Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Leibniz Universität Hannover

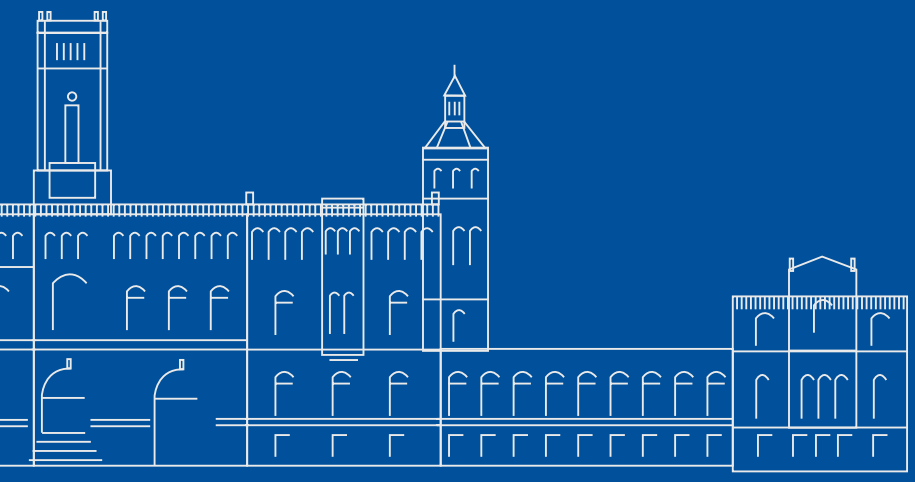
Daniel Kiechle

Fehlerraumapproximation durch Verwendung von Basisblock-Fehlerinjektion

Bachelorarbeit im Fach Informatik

15. Oktober 2018

Please cite as:
Daniel Kiechle, “Fehlerraumapproximation durch Verwendung von Basisblock-Fehlerinjektion” Bachelor’s Thesis, Leibniz Universität Hannover, Institut für Systems Engineering, October 2018.



Leibniz Universität Hannover
Institut für Systems Engineering
Fachgebiet System und Rechnerarchitektur
Appelstr. 4 · 30167 Hannover · Germany

Fehlerraumapproximation durch Verwendung von Basisblock-Fehlerinjektion

Bachelorarbeit im Fach Informatik

vorgelegt von

Daniel Kiechle

geb. am 18.12.1989
in Lindenberg im Allgäu

angefertigt am

Institut für Systems Engineering
Fachgebiet System- und Rechnerarchitektur

Fakultät für Elektrotechnik und Informatik
Leibniz Universität Hannover

Erstprüfer:	**Prof. Dr.-Ing. habil. Daniel Lohmann**
Zweitprüfer:	**Prof. Dr.-Ing. Bernardo Wagner**
Betreuer:	**Oskar Pusz, M.Sc.**
	Christian Dietrich, M.Sc.

Beginn der Arbeit:	**15. Juni 2018**
Abgabe der Arbeit:	**15. Oktober 2018**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Daniel Kiechle)
Hannover, 15. Oktober 2018

KURZFASSUNG

In der Entwicklung neuer Assistenzsysteme ist die Strukturbreite, also die Größe des Systems, ein relevanter Faktor, sei es in der Kraftfahrzeugbranche, um den Fahrer bei seiner Fahrt zu unterstützen oder in der Gesundheitsbranche, um Patienten besser zu überwachen. Durch diese Verkleinerung der Systeme sind sie auch anfälliger für äußere Störungen, wie z.B. kosmische Strahlung. Diese externen Störungen können ein solches System stark beeinflussen, daher wird in der Entwicklung mit Hilfe von Tests auf diese Probleme eingegangen. Da äußere Störungen im Allgemeinen nicht deterministisch auf das System wirken, sind die Tests sehr schwierig durchzuführen und sehr zeitaufwendig. In dieser Arbeit wird ein Verfahren beschrieben, welches den Testaufwand stark reduziert. Dieses Verfahren nutzt das Def-Use Pruningverfahren als Grundlage und verringert mit erweitertem Wissen über die Systemstruktur die möglichen Testpunkte. Das entwickelte Basisblock-Pruningverfahren hat gezeigt, dass durch die Verwendung von Basisblock-Informationen eine Verringerung des Testaufwands um 50% im Testumfeld von CPU-Registern möglich ist, ohne die Aussage über das Verhalten des Systems unter Störungen signifikant zu verfälschen.

ABSTRACT

In the development of new assistance systems the structural width, i.e. the size of the system is a relevant factor, be it in the automotive industry to support the driver during his journey or in the health industry to better monitor patients. This reduction in the size of the systems also makes those systems more susceptible to external interference such as cosmic radiation. These external interferences can have a strong influence on systems, therefore these problems are addressed in development with the help of tests. Since external disturbances generally occur nondeterministically with respect to the system, these tests are very difficult to perform and very time-consuming. This thesis describes a procedure that greatly reduces the amount of testing required. This procedure uses the def-use pruning method as a basis and reduces the possible test points with extended knowledge of the system structure. The developed basicblock pruning procedure has shown that by using basicblock information the test effort can be reduced by about 50% in the test context of CPU registers without significantly distorting the statement about the behavior of the system under disturbances.

INHALTSVERZEICHNIS

1 EINLEITUNG

1.1 Motivation

Wir leben in einer Zeit, in der die Digitalisierung in allen Bereichen Einzug hält. Beispielsweise in der Gesundheitsbranche, wo am Menschen platzierte Systeme (Fitnessarmbänder) Daten erfassen, welche Krankenkassen über den aktuellen Zustand der Person informieren, um eine eventuelle Erkrankung zu diagnostizieren [SSU16] oder in der Automobilbranche, wo in Kraftfahrzeugen immer mehr Assistenzsysteme zur Überwachung und Unterstützung des Fahrers eingesetzt werden [Kuf+12]. Somit interagieren diese Systeme immer mehr mit dem Menschen und beeinflussen dadurch auch dessen Entscheidungen. Hierbei muss der Mensch darauf vertrauen können, dass die Funktionszuverlässigkeit (engl. *reliability*) des Systems gewährleistet ist und somit das Verhalten des Systems immer dem vom Nutzer Erwarteten entspricht.
Da in der Entwicklung von Systemen Preis, minimale Größe und Energieeffizienz besonders wichtige Ziele sind, wird versucht die benötigte Technik auf minimalem Platz unterzubringen. Dies hat zur Folge, dass Bausteine wie z.B. Transistoren immer kleiner und energiesparender werden müssen: Wo im Intel Core 2 (Yorkfield) von 2007 noch ein Transistor mit einer Gate-Strukturbreite von 45 nm verbaut wurde [Wik18] haben Forscher in den USA 2016 einen Transistor mit einer Gate-Strukturbreite von lediglich 1 nm entwickelt [Des+16], wodurch Größe und Energieeffizienz innerhalb von 9 Jahren stark optimiert wurden.
Durch die stetige Verkleinerung der Bausteine wird immer weniger Energie benötigt, um Informationen in digitalen Systemen zu speichern. Aus dem geringeren Energiebedarf folgt, dass die Ladungen, die die Information beinhalten, auch deutlich geringer sind. Dieses stellt die Informatik vor ein Problem: In der Umwelt existieren Energieträger, die mit ihrer Energie die Möglichkeit besitzen die geringeren Ladungen zu verändern und somit auch die darin enthaltene Information [MW79]. Energieträger können z.B. die Versorgungsspannung, Teilchen- und Elektromagnetische Strahlung sein [Zie+96]. Eine von diesen Energieträgern verursachte, zufällig auftretende Veränderung nennt man transienter Hardwarefehler (engl. *soft error*), der ein System in seiner Funktionszuverlässigkeit und somit auch das Verhalten des Systems beeinträchtigen kann.
Die Schwierigkeit, ein System auf Funktionszuverlässigkeit zu testen, besteht darin, dass der Fehlerraum, in dem ein transienter Hardwarefehler auftreten kann, sehr groß ist: Zu jedem Zeitpunkt der Ausführung kann jede gespeicherte Information verändert werden.
Um diesen großen Fehlerraum zu testen, sind mehrere Verfahren möglich: Eins davon ist es, das System realen Energieträgern auszusetzen, um das Verhalten dabei zu beobachten, eine andere, das System mit Hilfe von software-simulierten, transienten Hardwarefehler zu testen. Bei einer solchen Simulation ist der Testaufwand weiterhin extrem hoch, da jede durch einen transienten Hardwarefehler mögliche Veränderung getestet werden müsste, was somit angesichts des großen

Fehlerraumes immer noch sehr zeitintensiv ist. Wird das System realer Strahlung ausgesetzt, wirkt diese im Allgemeinen nicht deterministisch - somit sind Testergebnisse praktisch nicht reproduzierbar. Um reproduzierbare Testergebnisse zu ermöglichen wird der Test auf Simulationsbasis durchgeführt. Des Weiteren wird der Ansatz verfolgt, den Testaufwand innerhalb der Simulation selbst zu verringern.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, im Bereich der software-simulierten transienten Hardwarefehler ein Verfahren auf seine Tauglichkeit zu prüfen. Der Schwerpunkt liegt hierbei auf der Verringerung des Testaufwandes ohne die Aussage über die Funktionszuverlässigkeit des Systems zu verfälschen.
Das zu prüfende Verfahren baut auf einem bereits bekannten Verfahren namens Def-Use Pruning [Smi+95] auf. Das neue Verfahren erweitert das Def-Use Pruning mit Informationen aus der Programmstruktur, sodass eine weitere Reduktion des Testaufwandes möglich ist. Um das Ziel der unverfälschten Aussage zu garantieren, wird die Aussage des Def-Use Prunings als Baseline genommen.

1.3 Struktur der Arbeit

Diese Arbeit ist in sieben weitere Kapitel gegliedert. Im folgenden Kapitel wird auf die grundlegenden Begriffe, die in dieser Arbeit verwendet werden, eigegangen, um eine fundierte Grundlage für die nachfolgenden Kapitel zu schaffen. Das dritte Kapitel ordnet das Thema dieser Arbeit in aktuelle Forschungsthemen ein und greift verwandte Themen kurz auf. Das vierte Kapitel stellt das in dieser Arbeit neu erarbeitete Verfahren des Prunings vor und beschreibt die Methodik, wie das Verfahren in Bezug auf die Aussage über das System mit dem Def-Use Pruning verglichen werden kann. Im fünften Kapitel wird zuerst auf die Evaluationsumgebung und auf das zugrunde ligende Framework FAIL* - FAult Injection Leveraged [Sch+12] eingegangen. Nach der Beschreibung aller Grundlagen werden am Ende des fünften Kapitels die Ergebnisse dieser Arbeit präsentiert, um im folgenden sechsten Kapitel die Bedeutung der Ergebnisse herzuleiten. Das siebte Kapitel fasst die Ergebnisse dieser Arbeit im Rahmen eines Fazits nochmals zusammen, um im letzten Kapitel einen Ausblick über weitere Möglichkeiten und Verbesserungen zu geben.

2 GRUNDLAGEN

In diesen Kapitel werden die Grundlagen, die dieser Arbeit zugrunde liegen, beschrieben. Als Erstes werden notwendige Begriffe eingeführt, um danach den Zusammenhang zwischen Zuverlässigkeit und Fehler näher zu erläutern. Im zweiten Abschnitt wird näher auf den betrachteten Fehlerraum und das Fehlermodell eingegangen, um anschließend die eigentliche Fehlerinjektion zu beschreiben. In letztem Teil dieses Kapitels wird dann das Verfahren des Prunings, in dem sich die in dieser Arbeit zu evaluierende Methode einzugliedern lässt, erläutert.

2.1 Zuverlässigkeit, Fehler, Defekt und transienter Fehler

Die Begriffe Zuverlässigkeit, Fehler und Defekt werden im alltäglichen Gebrauch oft undifferenziert verwendet. Daher werden die Begriffe in diesem Abschnitt im Kontext dieser Arbeit kurz erläutert. Der Begriff der *Zuverlässigkeit (engl. dependability)* ist, im Bezug auf ein System, ein Qualitätsmerkmal, welches sich aus den folgenden Attributen zusammengesetzt [VDI07]:

- **Funktionszuverlässigkeit (engl. reliability)** Die Eigenschaft des Systems, innerhalb eines definierten Zeitintervalls eine zu erfüllende Funktion fehlerfrei auszuführen.
- **Verfügbarkeit (engl. availability)** Die Eigenschaft des Systems, zu einem definierten Zeitpunkt zur Verfügung zu stehen.
- **Instandhaltbarkeit (engl. maintainability)** Die Eignung für die Instandhaltung mit festgelegten Mitteln und Verfahren.
- **Sicherheit (engl. safety)** Das Risiko, bei Ausführung der zu erfüllenden Funktion einen Zustand zu erreichen, der einen unvertretbaren Schaden mit sich führt.
- **Robustheit (engl. robustness)** Die inhärente Eigenschaft des Systems, auch bei Verletzung von Rahmenbedingungen die zu erfüllende Funktion auszuführen.

Das Qualitätsmerkmal Zuverlässigkeit kann durch *Fehler (engl. error)*, welche dann zum *Fehlverhalten bzw. Funktionsausfall (engl. failure)* des Systems führen, beeinträchtigt werden. Ein *Fehler* beschreibt einen im System befindlichen *Fehlzustand (engl. error state)*, welcher zu einer von außen erkennbaren Abweichung (Fehlverhalten oder Funktionsausfall) vom erwartetem Verhalten führt. Jedem *Fehlzustand* im System geht eine *Fehlerursache*, auch *Defekt (engl. fault)* genannt, voraus. Ein solcher Defekt kann, wie in der Einleitung schon beschrieben, durch äußere Einflüsse entstehen. Dieser ungewollte Zustandswechsel in einem verbauten Hardwareelement kann dann zu einem *Fehlzustand* führen, der das System in einen nicht erwarteten Zustand versetzt. Defekte, die nicht

vom System abgefangen werden, z.B. wenn der ungewollte Zustandswechsel direkt überschrieben wird oder außerhalb des benutzen Bereichs der Hardware liegt, haben keinen Einfluss auf die Zuverlässigkeit, in diesem Fall spricht man auch von einem maskierten beziehungsweise *gutartigen Defekt (engl. benign fault)*. Auch ein *Fehlzustand* muss, analog zu einem Defekt, nicht zwingend zu einem *Fehlverhalten* des System führen und kann somit ebenfalls gutartig sein. Ein Unterschied zwischen den Begriffen Fehlverhalten und Funktionsausfall ist, dass ein Fehlverhalten durchaus auch im Verborgenen bestehen kann, also von außen nicht erkennbar ist. Man spricht dann auch von einer *Silent data corruption*. Ein Funktionsausfall beschreibt zwar auch den Fehlzustand im System, dieser ist jedoch klar von außen erkennbar.
Im weiteren Verlauf der Arbeit wird die Funktionszuverlässigkeit und deren Bewertung betrachtet. Ein Fehler im Bezug auf die Funktionszuverlässigkeit des Systems hat als Ursache einen Defekt, welcher innerhalb eines definierten Zeitintervalls und unter definierten Rahmenbedingungen auftritt. In dieser Arbeit werden nur Defekte betrachtet, die durch einzelne so genannte *Single Event Upsets*, welche Information (Datenwort) in general purpose Registern oder im Hauptspeicher verändert, ausgelöst werden. Ein *Single Event Upset* ist ein *Transienter Fehler*, der ein kurzzeitiges Invertieren der in dem Transistor gespeicherten Information (*Bitkipper (engl. bit flipping)*) beschreibt. Diese kurzzeitigen Veränderungen können durch direkte oder indirekte Teilchenstrahlung und Elektromagnetische Strahlung hervorgerufen werden [Fec09].
In Abbildung 2.1 ist der Weg eines Bitkippers zum Fehler grafisch dargestellt. Der Pfeil zeigt den Weg eines einzelnes Bitkippers, der einen Defekt zur Folge hat, bis er zu einem Fehlverhalten des System führt. Die Hintergrundfarben stellen den Systemzustand dar: Im grünen Bereich führt das System seine definierte Funktion ohne Einschränkungen aus; der orange Bereich stellt ein nicht mehr abzuwendendes Fehlverhalten dar, dass entweder im Verborgenen bleibt (Silent data corruption) oder zum von außen ersichtlichen Funktionsausfall des Systems führt, welcher durch den roten Bereich angezeigt wird. Wie bereits beschrieben, können Defekte und Fehlzustände gutartig sein und so wird der grüne Bereich nicht verlassen. Ist ein Fehlzustand nicht gutartig, führt dieser unvermeidlich zu einem Fehlverhalten. Dann spricht man von einem nach außen hin sichtbaren Fehler, der einen Funktionsausfall zur Folge hat.

2.2 Fehlerraum und Fehlermodell

Mit Hilfe von *Fehlerhypothesen*, einer Annahme über Zeitpunkte, Orte und Muster von Fehlern, die ein System nicht in seiner Funktionszuverlässigkeit einschränken dürfen, wird ein *Fehlermodell* erstellt, welcher die aus der Fehlerhypothese entnommenen erwartbaren Fehler eines Systems spezifisch beschreibt [Kop06]. Aus diesem resultiert dann ein *Fehlerraum (engl. faultspace)*, welcher die Zeitpunkte und Orte der möglichen Defekte auf den konkreten Systemablauf abbildet [Bar+].
In dieser Arbeit wird ein Fehlermodell verwendet, welches von einer Einzelfehlerannahme ausgeht. Somit beschreibt das Fehlermodell nur Fehler, die durch einen einzelnen *Bitkipper* ausgelöst werden können [Ech13]. In Abbildung 2.2 wird der Fehlerraum des Fehlermodells dargestellt. Die vertikale Anordnung der Punkte beschreibt jeweils ein Bit in der Registerebene oder im Hauptspeicher, welches zur Ausführungszeit des Systems benutzt wird. Die Horizontale Achse zeigt in Bezug auf die Ausführungszeit (hier dargestellt als Abfolge von Instruktionen) mögliche Zeitpunkte, in denen ein Defekt im Datenwort der Registerebene oder des Hauptspeichers auftreten kann. Um eine qualitative Aussage über die Funktionszuverlässigkeit treffen zu können, müssen Tests diesen Fehlerraum möglichst vollständig abdecken. Da die Größe des Fehlerraums abhängig von benutzen Register- oder Speicherbereichen und betrachteter Ausführungszeit ist, kann dieser schnell anwachsen und somit praktisch fast nicht vollständig getestet werden. Ein kleines Beispiel soll dies verdeutlichen:

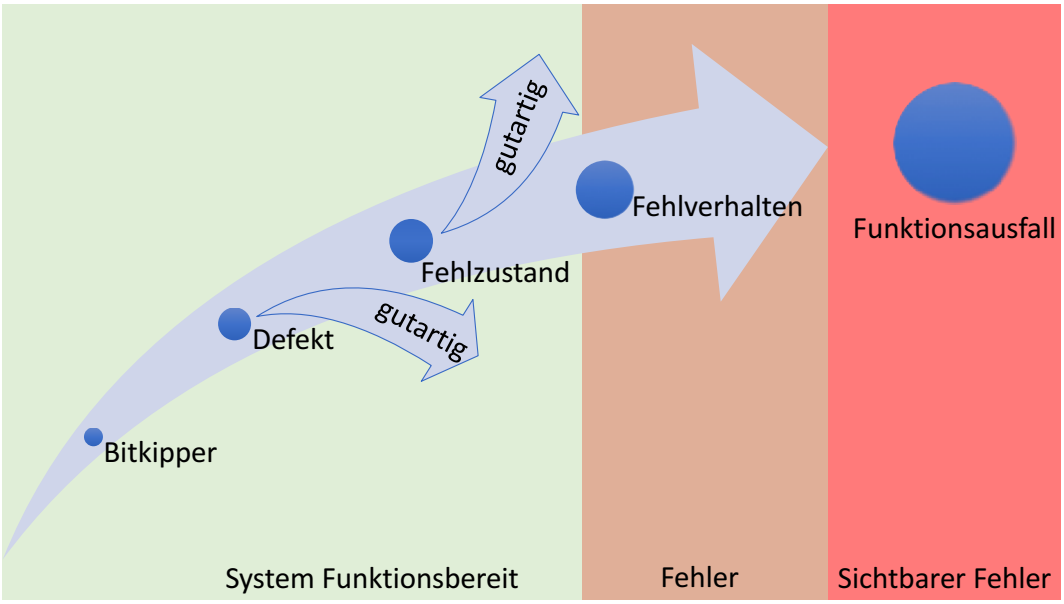


Abbildung 2.1 – *Fehlerpfad:* Vom Bitkipper zum Fehlverhalten

Nimmt man ein Programm, welches auf einem Mikrocontroller 100kB Daten während seiner Ausführung von 10 Sekunden in Register oder Speicher verarbeitet (lesend wie auch schreibend), dann ist der Fehlerraum bezogen auf das hier verwendete Fehlermodell 100000 Byte · 10 Sekunden groß, wenn man von einem einsekündigen Testintervall ausgeht. Bitweise getestet sind dies 8 Million mögliche Bitkipper, die sich als Defekt bemerkbar machen. Möchte man nun den vollständigen Fehlerraum für das Programm abdecken, müsste man das Programm mit jedem möglichen Bitkipper vollständig durchlaufen lassen. Somit würde dieser Test eine Zeit von mindenstens 8 Millionen mal 10 Sekunden benötigen. Dies entspricht circa 926 Tagen Testdauer. Jede weitere Sekunde, die das Programm länger liefe, würde den Testaufwand um circa 194 Tage erhöhen. Bei Erhöhung der Daten um ein Byte würde der Test 43 Stunden länger dauern.

2.3 Fehlerinjektion

Für eine Evaluation der Funktionszuverlässigkeit eines Systems werden künstlich Defekte in dem System platziert. Um dieses durchzuführen sind mehrere Techniken möglich, die in Kapitel 3 genauer erläutert werden. Das Verfahren, was in dieser Arbeit angewendet wird, ist die *Simulationsbasierte Fehlerinjektion*, welche das zu testende System in einer Hardwaresimulation ausführt. Durch die Simulation der Hardware bekommt die simulationsbasierte Fehlerinjektion die Möglichkeit, einen Defekt während der Simulation zielgerichtet zu platzieren und das Verhalten nach der Platzierung zu beobachten. Dabei wird immer nur ein Defekt platziert, um das Verhalten genau dieses Defekts

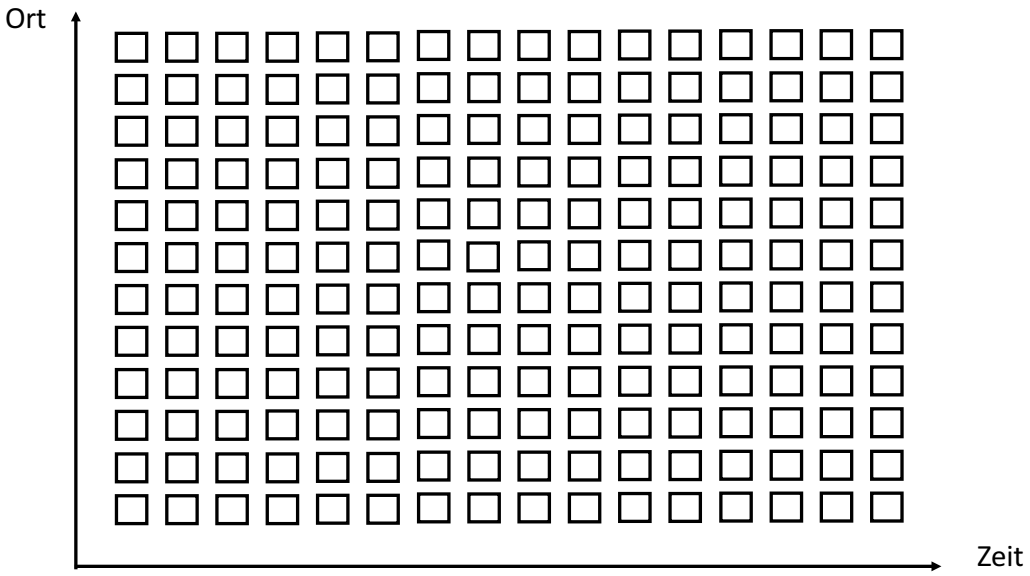


Abbildung 2.2 – Fehlerraum mit möglichen eintretenden Defekten

zu sehen. Diese Art der Fehlerinjektion ermöglicht es, durch die Hardwaresimulation die Tests auf verschiedenen Gastsystemen durchzuführen und somit eine Parallelisierung der Testausführungen zu ermöglichen. Im Weiteren wird ein Punkt des Fehlerraums (Zeitpunkt, Bit), in dem ein Defekt platziert wird, als Injektionspunkt (IP) bezeichnet. Das System wird einmalig ohne Defekt ausgeführt, um einen Referenzlauf des Systems zum Vergleich zu besitzen. Eine Fehlerinjektion beschreibt im Rahmen dieser Arbeit den Ablauf einer Injektion wie folgt:

- **Auswahl des zu testenden Defektes** Es wird im Fehlerraum, der in Abbildung 2.2 beschrieben wurde, ein zu testender Defekt ausgewählt.
- **Simulation bis zum IP** Das zu testende System wird bis zum Zeitpunkt, in dem der Defekt eintreten soll, simuliert und danach angehalten.
- **Platzierung des Defekts** In der simulierten Hardware wird der Defekt in Form eines Bitkippers platziert, welcher sich im Register oder im Hauptspeicher befinden kann.
- **Simulation wird fortgeführt** Nach der Platzierung des Defektes wird die Simulation fortgeführt.
- **Analyse des Systems** Das System wird auf sein Verhalten nach dem Defekt am Programmende analysiert. Mögliche Auswirkungen wie terminiert/terminiert nicht oder fehlerfrei/fehlerhaft werden protokoliert.

Durch die gewählte Methode der Injektion von Defekten ist es möglich, jeden Testlauf nochmals zu durchlaufen und die Ergebnisse jederzeit zu reproduzieren. Dieses ist möglich, da es sich bei der Simulationsbasierten Fehlerinjektion um ein deterministisches Verfahren handelt, was die Hardwarebasierte Fehlerinjektion durch z.B. Streuung bei der Bestrahlung nicht leisten kann.

2.4 Def-Use Pruning

Da die vollständige Abdeckung des Fehlerraumes mit der Injektion aller IPs sehr zeitaufwendig ist, wie in Abschnitt 2.2 beschrieben, wird versucht mit einer effektiven Methode den Injektionsaufwand bei vollständiger Fehlerraumabdeckung zu verringern. Dabei werden Injektionspunkte so zusammengefasst, dass eine geringere Menge von Injektionspunkten die Defekte aller IPs repräsentiert. Eine solche Methode wird auch *Pruning* genannt.
Eine dieser Methoden ist das Def-Use Pruning, welches basierend auf Bildung von Äquivalenzklassen die Menge der IPs reduziert. Äquivalenzklassen fassen IPs in effektive und unwirksame Defekte zusammen.
Effektive Äquivalenzklassen beinhalten alle IPs, die das Potenzial besitzen sich in einen Fehlzustand fortzupflanzen. Dies setzt voraus, dass die defekte Stelle im Register oder im Hauptspeicher vom System gelesen und verarbeitet wird. Betrachtet man die Nutzung von Register und Hauptspeicher in einem System, so werden Schreib- und Leseoperationen auf diese ausgeführt. Eine Schreiboperation setzt das gewählte Datenwort auf das gewünschte Datum und überschreibt somit das Vorige. Eine Leseoperation liest das gespeicherte Datum. Das bedeutet für die Unterscheidung von effektiven und unwirksamen Defekten, dass nur ein Defekt, welcher sich zwischen einem Schreib- und dem letzten Lesezugriff befindet, in einen Fehlzustand fortzupflanzen kann [Bar+]. Demnach sind alle vor einer Schreiboperation nicht gelesenen Defekte unwirksam und können somit nicht zu einem Fehlzustand im System führen.
In Abbildung 2.3 ist die Aufteilung des Fehlerraums in Äquivalenzklassen dargestellt. Dabei sind die effektiven Äquivalenzklassen von den Unwirksamen farblich getrennt, sowie Schreib- und Lesezugriffe farblich als Grenzen der effektiven Äquivalenzklassen dargestellt. Eine weitere Eigenschaft der Äquivalenzklassen, die beim Def-Use Pruning genutzt wird, um die erforderlichen Injektionspunkte weiter zu verringern, ist eine genauere Betrachtung der effektiven Äquivalenzklassen. So bedeutet die Klassifizierung, dass jeder in einer effektiven Äquivalenzklasse existierende IP erst beim Lesezugriff in das System gelangt. Daraus folgt, dass nur ein IP pro Klasse injiziert werden muss, um alle möglichen Defekte der Klasse abzudecken. Als einziger IP pro Klasse wird der letztmögliche IP injiziert, da nur durch die Leseoperation der Defekt in das System gelangt ist es egal, welche vorigen IPs innerhalb einer Äquivalenzklasse verändert worden sind.
Die Methode Def-Use Pruning deckt somit mit deutlich weniger IPs den gesamten Fehlerraum ab [Bar+05] . Die in dieser Arbeit vorgestellte neue Methode baut auf den oben beschriebenen Eigenschaften des Def-Use Prunings auf, um weitere IPs einzusparen.

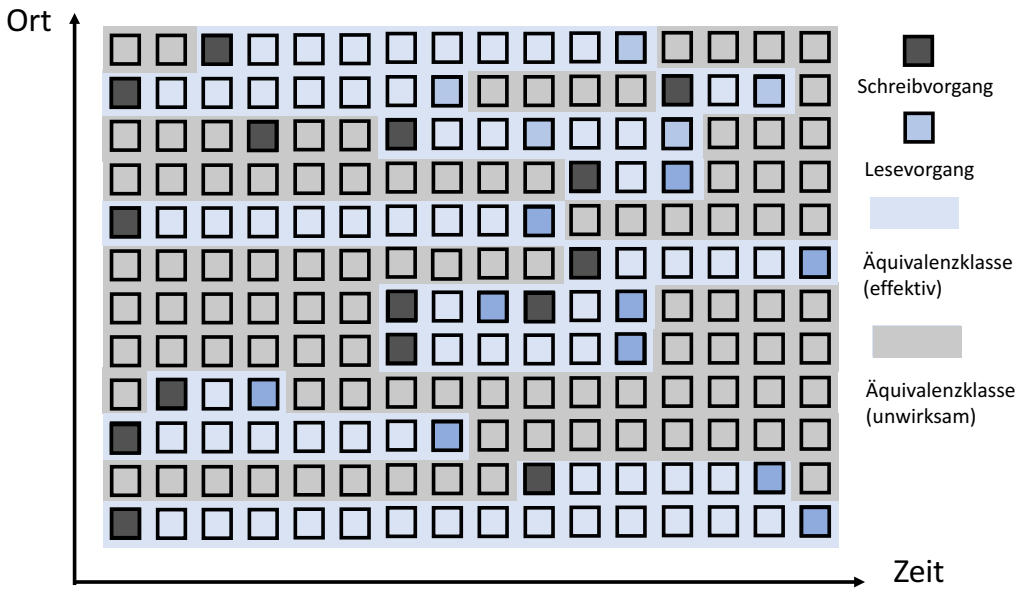


Abbildung 2.3 – Fehlerraum unterteilt in Äquivalenzklassen

3 STAND DER TECHNIK

Wie in Abschnitt 2.3 erwähnt, kann die Fehlerinjektion auch durch andere Techniken als die Simulationsbasierte Injektion durchgeführt werden. Als erstes werden weitere Techniken der Fehlerinjektion kurz beschrieben, um im weiteren Verlauf des Kapitels näher auf verwandte Arten des Prunings einzugehen.
Ein nahe liegendes Verfahren der Fehlerinjektion ist die Hardwarebasierte Fehlerinjektion, wobei das zu testende System **während der Laufzeit** erhöht Teilchenstrahlung [GKT89; San+14], elektromagnetischen Interferenzen [Kar+94] oder Spannungsschwankungen [Kar+91] ausgesetzt wird. Zudem gehört zur Hardwarebasierten Fehlerinjektion auch das Manipulieren von Prozessorpins [Mad+94]. Ein weiteres Verfahren ist die Softwarebasierte Fehlerinjektion. Dieses Verfahren injiziert mögliche Defekte **vor der Ausführung** in das System in Code- oder Datensegmente.
Ein Beispiel wären das FERRARI Tool [KKA95] oder das GOOFI Tool [Aid+01], welche Bitkipper in die Maschineninstruktionen injizieren, und bei der danach folgenden Ausführung das System beobachten. Ein weiteres Tool LLFI [TP13] injiziert mit Hilfe des LLVM Tools [LA04] Datentypen, Pointerzuweisungen und Kontrollpfade vor der Ausführung.
In der Simulationsbasierten Fehlerinjektion gibt es unterschiedliche Ansätze, auf welcher Ebene die Injektion im **laufenden System** stattfindet. Die Tools FIAT [Bar+90] und Segall [Seg+95] injizieren einzelne und mehrere Bitkipper in Code- oder Datensegmente der Zielanwendung. Das Tool DEFINE [KI94] implementiert einen Systemcall für den UNIX-Kern, um Bitkipper in Ausführungscode und Speicher zu injizieren. In dieser Arbeit wird das Tool FAIL* - FAult Injection Leveraged [Sch+12] verwendet, welches die Injektion mit Hilfe von Emulatoren in das Zielsystem durchführt.
Ein Ziel der Forschung ist es, nicht nur die Injektion zu ermöglichen, sondern den Testaufwand zu verringern. Demnach muss ein Pruning über die möglichen Defekte eine Menge von IPs zurück liefern, welche zu testen sind. Dazu gibt es verschiedene Ansätze, die hier kurz vorgestellt werden. Eine Art des Prunings bezieht sich auf eine *zielsystemunabhängige Injizierung*, welche über das Wissen der Hardware gezielt Flipflop-Gruppen injiziert [Ber+02]. Die hier betrachtete Art ist die *zielsystemabhängige Injizierung*, was bedeutet, dass Informationen über den Programmablauf genutzt werden, um die IPs auszuwählen. Eine dieser Pruningmethoden [Smi+95; GS95] wurde schon in den Grundlagen, Abschnitt 2.4 beschrieben. Eine andere Pruningmethode nutzt Informationen über kritische Funktionen im System, um nur in die Speicherbereiche, die diese Funktionen nutzen, zu injizieren [Gri+12].
Das Def-Use Pruning [Bar+] selbst wurde in mehreren Versionen implementiert. Die beiden bekanntesten sind die «Injection on write» [Ber+02; Ben+98] und «Injection on read» [Bar+; Bar+05] welche sich nur in dem Zeitpunkt der Injektion unterscheiden. «Injection on write» injiziert den Fehler beim Schreibbefehl und «Injection on read» beim Lesebefehl.

4 BASISBLOCK-PRUNING

Das Basisblock-Pruning unterscheidet sich zum bereits vorgestellten Def-Use Pruning (Abschnitt 2.4) dadurch, dass nun die effektiven Äquivalenzklassen zusammengefasst werden, die in einem Basisblock beginnen. Aus diesem Verbund werden dann einzelne repräsentative Injektionen ausgewählt, die injiziert werden. Das Ziel dieser Pruningmethode ist, die Anzahl der notwendigen Injektionen weiter zu verringern, ohne die Aussage des Ergebnisses im Vergleich zur Def-Use Pruningmethode statistisch signifikant zu verändern.

4.1 Basisblöcke

Der Begriff eines Basisblocks kommt aus der Compilerprogrammierung und bedeutet, dass ein Verbund von Instruktionen nur mit der ersten Instruktion betreten werden kann und mit der Letzten verlassen wird. Das Ende eines solchen Blockes ist immer eine Sprungentscheidung und der Beginn eines Blockes immer die nach einem Sprung folgende Instruktion.
Dieses bedeutet, für alle Instruktionen innerhalb der Grenzen eines solchen Verbundes die Abfolge der Instruktionen bei Programmausführung immer gleich ist. In Abbildung 4.2 sieht man, wie aus einem Programmcode die Basisblöcke entnommen werden. Auf der rechten Seite sind die Basisblöcke mit der schwarzen Umrandung hervorgehoben. Ein solcher Basisblock entspricht ebenfalls einem Knoten im Kontrollflussgraphen.

4.2 Pruning-Ansatz

In dieser Arbeit wird die Eigenschaft der sequenziellen Instruktionenabfolge innerhalb eines Basisblocks genutzt, um diese in Äquivalenzklassen einzuordnen. Entscheidend für die Einteilung der einzelnen Äquivalenzklassen zu den Basisblöcken ist der Begin einer Äquivalenzklasse (Datum wird geschrieben). Daraus folgt, dass alle Äquivalenzklassen die in einem Basisblock beginnen diesem zugeordnet werden. Abbildung 4.2, welche in der Grundform schon aus Kapitel 2 bekannt ist, wurde jetzt mit den Basisblockgrenzen erweitert. Nun sieht man, dass es Äquivalenzklassen gibt, die innerhalb eines Basisblocks beginnen und enden. Wenn in diesem Schreib- und Lesevorgang ein Bitkipper vorkommt, wird das veränderte Datum nicht in den nächsten Basisblock weitergetragen, was in Kontrollflussgraphen bedeutet, dass die defekte Stelle in einem Knotenpunkt verbleibt. Anders ist das bei Äquivalenzklassen, die den Basisblock, in dem sie beginnen, verlassen und somit herausragen. Bei diesen Äquivalenzklassen wird das Datum innerhalb des Basisblocks geschrieben und außerhalb in einem anderen Basisblock gelesen, dadurch kann ein möglicher Bitkipper sich im weiteren Programmablauf erst bemerkbar machen. Die herausragenden Äquivalenzklassen können somit

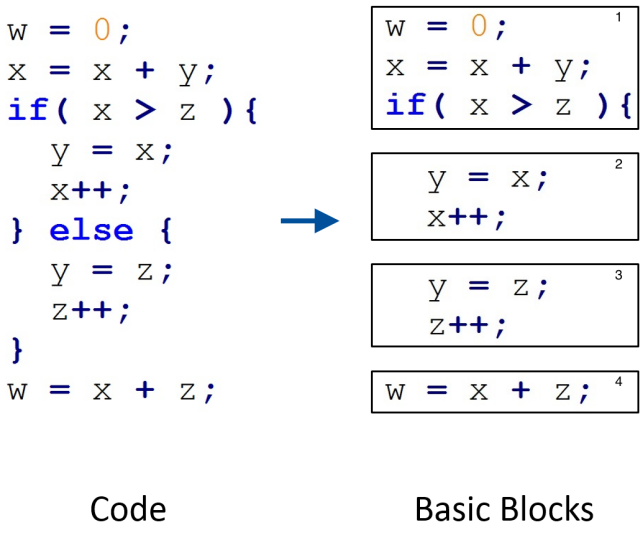


Abbildung 4.1 – Generierung der Basisblöcke aus vorhandenen Code

Bitkipper basisblockübergreifend beinhalten und somit die Funktionszuverlässigkeit beeinflussen. Genau an diesem Punkt soll das Basisblock-Pruning eine weitere Verminderung der IP bewirken. Beim Basisblock-Pruning sollen nur noch die Defekte der herausragenden Äquivalenzklassen überprüft werden, da diese sich in Form eines Fehlzustandes im weiteren Systemablauf auf Basis von Basisblöcken bemerkbar machen können.

4.3 Basisblöcke im Fehlerraum

Um das Basisblock-Pruning mit dem Def-Use Pruning zu vergleichen, müssen die Fehlerräume der beiden Methoden in erster Linie vollständig abgedeckt werden und zweitens müssen diese vergleichbar sein. Da durch das veränderte Fehlermodell beide einen anderen Fehlerraum aufspannen, muss eine Annäherung gefunden werden, welche diese unterschiedlichen Fehlerräume miteinander vergleichbar macht.
Ein Ansatz sind die genutzten Äquivalenzklassen, da diese unverändert in beiden Pruningmethoden verwendet werden können diese ein mögliches Gewicht beinhalten. Als Vergleichseinheit wird der Gewichtspunkt (GP) eingeführt, welcher mit einer Gewichtsfunktion berechnet wird. Als Gewicht einer Äquivalenzklasse wird die Zeit, gemessen in Anzahl von Zyklen, genommen, die zwischen Schreib- und Lesevorgang vergeht. Die daraus resultierenden GPs sind auch eine Gewichtung der einzelnen Äquivalenzklassen bzgl. ihrer Anfälligkeit auf Bitkipper, denn je mehr GPs eine Äquivalenzklasse bekommt, desto länger verweilt das geschriebene Datum im Register oder Hauptspeicher und desto länger ist die Zeit, in der ein Single Event Upset eintreten kann.
Im Folgenden werden die Gewichtsfunktionen formal beschrieben: Sei B die Menge aller Basisblöcke im Programm und $b_i \in B$ ein einzelner Basisblock. Ferner sei E die Menge aller Äquivalenzklassen,E_{b_i} die Menge der Äquivalenzklassen, die in einem Basisblock b_i beginnen, und $e_i \in E$ beschreibt eine einzelne Äquivalenzklasse. Die Äquivalenzklassen E_{b_i} lassen sich noch in $\overline{E}_{b_i}$, die Menge der

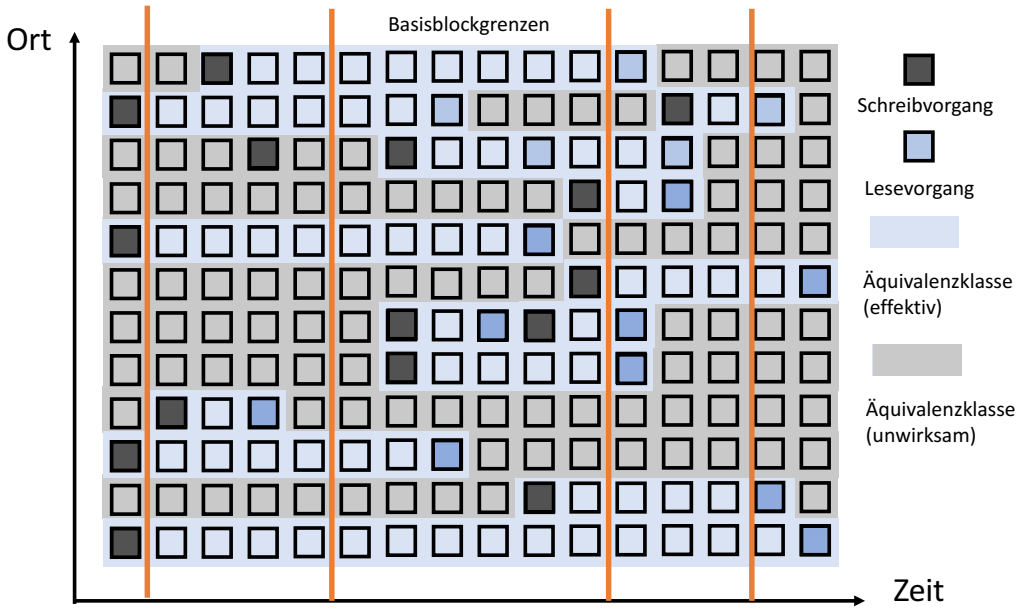


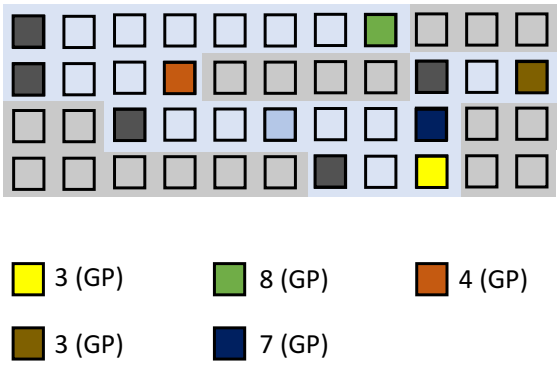
Abbildung 4.2 – Fehlerraum mit Basisblockgrenzen

herausragenden Äquivalenzklassen unterteilen, sodass die Mengen in einer zusammenhängenden Beziehung stehen: $\overline{E}_{b_i} \subseteq E_{b_i} \subseteq E$.
$g(x)$ ist die Gewichtsfunktion, die die GPs eines übergebenen Elements berechnet. Die Bestimmung der GPs für jede Injektion im Def-Use Pruning wird definiert durch die Differenz des Start- und Endzeitpunktes der zur Injektion zugehörigen Äquivalenzklasse [Bar+]. Die Berechnung des Gewichts für jede Injektion j im Basisblock-Pruning wird definiert als:

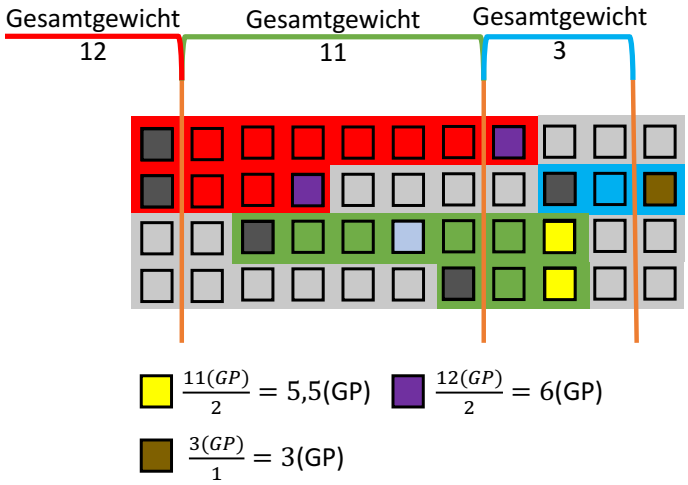
$$g(\mathrm{IP}_{b_i,j}) = \frac{\sum_k g(e_k)}{|\overline{E}_{b_i}|}, \text{ wobei } e_k \in E_{b_i} \text{ und } k = |E_{b_i}|$$

Beim Def-Use Pruning wird, wie bereits in Abschnitt 2.4 beschrieben, die Injektion auf die letzte Leseoperation innerhalb jeder Äquivalenzklasse gesetzt. Somit wird für jede Äquivalenzklasse eine Injektion durchgeführt und diese wird dann entsprechend gewichtet. Beim Basisblock-Pruning wird zuerst das Gesamtgewicht des Basisblocks bestimmt, welches sich aus der Summe der GPs der zu diesem Basisblock zugehörigen Äquivalenzklassen E_{b_i} zusammensetzt. Dieses Gesamtgewicht wird dann auf die Injektionen IP_{b_i} der herausragenden Äquivalenzklassen $\overline{E}_{b_i}$ des Basisblocks verteilt, welche (wie auch beim Def-Use Pruning) auf die letzte Leseoperation der jeweiligen Äquivalenzklasse gesetzt werden. In Abbildung 4.3 sind nochmals die Unterschiede der einzelnen Gewichtungen in einem kleinen Ausschnitt des Fehleraums grafisch dargestellt. In Abbildung 4.3a ist die Verteilung der Gewichte im Def-Use Pruning gezeigt. Jede letzte Instruktion einer effektiven Äquivalenzklasse

bekommt somit die GPs zugeteilt. Dies ist hier an den unterschiedlich gefärbten Punkten zu sehen. In der Abbildung 4.3b ist die Verteilung der Gewichte im Basisblock-Pruning gezeigt. Hier wird gezeigt, dass die einzelnen GPs der Äquivalenzklassen, die innerhalb eines Basisblocks beginnen (einheitliche Farbe) addiert werden und zu einem Gesamtgewicht des Basisblocks führen. Dieses Gesamtgewicht wird dann auf die ebenfalls einheitlich gefärbten Injektionspunkte gleichmäßig verteilt.
Durch die Einführung der Gewichte kann nun die Aussagekraft über die Funktionszuverlässigkeit des Systems vom Basisblock-Pruning mit der des Def-Use Prunings verglichen werden.



(a) Darstellung der Gewichtsfunktion der Äquivalenzklassen



(b) Darstellung der Gewichtsfunktion der Basisblöcke

Abbildung 4.3 – Vergleich der einzelnen Gewichtsfunktionen

5 EVALUATION

In diesem Kapitel wird als Erstes die Evaluationsumgebung beschrieben, um danach die Ergebnisse der Evaluation getrennt in Register- und Speicherinjektion zu betrachten.

5.1 Evaluationsumgebung

Der Abschnitt über die Evaluationsumgebung setzt sich aus der Beschreibung der verwendeten Benchmarksammlung und dem Framework FAIL* - FAult Injection Leveraged [Sch+12] zusammen. Das Framework bietet mit Hilfe des Bochs x86 Emulator [Law96] die Möglichkeit, Injektionen in Register und Hauptspeicher im Rahmen der simulationsbasierten Injektion vorzunehmen.

5.1.1 Benchmarksammlung

Für die Zusammenstellung der Benchmarksammlung wurden drei Kategorien berücksichtigt, um ein möglichst breites Spektrum abzudecken. In der ersten Kategorie wurden Programme konstruiert, die kleine und große Basisblöcke generieren. Die kleinen Basisblöcke sind im Short Basis Block Benchmark (SBB) mit Hilfe von fünf kleinen Funktionen generiert worden. Jede dort deklarierte Funktion ist ein eigener Basisblock mit einer Länge von 8 Zyklen. Jede Funktion in SBB wird 1001 mal ausgeführt.
Die großen Basisblöcke sind im Long Basis Block Benchmark (LBB) angelehnt an den SBB so generiert worden, dass dieselben Operationen wie im SBB ausgeführt werden. Dieses wurde durch das Zusammenfassen der kleinen Funktionen des SBB zu einer einzigen Funktion erreicht. Diese eine Funktion hat eine Länge von 36 Zyklen und wird ebenfalls 1001 mal ausgeführt. Die Differenz von 4 Zyklen gegenüber des SBB liegt an Optimierungen des Compilers.
Die zweite Kategorie beschäftigt sich mit unterschiedlichen Programmstrukturen. Ausgewählt wurden die Programmstrukturen einer Iteration und der Rekursion. Zur Untersuchung einer iterativen Programmstruktur wurde im Loop-Sum Benchmark (LSum) mit einem Schleifenkonstrukt eine Summe über Arrayeinträge, addiert mit einer Konstanten, berechnet. Dieses Benchmark wird in Listing 5.1 als Beispiel kurz dargestellt. Der Basisblock des Schleifenkonstrukts hat eine Länge von 14 Zyklen und wird 8 mal durchlaufen. Für eine rekursive Programmstruktur wurde im Fibonacci Benchmark (Fib) die Fibonacci-Folge gewählt. Die Fibonacci-Folge ist definiert als $f_n = f_{n-1} + f_{n-2}$ für $n > 2$ und $f_1 = 1; f_0 = 0$ und wurde rekursiv implementiert. Es werden $f(0)$, $f(1)$ und $f(10)$ innerhalb des Fib berechnet.
In der letzten Kategorie wurde in Anlehnung an die für eingebettete Systeme entwickelte Benchmarksammlung *miBench* [Gut+01] der QuickSort-Sortieralgorithmus ausgewählt. Der QuickSort

Algorithmus wurde in dieser Kategorie einmal als rekursive Version implementiert QSort(rekursiv) Benchmark (QSR) und als eine iterative Version QSort(iterativ) Benchmark (QSI). Beide Versionen sortieren Punkte in einem dreidimensionalen Raum, welche durch ein Dreiertupel $(x_1, x_2, x_3)^T$ beschrieben werden. Das Sortierkriterium ist die Distanz $d = \lfloor \sqrt{x_1^2 + x_2^2 + x_3^2} \rfloor$ zum Ursprung.
In der Tabelle 5.1 sind alle Benchmarks, die in dieser Arbeit verwendet werden, aufgelistet. Die Tabelle enthält für diese Benchmarks die Anzahl der im Benchmark vorhandenen Instruktionen, die Laufzeit des Benchmarks in Zyklen und die Anzahl der Basisblöcke.

```
1  int array[] = {1, 1, 2, 3, 5, 8, 13, 21};
2  int sum;
3  void os_main() {
4    sum = 20;
5
6    for (int i = 0; i < sizeof(array)/sizeof(*array); i++) {
7      sum += (array[i] * 23) + 1;
8    }
9
10   stop_trace();
11   if (sum != 1270)
12     fail_marker();
13 }
```

Listing 5.1 – Schleifenkonstrukt von LSum

5.1.2 FAIL*

Das Framework FAIL* - FAult Injection Leveraged [Sch+12] ist eine Plattform, welche es ermöglicht simulationsbasierte Injektion durchzuführen. FAIL* ist in einer Client-Server Architektur aufgebaut. Der FAIL*-Server verwaltet die einzelnen Injektionskampagnen, in der pro Benchmark die zu testenden IPs beschrieben sind. Zur Erstellung einer Injektionskampagne im FAIL*-Server wird das Benchmark einmal ohne Injektion ausgeführt, um einen Referenzlauf zu erhalten. Dieser Referenzlauf wird verwendet, um mögliche IPs für die Injektionskampagne zu generieren. Dazu wird ein Mitschnitt, auch bekannt als *Trace*, der Programmausführung gespeichert und analysiert. Die Analyse erfolgt nach dem in Abschnitt 2.2 beschriebenen Fehlermodell. Eine weitere Verwendung des Referenzlaufes ist, nach einer Injektion mögliches Fehlverhalten des Benchmarks zu erkennen. Zudem ist in dem FAIL*-Server eine Pruningeinheit implementiert, welche die aus dem Referenzlauf analysierten IPs nach der ausgewählten Pruningmethode reduziert. Diese Informationen speichert der FAIL*-Server in eine Datenbank ab und verteilt einzelne Injektionsexperimente an den FAIL*-

Benchmark	$\sum$ Instruktionen	Laufzeit (Zyklen)	$\sum$ Basisblöcke
LBB	86	15024	5
LSum	46	55	7
Fib	73	3245	12
SBB	91	65072	22
QSR	221	3942	31
QSI	246	3848	34

Tabelle 5.1 – *Vergleich der verwendeten Benchmarks:* Übersicht der Benchmarks sortiert nach der Anzahl der Basisblöcke

Client. Der FAIL*-Client holt sich ein Injektionsexperiment vom FAIL*-Server, welches einen IP enthält und simuliert mit Hilfe eines Emulators, im Rahmen dieser Arbeit ist das der Bochs x86 Emulator [Law96], den Benchmark, injiziert an der entsprechenden Stelle (Ort, Zeit) den Defekt und sendet den emulierten Programmverlauf zurück zum FAIL*-Server. Diese Struktur erlaubt es, im FAIL*-Server die Injektionskampagne anzulegen und die Simulation durch mehrere FAIL*-Clients zu parallelisieren. Der FAIL*-Server vergleicht den emulierten Programmablauf vom Client mit dem Referenzlauf und speichert das Ergebnis in eine Datenbank.
Das Framework FAIL* unterteilt das Ergebnis des Programmablaufs (mit Injektion) in mehrere Fehlerklassen:

- **OK Marker (OKM):** Kein Fehlverhalten festgestellt. Der Programmverlauf entspricht dem Referenzlauf.
- **Fail Marker (FAILM):** Ein Fehlverhalten ist eingetreten. Das Programm ist termininiert. Das entstandene Fehlverhalten ist nicht ohne Weiteres von außen sichtbar (Silent Data Corruption).
- **Timeout (Tout):** Ein Fehlverhalten ist eingetreten, und der Programmablauf wurde aufgrund einer durch den Defekt entstandenen Endlosschleife beendet.
- **Trap (Tp):** Ein Fehlverhalten ist eingetreten und das System meldete einen Trap (Tp).
- **Write textsegment (Wtext):** Ein Fehlverhalten ist eingetreten und der Programmablauf wollte unbefugt ins Textsegment des Programms schreiben.
- **Write outer space (WoS):** Ein Fehlverhalten ist eingetreten und der Programmablauf wollte unbefugt außerhalb des zugewiesenen Speicherbereichs schreiben.

Für die Bewertung der Eigenschaft Funktionszuverlässigkeit ist nur die Unterscheidung zwischen *kein Fehlverhalten konnte festgestellt werden* und *es ist ein Fehlverhalten eingetreten* notwendig, da es, wie in Abschnitt 2.1 beschrieben, dort nur um die fehlerfreie Ausführung der Funktion unter Berücksichtigung der Rahmenbedingungen geht. Die genauere Beschreibung, welches Fehlverhalten aufgetreten ist, hilft dem Entwickler mögliche Schwachstellen besser zu lokalisieren. Eine besondere Fehlerklasse dabei ist die FAILM, da diese einen Fehler beschreibt, der ein Fehlverhalten des Systems beschreibt, das von außen jedoch nicht sofort zu erkennen ist. Eine solche silent data corruption kann zu einer Fehlwahrnehmung des Systemzustands führen und ein hohes Risiko im Sinne der Sicherheit (Abschnitt 2.1) darstellen.

5.2 Ergebnisse

In diesem Abschnitt werden die Ergebnisse, aufgeteilt in Register- und Speicherinjektion, der Evaluation gezeigt.
Jedes Ergebnis eines Benchmarks wird mit zwei Balkendiagrammen beschrieben. Das erste Diagramm zeigt die Ergebnisse im Zusammenhang mit den aus den Pruningmethoden ermittelten benötigen Injektionen. Die x-Achse ist dabei in die einzelnen Fehlerklassen aufgeteilt (siehe Auflistung in Abschnitt 5.1.2), der letzte Achsenabschnitt entspricht immer der jeweiligen Summe aller Fehlerklassen eines Diagramms. Verursacht also eine Injektion einen Timeout, wird diese Injektion zur Spalte Tout der Pruningmethode angerechnet. Die y-Achse zeigt die Anzahl der durchgeführten Injektionen.
Im zweiten Diagramm werden die Ergebnisse im Zusammenhang mit der in Abschnitt 4.3 beschriebenen Gewichtsfunktion dargestellt. Auch im zweiten Diagramm ist die x-Achse in die einzelnen

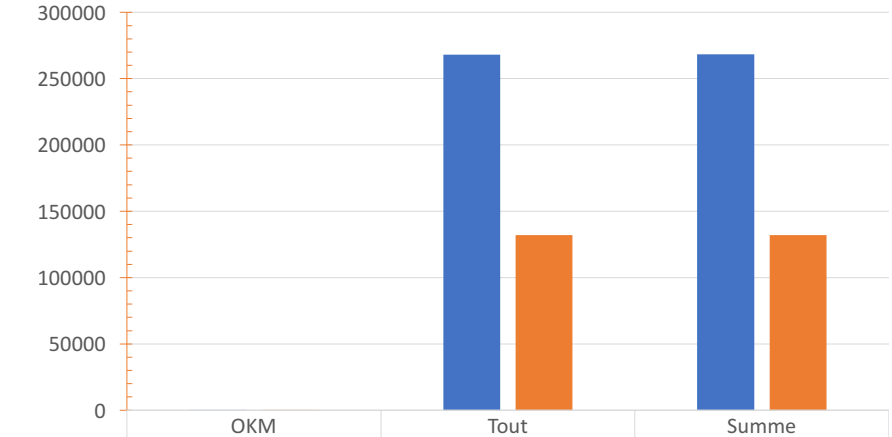
Fehlerklassen aufgeteilt, der letzte Achsenabschnitt entspricht wieder der Summe der vorigen Werte. Die y-Achse zeigt die Summe der GPs der einzelnen Fehlerklassen.
In beiden Diagrammen sind die Ergebnisse des Def-Use Prunings in blau und die des Basisblock-Prunings in orange dargestellt. Unter den Diagrammen werden in einer Tabelle für jeden x-Achsenabschnitt der konkrete Wert der y-Achse nochmals dargestellt. Demnach wird eine Injektion, die aufgrund von Basisblock-Pruning durchgeführt wird und zu keinem Fehlverhalten geführt hat, in den orangefarbene Balken der Fehlerklasse OKM einfließen. Der Injektionspunkt als solcher wird im ersten Diagramm im Balken dargestellt und das Gewicht, welches er von der Gewichtsfunktion (siehe Abschnitt 4.3) bekommen hat, im zweiten Diagramm. Es werden aufgrund der Fülle an Daten und Diagrammen nur repräsentative Ergebnisse der Evaluation in diesem Abschnitt gezeigt. Alle Ergebnisse der Evaluation sind im Anhang vorhanden.

5.2.1 Ergebnisse der Registerinjektion

In diesem Unterpunkt werden die Diagramme, welche die Ergebnisse der Registerinjektion der Benchmarks beschreiben, vorgestellt.

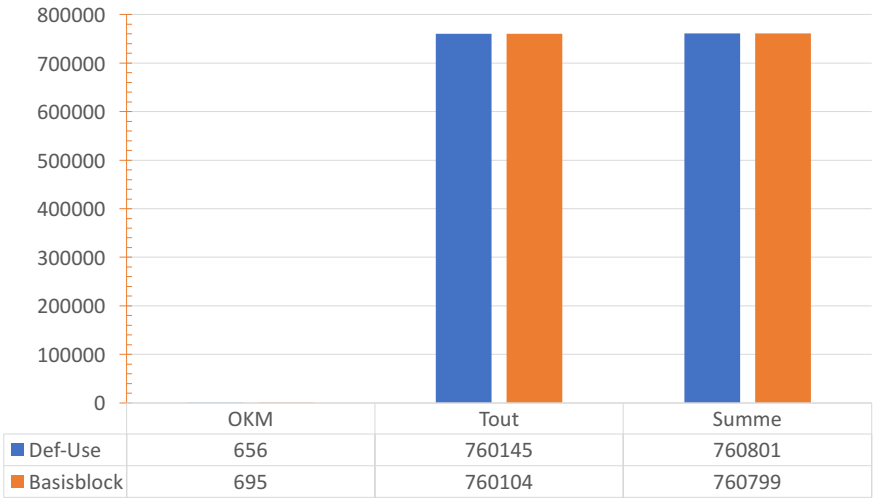
Short Basis Block Benchmark (SBB)

Im SBB sieht man bei der Injektionsverteilung (Abbildung 5.1), dass das Basisblock-Pruning nur 50, 7% der Injektionen vom Def-Use Pruning durchführen muss. Auch zu sehen ist, dass die Einsparung über alle Fehlerklassen nahezu gleichverteilt verläuft.



	OKM	Tout	Summe
Def-Use	200	268093	268293
Basisblock	108	132036	132144

Abbildung 5.1 – *Vergleich der Injektionen im Short Basis Block Benchmark (SBB):* Hier zeigt sich, dass die Einsparung der IPs rund 50% ausmachen und sich gleichmäßig auf die Fehlerklassen aufteilen.



	OKM	Tout	Summe
Def-Use	656	760145	760801
Basisblock	695	760104	760799

Abbildung 5.2 – *Vergleich der Gewichte im Short Basis Block Benchmark (SBB):* Trotz unterschiedlicher Anzahl von Injektionen verhalten sich die Gewichte beider Pruningmethoden nahezu identisch.

Schaut man sich dazu die Verteilung der Gewichte (Abbildung 5.2) an, so wird gezeigt, dass das Basisblock-Pruning trotz der geringeren Anzahl der Injektionen den Fehlerraum des Def-Use Pruning auch über die Fehlerklassen hinweg sehr gut abdeckt. Die kleine Abweichung von 2 Gewichtseinheiten in der Summe sind auf Rundungsfehler zurückzuführen. Ein kleiner Unterschied der Gewichte ist in den einzelnen Fehlerklassen zu sehen. Im Fall des SBB verlagern sich knapp 41 GPs von der Fehlerklasse Tout zum OKM. Was in diesem Falle bedeutet, dass System beim Basisblock-Pruning funktionszuverlässiger eingeschätzt worden ist als beim Def-Use Pruning.

Long Basis Block Benchmark (LBB)

Im LBB sieht man in dem Vergleich der IPs in Abbildung 5.3, dass eine Reduzierung von 78,9% der IPs durch Basisblock-Pruning möglich ist. Auch die Verteilung der Reduzierung ist nahezu gleich bezüglich der einzelnen Fehlerklassen.
Schaut man sich dann die Verteilung der Gewichte in Abbildung 5.4 an, wird auch beim LBB der vom Def-Use Pruning abgedeckte Fehlerraum mit dem Basisblock-Pruning in der Summe abgedeckt. Auch hier ist die minimale Abweichung der Gesamtsummen auf Rundungsfehler zurückzuführen. Hier zeigt sich in den einzelnen Fehlerklassen im Basisblock-Pruning der umgekehrte Fall zum SBB. Durch die Verschiebung von 66% der GPs der Fehlerklasse OKM in die Fehlerklasse Tout wird das System unzuverlässiger eingeschätzt, als im Def-Use Pruning.

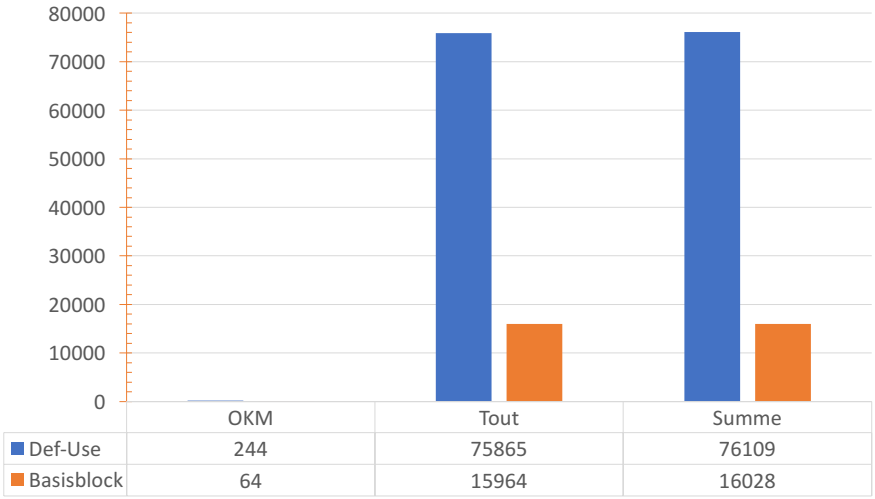


Abbildung 5.3 – *Vergleich der Injektionen im Long Basis Block Benchmark (LBB):* Bei der Gegenüberstellung der Injektionen ist die Differenz der gesamten IPs bei knapp 79%, auch hier ist die Injektionsreduzierung fast gleichmäßig auf die Fehlerklassen aufgeteilt.

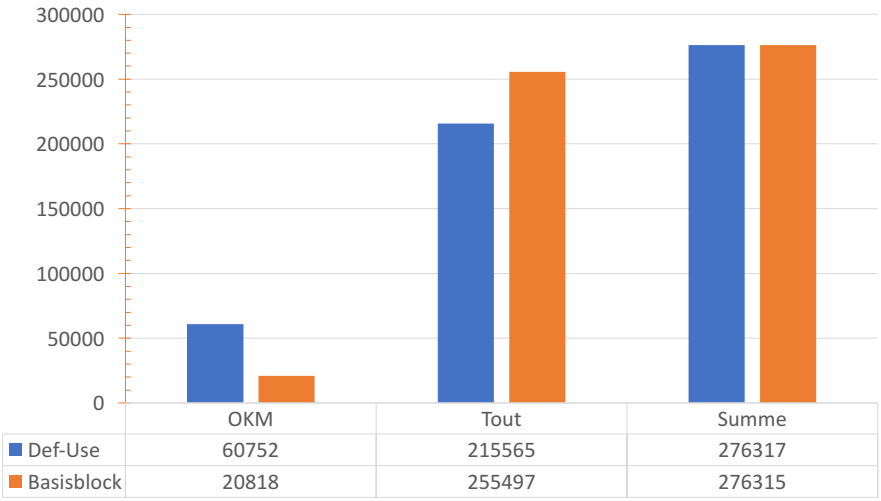


Abbildung 5.4 – *Vergleich der Gewichte im Long Basis Block Benchmark (LBB):* Die Gewichtsverteilung zeigt, dass die Summe der Gewichte beider Pruningmethoden nahezu identisch sind. Nur in den Fehlerklassen hat eine Verschiebung stattgefunden.

QSort(rekursiv) Benchmark (QSR)

Im QSR zeigt der Vergleich der Injektionen (Abbildung 5.5) eine Reduzierung der Injektionen durch das Basisblock-Pruning um 62,5%. Anders als bei den Benchmarks SBB und LBB sind die Reduzierungen in den jeweiligen Fehlerklassen unterschiedlich. In der Fehlerklasse Tout wurden 63,2% IPs eingespart, in der Fehlerklasse FAILM sind es 42% eingesparte IPs und in der Fehlerklasse OKM sind es 42,8% eingesparte IPs.
In der Gewichtsverteilung (Abbildung 5.6) des QSR sind die Gesamtsummen der beiden Pruningmethoden um 27 GPs unterschiedlich. Auch dieser Unterschied ist auf Rundungsfehler zurückzuführen, da im QSR eine höhere Anzahl von Wiederholungen einzelner Basisblöcke den Rundungsfehler aufsummiert. Schaut man sich die Verteilung der Gewichte in den einzelnen Fehlerklassen an, ist zu erkennen, dass wie schon beim LBB durch die verringerten GPs bei der Fehlerklasse OKM im Basisblock-Pruning das System unzuverlässiger eingeschätzt wird als beim Def-Use Pruning.

5.2.2 Ergebnisse der Speicherinjektion

In diesem Abschnitt werden die Diagramme, welche die Ergebnisse der Speicherinjektion der Benchmarks beschreiben, vorgestellt.

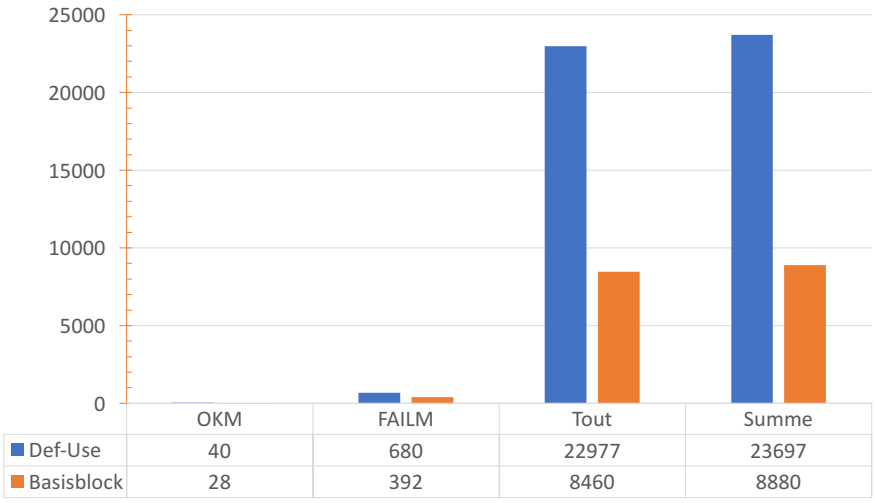
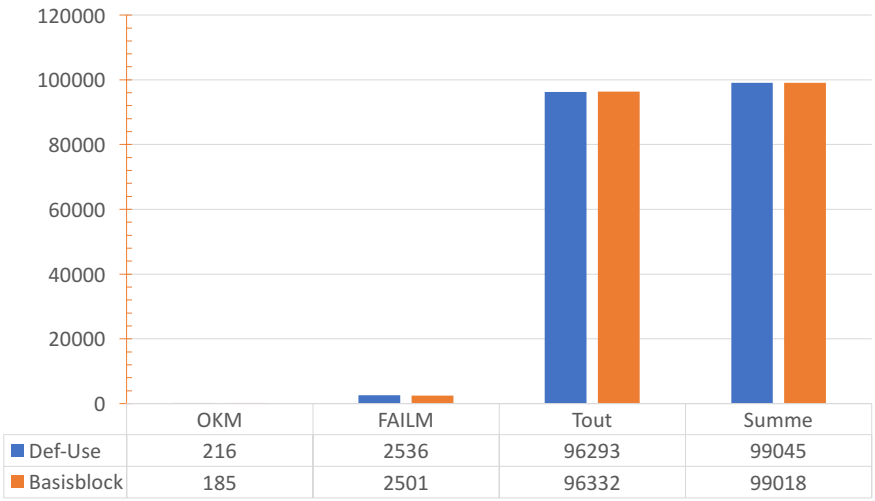
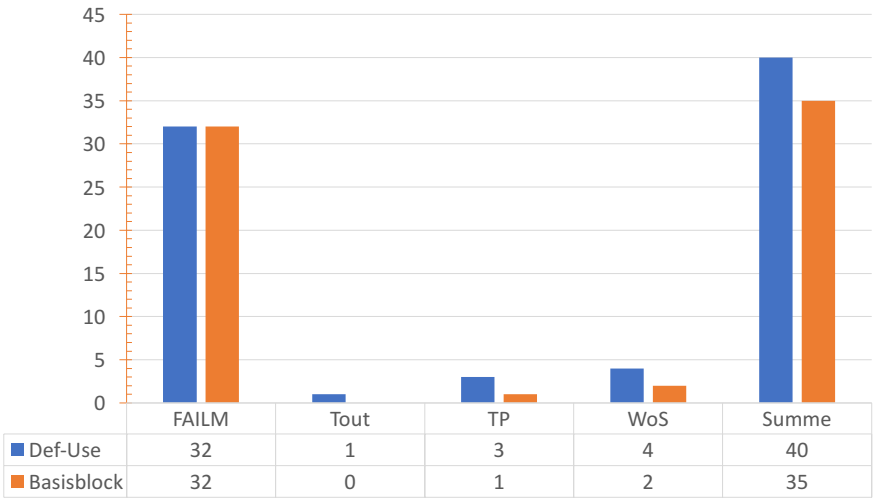


Abbildung 5.5 – *Vergleich der Injektionen im QSort(rekursiv) Benchmark (QSR):* Bei der Anzahl der Injektionen ergibt sich eine Differenz von rund 63%. Die Einsparungen an Injektionen und deren Einordnung in die einzelnen Fehlerklassen unterscheiden sich hier um: *OKM:* 30%, *FAILM:* 42% und *Tout:* 63%



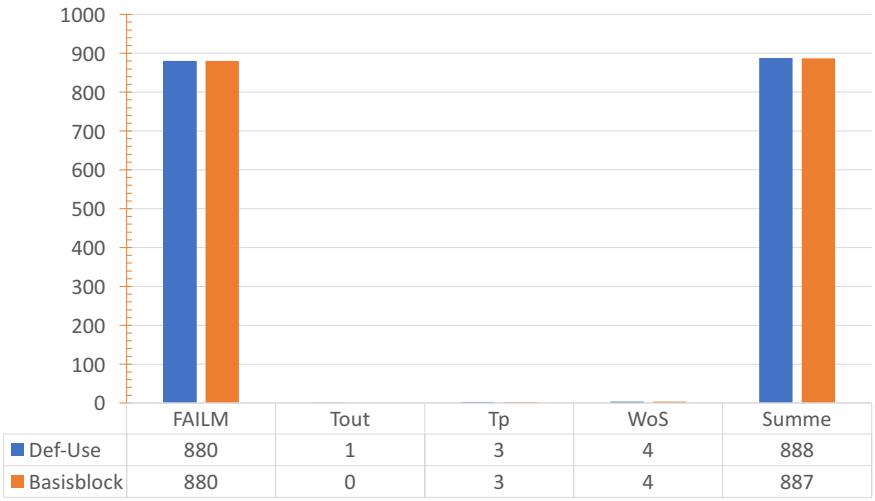
	OKM	FAILM	Tout	Summe
Def-Use	216	2536	96293	99045
Basisblock	185	2501	96332	99018

Abbildung 5.6 – *Vergleich der Gewichte im QSort(rekursiv) Benchmark (QSR):* Bei der Gewichtsverteilung ist die Summe hier nicht identisch, nähert sich aber stark an. Bei den Fehlerklassen ist eine Verlagerung der Gewichte von OKM und FAILM zu Tout zu erkennen.



	FAILM	Tout	TP	WoS	Summe
Def-Use	32	1	3	4	40
Basisblock	32	0	1	2	35

Abbildung 5.7 – *Vergleich der Injektionen im Loop-Sum Benchmark (LSum):* Bei der Anzahl der Injektionen liegt der Vorteil des Basisblock-Prunings bei genau 12,5%. Zudem sind bei der Fehlerklasse *FAILM* keine Einsparungen vorhanden.



	FAILM	Tout	Tp	WoS	Summe
Def-Use	880	1	3	4	888
Basisblock	880	0	3	4	887

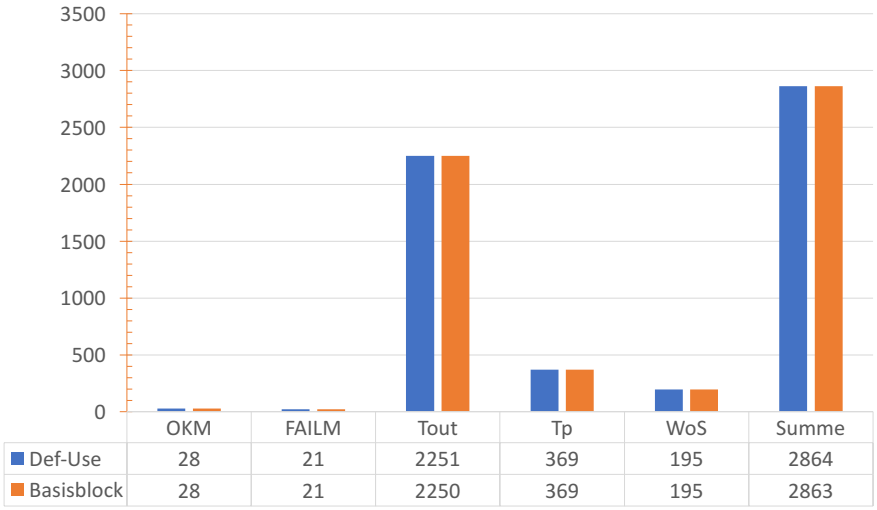
Abbildung 5.8 – *Vergleich der Gewichte im Loop-Sum Benchmark (LSum):* Die Gewichtsverteilung der beiden Pruningmethoden ist, außer bei der Fehlerklasse *Tout*, identisch.

Loop-Sum Benchmark (LSum)

Im LSum zeigt der Vergleich der Injektionen (Abbildung 5.7), dass die Reduzierung der IPs genau 12, 5% beträgt. Auffällig ist, dass bei der Fehlerklasse FAILM keine weitere Reduktion möglich ist. In der Fehlerklasse WoS ist eine Reduzierung der IPs von 50, 0% zu sehen und bei der Fehlerklasse Tp sogar von 66, 6%. Der IP in der Fehlerklasse Tout wird durch das Basisblock-Pruning nicht berücksichtigt.
In der Gewichtsverteilung in Abbildung 5.8 ist zu sehen, dass die Verteilung der GPs auf die einzelnen Fehlerklassen der beiden Pruningmethoden nahezu identisch sind. Nur der GP in der Fehlerklasse Tout im Basisblock-Pruning fehlt. Das ist darauf zurückzuführen, dass die einzige Injektion beim Pruning herausgefallen ist. Diese minimale Veränderung hat jedoch kein Einfluss auf die Aussage der Funktionszuverlässigkeit.

Fibonacci Benchmark (Fib)

Im Fib zeigt der Vergleich der Injektionen in Abbildung 5.9, dass die Reduzierung durch das Basisblock-Pruning nur eine Injektion beträgt. Diese sehr geringe Einsparung findet in der Fehlerklasse Tout statt. Bei dem Vergleich der Gewichtsverteilung (Abbildung 5.10) ist die Summe beider Pruningmethoden so gut wie identisch. Der Unterschied von 1 GP ist wieder auf Rundungsfehler zurückzuführen. Somit deckt das Basisblock-Pruning wieder den Fehlerraum des Def-Use Prunings vollständig ab.
Schaut man sich jedoch die Gewichte der einzelnen Fehlerklassen an, dann wird durch die Verschiebung von 90, 3% der GPs von der Fehlerklasse OKM zu den anderen Fehlerklassen die Aussage über



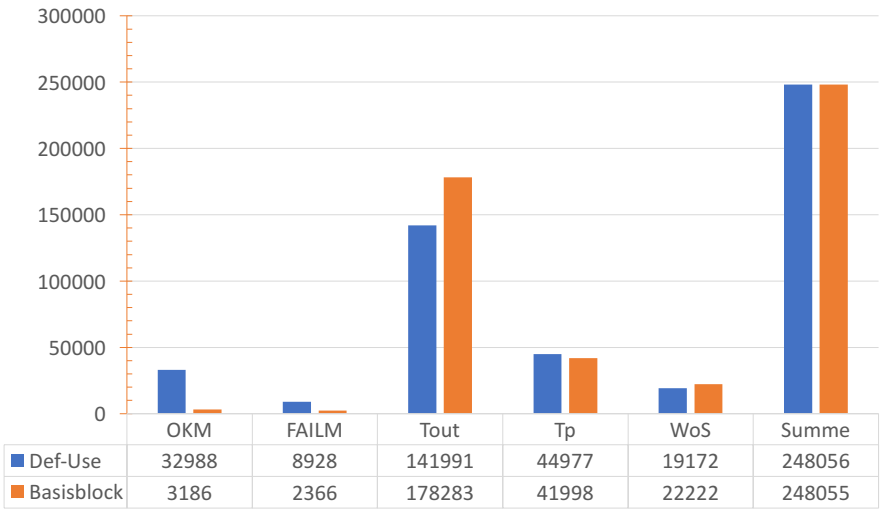
	OKM	FAILM	Tout	Tp	WoS	Summe
Def-Use	28	21	2251	369	195	2864
Basisblock	28	21	2250	369	195	2863

Abbildung 5.9 – *Vergleich der Injektionen im Fibonacci Benchmark (Fib):* Bei der Anzahl der Injektionen ist nahezu keine Differenz zu sehen. Bei der Fehlerklasse *Tout* ist eine Injektion beim Basisblock-Pruning weniger notwendig.

die Funktionszuverlässigkeit beim Basisblock-Pruning deutlich zum Negativen verändert. Außerdem verlieren die Fehlerklassen FAILM und Tp an GPs.

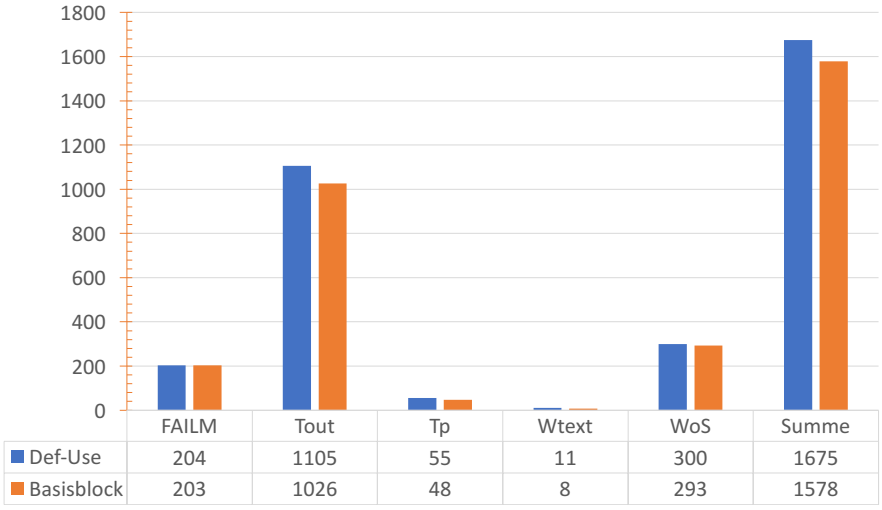
QSort(iterativ) Benchmark (QSI)

Im QSI zeigt der Vergleich der Injektionen (Abbildung 5.11), dass durch Basisblock-Pruning $5,8\%$ der Injektionen eingespart werden können. Wobei bei der Fehlerklasse FAILM nur eine kleine Reduzierung von einer Injektion möglich ist. Bei den anderen Fehlerklassen liegen die Reduktionen zwischen $2,4\% - 28,0\%$. In der Darstellung der Gewichte in Abbildung 5.12 sind die Summen der Gewichte der Pruningmethoden wieder identisch. Doch in den einzelnen Fehlerklassen gibt es wieder große Unterschiede.
Die Fehlerklassen FAILM und Tp verlieren $46,4\%$ ihres Gewichts und die Fehlerklassen Tout, Wtext und WoS gewinnen an Gewicht. Diese Verschiebung hat jedoch keinen Einfluss auf die allgemeine Aussage der Funktionszuverlässigkeit, jedoch verfälscht sie die Aussage über die Fehlzustände, die zu einem Fehler geführt haben.



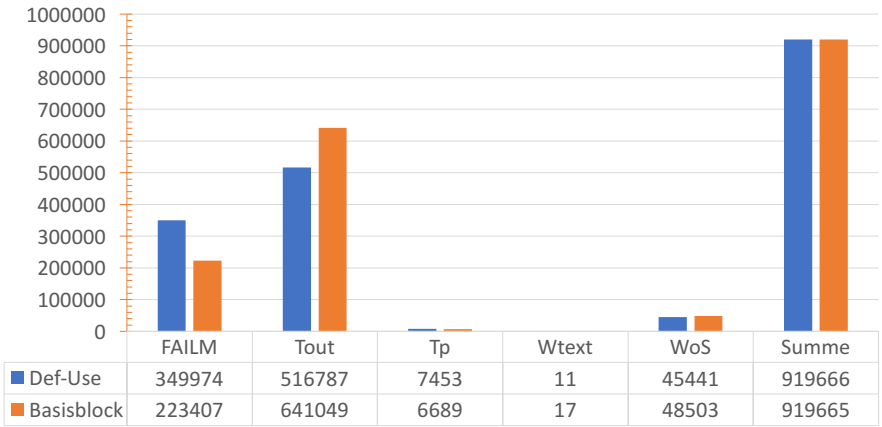
	OKM	FAILM	Tout	Tp	WoS	Summe
Def-Use	32988	8928	141991	44977	19172	248056
Basisblock	3186	2366	178283	41998	22222	248055

Abbildung 5.10 – *Vergleich der Gewichte im Fibonacci Benchmark (Fib):* Trotz nahezu identischer Gesamtgewichte der beiden Pruningmethoden unterscheiden sie sich teilweise stark in den einzelnen Fehlerklassen. Bei den Fehlerklassen *OKM*, *FAILM* und *Tp* fällt das Gewicht des Basisblock-Pruning gegenüber des Def-Use Pruning. Wobei bei den Fehlerklassen *Tout* und *WoS* das Gewicht steigt.



	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	204	1105	55	11	300	1675
Basisblock	203	1026	48	8	293	1578

Abbildung 5.11 – *Vergleich der Injektionen im QSort(iterativ) Benchmark (QSI):* Bei dem Vergleich der Injektionen ist eine Ersparnis von rund 6% zu sehen. Die Einsparungen der einzelnen Fehlerklassen unterscheiden sich leicht.



	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	349974	516787	7453	11	45441	919666
Basisblock	223407	641049	6689	17	48503	919665

Abbildung 5.12 – *Vergleich der Gewichte im QSort(iterativ) Benchmark (QSI):* Die Gesamtgewichte der beiden Pruningmethoden sind nahezu identisch. Bei den Fehlerklassen kommt es dennoch teilweise zu starken Schwankungen.

6 DISKUSSION

Im Abschnitt 5.2 wurden die Ergebnisse der einzelnen Bechmarks vorgestellt. Nachfolgend werden diese im Kontext dieser Arbeit erläutert, um eine Einschätzung über die Güte der Fehlerraumapproximation *Basisblock-Pruning* zu geben. Im ersten Abschnitt wird auf die Registerinjektion eingegangen. Im zweiten Abschnitt geht es dann um die Speicherinjektion. Der letzte Abschnitt fasst die Einschätzungen zu beiden Injektionsarten zusammen.

6.1 Registerinjektion

Im Bereich der Registerinjektion war zu sehen, dass eine erhebliche Reduktion der Injektionen für das Def-Use Pruning durch das Basisblock-Pruning möglich war. Im SBB waren es 50, 7%, im QSR sogar 62, 2% und die größte Reduktion fand bei den LBB mit 78, 9% statt, ohne dabei signifikante Änderungen über die Funktionszuverlässigkeit des Systems zu erzeugen.
Dieses wird auch über Benchmarks hinweg deutlich. Die Reduzierung der Injektionen durch das Basisblock-Pruning führt nicht dazu, dass sich die Fehlerraumabdeckung in Bezug auf das Def-Use Pruning statistisch signifikant ändert. Ein gutes Beispiel dafür ist Abbildung 5.2. Dort sind auch keine Abweichungen zwischen den einzelnen Fehlerklassen zu sehen. Dies zeigt klar, dass trotz der geringeren Anzahl an Injektionen die Aussage der Funktionszuverlässigkeit dieselbe bleibt.
Auch zu beobachten war, dass es trotz einer nahezu gleichen Abdeckung zu Unterschieden zwischen den einzelnen Fehlerklassen kommen kann. Das kann man beim LBB in Abbildung 5.4 gut erkennen. Hier hat die Reduktion der Injektionen dazu geführt, dass das System in der Aussage über die Funktionszuverlässigkeit gegenüber dem Def-Use Pruning schlechter eingeschätzt wird. Ob diese schlechtere Einschätzung bei einer Reduktion von 78, 9% der IPs die Güte der gesamten Fehlerraumapproximation einschränkt, ist hier noch nicht zu erkennen. Auch beim QSR ist die Aussage über die Funktionszuverlässigkeit im Basisblock-Pruning schlechter eingeschätzt worden. Hier ist die Verschlechterung aber deutlich geringer als beim LBB.
Zusammenfassend bestätigt sich, dass das Basisblock-Pruning in den Registern zu einer erheblichen Reduktion der Injektionspunkte führt und dabei die Aussage über die Funktionszuverlässigkeit nicht signifikant verändert. Die Gegebenheit, dass im Basisblock-Pruning eine schlechter eingeschätzte Funktionszuverlässigkeit des Systems gezeigt wird, ist zwar ein Unterschied zum Def-Use Pruning aber für die Bewertung der Pruningmethode nicht sonderlich erheblich.

6.2 Speicherinjektion

Im Bereich der Speicherinjektion war zu sehen, dass die Reduktion der Injektionen deutlich geringer ausgefallen ist als bei der Registerinjektion. Dieser Unterschied liegt darin begründet, dass Daten, die in den Speicher geschrieben werden, meist länger bis zum erneuten Lesen gespeichert bleiben, als in der Registerebene, welche für kürzere Intervalle zwischen Lesen und Schreiben benutzt wird. Demnach sind auch die Äquivalenzklassen in ihrer Länge größer und ein Überschreiten der Basisblockgrenzen passiert schneller als bei der Registerinjektion.
Da in den Ergebnissen der Speicherinjektion keine klare Tendenz zur Qualität des Basisblock-Prunings zu sehen ist, wird im Folgenden als erstes darauf eingegangen, wo sich Unterschiede zeigen, um danach auf mögliche Gründe dafür einzugehen.

Einschätzung der Ergebnisse

Schaut man sich die Abbildungen für LSum an, so sieht man in Abbildung 5.7, dass eine Reduktion von 12, 5% möglich ist und die GP in Abbildung 5.8 sich in beiden Pruningmethoden kaum unterscheiden; nur der GP der reduzierten Injektion in Tout fehlt. Somit verhält sich LSum identisch zu den Benchmarks in der Registerinjektion. Leider zeigen sich bei den anderen Benchmarks deutliche Unregelmäßigkeiten in der Verschiebung der GPs. Beim Vergleich der Injektionen in Abbildung 5.9 ist, wie bei LSum, der Testaufwand um nur eine Injektion geringer. Aber anders als zuvor verhalten sich die Gewichtsverteilungen der einzelnen Fehlerklassen (Abbildung 5.10) komplett anders, sodass eine Verschiebung von 90, 3% der GPs in der Fehlerklasse OKM zu der Fehlerklasse FAILM die Folge ist. Diese deutliche Veränderung führt zu einer verfälschten Aussage über die Funktionszuverlässigkeit und zeigt, dass das Basisblock-Pruning in diesem Benchmark ein anderes Verhalten zeigt als bei der Registerinjektion. Auch bei QSI ist eine solche Verschiebung zu erkennen. Hier wurden 2, 8% der Injektionen vom Def-Use Pruning reduziert (Abbildung 5.11), was aber im Vergleich zu den 50, 7%−78, 9% der Registerinjektionen sehr wenig ist. Bei der Gewichtsverteilung in Abbildung 5.12 ist bei QSI zwar die Aussage über die Funktionszuverlässigkeit nicht verändert, da keine GPs in der Fehlerklasse OKM vorliegen, aber die Verschiebung in den anderen Fehlerklassen sind wieder deutlich zu erkennen. Eine interessante Verschiebung hierbei ist die der Fehlerklasse FAILM zur Fehlerklasse Tout, da dieser genau den Übergang von einer Silent data corruption zu einem von außen erkennbaren Funktionsausfall zeigt. Somit werden beim Basisblock-Pruning weniger Silent data corruptions durch Injektionen erzeugt als beim Def-Use Pruning. Somit wird bei der Speicherinjektion sowohl die Einschätzung der Funktionszuverlässigkeit als auch der Schwerpunkt, bezüglich des Fehlverhaltens verändert. Dieses wirft nunmehr die Frage auf, warum es bei der Speicherinjektion diese großen Verschiebungen gibt und nicht bei der Registerinjektion.

Gründe für die Verschiebung der GPs

Um Gründe für diese Verschiebungen bei der Speicherinjektion zu suchen, wurden die herausragenden Äquivalenzklassen, die im Basisblock-Pruning als einziges injiziert wurden, genauer betrachtet. Diese Untersuchung dient dazu zu zeigen, ob ein Zusammenhang zwischen den Verschiebungen in der Gewichtsverteilung und den Längen der herausragenden Äquivalenzklassen besteht. Dazu wurden für alle herausragenden Äquivalenzklassen in einem Benchmark die Einzelgewichte $g(e_i)$ berechnet und in Boxplots dargestellt. Die Boxplots zeigen die Verteilung der GPs jeder herausragenden Äquivalenzklasse $g(e_i)$, wobei $e_i \in \overline{E}$ die Menge herausragender Äquivalenzklassen über alle Basisblöcke sind und $g(x)$ die Gewichtsfunktion aus Abschnitt 4.3 ist. Die Box entspricht dem Bereich, in dem die mittleren 50% der GP liegen. Sie wird durch das untere und obere Quartil

begrenzt. Das Kreuz beschreibt den Mittelwert aller GPs und die durchgezogene Linie den Median. Die Whiskers sind beschränkt durch das 1,5-fache des Interquartilsabstands. Punkte außerhalb der Whisker zeigen Ausreißer. Schaut man sich nun die Verteilung der Einzelgewichte in LSum für Speicherinjektion an (Abbildung 6.1), dann ist zu sehen, dass 50% der Einzelgewichte sich zwischen 15 und 35 GPs bewegen und es keine Ausreißer gibt. Zudem zeigt sich eine klare Symmetrie, welche auf eine gleichmäßige Verteilung der Einzelgewichte schließen lässt. Im Fib sind die Einzelgewichte der herausragenden Äquivalenzklassen sehr verstreut, was in Abbildung 6.2 sehr gut zu sehen ist. Hier sind die mittleren 50% der Einzelgewichte zwar sehr klein, die Box geht von $5,5$ (unteres Quartil) bis 48 GPs (oberes Quartil), der Median liegt bei 11 GPs, aber es gibt eine Vielzahl von Ausreißern, welche sich auf den Mittelwert von $86,61$ GPs auswirken, sodass dieser nicht mehr in der Box liegt. Diese große Streuung der Einzelgewichte kann dazu führen, dass beim Basisblock-Pruning die Gewichtsverteilung nicht mehr genau genug abgebildet wird. Dies könnte an der Gewichtsfunktion liegen, da sie durch die Ähnlichkeit zu einem Mittelwert nicht statistisch robust genug gegen Ausreißer ist. Diese fehlende Robustheit wirkt sich bei der Registerinjektion nicht so extrem aus, da dort die Einzelgewichte ähnliche Symmetrien wie LSum im geringeren Wertebereich gezeigt haben. Damit ist die gute Performance der Gewichtsfunktion bei der Registerinjektion nicht auf die Speicherinjektion übertragbar.

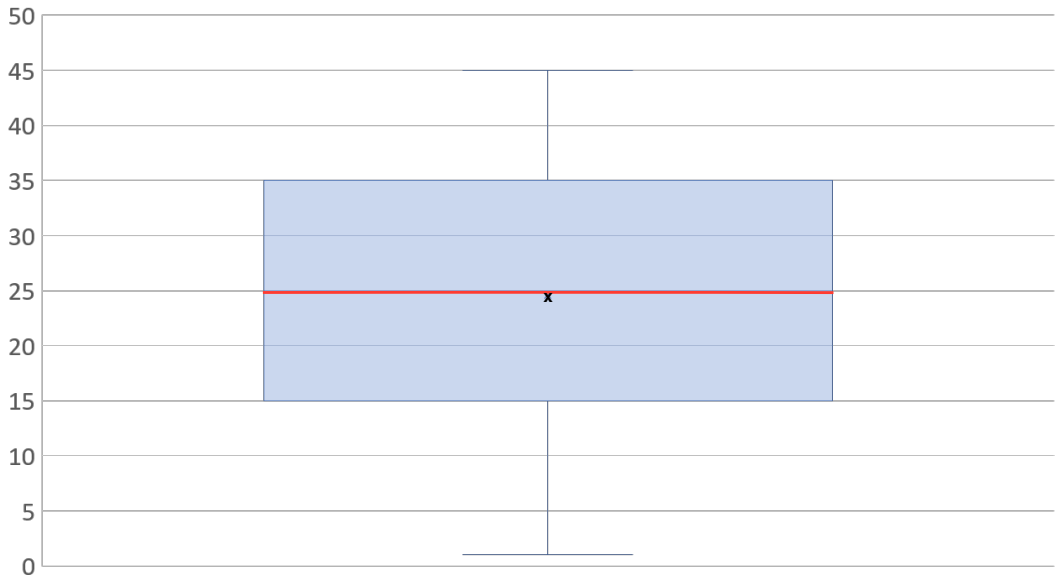
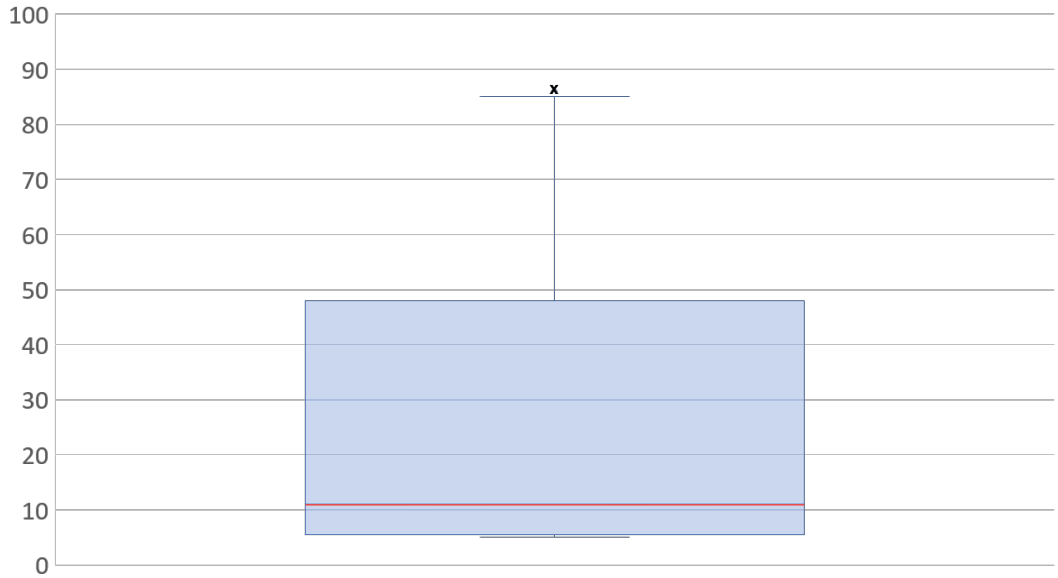
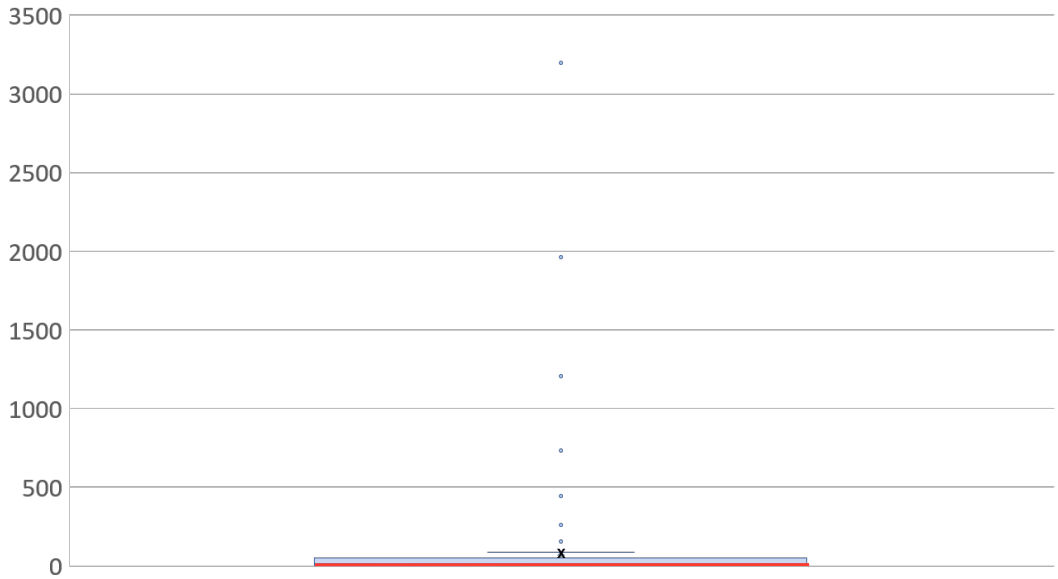


Abbildung 6.1 – *Gewichtserteilung herausragende Äquivalenzklassen beim Loop-Sum Benchmark (LSum):* Der Boxplot zeigt, dass 50% der herausragenden Äquivalenzklassen ein Gewicht zwischen 15 und 35 GPs besitzen. Ausreißer kommen nicht vor und die Verteilung ist symmetrisch.



(a) Darstellung der Box mit Whiskers (ohne Ausreißern)



(b) Darstellung mit Ausreißern

Abbildung 6.2 – *Gewichtsverteilung herausragende Äquivalenzklassen beim Fibonacci Benchmark (Fib):* Bei den herausragenden Äquivalenzklassen in Fib sind 50% der Gewichte in einem unteren Wertebereich. Hier zeigen sich jedoch extreme Ausreißer, die den Mittelwert stark beeinflussen.

6.3 Zusammenfassung der Diskussion

Als Zusammenfassung der Betrachtung der Fehlerinjektion in Register und Speicher kann klar festgestellt werden, dass durch die Einführung des Basisblock-Prunings in der Registerinjektion eine durchaus hohe Anzahl von Injektionen eingespart werden kann, ohne die Aussage der Funktionszuverlässigkeit signifikant zu verfälschen. Damit kann die Methode Basisblock-Pruning für Registerinjektionen durch den geringeren Zeitaufwand ein deutlichen Mehrwert bieten. In der Betrachtung der Speicherinjektion kann über den Mehrwert des Basisblock-Prunings in seiner jetzigen Implementierung keine klare Aussage getroffen werden. In der Evaluation hat sich gezeigt, dass die Einsparung von Injektionspunkten nicht in einer Deutlichkeit, wie bei der Registerinjektion möglich scheint. Das liegt, wie beschrieben, daran, dass Daten die im Speicher gespeichert werden, eine längere Zeit als im Register ungelesen bleiben, wodurch die dazugehörigen Äquivalenzklassen schneller Basisblockgrenzen überschreiten. In Bezug auf die Aussage der Funktionszuverlässigkeit ist doch eine signifikante Verfälschung zu sehen, wodurch das Basisblock-Pruning in seiner jetzigen Version in der Speicherinjektion nicht angewendet werden kann. Dies kann, wie im letzen Abschnitt beschrieben, an der gewählten Gewichtsfunktion liegen, welche die Vergleichbarkeit der Pruningmethoden derzeit durch die enorme Streuung in der Speicherinjektion nicht hinreichend zeigen kann. Eine Gewichtsfunktion, die eine bessere Robustheit gegenüber Ausreißern und hohen statistischen Varianzen besitzt, kann durchaus eine andere Einschätzung liefern.

7 FAZIT

In der Evaluation hat sich gezeigt, dass der Ansatz des Basisblockprunings in der Registerinjektion sehr gut funktioniert. Durch das erweiterte Pruning wurde die benötigte Anzahl von Injektionen, um eine qualitative Aussage über die Funktionszuverlässigkeit eines Systems zu treffen, deutlich reduziert. Bei der Speicherinjektion ist, wie in den Ergebnissen schon beschrieben, der Ansatz samt Gewichtsfunktion, durch die enorme Streuung der Gewichte einzelner Äquivalenzklassen, nicht zu verwenden.
Die Auswertung in der Registerinjektion hat gezeigt, dass die Annahme, das durch das Verwenden von Basisblöcken der Testaufwand verringert werden kann, sich als richtig dargestellt hat. Defekte, die sich nicht in herausragenden Äquivalenzklassen befinden, fallen laut der Pruning-Vorschrift heraus und werden somit nicht bei den IPs berücksichtigt. Das resultierende Fehlermodell kann somit deutlich mehr Injektionen einsparen als das Fehlermodell des Def-Use Pruning. Die Auswirkung auf die Speicherinjektion kann hier abschließend zeigen, dass der Ansatz bei hohen Streuungen der Einzelgewichte nicht funktioniert. In der Evaluation ließ sich jedoch eine kleine Tendenz dahingehend erkennen, dass auch bei der Speicherinjektion durch das Basisblock-Pruning eine Verringerung des Testaufwandes möglich ist, wenn auch Verbesserungsbedarf bzgl. der Gewichtsfunktion besteht.
Abschließend kann das Basisblock-Pruning als durchaus viel versprechende Pruningmethode in der Fehlerinjektion angesehen werden.

8 AUSBLICK

Wie schon im Fazit dargestellt, ist der Pruningansatz in der Registerebene mit positivem Resultat auf seine Eignung getestet worden. In der Speicherinjektion war das durch die verwendete Gewichtsfunktion nicht im selben Ausmaß möglich. Wie bereits in Abschnitt 6.2 beschrieben, besitzt die verwendete Gewichtsfunktion eine zu geringe Robustheit gegen Ausreißern und hohen statistischen Varianzen in den Einzelgewichten der herausragenden Äquivalenzklassen. Demnach müsste eine Gewichtsfunktion evaluiert werden, die diese Robustheit gegenüber Ausreißern besser beherrscht. Eine Variante wäre, die Einzelgewichte der herausragenden Äquivalenzklassen als Maß zur Verteilung des Basisblockgewichts zu nehmen. Dies würde bedeuten, dass dem IP der längsten Äquivalenzklasse im Verhältnis zu allen IPs des Basisblocks das meiste Gewicht zugeteilt würde. Demnach würde dieser Ansatz die Streuung der Äquivalenzklassenlängen berücksichtigen.
Ein weiterer Ansatz, die Gewichtsfunktion anzupassen wäre innerhalb eines Basisblocks Zusammenhänge zwischen einzelnen innenliegenden Äquivalenzklassen zu einzelnen herausragenden Äquivalenzklassen herauszufinden, um dann direkt über diese Zusammenhänge die Gewichte dem IP zuzuteilen. Ein solcher Zusammenhang könnte z.B. eine zusammenhängende Berechnung sein, wie in Abbildung 4.1 zu sehen. Dort wird im obersten Basisblock die Variable y gelesen, addiert und in x gespeichert. Die Äquivalenzklasse der Variablen y endet somit innerhalb des Basisblocks und würde dann der herausragenden Äquivalenzklasse von x zugerechnet. Das würde bedeuten, dass Äquivalenzklassen anhand des Datenpfades verbunden werden.
Abgesehen von den Anpassungsmöglichkeiten der Gewichtsfunktion, sind in der Betrachtung der gesamten Pruningmethode noch weitere Ansätze möglich.
Zu prüfen wäre, ob die Abstraktionsebene der Basisblöcke die beste Abstraktion darstellt; durchaus interessant wäre eine weitere Ebene in Richtung der Softwareapplikation zu schauen. Diese nachfolgende Ebene, in der dann der Funktionsrumpf als Abgrenzung dient und der Returnwert der IP sollte daraufhin untersucht werden, ob weitere Injektionen eingespart werden ohne die Aussage der Funktionszuverlässigkeit zu verfälschen.
Auch die Untersuchung, ob es beim Testen eines Systems wichtig ist, alle Basisblockinstanzen zu injizieren oder eine einmalige Injizierung eines Basisblocks ausreichend ist, könnte zur erheblichen Reduktion des Testaufwandes führen.

ANHANG

A.1 Ergebnisse Registerinjektion

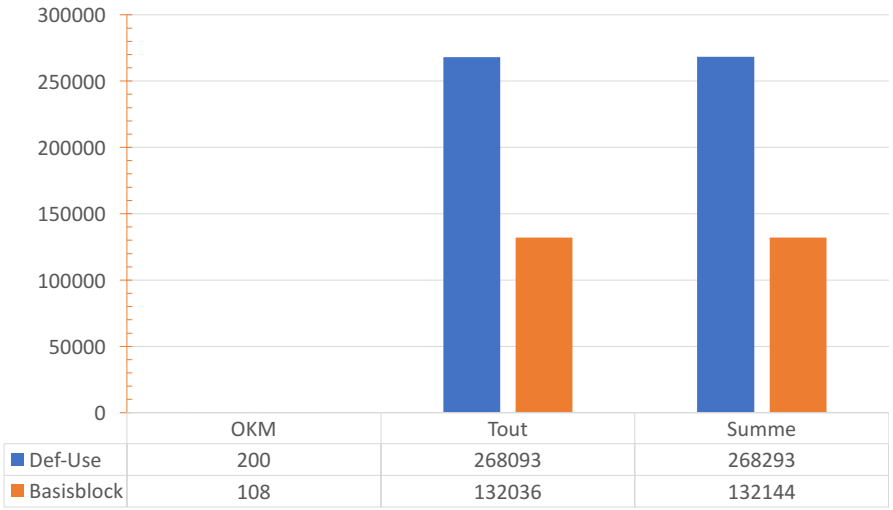
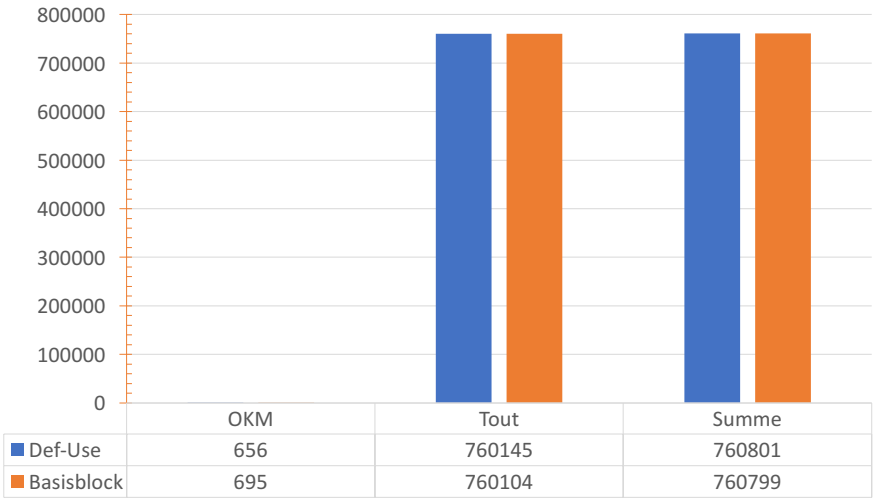
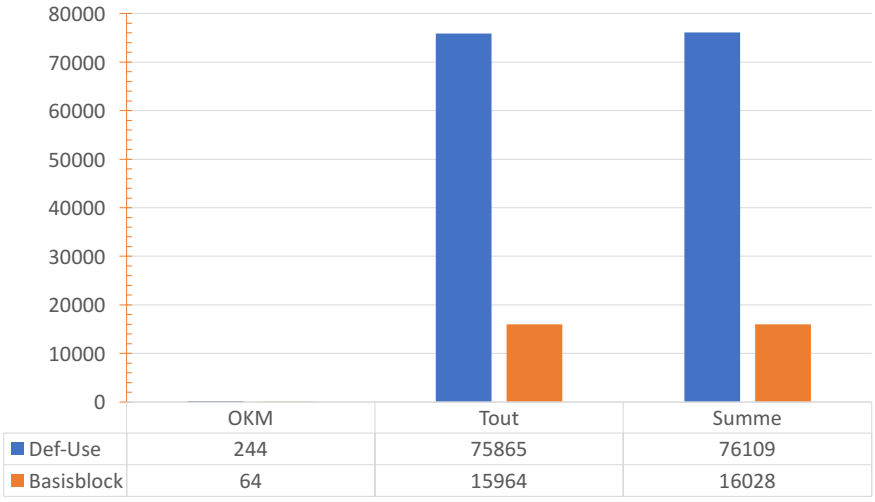


Abbildung A.1 – *Vergleich der Injektionen im Short Basis Block Benchmark (SBB)*



	OKM	Tout	Summe
Def-Use	656	760145	760801
Basisblock	695	760104	760799

Abbildung A.2 – *Vergleich der Gewichte im Short Basis Block Benchmark (SBB)*



	OKM	Tout	Summe
Def-Use	244	75865	76109
Basisblock	64	15964	16028

Abbildung A.3 – *Vergleich der Injektionen im Long Basis Block Benchmark (LBB)*

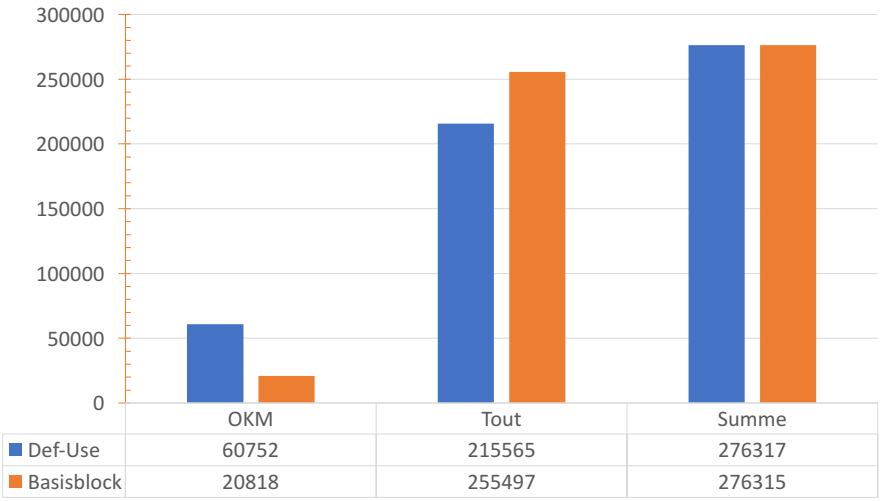


Abbildung A.4 – *Vergleich der Gewichte im Long Basis Block Benchmark (LBB)*

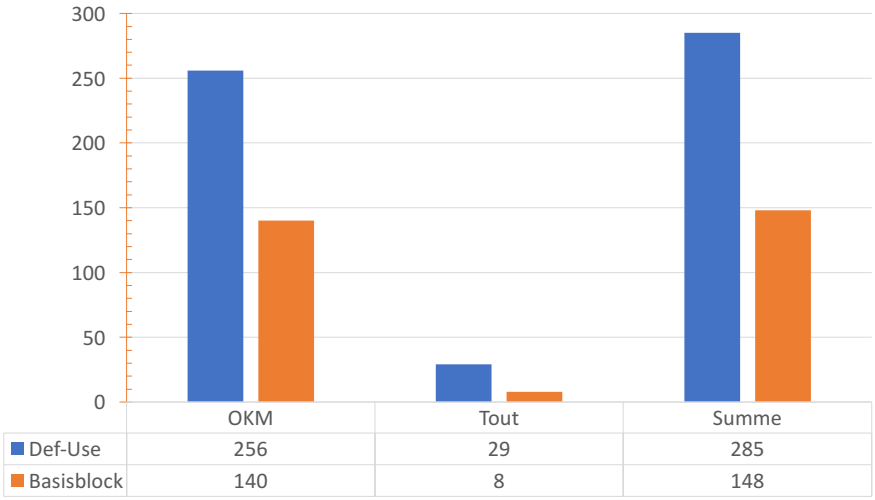


Abbildung A.5 – *Vergleich der Injektionen im Loop-Sum Benchmark (LSum)*

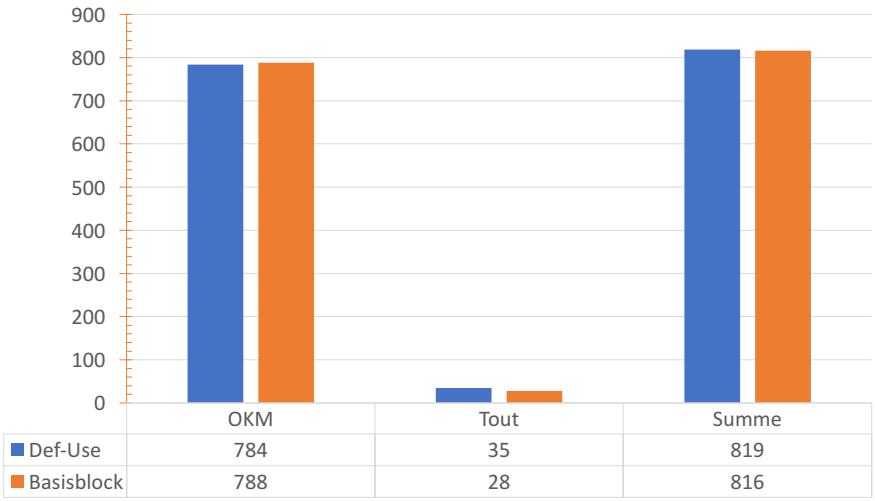


Abbildung A.6 – *Vergleich der Gewichte im Loop-Sum Benchmark (LSum)*

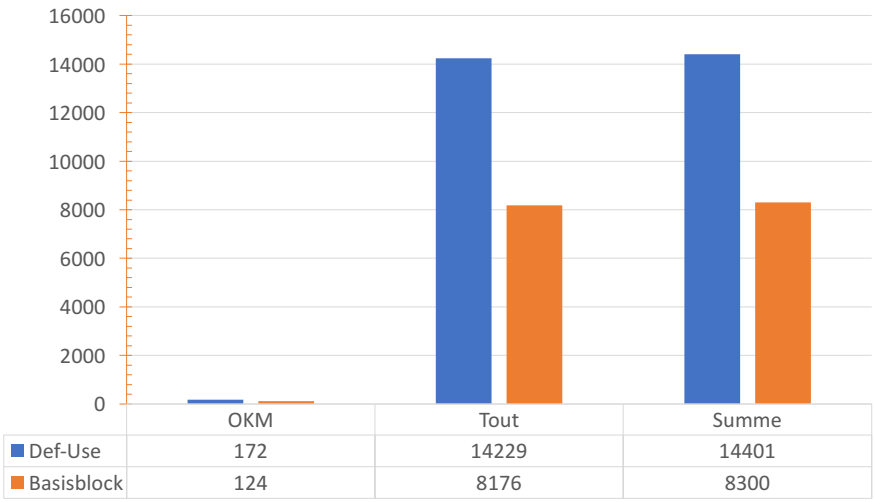
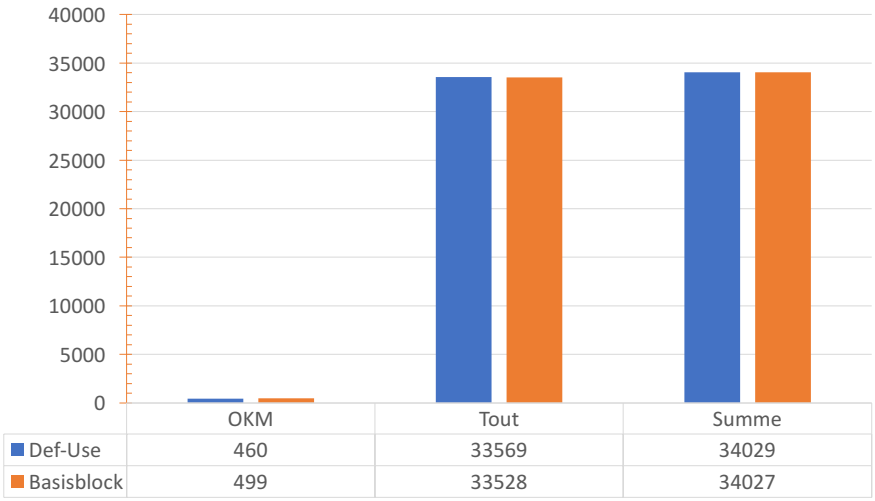
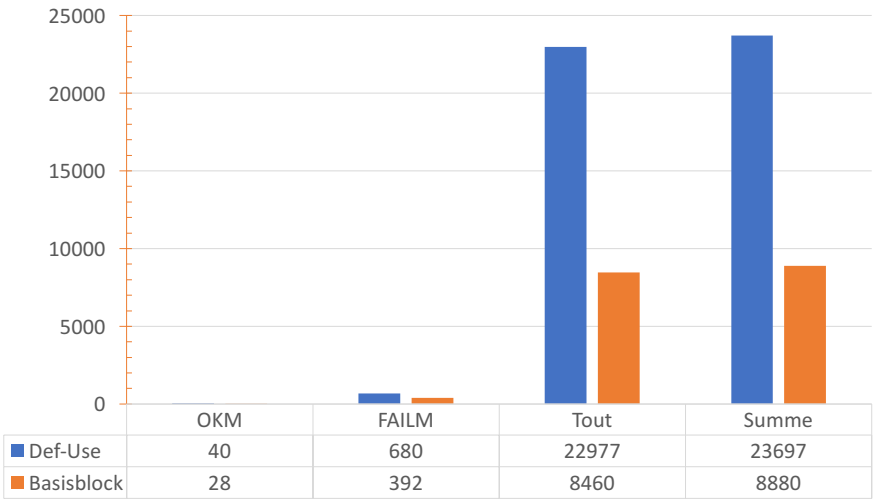


Abbildung A.7 – *Vergleich der Injektionen im Fibonacci Benchmark (Fib)*



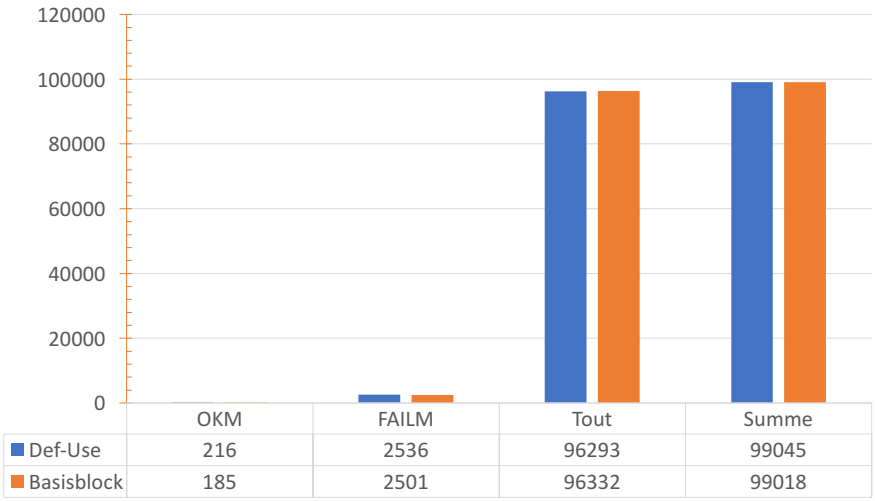
	OKM	Tout	Summe
Def-Use	460	33569	34029
Basisblock	499	33528	34027

Abbildung A.8 – *Vergleich der Gewichte im Fibonacci Benchmark (Fib)*



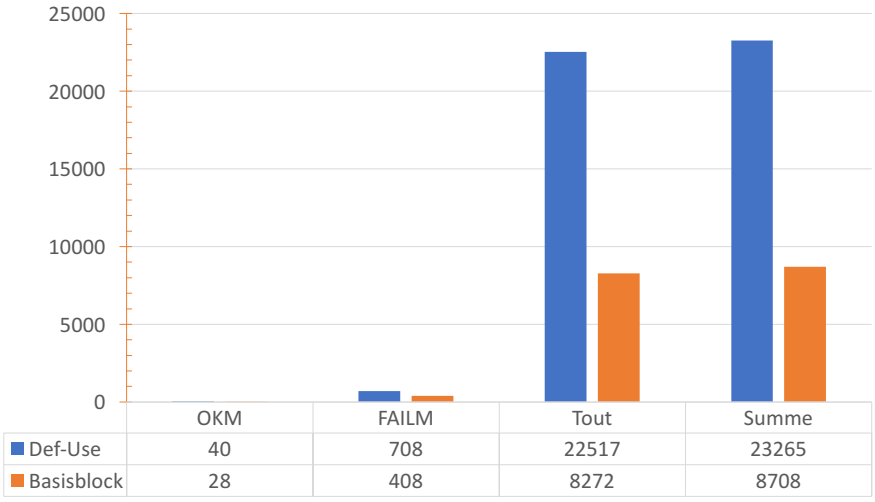
	OKM	FAILM	Tout	Summe
Def-Use	40	680	22977	23697
Basisblock	28	392	8460	8880

Abbildung A.9 – *Vergleich der Injektionen im QSort(rekursiv) Benchmark (QSR)*



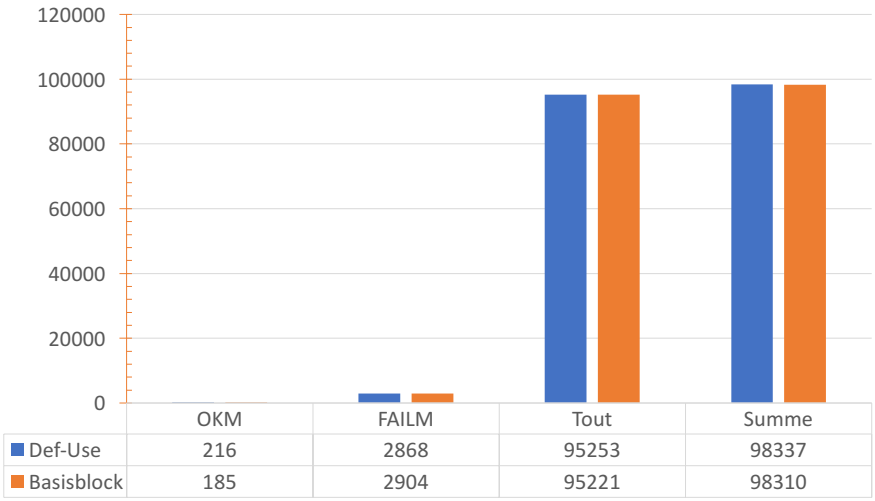
	OKM	FAILM	Tout	Summe
Def-Use	216	2536	96293	99045
Basisblock	185	2501	96332	99018

Abbildung A.10 – *Vergleich der Gewichte im QSort(rekursiv) Benchmark (QSR)*



	OKM	FAILM	Tout	Summe
Def-Use	40	708	22517	23265
Basisblock	28	408	8272	8708

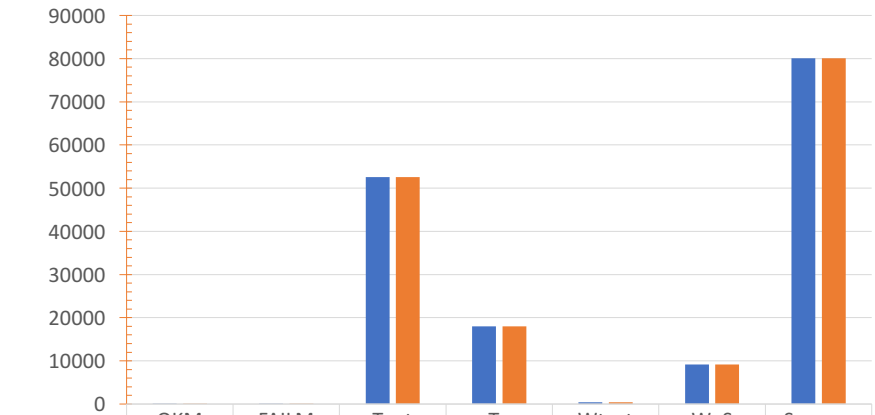
Abbildung A.11 – *Vergleich der Injektionen im QSort(iterativ) Benchmark (QSI)*



	OKM	FAILM	Tout	Summe
Def-Use	216	2868	95253	98337
Basisblock	185	2904	95221	98310

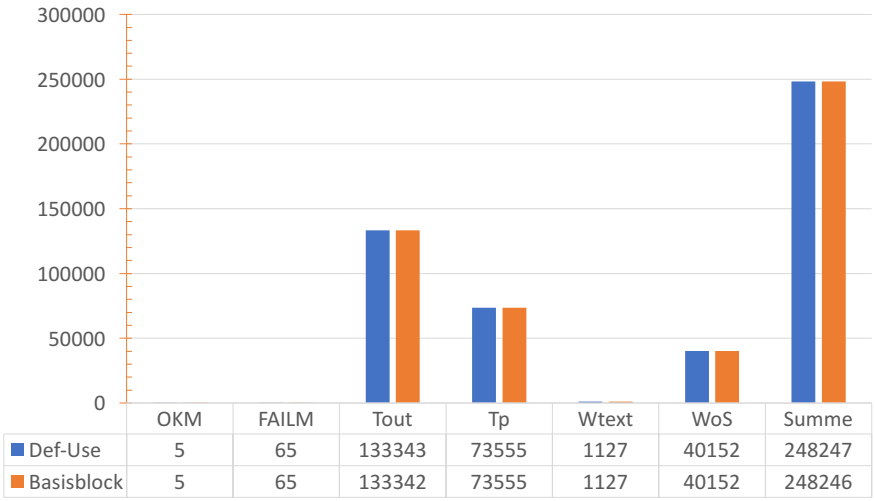
Abbildung A.12 – *Vergleich der Gewichte im QSort(iterativ) Benchmark (QSI)*

A.2 Ergebnisse Speicherinjektion



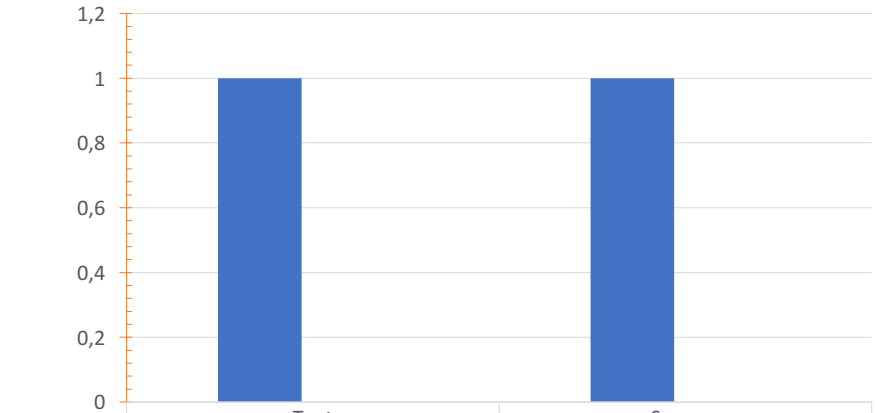
	OKM	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	1	29	52527	18014	375	9131	80077
Basisblock	1	29	52526	18014	375	9131	80076

Abbildung A.13 – *Vergleich der Injektionen im Short Basis Block Benchmark (SBB)*



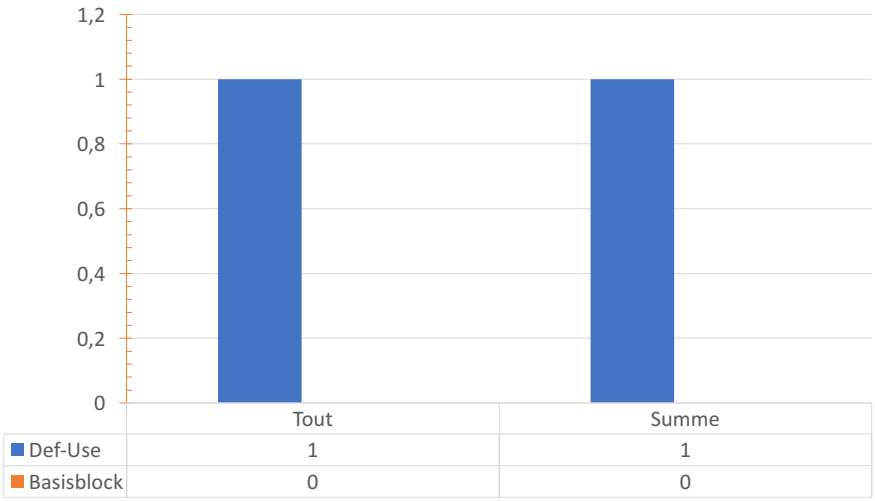
	OKM	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	5	65	133343	73555	1127	40152	248247
Basisblock	5	65	133342	73555	1127	40152	248246

Abbildung A.14 – *Vergleich der Gewichte im Short Basis Block Benchmark (SBB)*



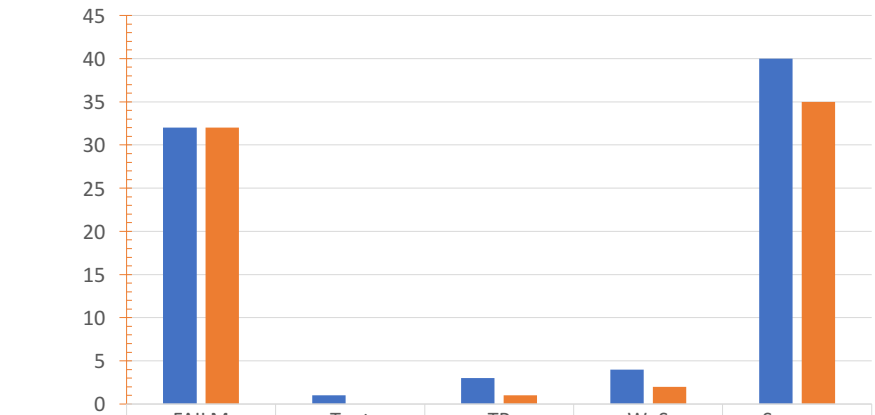
	Tout	Summe
Def-Use	1	1
Basisblock	0	0

Abbildung A.15 – *Vergleich der Injektionen im Long Basis Block Benchmark (LBB)*



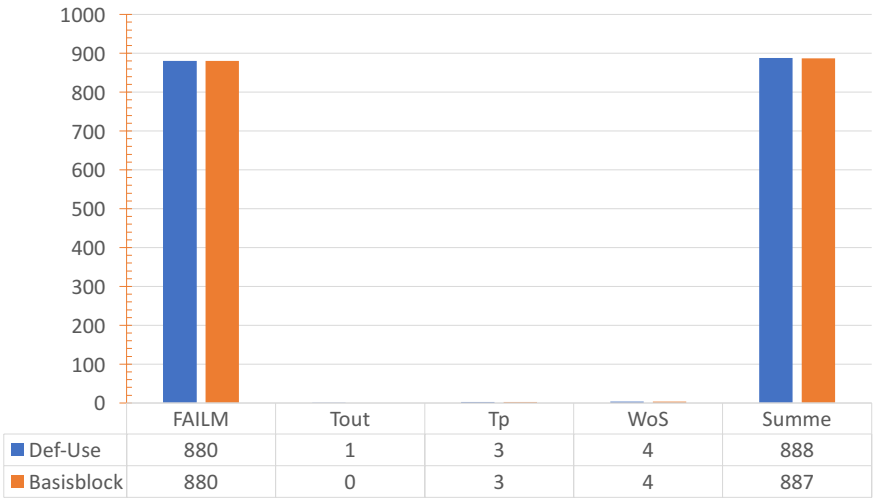
	Tout	Summe
Def-Use	1	1
Basisblock	0	0

Abbildung A.16 – *Vergleich der Gewichte im Long Basis Block Benchmark (LBB)*



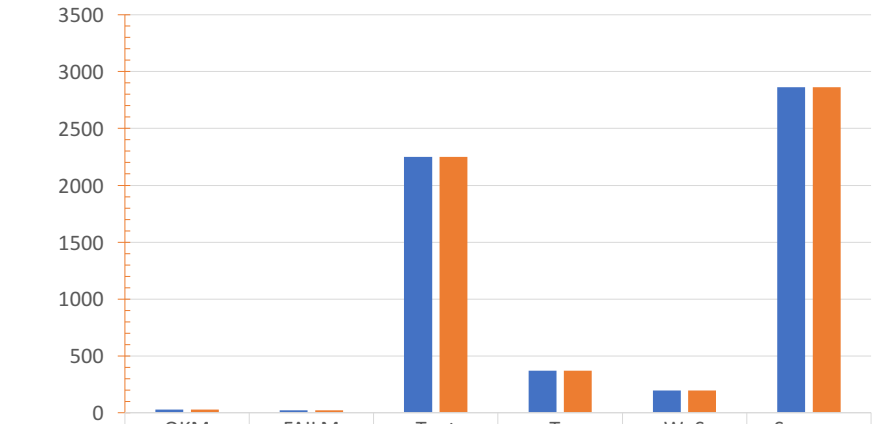
	FAILM	Tout	TP	WoS	Summe
Def-Use	32	1	3	4	40
Basisblock	32	0	1	2	35

Abbildung A.17 – *Vergleich der Injektionen im Loop-Sum Benchmark (LSum)*



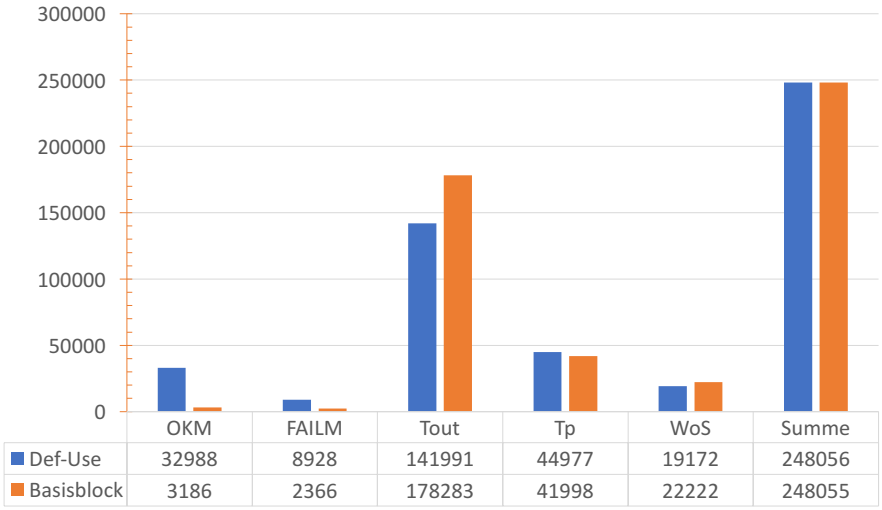
	FAILM	Tout	Tp	WoS	Summe
Def-Use	880	1	3	4	888
Basisblock	880	0	3	4	887

Abbildung A.18 – *Vergleich der Gewichte im Loop-Sum Benchmark (LSum)*



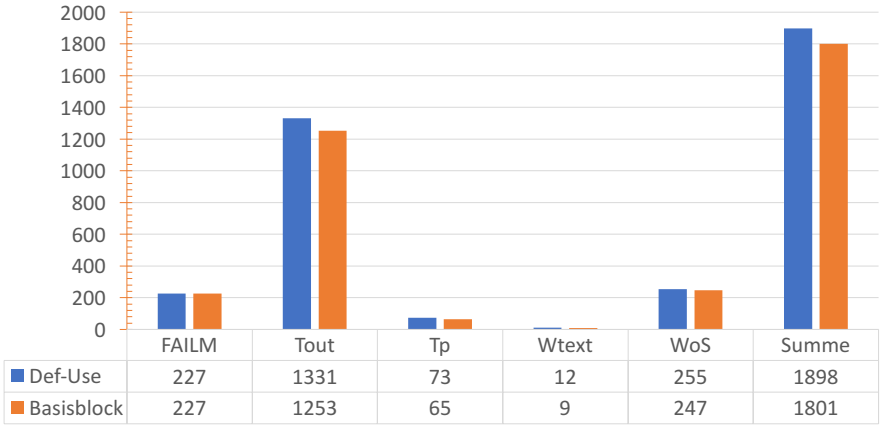
	OKM	FAILM	Tout	Tp	WoS	Summe
Def-Use	28	21	2251	369	195	2864
Basisblock	28	21	2250	369	195	2863

Abbildung A.19 – *Vergleich der Injektionen im Fibonacci Benchmark (Fib)*



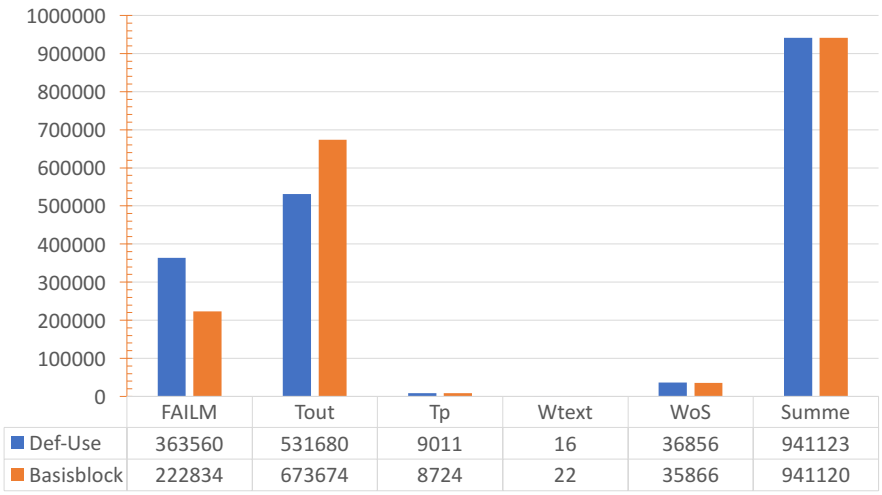
	OKM	FAILM	Tout	Tp	WoS	Summe
Def-Use	32988	8928	141991	44977	19172	248056
Basisblock	3186	2366	178283	41998	22222	248055

Abbildung A.20 – *Vergleich der Gewichte im Fibonacci Benchmark (Fib)*



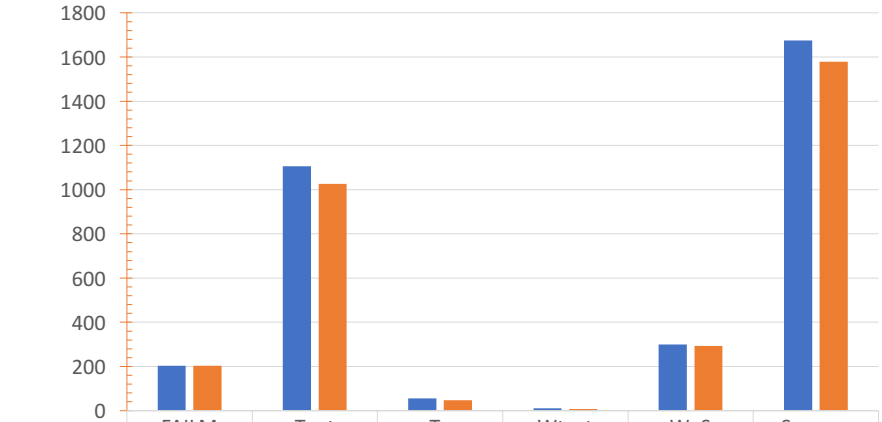
	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	227	1331	73	12	255	1898
Basisblock	227	1253	65	9	247	1801

Abbildung A.21 – *Vergleich der Injektionen im QSort(rekursiv) Benchmark (QSR)*



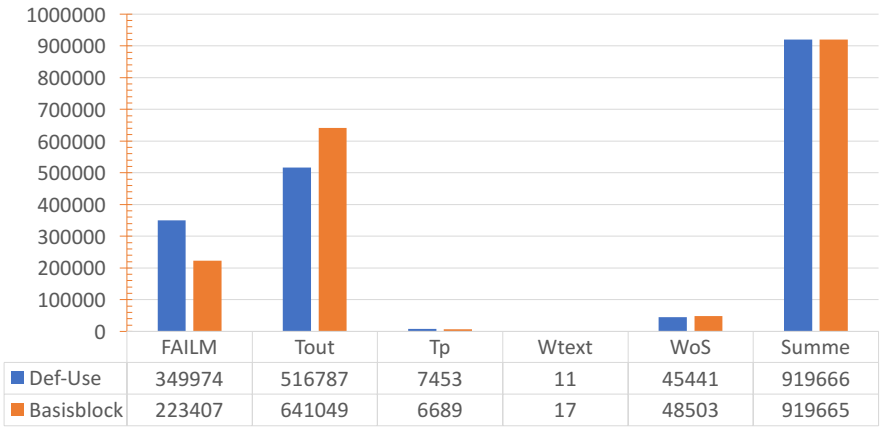
	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	363560	531680	9011	16	36856	941123
Basisblock	222834	673674	8724	22	35866	941120

Abbildung A.22 – *Vergleich der Gewichte im QSort(rekursiv) Benchmark (QSR)*



	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	204	1105	55	11	300	1675
Basisblock	203	1026	48	8	293	1578

Abbildung A.23 – *Vergleich der Injektionen im QSort(iterativ) Benchmark (QSI)*



	FAILM	Tout	Tp	Wtext	WoS	Summe
Def-Use	349974	516787	7453	11	45441	919666
Basisblock	223407	641049	6689	17	48503	919665

Abbildung A.24 – *Vergleich der Gewichte im QSort(iterativ) Benchmark (QSI)*

ABKÜRZUNGSVERZEICHNIS

IP	Injektionspunkt
GP	Gewichtspunkt
SBB	Short Basis Block Benchmark
LBB	Long Basis Block Benchmark
QSR	QSort(rekursiv) Benchmark
QSI	QSort(iterativ) Benchmark
LSum	Loop-Sum Benchmark
Fib	Fibonacci Benchmark
OKM	OK Marker
FAILM	Fail Marker
Tout	Timeout
Tp	Trap
Wtext	Write textsegment
WoS	Write outer space

ABBILDUNGSVERZEICHNIS

LITERATUR

[Aid+01] Joakim Aidemark u. a. “Goofi: Generic object-oriented fault injection tool”. In: *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE. 2001, S. 83–88.

[Bar+] Raul Barbosa u. a. “An approach to reducing the cost of fault injection”. In: *Proceedings of Real-Time in Sweden (RTiS’), pages–, August.(Cited on pages„ and.)* ().

[Bar+05] Raul Barbosa u. a. “Assembly-level pre-injection analysis for improving fault injection efficiency”. In: *European Dependable Computing Conference*. Springer. 2005, S. 246–262.

[Bar+90] J.H. Barton u. a. “Fault injection experiments using FIAT”. In: *IEEE Transactions on Computers* 39.4 (1990), S. 575–582. DOI: 10.1109/12.54853.

[Ben+98] A. Benso u. a. “Fault-list collapsing for fault-injection experiments”. In: *Annual Reliability and Maintainability Symposium. 1998 Proceedings. International Symposium on Product Quality and Integrity*. IEEE, 1998. DOI: 10.1109/rams.1998.653808.

[Ber+02] L. Berrojo u. a. “New techniques for speeding-up fault-injection campaigns”. In: *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*. IEEE Comput. Soc, 2002. DOI: 10.1109/date.2002.998398.

[Des+16] Sujay B. Desai u. a. “MoS2 transistors with 1-nanometer gate lengths”. In: *Science* 354.6308 (2016), S. 99–102. ISSN: 0036-8075. DOI: 10.1126/science.aah4698. eprint: http://science.sciencemag.org/content/354/6308/99.full.pdf. URL: http://science.sciencemag.org/content/354/6308/99.

[Ech13] Klaus Echtle. *Fehlertoleranzverfahren*. 7. März 2013. URL: https://www.ebook.de/de/product/33492328/klaus_echtle_fehlertoleranzverfahren.html.

[Fec09] Bernhard Fechner. *Transiente Fehler in Mikroprozessoren*. Springer, 13. Feb. 2009. ISBN: 978-3-8348-9278-2. URL: https://www.ebook.de/de/product/13912248/bernhard_fechner_bernhard_fechner_transiente_fehler_in_mikroprozessoren.html.

[GKT89] Ulf Gunneflo, Johan Karlsson und Jan Torin. “Evaluation of error detection schemes using fault injection by heavy-ion radiation”. In: *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE. 1989, S. 340–347.

[Gri+12] Johannes Grinschgl u. a. “Efficient fault emulation using automatic pre-injection memory access analysis”. In: *2012 IEEE International SOC Conference*. IEEE, 2012. DOI: 10.1109/socc.2012.6398361.

[GS95] J. Guthoff und V. Sieh. “Combining software-implemented and simulation-based fault injection into a single fault injection method”. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*. IEEE Comput. Soc. Press, 1995. DOI: 10.1109/ftcs.1995.466978.

[Gut+01] Matthew R Guthaus u. a. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE. 2001, S. 3–14.

[Kar+91] J. Karlsson u. a. “TWO FAULT INJECTION TECHNIQUES FOR TEST OF FAULT HANDLING MECHANISMS”. In: *1991, Proceedings. International Test Conference*. IEEE, 1991. DOI: 10.1109/test.1991.519504.

[Kar+94] Johan Karlsson u. a. “Comparison and integration of three diverse physical fault injection techniques”. In: *In PDCS 2: Open Conference*. Citeseer. 1994.

[KI94] Wei-Lun Kao und R.K. Iyer. “DEFINE: a distributed fault injection and monitoring environment”. In: *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*. IEEE Comput. Soc. Press, 1994. DOI: 10.1109/ftpds.1994.494497.

[KKA95] Ghani A Kanawati, Nasser A Kanawati und Jacob A Abraham. “FERRARI: A flexible software-based fault and error injection system”. In: *IEEE Transactions on computers* 2 (1995), S. 248–260.

[Kop06] H. Kopetz. “On the Fault Hypothesis for a Safety-Critical Real-Time System”. In: *Automotive Software – Connected Services in Mobile Networks*. Springer Berlin Heidelberg, 2006, S. 31–42. DOI: 10.1007/11823063_3.

[Kuf+12] Walter Kuffner u. a. “Hände weg vom Steuer!” In: *ATZagenda* 1.1 (2012), S. 62–65.

[LA04] Chris Lattner und Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society. 2004, S. 75.

[Law96] Kevin P. Lawton. “Bochs: A Portable PC Emulator for Unix/X”. In: *Linux J.* 1996.29es (Sep. 1996). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=326350.326357.

[Mad+94] Henrique Madeira u. a. “RIFLE: A general purpose pin-level fault injector”. In: *Dependable Computing — EDCC-1*. Springer Berlin Heidelberg, 1994, S. 197–216. DOI: 10.1007/3-540-58426-9_132.

[MW79] 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.

[San+14] Thiago Santini u. a. “Reducing embedded software radiation-induced failures through cache memories”. In: *2014 19th IEEE European Test Symposium (ETS)*. IEEE, 2014. DOI: 10.1109/ets.2014.6847793.

[Sch+12] Horst Schirmeier u. a. “FAIL*: Towards a versatile fault-injection experiment framework”. In: *ARCS Workshops (ARCS), 2012*. IEEE. 2012, S. 1–5.

[Seg+95] Z. Segall u. a. “FlAT – Fault Injection Based Automated Testing Environment”. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. IEEE, 1995. DOI: 10.1109/ftcsh.1995.532663.

[Smi+95] *A method to determine equivalent fault classes for permanent and transient faults*. IEEE, 1995. DOI: 10.1109/RAMS.1995.513278.

[SSU16] Henning Schmidt-Semisch und Monika Urban. “Universität Bremen, Fachbereich 11, Institut für Public Health und Pflegeforschung, Abteilung 6: Gesundheit & Gesellschaft Diskussionspapier, Bremen 2016”. In: (2016).

[TP13] Anna Thomas und Karthik Pattabiraman. “LLFI: An intermediate code level fault injector for soft computing applications”. In: *Workshop on Silicon Errors in Logic System Effects (SELSE)*. 2013.

[Wik18] Wikipedia. *Transistor — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 9. September 2018]. 2018. URL: https://de.wikipedia.org/w/index.php?title=Transistor&oldid=180650668.

[Zie+96] James F Ziegler u. a. “IBM experiments in soft fails in computer electronics (1978–1994)”. In: *IBM journal of research and development* 40.1 (1996), S. 3–18.

[VDI07] VDI-Gesellschaft Produkt- und Prozessgestaltung. “VDI Richtlinie 4003. Zuverlässigkeitsmanagement”. In: *VDI-Handbuch Zuverlässigkeit*. Verein Deutscher Ingenieure, 2007.