technische universität
dortmund

Bachelorarbeit

Vergleich von Hardwarefehlermodellen bei Fehlerinjektions-Experimenten

Marcel Sellung
18. März 2015

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

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

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

Dortmund, den 18. März 2015

Marcel Sellung

Zusammenfassung

Zur Steigerung der Performanz bei gleichzeitiger Reduzierung des Energiebedarfs moderne Hardware, wird die Strukturbreite immer weiter verringert und das Spannungslevel herabgesetzt, was zur Folge hat, dass, neben der Leistungssteigerung, immer häufiger, durch kosmische Strahlung verursachte, Soft-Errors auftreten. Um die Zuverlässige von Anwendungen trotz unzuverlässiger Hardware gewährleisten zu können, werden immer neue Fehlertoleranzverfahren entwickelt, die sehr unterschiedlich, teilweise auf Hardware- und teilweise auf Softwareebene, implementiert sind. Zur Abschätzung deren Effizienz werden Fehlerinjektionsexperimente durchgeführt, bei denen das Auftreten von Soft-Errors simuliert wird. Da es hierfür aber keinen einheitlichen Standard gibt, werden dabei unterschiedliche Fehlermodelle verwendet, die zu unterschiedlichen Ergebnissen führen können. In dieser Arbeit wird exemplarisch die Auswertung einiger Benchmarks anhand verschiedener Fehlermodelle miteinander verglichen. In der Abschließenden Evaluation wird dann herausgearbeitet, wie sehr die Ergebnisse variieren, also wie vergleichbar verschiedene Fehlermodelle sind, um so deren Bedeutung aufzuzeigen.

Inhaltsverzeichnis

1 Einleitung

Moderne Computer sowie eingebettete Systeme werden in immer mehr Bereichen eingesetzt und sind aus dem heutigen Alltag nicht mehr wegzudenken. Dabei reicht das Anwendungsgebiet von Haushalts- und Unterhaltungselektronik, die von der Waschmaschine bis zum Smartphone immer mehr Technik enthält, bis hin zu besonders kritischen Bereichen, wie Luft- und Raumfahrttechnik, Assistenzsysteme im Auto oder Steuerung in Atomkraftwerken. Verstärkt wird diese Verbreitung durch immer leistungsfähigere Hardware, welche durch immer kleinere Strukturbreiten die Anzahl der Transistoren pro Chip in den letzten Jahren, entsprechend der Prognose von Intel Gründer Gordon Moore aus dem Jahr 1965, exponentiell ansteigen lassen konnte.

Ungeachtet vieler Vorteile wächst mit dem Anstieg der Leistung aber auch die Anzahl transienter Fehler, sogenannter Soft-Errors. Um die Zuverlässigkeit wichtiger Systeme weiterhin gewährleisten zu können, sind Fehlertoleranzverfahren notwendig, die je nach Anwendung auf Hard- oder Softwareebene implementiert werden können und deren Effizienz und Kosten sehr unterschiedlich sind.

1.1 Kontext

Bereits in den ersten dynamischen Speichern (DRAM) wurden von Zeit zu Zeit Fehler entdeckt, die sich nicht auf den Programmcode zurückführen ließen und bei denen folglich die Hardware als Fehlerursache identifiziert wurde. Besonders hierbei war, dass die Speicherzellen nicht dauerhaft beschädigt wurden und nach Auftreten des Fehlers wieder einwandfrei funktionierten. Aus diesem Grund wurden diese Fehler als Soft-Errors bezeichnet.

Da Soft-Errors allerdings zufällig und nicht vorhersagbar auftreten, wurde die Ursache hierfür erst im Jahr 1979 durch May und Woods entdeckt [1], die Alphapartikel als Grund für die Bit-Flips identifizierten. Nicht völlig vermeidbare Spuren von Uran und Thorium im IC-Gehäuse ließen durch radioaktiven Zerfall zweifach positiv geladene Heliumionen entstehen, die aus historischen Gründen als Alphapartikel bezeichnet werden und genügend Energie besitzen, um einzelne Bits kippen zu lassen. Nach Bekanntwerden der Ursache, reagierten die Hersteller mit entsprechenden Tests und Standards. Durch

Verringerung der radioaktiven Verunreinigung in den verwendeten Materialien und Erhöhung der Ladungsmenge für einzelne Bits [2] konnten Alphapartikel als Ursache für Soft-Errors unter Kontrolle gebracht werden.

Da Soft-Errors aber weiterhin auftraten, vermuteten Landfort und Ziegler bereits 1979, dass auch kosmische Strahlung eine Ursache für Soft-Errors darstellt [3, 3]. Diese These wurde ob der Schwierigkeit, durch kosmische Strahlung verursachte Soft-Errors zu isolieren, zunächst mit viel Skepsis betrachtet und konnte erst 1996 durch IBM bestätigt werden. [4]. Zwar ist die Fehlerrate bei durch kosmische Strahlung verursachten Soft-Errors deutlich geringer, kann aber bei Verwendung bestimmter Materialien und großer Höhe um ein Vielfaches ansteigen. Außerdem lässt sich kosmische Strahlung, im Gegensatz zur radioaktiven Verunreinigung, in der Praxis nicht vermeiden, da es keine entsprechenden Schirmmaterialien gibt [5]. Zudem ermöglichen neue Herstellungsverfahren immer kleinere Strukturbreite und geringeren Energiebedarf, was bei gleichbleibender kosmischer Strahlung zu einem Anstieg der Soft-Error Rate führt, da immer weniger Energie benötigt wird, um die Ladung einzelner Bits umzukehren.

1.2 Motivation

Um auf die Tatsache zu reagieren, dass sich Soft-Errors in moderner und leistungsfähiger Hardware nicht vermeiden lassen, wurden entsprechende Gegenmaßnahmen in Form von Fehlertoleranzverfahren entwickelt. Gemeinsam haben fast alle Verfahren, dass eine Abwägung zwischen Zuverlässigkeit und Leistung getroffen wird, da Informationen meistens durch Redundanz abgesichert werden. Allgemein sind hierbei Fehlertoleranzverfahren auf Hardware- deutlich teurer, aber effizienter als Verfahren auf Softwareebene.

Um sich bei konkreten Anwendungen für ein Verfahren entscheiden zu können, ist es wichtig, möglichst genau zu wissen, wie gut das jeweilige Verfahren welche Teile des Programms absichert, um dann entscheiden zu können, welches Verfahren im Einzelfall am geeignetsten erscheint. Da Soft-Errors aber nicht vorhersehbar sind und nur sehr selten auftreten, sind echte Messungen sehr schwierig und nur in Spezialfällen geeignet. Damit Verfahren trotzdem miteinander verglichen werden können, werden die Soft-Errors in den meisten Fällen durch Fehlerinjektion simuliert. Hierfür gibt es drei verschiedene Arten von Fehlerinjektionstools, hardware-, software- und simulatorbasierte. Der Hauptunterschied dieser Tools besteht meistens darin, wie sehr sie auf bestimmte Anwendungen spezialisiert sind [6]. Um die Ergebnisse der Fehlerinjektionsexperimente anschließend aber vergleichen zu können, ist nicht das gewählt Injektionstool, sondern das verwendete Fehlermodell und dessen Umsetzung entscheidend. Hierbei geht es darum, an welcher Stelle im System Fehler injiziert werden. Da eine direkte Injektion auf Transistorebene zu aufwändig und die genauen Auswirkungen auf ISA-Ebene unbekannt sind, werden Fehler z.B. im RAM, in Registern oder in der ALU injiziert. Ebenso bedeu-

tend ist die Interpretation der gewonnen Daten, die sich daran orientiert, was simuliert werden soll. Zwei sehr unterschiedliche Ansätze sind hier z. B., die jeweiligen Ergebnisse mit oder ohne ihre Lebenszeit zu gewichten. Dabei geht es um die grundsätzliche Frage, wann Fehler auftreten können. Geschieht dies nur bei lesendem oder schreibendem Zugriff, so wird nicht gewichtet, geht man davon aus, dass Bits jederzeit verfälscht werden können, steigt die Wahrscheinlichkeit, dass ein Fehler auftritt, mit der Lebenszeit der jeweiligen Bytes und es muss gewichtet werden.

Die Ergebnisse für verschiedene Fehlertoleranzverfahren können unter Umständen hierbei sehr stark variieren, so dass es kaum möglich ist, einheitliche Aussagen zu treffen, wenn verschiedene Fehlermodelle verwendet werden. Diese Variation wird im Abschnitt 2.2 nochmals an einem Beispiel ausführlich beschrieben. Ein großes Problem ist, dass hier kein einheitlicher Standard existiert und in vielen Fällen auch gar nicht erwähnt wird, welches Fehlermodell den gewonnen Daten zugrunde liegt. Dies wirft die Frage auf, wie vergleichbar die Ergebnisse verschiedener Arbeiten sind.

1.3 Zielsetzung

Ziel der Arbeit ist es, anhand einiger Benchmarks verschiedene Fehlermodelle exemplarisch miteinander zu vergleichen, um so deren Bedeutung bei Fehlerinjektionsexperimenten hervorzuheben und die für das Ergebnis relevanten Unterschiede zu identifizieren. Fehlermodelle mit nur geringer Variation bei den Ergebnissen können für weitere Betrachtungen zusammengefasst und die Unterschiede ignoriert werden. Für die anderen Fehlermodelle, bei denen die Ergebnisse deutlich variieren, sollen Gründe und Folgen genauer betrachtet und jeweils die Aspekte untersucht werden, die Algorithmen bei bestimmten Fehlermodellen besonders tolerant oder intolerant machen.

Als weiteres Ziel der Arbeit, werden mit Sortieralgorithmen, eine häufig benutzte Gruppe von Algorithmen, auf ihre Fehlertoleranz untersucht, um neben klassischen Faktoren wie Laufzeit, Speicher und Energieverbrauch auch die Fehlertoleranz als Aspekt für die Auswahl eines bestimmten Algorithmus zu ergänzen.

2 Problemanalyse

Das unvermeidbare Auftreten von Soft-Errors führt dazu, dass die Zuverlässigkeit von Anwendungen nur noch bedingt gewährleistet werden kann. Dies kann in einigen besonders kritischen Bereichen unter Umständen nicht tolerierbar sein, sodass Fehlertoleranzverfahren notwendig sind. Diese Verfahren verursachen allerdings höhere Kosten und reduzieren zusätzlich die Leistung, sodass in vielen Fällen eine Abwägung stattfindet, wie aufwändig die Fehlertoleranzverfahren seien dürfen, um ein Ausgeglichenes Verhältnis zwischen Zuverlässigkeit und Kosten zu schaffen. Wie viel Geld und Leistungsverlust dabei tolerierbar sind, um einen bestimmten Grad an Zuverlässigkeit zu bekommen variiert dabei von Anwendung zu Anwendung sehr stark.

Um sich also konkret für einen Fehlertoleranzmechanismus zu entscheiden, müssen genaue Informationen bekannt sein, wie gut Fehler erkannt bzw. behoben werden können und dies ggf. sogar auf einzelne Fehlertypen bezogen. Unterschieden wird hier im Wesentlichen zwischen SDC (Silent Data Corruption), Timeout und Trap. In den jeweiligen Anwendungen sind die Auswirkungen dieser Fehlertypen teilweise sehr unterschiedlich, sodass ich manchen Fällen auch nur bestimmte Fehlerarten erkannt bzw. behoben werden müssen. Um darüber genaue Daten zu erhalten, wird die Wirksamkeit der Fehlertoleranzverfahren durch Fehlerinjektionsexperimente evaluiert.

Das Problem bei der Fehlerinjektion ist, dass diese auf Transistorebene zu aufwändig und folglich zu teuer ist, sodass Fehlermodelle auf ISA-Ebene entwickelt wurden und Fehler auch dort injiziert werden. Allerdings ist nicht genau bekannt, welches Fehlermodell die Realität am besten repräsentiert und wie dieses genau zu realisieren ist. Deswegen existiert kein einheitlicher Standard, welches Fehlermodell verwendet werden sollte und auch fehlt bei vielen Arbeiten das Bewusstsein für die Bedeutung eines Fehlermodells, sodass dieses nicht genau oder auch gar nicht beschrieben wird. Durch dieses uneinheitliche Vorgehen kann nicht mehr sichergestellt werden, dass Ergebnisse aus verschiedenen Arbeiten miteinander vergleichbar sind, sodass es sehr schwer ist, einen Überblick über die Wirksamkeit aller relevanten Fehlertoleranzverfahren zu bekommen.

Für ein besseres Verständnis dieser Arbeit, wird im Abschnitt 2.1 dieses Kapitels, das Vorgehen bei der Umsetzung eines Fehlermodells genauer beschrieben. Danach wird in Abschnitt 2.2 an einem Beispiel exemplarisch gezeigt, wie Ergebnisse unter Verwendung verschiedener Fehlermodelle variieren können. Der Abschnitt 2.3 gibt dann, anhand einiger Beispiele, eine Übersicht über tatsächlich verwendete Fehlermodelle. Abschnitt 2.4

geht zum Abschluss dann noch auf inhärente Fehlertoleranz bei Sortieralgorithmen ein, um einen zusätzlichen Aspekt zur Steigerung der Fehlertoleranz aufzuzeigen.

2.1 Fehlermodell

Das Aufstellen und Umsetzen eines Fehlermodells geschieht allgemein in mehreren Schritten und bietet so mehrere Möglichkeiten am Ende, unterschiedliche Implementierungen und dadurch unterschiedliche Ergebnisse zu erhalten. Für ein besseres Verständnis werden diese Schritte im Folgenden kurz erläutert.

1. **Realität**: Im erstens Schritt sollte man sich möglichst genau über das tatsächliche Auftreten von Soft-Errors informieren, da dies im Laufe der Zeit, mit der Technik, variiert. So sind heutzutage nicht mehr Spuren radioaktiver Teilchen, sondern kosmische Strahlung Ursache für Soft-Errors. Auch treten sie durch die immer kleineren Strukturbreiten und den immer geringeren Energiebedarf nicht mehr ausschließlich im RAM auf, da die Störenergie auch an anderen Stellen ausreicht, um die Ladung von Transistoren zu verfälschen.

2. **Fehlermodell**: Weiß man, welches Auftreten mit dem Fehlermodell repräsentieren werden soll, z. B. Soft-Errors im RAM, wird im nächsten Schritt ein Fehlermodell entwickelt, was die Fehler nicht mehr auf Transistor-Eben, sondern deren Auswirkungen auf ISA-Ebene beschreibt.

3. **Realisierung**: Unter Umständen kann es sein, dass sich ein Fehlermodell nur sehr Aufwändig, oder gar nicht, mit den verwendeten Tools realisieren lässt. Sieht das Fehlermodell Soft-Errors in einer Komponente vor, auf die mit dem Fehlerinjektionstool nicht direkt zugegriffen werden kann, z. B. im Cache, dann müssen die Auswirkungen, durch Injektion an geeigneter anderer Stelle, realisiert werden. Dies kann durch Injektion auf der gecacheten Komponente geschehen, indem dort nicht immer Fehler injiziert werden, sondern nur dann, wenn der Cache verwendet werden würde.

4. **Implementierung**: Der letzte Schritt ist dann die tatsächliche Implementierung, also die Entscheidung, wann, wo und wie, mit welchen Tools Fehlerinjiziert werden und wie die Ergebnisse auszuwerten sind.

2.2 Anwendungsbeispiel Fehlermodelle

Um die Bedeutung des verwendeten Fehlermodells genauer aufzuzeigen, werden in Tabelle 2.1 die Ergebnisse eines Fehlerinjektionsexperiments unter Verwendung zwei verschiedene Fehlermodelle gegenübergestellt. Die beiden Fehlermodelle (RAM Gleichverteilt und RAM Read/Write bzw. Datenbus) werden den Abschnitten 3.1.1 und 3.1.2 genauer beschrieben. Für dieses Beispiel ist nur wichtig, dass, nach dem Fehlermodell RAM Gleichverteilt, jederzeit ein Soft-Error auftreten kann, während das andere Fehlermodell dies nur bei einem Speicherzugriff (Read/Write) vorsieht.

Als Benchmark wurde *bin_sem3* ausgewählt, da der Unterschied hier sehr deutlich auftritt. Ein genauere Beschreibung folg in Abschnitt 3.2.1.3.

Tabelle 2.1: Fehlerinjektionsergebnisse

Fehlermodell	Variant	Benchmark	Result	Occurrences
RAM Gleichverteilt	mlqueue_baseline	bin_sem3	OK	2.779.122.696.545
RAM Gleichverteilt	mlqueue_baseline	bin_sem3	FAIL	377.544.865.695
RAM Gleichverteilt	mlqueue_CRC	bin_sem3	OK	6.128.528.502.091
RAM Gleichverteilt	mlqueue_CRC	bin_sem3	FAIL	98.437.448.597
RAM Read / Write	mlqueue_baseline	bin_sem3	OK	121.790
RAM Read / Write	mlqueue_baseline	bin_sem3	FAIL	35.018
RAM Read / Write	mlqueue_CRC	bin_sem3	OK	3.191.751
RAM Read / Write	mlqueue_CRC	bin_sem3	FAIL	234.945

Zur Übersicht wurden in der Tabelle alle möglichen Ergebnistypen zusammengefasst, OK falls der Soft-Error keine Auswirkung hat oder durch die CRC Absicherung entdeckt, FAIL, falls ein unentdeckter Fehler vorliegt. Beim ersten Fehlermodell ist deutlich zu erkennen, dass in der CRC Variante eine Verbesserung eintritt, da die Anzahl der unentdeckten Fehler um den Faktor drei bis vier sinkt, obwohl die Gesamtzahl durch den Overhead der Absicherung steigt.

Beim zweiten Fehlermodell hingegen steigt die Anzahl unentdeckter Fehler um mehr als das sechsfache an, sodass hier das Ergebnis wäre, dass die Absicherung durch CRC kontraproduktiv ist und Fehleranfälligkeit der Anwendung nur steigt.

Der Grund hierfür ist, dass beim ausgewählten Benchmark sehr lange Zeit zwischen zwei Speicherzugriffen liegt. Können wie im ersten Fehlermodell in dieser Zeit Soft-Errors auftreten, werden, falls die Änderung bemerkt wird sehr viele Fehler entdeckt. Im zweiten Fehlermodell hingegen kann in der Zeit kein Soft-Error auftreten und die Absicherung führt zu mehr Speicherzugriffen und damit auch zu mehr potentiellen Soft-Errors, die nicht alle entdeckt werden.

2.3 Verwendete Fehlermodelle

Durch das immer häufigere Auftreten von Soft-Errors steigt zunehmend auch die Zahl der wissenschaftlichen Arbeiten, die sich mit diesem Thema auseinandersetzen. Wie zu Beginn dieses Kapitel erwähnt, fehlt bei all diesen Arbeiten aber ein einheitlicher Standard für die Fehlerinjektion. Zudem wird in vielen Fällen das verwendete Fehlermodell nicht oder nur unzureichend angegeben, obwohl, wie in Abschnitt 2.2 gezeigt, die Ergebnisse der Arbeiten, bei Verwendung eines anderen Fehlermodells, stark variieren würden und möglicherweise zu völlig anderen Schlüssen führen. In Tabelle 2.2 wird exemplarisch an einigen ausgewählten Arbeiten gezeigt, welche Angaben zu den Fehlermodellen gemacht werden, falls dieses erwähnt wird.

Tabelle 2.2: Verwendete Fehlermodelle

Nr.	Titel der Arbeit	Informationen zum Fehlermodell
1	A Study of the Impact of Bit-flip Errors on Programs Compiled with Different Optimization Levels [7]	Sigle-Bit Fehler im RAM und in Registern, injiziert vor Reads (inject-on-read)
2	Soft-error Detection through Software Fault-Tolerance techniques [8]	Single-Bit Fehler im Speicher
3	Eliminating single points of failure in software-based redundancy [9]	Single-Bit Fehler in der CPU zwischen zwei Assemblerinstruktionen
4	On the Impact of Hardware Faults - An investigation of the Relationship between Workload Inputs and Failure Mode Distributions [10]	Sigle-Bit Fehler im RAM und in Registern, injiziert vor Reads (inject-on-read)
5	Soft-Error Detection Using Control Flow Assertions [11]	Single-Bit Fehler - Keine Angabe, wo Fehler auftreten nur wann bezüglich des Programmablaufs
6	SWIFT - Software Implemented Fault Tolerance [12]	Single-Bit Fehler - Fehler treten im Speicher und der CPU auf - Injektion nicht beschrieben

2.4 Sortieralgorithmen

Um die Zuverlässigkeit einer Anwendung zu erhöhen, gibt es nicht nur die Möglichkeit, verschiedene Fehlertoleranzverfahren zu implementieren, auch die Wahl eines Algorithmus, kann die Zuverlässigkeit beeinflussen. Ein Algorithmus kann bei der Lösung eines Problems besonders Fehleranfällig oder besonders Fehlertolerant seien, ohne, dass dies Einfluss auf das Ergebnis hat.

Ein Beispiel für solch eine Gruppe von Algorithmen, die das gleiche Problem auf unterschiedliche Arten lösen, sind die Sortieralgorithmen. Bei der Auswahl eines geeigneten Algorithmus für eine konkrete Anwendung, können Messungen zur inhärenten Fehlertoleranz der jeweiligen Sortieralgorithmen, ein weiteres, wichtiges Auswahlkriterium sein und die bekannten Kennziffern, wie Laufzeit und Speicherplatzbedarf, ergänzen.

3 Entwurf

Um einen sinnvollen Vergleich von Fehlermodellen durchführen zu können, müssen zunächst einige wichtige und grundlegende Entwurfsentscheidungen getroffen werden, um anschließend die Ergebnisse sinnvoll verwenden zu können. Dabei geht es im Wesentlichen um zwei Aspekte.

1. Fehlermodelle: Um überhaupt einen Vergleich durchführen zu können, müssen zunächst verschiedene Fehlermodelle ausgewählt werden, die im weiteren Verlauf untersucht werden sollen. Diese Entscheidung hat eine besondere Tragweite, da nur durch plausible und tatsächlich verwendete Fehlermodelle, eine Relevanz der Arbeit gegeben ist.

2. Benchmarks: Die verwendeten Benchmarks dienen als Grundlage für die Interpretation und Auswertung der Fehlermodelle, es sollten also genügend Benchmarks ausgewählt werden, um eine statistische Relevanz zu erzielen. Außerdem ist es möglich, dass einige Besonderheiten nur bei bestimmten Arten von Algorithmen auftreten, sodass auch darauf geachtet werden sollte, eine gewisse Vielfalt bei der Auswahl zu gewährleisten.

Im Folgenden werden diese Vorarbeiten für den Vergleich der Fehlermodelle genauer erklärt, in Kapitel 3.1 werden zunächst die verwendeten Fehlermodelle detailliert beschrieben und ihre Auswahl erläutert. Kapitel 3.2 befasst sich dann genauer mit den Benchmarks, den verwendeten Algorithmen und Datensätzen.

3.1 Fehlermodelle

Die Auswahl der Fehlermodelle ist der wichtigste Schritt in der gesamten Vorbereitung, da sämtliche Ergebnisse hiervon beeinflusst werden. Im Wesentlichen unterscheiden sich die Fehlermodelle dabei in drei Kriterien, nämlich wo, wann und wie Fehler auftreten.

- Wo: Ein wesentliches Merkmal aller Modelle ist die Entscheidung, in welcher Komponente die Fehler auftreten werden. Ursprünglich wurden Soft-Errors im RAM entdeckt und traten mutmaßlich auch nur dort auf. Die immer kleinere Bauweise

und das folglich immer geringere Energieniveau ermöglicht Soft-Errors allerdings auch an anderer Stelle, sodass Fehler in Registern, in der ALU, auf dem Datenbus oder in Caches denkbar sind und dort teilweise bereits nachgewiesen wurden.

- Wann: Bei den meisten dieser unterschiedlichen Komponenten, gibt es zudem verschiedene Möglichkeiten, wann Fehler auftreten können. Dies kann gleichverteilt zu jedem beliebigen Zeitpunkt oder aber nur bei bestimmten Ereignissen wie Speichzugriffen (Read/Write) sein.

- Wie: Auch die Art des auftretenden Fehlers ist nicht immer gleich und unterscheidet sich meistens darin, ob ein, mehrere oder alle Bits des entsprechenden Bytes geflippt werden. Auch sind, neben den transienten Fehler, intermittierende oder permanente Fehler möglich.

Da die Anzahl der durchzuführenden Fehlerinjektionsexperimente mit jeder möglichen Ausprägung eines dieser Merkmale stark ansteigt, werden ein paar Einschränkungen, für alle im Folgenden ausgewählten Fehlermodelle angenommen, damit das Ganze in einem durchführbaren Rahmen bleibt.

Obwohl Speicherzellen durch kosmische Strahlung durchaus auch dauerhaft beschädigt werden können, werden in dieser Arbeit nur die sogenannten Soft-Errors, also transiente Fehler untersucht und intermittierende und permanente bei den Fehlermodellen im Folgenden nicht berücksichtigt.

Auch werden in allen Fehlermodellen nur Single-Bit Fehler angenommen und keine Burst Fehler. Dies hat gleich zwei Gründe, zum einen gibt es bereits Arbeiten, die Single-Bit Fehler mit Double-Bit Fehlern vergleichen [13] und deren Ergebnis vermuten lässt, dass die Unterschiede relativ gering sind. Zum anderen werden in den meistens Arbeit nur bei den Fehlermodellen nur Single-Bit Fehler angenommen, sodass die ausgewählten Modelle eine größere Relevanz für die aktuelle Forschung haben.

Auch bei der späteren Realisierung der ausgewählten Fehlermodelle kann, durch einige Vorüberlegungen und geschickte Implementierung, die Anzahl der anstehenden Experimente erheblich reduziert werden. Der Fehlerraum, in dem ein Soft-Error auftreten kann, umfasst den kompletten Speicher, also alle möglichen Speicheradressen und die komplette Laufzeit des Programms, die als Zeiteinheit die Anzahl der bisher durchgeführten Instruktionen verwende. Abbildung 3.1 zeigt einen Ausschnitt solch eines Fehlerraums und die Stellen, an denen lesend (R) oder schreibend (W) auf den Speicher zugegriffen wird. Relevant für die spätere Auswertung sind allerdings nur die Soft-Errors, die in wirklich benutzten Speicherbereichen liegen. Zwar kann ein Fehler theoretisch auch an jeder anderen Stelle auftreten, z. B. an der Position X (grün), für den Ablauf des Programms ist dies aber nicht von Bedeutung.

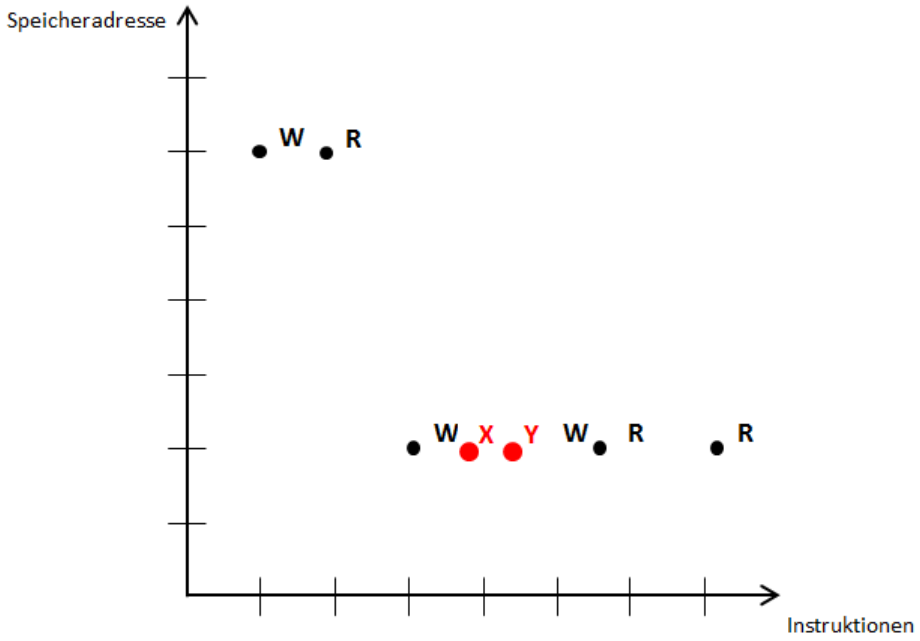

Abbildung 3.1: Fehlerraum

Ebenfalls ist der genau Zeitpunkt, an dem ein Fehler auftritt, für die weitere Ausführung des Programms unerheblich. Ein Fehler an der Position Y (rot) hat die gleichen Auswirkungen, wie ein Fehler an der Position Z (rot) oder jedem anderen Zeitpunkt zwischen den beiden Speicherzugriffen an dieser Adresse. So können, bei der späteren Realisierung, die Auswirkungen, für mehrere potentielle Soft-Errors, durch eine einzige Fehlerinjektion überprüft werden.

3.1.1 Fehlermodell RAM Gleichverteilt

Als erstes Fehlermodell werden transiente Single-Bit Fehler gleichverteilt im RAM angenommen. Dies beruht auf der Idee, dass kosmische Strahlung völlig unabhängig von den jeweiligen Abläufen der einzelnen Komponenten ist und diese folglich auch keinen Einfluss auf die Strahlungsintensität, also das Auftreten von Soft-Errors haben können. Demzufolge kann ein Fehler zu jeder Zeit und an jeder Stelle in der betroffenen Komponente gleich wahrscheinlich vorkommen.

Da es, wie gerade beschrieben, nicht wichtig ist, wann genau ein Fehler zwischen zwei Speicherzugriffen auftritt, sondern nur, dass die Daten verfälscht sind, werden, bei der Realisierung nur Fehler vor den Speicherzugriffen (Read/Write) betrachtet und diese dann mit der Lebenszeit der Daten, also der Dauer seit dem letzten Speicherzugriff, gewichtet.

3.1.2 Fehlermodell RAM Read/Write & Fehlermodell Datenbus

Da Soft-Errors nach [14, 3] allerdings besonders häufig während Lese- oder Schreibaktivitäten auftreten, werden, als zweites Fehlermodell, ungleichverteilte, transiente Single-Bit Fehler im RAM, verwendet. Ungleichverteilt heißt in diesem Fall dann, dass Fehler nur bei Speicherzugriffen (Read/Write) auftreten. Zwar ist dieses Modell nicht ganz perfekt zur Umsetzungen der, im Paper beschriebenen, Häufung von Soft-Errors während Speicherzugriffen geeignet, weil ein häufiges Auftreten dort nicht bedeutet, dass in den Zwischenzeiten gar keine Soft-Errors auftreten können. Allerdings wäre solch ein Fehlermodell nicht zu realisieren, da genaue Zahlen für diese Häufung nicht bekannt sind. Außerdem spricht für dieses Fehlermodell, dass es so tatsächlich in anderen Arbeiten verwendet wird.

Dieses Fehlermodell entspricht in etwa einem weiteren Fehlermodell, nämlich transienter Single-Bit Fehler auf dem Datenbus. Diese beiden Fehlermodelle werden im Folgenden nicht mehr getrennt betrachtet, da die einfachste Realisierung des Fehlermodells Datenbus genau der Realisierung des Fehlermodells RAM Read/Write entspricht. Fehler auf dem Datenbus würden nur bei Benutzung des Datenbus, also bei Speicherzugriffen, Auswirkungen haben und können folglich durch RAM Fehler beim Lesen oder Schreibaktivitäten simuliert werden. Dass solch eine Simulation als Realisierung verwendet wird, liegt daran, dass dann die Ergebnisse des Fehlermodells RAM Gleichverteilt verwendet werden können, wenn ihre Gewichtung ignoriert wird.

3.1.3 Fehlermodell Datenbus-Undo

Die Simulation des Fehlermodells transiente Single-Bit Fehler auf dem Datenbus durch Fehler im RAM hat, gegenüber der tatsächlichen Realisierung durch Fehler auf dem Datenbus, den Vorteil, dass hierbei wesentlich geringere Anpassungen der Experimente notwendig sind. Wenn beide Ergebnisse sich dabei tatsächlich ähneln, würde dies die Implementierung des Fehlermodells Datenbus erheblich vereinfachen.

Um diesem Fehlermodell allerdings näher zu kommen, sind noch einige weitere Anpassungen geeignet. Tritt ein Fehler auf dem Datenbus und nicht im RAM auf, so würde bei mehrfachem Lesen, der Fehler nicht erneut auftreten, da die Daten im RAM nicht wirklich korrumpiert wurden. Deswegen müsste, bei der Realisierung, ein Fehler nachdem er gelesen wurde, wieder korrigiert werden.

3.1.4 Fehlermodell Datenbus-AfterWrite

Auch eine andere Anpassung der bei der Realisierung wäre interessant, um die Genauigkeit der Simulation zu verbessern. Ein Fehler der auf dem Datenbus auftritt, würde sich, bei einem Schreibzugriff im RAM, erst nach dem Schreiben auswirken und nicht davor. Die AfterWrite Variante des Fehlermodells transient Single-Bit Fehler auf dem Datenbus, würde, bei der Realisierung, also durch Fehler im RAM simuliert werden, die vor dem Lesen und nach dem Schreiben auftreten.

3.1.5 Fehlermodell Datenbus-AfterWrite Undo

Beide Varianten sollen die Realisierung näher an eine perfekte Implementierung des tatsächlie Fehlermodells bringen. Um zu vergleichen, wie sinnvoll diese Änderungen sind werden bei der Variante AfterWrite Undo des Fehlermodells transiente Single-Bit Fehler auf dem Datenbus beide Änderungen gleichzeitig betrachtet, es werden also Fehler im RAM implementiert, die vor Lese- und nach schreibzugriffen auftreten.

3.1.6 Fehlermodell CPU

Weiteres Fehlermodell sind transiente Single-Bit Fehler in der CPU. Dies trennt dabei nicht, wo in der CPU die Fehler auftreten und fasst also bspw. Fehler in der ALU und in den Registern als CPU Fehler zusammen. Auch hier wäre ebenso wie bei Fehlermodell Datenbus eine Simulation des Fehlermodells durch Fehler im RAM hilfreich und folglich für einen Vergleich interessant. In diesem Fall würden bei der Realisierung Fehler im RAM nach Schreibzugriffen auftreten, da der Wert, der geschrieben wird, durch Fehler in der CPU nicht mehr korrekt ist.

3.2 Benchmarks

Die Auswahl der Benchmarks ist der zweite wichtige Aspekt in der Vorbereitung des Vergleichs, da dadurch, ebenso wie durch die Fehlermodelle, die Ergebnisse maßgeblich beeinflussen werden können. Um also eine verlässliche Grundlage zu haben, wird statistische Relevanz, durch eine ausreichend große Anzahl an Benchmarks, benötigt. Außerdem kann es sein, dass auch die Art der verwendeten Algorithmen Einfluss auf die Ergebnisse hat. Damit keine wichtigen Erkenntnisse übersehen werden, ist also auch eine Vielfalt bei den Benchmarks ratsam.

3.2.1 Algorithmen

Die verwendeten Algorithmen lassen sich grob in drei Bereiche einteilen, Sortieralgorithmen, Benchmark-Algorithmen aus der MiBench Benchmark Suite und eCos-Kernel-Test-Algorithmen aus anderen Arbeiten der Arbeitsgruppe Eingebettete Systemsoftware der TU Dortmund, Lehrstuhl Informatik 12.

3.2.1.1 Sortieralgorithmen

Sortieralgorithmen sind in der Lehre und Forschung ein häufiges Thema und werden auch im Alltag immer wieder eingesetzt. Besonders durch die immer wachsenden Speicherkapazitäten und Datenmengen steigt ihre Bedeutung auch weiterhin. Um sich bei Anwendungen für einen bestimmten Algorithmus zu entscheiden, werden meist Größen, wie die Laufzeit, Speicherplatz oder Energieverbrauch, miteinander verglichen und abgewogen. Da das Thema Fehlertoleranz, durch immer häufigeres Auftreten von Soft-Errors bei gleichzeitig steigender Leistung, an Bedeutung gewinnt, ist, einen Betrachtung von Sortieralgorithmen unter diesem Aspekt, eine hilfreiche Ergänzungen, für die Auswahl eines konkreten Algorithmus und wird in dieser Arbeit genauer untersucht.

Da auch die Implementierung des jeweiligen Sortieralgorithmus einen Einfluss auf die inhärente Fehlertoleranz haben kann, ist in diesem Fall ein einheitlicher Programmierstil hilfreich, sodass die jeweiligen Algorithmen alle von der gleichen Quelle, dem Rosetta Code Projekt (`http://rosettacode.org/wiki/Rosetta_Code`), stammen. Hier werden, für viele Algorithmen, Implementierungen in verschiedenen Programmiersprachen angeboten, unter anderem auch für viele gängige Sortieralgorithmen.

Konkret werden, die häufig in der Lehre vorkommenden und die effizienteren Algorithmen, sowie auch zwei andere zufällig ausgewählte Algorithmen, für den Vergleich benutzt und zwar folgende:

- **Insertionsort**
- **Selectionsort**
- **Bubblesort**
- **Quicksort**
- **Heapsort**
- **Mergesort**

- **Gnomesort**
- **Shellsort**

3.2.1.2 MiBench

Eine Beschränkung der Benchmarks nur auf verschiedene Sortieralgorithmen wäre allerdings nicht sinnvoll, da diese einige grundlegende algorithmische Gemeinsamkeit haben. Beim Sortieren werden allgemein nur Daten miteinander verglichen und vertauscht, es finden aber kaum Berechnungen statt. Da, für den Vergleich, eine größere Variation bei den Algorithmen benötigt wird, um allgemeingültigere Resultate zu erzielen, werden auch Benchmarks aus der MiBench Benchmark Suite verwendet. Diese hat auch den Vorteil, dass diese Algorithmen, im Gegensatz zu den verschiedenen Sortieralgorithmen, sehr häufig in der Forschung verwendet werden.

Konkret sind jeweils drei Algorithmen aus den Bereichen Automobil und Sicherheit für den Vergleich ausgesucht [15]:

- **Susan** (Automobil): Ein Bildbearbeitungsalgorithmus
- **Bitcount** (Automobil): Ein Zählalgorithmus für Bits (Auf einem Integer-Array)
- **Quicksort** (Automobil): Ein Sortieralgorithmus (Auf einem String-Array)
- **Rijndael** (Sicherheit): Ein Symmetrischer Verschlüsselungsalgorithmus (NIST Standard)
- **Blowfish** (Sicherheit): Ebenfalls ein verbreiteter Symmetrischer Verschlüsselungsalgorithmus
- **Sha** (Sicherheit): Ein Hashalgorithmus

3.2.1.3 eCos-Kernel-Test

Ebenso wie die MiBench Algoriothmen, sind diese Algorithmen vielfältiger als Sortieralgorithmen. Allerdings werden hier Anwendungen auf Betriebssystemebene getestet und nicht wie bei MiBench auf Anwenderebene. Zudem wurden diese Algorithmen ebenfalls bereits in der Forschung bei Fehlerinjektionsexperimenten verwendet [16]. Deswegen liegen sie auch als normaler Algorithmus und als abgesicherte CRC Variante vor, sodass es ein weiterer Vorteil ist, dass direkt ein konkretes Fehlertoleranzverfahren unter Ver-

wendung der verschiedenen Fehlermodelle betrachtet werden kann.

- **bin_sem1**: Funktionalität binärer Semaphore
- **bin_sem3**: Timeout bei binären Semaphoren
- **cnt_sem1**: Zählfunktionalität bei Semaphoren
- **except1**: Funktionalität von Exceptions
- **mbox1**: Funktionalität von Message-Boxen
- **mqueue1**: Warteschlangen
- **mutex1**: Grundlegende Mutex Funktionalität
- **mutex2**: Mutex-Release Funktionen
- **release**: Thread Releasefunktionen
- **sched1**: Scheduler Funktionen
- **sync3**: Prioritäten und Prioritätsvererbung
- **thread0**: Thread-Konstruktor und Thread-Destruktor
- **thread1**: Elementare Threadfunktionen

3.2.2 Datensätze

Eine Vielzahl Fehlermodelle und eine große Menge Benchmarks ist für die Untersuchung zwar sehr wichtig, lässt die Anzahl der Fehlerinjektionsexperimente allerdings sehr stark ansteigen, sodass der Zeitaufwand leicht zu groß werden kann und sie nicht mehr praktikabel zu bearbeiten sind. Wählt man die Datensätze für die Benchmarks allerdings zu klein, steigt die Wahrscheinlichkeit, dass wichtige Erkenntnisse übersehen werden, oder nur zufällig bei den verwendeten Daten Gültigkeit besitzen. Ziel muss es hier sein, einen goldenen Mittelweg zu finden, der, zwischen einer ausreichenden Größe und einem ausufernden Aufwand, abwägt.

Bei den MiBench Algorithmen wurden die Input-Daten aus der kleineren Variante (small) nochmals gekürzt, in dem zufällig Teile gelöscht wurden, wobei die Algorithmen aus dem Bereich Security weiterhin auf dem gleichen Input, einer Textdatei mit zufälligen Zah-

len und Buchstaben, arbeiten. Bei Quicksort wurde das Input-Array auf 100 Elemente gekürzt und bei Bitcount die Anzahl der Iterationen verringert.

Die Sortieralgorithmen arbeiten zwei drei verschiedenen Integer-Arrays als Input-Daten. Das erste Array ist bereits aufsteigend sortiert und enthält 100 Elemente, in diesel Fall einfach die Zahle 1-100. Das zweite enthält ebenfalls 100 Elemente, die aber zufällig mit $RANDOM erzeugt wurden und noch nicht sortiert sind.

3.2.3 Zeitliche Einschränkungen

Durch den Rahmen einer Bachelorarbeit, gibt es ein paar zeitliche Einschränkungen, die nicht zu vermeiden sind. Die verwendeten Datensätze wurden, wie im vorherigen Abschnitt beschrieben, sehr klein gehalten. Besonders bei den Aussagen über einzelne Algorithmen sollten diese durch weitere Daten überprüft werden. Für einen umfassenderen Vergleich müssten außerdem noch einige weitere Fehlermodelle mit einbezogen werden. Insbesondere sollten Fehler in anderen Komponenten als im RAM nicht nur durch andere Interpretation der Daten simuliert werden, sondern auch wirklich durchgeführt werden und anschließend deren Simulation verglichen werden. Dabei könnten dann mögliche Annäherungen und Gemeinsamkeiten aufzeigt werden und die Praktikabilität solcher Lösungen als Ersatz bewertet werden. Hier wäre besonders die Fehlerinjektion in Registern bedeutend, da diese, neben der Injektion im RAM, besonders häufig in der Praxis Verwendung findet.

4 Implementierung

Im folgenden Kapitel wird die Implementierung der Fehlermodelle und der dafür notwendigen Fehlerinjektion, auf Grundlage der vorangehenden Überlegungen aus Kapitel 3, erläutert. Dies ist dabei in zwei Teile gegliedert, zum einen muss erstmal überhaupt Fehlerinjektion durchgeführt werden, um anschließend Ergebnisse zu haben, die miteinander verglichen werden können. Die dafür verwendeten Implementierungen werden zunächst im Abschnitt 4.1 beschrieben. Zunächst wird hier die Verwendung des Fehlerinjektionstools, in diese Fall Fail*, erläutert und anschließend das konkret verwendete Projekt und die notwendigen Anpassungen genauer betrachtet.

Anschließend wird in Abschnitt 4.2 dargestellt, wie die verwendeten Fehlermodelle realisiert werden. Dazu wird erläutert, wie die Daten für das jeweilige Fehlermodell interpretiert werden müssen und deren, hauptsächlich auf Auswertung durch verschiedene SQL-Statements basierende, Implementierung beschrieben.

4.1 Fehlerinjektion

Um die Auswirkungen eines Soft-Errors simulieren zu können, sind Fehlerinjektionsexperimente ein geeignetes Mittel. Anstatt auf einen seltenen und nicht vorhersagbaren Soft-Error zu warten, wird ein Programm, zu eine festgelegten Zeitpunkt unterbrochen und an vorgegebener Stelle der Wert eines Bytes bzw. eines Bits geändert und anschließend das Ergebnis protokolliert. Dazu werden spezielle Fehlerinjektionstools benutzt, die auf hardware-, software- oder simulatorbasiert arbeiten. Wann und wo genau dabei die Fehler injiziert werden, wird in einem konkreten Experiment festgelegt.

4.1.1 FAIL*

Die im Rahmen dieser Arbeit durchgeführten Fehlerinjektionsexperimente wurden mit FAIL* durchgeführt, einem simulatorbasierten Fehlerinjektionstool, welches den Bochs x86 Simulator benutzt und in der Lage ist, Fehler im RAM und in Registern zu injizieren. Dabei kann das ganze System zu jedem Zeitpunkt angehalten, gespeichert und später

wieder geladen werden. So muss nicht für jede Fehlerinjektion das ganze Betriebssystem erneut geladen werden, da der Zustand zu Beginn des eigentlichen Programms, beim Ausführen des GoldenRuns, eines Durchlaufs, bei dem keine Fehler injiziert werden und der später als Referenz dient, gespeichert wird.

Eine vollständige Fehlerinjektion, die tatsächlich zu jedem Zeitpunkt an jeder Speicheradresse einen Fehler injiziert, wäre dennoch nicht machbar, da die Anzahl der Fehlerinjektionen zu groß wird. Allgemein wird deswegen erstmal nur der tatsächlich vom Programm verwendete Speicher betrachtet und die restlichen Bereiche ignoriert. Ein weiterer Mechanismus zur Reduzierung der Experimente sind Äquivalenzklassen im Fehlerraum. Eine Äquivalenzklasse fasst dabei alle möglichen Soft-Errors zwischen zwei Speicherzugriffen zusammen und führt für diese nur eine Fehlerinjektion durch.

Abbildung 4.1 zeigt einen Ausschnitt des Fehlerraums, mit einigen Speicherzugriffen, in dem zwei Äquivalenzklassen als Beispiel eingezeichnet sind. Alle Soft-Errors die zwischen W_0 und R_0 bzw. zwischen R_0 und R_1 auftreten haben die gleiche Auswirkung auf den Ablauf des Programms, da der Zeitpunkt, an dem der Fehler auftritt keine Relevanz besitzt, nur die Tatsache, dass die Daten an der Stelle verfälscht wurden. Es wird also

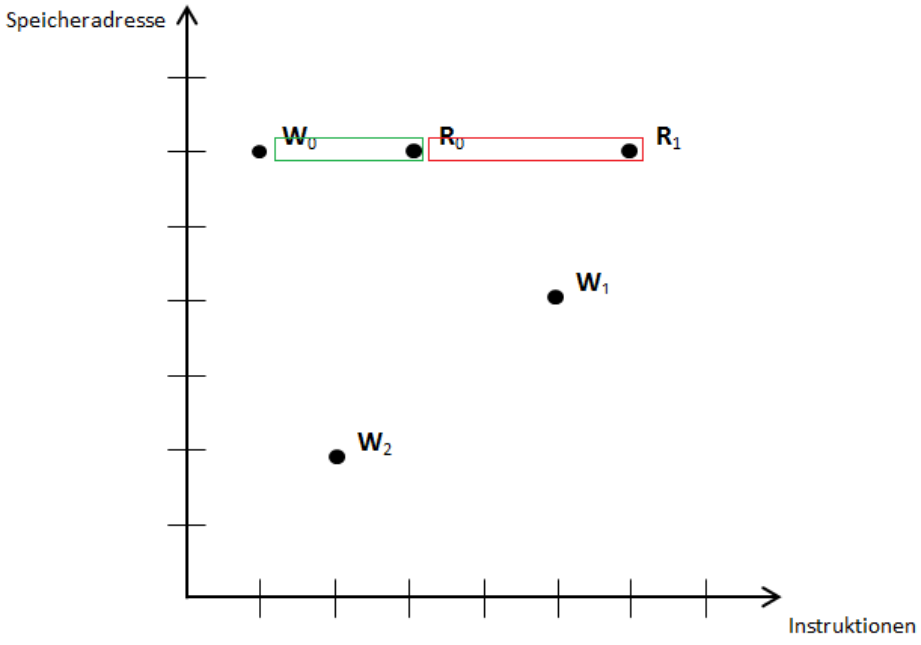

Abbildung 4.1: Beispiel Äquivalenzklassen im Fehlerraum

nur an der Stelle R_0 bzw. R_1 das Programm unterbrochen und vor dem Lesen des Bytes ein Bit verändert. Besonders bei Daten, die lange unbenutzt im Speicher liegen, ergibt dies eine erhebliche Verbesserung.

Eine weitere Optimierung, die FAIL* vor der tatsächlichen Fehlerinjektion anwendet,

ist die Zusammenfassung von Äquivalenzklassen, die mit einem Write enden. Da ein Soft-Error keine Auswirkungen haben kann, wenn er nicht benutzt, sondern nur ungelesen überschrieben wird, werden alle Injektionen vor einem Write keine Unterschiede im Ablauf des Programms verursachen. Um das zu realisieren wird exemplarisch genau einmal vor einem Write injiziert und das Ergebnis für alle theoretisch notwendigen Fehlerinjektionen verwendet.

Für die Umsetzung der Fehlerinjektion und zur Realisierung dieser Optimierungen werden die Tools Import-Trace und Prune-Trace verwendet, die in FAIL* integriert sind. Diese befüllen die Tabellen variant, trace, fsppilot, fspgroup und result_ExperimentSuffix, aus denen die durchzuführenden Experimente gelesen werden und in die die Ergebnisse anschließend geschrieben werden.

- **variant**: Für alle Kombinationen aus Variante und Benchmark wird eine ID festgelegt und in dieser Tabelle gespeichert (Beim Lesen des GoldenRuns durch Import-Trace)

- **trace**: In dieser Tabelle stehen die Speicherzugriffe, die während des Benchmarks ausgeführt werden und sie ist folglich die Grundlage, auf den die Experimente bestimmt werden. Sie enthält Informationen über die Äquivalenzklasse, die Zugriffsart, die Adresse, die Anzahl der Bytes und den Zeitpunkt

- **fsppilot**: Hier stehen alle tatsächlich durchzuführenden Fehlerinjektionen, mit einer eindeutigen ID.

- **fspgroup**: In dieser Tabelle stehen die Informationen, die zur Verknüpfung der theoretischen Fehlerinjektionen, die sich aus der trace Tabelle ergeben und der tatsächlichen Fehlerinjektionen notwendig sind.

- **result_ExperimentSuffix**: Hier werden die Ergebnisse eingetragen, jeweils für alle acht Bits, des Bytes, in das ein Fehler injiziert wurde. Zur Verknüpfung mit den anderen Tabellen dient die ID des dazugehörigen Pilots.

4.1.2 eCos-Kernel-Test Projekt

Für die konkrete Umsetzung der Fehlerinjektion in FAIL* wird ein Projekt benötig, in dem die entsprechenden Konfigurationen vorgenommen werden. Da das eCos-Kernel-Test Projekt bereits die meistens zu implementierenden Aspekte realisiert und ein Teil der Benchmarks dort ohne Änderungen läuft, wird dies mit kleinen Änderungen als Grundlage für die Fehlerinjektion verwendet.

Wichtigste Änderung ist dabei die Implementierung des Wiederherstellens eines Bytes,

auf dem ein Fehler injiziert wurde. Dies geschieht in den folgenden, in Abbildung 4.2 dargestellten, Schritten.

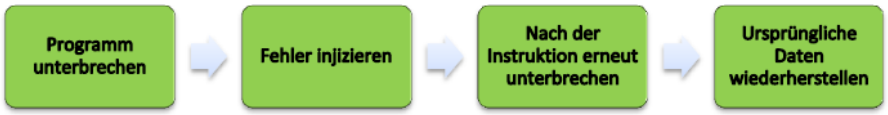

Abbildung 4.2: Beheben des injizierten Fehlers

Ein injizierter Fehler wird direkt nachdem er gelesen wurden, also vor einem erneuten Lesen, wieder auf den ursprünglichen Wert zurück gesetzt. Dies wird später für die Datenbus Fehlermodelle benötigt. Zur Umsetzung wird eine Namenskonvention verwendet, bei allen Benchmarks, deren Name auf _undo endet, werden Fehler wieder rückgängig gemacht. Die entsprechende Implementierung nutzt dabei ein Listener, der, wenn Fehler behoben werden sollen, direkt nach der Injektion, zum Simulator hinzugefügt wird und bei jedem Speicherzugriff (ANY_ADDR), also auch beim nächsten, das Programm unterbricht. Wie der Code-Ausschnitt 4.1 zeigt, wird der ursprüngliche Wert in der Variablen data gespeichert und nicht verändert, sodass das Beheben sehr einfach ist und analog zur Injektion funktioniert.

Listingverzeichnis 4.1: Fehlerinjektion

```
1 MemoryManager& mm = simulator.getMemoryManager();
2 byte_t data = mm.getByte(mem_addr);
3 byte_t newdata;
4 newdata = data ^ (1 << bit_offset);
5 mm.setByte(mem_addr, newdata);
```

Eine wichtige Kleinigkeit ist allerdings zu beachten, wenn vor eine Write injiziert wurde, darf natürlich nicht der alte Wert geladen werden, da sonst das Write und nicht der Fehler rückgängig gemacht wird. Im Ausschnit 4.2, ist der entsprechende Vergleich mit dem aktuellen Wert zu sehen. Würde dies nicht beachtet, sind die Auswirkungen ziemlich große, da es nur eine Injektion vor einem Write gibt und das Ergebnis für alle anderen verwendet wird.

Listingverzeichnis 4.2: Undo Fehlerinjektion

```
1 byte_t currentdata = mm.getByte(mem_addr);
2                 if(currentdata==newdata) {
3                   mm.setByte(mem_addr, data);
4                 }
```

4.2 Fehlermodelle

Der letzte Schritt zur Implementierung der Fehlermodelle, sind SQL-Statements zur Auswertung der in der Datenbank abgelegten Tabellen. In den folgenden Abschnitten wird dabei für jedes Fehlermodell zuerst kurz beschrieben, welche Einträge aus den Tabellen gelesen werden soll und anschließend das gegeben SQL-Statement erläutert.

4.2.1 Fehlermodell RAM Gleichverteilt

Das Fehlermodell sieht ein mögliches Auftreten von Soft-Errors zu jedem Zeitpunkt an jeder Stelle vor, sodass bei der Auswertung alle Ergebnisse der Fehlerinjektion verwendet werden. Um auch die Fehler zwischen den Speicherzugriffen mit einzubeziehen, wird mit der Größe der Äquivalenzklasse, also der Dauer seit dem letzten Speicherzugriff, gewichtet. Da keine Fehler wieder behoben werden, werden nur die Benchmarks verwendet, die gemäß der Namenskonvention nicht auf *_undo* enden.

Listingverzeichnis 4.3: SQL - Fehlermodell RAM Gleichverteilt

```
1  SELECT v.variant, v.benchmark, r.resulttype,
2  SUM((t.time2-t.time1+1)*t.width) AS occurrences
3  FROM result_EcosKernelTestProtoMsg r
4  JOIN fspgroup g ON r.pilot_id=g.pilot_id
5  JOIN trace t ON t.variant_id=g.variant_id
6  AND t.data_address=g.data_address
7  AND t.instr2=g.instr2
8  JOIN variant v ON t.variant_id=v.id
9  WHERE NOT v.benchmark LIKE '%undo'
10 GROUP BY v.id, r.resulttype
11 ORDER BY v.variant, v.benchmark, r.resulttype
12 ;
```

Die Größe der Äquivalenzklasse ist direkt in der *trace*-Tabelle gespeichert, sie beginnt zum Zeitpunkt time1 und ende zum Zeitpunkt time2. Dies wird mit der große der Speicherstelle an der injiziert wurde multipliziert, um so den ganzen abgedeckten Fehlerraum zu bekommen. Aufsummiert wird das Ergebnis dann nach ID der Variant/Benchmarks Kombination und dem Ergebnistyp gruppiert. Die Join-Befehle verknüpfen wie in 4.1.1 beschrieben dabei die benötigten Tabellen und in Zeile 9 werden nur die Benchmarks betrachtet, bei denen keine Behebung der Fehler stattfindet.

4.2.2 Fehlermodell Datenbus

Bei diesem Fehlermodell treten Soft-Errors nur bei Speicherzugriffen auf, sodass die Größe der Äquivalenzklasse keine Rolle spielt. Ansonsten gibt es keinen Unterschied zum vorangehenden Statement, es werden auch hier alle Ergebnisse verwendet, nur findet keine Gewichtung statt.

Listingverzeichnis 4.4: SQL - Fehlermodell Datenbus

```
1  SELECT v.variant, v.benchmark, r.resulttype,
2  SUM(t.width) AS occurrences
3  FROM result_EcosKernelTestProtoMsg r
4  JOIN fspgroup g ON r.pilot_id=g.pilot_id
5  JOIN trace t ON t.variant_id=g.variant_id
6  AND t.data_address=g.data_address
7  AND t.instr2=g.instr2
8  JOIN variant v ON t.variant_id=v.id
9  WHERE NOT v.benchmark LIKE '%undo'
10 GROUP BY v.id, r.resulttype
11 ORDER BY v.variant, v.benchmark, r.resulttype
12 ;
```

Die Implementierung unterscheidet sich folglich auch kaum, nur in Zeile 2, wird als Größe der Äquivalenzklasse ein angenommen, sodass diese in der Multiplikation komplett entfällt.

4.2.3 Fehlermodell Datenbus-Undo

Diese Variante des Fehlermodells Datenbus unterscheidet sich darin, dass Fehler wieder behoben werden. Da dies bei der Ausführung abhängig vom Namen des Benchmarks ist, müssen die Einträge gewählt werden, bei denen der Name des Benchmarks mit *_undo* endet.

Listingverzeichnis 4.5: SQL - Fehlermodell Datenbus Undo

```
1  SELECT v.variant, v.benchmark, r.resulttype,
2  SUM(t.width) AS occurrences
3  FROM result_EcosKernelTestProtoMsg r
4  JOIN fspgroup g ON r.pilot_id=g.pilot_id
5  JOIN trace t ON t.variant_id=g.variant_id
6  AND t.data_address=g.data_address
7  AND t.instr2=g.instr2
8  JOIN variant v ON t.variant_id=v.id
9  WHERE v.benchmark LIKE '%undo'
10 GROUP BY v.id, r.resulttype
11 ORDER BY v.variant, v.benchmark, r.resulttype
12 ;
```

Die Implementierung ändert sich folglich nur in Zeile 9.

4.2.4 Fehlermodell Datenbus-AfterWrite

Diese Variation des Fehlermodells sieht vor, dass Fehler im RAM erst nach eine Write stehen und nicht davor, wie es bei den vorherigen Implementierungen der Fall ist. Um für diese Änderungen nicht, wie beim Beheben der Fehlerinjektion, alle Experimente erneut in leicht veränderter Form durchführen zu müssen, werden die Ergebnisse alle Speicherzugriffe die nach einem Write kommen doppelt gezählt. Dies führt zum gleichen Ergebnis, da es kein Unterschied macht, wann zwischen diesen beiden Speicherzugriffen der Fehler auftritt und das Auftreten nach dem Write zum gleichen Ergebnis führt, wie das Auftreten vor dem nächsten Speicherzugriff.

Listingverzeichnis 4.6: SQL - Fehlermodell Datenbus AfterWrite

```
1  SELECT erg.variant, erg.benchmark, erg.resulttype,
2  SUM(erg.width) AS occurrences
3  FROM
4  (
5      SELECT  t.variant, t.benchmark, r.resulttype, t.width, t.id AS
           variant_id
6      FROM result_EcosKernelTestProtoMsg r
7      JOIN fspgroup g ON r.pilot_id=g.pilot_id
8      JOIN
9      (
10             SELECT v.variant, v.benchmark, v.id, tr.variant_id,
                   tr.data_address, tr.width, MIN(tr.instr2) AS instr2
11             FROM trace tr
12         JOIN variant v ON tr.variant_id=v.id
13             JOIN trace tw ON tr.variant_id = tw.variant_id
14             AND tr.data_address = tw.data_address
15             AND (tw.instr2+1) = tr.instr1
16             AND tw.accesstype = 'W'
17         WHERE NOT v.benchmark LIKE '%undo'
18             GROUP BY tw.variant_id, tw.data_address, tw.instr2
19     ) t ON t.variant_id=g.variant_id
20     AND t.data_address=g.data_address
21     AND t.instr2=g.instr2
22
23     UNION ALL
24
25     SELECT v2.variant, v2.benchmark, r2.resulttype,t2.width, v2.id AS
           variant_id
26     FROM result_EcosKernelTestProtoMsg r2
27     JOIN fspgroup g2 ON r2.pilot_id=g2.pilot_id
28     JOIN trace t2 ON t2.variant_id=g2.variant_id
29     AND t2.data_address=g2.data_address
30     AND t2.instr2=g2.instr2
31     JOIN variant v2 ON t2.variant_id=v2.id
```

```
32      WHERE t2.accesstype='R' AND NOT v2.benchmark LIKE '%undo'
33  ) erg
34  GROUP BY  erg.variant_id, erg.resulttype
35  ORDER BY erg.variant, erg.benchmark, erg.resulttype
36  ;
```

Implementiert wird dies über mehrere gegliederte Unteranfragen. In Zeile 10 bis 19 wird der relevante Teil ein *trace*-Tabelle ausgegeben, allerdings nur die Einträge, die auf ein Write folgen. Um dies zu bestimmen, werden wieder die Äquivalenzklassen verwendet, auch wenn diese nicht ins Ergebnis einfließen. Ein Speicherzugriff erfolgt nach eine Write, wenn die erste Instruktion einer Äquivalenzklasse direkt auf die Write-Instruktion folgt. Dies wird in Zeile 15 geprüft. Diese veränderte *trace*-Tabelle wird statt einer normalen verwendet um in dem Bereich von Zeile 5 bis 21 alle Ergebnisse für Fehler nach einem Write zu erhalten. Da allerdings auch zusätzlich vor Reads Fehler auftreten können, wird das Ergebnis mit dem Ergebnis einer zweiten Abfrage vereinigt. Diese steht von Zeile 25 bis 33 und ist das gleiche Statement wie beim Fehlermodell Datenbus, nur auf alle Reads reduziert (Zeile32). Das Resultat wird dann nur noch sortiert und aufsummiert.

4.2.5 Fehlermodell Datenbus-AfterWrite Undo

Dieses Fehlermodell beinhaltet beide Variationen, sodass es sich nur bei den zugrunde liegenden Benchmarks vom vorherigen Fehlermodell unterscheidet.

Listingverzeichnis 4.7: SQL - Fehlermodell Datenbus AfterWrite Undo

```
1  SELECT erg.variant, erg.benchmark, erg.resulttype,
2  SUM(erg.width) AS occurrences
3  FROM
4  (
5      SELECT  t.variant, t.benchmark, r.resulttype, t.width, t.id AS
           variant_id
6      FROM result_EcosKernelTestProtoMsg r
7      JOIN fspgroup g ON r.pilot_id=g.pilot_id
8      JOIN
9      (
10             SELECT v.variant, v.benchmark, v.id, tr.variant_id,
                   tr.data_address, tr.width, MIN(tr.instr2) AS instr2
11             FROM trace tr
12         JOIN variant v ON tr.variant_id=v.id
13             JOIN trace tw ON tr.variant_id = tw.variant_id
14             AND tr.data_address = tw.data_address
15             AND (tw.instr2+1) = tr.instr1
16             AND tw.accesstype = 'W'
17         WHERE v.benchmark LIKE '%undo'
18             GROUP BY tw.variant_id, tw.data_address, tw.instr2
19     ) t ON t.variant_id=g.variant_id
20     AND t.data_address=g.data_address
```

```
21      AND t.instr2=g.instr2
22
23      UNION ALL
24
25      SELECT v2.variant, v2.benchmark, r2.resulttype,t2.width, v2.id AS
            variant_id
26      FROM result_EcosKernelTestProtoMsg r2
27      JOIN fspgroup g2 ON r2.pilot_id=g2.pilot_id
28      JOIN trace t2 ON t2.variant_id=g2.variant_id
29      AND t2.data_address=g2.data_address
30      AND t2.instr2=g2.instr2
31      JOIN variant v2 ON t2.variant_id=v2.id
32      WHERE t2.accesstype='R' AND v2.benchmark LIKE '%undo'
33  ) erg
34  GROUP BY erg.variant_id, erg.resulttype
35  ORDER BY erg.variant, erg.benchmark, erg.resulttype
36  ;
```

Die Implementierung ändert sich folglich auch nur kaum, Zeile 17 und 32 wurden entsprechend angepasst.

4.2.6 Fehlermodell CPU

Beim letzten Modell wird versucht, CPU Fehler durch ihre Auswirkungen im RAM zu realisieren. Die Idee ist, dass unabhängig davon, wo genau ein Fehler bei der Berechnung in der CPU entsteht, dieser irgendwann in den RAM geschrieben wird und sich so auf das Programm auswirkt. Es werden also nur Fehler nach Writes injiziert. Wie eben ist auch hier die Überlegung, stattdessen die Ergebnisse der Fehlerinjektion beim nächsten Speicherzugriff zu verwenden, da so keine neuen Experimente durchgeführt werden müssen.

Listingverzeichnis 4.8: SQL - Fehlermodell CPU

```
1  SELECT t.variant, t.benchmark, r.resulttype,
2  SUM(t.width) AS occurrences
3  FROM result_EcosKernelTestProtoMsg r
4  JOIN fspgroup g ON r.pilot_id=g.pilot_id
5  JOIN (
6      SELECT v.variant, v.benchmark, v.id AS variant_id, tr.data_address,
           tr.width, MIN(tr.instr2) AS instr2
7      FROM trace tr
8      JOIN variant v ON v.id = tr.variant_id
9      JOIN trace tw ON tr.variant_id = tw.variant_id
10     AND tr.data_address = tw.data_address
11     AND (tw.instr2+1) = tr.instr1
12     AND tw.accesstype = 'W'
13     WHERE NOT v.benchmark LIKE '%undo'
14     GROUP BY tw.variant_id, tw.data_address, tw.instr2
```

```
15 )
16 t ON t.variant_id=g.variant_id
17 AND t.data_address=g.data_address
18 AND t.instr2=g.instr2
19 GROUP BY t.variant_id, r.resulttype
20 ORDER BY t.variant, t.benchmark, r.resulttype
21 ;
```

Die Implementierung entspricht dabei dem ersten Teil des Fehlermodells Datenbus-AfterWrite.

5 Evaluation

Die Ergebnisse der Fehlerinjektion müssen abschließend noch miteinander verglichen werden, um konkrete Ergebnisse bezüglich der Fehlermodelle zu erhalten. Dabei gliedert sich die Evaluation in mehrere Teile, zunächst wird in Abschnitt 5.1 ein Vergleich der Variationen des Fehlermodells Datenbus angestellt, um bewerten zu können, welche Präzisierungen einen Einfluss auf das Ergebnis haben und wie weit die ursprüngliche Realisierung Fehler auf dem Datenbus tatsächlich repräsentiert. Anschließend werden in Abschnitt 5.2 alle relevanten Fehlermodelle miteinander verglichen und dadurch der Einfluss auf die Auswertung von Fehlerinjektionsexperimenten gezeigt. Abschließend werden die gewonnen Daten genauer untersucht, um das Nebenziel dieser Arbeit zu erreichen und Aussagen über die inhärente Fehlertoleranz von Algorithmen zu machen, insbesondere bei Sortieralgorithmen.

Im Folgenden Kapitel werden sehr viele Diagramme verwendet, die alle die Anzahl der unterschiedlichen Ergebnistypen der Benchmarks repräsentieren. Um die Übersichtlichkeit dabei nicht einzuschränken, verwenden alle die gleiche, in Abbildung 5.1 angegebene Färbung für die jeweiligen Ergebnisarten.

5.1 Fehlermodell Datenbus

Für den Großteil der Benchmarks, lassen sich einige Gemeinsamkeiten feststellen, wie in Abbildung 5.2 gut zu erkennen ist. So steigt die Anzahl der erfolgreichen Ausführung, wenn Fehler wieder behoben werden. Dies ist leicht zu erklären, denn ein Fehler der mehrfach gelesen wird, kann, wenn er nicht behoben wird auch mehrfach eine fehlerhafte Ausführung verursachen. Bei der Variation Fehler erst nach einem Write zu injizieren und nicht davor, sinkt die Gesamtzahl der Ergebnisse, leicht, weil nach einem Write kein Fehler mehr injiziert wird, wenn es kein darauffolgenden Speicherzugriff gibt. Allerdings steigt die Anzahl der fehlerhaften Ausführungen, teilweise sehr deutlich an. Dies liegt daran, dass ein Fehler vor einem Write keine Auswirkungen hat, während die Injektion danach, einen fehlerhaften Ablauf verursachen kann.

Um die Relevanz der einzelnen Variationen beurteilen zu können, muss nun geschaut werden, wie groß die Unterschiede jeweils sind und folglich, ob man sie vernachlässigen

kann. Besonders die Variante, einen Fehler zu beheben scheint dabei keine großen Auswirkungen auf die Ausführung der jeweiligen Benchmarks zu haben, da die Ergebnisse kaum variieren. Es gibt aber einige Ausnahmen, besonders bei den Benchmarks, die in Abbildung 5.3 gezeigt werden.

SelectionSort teilt das zu sortierende Array in einen sortiert und einen unsortierten Bereich und sucht dann im unsortierten Bereich jeweils das Minimum, vertauscht es mit dem ersten Eintrag dieses Bereichs und vergrößert den sortierten Bereich um 1. Entscheidend für den Unterschied der Varianten ist also hier die Tatsache, dass alle Werte sehr häufig gelesen werden, ohne anschließend benutzt zu werden. Wird also ein falscher Wert gelesen, hat dies keine Auswirkungen, wenn es nicht das Minimum ist, da der Wert anschließend erneut gelesen wird. Bleibt der injizierte Fehler allerdings erhalten, steht ein falscher Wert im Array, der entsprechend auch das Ergebnis verfälscht, selbst wenn er richtig einsortiert ist.

Interessant ist auch, dass dieser Effekt auch andersherum auftreten kann und ein Benchmark deutlich mehr fehlerhafte Ergebnisse aufweist, wenn ein injizierter Fehler behoben wird. Dies tritt z. B. bei *except1* auf. Die Variation des Fehlermodells ist also nur sehr selten relevant, es gibt aber einige Benchmarks, bei denen teilweise sehr große Unterschiede auftreten, sodass sie als Verbesserung des Fehlermodells Datenbus auch weiterhin betrachtet werden muss.

Für die Variation, Fehler erst nach einem Write zu injizieren, ist der Unterschied allgemein deutlich Größer, sodass auch diese Variation relevant scheint. Besonders bei Benchmarks mit vielen Writes, kann der Unterschied sehr groß werden, weil jede Injektion vor einem Write keinen Fehler verursacht, dies bei einer Injektion danach allerdings möglich ist. Ein Teil der normalen Durchläufe des Programms wird also durch fehlerhafte ersetzt. Wir stark die Anzahl dabei steigt, hängt vom jeweiligen Algorithmus ab. Besonders auffällige Beispiele sind dabei in Abbildung 5.4 zu sehen. Beim *sha*-Algorithmus z. B. hat fast jeder injizierte Fehler eine Auswirkung auf das Ergebnis, wenn er berücksichtigt wird, sodass es kaum noch korrekte Ausführungen gibt, wenn der Fehler nicht vor seiner Benutzung durch ein Write wieder überschrieben wird.

Da also beide Variationen relevant für die Umsetzung des Fehlermodells Datenbus sind, wird im Folgenden beim Vergleich nur noch das Fehlermodell verwendet, das beide Änderungen enthält und nicht mehr die Fehlermodelle *Datenbus-Undo* und *DatenbusAfter-Write*. Das Fehlermodell Datenbus wird auch weiter berücksichtigt, weil es in der Praxis häufig verwendet wird, da es dem Fehlermodell *RAM Read /Write* entspricht.

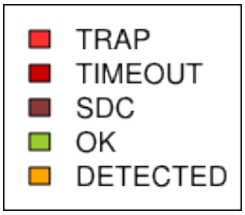

Abbildung 5.1: Färbung der jeweligen Ergebnistypen (Legende)

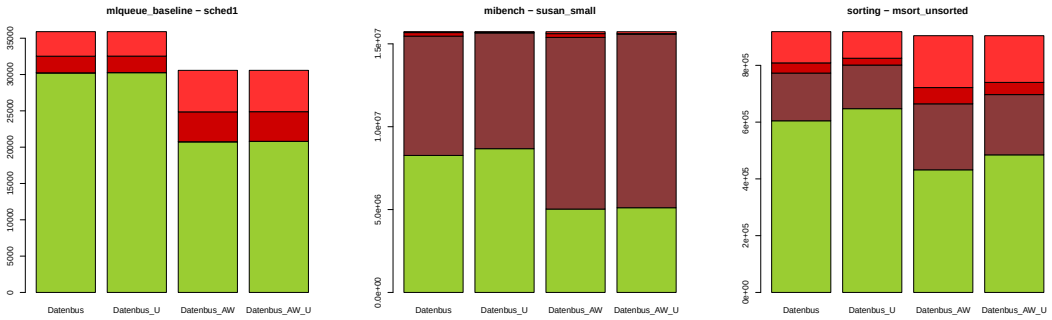

Abbildung 5.2: sched1 / susan_small / msort_unsorted

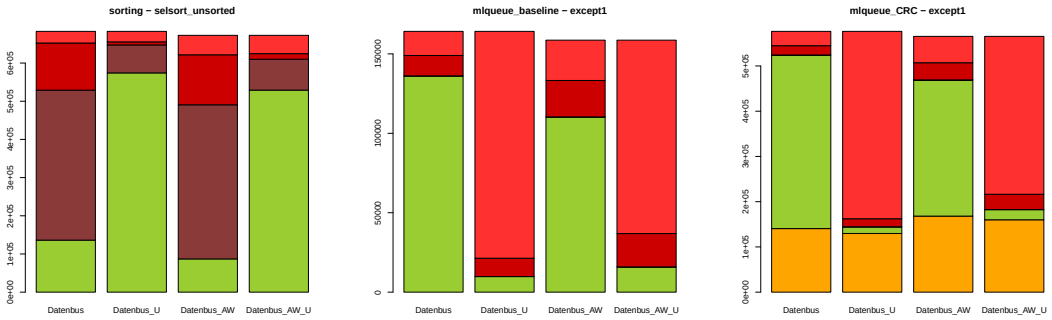

Abbildung 5.3: selsort_unsorted / except1 (CRC & baseline)

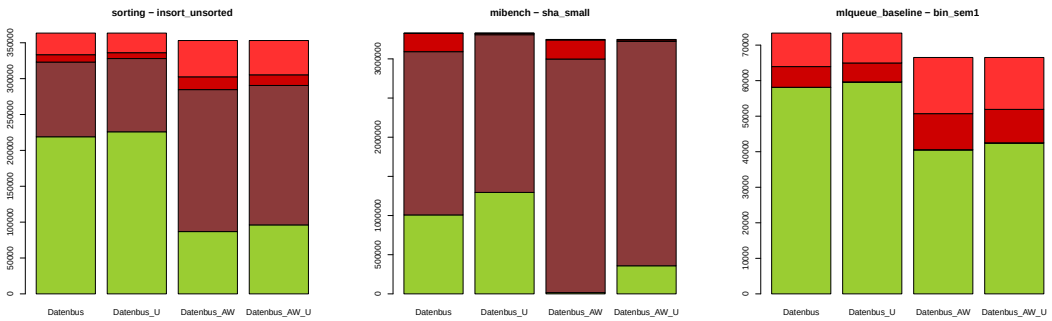

Abbildung 5.4: insort_unsorted / sha_small / bin_sem1

5.2 Vergleich Fehlermodelle

In diesem Abschnitt werden die Fehlermodelle *RAM*, *Datenbus*, *Datenbus AfterWrite Undo* und *CPU miteinander verglichen.*. Abbildungen 5.5, 5.6, 5.7 und 5.8 geben zunächst einen Überblick, über die Ergebnisse der Benchmarks.Die drei Benchmarks *bin_sem3*, *mbox1* und *thread1* werden zur besseren Skalierung nicht mit angezeigt.

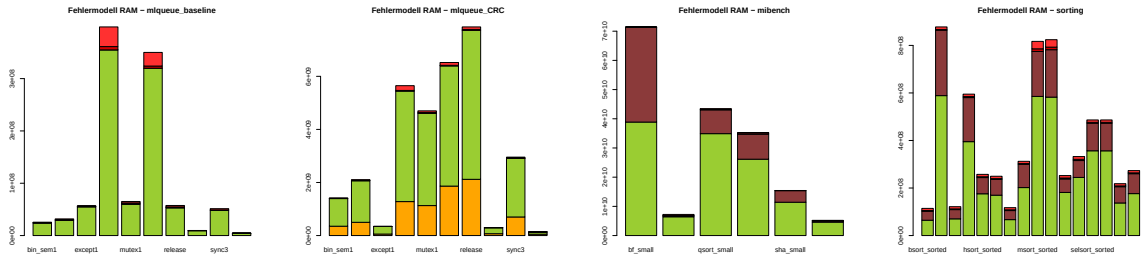

Abbildung 5.5: Fehlermoell RAM: Base / CRC / Mibench / Sort

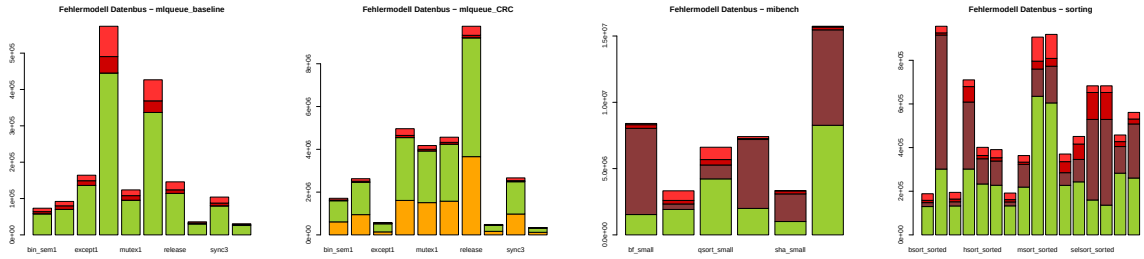

Abbildung 5.6: Fehlermoell Datenbus: Base / CRC / Mibench / Sort

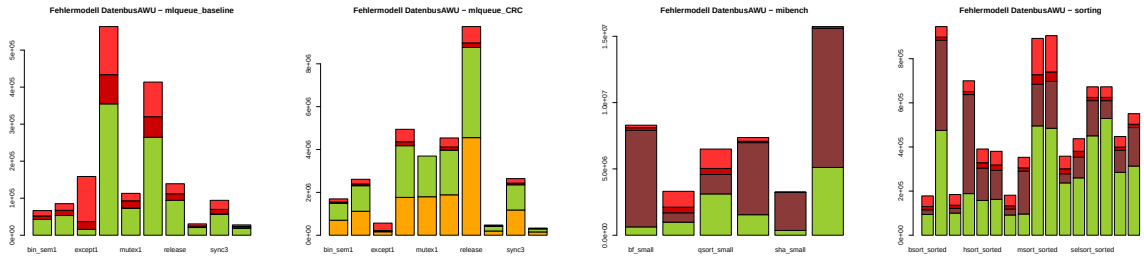

Abbildung 5.7: Fehlermoell Datenbus AfterWrite Undo: Base / CRC / Mibench / Sort

Als erste ist zu bemerken, dass die Anzahl der fehlerhaften Programmabläufe im Fehlermodell *RAM* natürlich mit Abstand am größten ist, da hier jederzeit auftretende Fehler angenommen werde und es folglich viel mehr Möglichkeiten gibt, den Programmablauf zu verfälschen. Beim Fehlermodell *Datenbus* hingegen sind Soft-Errors nur noch bei einem Speicherzugriff möglich, sodass die Gesamtzahl natürlich sinkt. Da beim Fehlermodell

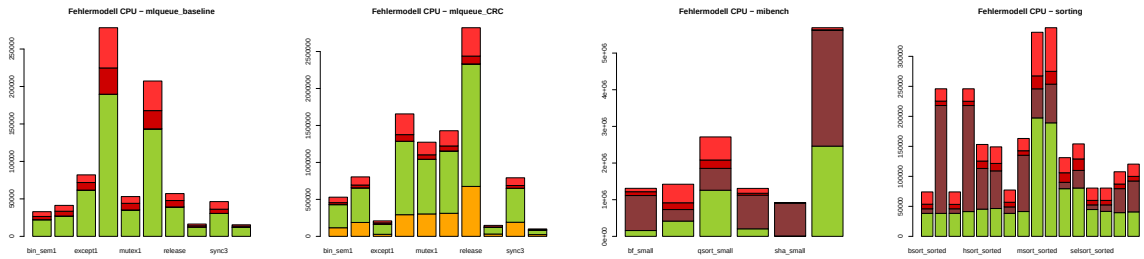

Abbildung 5.8: Fehlermoell CPU: Base / CRC / Mibench / Sort

CPU auch von dieser Menge nur noch ein Teil angenommen wird, wobei noch zu prüfen bleibt, wie weit die Implementierung tatsächlich an Fehler in der CPU herankommt, gibt es auch hier weniger Möglichkeiten einen Fehler zu injizieren. Dieser Unterschied in der Anzahl ist allerdings nur bei Abschätzungen bezüglich des realen Auftretens von gestörten Programmausführungen relevant. Die Angabe, welches Fehlermodell verwendet wurde, ist also besonders wichtig, wenn Ergebnisse in tatsächlich auftretenden Fehler angegeben werden und nicht die anteilige Verbesserung als Maßstab gewählt wird.

Zudem variiert je nach Fehlermodell nicht nur die tatsächliche Anzahl der jeweiligen Ergebnisse, auch das Verhältnis zueinander fällt bei einigen Benchmarks besonders unterschiedlich aus. Die Tabelle 5.1 zeigt die Ergebnisse für den *blowfish*-Algorithmus, bei denen nach Fehlermodell *RAM* die Anzahl der fehlerhaften und normalen Ausführungen fast gleich ist. Unter Verwendung eines der anderen Fehlermodelle machen die normalen Ausführungen nur noch einen Bruchteil aus. Auch zwischen den anderen Modellen unterscheiden sich die Verhältnisse sehr stark. Ein weiteres Beispiel sind die Anzahl der Traps, die in den Fehlermodellen *Datenbus* und *CPU* nahezu identisch ist, während die Gesamtzahl in etwa einem Sechsteil entspricht.

Dies gilt dabei auch für andere Benchmarks, besonders bei den Mibench Algorithmen. Tabelle 5.2 zeigt die Ergebnisse für den *sha*-Algorithmus bei dem der Anteil, der fehlerfreien Ausführung von einem Großteil, zur knapp der Hälfte, einem Bruchteil und fast gar nicht mehr vorhanden, je nach Fehlermodell variiert.

Und das wichtigste bezüglich der Vergleichbarkeit, auch Änderungen, wie z. B. Fehlertoleranzmechanismen können unterschiedliche Auswirkungen haben. In Kapitel 2.2 wurde an einem Beispiel bereits gezeigt, dass ein gutes Fehlertoleranzverfahren bezüglich eines bestimmten Fehlermodells nicht immer gute Ergebnisse bei der Verwendung eines anderen Fehlermodells bringt. Ebenso wie beim Fehlermodell *Datenbus*, steigt die Anzahl fehlerhafter Ergebnisse bei den Fehlermodellen *Datenbus AfterWrite Undo* und *CPU*, wie Tabelle 5.3, eine Erweiterung der Tabelle 2.1, zeigt.

Zusammenfassend ist zu erkennen, dass Ergebnisse, die unter Annahme verschiedener Fehlermodelle gewonnen wurden, nicht miteinander vergleichbar sind. Zwar gilt das nicht

Tabelle 5.1: Fehlerinjektionsergebnisse - Mibench - bf_small

Fehlermodell	Variant	Benchmark	Result	Occurrences
RAM Gleichverteilt	mibench	bf_small	OK	38.874.595.373
RAM Gleichverteilt	mibench	bf_small	SDC	32.473.845.708
RAM Gleichverteilt	mibench	bf_small	TIMEOUT	72.162.224
RAM Gleichverteilt	mibench	bf_small	TRAP	37.535.359
Datenbus	mibench	bf_small	OK	1.534.096
Datenbus	mibench	bf_small	SDC	6.504.058
Datenbus	mibench	bf_small	TIMEOUT	273.807
Datenbus	mibench	bf_small	TRAP	92.983
Datenbus AW U	mibench	bf_small	OK	628.123
Datenbus AW U	mibench	bf_small	SDC	7.274.031
Datenbus AW U	mibench	bf_small	TIMEOUT	208.908
Datenbus AW U	mibench	bf_small	TRAP	186.258
CPU	mibench	bf_small	OK	158.844
CPU	mibench	bf_small	SDC	954.968
CPU	mibench	bf_small	TIMEOUT	104.204
CPU	mibench	bf_small	TRAP	92.432

Tabelle 5.2: Fehlerinjektionsergebnisse - Mibench - sha_small

Fehlermodell	Variant	Benchmark	Result	Occurrences
RAM Gleichverteilt	mibench	sha_small	OK	11.418.047.960
RAM Gleichverteilt	mibench	sha_small	SDC	3.975.284.518
RAM Gleichverteilt	mibench	sha_small	TIMEOUT	21.435.713
RAM Gleichverteilt	mibench	sha_small	TRAP	22.222.513
Datenbus	mibench	sha_small	OK	1.007.902
Datenbus	mibench	sha_small	SDC	2.084.136
Datenbus	mibench	sha_small	TIMEOUT	234.000
Datenbus	mibench	sha_small	TRAP	3.314
Datenbus AW U	mibench	sha_small	OK	358.692
Datenbus AW U	mibench	sha_small	SDC	2.863.650
Datenbus AW U	mibench	sha_small	TIMEOUT	19.120
Datenbus AW U	mibench	sha_small	TRAP	6.626
CPU	mibench	sha_small	OK	10.636
CPU	mibench	sha_small	SDC	897.787
CPU	mibench	sha_small	TIMEOUT	9.391
CPU	mibench	sha_small	TRAP	3.314

Tabelle 5.3: Fehlerinjektionsergebnisse (Erweitert)

Fehlermodell	Variant	Benchmark	Result	Occurrences
RAM Gleichverteilt	mlqueue_baseline	bin_sem3	OK	2.779.122.696.545
RAM Gleichverteilt	mlqueue_baseline	bin_sem3	FAIL	377.544.865.695
RAM Gleichverteilt	mlqueue_CRC	bin_sem3	OK	6.128.528.502.091
RAM Gleichverteilt	mlqueue_CRC	bin_sem3	FAIL	98.437.448.597
Datenbus	mlqueue_baseline	bin_sem3	OK	121.790
Datenbus	mlqueue_baseline	bin_sem3	FAIL	35.018
Datenbus	mlqueue_CRC	bin_sem3	OK	3.191.751
Datenbus	mlqueue_CRC	bin_sem3	FAIL	234.945
Datenbus AW U	mlqueue_baseline	bin_sem3	OK	91.794
Datenbus AW U	mlqueue_baseline	bin_sem3	FAIL	53.174
Datenbus AW U	mlqueue_CRC	bin_sem3	OK	2.980.037
Datenbus AW U	mlqueue_CRC	bin_sem3	FAIL	423.307
CPU	mlqueue_baseline	bin_sem3	OK	44.825
CPU	mlqueue_baseline	bin_sem3	FAIL	24.391
CPU	mlqueue_CRC	bin_sem3	OK	853.871
CPU	mlqueue_CRC	bin_sem3	FAIL	211.489

immer, manche ähneln sich sogar bei vielen Benchmarks, allerdings nicht bei allen. Daraus folgt der Schluss, dass der Grad der Vergleichbarkeit von den jeweiligen Algorithmen und Benchmarks abhängt. Je nach Fehlermodell sind die Ergebnisse von Fehlerinjektionen an unterschiedlichen Stellen besonders wichtig und es ist entscheidend, wie sehr Fehlertoleranzverfahren an genau dieser Stelle greifen.

5.3 Fehlertoleranz bei Sortieralgorithmen

Als Nebenziel dieser Arbeit, werden die Sortieralgorithmen auf ihre Fehlertoleranz bei Annahme der verschiedenen Fehlermodelle miteinander verglichen.

- **Fehlermodell RAM**: Bei sortierten Daten sind hier die Algorithmen besser, die im Falle einer sortierten Datenstruktur nichts mit den Datenmachen und diese nur einmal angucken. Es gibt so einfach sehr wenige Möglichkeiten, bei denen ein Fehler etwas zerstören kann. Ansonsten sind Quicksort und Heapsort und Insertionsort die fehlertolerantesten Algorithmen.

- **Fehlermodell Datenbus**: Die Ergebnisse sind für dieses Fehlermodell sehr ähnlich, wenn tatsächlich sortiert wird, sind auch hier Quicksort, Heapsort und Insertionsort gute Algorithmen

- **Fehlermodell Datenbus AWU**: Bei diesem Fehlermodell ist der Unterschied bei unsortierten Datensätzen relativ groß, Selectionsort und Quicksort sind hier die tolerantesten Algorithmen

- **Fehlermodell CPU**: Bei diesem Fehlermodell ist Selectionsort deutlich der beste Algorithmus für unsortierte Eingaben.

Quicksort zeigt sich allgemein als einer der fehlertolerantesten Sortieralgorithmen bei allen verwendeten Fehlermodellen. Dies liegt, dass Quicksort Werte mehrfach vertauscht und ein falscher Wert so nicht immer dauerhafte Auswirkungen hat. Die Werte werden dabei nur in einen bestimmten Bereich einsortiert und erst später genau positioniert. Nur beim letzten Fehlermodell ist Quicksort im Vergleich nicht so gut, Werte sehr häufig vertauscht werden und folglich viele Schreibzugriffe. Deswegen ist beim Fehlermodell *CPU* Selectionsort sehr fehlertolerant, da fast nur lesende Speicherzugriffe gemacht werden und nur einmal der Wert an die richtige Stelle geschrieben wird.

6 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde exemplarisch an einigen Fehlermodellen untersucht, wir stark sich Ergebnisse von Fehlerinjektionsexperimenten unterscheiden, wenn unterschiedliche Fehlermodelle verwendet werden. Hierzu wurde zunächst an einem Beispiel demonstriert, das entwickelte Fehlertoleranzverfahren, die eine deutlich messbare Steigerung der Fehlertoleranz erzielen, bei Annahme anderen Fehlermodellen, die Fehleranfälligkeit sogar noch erhöhen. Die Evaluation des Vergleichs ergab dabei, dass für es alle untersuchten Fehlermodelle, Fälle gibt, bei denen deutliche Unterschiede auftreten und folglich Ergebnisse von Fehlertoleranzuntersuchungen nur in Abhängigkeit vom verwendeten Fehlermodell verwertbare Ergebnisse liefern.

Trotz dieser Bedeutung der Fehlermodelle, werden sie häufig nicht ausreichend angegeben und lassen sich teilweise nur aus dem Kontext oder gar nicht erkennen. Diese Problematik wird zukünftig noch zunehmen, da mit sinkender Strukturbreite und geringer werdendem Energieniveau neuer Hardware, Soft-Errors auch in weiteren Komponenten wahrscheinlicher werden.

Die inhärente Fehlertoleranz von Algorithmen hängt dabei ebenfalls vom verwendeten Fehlermodell ab. Allgemein gilt, dass in den meisten Fällen kurze Laufzeit wenig Möglichkeiten zum Auftreten von Soft-Errors bieten und damit die Fehlertoleranz steigern, ein wichtigerer Aspekt ist aber eine dem Algorithmus innewohnende Redundanz, sodass durch mehrfachen Verwendung von Daten, Fehler durch erneute Benutzung wieder behoben.

Für zukünftige Untersuchungen im Bereich von Fehlermodellen, ist der Vergleich mit weiteren Fehlermodellen Interessant, auch die Realisierung von Fehlern auf dem Datenbus oder in der CPU durch Injektion im RAM, sollte mit tatsächlicher Injektion in diesen Komponenten verglichen werden, da dies eine leichtere Implementierung anderer Fehlermodelle ermöglicht.

Literaturverzeichnis

[1] May, Timothy C. ; Woods, Murray H.: Alpha-Particle-Induced Soft Errors in Dynamic Memories. In: *Electron Devices, IEEE Transaction on*, IEEE, Januar 1979, S. 2–9

[2] Juhnke, Dipl.-Ing. T.: *Die Soft-Error-Rate von Submikrometer-CMOS-Logischaltungen*, Technische Universität Berlin, Diss., 2003

[3] Mukherjee, Shubu: *Architecture Design for Soft Errors*. Morgan Kaufman Publishers, 2008

[4] Srinivasan, G. R.: Modeling the cosmic-ray-induced soft-error rate in integrated circuits. In: *IBM Journal of Research and Development* 40 (1996), Januar, Nr. 1, S. 77–89

[5] Siemers, Prof. Dr. C.: Soft Errors durch Störenfriede aus dem Mikro- und Makro-Kosmos. In: *Elektronik Praxis* (2008), Januar

[6] Schirmeier, H. ; Hoffmann, M. ; Kapitza, R. ; Lohmann, D. ; Schröder-Preikschat, W. ; Schmid, R.: 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

[7] Sangchoolie, Behrooz ; Ayatolahi, Fatemeh ; Johansson, Roger ; Karlsson, Johan: A Study of the Impact of Bit-flip Errors on Programs Compiled with Different Optimization Levels. In: *Dependable Computing Conference (EDCC)*, IEEE, Mai 2014

[8] Rebaudengo, M. ; Reorda, M.S. ; Torchiano, Marco ; Violante, M.: Soft-error detection through software fault-tolerance techniques. In: *Defect and Fault Tolerance in VLSI Systems*, IEEE, November 1999, S. 210–218

[9] Ulbrich, P. ; Schroder-Preikschat, Wolfgang ; Hoffmann, M. ; Kapitza, R. u. a.: Eliminating single points of failure in software-based redundancy. In: *Dependable Computing Conference (EDCC)*, IEEE, 2012

[10] Leo, Domenico D. ; Ayatolahi, Fatemeh ; Sangchoolie, Behrooz ; Karlsson, Johan ; Johansson, Roger: *Lecture Notes in Computer Science*. Bd. 7612: *On the Impact of Hardware Faults – An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions*. Springer Berlin Heidelberg, 2012. – 198–209 S.

[11] Goloubeva, O ; Rebaudengo, M ; Reorda, M.S. ; Violante, M: Soft-error detection using control flow assertions . In: *Defect and Fault Tolerance in VLSI Systems*, IEEE, November 2003

[12] Reis, George A. ; Chang, Jonathan ; Vachharajani, Neil ; Ranga, Ram ; August, David I.: SWIFT - Software Implemented Fault Tolerance. In: *International symposium on Code generation and optimization* , IEEE Computer Society, 2005

[13] Ayatolahi, Fatemeh ; Sangchoolie, Behrooz ; Johansson, Roger ; Karlsson, Johann: *A Study of the Impact of Sinlge Bit-Flip and Double bit-Flip Errors on Program Execution*. Springer, 2013. – 265–276 S.

[14] Tezzaron Semiconductor: Soft Errors in Electronic Memory - A White Paper. Tezzaron Semiconductor, Januar 2004. – Forschungsbericht

[15] Guthaus, M.R. ; Ringenberg, J.S. ; Ernst, D. u. a.: MiBench: A free, commercially representative embedded benchmark suite. In: *IEEE International Workshop Workload Characterization*, IEEE, Dezember 2001

[16] Borchert, C. ; Schirmeier, H. ; Spinczyk, O.: 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

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis