufzeiten, z. B. Dauer einer Simulatorinstruktion, würde sonst ein unnötiger Mehraufwand entstehen.

- Abschalten des Kampagnen-Evaluators: `--no-campaign-evaluation`

  Skipsections können für den realen Einsatz in einer Fehlerinjektionskampagne auch ohne Laufzeitabschätzung erzeugt und verwendet werden.

- Mindestlänge: `--skipsec-mincut` X, mit X = 32 Bit Ganzzahl.

  Wichtiger Parameter, der durch den Anwender entsprechend des Benchmarks gesetzt werden sollte.

- Wiederholungen: `--mincut-repetition` X, mit X = 32 Bit Ganzzahl.

  Der Parameter erleichtert den Test verschiedener Mindestlängen für die ausgewählten Strategien. Bis zum X-fachen der verwendeten Mindestlänge werden alle Vielfachen ebenfalls als Parameter verwendet.

- Trace-Limit: `--trace-limitation` X, mit X = 32 Bit Ganzzahl.

  Begrenzt den Trace auf die ersten X Instruktionen. Das erleichtert den Vergleich verschiedener Benchmarks, da deren Länge das Ergebnis nicht mehr unterschiedlich beeinflussen kann.

generic-skip-experiment Zur Verwendung des `SkippingPlugins` ergeben sich kaum Änderungen, es muss nur der Dateipfad für die Skipsections angegeben werden. Auch für die dazugehörigen Aufzeichnungen und den `SkipGraph`, falls diese sich nicht an der gleichen Stelle befinden. Dazu wird der Parameter `--skipsec-file` verwendet. Für die anderen Dateien wird ohne Angabe des Dateipfads nur die Endung der Datei angepasst.

5.4 Zusammenfassung

In diesem Kapitel wurden die weiteren Implementierungsdetails vorgestellt, die sich noch nicht aus dem Entwurf ergeben, oder die zur Verwendung der Implementierung genutzt werden müssen. So kann FAIL* Fehlerinjektionskampagnen mit Skipsections verwenden. Die daraus resultierenden Ergebnisse werden im folgenden Kapitel vorgestellt.

6 Auswertung

In diesem Kapitel soll die Nützlichkeit der in Kapitel 4 entworfenen Strategien untersucht und die Nützlichkeit des Kampagnen-Evaluators bewertet werden. Nach einer Festlegung der Evaluationskriterien und genutzten Benchmarks sollen die Laufzeitänderungen durch die jeweiligen Skipsections genauer untersucht werden.

6.1 Kriterien

Die Kriterien zur Beurteilung der unterschiedlichen Strategien wurden in Abschnitt 3.1 erarbeitet. Da das Ziel der Verwendung von Skipsections die Beschleunigung von Fehlerinjektionskampagnen ist, sollte das Augenmerk beim Vergleich der verschiedenen Strategien auf diesem Punkt liegen, wobei die restlichen Faktoren nicht ignoriert werden dürfen.

Dabei sind nicht alle aufgestellten Kriterien von der jeweiligen Strategie beeinflusst, sondern ergeben sich aus der allgemeinen Umsetzung:

Ergebnistreue Die Ergebnistreue bei der Idee der Skipsections ist inhärent. Die Anwendung passt den Zustand so an, dass er mit dem Zustand identisch ist, der durch Ausführen der übersprungenen Instruktionen erreicht worden wäre. Da der Simulator deterministisch ist, folgt aus einem identischen Systemzustand auch ein gleiches Ergebnis. Nur durch Anwendung des *Reuse-Check* kann sich der Zustand des Systems nach der Anwendung einer Skipsection unterscheiden, sodass dieser Aspekt in Unterabschnitt 6.3.9 nochmal separat betrachtet wird.

Zur Überprüfung der Ergebnistreue wurden für alle Strategien reale Kampagnen mit *Samples* durchgeführt. Für die gleichen Fehlerinjektionen werden die Experimente ohne Skipsections wiederholt und die Ergebnisse auf mögliche Abweichungen verglichen. Dabei wurde die Ergebnistreue verifiziert.

Generalität Die Anpassung an das genutzte *Backend*, hier *Bochs*, ist nur sehr gering, sodass eine Portierbarkeit ohne großen Aufwand möglich ist. Die notwendigen Anpassungen des Zielprogramms bestehen dabei aus nur sehr wenigen Zeilen Programmcode, von denen die meisten für die grundsätzliche Verwendung mit FAIL* notwendig sind. Die untersuchten Benchmarks und konkreten Einschränkungen werden in Abschnitt 6.2 beschrieben.

Nutzerfreundlichkeit Zur Nutzung von Skipsections muss das `generic-tracing` Experiment nur in der Art angepasst werden, dass die Strategien ausgewählt werden, mit denen eine Einteilung erfolgt. Die weitere Nutzung lässt sich über die Kommandozeile sehr einfach auf die jeweiligen Bedürfnisse anpassen, also z. B. bei der Verwendung mehrere Strategien die Aktivierung der Evaluation. Zur Anwendung bei der später durchgeführten Kampagne muss nur der Speicherort der gewünschten Skipsections über die Kommandozeile ausgewählt werden.

Die qualitativen Kriterien werden also unabhängig von der konkreten Strategie erfüllt, deren Beurteilung dann nur noch unter Berücksichtigung der beiden Aspekte Effektivtät bei der Verringerung der Laufzeit und Overhead durchgeführt wird.

6.2 Verwendete Zielprogramme

Zur Untersuchung der jeweiligen Strategien wurden verschiedene Zielprogramme ausgewählt, die unterschiedliche Anwendungsfelder abdecken. Alle wurden bereits zur Fehlerinjektion mit FAIL* verwendet, sodass die notwendigen Vorbereitungen bereits erledigt wurden. Das beinhaltet im Wesentlichen nur die Ergänzung des Stop-Symbols `FAIL_FINISHED()`.

Die Algorithmen lassen sich in drei unterschiedliche Gruppen einteilen und sind in Tabelle 6.1 zusammengefasst.

Als Betriebssystem wurde für alle Benchmarks *eCos* [10] verwendet, ein konfigurierbares Betriebssystem, das für den Bereich der eingebetteten Systeme entwickelt wurde. Dabei gelten für alle Zielprogramme folgende Einschränkungen:

Timerinterrupt Die Anwendung von Skipsections kann durch Timerinterrupts gestört werden. Um dies zu verhindern wäre ein tiefer Eingriff in den Simulator notwendig, sodass solche Zielprogramme aus zeitlichen Gründen und im Sinne der Portierbarkeit ausgeschlossen wurden. Bei allen genutzten Zielprogrammen wurden Interrupts zu Beginn durch einen Inline-Assembler-Befehl deaktiviert.

Gelitkommaarithmetik Wegen der speziellen Register, die 80 Bit breit sind, ist eine Umsetzung mit FAIL* schwierig und mit der verwendeten Version nicht möglich.

Memory-Mapped I/O Der zeitliche Unterschied durch die Beschleunigung könnte ein anderes Verhalten bedeuten, sodass strukturell solche Zielprogramme ausgeschlossen werden.

Gruppe	Benchmark	Kurzeschreibung
MiBench [6]	blowfish	Algorithmus zur Ver- und Entschlüsselung.
	crc	Zyklische Redundanzprüfung.
	dijkstra	Graphalgorithmus zum Finden kürzester Wege.
	rijndael	Algorithmus zur Ver- und Entschlüsselung.
	sha	Hash-Algorithmus.
	stringsearch	Wortsuche auf *Strings*
Parboil [20]	bfs	Graphalgorithmus zum Finden kürzester Wege.
	sad	Algorithmus zum Vergleich von Bildern
	histo	Erstellung eines Histogramms.
Sortier-algorithmen [16]	bubblesort gnomesort heapsort mergesort quicksort selectionsort shellsort	Verschiedene Standard-Sortieralgorithmen mit unterschiedlichen Eingaben, sortiert und unsortiert

Tabelle 6.1: Kurze Beschreibung aller verwendeter Benchmarks.

6.3 Untersuchung der Effektivität von Strategien

Das Ziel der Verwendung von Skipsections ist die Beschleunigung von Fehlerinjektionskampagnen, sodass Änderungen der Laufzeit der wichtigste Gradmesser für die Güte einer Strategie ist und im Folgenden für jede Strategie genauer untersucht wird. Der erzeugte Overhead für die Skipsections wird abschließend als einschränkender Faktor untersucht.

6.3.1 Aufbau

Zur Untersuchung der Laufzeit werden zum einen die Ergebnisse des Kampagnen-Evaluators verwendet, zum anderen aber auch Fehlerinjektionskampagnen durchgeführt. Die Ergebnisse des Kampagnenevaluators bieten bei der Beurteilung den entscheidenden Vorteil, dass der Trace auf eine feste Länge begrenzbar ist, sowohl für die Erstellung der Skipsections, als auch für die Abschätzung der Laufzeit. Das bietet den Vorteil, einer deutlich größeren Vergleichbarkeit der zugrunde liegenden Benchmarks.

Um diese Ergebnisse zu verifizieren und den Kampagnen-Evaluator an sich zu bewerten, wurden zusätzlich Fehlerinjektionskampagnen durchgeführt. Der Fehlerraum wurde dabei nicht vollständig abgedeckt, sondern nur jeweils 1.000 *Samples*

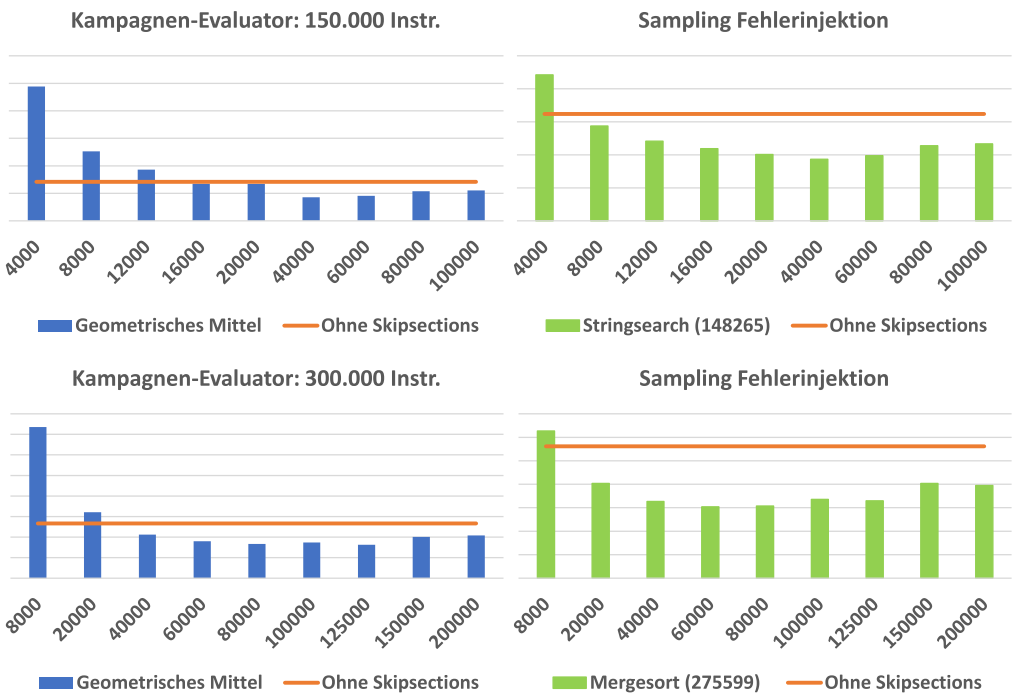

Abbildung 6.1: Die mit dem Kampagnen-Evaluator berechneten Ergebnisse im Vergleich zu gemessenen Ergebnissen für einen Benchmark ähnlicher Größe.

zufällig ausgewählt und zur Messung[1] der Laufzeit verwendet.

6.3.2 Equidistant-Strategie

Die *Equidistant*-Strategie unterteilt den Trace nacheinander in Skipsections mit der als Parameter übergebenen Mindestlänge. Dabei zeigt sich, dass dieser Ansatz auch ohne aufwändiges Verfahren eine deutliche Verbesserung der Laufzeit erreichen kann, eine Verschlechterung aber auch möglich ist.

Die Ergebnisse werden in den Diagrammen in Abbildung 6.1 zusammengefasst. Die beiden Diagramme auf der linken Seiten (blau) zeigen die mit dem Evaluator ermittelten Laufzeiten bei einer Begrenzung auf 150.000 bzw. 300.000 Instruktionen. Die Diagramme auf der rechten Seite (grün) zeigen die gemessenen Ergebnisse für einen Benchmark, der jeweils eine ähnliche Größe hat: Stringsearch mit 148.265 Instruktionen und Mergesort mit 275599 Instruktionen.

Dabei zeigt sich, dass eine Mindestlänge von ca. 10% des Benchmarks notwendig ist, um eine Verbesserung zu erreichen und im Mittel eine Länge zwischen 25%

[1] Als Laufzeit wird der mit `clock()` ermittelten Wert der Prozessorzeit verwendet. Schwankungen durch Auslastung der Hardware beeinflussen das Ergebnis so weniger stark.

und 30% die besten Ergebnisse erzielen. Dies wird durch beide Beispielmessungen bestätigt, bei denen die geringste Laufzeit mit 40.000 bzw. 80.000 Instruktionen Mindestlänge erreicht wurde. Bei Stringsearch wurde eine Verbesserung von 16,19 Sekunden auf 9,32 Sekunden für 1000 Fehlerinjektionen erreicht, also eine Einsparung von 42%. Für Mergesort sogar eine Verbesserung um 46%, mit einer Verringerung von 28,08 auf 15,19 Sekunden. In der Abschätzung wurden sogar Verbesserungen von über 70% erreicht, was sich bei den Messungen nicht bestätigen ließ. Das liegt daran, dass zur Evaluation die in Unterabschnitt 4.4.2 beschriebene Annahme getroffen wurde, dass nach der Fehlerinjektion jede Skipsection noch angewendet werden kann. Das ist in der Realität nicht immer so. Auch wurden die besten Ergebnisse für den Benchmark *rijndael* errechnet, aber in der limitierten Form, sodass eine tatsächliche Messung nur für eine deutlich größere Mindestlänge möglich wäre.

Neben einem absoluten Minimum zeigt sich bei den Ergebnissen, dass es jeweils auch ein oder mehrere weitere lokale Minima gibt. Für die Größen Benchmarks, also 300.000 Instruktionen bzw. Mergesort, liegt dies bei 125.000. Das liegt daran, dass bei der Mindestlänge 80.000 Instruktionen noch jeweils drei Skipsections bei der Unterteilung erstellt wurden, ab 100.000 jeweils nur noch zwei. Dadurch werden deutlich weniger Simulatorinstruktionen durch Skipsections überdeckt, nur noch 200.000 statt wie vorher 240.000. Eine Vergrößerung dieser Skipsections kann daher die Laufzeit nochmal etwas stärker verringern.

6.3.3 Overlapping

Die Strategie ist ähnlich zur *Equidistant*-Strategie und unterteilt den Trace in Skipsections fester Länge, welche als Parameter übergeben wird. Nach der Hälfte jeder Skipsection startet zusätzlich noch eine weitere Skipsection, damit die Distanz zwischen Fehlerinjektionspunkt und Start einer Skipsection durchschnittlich kleiner wird.

Abbildung 6.2 zeigt aber nur in den Randfällen eine stärkere Abweichung, im Bereich der optimalen Mindestlänge ähneln sich die Ergebnisse sehr. Das liegt daran, dass der Effekt der Überlappung bei konstanter Länge aller Skipsections verschwindet. Wird eine der zusätzlichen, überlappenden Skipsections nach der Fehlerinjektion angewendet, so werden danach in etwa die gleichen Sprünge durchgeführt, nur etwas verschoben, der Vorteil tritt also nur einmal zu Tage. Zusätzlich besteht die Möglichkeit, dass dieser Vorteil sich am Ende wieder ausgleicht. Ist die letzte vorhandene Skipsection eine der ursprünglich erzeugten, so müssen nach der Anwendung der überlappenden Skipsections die gleiche Anzahl vorher eingesparter Simulatorinstruktionen zusätzlich am Ende ausgeführt werden.

Bei einer geringen Mindestlänge, also in dem Bereich, in dem equidistante Skipsections eine Verschlechterung der Laufzeit bewirken, gleichen die eingesparten Simulatorinstruktionen je Skipsection die zusätzlichen Kontrollflusswechsel nicht

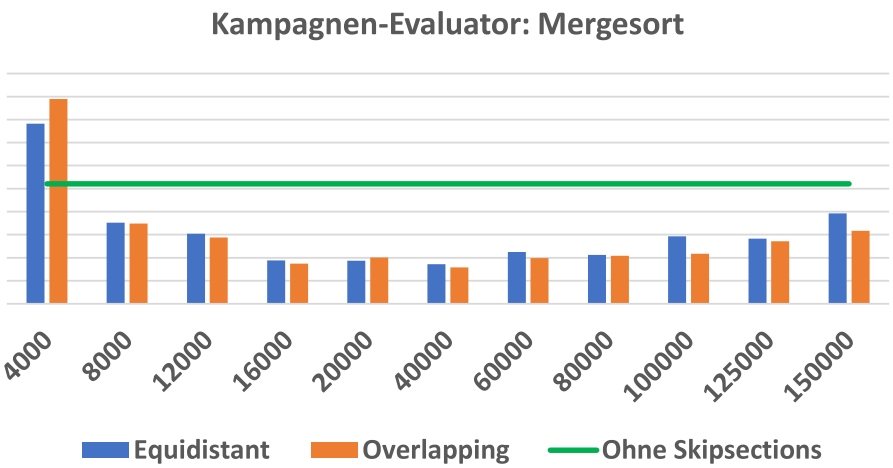

Abbildung 6.2: Vergleichen der Laufzeiten für die Strategien *Overlapping* und *Equidistant*.

aus. Das heißt, die zusätzlichen statischen Startinstruktionen, die durch neue Skipsections dazu kommen, verschlechtern die Laufzeit noch weiter.

Bei sehr großen Mindestlängen hingegen überdeckt die *Overlapping*-Strategie mehr Simulatorinstruktionen und ist dadurch effektiver. Bei einer Mindestgröße von 150.000 Instruktionen und der Länge des Benchmarks Mergesort von 275.599 Instruktionen, kann immer nur eine Skipsection angewendet werden, sodass es keinen Unterschied bedeutet, wo diese liegt. Bei einer Fehlerinjektion in den ersten 75.000 Instruktionen würden bei der *Equidistant*-Strategie also keine Skipsection verwendet werden können, bei *Overlapping* hingegen schon.

6.3.4 Random-Strategie

Die Bewertung *der Random*-Strategie gestaltet sich sehr schwierig, da die erstellten Skipsections sehr stark variieren und die Frage ist, womit diese verglichen werden sollten. Grundsätzlich ergibt sich als Ergebnis eine Mischung aus *Equidistant* und *Overlapping* mit verschiedenen Größen der Skipsections. In der verwendeten Implementierung können die Skipsections bis zu 9.000 Instruktionen länger sein, als die per Parameter angegebene Mindestlänge. Dadurch ist die Strategie bei kleinen Mindestlängen effektiver als *Equidistant*, weil die verwendeten Skipsections näher am Optimum liegen.

In diesem Bereich ähneln die Ergebnisse sich, da mit steigender Mindestlänge die feste Änderung der Größe nur noch weniger stark ins Gewicht fällt. Da die Strategie aber aufwändiger zu implementieren ist und im Bereich der zu verwendenden Mindestlänge zumindest etwas schwächer ist, besteht allgemein kein Grund zu deren Verwendung bei realen Fehlerinjektionskampagnen.

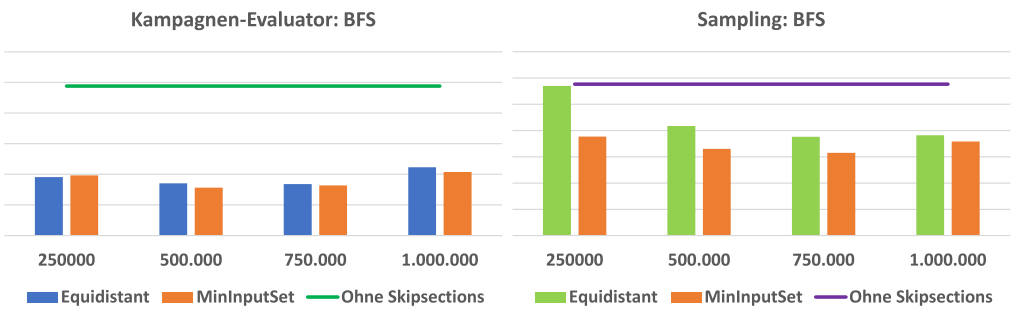

Abbildung 6.3: Die mit dem Kampagnen-Evaluator berechneten Ergebnisse im Vergleich zu gemessenen für den Benchmark *bfs*.

6.3.5 MinInputSet-Strategie

Die *MinInputSet*-Strategie unterteilt den Trace vom Ende aus beginnend in aufeinander folgende Skipsections unterschiedlicher Länge. Dabei wird jeweils bis zur doppelten Mindestlänge ein Startpunkt gesucht, bei dem die später zu prüfende Eingabe der Skipsection möglichst gering ist.

Wie Abbildung 6.3 zeigt, ist die Laufzeit dabei sowohl in den berechneten Abschätzungen, als auch bei den gemessenen Werten besser als die Ergebnisse der *Equidistant*-Strategie. Die Verbesserung gegenüber einer Kampagne ohne Skipsections liegt alledings deutlich unter den aus der Abschätzung zu erwartenden Ergebnissen.

Die zugrunde liegende Idee, dass eine kleinere Eingabe auch schneller verglichen werden kann, hat dabei nur sehr begrenzt einen Einfluss auf das Ergebnis. Für den Beispiel-Benchmark *bfs*[2] wurden bei einer Mindestlänge von 500.000 Instruktionen neun Skipsections erzeugt, deren Eingabe zusammen 816.419 Bytes groß ist. Für die *Equidistant*-Strategie wurden bei gleichem Benchmark und gleicher Mindestlänge ebenfalls neun Skisections erstellt, deren Eingabe allerdings nur 807.792 Bytes groß ist. Dass die Eingabe größer ist, liegt daran, dass die Unterteilung von unterschiedlichen Enden des Traces startet und dadurch völlig unterschiedliche Skipsections erzeugt werden.

Die Unterschiede bei der Laufzeit ergeben sich viel mehr durch das unterschiedliche Vorgehen bei der Einteilung, die je einen großen Vor- und Nachteil hat:

Nachteile Mit steigender Mindestlänge der Skipsections wächst auch die Anzahl an Simulatorinstruktionen, die am Ende nicht mehr durch eine Skipsection überdeckt werden. Da bei dieser Strategie vom Ende aus begonnen wird, befinden sich die restlichen Instruktionen, die zu wenige für eine eigene Skipsection sind, am Beginn des Traces. In diesem konkreten Beispiel fängt die erste Skipsection für den Benchmark *bfs* erst nach 370.982 Instruktionen an.

[2]Der Benchmark ist 4.994.496 dynamische Instruktionen lang und damit einer der Größten, die in dieser Arbeit untersucht wurden.

Strategie	Mindestlänge	SkipSecs	SkipSecs nach FI	Deaktivierung
MinInputSet	250.000	15030	5259	2849
MinInputSet	500.000	6634	2486	1299
MinInputSet	750.000	4256	1728	664
MinInputSet	1.000.000	2605	1292	398
Equidistant	250.000	10517	0	7506
Equidistant	500.000	4993	0	3213
Equidistant	750.000	3218	0	1997
Equidistant	1.000.000	2194	0	1132

Tabelle 6.2: Übersicht, wie häufig bei den Strategien *Equidistant* und *MinInputSet* nach der Fehlerinjektion noch Skipsections angewendet wurden.

Diese Instruktionen liegen aber zumeist vor der Fehlerinjektion, also zu einem Zeitpunkt, in dem keine unnötigen Vergleiche oder Kontrollflusswechsel durchgeführt werden und ist damit deutlich effizienter, was zur Folge hat, dass die Laufzeit bis zur FI im Vergleich schlechter wird.

Vorteile Zwischen einer Skipsection und einer nicht überspringbaren I/O-Instruktion entstehen viele Abschnitte mit Simulatorinstruktionen, die nicht überdeckt sind, aber nicht ausreichen, um eine eigene Skipsection zu bilden. Durch die variable Länge bei *MinInputSet* wird die Anzahl dieser Instruktionen reduziert. Der entscheidende Punkt dabei ist, dass durch die kleineren Lücken zwischen zwei Skipsections die Chance deutlich geringer wird, dass eine noch anwendbare Skipsection zu früh deaktiviert wird.

Das wird aber besonders bei langen Benchmarks mit großen Skipsections zu einem enormen Problem nach der Fehlerinjektion. In Tabelle 6.2 ist die Nutzung vor und nach der Fehlerinjektion dargestellt. Die schlechtere Laufzeit ergibt sich hauptsächlich dadurch, dass in diesem konkreten Fall bei der *Equidistant*-Strategie keine einzige Skipsection nach der Fehlerinjektion noch genutzt wird.

Das Problem Skipsections zu früh zu deaktivieren hat nach der Fehlerinjektion einen sehr großen Einfluss auf die Laufzeit und ist der Grund, warum bei der Laufzeitmessung die Ergebnisse deutlich schlechter sind,durch den Evaluator berechnet. Das Problem wird in Abschnitt 6.4 noch genauer betrachtet.

6.3.6 RareStartInstruction-Strategie

Ziel dieser Strategie ist es möglichst effektive Skipsections auszuwählen, ohne dabei unbedingt auch möglichst große Teile des Traces zu überdecken. Als effektiv wird dabei eine Skipsection angenommen, deren statische Startadresse im Trace selten vorkommt, also auch nur selten Kontrollflusswechsel und Vergleiche auslöst,

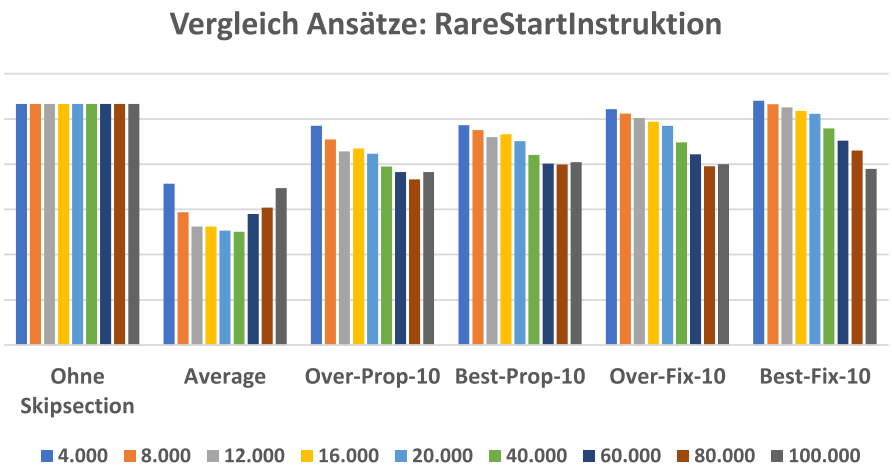

Abbildung 6.4: Vergleich der unterschiedlichen Umsetzungen der *RareStartInstruktion*-Strategie am Beispiel des Geometrischen Mittels aller Benchmarks mit mindestens 300.000 Instruktionen.

idealerweise nur zu dem Zeitpunkt, an dem die Skipsection der Erwartung nach starten müsste.

Die Effektivität der einzelnen Variationen am Beispiel des geometrischen Mittelwerts aller Benchmarks mit mindestens 300.000 Instruktionen wird in Abbildung 6.4 für verschiedene Mindestlängen dargestellt. Dabei zeigt sich, dass die Orientierung am Durchschnittswert ein gutes Maß ist. Für die getesteten Benchmarks würden hier die besten Ergebnisse erzielt.

Die anderen Umsetzungen sind allerdings noch über den Parameter (in der Untersuchung jeweils zehn) anpassbar, sodass auch eine Verbesserung noch möglich ist. Dafür sind aber für jeden Benchmark Untersuchungen notwendig, die bei der Verwendung des Mittelwerts entfallen können.

Ebenfalls zu beachten ist, dass es bei jeder Umsetzung dazu kommen kann, dass nur sehr wenige Teile des Traces überdeckt oder gar keine Skipsections erstellt werden. Unter diesen Umständen ist dann natürlich kein Laufzeitgewinn zu erwarten, sodass die Anzahl der Skipsections vor der Verwendung geprüft werden sollte. Gute Ergebnisse erzielt eine Überdeckung von mehr als zwei Dritteln.

Bei überlappenden Skipsections kann es auf der anderen Seite ebenfalls sehr leicht passieren, dass zu viele Skipsections erzeugt werden und dadurch ein zu großer Overhead entsteht. Dies wird in Unterabschnitt 6.3.10 noch genauer betrachtet.

Zur Bewertung des Ansatzes wurde die *AverageRareStartInstruktion*-Strategie für den Benchmark *dijkstra* mit der *Equidistant*-Strategie verglichen. Bei diesem Benchmark wurden durch beide Strategien die gleiche Anzahl an Skipsections erzeugt, sodass der Einfluss der Häufigkeit der statischen Startadresse ersichtlicht wird.

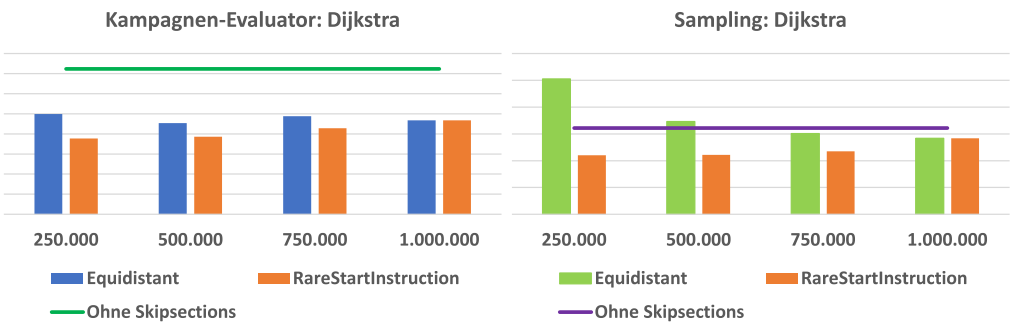

Abbildung 6.5: Die mit dem Kampagnen-Evaluator berechneten Ergebnisse im Vergleich zu gemessenen für den Benchmark *dijkstra*.

In Abbildung 6.5 wird ersichtlich, dass dieser Ansatz deutlich unabhängiger von der Mindestlänge als die bisherigen ist und das auf einem sehr guten Niveau im Vergleich zu anderen Strategien. Der Vergleich zwischen errechneten und gemessenen Kosten zeigt außerdem, dass die Abhängigkeit von frühzeitig deaktivierten Skipsections deutlich geringer ist. Durch das seltenere Vorkommen wird ein Deaktivieren deutlich unwahrscheinlicher, sodass dieser Ansatz in der aktuellen Umsetzung einer der effektivsten nach der Fehlerinjektion ist. Zudem zeigte er in keinem Fall eine Verschlechterung zur Laufzeit und verursacht durch wenige Skipsections nur einen sehr kleinen Speicher-Overhead.

6.3.7 Hierarchie

Der hierarchische Ansatz ermöglicht es die Flexibilität kürzerer Skipsections mit der Effektivität längerer Skipsections zu kombinieren. Die als Parameter festgelegte Mindestgröße beeinflusst also die Maximalgröße kaum, da den verschiedenen hierarchischen Ebenen keine Grenzen gesetzt sind. Der Vorteil gegenüber einer Basisstrategie (hier: *Equidistant*) wird in Abbildung 6.6 veranschaulicht. Dabei sein die blauen Skipsections eine Unterteilung mit der Länge l. Die Ergebnisse aus Unterabschnitt 6.3.2 haben gezeigt, dass diese Strategie für eine bestimmte Mindestlänge besonders effizient ist, sei diese optimale Länge in diesem Beispiel $4 \cdot l$, sodass die Unterteilung in die lilanen Skipsections erfolgen würde.

In diesem Fall würde die erste Skipsection angewendet werden und anschließend $4 \cdot l$ Instruktionen im Simulator ausgeführt werden, bis der Beginn der nächsten möglicherweise anwendbaren Skipsection erreicht wird. Bei der hierarchischen Strategie werden zu Beginn ebenfalls $4 \cdot l$ Instruktionen übersprungen, aber danach noch eine weitere Skipsection der Länge l angewendet. So sinnk die Anzahl der notwendigen Instruktionen vor der Fehlerinjektion deutlich. Auch nach der Fehlerinjektion startet eher eine neue Skipsection, sodass insgesamt nur l Instruktionen im Simulator ausgeführt werden müssen.

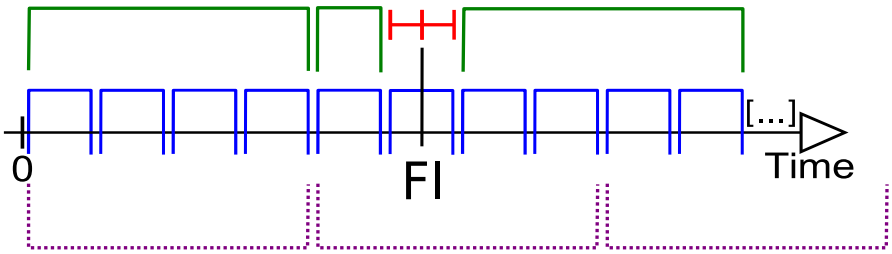

Abbildung 6.6: Verwendung einer hierarchischen Strategie im Vergleich zur Basisstrategie.

Nach der Fehlerinjektion kann der Ansatz aber auch Nachteile bringen, falls die zur Entwicklung aller Strategien verwendete Annahme, dass die Fehlerinjektion die Anwendung von Skipsections kaum beeinträchtigt, nicht stimmt. Durch den hierarchischen Ansatz wird bewusst erreicht, dass die der Fehlerinjektion folgende Skipsection nach vergleichsweise wenigen Instruktionen beginnt. Es ist aber möglich, dass der Fehler kurz nach der Injektion noch den Zustand und damit die Eingabe der Skipsection beeinflusst. Die Chance könnte unter Umständen also kurz nach der Fehlerinjektion höher sein, dass eine Anwendung nicht möglich ist. Zudem wird eine möglichst lange Skipsection genutzt, sollte diese nicht anwendbar sein, ist der nicht überdeckte Abschnitt, der also im Simulator ausgeführt werden muss, um so größer. Diese Einschränkung der zugrunde liegenden Annahme wurde im Rahmen dieser Arbeit aber aus zeitlichen Gründen nicht mehr genauer untersucht und müsste in weiterführenden Arbeiten analysiert werden.

Die verschiedenen Variationen des hierarchischen Ansatzes sind dabei sowohl bezüglich der berechneten Zeit, als auch bezüglich der gemessenen Werte sehr ähnlich, sodass für die weiteren Vergleiche nur jeweils ein Vertreter verwendet wird. Vereinfacht gilt dabei, dass *All* besser als *Level* ist und *Level* besser als *Bin*. Da die Anzal der Skipsections im ersten Ansatz nicht skaliert und leicht zu groß werden kann, empfiehlt sich die Verwendung einer der anderen beiden. Der durch zusätzlichen Speicher benötigte Overhead wird in Unterabschnitt 6.3.10 nochmal betrachtet.

Die Laufzeiten des hierarchischen Ansatzes werden in Abbildung 6.7 mit den anderen Laufzeiten verglichen. Dabei zeigt sich das enorme Potenzial des Ansatzes. Vor der Fehlerinjektion wurde eine Reduzierung der Laufzeit um 97% (*bfs*) bzw. 83% (*dijkstra*) erreicht, also im geometrischen Mittel eine Verbesserung um 93%.

Damit ist der Ansatz vor der Fehlerinjektion um ein Vielfaches besser als die anderen Ansätze, für die Gesamtlaufzeit kann diese große Verbesserung der Laufzeit allerdings nicht erreicht werden. Da hier unter Umständen deutlich längere Skipsections nach der Fehlerinjektion nicht genutzt werden können, kann danach kaum noch eine Verbesserung erzielt werden.

In den Untersuchten Benchmarks *bfs* und *dijkstra* wurde nach der Fehlerinjek-

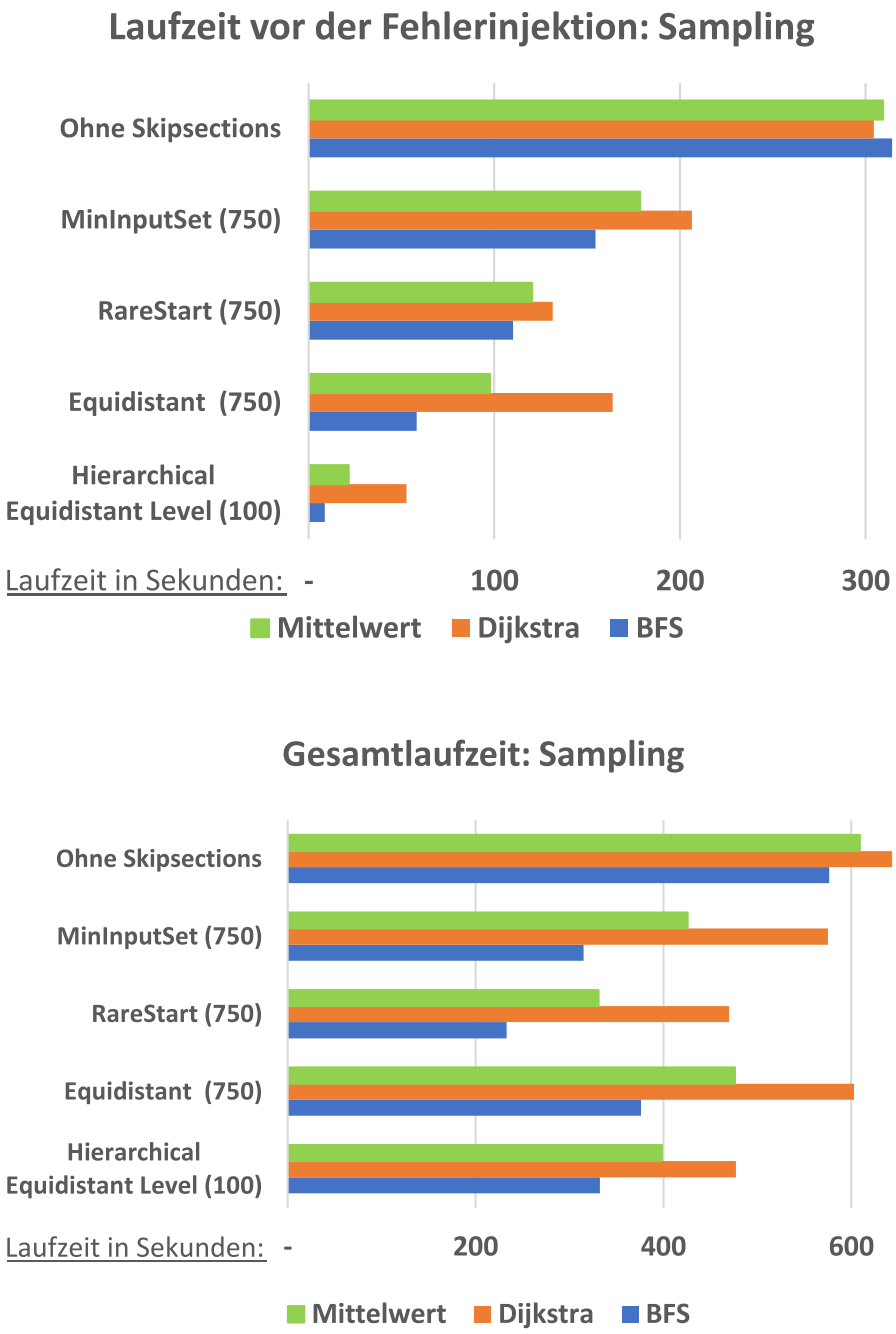

Abbildung 6.7: Die Effektivität des hierarchischen Ansatzes im Vergleich zu den anderen Strategien. Die Ergebnisse sind in die gemessene Laufzeit vor der Fehlerinjektion und die Gesamtlaufzeit aufgeteilt.

tion keine Skipsection mehr angewendet, sodass bezogen auf die Gesamtlaufzeit die verschiedenen *RareStartInstruction*-Strategien besser sind. Diese zerlegen den Trace aber nicht in zusammenhängende Skipsections, sodass hier der hierarchische Ansatz nicht erfolgreich angewendet werden kann.

In Abschnitt 6.4 wird das Problem nochmal genauer betrachtet und eine mögliche Lösung vorgestellt. Wenn die Anzahl der verwendeten Skipsections nach der Fehlerinjektion damit deutlich steigt, so würden alle Ansätze sich im Vergleich zu den *RareStartInstruction*-Strategien deutlich stärker verbessern, sodass anzunehmen ist, dass hierarchische Strategien dann die besten Resultate erzielen.

6.3.8 Mindestlänge

Dieser querschneidende Belang lässt sich nicht unabhängig von der jeweiligen Strategie betrachten. Für Strategien, bei denen die Länge der Skipsections in etwa der Mindestlänge entspricht, zeigte sich in den vorherigen Abschnitten dieser Auswertung, dass eine Mindestlänge zwischen 25% und 30% die besten Ergebnisse erzielt.

Für hierarchische Strategien hingegen sollte die Mindestlänge deutlich kleiner sein, vor der Fehlerinjektion sogar annähernd der kleinstmöglichen Mindestlänge. Das meint die Länge, bei der die Kosten für einen einmaligen Kontrollflusswechsel inklusive Vergleich und Änderung noch gerade so geringer sind, als die Kosten zum Ausführen der übersprungenen Instruktionen. Da allerdings nicht alle Skipsections passen, wird dieser Wert noch etwas höher seien müssen, eine genaue Untersuchung sollte aber in Abhängigkeit davon erfolgen, wie wahrscheinlich die Anwendung einer Skipsection nach der Fehlerinjektion wirklich ist.

Dabei würden aber zu viele Skipsections erzeugt werden können, sodass die Einschränkungen aus Unterabschnitt 6.3.10 beachtet werden müssen und die Mindestgröße soweit reduziert wird, dass die Gesamtanzahl von Skipsections nicht zu groß wird.

6.3.9 Reuse-Check

Der Effekt des *Reuse-Check* hängt sehr stark vom verwendeten Benchmark und der konkreten Unterteilung ab. Allgemein gilt, dass das Ändern einer Zustandsvariable im Verhältnis zu den anderen Kosten bei der Anwendung einer Skipsection sehr gering ist, sodass die Laufzeit nicht sehr stark verändert wird. Allerdings kann es nur eine Verbesserung geben, auch wenn diese gering sein mag.

Bei der Anwendung für alle Benchmarks mit mehr als einer Millionen Instruktionen und Skipsections der Länge 300.000 schwankte das Ergebnis zwischen 34.592 gelöschten Einträgen (*bfs*) und nur 70 gelöschten Einträgen(*sad_small*), sodass der Nutzen nur schwer abschätzbar ist.

Dafür zeigte sich bei keinem der getesteten Benchmarks eine Änderung im Ergebnis, sodass davon auszugehen ist, dass dieser Fall extrem selten eintritt. Dafür eignet sich diese Methode in jedem Fall in Kombination mit *Sampling*.

6.3.10 Speicherverbrauch

Der benötigte Speicherverbrauch einer einzelnen als Probuf-Datei gespeicherten Skipsection ist im Verhältnis zu den anderen Dateien sehr gering und stellt zumeist kein Problem dar.

Folgende Übersicht zeigt den Speicherverbrauch exemplarisch an einigen Beispielen:

blowfish State[3]0,9MB / *Equidistant*-Strategie mit Mindestlänge 250.000[4] 72KB / Trace 708KB / Extended-Trace 16MB.

dijkstra State 1,1MB / *MinInputSet*-Strategie mit Mindestlänge 250.000 40KB / Trace 4,8MB / Extended-Trace 40MB.

histo State 5,3MB / *Hierarchical-Equidistant-All*-Strategie mit Mindestlänge 4.000 121MB / Trace 35MB / Extended-Trace 481MB.

Dabei zeigt sich, dass für die mit *Hierarchical-All* erzeugten Skipsections deutlich mehr Speicherplatz benötigt wird. Das liegt daran, dass hier der Trace mehrfach überdeckt wird und sehr viele Skipsections erzeugt werden. Dabei ist zu beachten, dass der Trace 108.890.072 Instruktionen lang ist, die Skipsections wurden aber nur für den auf 150.000 Instruktionen limitierten Trace erzeugt. Für einen fast 726 Mal so großen Trace würden aber auch entsprechend mehr Skipsections erzeugt werden. Da die Anzahl dabei quadratisch mit der Anzahl der Instruktionen wächst, könnten Speichergrößen im Terabyte-Bereich resultieren. Das eignet sich weder zum Speichern, noch für die Nutzung bei der Anwendung.

Die Skipsections müssen dabei dem ausführenden *Client* irgendwie zur Verfügung stehen, das heißt sie müssen aus dem Speicher geladen werden. Das Laden verbraucht dabei auch eine gewisse Laufzeit, die in den vorherigen Betrachtungen nicht gemessen wird.

Bei einer effizienten Umsetzung kann ein *Client* die Daten einmalig laden und für jedes Experiment verwenden, sodass der einmalige Overhead bei einer großen Anzahl von Experimenten an Bedeutung verliert. Alternativ könnte der zusätzliche Aufwand zum Laden durch Speicherung als einzelne Protobuf-Datei je Skipsection gesenkt werden, da nur der kleine benötigte Teil der Aufzeichnungen geladen werden muss. Ohne ein sinnvolles Verfahren oder sehr gute *Caches* kann der Overhead zum Laden der Dateien den jeweiligen Laufzeitgewinn sehr leicht übersteigen, insbesondere bei sehr vielen Skipsections und entsprechend großen Dateien.

[3]Maschinenzustand zum Beginn des eigentlichen Programms.

[4]Jeweils die Größe aller drei Dateien (Skipsection, Aufzeichnung und Graph) zusammen.

Strategie	Länge	Skips	Skips nach FI	Mismatches	Deaktivierung
Equidistant	250.000	1680	678	577398	109
Equidistant	500.000	765	292	585586	45
MinInputSet	250.000	1634	710	840035	138
RareStart	500.000	592	2256	421	39

Tabelle 6.3: Übersicht über die Verwendung von Skipsections nach der Fehlerinjektion, wenn die Deaktivierung vor dem möglichen Erreichen ausgeschlossen wird.

6.4 Deaktivieren von Skipsections

In Unterabschnitt 6.3.5 wurde in Tabelle 6.2 gezeigt, dass insbesondere bei langen Benchmarks nur noch sehr selten Skipsections nach der Fehlerinjektion angewendet werden, was ein erheblicher Nachteil bezüglich der Laufzeit ist. Dieser Unterschied wurde auch bei den Laufzeiten der folgenden Effizienzuntersuchungen bestätigt.

Das kann entweder daran liegen, dass die Annahme aus Unterabschnitt 4.4.2 falsch ist und keine Skipsections mehr passen, oder die Skipsections werden entsprechend der Idee aus Unterabschnitt 4.3.3 deaktiviert, obwohl sie noch anwendbar sind.

Zum Überprüfen wurde das Skipping in der Art geändert, dass Skipsections nicht deaktiviert werden können, bevor sie zeitlich im *Golden Run* auftauchen würden. Dazu wurde die Differenz zwischen dynamischer Instruktion der Fehlerinjektion und dynamischer Startinstruktion der aktuellen Skipsection mit der Anzahl *Ticks* verglichen, die seit der Fehlerinjektion im Simulator passiert sind. Sind weniger *Ticks* als dynamische Instruktionen vergangen, bleibt der Zähler zum Deaktivieren auf null.

Das Ergebnis wird für 100 *Samples* und den Benchmark *bfs* auszugsweise in Tabelle 6.3 dargestellt (für die weiteren Benchmarks waren die Werte sehr ähnlich). Daran lassen sich mehrere Dinge ablesen:

1. Die meisten Skipsections können auch nach der Fehlerinjektion noch angewendet werden, die Annahme, dass die Fehlerinjektion keine zu großen Änderungen bewirkt, bleibt weiter plausibel.

2. Die Skipsections werden für fast alle Strategien zu früh deaktiviert, obwohl sie anwendbar sind.

3. Den Zähler zum Deaktivieren erhöhen, bzw. wie in diesem Fall erst später zu starten ist keine Lösung:

   Die enorme Anzahl von 840.035 fehlerhaften Überprüfungen bei MinInputSet wirkt sich um ein Vielfaches stärker auf die Laufzeit aus, als der Gewinn

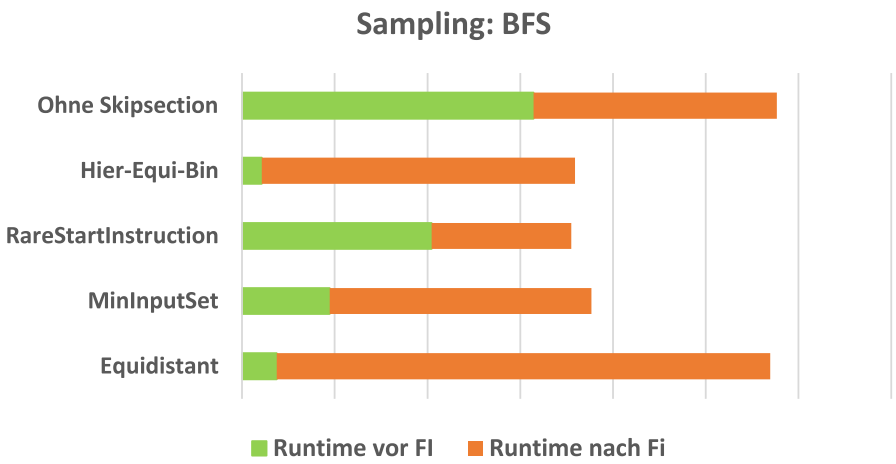

Abbildung 6.8: Übersicht der Laufzeit vor und nach der Fehlerinjektion.

durch die Anwendung der Skipsections. Die Messung ergab eine Steigerung der Laufzeit um fast 4.000%.

4. Skipsections mit seltenen statischen Startadressen sind wesentlich weniger stark davon betroffen.

Das Problem ist, bei sehr langen Skipsections müssen nach der Fehlerinjektion meistens sehr viele Instruktionen im Simulator ausgeführt werden, bis die nächste Skipsection erreicht wird. In dieser Zeit kann die statische Startadresse der nächsten Skipsection aber schon häufig auftreten, ohne dass sie angewendet werden kann. Passiert dies häufig genug, übersteigt das den Nutzen, den die Skipsection bringen würde.

Wenn diese Skipsection dann nicht anwendbar ist oder deaktiviert wird, vergrößert sich das Problem für die folgende Skipsection nur noch, da jetzt eine noch größere Anzahl von Simulatorinstruktionen ausgeführt werden muss. In der aktuellen Umsetzung wird so eine deutlich bessere Laufzeit erreicht und in den meisten Fällen auch eine Verbesserung im Vergleich zur Fehlerinjektion ohne Skipsections.

In Abbildung 6.8 wird die Laufzeit für den Benchmark *bfs* und die Mindestlänge 250.000 Instruktionen in die Dauer vor und nach der Fehlerinjektion aufgeteilt. Alle Strategien schaffen es unterschiedlich stark die Laufzeit zu verkürzen. Außer bei der *RareStartInstruction*- Strategie erzielen alle den Laufzeitgewinn nur vor der Fehlerinjektion und verschlechtern sich danach. Dadurch geht sehr viel Potenzial verloren, zumal die meisten Skipsections noch immer genutzt werden könnten.

Als Lösung für das Problem würden sich der Ansatz aus Unterabschnitt 4.3.3 Punkt 5 eignen. Zwar muss dann ein verhältnismäßig langsamerer *Listener* mit einem Zähler verwendet werden, die großen Laufzeitgewinne vor der Fehlerinjektion legen aber nahe, dass dies noch immer einen sehr großen Nutzen bringt.

Umgesetzt wurde diese weitere potenzielle Verbesserung aus zeitlichen Gründen nicht mehr.

6.5 Diskussion der Ergebnisse

Alle Benchmarks können durch die Verwendung geeigneter Auswahlstrategien deutlich beschleugnigt werden. Dabei beträgt die mögliche Verbesserung für die meisten Benchmarks zwischen 50% und 60%. Diese guten Ergebnisse werden trotz noch fehlender Optimierung für die Skipsections nach der Fehlerinjektion erreicht. Vor der Fehlerinjektion wird vielfach eine Beschleunigung von über 90% erreicht. Dieser Ansatz muss daher noch nach der Idee aus Unterabschnitt 4.3.3 Punkt 5 umgesetzt werden, um den Effekt weiter zu verstärken.

Größte Einschränkung ist dabei die zu seltene Anwendung von Skipsections nach der Fehlerinjektion. Dies limitiert für alle Ansätze, die nicht die Häufigkeit der statischen Startadresse einer Skipsection als Hauptkriterium verwenden, den möglichen Laufzeitgewinn enorm. Hier muss eine weitere Verbesserung erfolgen, eine mögliche Lösung wurde dazu in Abschnitt 6.4 besprochen. Dabei würde die vor der Fehlerinjektion sehr erfolgreiche Verbesserung durch Verwendung von dynamischen Startpunkten auch nach der Fehlerinjektion mit angewendet werden.

Die weiteren Kriterien, Ergebnistreue, Generalität und Nutzerfreundlichkeit ergeben sich aus dem Ansatz der Skipsections an sich und bleiben unabhängig von der jeweiligen Strategie erhalten.

6.6 Bewertung des Evaluators

Für die meisten betrachteten Fälle ähnelten die Ergebnisse des Kampagnen-Evaluators denen, die durch Messung bei realen Fehlerinjektionen gewonnen wurden. Strategien, die allgemein besser bewertet wurden, zeigten sich auch bei den Messungen schneller und auch der Einfluss verschiedener Parameter wurde gut dargestellt.

Damit ist die vorliegende Erweiterung durchaus zur Bewertung von Strategien nutzbar, allerdings mit den in Abschnitt 6.4 beschriebenen Einschränkungen für lange Skipsections und die Anwendung nach der Fehlerinjektion. Das Deaktivieren wurde bei der Implementierung nicht berücksichtigt und sorgt für Abweichungen, je seltener Skipsections noch genutzt werden. Bei einer Anpassung des Skippings würde diese Abschwächung des Laufzeitgewinns möglicherweise aber deutlich zurück gehen und nur noch für die selteneren Fälle, dass eine Skipsection tatsächlich nicht anwendbar ist, bestehen bleiben. Dann müssten die Ergebnisse deutlich näher an den berechneten Werten liegen.

Bei kleineren Benchmarks wie in Abbildung 6.1 waren die Ergebnisse aber auch schon sehr exakt und konnten die gemessenen Werte sehr gut darstellen.

6.7 Zusammenfassung

Unter Verwendung der zuvor aufgestellten Kriterien zeigt sich das große Potenzial des vorliegenden Ansatzes. Es wurde ein Überblick über die untersuchten Strategien gegeben und mögliche Änderungen zur weiteren Verbesserung identifiziert.

7 Zusammenfassung

Nach Recherchen zur simulatorbasierter Fehlerinjektion und dem am Lehrstuhl entwickelten Tool FAIL*, wurde der durch Stampa entwickeltet Ansatz zur Beschleunigung von Fehlerinjektionskampagnen[19] genauer untersucht und Möglichkeiten zur Verbesserung identifiziert. Dabei lag besonderes Augenmerk auf der verwendeten Strategie zur Unterteilung des Traces in verschiedene überspringbare Abschnitte. Zusätzlich wurde ein Werkzeug zur einfacheren Beurteilung dieser Strategien entwickelt. Die verschiedenen Strategien wurden anschließend mit verschiedenen Benchmarks bewertet. Dazu wurden neben der Abschätzung der Laufzeit auch Messungen bei ausgeführten Fehlerinjektionsexperimenten verwendet.

7.1 Fazit

Die Erweiterung des vorliegenden Ansatzes um die Verwendung dynamischer Instruktionen basiert auf der Beobachtung, dass zumindest vor der Fehlerinjektion der Zeitpunkt der jeweiligen Anwendung einer Skipsection genau bestimmt werden kann und so der größte Kostenfaktor bei der Verwendung von Skipsections eliminiert werden kann. Zur Anwendung wird weder ein Vergleich benötigt, noch erfolgt ein unnötiger Kontrollflusswechsel, wie es bei der Verwendung der statischen Startadressen zur Bestimmung des richtigen Zeitpunkts passieren kann.

Der Nachteil, dass ein *Listener* auf eine statische Adresse deutlich schneller ist als ein *Listener* mit Zähler, hat zumindest vor der Fehlerinjektion keinen Einfluss.

Eine Reihe möglicher Auswahlstrategien wurde entwickelt und genauer untersucht, sodass eine Verwendung bei Fehlerinjektionskampagnen sehr einfach erfolgen kann und der Verwender sich im Rahmen des Anwendungskontexts einfach für eine konkrete Strategie entscheiden kann. Bezüglich notwendiger Kriterien, wie der Mindestlänge von Skipsections, wurden sinnvolle Werte in Abhängigkeit des jeweiligen Benchmarks vorgestellt

Zur Entwicklung neuer Strategien wurde ein Werkzeug entwickelt, dass eine Beurteilung der Effizienz sehr einfach ermöglicht.

7.2 Ausblick

In weiterführenden Arbeiten sollte die Anwendung von Skipsections nach der Fehlerinjektion optimiert werden, ein mögliches Verfahren dazu wurde im Rahmen der

Arbeit bereits vorgestellt, aber noch nicht umgesetzt. Sollte hier keine hinreichende Verbesserung erreicht werden, so sollten weitere Strategien auf Grundlage dieser Einschränkung entwickelt werden.

Eine Möglichkeit wäre es z. B. den hierarchischen Ansatz mit der Suche nach möglichst seltenen statischen Startadressen zu verbinden, indem die größtmöglichen Skipsections ermittelt und dann in regelmäßigen Abständen unterteilt werden. Die Grenzen, die sowohl Start- als auch Endpunkt einer Skipsection sind, müssen anschließend so verschoben werden, dass sie auf Instruktionen liegen, die möglichst selten sind.

Zudem sollten die Einschränkungen der in dieser Arbeit verwendeten Annahme genauer untersucht werden, da sich so neue Ansätze für weitere Strategien ergeben können. Unter der Annahme, dass nach der Fehlerinjektion jede Skipsection anwendbar ist, ist das Ziel, sehr kurz nach dem Injektionszeitpunkt eine möglichst große Skipsection zu starten. Steigt allerdings die Wahrscheinlichkeit, dass eine Skipsection nicht angewendet werden kann mit der Nähe zum Zeitpunkt der Injektion, so müssen Strategien dahingehend angepasst werden.

Der Kampagnen-Evaluator kann außerdem in der Art verbessert werden, dass die Wahrscheinlichkeit mit der eine Skipsection anwendbar ist, genauer untersucht und in der Implementierung berücksichtig wird.

Literatur

[1] L. Berrojo u. a. „New techniques for speeding-up fault-injection campaigns“. In: *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*. 2002, S. 847–852. DOI: 10.1109/DATE.2002.998398.

[2] C. Borchert, H. Schirmeier und O. Spinczyk. „Generative Software-based Memory Error Detection and Correction for Operating System Data Structures“. In: *International Conference on Dependable Systems and Networks (DSN'13)*. IEEE Computer Society Press, Juni 2013.

[3] Z. Chen u. a. „BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356177. URL: https://doi.org/10.1145/3295500.3356177.

[4] T. Coughlin. „A Road Map for Technologies That Drive Consumer Storage [The Art of Storage]“. In: *IEEE Consumer Electronics Magazine* 8.2 (2019), S. 97–99. DOI: 10.1109/MCE.2018.2880855.

[5] E. W. Dijkstra. „A Note on Two Problems in Connexion with Graphs“. In: *Numer. Math.* 1.1 (Dez. 1959), S. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: https://doi.org/10.1007/BF01386390.

[6] M. Guthaus, J. Ringenberg, D. Ernst u. a. „MiBench: A free, commercially representative embedded benchmark suite“. In: *IEEE International Workshop Workload Characterization*. IEEE, Dez. 2001.

[7] R. Leveugle u. a. „Statistical fault injection: Quantified error and confidence“. In: *2009 Design, Automation Test in Europe Conference Exhibition*. 2009, S. 502–506. DOI: 10.1109/DATE.2009.5090716.

[8] J. Li und Q. Tan. „SmartInjector: Exploiting intelligent fault injection for SDC rate analysis“. In: *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. 2013, S. 236–242. DOI: 10.1109/DFT.2013.6653612.

[9] H. Madeira u. a. „RIFLE: A general purpose pin-level fault injector“. In: *Dependable Computing — EDCC-1*. Hrsg. von K. Echtle, D. Hammer und D. Powell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, S. 197–216. ISBN: 978-3-540-48785-2.

[10] A. J. Massa. *Embedded software development with eCos*. Prentice Hall Professional, 2002.

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

[12] S. Mukherjee. *Architecture Design for Soft Errors* -. San Francisco, Calif: Morgan Kaufmann, 2011. ISBN: 978-0-080-55832-5.

[13] H. Schirmeier u. a. „FAIL*: Towards a versatile fault-injection experiment framework“. In: *25th International Conference on Architecture of Computing System (ARCS' 12)*. Bd. 200. German Society of Informatics, März 2012, S. 201–210.

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

[15] H. B. Schirmeier. „Efficient Fault-Injection-based Assessment of Software-Implemented Hardware Fault Tolerance“. dissertation. Technischen Universität Dortmund, 2016.

[16] M. Sellung. „Vergleich von Hardwarefehlermodellenbei Fehlerinjektionsexperimenten“. Bachelor Thesis. Technischen Universität Dortmund, März 2015.

[17] D. T. Smith u. a. „A method to determine equivalent fault classes for permanent and transient faults“. In: *Annual Reliability and Maintainability Symposium 1995 Proceedings*. 1995, S. 418–424. DOI: 10.1109/RAMS.1995.513278.

[18] *Software Error Doomed Japanese Hitomi Spacecraft*. https://www.scientificamerican.com/article/software-error-doomed-japanese-hitomi-spacecraft/. Eingesehen am 03.01.2021. URL: https://www.scientificamerican.com/article/software-error-doomed-japanese-hitomi-spacecraft/.

[19] M. Stampa. „Beschleunigung von Fehlerinjektionsexperimenten durch Zusammenfassung von Zustandsraumtransformationen“. Master Thesis. Technischen Universität Dortmund, Aug. 2017.

[20] J. A. Stratton u. a. „Parboil: A revised benchmark suite for scientific and commercial throughput computing“. In: *Center for Reliable and High-Performance Computing* 127 (2012).

[21] H. Ziade, R. A. Ayoubi und R. Velazco. „A Survey on Fault Injection Techniques“. In: *Int. Arab J. Inf. Technol.* 1 (2004), S. 171–186.

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis