technische universität
dortmund

Masterarbeit

Selektion von Compileroptimierungen für fehlertoleranten Code

Marcel Johannfunke
10. Januar 2019

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

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

Zusammenfassung

Im Bereich der Fehlertoleranz werden Optimierungen zur Verbesserung selbiger typischerweise pro Programm bzw. Problemfall selektiert. Das benötigt jedoch immer etwas Handarbeit und viel Rechenzeit. Für kleinere Projekte, z. B. Kleinstsatelliten, stehen die dafür nötigen Ressourcen oft nicht zur Verfügung. In dieser Arbeit wird daher ein Verfahren entwickelt, welches für einen gegebenen Compiler eine Menge an Optimierungen sucht, die im Durchschnitt besser als die Standard-Optimierungslevel bzgl. der Fehlertoleranz sind. Dafür wird ein Optimieralgorithmus entwickelt, welcher auf Basis eines genetischen Algorithmus arbeitet. Er beachtet dabei einige Probleme, die durch die Injektion in größere (Benchmark-)Programme verursacht werden. In Verbindung mit dem Fehlerinjektionstool FAIL* wird so ein Gesamtsystem entwickelt, welches genau solche gesuchten Optimierungsmengen finden kann. In der Evaluation zeigt sich, am Beispiel der GNU Compiler Collection, dass dieses auch erfolgreich funktioniert: Es wurden mehrere Mengen gefunden, welche besser sind und dabei mindestens eine Verbesserung von $10\,\%$ gegenüber den Standard-Optimierungslevel erreichen.

Inhaltsverzeichnis

Akronyme

DyFIGA

Dynamic Fault Injection Genetic Algorithm.

EAFC

Extrapolated Absolute Failure Count.

GA

genetischer Algorithmus.

GCC

GNU Compiler Collection.

SDC

Silent Data Corruption.

1 Einleitung

Die internen elektrischen Komponenten (z. B. der Prozessor) eines Computers sind immer Umwelteinflüssen ausgesetzt, welche die Ausführung beeinflussen können. Geht man von vom Hersteller geprüften Komponenten aus, so dass man Produktionsfehler ausschließen kann, können zwei mögliche Arten von Fehlern auftreten [4]: Die erste Möglichkeit ist ein permanenter Fehler in einem Schaltkreis. Zur Behebung ist ein Austauschen der fehlerhaften Komponente notwendig. Die zweite Art von Fehlern ist ein transienter, also vorübergehender Fehler, bei dem ein Schaltkreis für einen begrenzten Zeitraum fehlerhafte Ergebnisse liefert. Unter anderem kann dies durch α-Partikel oder Neutronen ausgelöst werden, welche auch in kosmischen Strahlen vorkommen. Diese manifestieren sich als ein einzelnes oder mehrere umgeschaltete Bits (engl. *Bit Flip(s)*) in Speicherzellen oder CPU-Registern. Einer Studie aus 1986 zufolge ist die Mehrheit ($85\,\%$) der auftretenden Fehler dieser Art [14]. Daher konzentriert sich diese Arbeit auf transiente Fehler.

Die Halbleiterelemente in Prozessoren wurden in den vergangenen Jahren immer weiter verkleinert, weil dieses eine Kostensenkung und höhere Energieeffizienz ermöglicht. Jedoch erhöht sich damit gleichzeitig die Anfälligkeit gegenüber Außeneinwirkungen [3]. Denn kleinere Halbleiterschaltungen können weniger Gesamtladung halten, so dass eine unbeabsichtigte Schaltung durch einen Treffer von α-Partikel oder Neutronen wahrscheinlicher wird. Durch die kleineren Elemente wird eine einzelne Schaltung seltener getroffen, doch da die Dichte der Halbleiterschaltungen erhöht wird (die Fläche bleibt gleich), steigert sich die Gesamtwahrscheinlichkeit, dass ein Bauteil einen Fehler erfährt. Da traditionellere Methoden wie Redundanz der Komponenten durch höhere Kosten bzw. vermehrten Energiebedarf teils untragbar werden, wird stattdessen die Software diesen Fehlern direkt ausgesetzt [19]. Daher muss beim Design von Software, welche eine Häufung von Auftreten von Fehlern durch die Einsatzumgebung bedingt erwartet, das Vorkommen von transienten Fehlern beachtet werden. Ein möglicher Fall dafür sind Kleinstsatelliten, z. B. der OPS-SAT der European Space Agency (ESA) [7]. Diese werden typischerweise mit Hardwarebauteilen ausgestattet, die jeder normale Nutzer kaufen kann (engl. *commercial off-the-shelf (COFS)*) und daher nicht speziell strahlen-gehärtet sind. Im Falle des OPS-SAT ist der Prozessor ein ARM Cortex-A9. Entsprechend sollten Vorkehrungen getroffen werden, damit die Auswirkungen von Fehler begrenzt werden.

```
1  #include <stdio.h>
2
3  int myAbs (int value)
4  {
5      if (value < 0)
6          return -value;
7      else
8          return value;
9  }
10
11 void main()
12 {
13     int x = -5;
14     printf("%d %d\n", x, myAbs(x));
15 }
```

Abbildung 1.1: Ein einfaches Beispielprogramm in C, welches den Wert und den Betrag von x ausgibt.

1.1 Motivation

Eine Möglichkeit zur Begrenzung der Fehlerauswirkungen sind Compileroptimierungen. Sie verkürzen typischerweise die Laufzeit eines Programms (andere mögliche Ziele sind z. B. Binarygröße oder Energieverbrauch minimieren). Durch eine kürzere Laufzeit verringert sich die Wahrscheinlichkeit, dass ein Programm einen Fehler sieht. Denn auch wenn die Häufigkeit, dass ein auftretender Fehler zu einer fehlerhaften Ausgabe führt, gleich bleibt, so verringert sich die Anzahl der auftretenden Fehler und damit die fehlerhaften Ausgaben. Jedoch wird bei der Optimierung die zugrunde liegende Codestruktur geändert, z. B. werden redundante Speicherzugriffe entfernt [5]. Dadurch wird das Programm anfälliger gegenüber Fehlern [6]. Daher sind Optimierungen im Kontext dieser Arbeit eine Abwägung von höherer Performance und verringerter Fehlertoleranz.

Ein einfaches Beispiel für die Auswirkungen von Optimierungen wird hier vorgestellt. Dafür wird das minimale Beispielprogramm in Abbildung 1.1 betrachtet, welches *x=-5* definiert und danach *x* als auch den Betrag von *x* ausgibt. Nach der Übersetzung ergibt sich daraus Assembler Code, welcher in Abbildung 1.2 zu sehen ist. Das Programm wurde einmal ohne Optimierungen (Abbildung 1.2a) und einmal mit dem Standardlevel `-O1` (Abbildung 1.2b) übersetzt. Es fällt direkt auf, dass der resultierende Code mit Optimierung deutlich kürzer ist. Der größte Unterschied liegt darin, dass in der optimierten Version die Funktion *myAbs* nicht mehr aufgerufen wird (in der unoptimierten Variante ist der Aufruf in Zeile 8). Dadurch läuft der Code schneller, außerdem gibt es auch weniger Operationen, welche fehlschlagen können. Weder

```
0000000000000663 <main>:
 663:   55                      push   %rbp
 664:   48 89 e5                mov    %rsp,%rbp
 667:   48 83 ec 10             sub    $0x10,%rsp
 66b:   c7 45 fc fb ff ff ff    movl   $0xfffffffb,-0x4(%rbp)
 672:   8b 45 fc                mov    -0x4(%rbp),%eax
 675:   89 c7                   mov    %eax,%edi
 677:   e8 ce ff ff ff          callq  64a <myAbs>
 67c:   89 c2                   mov    %eax,%edx
 67e:   8b 45 fc                mov    -0x4(%rbp),%eax
 681:   89 c6                   mov    %eax,%esi
 683:   48 8d 3d 9a 00 00 00    lea    0x9a(%rip),%rdi
                # 724 <_IO_stdin_used+0x4>
 68a:   b8 00 00 00 00          mov    $0x0,%eax
 68f:   e8 8c fe ff ff          callq  520 <printf@plt>
 694:   90                      nop
 695:   c9                      leaveq
 696:   c3                      retq
 697:   66 0f 1f 84 00 00 00    nopw   0x0(%rax,%rax,1)
 69e:   00 00
```

(a) Assembler Code der `main` Methode ohne Optimierungen

```
0000000000000676 <main>:
 676:   48 83 ec 08             sub    $0x8,%rsp
 67a:   b9 05 00 00 00          mov    $0x5,%ecx
 67f:   ba fb ff ff ff          mov    $0xfffffffb,%edx
 684:   48 8d 35 99 00 00 00    lea    0x99(%rip),%rsi
                # 724 <_IO_stdin_used+0x4>
 68b:   bf 01 00 00 00          mov    $0x1,%edi
 690:   b8 00 00 00 00          mov    $0x0,%eax
 695:   e8 a6 fe ff ff          callq  540 <__printf_chk@plt>
 69a:   48 83 c4 08             add    $0x8,%rsp
 69e:   c3                      retq
 69f:   90                      nop
```

(b) Assembler Code der `main` Methode mit `-O1` Optimierungen

Abbildung 1.2: Das Beispielprogramm aus Abbildung 1.1 compiliert (mit `gcc` 7.3.0). In Abbildung 1.2a wurde es ohne Optimierungen übersetzt, in Abbildung 1.2b mit dem Standardlevel `-O1`. Der Assembler-Code wurde mit Hilfe von *objdump -d* extrahiert.

können die nötigen Operationen für einen Funktionsaufruf verfälscht werden, noch kann der Vergleich < 0 in der *myAbs* ein falsches Ergebnis liefern. In diesem Fall hat die Optimierung die Fehlertoleranz auf zweierlei Art (Laufzeit und Angriffsfläche) und damit im Gesamten verbessert.

Üblicherweise werden Optimierungen in Level wie O1, O2 und O3 zusammengefasst. Diese betrachten jedoch nur die Performance und nicht die Fehlertoleranz. Eine Menge von Optimierungen, welche die Fehlertoleranz maximiert, muss daher nicht mit einem dieser Level übereinstimmen. Dies zeigten bereits Narayanamurthy, Pattabiraman und Ripeanu in ihrer Arbeit [19]. Ihr Ergebnis war, dass Compileroptimierungen die Fehlertoleranz erhöhen können und diese sich in der Regel von den üblichen Levels unterscheiden. Diese Untersuchung war jedoch immer nur auf ein einzelnes Programm beschränkt. Eine Menge von Optimierungen, welche im Allgemeinen die Fehlertoleranz von Programmen erhöht, wurde nicht erarbeitet.

Jedoch in Anbetracht des starken Anstiegs von Starts von Kleinstsatelliten [10] wäre eine solche Menge ein guter Startpunkt bzw. eine einfache Erweiterung, um die Fehlertoleranz der darauf laufenden Software zu erhöhen. Wie zuvor erwähnt werden in diesen häufig COFS Hardware verbaut, die entsprechend nicht auf den Betrieb im Orbit spezialisiert ist. Daher besteht eine starke Anfälligkeit gegenüber der kosmischen Strahlung und den damit verbundenen transienten Fehlern. Es bedeutet aber auch, dass ganz gängige Open-Source-Compiler wie die GNU Compiler Collection (GCC) zum Einsatz kommen, und damit auch deren Optimierungen genutzt werden können. Für eine konkrete Einzelfalloptimierung sind bei solchen Projekten möglicherweise die Zeit bzw. die Mittel nicht ausreichend. Statt einer unoptimierten bzw. mit Standardlevel compilierte Version wäre hier eine im Allgemeinen für Fehlertoleranz optimierte Version vorzuziehen.

1.2 Zielsetzung

Ziel der Arbeit ist die Erarbeitung einer Menge an Optimierungen, welche im Allgemeinen die Fehlertoleranz erhöht. Dafür muss ein Algorithmus entworfen werden, welcher verschiedene Mengen an Optimierungen auf ihre Fehlertoleranz überprüft und danach optimiert. Da ein möglichst allgemeiner Satz von Optimierungen gefunden werden soll, muss diese Toleranz anhand vieler verschiedener Benchmarks gemessen werden. Bei einer zu geringen Menge an genutzten Programmen wäre das Resultat unter dem Gesichtspunkt der Allgemeingültigkeit ohne nennenswerte Erkenntnisse.

1.3 Aufbau der Arbeit

Zunächst werden im folgenden Kapitel verwandte Arbeiten vorgestellt. Dabei wird zuerst die bereits erwähnte Arbeit von Narayanamurthy, Pattabiraman und Ripeanu [19]

diskutiert. Danach werden weitere relevante Arbeiten vorgestellt, welche ähnliche Probleme oder Teilprobleme dieser Arbeit gelöst haben. Im Anschluss werden in Kapitel 3 benötigte Grundlagen für das Verständnis der restlichen Arbeit erarbeitet. Diskutiert werden Fehler, Fehlermetriken, das genutzte Fehlerinjektionstool und Compiler sowie deren Optimierungen. In Kapitel 4 wird das vorliegende Problem genau analysiert und daraus Anforderungen für den folgenden Entwurf gewonnen. Dieser wird in Kapitel 5 vorgestellt. Dort wird zudem erläutert, wie die einzelnen Anforderungen erfüllt werden. Im folgenden Kapitel 6 werden konkrete Implementierungsdetails besprochen. Danach wird der gesamte Ansatz und die Implementierung in Kapitel 7 evaluiert sowie werden die Ergebnisse präsentiert. Im letzten Kapitel wird eine Zusammenfassung gegeben und ein Ausblick auf mögliche fortführende Arbeiten.

2 Verwandte Arbeiten

Im folgenden wird ein Überblick gegeben über Arbeiten, welche ähnliche Probleme gelöst haben oder ein Teilproblem dieser Ausarbeitung behandeln. Zuerst wird die relevanteste Arbeit von Narayanamurthy, Pattabiraman und Ripeanu vorgestellt. Danach werden weitere Abhandlungen im Bereich der Untersuchungen zur Fehlertoleranz und Fehlerdetektion betrachtet. Anschließend werden noch weitere relevante Arbeiten vorgestellt und zum Abschluss eine kurze Zusammenfassung dieses Kapitels gegeben.

2.1 Finding Resilience-Friendly Compiler Optimizations Using Meta-Heuristic Search techniques

Die Arbeit von Narayanamurthy, Pattabiraman und Ripeanu [19] ist dieser Ausarbeitung am nächsten. Sie haben untersucht, ob es bessere Konfigurationen von Optimierungen gibt als die Standardlevel bzgl. der Fehlertoleranz. Dafür haben sie zwölf verschiedene Benchmarks, fünf aus der PARSEC und sieben aus der Parboil Benchmark Suite, genutzt. Sie haben den LLVM-Compiler untersucht und als Fehlerinjektionstool LLFI verwendet. Im Gegensatz zur GCC erzeugt der LLVM-Compiler aus C- bzw. C++-Code nur eine Zwischenrepräsentation. Der untersuchte Compiler bot etwa 50 verschiedene Optimierungen. Das Fehlerinjektionstool LLFI arbeitet direkt auf der erzeugten Zwischenebene, so dass nicht von echter x86-Register-Fehlerinjektion gesprochen werden kann. Nach Aussage der Autoren der Originalarbeit zu LLFI [16] ist dies jedoch gleichwertig: „In recent work, we have found that fault injections performed at the LLVM compiler's intermediate code level is accurate compared to assembly-level fault injections“ [16]. Dabei wird als Beleg auf eine einzelne frühere Veröffentlichung verwiesen [32]. Unabhängig davon, wie genau diese Art der Fehlerinjektion ist, sind die gefundenen Resultate dennoch interessant.

Zunächst soll aber die Vorgehensweise der Autoren vorgestellt werden. Ein genetischer Algorithmus (GA) wurde zur Suche einer optimalen Lösung genutzt. Sie haben zunächst einige kleinere Studien angestellt, um die Parameter des GA möglichst gut zu wählen. Genutzt wurden dafür nur zwei Benchmarks, und dabei stand im Vordergrund, dass der Algorithmus möglichst schnell konvergiert. Ob diese Wahl, gerade in Anbetracht der später angelegten größeren Studien, immer auch zu den besten Ergebnissen führt, ist nicht garantiert. Interessant ist aber dennoch, dass auch geprüft wurde, ob

bestimmte Arten von Optimierungen (Datenfluss-, Schleifen-, globale Optimierungen und andere) immer einen positiven bzw. negativen Einfluss haben. Dabei zeigte sich, dass eine solche Einteilung nicht möglich ist und alle Schalter betrachtet werden müssen. Dies ist für diese Arbeit ein guter Hinweis keine Optimierungen ohne Grund auszuschließen.

Weiter wurden auch zwei Fehlermetriken untersucht, einerseits $Resilience$ und andererseits $Vulnerability$. Diese werden genauer in Abschnitt 3.2 betrachtet. Zunächst reicht es zu wissen, dass bei der $Vulnerabililty$ auch die Laufzeit eines Programms in die Metrik einfließt. Dies könnte einen Teil der Resultate der Arbeit von Narayanamurthy, Pattabiraman und Ripeanu erklären, dazu später mehr. Die Autoren haben bei einer Untersuchung festgestellt, dass die $Resilience$ Metrik ohne größere Nachteile schneller konvergiert und sich daher darauf konzentriert.

Generell sei gesagt, dass der Optimierungsprozess immer nur für einen einzelnen Benchmark durchgeführt wurde. Es wurden zwar insgesamt 10 Benchmarks untersucht, aber jeder unabhängig von einander. Damit wurde gezeigt, dass der Ansatz funktioniert, um ein einzelnes Programm auf Fehlertoleranz zu optimieren. Für die $Resilience$ haben sie gezeigt, dass sie diese von der unoptimierten Version mit $76{,}14\,\%$ auf $79{,}125\,\%$ im Durchschnitt mit ihrer Lösung steigern konnten (größer ist besser). Die mit Standardlevel übersetzte Versionen waren allesamt schlechter als die unoptimierte bzgl. der $Resilience$. Bei der Evaluation der $Vulnerability$ wurden dagegen zwei Ausreißer gefunden – in anderen Worten $20\,\%$ der untersuchten Benchmarks: Für die beiden Benchmarks war es besser bzgl. dieser Metrik, wenn sie mit `-O3` compiliert wurden statt mit der gefundenen Lösung. Dies wird von den Autoren durch die starke Reduzierung der Laufzeit (etwa $40\,\%$) und damit kleiner werdenden Metrik erklärt. Daher stellt sich doch die Frage, ob die Optimierung nach der $Resilience$ Metrik wirklich die beste Wahl war. Der GA hätte immerhin auch die Möglichkeit gehabt, die Menge der `-O3` Optimierungen auszuwählen. Für alle anderen Fälle wurden jedoch bessere Konfigurationen gefunden. Die $Vulnerability$ der unoptimierten Version ist im Durchschnitt $9{,}25$ (Standardlevel ähnlich), ihre Lösung erreicht dagegen $8{,}12$ (kleiner ist besser).

Für diese Arbeit sollen diese Erkenntnisse als Grundlage dienen. Es werden alle Optimierungen der GCC betrachtet werden und es werden keine Optimierungen grundlos nicht untersucht. Mit der GCC werden zwei Compiler (`gcc` und `g++`) untersucht, die echten Maschinencode erzeugen, aber auch deutlich mehr Optimierungen besitzen. Die von den Autoren genutzten Parameter und Verfahren für den GA sollen nur als Anhaltspunkt dienen, weil die Optimierung zur schnellsten Konvergenz unerwünschte Nebeneffekte mit sich bringen könnte. Es wird die EAFC Metrik, welche unterschiedlich zu den zwei betrachteten Metriken ist, genutzt. Die Metrik selbst sowie die Beweggründe und Vorteile werden in Abschnitt 3.2 genau dargelegt. Der größte Unterschied zur Arbeit von Narayanamurthy, Pattabiraman und Ripeanu liegt jedoch im Ziel dieser Ausarbeitung: statt Benchmarks einzeln zu optimieren soll die Gesamtmasse der genutzten Benchmarks gleichzeitig optimiert werden. Dies bringt auch einige Probleme

mit sich, die in Kapitel 4 diskutiert werden.

2.2 Weitere Arbeiten zur Fehlertoleranz/-detektion

Nachdem die Arbeit vorgestellt wurde, an der sich diese orientiert, sollen noch einige weitere relevante Artikel im Feld der Fehlertoleranz bzw. Fehlererkennung erwähnt werden.

In einer weiteren Arbeit, die zur Zeit dieser Ausarbeitung nur als Entwurf vorliegt, haben Narayanamurthy, Pattabiraman und Ripeanu ihre vorherigen Ergebnisse mit einem zweiten Verfahren, dem *Simulated Annealing* verglichen [18]. Dies ist eine andere Möglichkeit eine möglichst optimale Lösung für ein Problem zu finden. Dabei hat sich jedoch herausgestellt, dass der GA das bessere Mittel zur Optimierung des vorliegenden Problems ist. Es spricht entsprechend nicht dagegen, dass in dieser Arbeit ein angepasster GA verwendet wird.

Diese beiden Artikel waren aber nicht die ersten, die sich mit dem Effekt von Optimierungen auf die Zuverlässigkeit von Programmen beschäftigt haben. Im Jahr 2011 haben sich Demertzi, Annavaram und Hall [5] mit dem Thema auseinandergesetzt. Sie untersuchten, welche Auswirkungen die Standardlevel `-O1`, `-O2`, `-O3` des `gcc` (Version 3.6.7) haben. Dabei zeigte sich, dass durch die kürzere Laufzeit eine Verbesserung der Fehlertoleranz im Gesamten erreicht wird, auch wenn die Wahrscheinlichkeit, dass ein Fehler zu einer falschen Ausgabe führt, erhöht wird.

In 2014 haben Sangchoolie et al. [22] eine Studie zum Einfluss von einzelnen Bit Flips auf Programme, welche mit unterschiedlichen Standardlevels compiliert wurden, geführt. Dabei wurde einerseits die Fehlersensitivität (engl. *error sensitivity*) untersucht, welche die Wahrscheinlichkeit beschreibt, dass aus einem Fehler eine falsche Ausgabe folgt. Hier wurde festgestellt, dass die Sensitivität im schlimmsten Fall kaum erhöht wird („highest increase in the percentage of SDCs was only around 6 percentage points“ [22]) und in zwei Fällen sogar verringert (um vier und zehn Prozentpunkte). Die daraus gezogene Schlussfolgerung ist, dass Optimierungen in diesem Kontext positiv anzusehen sind. Andererseits wurde auch (in einer kleinen Studie) evaluiert, in wie fern Änderungen am Quellcode die Fehlersensitivität beeinflussen. Dabei wurde festgestellt, dass sie starke Variationen bei der Fehlersensitivität bewirken. Die Folgerung daraus ist – mit einem Vorbehalt auf Grund der kleinen Größe – dass eine geschickte Wahl der Daten- und Programmstrukturen die Sensitivität positiv beeinflussen kann.

Eine weitere Methode, um Hardwarefehler entgegenzutreten ist Diverses Compilieren. Dabei ist bereits redundante Hardware gegeben, aber um unterschiedliche Binaries zu erhalten wird das Programm mit verschiedenen Compilern und Optimierungen übersetzt. Höller et al. [13] haben dieses für diesen Fall bekannte Verfahren genutzt, um Fehler in der Software selbst zu finden. Fehler in der Software sind nicht auszuschließen, jedoch kann ein Compiler diese in gewisser Weise verschleiern. Nutzt man jedoch den Ansatz der Diversen Compilierung, dann können dadurch nach Aussage der Autoren

bis 70 % von Softwarefehler im Zusammenhang mit Speicher gefunden werden.

2.3 Andere relevante Arbeiten

Es hat sich herausgestellt, dass das vorliegende Problem dieser Arbeit als Optimierung mit Rauschen betrachtet werden kann. Es gibt eine Vielfalt an Möglichkeiten das Rauschen zu handhaben, wie die Studie von Rakshit, Konar und Das zeigt [21]. In dieser Arbeit soll speziell das Verfahren des dynamischen Resamplings angewendet werden. Dynamisches Resampling wurde bereits in verschiedenen Varianten entwickelt, wie die Vergleichsstudie von Siegmund, Ng und Deb aufzeigt [25]. Die Studie behandelt nur Verfahren für mehrkriterielle Optimierungen. Im Rahmen dieser Arbeit wird eine Abwandlung des in der Studie vorgestellte konfidenzbasierte Resampling genutzt, welches von Syberfeldt et al entworfen wurde [31]. Es diente als Leitfaden für das später entworfene Resamplingverfahren, welches nur ein einzelnes, aber komplexes Kriterium betrachtet (vgl. Abschnitt 5.4).

Eine automatische Anpassung des `gcc` Compilers haben auch Blackmore, Ray und Eder [2] vorgenommen. Deren Ziel war es, dass eine bessere Zusammenstellung von Optimierungen (für die Performance) als `-O3` gefunden wird für Programme, welche auf eingebetteten Systemen laufen. Dieses wurde erfolgreich für die Prozessoren ARM Cortex-M3, ARM Cortex-A8 und 8 Bit AVR über 84 Benchmarks erreicht. Es wurden `-O3` noch weitere Optimierungen hinzugefügt, die abhängig von dem jeweiligen Prozessor sind. Diese können nach Aussage der Autoren als Alternative zur Compilierung mit `-O3` dienen. Das Ergebnis dieser Ausarbeitung soll genauso eine Zusammenstellung von Optimierungen sein, welche man beim Compilieren nutzen kann. Doch im Gegensatz zur Arbeit von Blackmore, Ray und Eder ist hier das Ziel eine möglichst hohe Fehlertoleranz zu erreichen.

2.4 Zusammenfassung

Zuerst wurde ausführlich die Arbeit von Narayanamurthy, Pattabiraman und Ripeanu diskutiert, welche als Anhaltspunkt für diese Ausarbeitung dient. Dabei wurden die Ergebnisse erläutert und welche davon als Grundlage für diese Arbeit dienen. Anschließend wurden weitere Arbeiten im Feld der Untersuchungen zur Verbesserung der Fehlertoleranz und Fehlerdetektion vorgestellt. Als letztes wurden weitere relevante Arbeiten beschrieben. Dabei wurde Optimierung mit Rauschen und die automatische Selektion von Compileroptimierungen zur Performancesteigerung vorgestellt.

Viele der bereits erwähnten und nur angerissenen Konzepte werden in dem folgenden Kapitel erläutert. Diese bilden auch die Grundlage für diese Ausarbeitung.

3 Grundlagen

Zuerst werden einige Grundlagen und -begriffe erläutert werden. In Abschnitt 3.1 werden das Fehlermodell, die genaue Fehlerdefinition und Unterscheidung zwischen deren Zuständen, sowie das Prinzip des Fehlerraums vorgestellt. Im darauf folgenden Abschnitt 3.2 werden verschiedene Metriken diskutiert und bestimmt, welche in dieser Arbeit genutzt wird. Danach wird FAIL*, welches zur Fehlerinjektion genutzt wird, in Abschnitt 3.3 vorgestellt. Im Anschluss werden Grundzüge eines Genetischen Algorithmus in Abschnitt 3.4 erklärt. Als letztes wird eine Zusammenfassung über die kennengelernten Grundlagen gegeben.

3.1 Fehler

Zunächst sollen die Begrifflichkeiten um Fehler definiert werden. Da sich diese Arbeit mit Fehler und Fehlertoleranz beschäftigt, sollten die Grundlagen dafür eindeutig sein.

Als Fehlermodell für diese Arbeit werden allgemeine CPU-Fehler genommen. Diese können durch auftreffende Teilchen oder kosmische Strahlung hervorgerufen werden. Die Fehler treten dabei an verschiedenen Stellen in der CPU auf, und zwar z. B. in Registern, Instruktionscache, Datencache, Arithmetisch-logische Einheit (ALU) oder der Pipeline [19]. Typischerweise ist Cache und Speicher Fehler detektierend, einzelne Bit-Fehler werden erkannt und behoben. Ähnliches gilt für die Instruktionen [19]. Fehler in der ALU, Registern, Pipeline, Logikgatter etc. werden dagegen nicht erkannt und können sich auf das Programm auswirken [19].

Interessant für diese Arbeit sind nur Fehler, welche nicht durch Hardware erkannt bzw. abgedeckt werden. Abgebildet können diese Art von Fehlern, in dem Bit Flips direkt in die CPU-Register injiziert werden.

In Abbildung 3.1 sind die möglichen Zustände und Verläufe für einen auftretenden Fehler aufgezeigt. Ein sich realisierender Fehler (Bit Flip) in einem Register wird zunächst Fault genannt. Falls das Register nicht mehr ausgelesen bzw. überschrieben wird, dann handelt es sich dabei um einen Dormant fault. Der Fehler hat keinerlei Auswirkung auf die Ausführung des Programms. Wird das Register ausgelesen, so verwandelt sich der Fault in einen Error. Das Programm muss aber nicht weiter von dem Error beeinflusst werden, sondern es kann auch maskiert bzw. toleriert werden. Der Error kann sich aber auch auf das Ergebnis auswirken, in dem Falle wird es ein Failure.

Die Abbildung kann so interpretiert werden für diese Arbeit wie folgt interpretiert

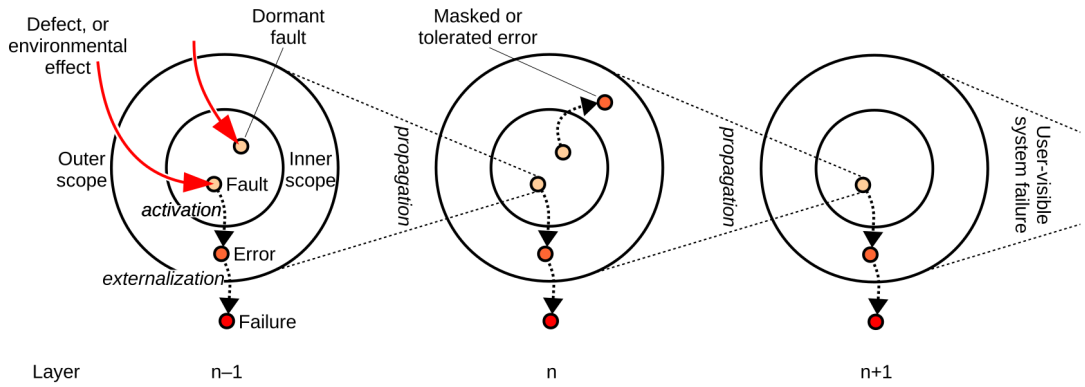

Abbildung 3.1: Verlauf eines Fehlers in einem System. Grafik entnommen aus der Arbeit von H. Schirmeier [24].

werden: Ein Programm hat typischerweise mehrere Schichten, in der jeder Funktionsaufruf als eine neue, tiefere Schicht angesehen werden kann. Sollte ein Fehler in einer tieferen Schicht auftreten und dieser sich zu einem Failure entwickeln, so ist dies für die darüber liegende Schicht ein Fault. Sollte der Fehler sich bis zu der Nutzerschicht propagieren, so kann dieser sich z. B. als Silent Data Corruption (SDC) oder eine deutlich längeren Laufzeit bis hin zur unendlichen Schleife realisieren. Das Programm wurde vollständig ausgeführt, jedoch wurde der Fehler nicht erkannt und hat das Ergebnis verfälscht.

Ein weiterer wichtiger Aspekt ist der Fehlerraum. Dargestellt ist dieser konzeptionell in Abbildung 3.2. Er bestimmt, wie viele Möglichkeiten bzw. Punkte es gibt, an denen ein Fehler in ein Programm injiziert werden kann. Er wird aufgespannt durch den Speicher, in den injiziert werden soll, und die Laufzeit, gemessen in CPU-Zyklen. Weil im Rahmen dieser Arbeit in CPU-Register injiziert wird, ist die Größe der Y-Achse durch die Anzahl und Größe der Register beschränkt, und fix für alle Programme. Dagegen ist die Laufzeit offensichtlich von dem konkreten Programm abhängig.

3.2 Fehlermetrik

Die Fehlermetrik ist ein weiterer essentieller Teil, denn sie wird zum Vergleich zwischen verschiedenen Zwischenergebnissen genutzt. Daher sollen zuerst die bereits in Abschnitt 2.1 erwähnten Metriken und von Narayanamurthy, Pattabiraman und Ripeanu genutzten Metriken $Resilience$ und $Vulnerability$ [19] genau erklärt und diskutiert werden. Da diese jedoch Probleme mit sich bringen, wird im Anschluss die in dieser Arbeit genutzte EAFC-Metrik vorgestellt und deren Vorzug gegenüber den beiden anderen erläutert.

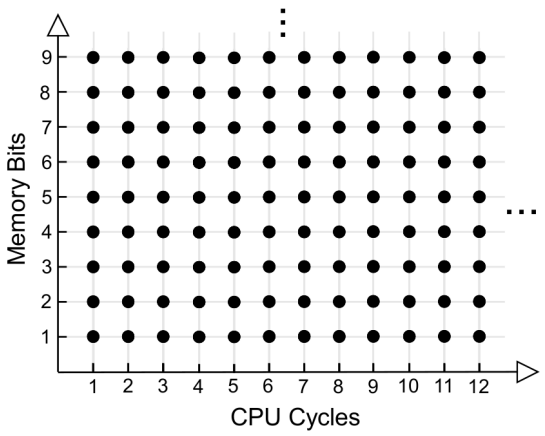

Abbildung 3.2: Visualisierung des Fehlerraums. Die Gesamtgröße ist abhängig von der Größe des Speichers, in den injiziert wird (y-Achse, in Memory Bits), sowie die Laufzeit (x-Achse, in CPU Cycles). An jeden eingezeichneten Punkt im Koordinatensystem kann ein Fehler injiziert werden. Grafik entnommen aus der Arbeit von H. Schirmeier [24].

3.2.1 Resilience

Die erste Metrik ist

$$Resilience = (1 - SDCrate). \tag{3.1}$$

Die *SDCrate* ist dabei die Anzahl von beobachteten SDCs F gegenüber den insgesamt injizierten Fehlern N (gleichbedeutend mit Failure F und Fault N).

$$SDCrate = \frac{F}{N} \tag{3.2}$$

Diese Metrik hat den Nachteil, dass die Laufzeit unbeachtet bleibt. Dies kann je nach realem Anwendungsfall von großer Bedeutung sein.

Angenommen, dass für eine bestimmte Berechnung ein Zeitfenster von $1\,\mathrm{s}$ zur Verfügung steht. Nach der Berechnung läuft das System leer, bis das nächste Zeitfenster erreicht wird. Weiter sei die Wahrscheinlichkeit für einen real auftretenden Fault in der Hardware pro Zeiteinheit t konstant mit $N/t = 1/1.000\,\mathrm{s}$. Angenommen wir haben zwei Programme A und B, welche die gewünschte Berechnung durchführen. Die Laufzeit von A sei dabei $0{,}5\,\mathrm{s}$ und von B $1\,\mathrm{s}$, die *SDCrate* bei beiden $0{,}1$. Es ist offensichtlich, dass ein Fehler, der im System von A auftritt, nur eine $50\,\%$-ige Wahrscheinlichkeit hat, zur Laufzeit von A aufzutreten (die anderen $50\,\%$ tritt der Fehler während des Leerlaufs auf). Dagegen tritt im System von B der Fehler immer zur Laufzeit von B auf. Betrachtet man nun die Mean Time To Failure (MTTF) [17] der beiden System, so stellt sich heraus, dass sie von Programm A doppelt so groß ist wie die von B. Die $Resilience$, gemessen an der reinen Zeit, die ein Programm rechnet, ist jedoch

identisch für beide Programme, der Unterschied für die reale Anwendung spiegelt sich nicht wider.

3.2.2 Vulnerability

Die zweite Metrik ist $Vulnerability = (SDCrate \cdot Executiontime)$. Sie behebt das Problem der nicht beachteten Laufzeit. Jedoch lässt sie auch einen Aspekt des Fehlerraums außen vor: den Speicherbedarf.

Hier lässt sich ebenso wie zuvor ein Beispiel konstruieren, in der alle Parameter (*SDCrate*, N/t sowie Laufzeit) von zwei Programmen A und B identisch sind, jedoch der Speicherbedarf bei einem halbiert ist. Genau wie zuvor würde das Programm mit halben Speicherbedarf nur halb so oft von realen SDCs betroffen sein.

Außerdem birgt die Metrik (wie auch *Resilience*) das Potenzial, geschönt zu werden. Erhöht man den von einem Programm reservierten Speicher ohne ihn zu nutzen, so würden in dem Bereich trotzdem Fehler injiziert werden (die Laufzeit ist davon offensichtlich unberührt). Die dort injizierten Fehler können aber nie zu einem SDC führen. Entsprechend würde um den gleichen Faktor die *SDCrate* sinken und bei konstanter Laufzeit sinkt damit die *Vulnerability*.

Im Rahmen dieser Arbeit ist die Menge der Register, in die injiziert wird, konstant und damit auch der Speicherbedarf. Daher wäre es – rein vom mathematischen Standpunkt aus – für diese Arbeit genauso gut, die *Vulnerability* zu verwenden. Jedoch wurde sie von Narayanamurthy, Pattabiraman und Ripeanu nur intuitiv motiviert, jedoch nie theoretisch hergeleitet oder in Bezug zu vorhandenen Metriken gesetzt.

3.2.3 EAFC

In dieser Arbeit wird daher die Extrapolated Absolute Failure Count (EAFC)-Metrik [23] verwendet. Sie ist wie folgt definiert:

$$EAFC = \omega \cdot \frac{F}{N} = \omega \cdot SDCrate \tag{3.3}$$

Dabei ist $\omega = Executiontime \cdot Memory$ die Größe des Fehlerraums. Zur Erinnerung: Die Laufzeit ist gemessen in CPU-Zyklen, der Speicher fix gegeben durch die CPU-Register und deren Breite. Es wird daher nicht nur die Laufzeit sondern auch der genutzte Speicher berücksichtigt. Die problematischen Teile der $Resilience$ und $Vulnerability$ Metriken sind damit behoben. Deshalb wird diese Metrik für diese Ausarbeitung genutzt.

Bei der Extrapolated Absolute Failure Count (EAFC)-Metrik wird die Anzahl der Fehler extrapoliert, die aufgetreten wären, wenn man an jeden möglichen Punkt im Fehlerraum injiziert hätte. Entsprechend große Werte sind bei dieser Metrik zu erwarten.

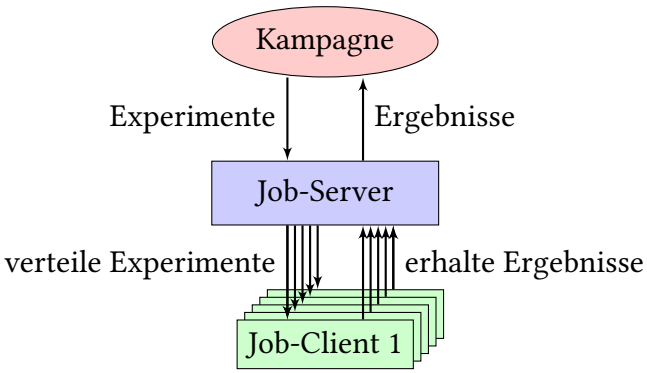

Abbildung 3.3: Grundlegendes Schema wie FAIL* aufgebaut ist. Die Kampagne liefert Experimente an den Job-Server. Dieser verteilt die Experimente auf die zur Verfügung stehenden Clients. Diese liefern nach Durchführung des Experiments die Ergebnisse zurück an den Job-Server, welcher sie weiter zur Kampagne leitet.

3.3 FAIL*

Fault Injection Leveraged (FAIL*) [24] ist das Fehlerinjektionssystem, welches im Rahmen dieser Arbeit genutzt wird. Es erlaubt die Fehlerinjektion auf x86-Ebene, dabei sowohl in Speicher als auch CPU-Register. Zum besseren Verständnis der weitergehenden Arbeit werden hier die Begrifflichkeiten erklärt, die im Rahmen von Fault Injection Leveraged (FAIL*) genutzt werden.

Der Golden Run ist der initiale Durchlauf für ein Programm. Dabei wird das Programm im Simulator ausgeführt und die Länge (Anzahl der Instruktionen bzw. CPU-Zyklen) sowie genutzter Hauptspeicherbereich ermittelt. Außerdem wird die Ausgabe mitgeschrieben, welche später als Vergleich dient.

Ein Experiment führt das Programm auch im Simulator aus, injiziert jedoch an einer zuvor festgelegten Stelle einen Fehler. Dafür läuft der Simulator bis zu der entsprechenden Instruktion, hält das Programm an und manipuliert das vorgegeben Register. Danach wird die Durchführung fortgesetzt und das Verhalten beobachtet. Es kann zu fünf verschiedenen Resultaten führen:

- OK_MARKER: das Programm ist fehlerfrei durchgelaufen
- DETECTED_MARKER: das Programm hat selbst einen Fehler erkannt
- SDC: das Programm ist durchgelaufen, hat jedoch eine falsche Ausgabe gemacht
- TRAP: das Programm hat ein Trap ausgelöst
- TIMEOUT: das Programm ist deutlich länger gelaufen, sodass es vorzeitig beendet wurde

Eine Kampagne definiert, welche Experimente durchgeführt werden sollen. Das umfasst die Wahl des Programms, die Anzahl der Experimente und die Stellen, an denen ein Experiment einen Fehler injizieren soll. Wie in Abbildung 3.3 zu sehen ist, liefert die Kampagne Experimente bzw. Experiment-Anfragen an einen Job-Server. Der Server verteilt diese Anfragen an Clients, welche die Experimente durchführen. Nach der Durchführung liefern diese das Ergebnis zurück an den Job-Server, welcher sie an die Kampagne weiterleitet. Dies ermöglicht eine hohe Parallelität bei der Durchführung von Experimenten.

3.4 Genetische Algorithmen

Wie bereits in Abschnitt 2.1 erwähnt, wird in dieser Arbeit auch ein genetischer Algorithmus (GA) verwendet. Narayanamurthy, Pattabiraman und Ripeanu haben in ihrer Arbeit [19] bereits gezeigt, dass damit erfolgreich Ergebnisse im Bereich der Optimierung der Fehlertoleranz erzielt werden können.

Ein GA ist an die natürliche Evolution angelehnt [26]. Entsprechend gibt es die Konzepte von *Generationen*, welche aus einer Menge von *Individuen* bestehen, so wie *Selektion*, *Mutation* und *Rekombination*. Diese müssen an das spezifische Problem angepasst werden.

Ziel eines GA ist die Optimierung einer Funktion. Abhängig von dem gegebenen Problem ist eine Minimierung oder Maximierung der Funktion gewünscht. Die zu optimierende Zielfunktion ist dabei oft hoch dimensional und eine algebraische Lösung unmöglich. Ist nicht viel über das zugrunde liegende Problem bekannt, ist ein GA eine gute Wahl, weil er ein sehr generisches Konzept hat.

Eine schematische Grafik eines GA ist abgebildet in Abbildung 3.4 (die Reihenfolge von Selektion, Mutation und Rekombination können sich je nach konkreter Umsetzung unterscheiden). Eine Generation besteht dabei aus n Individuen. Jedes Individuum ist eine mögliche Lösung für das gegebene Problem, für die sich die Fitness (Zielfunktionswert des Individuums) berechnen lässt. Die Lösung wird durch das Genom codiert, bestehend aus einzelnen Genen. Die genaue Umsetzung des Genoms ist abhängig vom Problem.

Nachdem für alle Individuen einer Generation die Fitness berechnet wurde, wird der Prozess der Selektion, Mutation und Rekombination gestartet. Für alle drei Schritte gibt es verschiedenste Möglichkeiten zur konkreten Durchführung, daher werden sie hier nur konzeptionell vorgestellt. Dabei muss gesagt sein, dass abhängig von der Reihenfolge die Ziele der Einzelschritte leicht variieren können. Darauf wird hier nicht näher eingegangen, es wird von der in Abbildung 3.4 gezeigten Reihenfolge ausgegangen und darauf beschränkt.

Die Selektion dient zur Auswahl der Individuen, welche für die Mutation und Rekombination genutzt werden soll. Dabei können Verfahren verwendet werden, welche das gleiche Individuum mehrfach selektieren. Dies ist auch notwendig, wenn der Ablauf

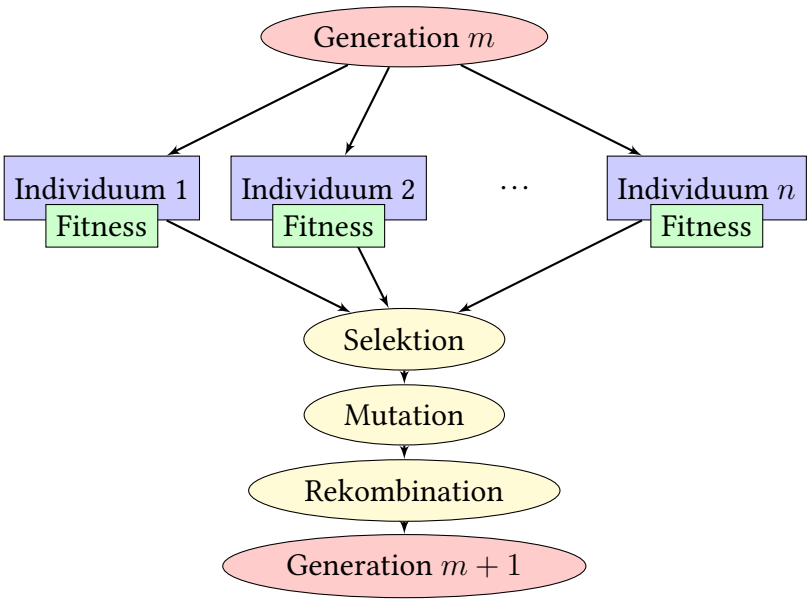

Abbildung 3.4: Schematische Abbildung des Ablaufs eines GA, wie aus einer Generation n eine neue Generation $n+1$ gewonnen wird.

wie hier angenommen ist, denn ansonsten müsste keine Selektion durchgeführt werden, es würden einfach die bereits bestehenden n Individuen übernommen. Eine Mehrfachwahl ermöglicht auch, dass bereits gute Individuen sich durch die folgende Mutation und Rekombination in bessere Individuen entwickeln, aber dabei in unterschiedliche Richtungen im Raum aller möglichen Individuen.

Der nächste Schritt ist die Mutation. Die Hintergedanke dabei ist, dass durch eine leichte Änderung eventuell ein besseres Individuum entsteht. Dies ist auch der Schritt, durch den Gene, welche bisher immer deaktiviert waren, aktiviert werden können. In dieser Phase wird ein Individuum mit der Wahrscheinlichkeit p_{mut} mutiert, und das für jedes gegebene Individuum. Wie eine Mutation im Detail umgesetzt wird, ist u. a. abhängig vom Problem und wird hier nicht näher erläutert.

Der letzte Schritt ist die Rekombination. An dieser Stelle wird versucht, durch Mischung der Gene bessere Individuen zu erreichen. Dafür müssen zunächst zwei Individuen ausgewählt werden, welche rekombiniert werden sollen. Dann werden nach einem zuvor festgelegten Verfahren Gene zwischen den beiden Individuen ausgetauscht. Für beide Schritte gibt es verschiedene Ansätze, diese zu erläutern liegt außerhalb des Rahmens dieser Arbeit.

Nach diesen drei Schritten hat der GA die Nachfolger ermittelt. Diese bilden die nächste Generation, mit der das gesamte Verfahren wiederholt werden. Diese Wiederholung wird so oft durchgeführt, bis eine zuvor definierte Abbruchbedingung erreicht ist.

```
1 main():
2     return sum(1,2)
3
4 sum(int a, int b):
5     return a+b
```

(a) Unoptimierter Code

```
1 main():
2     return 1+2
```

(b) Optimierter Code

Abbildung 3.5: Quellcode vor und nach Funktion-Inlining.

3.5 Compiler und Optimierungen

Im Rahmen dieser Arbeit wird die GNU Compiler Collection (GCC) (Version 7.3.0) betrachtet. Sie beinhaltet unter anderem die Compiler `gcc` und `g++`, welche respektive C- bzw. C++-Code kompilieren. Die Anzahl an einzelnen Optimierungsschaltern beträgt 210 (exkludiert der Standardoptimierungen wie `-O1`, `-O2`, `-O3`; außerdem gibt es noch viele weitere Schalter, unter anderem spezifische für C- bzw. C++-Code). Damit ergibt sich ein ebenso hoch dimensionaler Raum. Bereits unter der Annahme, dass es sich bei allen um binäre Optimierungsschalter handelt (was nicht ganz korrekt ist, einige sind parametrisiert und haben damit mehr als zwei Einstellungsmöglichkeiten), ergeben sich $2^{210} \approx 10^{63}$ verschiedene Möglichkeiten, alle Optimierungen zu kombinieren.

Optimierungen sind prinzipiell Transformationen, die ein bestimmtes Ziel verfolgen. Bei einem Großteil der verfügbaren Schalter handelt es sich um Laufzeitoptimierungen, jedoch gibt es auch andere Ziele (z.B. versucht `-Os` das entstehende Binary möglichst klein zu halten).

Dass Optimierungen den entstehenden Assembler-Code beeinflussen, haben wir bereits in Abschnitt 1.1 gesehen. Einige Optimierungen können auch einfach auf Quellcode-Ebene dargestellt werden. So zum Beispiel das *Inlining* von Funktionen. Ein einfaches Beispiel dafür ist in Abbildung 3.5 gezeigt. Der `gcc` hat dafür mehrere Optionen zur Optimierung [27]:

- `-finline-functions`
- `-finline-functions-called-once`
- `-finline-limit=n`
- `-finline-small-functions`
- `-findirect-inlining`
- `-fearly-inlining`

Die genauen Bedeutungen werden hier nicht weiter erläutert, aber es zeigt, dass Inlining ein wichtiges Konzept ist und soll daher als Beispiel dienen. Inlining heißt, dass

der Rumpf einer Funktion an die Stelle kopiert wird, wo die Funktion aufgerufen wird. Dieses ist auch in der Abbildung 3.5 zu sehen, wo die Addition der beiden Zahlen, welche eigentlich in der Funktion *sum* geschieht, direkt in die *main*-Funktion geschrieben wird. Der Vorteil ist, dass dadurch einerseits der Funktionsaufruf selbst gespart wird, andererseits können weitere Optimierungen durchgeführt werden. Zum Beispiel sind dadurch die Parameter bekannt und bedingte Anweisungen können bereits zur Übersetzungszeit ausgewertet werden. Der Nachteil vom Inlining ist, dass dadurch die Code Größe erhöht werden kann. Dies geschieht, wenn ein großer bzw. komplexer Rumpf an mehrere Stellen kopiert wird.

Eine genauere Analyse, welche Effekte Optimierungen im Bezug auf die Fehlertoleranz haben kann, wird in Abschnitt 4.1 vorgenommen.

3.6 Zusammenfassung

In diesem Kapitel wurden alle Grundlagen erläutert, die für das weitere Verständnis notwendig sind. Zuerst wurden Fehler genauer definiert. Dabei wird zwischen einem Fault, Error und Failure unterschieden. Außerdem wurde das Konzept des Fehlerraums erläutert. Die verschiedenen Fehlermetriken wurden diskutiert, zuerst die $Resilience$ und $Vulnerability$, welche von Narayanamurthy, Pattabiraman und Ripeanu genutzt wurden. Darauf wurde die EAFC-Metrik vorgestellt, welche in dieser Arbeit verwendet wird. Das Fehlerinjektionssystem FAIL* und die Begrifflichkeiten Experiment, Golden Run, Resultate und Kampagne wurden erklärt. Das Konzept eines GA wurde vorgestellt mit seinen Generationen, Individuen, Selektion, Mutation und Rekombination. Als letztes wurden die im Rahmen dieser Arbeit genutzten Compiler vorgestellt und gezeigt, dass Optimierungen als Transformationen angesehen werden können.

Im folgenden Kapitel wird das vorliegende Problem genau analysiert. Dabei wird auch eine Liste an Anforderungen an einen Entwurf erarbeitet.

4 Problemanalyse

Zur Zeit dieser Arbeit ist dem Autor keine (allgemeine) Menge von Optimierungen bekannt, die speziell zur Verbesserung der Fehlertoleranz dient. Das Ziel dieser Arbeit ist daher, genau eine solche Menge zu finden. Was es dabei zu beachten gibt, wird in diesem Abschnitt untersucht, und am Ende werden daraus Anforderungen an einen Designentwurf abgeleitet.

4.1 Einflüsse von Optimierungen auf Fehlertoleranz

Optimierungen wurden allgemein bereits in Abschnitt 3.5 erläutert. Im Folgenden sollen sie im Rahmen der Fehlertoleranz untersucht werden.

Optimierungen allgemein transformieren den Code, was Auswirkungen auf die Fehlertoleranz haben kann. Der große Teil der Optimierungen zielt auf eine bessere (kürzere) Laufzeit ab, was positiv ist, wenn die *SDCrate* gleich bleibt, da sich der Fehlerraum verringert. Da jedoch unbekannt ist, ob diese wirklich konstant bleibt, kann nicht einfach davon ausgegangen werden, dass alle Optimierungen sich positiv auf die Fehlertoleranz auswirken. Dies soll an zwei Beispielen gezeigt werden.

Ein gutes positives Beispiel sind die bereits in Abschnitt 3.5 vorgestellten Inline-Funktionen. Durch das Einfügen des Rumpfs einer aufgerufenen Funktion an die aufrufende Stelle (wie in Abbildung 3.5 dargestellt) wird die Fehlertoleranz erhöht. Grund dafür ist die einerseits verringerte Angriffsfläche – Parameter müssen nicht abgelegt und abgerufen werden, es gibt keinen Rücksprung – und andererseits die kürzere Laufzeit – kein Funktionsaufruf, stattdessen integrierter Teil der übergeordneten Funktion. Einziger Nachteil ist der erhöhte Speicherbedarf für das Binary, welcher diese Option in manchen Anwendungsgebieten (z. B. eingebettete Systeme) nicht immer möglich macht.

Eine Art von Schleifen-Optimierungen dagegen kann unerwünschte Effekte mit sich bringen. Ein Beispiel findet sich in Abbildung 4.1. Dabei handelt es sich um eine Schleifeninvariante (`t` ist immer 10). Vor der Optimierung wird in jedem Schleifendurchlauf die Variable `t` erstellt und 10 zugewiesen. Nach der Optimierung steht diese Definition vor der Schleife; in Hinblick auf die Laufzeit ist dies auch sinnvoll. Im Rahmen der Fehlertoleranz hat diese Optimierung jedoch einen negativen Seiteneffekt: wird im optimierten Fall zu einem beliebigen Zeitpunkt des Schleifendurchlaufs die Variable `t` korrumpiert, dann wirkt sich das auf alle folgenden Durchläufe aus, das Feld `array` ist für alle größeren Indizes mit falschen Werten gefüllt. Im unoptimierten Fall

```
int array[100];
for (int i=0; i<100; i++) {
  int t = 10;
  array[i] = i*t;
}
```

```
int array[100];
int t = 10;
for (int i=0; i<100; i++) {
  array[i] = i*t;
}
```

Abbildung 4.1: Codeausschnitt, welcher in Pseudocode die Auswirkung der Schleifen-Optimierung zeigt. Links ist das Beispiel vor der Optimierung, rechts danach.

dagegen kann eine Verfälschung von `t` maximal einen Eintrag in `array` beeinflussen. Abhängig vom konkreten Fall kann eine unoptimierte Version der schnelleren Variante im Rahmen von Fehlertoleranz vorgezogen werden.

4.2 Abhängigkeiten von Optimierungen beim `gcc`

Die Optimierungen des `gcc` sind nicht alle unabhängig voneinander, eine Teilmenge besitzen Abhängigkeiten. Dabei gibt es zwei unterschiedliche Typen: Einige der Optimierungen zeigen nur Wirkung, falls andere auch aktiv sind: „This option has any effect only when inlining itself is turned on by the -finline-functions or -finline-small-functions options“ (aus der Beschreibung von `-findirect-inlining`) [27]. Im Folgenden wird diese Art der Abhängigkeit *notwendige Optimierungen* genannt.

Andere Optimierungen schalten automatisch weitere ein. Die offensichtlichen Beispiele hierfür sind die Optimierungslevel `-O1`, `-O2` und so weiter. Jedoch auch einzelne Optimierungen können weitere aktivieren, so zum Beispiel `-ffast-math`: „Sets the options -fno-math-errno, -funsafe-math-optimizations, -ffinite-math-only, -fno-rounding-math, -fno-signaling-nans, -fcx-limited-range and -fexcess-precision=fast“ [27]. Im weiteren Verlauf werden diese Abhängigkeiten *implizierte Optimierungen* genannt.

Im Bezug auf die Fehlertoleranz kann durch diese Abhängigkeiten ein Anschalten einer einzelnen Optimierung zu verschiedenen Resultaten führen. Sollte die *notwendigen Optimierungen* nicht angeschaltet sein, so ergibt sich keine Änderung. Wird die letzte der *notwendigen Optimierungen* eingeschaltet, so kann dies stärkere Effekte nach sich ziehen, da effektiv mehrere Optimierungen auf einmal aktiv werden. Dagegen wenn eine Optimierung mit vielen *implizierten Optimierungen* genutzt wird und die *implizierten Optimierungen* zuvor ausgeschaltet waren, so kann das genauso starke Änderungen bewirken. Plötzlich wurden auch hier mehrere Optimierungen gleichzeitig aktiv. Es muss daher eine Umgangsform mit diesem Verhalten gewählt werden.

Optimierung	Mögliche Parameter
`-ffp-contract=`*style*	fast, off
`-finline-limit=`*n*	0...
`-fira-algorithm=`*alg*	priority, CB
`-fira-region=`*region*	all, mixed, one
`-flto-partition=`*alg*	1to1, balanced, max, none
`-flto-compression-level=`*n*	[0, 9]
`-freorder-blocks-algorithm=`*alg*	simple, stc
`-ftree-parallelize-loops=`*n*	0...

Tabelle 4.1: Optimierungen, die einen Parameter annehmen und nicht nur ein Ein-/Ausschalter Verhalten haben.

4.3 Parametrisierte Optimierungen beim `gcc`

Eine weitere Besonderheit ist, dass der `gcc` acht Optimierungen besitzt, die kein einfacher An/Aus-Schalter sind, sondern einen Parameter benötigen. Sie sind in Tabelle 4.1 gelistet. Welche Auswirkungen die verschiedenen Parameter auf die Fehlertoleranz haben ist nicht vorherzusagen. Entsprechend müssen alle verschiedenen Parameter für diese Optimierungen in Betracht gezogen werden. Zusätzlich ist anzumerken, dass nicht zwangsweise ein bestimmt gewählter Parameter dem Standardverhalten entspricht: „Note: there may be no value to -finline-limit that results in default behavior“ (aus der Beschreibung von `-finline-limit`) [27]. Daher muss auch auch ein Ausschalten eine mögliche Einstellung sein.

4.4 Bewertung der Einflüsse von Optimierungen auf Fehlertoleranz

Nachdem zuvor gezeigt wurde, dass Optimierungen Einfluss nehmen auf die Fehlertoleranz, stellt sich die Frage, wie man sie bewerten kann. Ein erster Ansatz wäre, dass alle Optimierungen einzeln händisch betrachtet werden, also eine rein qualitative von einem Domänenexperten durchgeführte Einordnung von Compileroptimierungen. Dafür könnte man sich die hervorgerufenen Transformationen anschauen und versuchen sie im Bezug auf Fehlertoleranz zu bewerten. Aber dies ist äußert zeitaufwendig (man müsste im Detail genau betrachten, was jede einzelne Optimierung verändert) und fehleranfällig (eventuell erfasst man nicht alle Transformationen einer Optimierung). Weiter können einige Optimierungen fallabhängig positiv oder negativ sein. Außerdem werden dabei die Abhängigkeiten ignoriert und daher kann man nicht vollständig sein, denn einige Effekte treten erst bei der Kombination von Optimierungen auf.

Ein besserer und praktikablerer Ansatz ist daher, eine empirische Untersuchung

durchzuführen. Diese muss entsprechend ausführlich und umfassend gestaltet werden, damit man allgemeingültige Aussagen treffen kann. Doch sie hat den Vorteil, dass die Probleme des händischen Ansatzes umgangen werden. Es ist kein Detailwissen über die Optimierungen notwendig und Kombinationen können auch getestet werden.

4.5 Anforderungen an den Entwurf

Aus den zuvor betrachteten Thematiken ergeben sich Anforderungen an einen Entwurf, der eine Lösung des Problems erarbeitet.

1. Aus der Erkenntnis, dass Optimierungen auch negative Einflüsse haben können (Abschnitt 4.1) folgt, dass alle Optimierungen betrachtet werden müssen. Es kann nicht von einer generellen Verbesserung durch eine Optimierung ausgegangen werden.

2. Der Entwurf muss behandeln, dass es Abhängigkeiten zwischen Optimierungen (Abschnitt 4.2) gibt.

3. Es müssen auch die parametrisierten Optimierungen (Abschnitt 4.3) modelliert werden.

4. Die Untersuchung muss ausführlich und umfassend gestaltet werden (Abschnitt 4.4). Im Konkreten bedeutet dies, es müssen genügend verschiedene Möglichkeiten von Optimierungskombinationen getestet werden sowie die Menge der Programme, auf welche die Optimierungen angewendet werden, groß genug sein.

5. Die Laufzeit muss trotz des Testens im Raum mit über 200 Dimensionen (mehr als 10^{63} Kombinationsmöglichkeiten, vgl. Abschnitt 3.5) in einem vertretbaren Rahmen liegen. Ein vollständiges Abtasten des Raumes ist unter keinen Umständen möglich.

5 Entwurf

Ziel des Entwurfs ist es, eine allgemein gültige Konfiguration von Compiler-Optimierungen zu finden, welche im Mittel die Fehlerresistenz von Programmen erhöht. Eingesetzt werden kann diese Konfiguration z. B. bei der Übersetzung einer Linuxdistribution und den Programmen für einen Kleinstsatelliten, ohne dabei jede Anwendung einzeln optimieren zu müssen.

Um dieses Ziel zu erreichen, werden verschiedene Komponenten benötigt. Als Schema dargestellt sind diese in Abbildung 5.1. Es wird ein Algorithmus zur Selektion und Optimierung von „guten“ Compileroptimierungen benötigt. Dieser wurde Dynamic Fault Injection Genetic Algorithm (DyFIGA) getauft und nutzt einen GA für die Optimierung. Weiter wird ein Fehlerinjektionssystem benötigt, dafür wird FAIL* genutzt. Außerdem werden ein Compiler und Benchmarks benötigt, hier wird die GCC verwendet und 17 verschiedene Benchmarks.

Zunächst werden die untersuchten Optimierungen sowie genutzen Benchmarks vorgestellt. Weiter wird die Definition des Genoms für den GA erläutert und es werden die konkreten Umsetzungen für GA spezifische Operationen erläutert. Im Anschluss wird das dynamische Resampling erklärt. Außerdem wird die Verbindung zum dem bereits in Abschnitt 3.3 vorgestellten FAIL* System zur Fehlerinjektion vorgestellt.

5.1 Betrachtete Optimierungen

Untersucht wurden alle Optimierungen der GCC Version 7.3.0, genauer gesagt alle, welche in der Anleitung [27] unter „Optimization Options“ gelistet sind. Einige Ausnahmen mussten dabei getroffen werden:

- `-fauto-profile*`, `-profile*` - diese benötigen ein Profiling des Binaries, was außerhalb des Rahmens dieser Arbeit liegt
- `-fwpa` - dieser Schalter ist nicht weiter dokumentiert, führte jedoch vielfach zu Compilerfehlern
- `-fno*` - all diese Schalter schalten explizit Optimierungen aus, was hier nicht notwendig und durch weglassen erreicht werden kann
- `-fexcess-precision` - hat zwei mögliche Parameter, wobei einer nur für C gilt

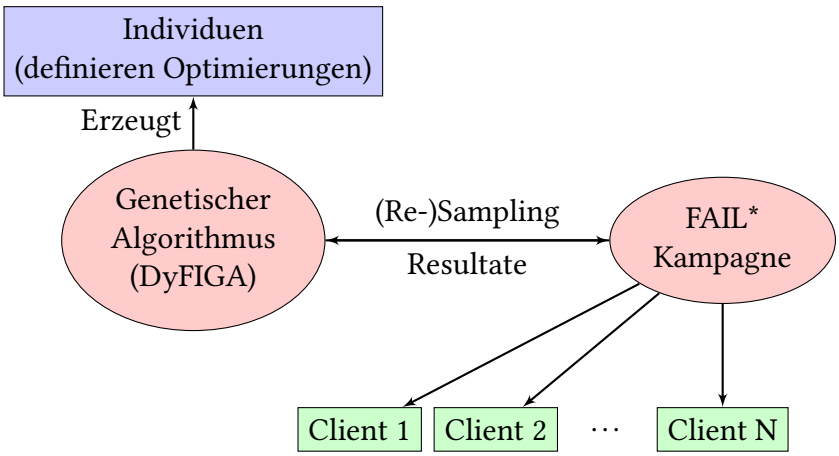

Abbildung 5.1: Schema des Systems, welches entworfen wurde.

- `-fgraphite*`, `-floop*` - diese waren mit dem getesteten Compiler nicht verfügbar (benötigt bestimmte Parameter bei der Compilierung des Compilers)

- `-ffast-math` - hat das Potential, bereits die Ausgabe des Golden Runs zu verändern bzw. Ausgaben maßgeblich zu beeinflussen, sollte separat betrachtet werden

Alle Optimierungen mit einem * sind so zu lesen, dass alle Optimierungen beginnend mit dem Präfix exkludiert wurden. Nach diesen Ausnahmen bleiben 177 verschiedene Optimierungen, die betrachtet wurden.

Außerdem sei hier erwähnt, dass die Abhängigkeiten zwischen den Optimierungen auf Grund der höheren Komplexität bei der Umsetzung nicht modelliert werden. Es wird eine Unabhängigkeit angenommen und es dem GA überlassen, trotzdem eine gute Menge von Optimierungen zu finden, wodurch die 2. Anforderung umgesetzt wird. Dies sollte vor allem dazu führen, dass der abzutastende Raum größer als notwendig ist. Wenn man die Abhängigkeiten abbilden wollen würde, so müssten diese zunächst aus dem `gcc`extrahiert werden. Dann könnte man z. B. die implizierten Optimierungen immer auch anschalten, um die Liste zu vervollständigen, und Optimierungen, bei denen notwendige Optimierungen fehlen und daher keinen Effekt haben, ausschalten.

Die Ausnahmen sind notwendiger Natur, alle weiteren Optimierungen werden betrachtet. Damit wird die 1. Anforderung erfüllt. Aus Komplexitätsgründen wird die Behandlung der Abhängigkeiten durch eine (bekannt unkorrekte) Unabhängigkeitsannahme realisiert, durch den der Suchraum nicht kleiner wird sondern die Suche ineffizienter gestaltet.

Benchmark	**Benchmark Suite**	Berechnung
blackscholes	PARSEC	Berechnet Preis von Optionen mit Partieller Differentialgleichung
fluidanimate	PARSEC	Hydrodynamik für Animationszwecke mit Smoothed Particle Hydrodynamics (SPH) Methode
freqmine	PARSEC	Frequent itemset mining
swaptions	PARSEC	Preis von einem Portfolio von Optionen auf einen Swap
bfs	Parboil	Implementiert Breitensuche, welche die kleinsten Kosten von einem Startknoten zu allen erreichbaren anderen Knoten berechnet
cutcp	Parboil	Berechnet die Kurzstreckenpotentiale in einem 3D Raum induziert durch Punktladungen
histo	Parboil	Berechnet ein 2D Histogramm mit einem Maximalwert von 255 pro Klasse
sad	Parboil	Berechnet die Summe der absoluten Distanzen für Blockpaare basierend auf dem MPEG-Algorithmus
spmv	Parboil	Implementiert die Berechnung des Produkts eines dichten Vektors mit einer dünnbesetzen Matrix
stencil	Parboil	Iterative Jacobi Stencil Berechnung auf einem 3D Gitter
basicmath	Mibench Automotive	Einfache mathematische Berechnungen, welche häufig keine Hardwareunterstützung auf Embedded Systems Prozessoren haben
bitcount	Mibench Automotive	Testet Bit-Manipulationsfunktionen eines Prozessors
qsort	Mibench Automotive	Sortierung eines String-Arrays mit Quicksort-Algorithmus
dijkstra	Mibench Network	Berechnet die Pfade auf einem Graph mit dem Dijkstra-Algorithmus
patricia	Mibench Network	Patricia ist eine spezielle Datenstruktur, häufig genutzt statt Bäumen, z. B. in Routingtabellen
blowfish	Mibench Security	Ver- und Entschlüsselung mit Blowfish, einem Algorithmus mit symmetrischer Blockchiffre und variabler Schlüssellänge
rijndael	Mibench Security	Ist die Implementierung von AES, einer Blockverschlüsselung mit 128-, 192- und 256-Bit Schlüsseln und Blöcken

Tabelle 5.1: Liste der Benchmarks für die Fehlerinjektion, deren Herkunft und welche Art von Berechnungen sie durchführen.

5.2 Benchmarks

Im Rahmen dieser Arbeit werden 17 Benchmarks als Basis für den Optimierungsprozess genutzt. Dabei wurden ähnlich zu der Arbeit von Narayanamurthy, Pattabiraman und Ripeanu [19] Benchmarks aus den beiden Suiten PARSEC (Version 3.0)[1] [1] und Parboil (Version 2.5)[2] [29] genommen. Um eine noch breitere Masse von Programmen abzudecken wurden zusätzlich Benchmarks aus der Mibench Suite[3] [12]. Eine vollständige Auflistung findet sich in Tabelle 5.1. Die Auswahl ist vor allem darauf begründet, ob eine Umsetzung funktioniert hat (hat problemlos das Binary compiliert) und dass die Übersetzungszeit und Laufzeit des Goldenen Runs in einem vertretbaren Rahmen waren (insgesamt maximal etwa eine Minute). Die hohe Zahl an Benchmarks dient der Erfüllung der 4. Anforderung, die Untersuchung ausführlich und umfassend zu gestalten.

5.3 Details des Genetischen Algorithmus

Zunächst wird die Codierung der Individuen des GA vorgestellt. Danach werden die genutzten Operationen für Selektion, Mutation und Rekombination erläutert. Allgemein dient der GA dazu, dass die Laufzeit der Untersuchung in einem vertretbaren Rahmen bleibt, wie es die 5. Anforderung verlangt. Außerdem können so eine hohe Anzahl von Kombinationsmöglichkeiten getestet werden, wie gefordert, die nicht nur rein zufällig gewählt sind.

5.3.1 Codierung der Individuen

Auf Grund der Optimierungen mit Parameter (vgl. Tabelle 4.1) ist eine binäre Codierung schwierig bzw. umständlich. Statt dessen wird eine ganzzahlige Repräsentation gewählt.

Es wurden beispielhaft drei Arbeiten gesichtet [20, 30, 33], welche sich auch mit dieser Art der Codierung befassen, um eventuell eine Art von Best Practices zu erkennen. Dies konnte nicht festgestellt werden. Daher wurden die folgenden Codierungen nach bestem Ermessen gewählt.

Bei allen Schaltern, die ein einfaches Ein-/Ausverhalten zeigen, ist 0 für nicht aktiv und 1 für aktiv. Bei den acht Optimierungen mit Parameter steht die 0 für nicht aktiv, und die Optionen sind von 1 beginnend durch nummeriert. Die Optionen `-finline-limit` und `-ftree-parallelize-loops` haben keine natürlichen Schranken, daher wurden sie auf $[0, 9]$ beschränkt.

So ergibt sich für das Genom ein Array von ganzen Zahlen, wobei jede Position in dem Array genau einer Optimierung zugeordnet ist, und der Wert repräsentiert die

[1] `http://parsec.cs.princeton.edu/`
[2] `http://impact.crhc.illinois.edu/parboil/parboil.aspx`
[3] `http://vhosts.eecs.umich.edu/mibench/`

Individuum 1:	0	0	1	0	2	1	Individuum 1:	0	0	**0**	**1**	**3**	**1**
Individuum 2:	1	0	0	1	3	1	Individuum 2:	1	0	**1**	**0**	**2**	**1**

Abbildung 5.2: Veranschaulichung des 1-Point-Crossover Verfahrens. Dabei wird ein Punkte gewählt, ab dem die Gene getauscht werden. Dieser Punkt ist durch die doppelte vertikalen Linie gekennzeichnet.

spezifische Einstellung einer Optimierung. Durch diese Realisation ist es auch möglich, dass die 3. Anforderung erfüllt wird, parametrisierte Optimierungen zu betrachten.

5.3.2 Selektion, Rekombination und Mutation

Für die Selektion wurde die Turnier-Selektion [26] (mit zurücklegen) verwendet, da sie entgegen der von Narayanamurthy, Pattabiraman und Ripeanu verwendeten Zufallsselektion (engl. *random selection*) [19] die besseren Individuen selektiert. Dadurch bekommen die bereits guten Individuen die Möglichkeit, sich durch Mutation und Rekombination zu noch besseren Individuen zu entwickeln. Bei der Tunier-Selektion werden aus der aktuellen Generation 5 Individuen zufällig ausgewählt, miteinander verglichen und das beste Individuum wird den Nachfolgern hinzugefügt. Dies wird so oft wiederholt bis die gewünschte Anzahl an Nachfolgern selektiert wurde, im konkreten Fall bis erneut die Generationsgröße erreicht wurde. Dabei ist zu erwarten, dass (die besonders guten) Individuen mehrfach ausgewählt werden

Die Implementierung der Rekombination gestaltet sich durch die gewählte Codierung entsprechend einfach: Jedes Point-Crossover Verfahren kann einfach übernommen werden. Für alle Experimente wurde das 1-Point-Crossover Verfahren [26] verwendet mit einer Rekombinationswahrscheinlichkeit $p_{rekomb} = 0.5$. Dieses Verfahren wurde genutzt, weil es das simpelste Verfahren ist und zu einer gute Mischung der Gene führt, und die hohe Wahrscheinlichkeit dient der Exploration des Suchraums. Dabei wird ein Punkte bestimmt, ab dem die Gene vertauscht werden. Ein Beispiel dazu findet sich in Abbildung 5.2.

Das Verfahren für die Mutation ist nicht viel komplizierter. Ein Individuum wird mit der Wahrscheinlichkeit $p_{mut} = 0.4$ mutiert. Sie liegt damit etwas höher als die von Narayanamurthy, Pattabiraman und Ripeanu gewählte Mutationsrate von $0{,}3$, doch da die Menge der möglichen Optimierungen deutlich großer ist, dürfte der höhere Wert zu einer besseren Exploration des Suchrraums führen. Die Wahrscheinlichkeit für eine Mutation eines einzelnen Gens eines Individuums, welches mutiert werden soll, liegt bei $p_{mut,gen} = 0.05$. Damit werden im Durchschnitt ca. 9 der 177 Gene eines Individuums mutiert, was eine gute Rate für die Erkundung des Suchraums sein sollte. Außerdem ist es nur eine durchschnittliche Wahrscheinlichkeit, die damit jedoch hoch genug sein sollte, dass bei vorhergesehener Mutation wirklich mindestens eins Gen mutiert wird (dies wird nicht weiter sichergestellt). Bei allen binären Einstellungen wird die Zahl geflippt. Bei solchen mit mehr Möglichkeiten wird ein neuer Wert zufällig

gleich verteilt ausgewürfelt, wobei verhindert wird, dass der gleiche Wert ausgewählt wird. Das führt zwar dazu, dass diese prinzipiell öfter „an“ sind, jedoch ist die Annahme, dass diese verschiedenen Status auch unterschiedliche Effekte haben. Daher wird „aus“ nicht mit $50\,\%$ gewählt, sondern einem entsprechend geringeren Anteil. Durch die Erzwingung eines anderen Wertes wird eine echte Mutation des Gens garantiert.

5.3.3 Zwischenspeicher für Individuen

Durch das Selektionsverfahren und die hohen Wahrscheinlichkeiten für Mutation und Rekombination ist es zu erwarten, dass gute Individuen verschwinden. Außerdem besteht die Möglichkeit, dass ein Individuum mehrfach parallel erzeugt wird oder später erneut entsteht. Daher werden alle Individuen, identifiziert durch einen Hash, zwischengespeichert. Bei Erzeugung bzw. Entstehung eines Individuums wird geprüft, ob ein Individuum mit gegebenem Hash bereits gespeichert ist. Falls ja, so wird das neue Individuum durch das gespeicherte ersetzt, ansonsten wird es im Zwischenspeicher abgelegt. So wird verhindert, dass Rechenzeit auf ein eigentlich schon bekanntes Individuum verschwendet wird. Außerdem wird mit Hilfe des Zwischenspeichers die letzte Generation künstlich erzeugt: Es werden nur die besten Individuen, entnommen aus dem Zwischenspeicher, für die Generation verwendet. So kann hier explizit geprüft werden, ob sich auch die besten Individuen ausreichend von einander unterscheiden, Denn zum Beispiel kann es sein, dass das beste nur in einer Generation und das zweitbeste nur in einer anderen Generation vorgekommen ist, sie konnten daher bisher nicht verglichen werden.

5.3.4 Fitness

Für die Fitness der Individuen wird das geometrische Mittel verwendet, welches nach

$$\overline{x} = \sqrt[n]{\prod_{i=1}^{n} x_i} \tag{5.1}$$

berechnet wird. Danach werden die Individuen einer Generation im Rahmen des dynamischen Resamplings sortiert, um die Rangfolge zu bestimmen.

5.4 Dynamisches Resampling

Das *Dynamic* in DyFIGA steht für das dynamische Resampling. Die Motivation hierfür ist, dass der Fehlerraum zu groß für eine vollständige Abtastung ist, aber ein einzelner festgelegter Wert (statisches Sampling) dagegen sowohl zu niedrig (Abdeckung ist zu niedrig, der EAFC-Wert ist zu ungenau) als auch zu hoch sein kann (in dem Sinne, dass CPU Zeit verschwendet wird und der Algorithmus länger als nötig läuft). Daher sollten

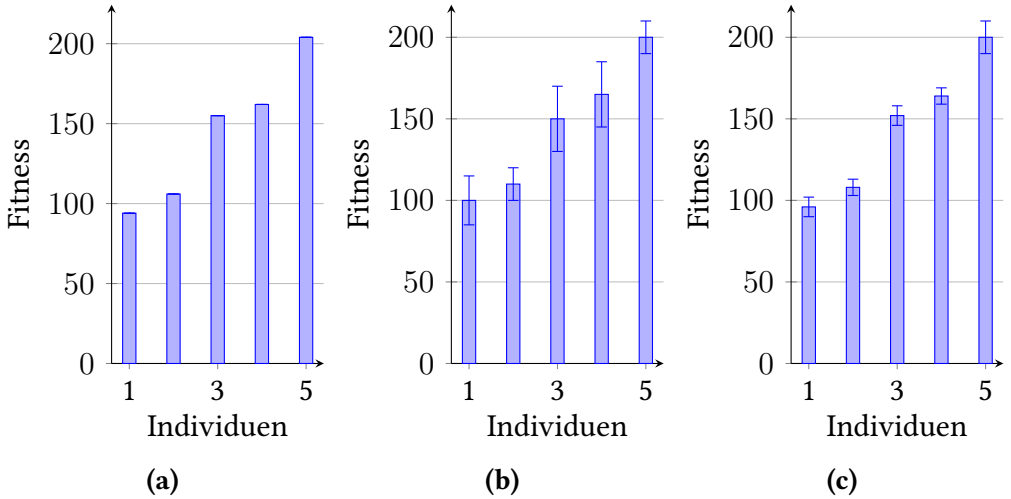

Abbildung 5.3: Drei Balkendiagramme mit 5 Individuen und deren Fitnesswert. In 5.3a sind die Fitnesswerte ohne Fehler. In 5.3b sind die Werte ungenauer und fehlerbehaftet. Die Individuen 1 und 2 sowie 3 und 4 haben überlappende Fehlerbereiche – eine eindeutige Sortierung ist nicht möglich. In 5.3c wurden die Individuen so weit genauer bestimmt, dass die Fehler nicht mehr überlappend sind – eine eindeutige Sortierung ist nun möglich.

die Anzahl der durchgeführten Fehlerinjektionen minimiert werden, aber darunter darf die Qualität des Algorithmus bzw. der Optimierung nicht leiden.

5.4.1 Problem der nicht vollständigen Abdeckung

Das damit einhergehende Problem ist in Abbildung 5.3 dargestellt. Wird der komplette Fehlerraum abgedeckt, so ist eine fehlerfreie Bestimmung des Fitnesswerts möglich (vgl. Abbildung 5.3a). Der Aufwand dafür ist jedoch zu hoch, statt dessen werden nur Stichproben aus dem Fehlerraum gezogen. Auf Grund dessen sind die ermittelten Fitnesswerte jedoch fehlerbehaftet. Wie in Abbildung 5.3b abgebildet ist kann es dazu führen, dass sich die Fehlerbereiche überlappen. Eine eindeutige Sortierung bzw. Bewertung ist dann nicht mehr möglich. Daher müssen bei einer vorliegenden Überlappung die betroffenen Individuen genauer bestimmt werden. Dies geschieht durch das (dynamische) Nachziehen von Stichproben – die Fitnesswerte werden genauer bestimmt und der Fehler reduziert, dargestellt in 5.3c. Dadurch wird eine eindeutige Sortierung bzw. Bewertung ermöglicht.

Bis hier sind wir jedoch davon ausgegangen, dass sich Fitness und Fehler für jedes Individuum einfach und eindeutig berechnen lassen. Würde das Verfahren nur auf Basis eines einzelnen Benchmarks durchgeführt, wäre dem auch so: Die Fitness wäre die EAFC Metrik (vgl. Unterabschnitt 3.2.3). Der Fehler wird nach einer von Li u. a. [15]

vorgestellte Formel berechnet:

$$e = \zeta \sqrt{\frac{c_{true}(1 - c_{true})}{F} \frac{\omega - F}{\omega - 1}} \tag{5.2}$$

Diese gibt an, wie groß der Fehler ist abhängig von dem Konfidenzniveau ζ, der Fehlerraum ω und der Samplezahl F relativ zur Raumgröße. Entsprechend muss nach der Berechnung des Wertes e dieser mit ω multipliziert werden, um den absoluten Fehler $E = \omega \cdot e$ zu erhalten. Im Rahmen der Arbeit wird ein Konfidenzniveau von $95\,\%$ verwendet und die damit einhergehende Schwelle $\zeta = 1{,}96$. Da c_{true} unbekannt ist, soll es gemäß dem schlimmsten Fall auf $c_{true} = 0{,}5$ gesetzt werden. Entsprechend kann man die Formel vereinfachen:

$$e = \zeta \sqrt{\frac{0{,}25}{F} \frac{\omega - F}{\omega - 1}} \tag{5.3}$$

Diese Berechnung kann man für jedes Individuum und jeden Benchmark vornehmen. Jedoch ist die mehrkriterielle Optimierung nach 17 Kriterien (Benchmarks) nicht sinnvoll. Daher müssen die einzelnen Ergebnisse der Benchmarks zu einem Wert zusammengefasst werden, mit dem entschieden werden kann, ob ein Resampling notwendig ist.

5.4.2 Berechnung eines einzelnen Resampling-Kriteriums

Dafür sei $EAFC(I, b)$ der EAFC-Wert des Benchmarks $b \in B$ für das Individuum I. Dabei ist B die Menge aller untersuchten Benchmarks. Analog dazu sei der Fehler $E(I, b)$. Mit dem Welch-Test [28], dessen p-Wert in allgemeiner Form nach

$$p = \frac{\overline{x}_1 - \overline{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5.4}$$

berechnet wird, kann getestet werden, wie weit sich zwei unabhängige Stichproben von einander unterscheiden. Dabei ist $\overline{x}_i$ der Mittelwert, s_i die Standardabweichung und n_i die Anzahl der Stichproben. Das Ergebnis des Tests gibt an, wie hoch die Wahrscheinlich ist, dass beide Stichproben der selben Grundmenge entstammen.

Im Rahmen des Algorithmus muss dieser Wert für jeden einzelnen Benchmark b eines Individuen-Nachbarpaars I_1, I_2, gegeben durch die Sortierung nach dem Fitnesswert, berechnet werden. Entsprechend ergibt sich aus

$$p(I_1, I_2, b) = \frac{EAFC(I_1, b) - EAFC(I_2, b)}{\sqrt{\frac{E(I_1,b)^2}{N(I_1)} + \frac{E(I_2,b)^2}{N(I_2)}}} \tag{5.5}$$

der Test, wobei $N(I_i)$ die Anzahl der gezogenen Samples für das Individuum I_i ist (diese ist gleich für alle Benchmarks). Damit erhält man die Testwerte zweier Individuen

$P(I_1, I_2) = \{p(I_1, I_2, b) | b \in B\}$. Diese müssen noch in einem einzelnen Wert zusammengefasst werden. Dafür wird Fishers Methode [8] zum Kombinieren von p-Werten verwendet. Diese führt zu der Statistik

$$X = -2 \sum_{i=1}^{k} \ln(p_i) \tag{5.6}$$

wobei X einer X_{2k}^2 Verteilung folgt. Aus dieser kann der kombinierte p-Wert entnommen werden. Nach diesem kombinierten p-Wert zwischen zwei Individuen richtet sich die Entscheidung, ob diese weiter gesamplet werden.

Im Konkreten läuft das dynamische Resampling wie folgt ab: jedes Individuum wird 200 mal gemessen für einen initialen Wert. Sollten weitere Messwerte erforderlich sein, so werden immer 100 neue Messungen angefordert. Es ist nicht immer nur eine Messung, da erstens vermutlich mehr als nur eine weitere Messung notwendig ist und zweitens um zu verhindern, dass der FAIL*-Server und seine Clients leer laufen (statt leer zu laufen können genauso gut weitere Messungen durchgeführt werden). Um nicht in eine zu lange Schleife zu laufen wurde ein Limit von maximal 1000 gesetzt.

Es hat sich außerdem gezeigt, dass Benchmarks nicht immer SDCs produzieren. Dies ist vor allem ein Problem, weil damit rechnerisch die EAFC-Metrik gleich 0 ist. Damit das geom. Mittel und dadurch die Fitness nicht auch auf 0 fällt, wird sie künstlich auf 1 gesetzt. Dennoch hat es einen maßgeblichen Einfluss auf die Fitness, denn typischerweise liegt die EAFC-Metrik eines einzelnen Benchmarks im Bereich von 100 Millionen oder Milliarden. Man stelle sich dazu folgendes vor: Es gibt zwei Individuen, welche identisch sind bis auf ein einzelnes Experiment. Dieses Experiment führt dazu, dass das erste Individuum bei einem Benchmark einen SDC sieht, das zweite hat dort keinen SDC. Sie würden sich gemessen am geometrischen Mittel deutlich unterscheiden.

Um dieses Problem möglichst gering zu halten, wurde deswegen noch eine weitere Bedingung für ein Resampling eingeführt: Sollte mindestens ein Benchmark eines Individuums keinen SDC erzeugt haben, dann werden weitere Stichproben gezogen, jedoch nur bis zu einem Maximum von 500. Eine Übersicht über den sich daraus ergebenen Gesamtablauf beim Resampling ist in Algorithmus 1 dargestellt.

Das hier vorgestellte dynamische Resampling hilft maßgeblich dabei, die Laufzeit, wie in Anforderung 5 verlangt, in einem vertretbaren Rahmen zu halten Trotzdem kann so eine höhere Abtastung bei gegebener Notwendigkeit ermöglicht werden.

Eine entsprechend angepasste Grafik des Ablaufs ist in Abbildung 5.4 abgebildet. Es ist sehr ähnlich zur Abbildung 3.4 für den generischen GA und wurde um die Schritte *(Re)Sampling* und *Verifikation Individuen sind unterschiedlich* erweitert.

```
while waiting on results do
    get result ;
    sort individuals;
    foreach individual do
        if individual has benchmark with 0 SDC then
            if number samples < 500 then
                draw 100 samples for individual;
            end
        end
        for i ← 1 to 3 do
            if no significant difference between individual and neighbour i then
                if individual has all results and number samples < 1000 then
                    draw 100 samples for individual;
                end
                if neighbour i has all results and number samples < 1000 then
                    draw 100 samples for neighbour i;
                end
            end
        end
    end
end
```

Algorithmus 1 : Pseudocode des Ablaufs des Resamplings.

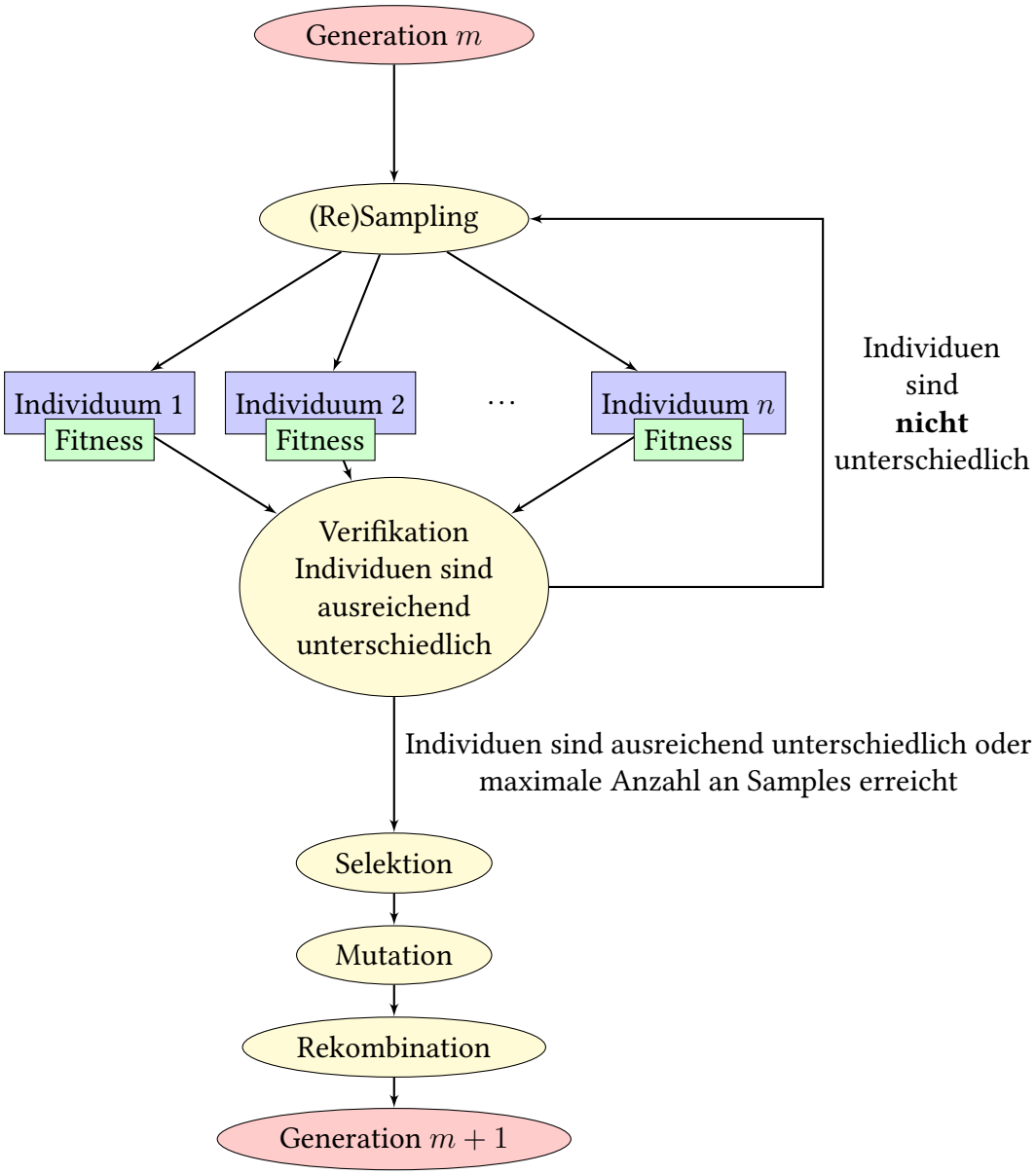

Abbildung 5.4: Schematische Abbildung des Ablaufs eines GA, erweitert um die Funktionalität des dynamischen Resamplings.

5.5 Anpassungen und Verbindung zu FAIL*

FAIL* unterstützte zuvor nur statische Kampagnen. Das bedeutet, dass sowohl das Binary als auch die Fehlerinjektionspunkte vor dem Start feststehen müssen. Jedoch werden für diese Arbeit dynamische Experimente benötigt: Die Injektionspunkte werden zur Laufzeit ermittelt, die Binaries werden erst beim Durchlaufen erzeugt. Entsprechend müssen Anpassungen an FAIL* vorgenommen werden.

Es muss eine Kampagne erstellt werden, die Aufträge von DyFIGA annimmt und an die Clients weiterleitet (und die Ergebnisse zurückschickt). Weiter muss ein Experiment erstellt werden, welches sich zur Laufzeit das benötigte Binary beschaffen kann und dann einen Fehler injiziert.

Die Kommunikation zwischen DyFIGA und FAIL* soll mit Google Protocol Buffer [11] Nachrichten realisiert werden. Diese werden bereits intern von FAIL* genutzt. Daher bietet es sich an, sie auch für die Kommunikation zwischen DyFIGA und FAIL* zu nutzen.

FAIL* ermöglicht durch die bereits vorhandene Möglichkeit zur hohen Parallelität die geforderte vertretbare Laufzeit trotz der vielen geforderten Experimente.

5.6 Zusammenfassung

In diesem Kapitel wurde der Entwurf vorgestellt. Dabei wurden zuerst die betrachteten Optimierungen vorgestellt. Anschließend wurden die Details des GA erläutert, sowohl die Codierung mit ganzen Zahlen als auch die Umsetzung der Selektion, Mutation und Rekombination. Danach wurden die Liste der 17 genutzten Benchmarks genannt, ausgewählt aus den Suiten PARSEC, Parboil und Mibench. Das dynamischen Resampling Verfahren wurde anschließend erklärt, welches dazu dient, dass nur so viele Experimente durchgeführt werden wie notwendig. Als letztes wurde erläutert, welche Anpassungen an FAIL* vorgenommen werden müssen und wie die Verbindung zwischen DyFIGA und dem System realisiert werden soll. Im folgenden Kapitel werden einige Implementierungsdetails vorgestellt.

6 Implementierung

In diesem Kapitel werden Details der Implementierung gezeigt, die sich nicht direkt aus dem Entwurf ergeben. Dabei werden der neu entwickelte DyFIGA, die Änderungen an FAIL*, die Kommunikation zwischen den beiden und die Anpassungen der Benchmarks erläutert.

Ein Schema des Gesamtsystems ist in Abbildung 6.1 dargestellt. DyFIGA erzeugt auf der einen Seite Individuen, welche die genutzten Optimierungen definieren. Mit diesen Optimierungen werden die Binaries der 17 Benchmarks gebaut und der Golden Run durchgeführt. Anschließend werden sie über einen Webserver den Clients bereitgestellt. Auf der anderen Seite kommuniziert DyFIGA mit der entwickelten (DyFIGA) FAIL* Kampagne durch *DyfigaProtoMsg* und Experimentnachrichten. Die Kampagne leitet die Experimentnachrichten zur Durchführung des definierten Experiments an die Clients weiter. Die Clients laufen dabei auf dem LiDO3 Cluster der TU Dortmund. Die Maschinen besitzen ein gemeinsames geteiltes Dateisystem sowie ein separates lokales Verzeichnis, dies wird von wesentlicher Bedeutung in Unterunterabschnitt 6.3.2.2 sein.

6.1 DyFIGA

DyFIGA wurde in Python entwickelt. Dabei wurde für den unterliegenden GA das DEAP Framework[1] [9] verwendet. Es stellt Standardimplementierung für u. a. Individuen, die Erstellung der ersten Generation, für Selektion, Mutation und Rekombination bereit. Aufgrund der Codierung mit ganzen Zahlen musste jedoch eine eigene Implementierung der Individuen, der Mutation und Rekombination vorgenommen werden.

6.1.1 Implementierung der Individuen

Für die Individuen wurde eine eigene Klasse erstellt. Sie wird initialisiert, in dem aus einer Datei alle Optimierungen ausgelesen werden. Dabei werden die Optimierungen, die nicht betrachtet werden sollen, ausgefiltert. Weiter wird zwischen einfachen und parametrisierten Optimierungen unterschieden, da solche mit Parameter eine aufwendigere Initialisierung benötigen (setzen der verschiedenen Parameter als Möglichkeiten). Zusätzlich werden die Standardlevel `-O1`, `-O2`, `-O3` in die Form eines Genoms überführt. Später kann so für jedes Individuum das „nächste“ Standardlevel gefunden werden.

[1] https://github.com/deap/deap

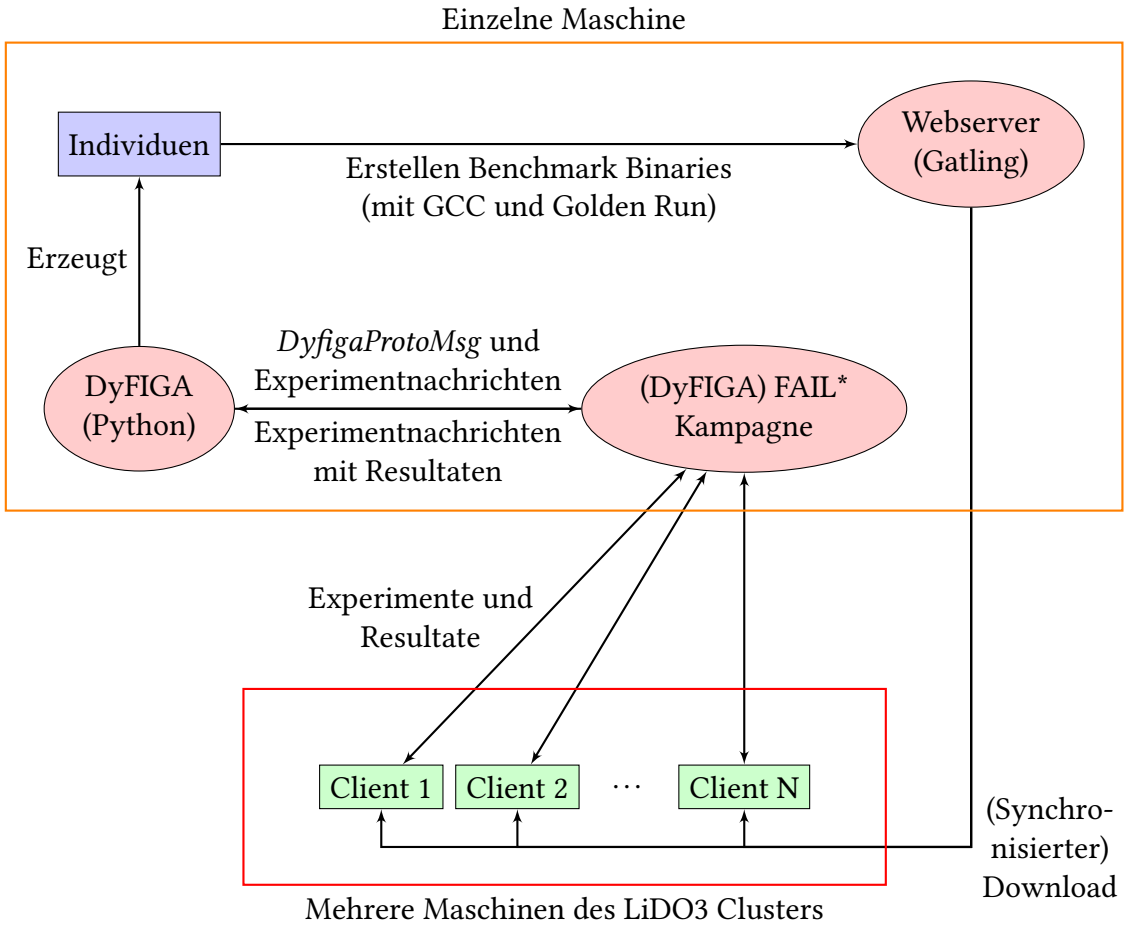

Abbildung 6.1: Schema der Implementierung. Die eingerahmten Komponenten laufen alle auf einer einzelnen Maschine

Bei der Instanziierung eines Individuums wird das Genom zufällig bestimmt und die daraus resultierende Liste von Optimierungen erstellt. Aus diesem String wird dann ein Hash (mit SHA1) in Hexadezimaldarstellung erzeugt, der zur kompakten eindeutigen Identifizierung dient. Außerdem wird für jedes Individuum bestimmt, was das nächste Standardlevel gemessen an dem Hamming-Abstand ist (entspricht der Anzahl der Gene, die geändert werden müssen). Würden die Abhängigkeiten von Optimierungen implementiert werden, dann könnten damit kürzere Optimierungsstrings erstellt werden. Dafür kann das Standardlevel genommen werden, dabei fehlende Optimierungen hinzugefügt werden und nicht vorkommende Optimierungen, die aber in dem Standardlevel sind, durch den entsprechenden `-fno-...`-Schalter deaktiviert werden.

6.1.2 Evaluation der Individuen

Vor der Evaluation eines Individuums muss zunächst sichergestellt werden, dass die zugehörigen Binaries existieren. Sollte dies nicht der Fall sein, so werden alle Benchmarks mit den gegebenen Optimierungen compiliert und der Golden Run durchgeführt. Dafür wurden Makefiles verwendet, welche eine parallele Übersetzung und Ausführung des Golden Run ermöglichen. Dieser Prozess dauert typischerweise etwa 80 Sekunden, daher wurde hier ein Timeout von drei Minuten verwendet, um zu verhindern, dass der Algorithmus eine unbestimmte Zeit für den Golden Run eines einzelnen Individuums benötigt. Individuen, bei den das Erzeugen der Binaries fehlschlägt, werden als invalide markiert. Das ist der Fall bei Compilerfehlern, Timeout oder wenn die Ausgabe nicht mit einem vordefinierten Ergebnis, welches durch den Golden Run der unoptimierten Version gewonnen wurde, übereinstimmt. Diese Prüfung wird durchgeführt, um solche Individuen auszufiltern, welche bereits ohne die Injektion eines Fehlers ein anderes bzw. falsches Ergebnis liefern. Bei Erfolg werden die Binaries an einer vordefinierten Stelle abgelegt, wo ein Webserver sie für die Clients bereitstellt. Für den Webserver wurde gatling[2] verwendet. Nach dem Erstellen der Binaries werden die initialen Jobs an die Kampagne geschickt. Für jedes Individuen sind es 3400 Experimente (200 Experimente · 17 Benchmarks). Dabei werden das CPU-Register (ausgewählt aus den Allzweck- und FPU-Registern), der Zeitpunkt und das Bit-Offset im Register zufällig bestimmt.

6.1.3 Dynamisches Resampling

Wurden für alle Individuen die Binaries erzeugt, dann geht DyFIGA dazu über, die Ergebnisse zu empfangen und zu prüfen, ob weitere Samples notwendig sind. Es hat sich beim Ausführen herausgestellt, dass diese Überprüfung den Algorithmus ausbremsen kann. Daher findet diese Prüfung nur mit einem vorgegebenen aber dynamischen Abstand statt und nicht bei jedem neuen eingehenden Resultat. Dabei wird der Abstand verringert, je weniger Experimente noch ausstehen, um eventuell notwendige

[2] `https://www.fefe.de/gatling/`

Ausstehende Experimente	Abstand der Prüfung
> 10.000	1000
> 5.000	250
> 1.000	100
> 500	50
≤ 500	25

Tabelle 6.1: Grenzen und Abstand für Prüfungen, ob weitere Stichproben notwendig sind.

weitere Stichproben schneller zu erkennen und ein Leerlaufen der Injektionsclients zu vermeiden. Eine genaue Auflistung der Grenzen und Abstände findet sich in Tabelle 6.1. Hier sei bereits angemerkt, dass teilweise 3.000 bis 4.000 Clients parallel liefen. Entsprechend beginnen erste Clients ab nur noch 3.000 bis 4.000 verbleibenden Experimenten leerzulaufen.

Für die Sortierung der Individuen wird das geometrischen Mittel der EAFC-Metriken genutzt. Nach dem Sortieren wird über alle Individuen für die Prüfung, ob weitere Samples notwendig sind, iteriert. Zuerst wird geprüft, ob ein Benchmark noch keinen SDC produziert hat. Anschließend wird mit den folgende drei Individuum verglichen, ob sie sich signifikant unterscheiden – zunächst war es nur der direkte Nachbar, doch durch erste Testläufe, welche in Abschnitt 7.1 präsentiert werden, wurde eine Erhöhung auf drei Nachbarn motiviert. Der Test wird jedoch nur durchgeführt, falls für mindestens ein Individuum alle Ergebnisse vorliegen. Damit soll verhindert werden, dass frühzeitig weitere Samples angefordert werden, welche eigentlich nicht gebraucht werden. Stellt sich dabei heraus, dass weitere Stichproben nötig sind, so werden für das Individuum, welches alle Ergebnisse vorliegen hat, weitere Samples angefordert (falls beide vollständig sind, werden für beide weitere Samples gefordert). Hier wurde eine Ausnahme eingeführt: Ist die Gesamtzahl der ausstehenden Experimente kleiner oder gleich 500, dann wird der Test auch durchgeführt, falls weder für das eine noch das andere Individuum alle Ergebnisse vorliegen. Dies soll verhindern, dass erst alle Ergebnisse abgewartet werden und die angeforderten Experimente auf 0 sinken, bevor festgestellt wird, dass weitere Samples notwendig sind. Zur Berechnung des Welch-Tests und dem kombinierten p-Wert nach Fishers Methode wurden die entsprechenden Methoden aus dem `scipy.stats` Python-Modul verwendet.

6.2 Abschluss einer Generation

Am Ende einer jeden Generation werden die Werte der Individuen der aktuellen Generation in eine Datei geschrieben und Schaubilder erzeugt. Die Bilder werden mit dem Python-Module `plotnine` erstellt, welches eine Schnittstelle zu ggplot2[3] bereitstellt.

[3] https://ggplot2.tidyverse.org/

Weiter wird die Datei, welche die Aufschlüsselung von Hash zu Optimierungsstring enthält, aktualisiert. Außerdem werden die Gesamtzahl der vergebenen Experimente und die aufgewendete CPU-Zeit (mitgeschrieben von den Injektionsclients) in eine Datei geschrieben.

6.3 FAIL*

Wie bereits in Abschnitt 5.5 erwähnt, müssen einige Anpassungen an FAIL* vorgenommen werden. Es muss eine eigene Kampagne geschrieben werden, für das Experiment genügt die Anpassung des *generic-experiment*. Außerdem wurde das *generic-tracing* modifiziert, welches den Golden Run durchführt, so dass es schneller läuft.

6.3.1 Kampagne

Es wurde eine eigene neue Kampagne geschrieben, welche recht simpel ist: Sie muss einerseits Aufträge von DyFIGA empfangen und dem Job-Server zur Verfügung stellen. Auf der anderen Seite muss sie die Ergebnisse der Experimente zurück an DyFIGA schicken. Dafür wird ein Socket mit einem zuvor definierten Port geöffnet und darauf gewartet, dass sich DyFIGA verbindet. Das Senden und Empfangen wird jeweils in einem eigenen Thread gestartet, damit diese vollständig unabhängig von einander agieren können und sich nicht blockieren. Die Experiment-Nachrichten werden dabei als Blackbox gehandhabt und einfach weitergereicht.

Außerdem kann die Kampagne mit einer speziellen Nachricht abgeschlossen werden. Dadurch endet sie ordnungsgemäß und kann zusätzlich den Injektionsclients eine Nachricht senden, dass sie stoppen können.

6.3.2 Injektionsclient

Das *generic-experiment*, welches als Basisklasse das *DatabaseExperiment* hat, erfüllt eigentlich alle Anforderungen, außer dass es vorher nicht möglich war das Binary erst zur Laufzeit herunterzuladen (diese werden auch erst zur Laufzeit erzeugt). Entsprechend wurde das Experiment um diese Funktionalität erweitert. Dabei wurde die Struktur zur Kommunkation zwischen dem *DatabaseExperiment* und *generic-experiment* beibehalten, wobei das *DatabaseExperiment* Callback-Funktionen bereitstellt und diese vom *generic-experiment* implementiert werden.

Dafür wurden die beiden Callback-Funktionen `cb_expect_download` und `cb_after_download` dem *DatabaseExperiment* hinzugefügt. Die erste fragt ab, ob ein Download vorgesehen ist, was durch einen Kommandozeilenparameter beim Start des Clients festgelegt wird. Ist das der Fall, so wird der Downloadprozess gestartet, und nach erfolgreichem Abschluss die zweite Funktion aufgerufen, damit sich das *generic-experiment* vollständig initialisieren kann.

```
Input : lockfile Name
if Binary does not exist then
    lockfd = open(lockfile, O_CREAT|O_RDWR|O_EXCL,
     0600);
    if open fehlgeschlagen mit EEXIST then
        lockfd=open(lockfile, O_RDWR, 0600);
    end
    flock(lockfd, LOCK_EX);
    if Binary does not exist then
        download Binary;
    end
    delete and close Lockfile;
end
```

Algorithmus 2 : Ansatz, um Downloads zu synchronisieren.

6.3.2.1 Synchronisation der Downloads

Da parallel mehrere Clients auf einer Maschine laufen, sollten die Downloads synchronisiert werden. Diese Synchronisation wurde zunächst auf Binary-Ebene umgesetzt, so dass nur gleiche Downloads miteinander synchronisiert werden.

Dieses wurde durch den in Algorithmus 2 gezeigten Ansatz umgesetzt. Die Fehlerprüfungen der verschiedenen Aufrufe sind dabei nicht zu sehen. Die Aufrufe von `open` und `flock` sind die original Aufrufe aus dem C++-Quellcode[4]. Der erste Aufruf von `open` kann fehlschlagen, daher wird bei Fehlschlag der Grund geprüft. Ist die Ursache, dass die Datei existiert, so wird die (bereits existierende) Datei mit O_RDRW geöffnet.

Diese Variante funktionierte bei den Testläufen problemlos, bei denen nur ca. 50 Clients auf dem Maschine gestartet wurden, auf dem auch das restliche System lief.

6.3.2.2 Ressourcenschonende Downloadsynchronisation

Bei der Nutzung des LiDO3 Clusters und den dadurch mehreren 1.000 Clients zeigte sich jedoch ein neues Problem: Da nun auf vielen Maschinen mehrere Clients liefen, wurde der Webserver, welcher die Downloads bereitstellt, und das Netzwerk stark be- bis überlastet[5]. Daher musste eine neue Lösung gefunden werden, welche diese Ressourcen schont. Sie ist nicht mehr generisch, sondern ist speziell für die genutzte Umgebung erstellt worden – eine bessere und allgemeine Lösung wäre hier, den Downloadprozess

[4] POSIX garantiert beim Aufruf von `open` ohne O_EXCL nicht, dass die Existenzprüfung und Erstellung der Datei atomar ist. Beobachtungen bei der Ausführung haben darauf hingedeutet, dass dies nicht der Fall ist.

[5] Einerseits konnte der Webserver die ganzen Anfragen nicht gleichzeitig beantworten, andererseits wurde einer der Router zwischen dem kos-Server (auf dem DyFIGA und Webserver liefen) und dem Cluster zu 100 % ausgelastet.

```
Input : Benchmark und Hash
if Binary does not exist local then
    lock local ;
    if Binary does not exist shared then
        calculate CRC from Benchmark and Hash ;
        use first two hexadecimal digits of CRC as shared lock ;
        lock shared ;
        download Binary;
        release lock shared ;
    end
    copy Binary from shared to local ;
    release lock local ;
end
```

Algorithmus 3 : Ansatz, um Downloads bei tausenden Clients zu synchronisieren.

komplett auszulagern, z. B. in ein Shell-Skript. Dafür war jedoch die nötige Zeit nicht mehr vorhanden.

Dieses Synchronisationsverfahren verlief wie folgt. Jede Maschine des Clusters hat ein separates lokales Dateiverzeichnis und ein mit allen anderen Maschinen geteiltes Verzeichnis. In dem geteilten Dateiverzeichnis liegen 256 verschiedene Dateien (alle möglichen Kombinationen von zwei Hexadezimalziffern). Sie dienen als Lock, um ein Binary in das gemeinsame Dateiverzeichnis herunterzuladen. Außerdem existiert eine lokale Datei, die als Lock für lokale Dateiänderungen dient, genauer gesagt das Kopieren von Binaries aus dem geteilten Bereich in den lokalen.

Das Verfahren ist in Algorithmus 3 in Pseudocode zu sehen. Dabei wurden die erneuten Überprüfungen, ob das Binary nach dem Nehmen des Locks existiert, nicht notiert, sie sind aber implementiert. Außerdem sei hier explizit erwähnt, dass das Verfahren, die Lock-Datei in unterschiedlichen Modi zu öffnen, hier nicht notwendig gewesen ist, weil sie alle bereits existieren und nicht gelöscht werden.

Es wird zunächst geprüft, ob das Binary lokal bereits existiert. Falls nicht, dann wird das lokale Lock genommen und überprüft, ob es bereits im geteilten Dateiverzeichnis vorliegt. Ist das der Fall, dann kann es ins lokale Verzeichnis kopiert und das Lock freigegeben werden. Ist die Situation jedoch so, dass es auch geteilt nicht existiert, dann wird zunächst die zyklische Redundanzprüfung CRC für die Kombination aus Benchmark und Hash (beides von DyFIGA übermittelt) berechnet. Von dem CRC Wert dienen die ersten beiden Hexadezimalziffern als Name für das geteilte Lock – der original Hash wurde hier nicht genommen, um zu ermöglichen, dass verschiedene Benchmarks mit gleichem Hash gleichzeitig heruntergeladen werden können. Das Lock an der so ermittelten Lock-Datei wird genommen und dann wird das Binary heruntergeladen. Anschließend wird das geteilte Lock freigegeben, das Binary ins

lokale Verzeichnis kopiert und das lokale Lock auch freigegeben.

Durch dieses relativ komplexe Zwei-Phasen-Locking konnte nahezu die komplette Last von dem Webserver und Netzwerk genommen werden – jedes Binary wird nur ein einziges mal angefordert, die weitere Verteilung geht über das geteilte Dateisystem. Trotzdem ist die hohe Parallelitätsmöglichkeit dadurch nicht stark eingeschränkt: Spätere Durchläufe legen die Vermutung nahe, dass so über 2.000 Clients (etwa 4.000 waren gestartet) problemlos arbeiten können, auch wenn es sich bei den Experimenten um neue Benchmark-Hash-Kombinationen handelt. Werden von Kombinationen viele weitere Experimente gefordert, so sind die Dateien mit der Zeit überall lokal vorhanden und es wird vollständige gleichzeitige Parallelität erreicht. Dabei sei gesagt, dass ansonsten – wie bereits erwähnt – die Auslieferung der Binaries einen Flaschenhals bildet und dadurch die Parallelität begrenzt ist.

6.3.2.3 Experimentnachrichten

Natürlich mussten die verwendeten Nachrichten, die die Experimente definieren, angepasst werden. So musste der Timeout, der zuvor per Kommandozeilenparameter bestimmt war, nun über die Nachricht gesetzt werden, denn er ist für jedes Binary anders. Weiter wird der Downloadpfad mitgeschickt, von dem die Binaries heruntergeladen werden können. Außerdem wird jetzt von dem Client die verbrauchte Realzeit für das Experiment gemessen und mit dem Ergebnis zurückgeliefert.

6.3.2.4 Reproduzierbarkeit der Ergebnisse

Ein anderes Problem war, dass zu Beginn die Ergebnisse der Injektionen nicht reproduzierbar waren – das Ergebnis war z. B. SDC, aber bei Überprüfung ergab sich ein OK_MARKER. Dieses wurde erst an Einzelbeispielen, dann durch doppelte Vergabe jedes Experiments und späteren Vergleich der Ergebnisse flächendeckend festgestellt. Der Client war so implementiert, dass er mehrere Experimente nacheinander durchführt, bevor er sich beendet und neu gestartet wird – der Neustart war wegen einem (bereits zuvor vorhandenem) Speicherleck erforderlich. Es hat sich herausgestellt, dass genau dieses Verhalten zu den falschen Ergebnissen führte. Hier wird vermutet, dass einige Zustände oder Ähnliches zwischen den Experimenten nicht korrekt bereinigt werden und so nachfolgende Experimente beeinflussen. Dies scheint vor allem dadurch ein Problem zu sein, dass jetzt verschiedenste Binaries geladen werden. Entsprechend ist die Anzahl an Experimente vor einem Neustart auf 1 begrenzt worden. Das hat zwar den Nachteil, dass dadurch die Verbindung zum Server immer neu und damit deutlich häufiger erstellt werden muss, jedoch ist es die einzige verhältnismäßige Maßnahme gewesen dem Problem aus dem Weg zu gehen.

```
message DyfigaProtoMsg {

  enum Command
  {
    WORK_FOLLOWS = 1;
    END_CAMPAIGN = 2;
  }
  required Command command = 1;
  optional uint32 size = 2;
}
```

Abbildung 6.2: Steuernachricht zwischen DyFIGA und FAIL*.

6.3.3 Golden Run

Der Golden Run wird durch das *generic-tracing* realisiert. Dabei wurden in der ursprünglichen Umsetzung alle Instruktionen und Speicherzugriffe mitgeschrieben. Beides war nicht notwendig: Einzig interessant ist die Gesamtlaufzeit, also die Anzahl der Instruktionen. Die genauen Details und Abfolge der Instruktionen und Speicherzugriffe dagegen sind für diesen Anwendungsfall uninteressant. Denn der Injektionszeitpunkt wird zufällig aus der Gesamtlaufzeit gewählt, und die Grenzen des vom Programm genutzten Speicherbereichs werden nicht benötigt, da in Register injiziert wird.

Daher wurde das *generic-tracing* so angepasst, dass es bei gegebenem Kommandozeilenparameter einzig die Gesamtlaufzeit misst. Dadurch konnte die Laufzeit des Golden Run von mehreren Minuten auf unter eine Minute oder knapp darüber reduziert werden.

6.4 Kommunikation zwischen DyFIGA und FAIL*

Wie bereits erwähnt, werden zur Kommunikation Google Protocol Buffer [11] (Version 2) verwendet. Die bereits bestehenden Nachrichten für das *generic-experiment*, welche auch eine untergeordnete Nachricht passend für das *DatabaseExperiment* enthält, wurden angepasst und für das gesamte System verwendet. Sie werden auf der Seite von DyFIGA erstellt, die Kampagne leitet sie weiter und vom Client empfangen. Dieser führt das Experiment aus, schreibt das Ergebnis in die Nachricht und sendet sie zurück zur Kampagne, und diese zurück an DyFIGA.

Zur Steuerung wurde zusätzlich eine weitere Nachrichtenart erstellt, die *DyfigaProtoMsg*. Zu sehen ist sie in Abbildung 6.2. Es wird immer ein *command* geschickt. Bei Setzung auf `WORK_FOLLOWS` wird zusätzlich *size* angegeben, welche aussagt, wie viele Experiment-Nachrichten folgen, bevor die nächste Steuernachricht zu erwarten ist. Die andere Möglichkeit ist `END_CAMPAIGN`, die Kampagne hört dann auf Nachrichten zu empfangen und bereitet das Ende der Kampagne vor.

Da die Kommunikation über einen Socket als Stream geschieht und die Nachrichten variable Größe haben, musste hier ein System entwickelt werden, welches die Kommunikation korrekt abwickelt. Dafür werden zuerst immer vier Bytes geschickt, in denen die Größe der folgenden Nachricht in Bytes steht. Danach wird die Nachricht selbst übertragen. Dafür muss der Ablauf der Nachrichtentypen festgelegt sein, ansonsten würde die Kommunikation fehlschlagen. Auf Seite von FAIL* wird zuerst immer eine Steuernachricht, und dann bei anstehender Arbeit die gegebene Anzahl an Experimentnachrichten erwartet. DyFIGA dagegen rechnet immer nur mit Ergebnissen gefüllte Experimentnachrichten.

6.5 Benchmarks

Die Benchmarks mussten auf das Embedded-Betriebssystem eCos[6] portiert werden, damit sie in Verbindung mit FAIL* liefen. Dieses bietet neben Start- und Initialisierungscode auch viele Standardfunktionen wie *printf*. Das *printf* schreibt dabei an die serielle Schnittstelle, welche vom Client überwacht wird und hierdurch die Ausgabe mitgelesen werden kann. Wofür jedoch keine Implementierungen vorlagen waren *float*-Operationen wie *sqrtf*. Diese wurden durch Konvertierung in ein *double*, Ausführung der entsprechenden *double*-Operation und Rückkonvertierung in *float* umgesetzt.

Des Weiteren mussten die Eingabedaten in das Binary eingebettet werden, welche üblicherweise aus einer Datei ausgelesen werden. Dafür werden die Eingabedateien mit *objcopy --input binary --output elf32-i386 --binary-architecture i386* in eine Objekt-Datei gewandelt. Aus dieser wurden die Eingabedaten, welche als ein langer String vorliegen, mit eigenen Implementierungen für die typischen Leseoperationen wie *fscanf* oder *getc* ausgelesen. Die Eingabedaten wurde teils gekürzt, um eine vertretbare Laufzeit von maximal etwa einer Minute zu erreichen.

Ein weiteres Problem war die Ausgabe: Die Benchmarks geben komplette Lösungen aus, welche sich über viele tausende oder auch zehntausende Zeilen erstrecken. Eine solch lange Ausgabe nimmt sehr viel Zeit in Anspruch: Zu einem Zeitpunkt der Entwicklung lief der Golden Run von *bfs* ohne Ausgabe ca. 5 Minuten und mit Ausgabe ca. 55 Minuten. Auf Grund dessen musste eine Möglichkeit gefunden werden, die Ausgaben zu komprimieren, aber Fehler trotzdem zu erkennen. Dafür wurden als Prüfzahlen die Summe und das XOR über die Ausgabe gewählt. Für *int*-Ausgaben ist dies unproblematisch und einfach umzusetzen. Für ein *float* ist dagegen ein XOR nicht definiert. Daher wurde eine eigene Implementierung erstellt, welche die Bytes eines *float* als *uint32_t* interpretiert und das XOR mit einem gegebenen *uint32_t* bildet.

Mit Hilfe dieser Prüfzahlen konnte der Umfang der Ausgabe deutlich reduziert werden. Damit ist die Ausgabe nur noch ein Bruchteil der Laufzeit und die eigentliche Berechnung des Benchmarks nimmt den größten Teil ein. Das ist wichtig, damit die

[6] `http://ecos.sourceware.org/`

Fehler (meistens) während der Berechnungszeit injiziert werden und nicht bei der Ausgabe.

Es soll noch erwähnt werden, dass für den Mibench Benchmark qsort der Aufruf der (eCos-)Bibliotheksmethode durch eine Implementierung im Benchmark-Quellcode ersetzt wurde[7]. Hintergrund dafür ist, dass die Bibliothek bereits compiliert vorliegt und nicht mit optimiert wird. Die Optimierungen, mit dem der Benchmark übersetzt wird, hätten dann praktisch keinen Einfluss auf den Benchmark.

6.6 Zusammenfassung

Es wurden einige Details von der Implementierung von DyFIGA erläutert. Dabei wurde erläutert wie die Individuen umgesetzt wurden und wie der Vorgang bei der Evaluation von Individuen ist. Weiter wurde ausgeführt, wie die Notwendigkeit von weiteren Samples geprüft wird. Für FAIL* wurde die Kampagne, der Client und der Golden Run näher dargelegt. Die Kampagne ist simpel und reicht die Nachrichten nur weiter. Dagegen waren die Änderungen am Client auf Grund der notwendigen Download-Funktionalität umfangreicher. Die Probleme bei der Synchronisation der Downloads wurden erklärt und Lösungen erarbeitet. Außerdem wurde das Phänomen der nicht reproduzierbaren Ergebnisse erläutert und gelöst. Anschließend wurde erklärt, wie die Laufzeit des Golden Run deutlich reduziert werden konnte. Weiter wurde die Kommunikation zwischen den beiden Systemen erläutert. Zuletzt wurden die notwendigen Änderungen an den Benchmarks diskutiert. Damit sind die wichtigsten Implementierungsdetails dargelegt und das Gesamtsystem kann im folgenden Kapitel evaluiert werden.

[7] Implementierung übernommen von:
`https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Quicksort#C`

7 Evaluation

Im folgenden soll das entworfene System evaluiert werden. Dabei werden zuerst einige Zwischenergebnisse präsentiert, welche zu Anpassungen an dem System geführt haben. Im Anschluss werden Details zu dem finalen Durchlauf gegeben, dessen Ergebnisse danach mit den Standard-Optimierungslevels verglichen wird.

7.1 Zwischenergebnisse

Nachdem das System erstmalig fertiggestellt wurde, konnte auch ein erster Lauf gestartet werden. Dabei wurden zwölf Generationen berechnet: initiale Generation, zehn Generationen mit Selektion, Mutation und Rekombination und die letzte künstliche Generation (folgend auch Vergleichsgeneration genannt), für welche die besten bekannten Individuen selektiert werden. Jede Generation (bis auf die letzte) bestand dabei aus 100 Individuen.

Die Größe der letzten Generation sollte so gewählt, dass die doppelte Anzahl an Individuen selektiert wird, wie es Individuen gibt, bei denen in mindestens einem Benchmark kein SDC aufgetreten ist. Dass dies ein Problem ist, wurde bereits in Abschnitt 5.4 erläutert: Die EAFC-Metrik wird dadurch 0 (bzw. auf 1 gesetzt) und hat maßgeblichen Einfluss auf die Sortierung der Individuen. Um dieses Phänomen möglichst gering zu halten, wurde das genannte Verfahren zur Selektion der letzten Generation gewählt, so dass dabei möglichst durch Resampling alle Benchmarks einen SDC bekommen. Wie sich dann die zuvor besten Individuen verhalten ist dabei unklar, eventuell sind sie schlechter als die Individuen, die bereits bei allen Benchmarks einen SDC hatten. Deswegen wurde die doppelte Anzahl genommen.

Für jedes Individuum wurden initial 100 Samples gezogen, und weitere Samples wurden nur angefordert, falls der Unterschied zum Nachbarn nicht groß genug war. Dabei wurde zu der Zeit für den Fehler E noch die Standardabweichung genutzt (das ζ aus Gleichung 5.3 ist dabei 1 statt $1{,}96$).

7.1.1 Erster Testlauf

Das Ergebnis der letzten Generation, bestehend aus 582 Individuen, ist in Abbildung 7.1 zu sehen bzw. der grobe Verlauf. Aufgetragen ist das geometrische Mittel der EAFC-Metriken der einzelnen Benchmarks. Die genauen Werte spielen hierbei eine untergeordnete Rolle, wichtiger sind die sichtbaren Stufen im Verlauf der Generation. Insgesamt

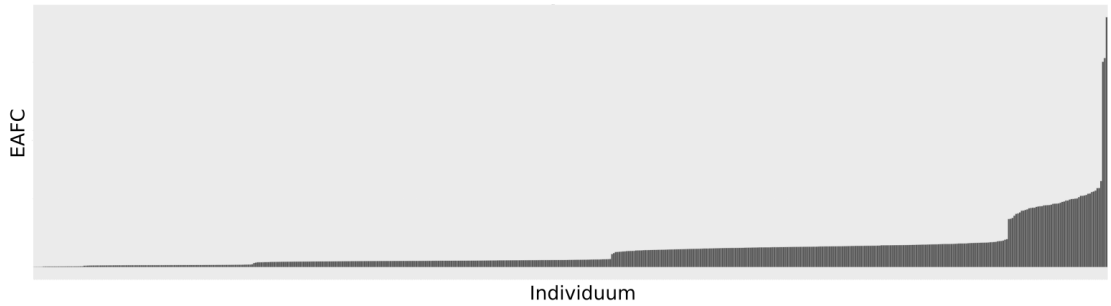

Abbildung 7.1: Ergebnis des ersten Testlaufs des Gesamtsystems. Zu sehen ist die letzte Generation, hier bestehend aus 582 Individuen. Aufgetragen ist das geometrische Mittel der EAFC-Metriken für jedes Individuum.

gibt es fünf dieser Stufen. Ausgelöst werden sie durch Benchmarks, die keinen einzigen SDC verursacht haben, und entsprechende EAFC-Werte gleich 1.

Ganz rechts im Bild sind drei Individuen zu sehen, welche bei allen Benchmarks einen SDC erkannt haben. Pro Stufe kommt ein Benchmark hinzu, der kein SDC gesehen hat (es müssen nicht die selben Benchmarks sein, nur die Anzahl ist bedeutend). Entsprechend sind ganz links in der Abbildung Individuen aufgetragen, die bei sechs verschiedenen Benchmarks nicht einen SDC erkannt haben. Außerdem sei erwähnt, dass während des gesamten Durchlaufs nicht einmal ein Resampling durchgeführt wurde.

7.1.2 Phänomen der fehlenden SDCs

Um dieses Phänomen zu verringern, wurden drei Möglichkeiten getestet: die initiale Anzahl an Samples auf 200 erhöhen, das Konfidenzniveu von $50\,\%$ (entspricht der Standardabweichung) auf $95\,\%$ erhöhen und statt nur den direkten Nachbarn die nächsten drei Nachbarn zum Vergleich zu nehmen. Dabei hat sich ergeben, dass die erhöhte Samplezahl die maximalen Benchmarks ohne SDC auf drei reduziert hat. Das erhöhte Konfidenzniveu hat zumindest zu einigen Fälle von Resampling geführt, dennoch keinen bedeutenden Unterschied bzgl. der fehlenden SDCs erreicht (es gab noch immer ein Individuum, welches sechs Benchmarks ohne SDC hatte). Die erweiterte Nachbarschaft bei der Prüfung, ob Resampling notwendig ist, hat keinerlei Unterschied gebracht. Es wurden dennoch alle drei Varianten übernommen, denn ein Nachteil ist dadurch nicht zu erwarten.

Bei einem entsprechenden Durchlauf, welcher alle drei Möglichkeiten kombiniert, gab es zwei Fälle von Resampling, und alle Individuen hatten maximal zwei Benchmarks ohne SDC. Dass hierbei kein Fall mit drei Benchmarks ohne SDC aufgetreten ist, wie bei dem Lauf mit nur 200 initialen Samples, ist vermutlich nur Zufall.

Nun war dies immer noch nicht zufrieden stellend, sollte doch möglichst erreicht werden, dass alle Benchmarks einen SDC sehen. Daher wurde das bereits im Entwurf

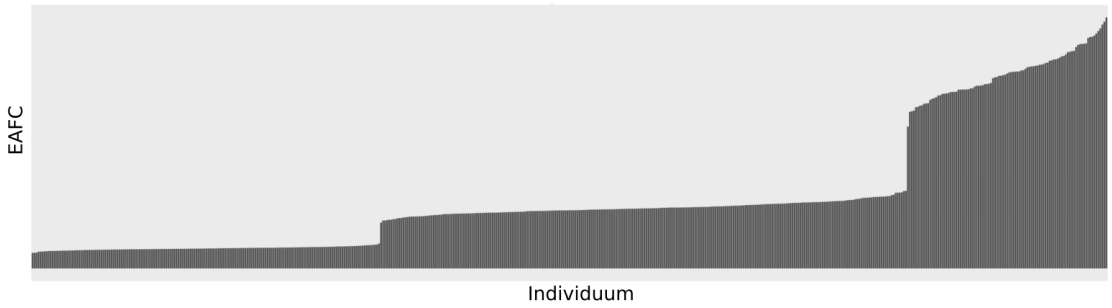

Abbildung 7.2: Ergebnis des Testlaufs, bei dem die initiale Samplezahl 200 ist, ein Konfidenzniveu von $95\,\%$ verwendet wurde und die nächsten drei Nachbarn verglichen wurden. Außerdem wurden weitere Samples gezogen, falls kein SDC erkannt wurde. Zu sehen ist die letzte Generation, hier bestehend aus 531 Individuen. Aufgetragen ist das geometrische Mittel der EAFC-Metriken für jedes Individuum.

in Abschnitt 5.4 erwähnte Resampling im Falle von fehlenden SDCs eingeführt. Das Limit dabei lag in dem Testlauf für diese Erweiterung bei 400 (doppelte Anzahl der initialen Samples).

Die letzte Generation dieses Laufs ist in Abbildung 7.2 zu sehen. Dabei sind nur noch zwei Stufen vorhanden, das Problem fällt deutlich geringer aus. An dieser Stelle soll auch einmal betrachtet werden, wie die Verteilung von Resultaten abhängig vom Benchmark ist. Dies ist zu sehen in Abbildung 7.3, wobei die OK_MARKER weggelassen wurden, da sie irrelevant für die EAFC-Metrik sind. Dabei fällt insbesondere auf, dass dijkstra und freqmine verhältnismäßig wenige SDCs produzieren. Die 218 bei dijkstra und 445 bei freqmine reichen nicht einmal aus, damit jedes der 531 Individuen mindestens einen SDC sehen könnte. Es ist daher verständlich, dass die zwei Stufen in Abbildung 7.2 entstehen.

Für den vollständigen Durchlauf wurde die maximale Anzahl an Samples, wenn ein Benchmark noch keinen SDC erzeugt hat, auf 500 (Hälfte der maximalen Samples fürs Resampling bei Ähnlichkeit) erhöht. Außerdem wurden die Vergleichsgeneration so gewählt, dass maximal die doppelte ursprüngliche Generationsgröße selektiert wird. Als letzte Anpassung wurde es so eingerichtet, dass DyFIGA beliebig lange rechnet kann und erst stoppt, wenn eine spezifische Stop-Datei angelegt wurde. Liegt diese vor, wird die aktuelle Generation beendet und eine letzte Vergleichsgeneration gestartet.

7.2 Vollständiger Durchlauf

Der Durchlauf lief über insgesamt 22 Generationen, wobei die 12. und 22. eine Vergleichsgeneration waren. Insgesamt wurden dafür $13.431.700$ Experimente durchgeführt, wobei im Durchschnitt ein Experiment $22{,}7\,\mathrm{s}$ benötigte. Damit ergibt sich eine

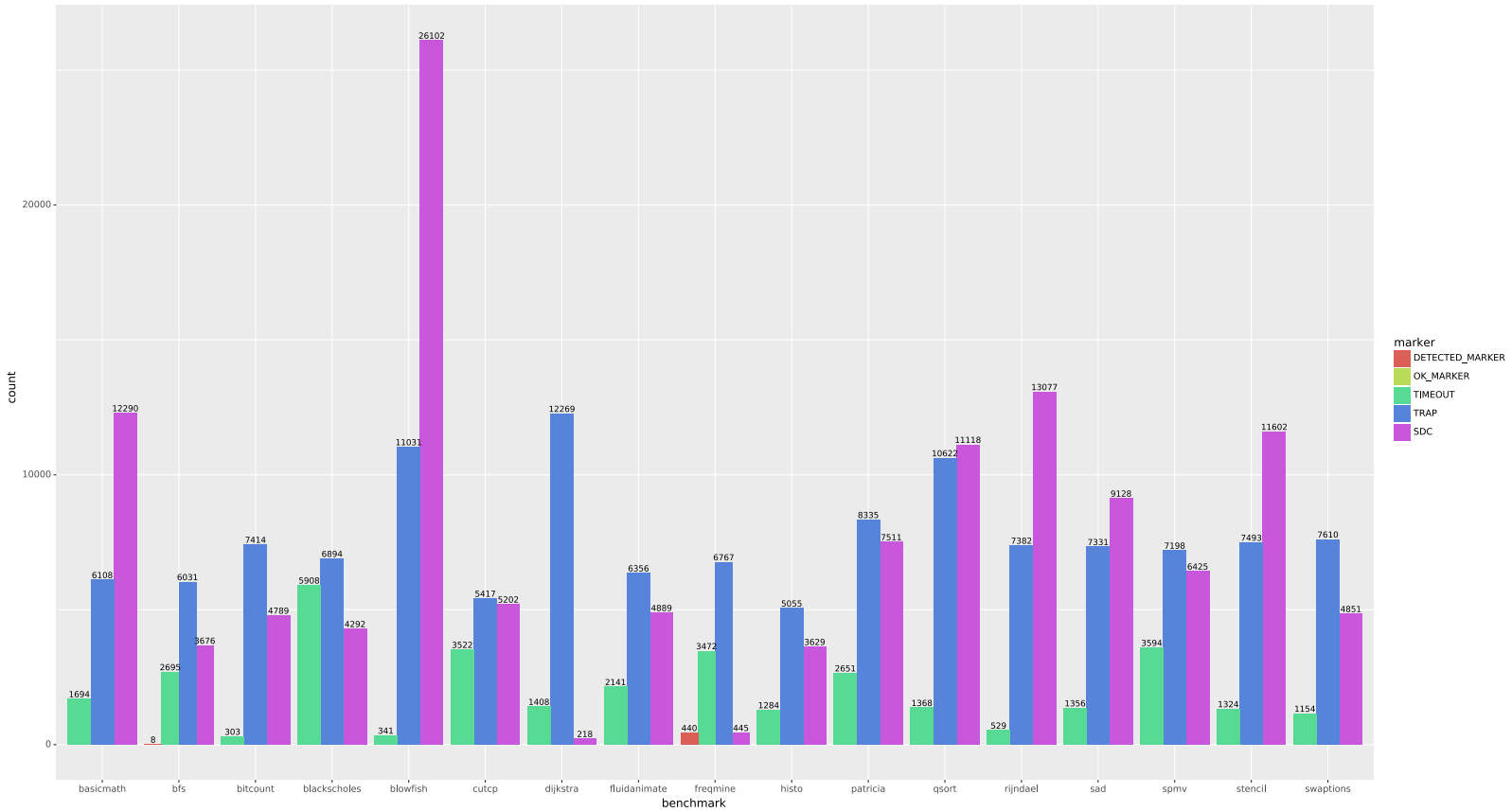

Abbildung 7.3: Verteilung der Resultat-Typen (Marker) pro Benchmark. Dabei wird die Summer über alle Experimente, die zu den Individuen der elften Generation gehören, betrachtet. Die OK_MARKER wurden weggelassen, da sie für die EAFC-Metrik irrelevant sind.

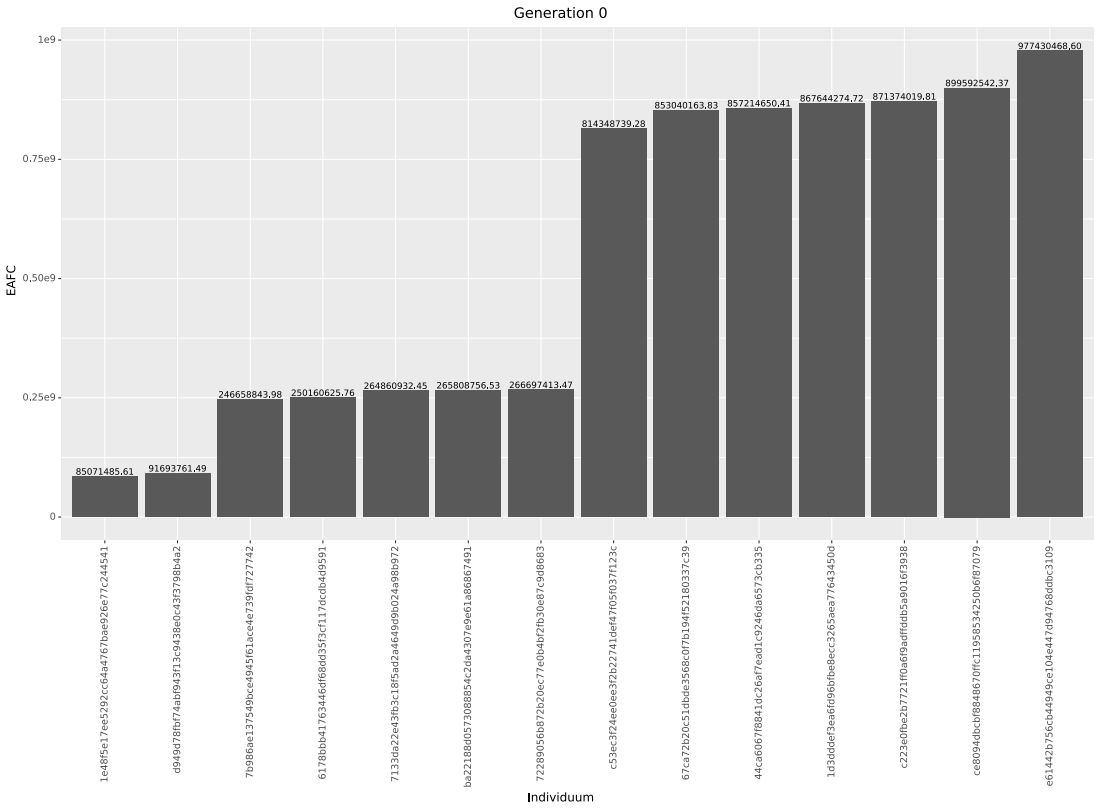

Abbildung 7.4: Die geometrischen Mittel der Individuen der initialen Generation. Von den 100 erzeugten Individuen konnten nur 14 erfolgreich compiliert werden.

aufgewendete CPU-Zeit von etwa 3.531 Tagen, das entspricht 9 Jahren und 246 Tagen bzw. etwa 9,67 Jahren. Die Zeitdifferenz zwischen Start und Ende des Durchlaufs betrug dabei nur 2 Tage, 18 Stunden und 20 Minuten. Möglich war dies, weil der Hochleistungsrechner LiDO3 der TU Dortmund genutzt wurde. Dadurch konnten bis zu 4.000 Clients gleichzeitig arbeiten (soweit es trotz der Synchronisation möglich war). Insgesamt wurden in dem Durchlauf 2.009 verschiedene Individuen erzeugt, dabei konnten 356 nicht erfolgreich compiliert werden oder der Golden Run schlug fehl. Dies scheint eine Schwäche des Compilers zu sein, da es auch in anderen Versuchen häufiger aufgetreten ist – außerdem wurde jede Optimierung einzeln getestet und dabei traten keine Fehler auf.

7.2.1 Initiale Generation

Die initiale Generation ist in Abbildung 7.4 dargestellt. Von den 100 erzeugten Individuen konnten nur 14 erfolgreich erzeugt werden – prinzipiell führt dies zu einer geringeren Vielfalt über die Dauer des gesamten Algorithmus, doch auf Grund der hohen Mutations- und Rekombinationswahrscheinlichkeiten dürfte sich dieses Pro-

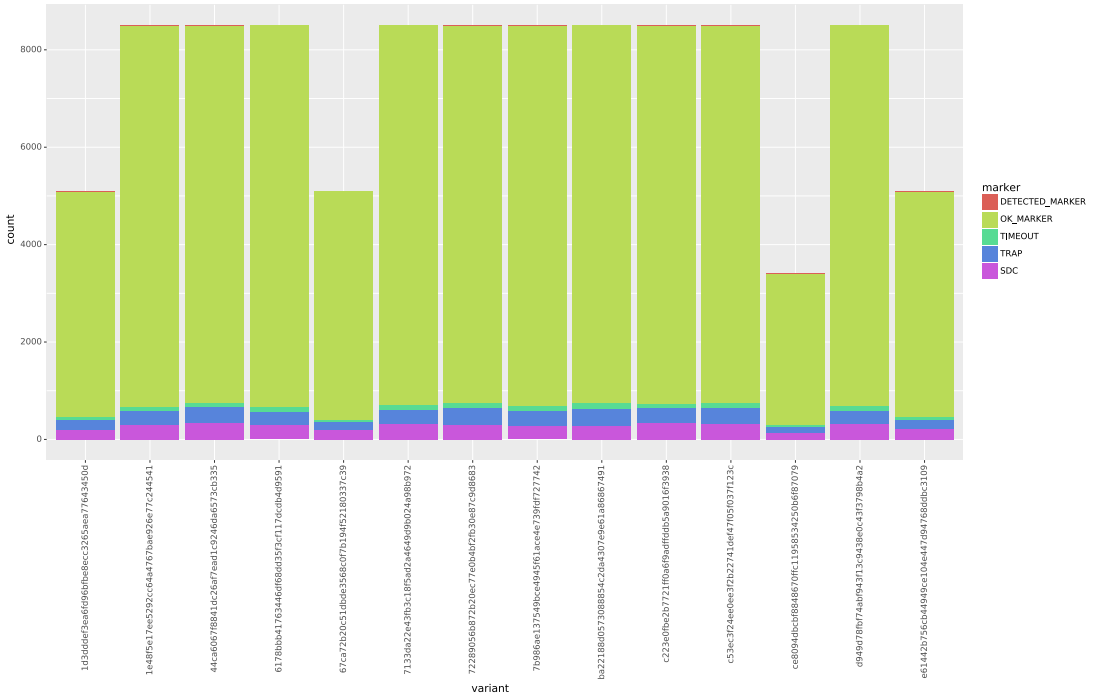

Abbildung 7.5: Gesamtzahl der verschiedenen Resultate (marker) pro Individuum. Dafür wurden alle Ergebnisse der verschiedenen Benchmarks aufsummiert.

blem in Grenzen halten. Wie zuvor sind zwei Abstufungen vorhanden, eine zwischen dem zweiten und dritten Individuum, die zweite zwischen dem siebten und achten – trotz der 500 Samples. Es gab aber außerdem vier Individuen, für die nur 200 bzw. 300 Stichproben gezogen wurden. Dieses lässt sich aus Abbildung 7.5 ableiten. Dort ist die Anzahl der Ergebnisse pro Individuum nach den Resultattypen aufgeschlüsselt (als Summe über alle Benchmarks). Teilt man die Gesamtzahl der Ergebnisse durch die 17 Benchmarks, so erhält man die Anzahl, wie oft ein einzelner Benchmark gemessen wurde.

Etwas verwunderlich dabei kann sein, dass kein Resampling für z. B. die beiden ersten Individuen erforderlich ist. Die Einzelwerte für die Benchmarks der beiden Individuen sind in Abbildung 7.6 zu sehen. Einerseits erkennt man, dass in beiden Fällen kein SDC bei dijkstra und freqmine aufgetreten ist. Andererseits sind auch Unterschiede bei den Werten pro Benchmark festzustellen. Da die Benchmarks einzeln verglichen werden und der kombinierte Wert genutzt wird, um über ein Resampling zu entscheiden, genügt es, dass die einzelnen Benchmarks unterschiedlich genug sind.

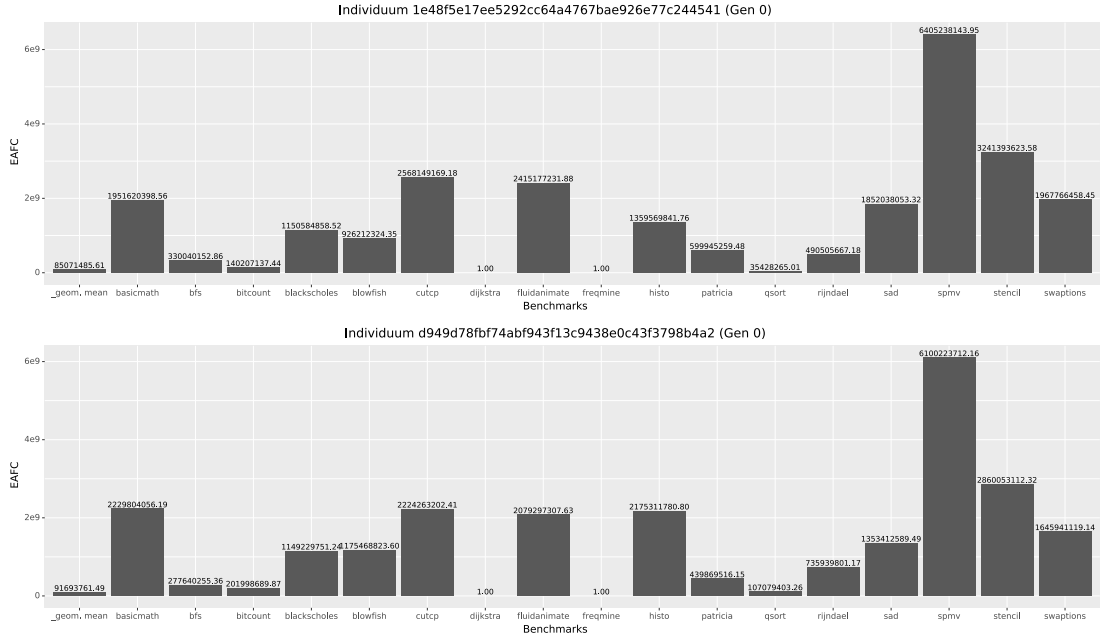

Abbildung 7.6: Die beiden besten Individuen der ersten Generation, aufgeschlüsselt nach den einzelnen EAFC-Werten der Benchmarks.

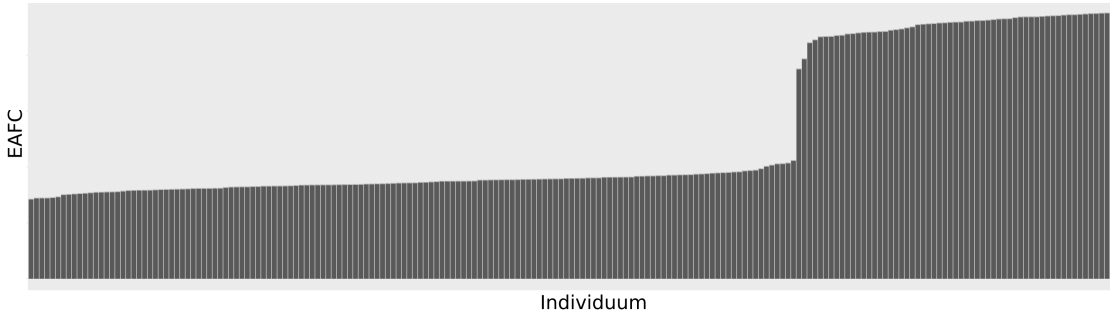

Abbildung 7.7: Die erste Vergleichsgeneration (12. Generation) des vollständigen Durchlaufs.

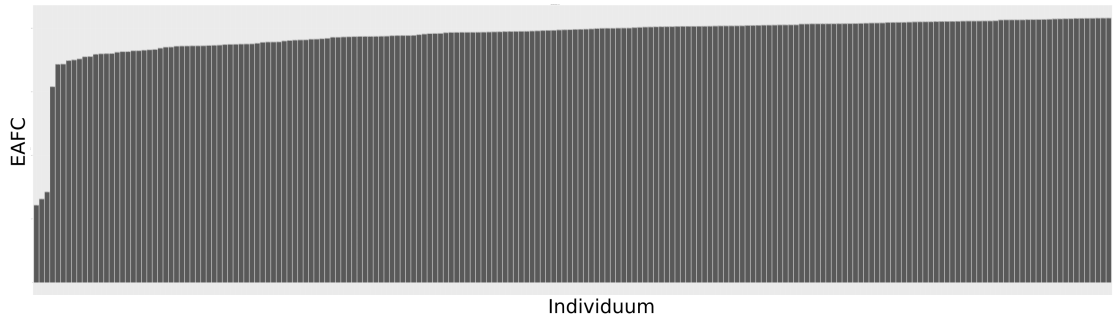

Abbildung 7.8: Letzte Generation des vollständigen Durchlaufs.

7.2.2 Vergleichsgenerationen

Mit der zwölften Generation, die erste Vergleichsgeneration, sieht das Bild aller Individuen der Generation bereits anders aus. Wie in Abbildung 7.7 zu sehen ist, gibt es nur noch eine Stufe (ein oder zwei Benchmarks haben kein SDC). Aber im gesamten Verlauf ist ein Gefälle zu erkennen, es gibt daher relevante Unterschiede zwischen den einzelnen Individuen. Jedoch gilt auch hier, dass kein Resampling für nötig erachtet wurde.

Das gleiche Verhalten bleibt bis zur letzten Generation bestehen. Es gibt kein Resampling auf Grund von fehlendem Unterschied zwischen zwei Individuen. Jedoch finden sich in der letzten Generation drei Individuen, bei denen drei Benchmarks keinen SDC produziert haben, wie in Abbildung 7.8 an dem deutlich niedrigeren EAFC-Wert zu erkennen ist. Die restlichen Individuen haben alle bei zwei Benchmarks keinen SDC. Als Referenz zur vorherigen Vergleichsgeneration: das damalige erste Individuum hat nun den zehnten Rang eingenommen. Nur neun bessere Individuen wurden in den neun dazwischen liegenden Generationen gefunden.

Die Verteilung der Resultate pro Benchmark für die Individuen der letzten Generation ist in Abbildung 7.9 dargestellt. Es fällt ganz besonders auf, dass kein Individuum bei dijkstra einen SDC erzeugt hat, dafür hat der Benchmark ein verhältnismäßig großes TRAP-Vorkommen. Der Benchmark freqmine hat mit fünf Auftreten von SDC praktisch auch keine Vorkommen, vor allem wenn man es mit der Gesamtzahl der Experimente, welche sich auf 100.000 beläuft, vergleicht. Es zeigt sich auch ganz deutlich, dass blowfish sehr anfällig ist mit der höchsten Anzahl von SDCs.

Bisher unerwähnt geblieben sind die DETECTED_MARKER. Diese dürften theoretisch nicht auftreten, da keine Maßnahmen zur Detektion implementiert wurden. Es kann jedoch in sehr seltenen Fällen vorkommen, dass ein Funktionsaufruf gerade so manipuliert wird, dass statt der eigentlichen Methode die DETECTED-Methode aufrufen wird und es so als DETECTED_MARKER gewertet wird. Warum gerade bei freqmine sich eine Häufung dieses Resultats findet, konnte leider nicht mehr näher untersucht werden.

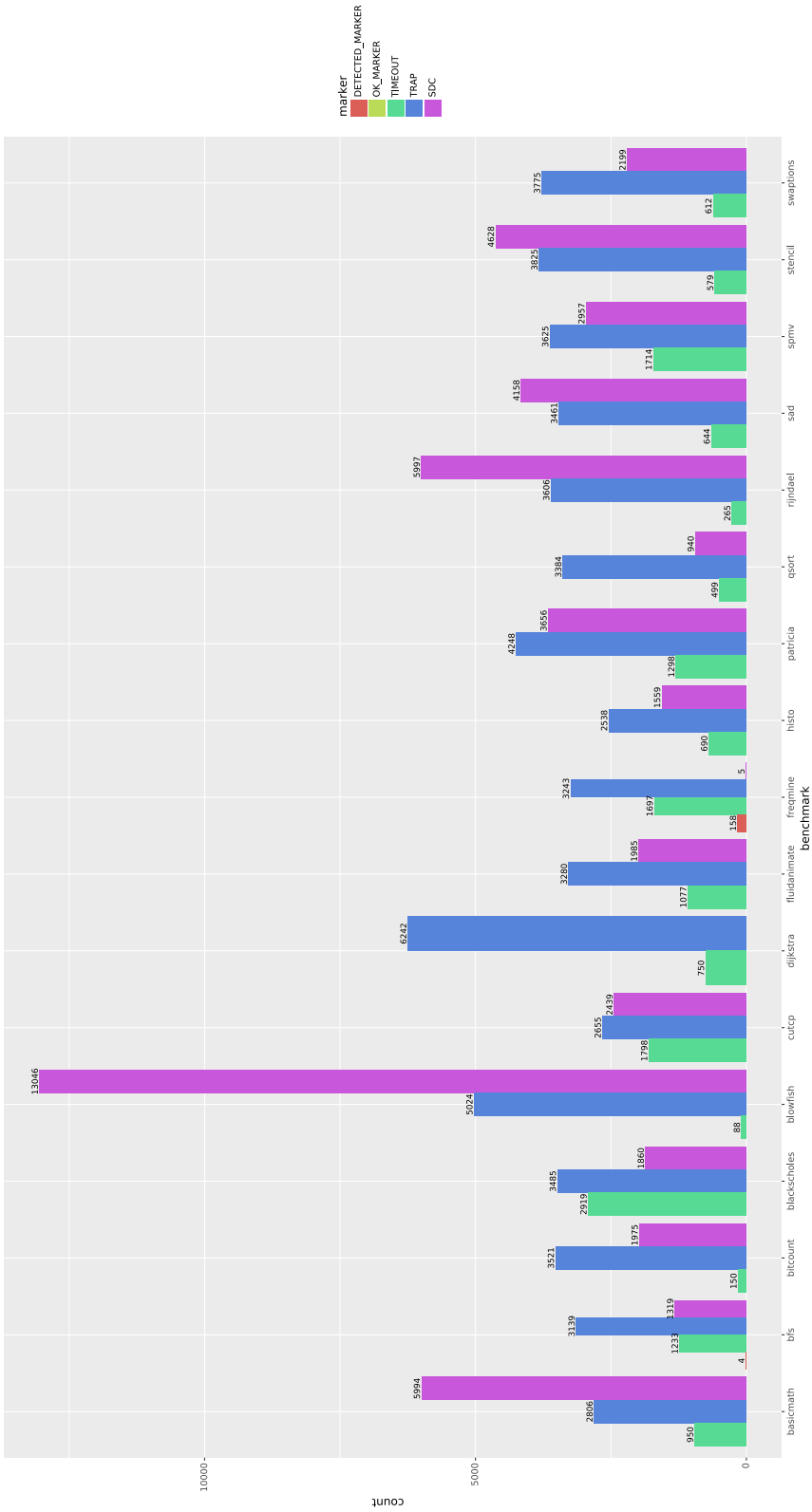

Abbildung 7.9: Verteilung der Resultat-Typen (Marker) pro Benchmark der letzten Generation des vollständigen Durchlaufs. Zur besseren Lesbarkeit wurden die OK_Marker nicht aufgetragen.

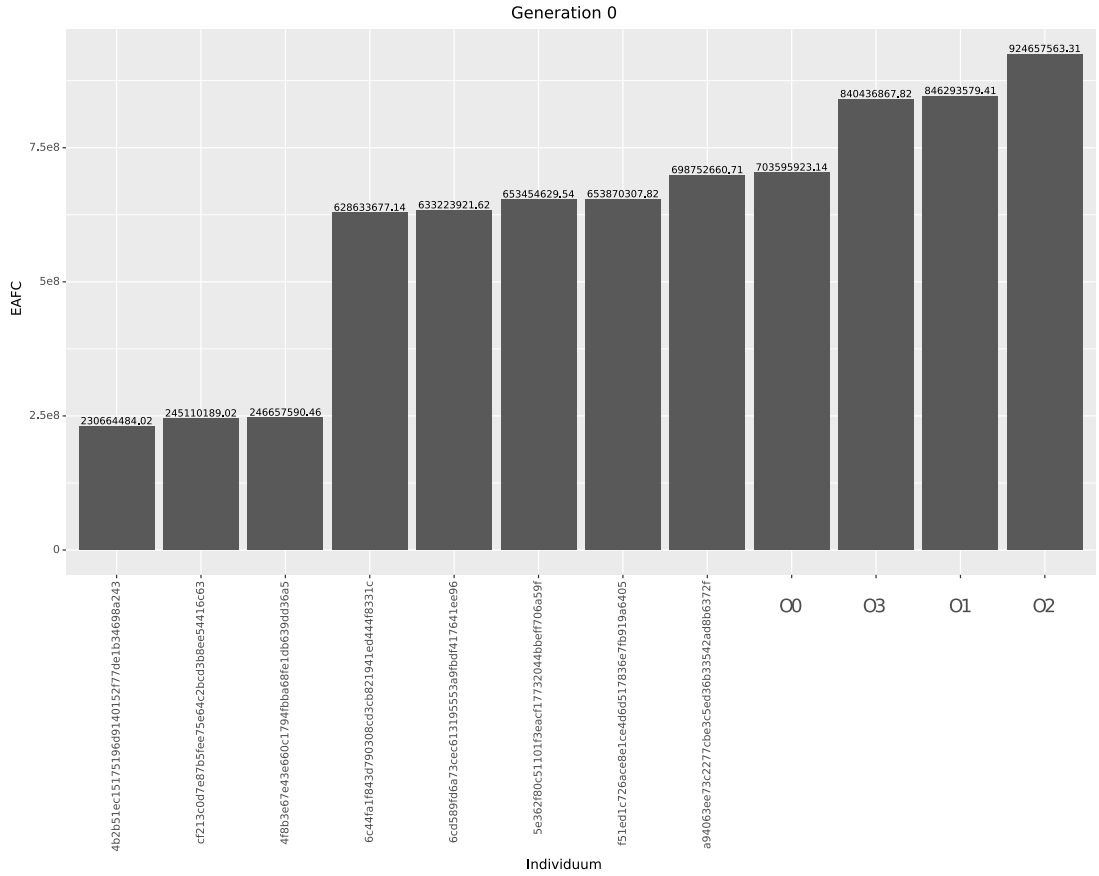

Abbildung 7.10: Vergleich der besten Individuen mit den Standardlevel.

7.3 Vergleich mit den Standardlevel

Als letztes sollen die besten Individuen mit den Standard-Optimierungslevels verglichen werden. Dafür wurden alle verfügbaren Standardlevel `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast` und `-Og` mit den acht besten Individuen des vollständigen Durchlaufs evaluiert. Es wurden acht gewählt, da vier Individuen deutlich besser sind als der Rest der Generation, wie in Abbildung 7.8 zu sehen ist, und diese Zahl wurde verdoppelt. So ergeben sich insgesamt 15 Individuen, die miteinander verglichen werden.

Jedoch ergab sich bei `-Ofast` ein Fehler beim compilieren von patricia, bei `-Og` warf der Golden Run von cutcp einen TRAP. Das Compilieren von `-Os` hat zu lange gedauert und wurde durch den Timeout beendet. Daher konnten nur die Level `-O0`, `-O1`, `-O2`, `-O3` evaluiert werden. Hierfür wurden für alle Individuen 5.000 (neue) Samples gezogen, um eine belastbarere Grundlage als nur wenige hunderte zu haben. Außerdem war es hier möglich, da die Anzahl der Individuen streng begrenzt war.

Es wurde für diese Evaluation DyFIGA so abgeändert, dass die 15 Individuen fest vorgeschrieben waren und als Generation 0 (initiale Generation) fungierten. Das Er-

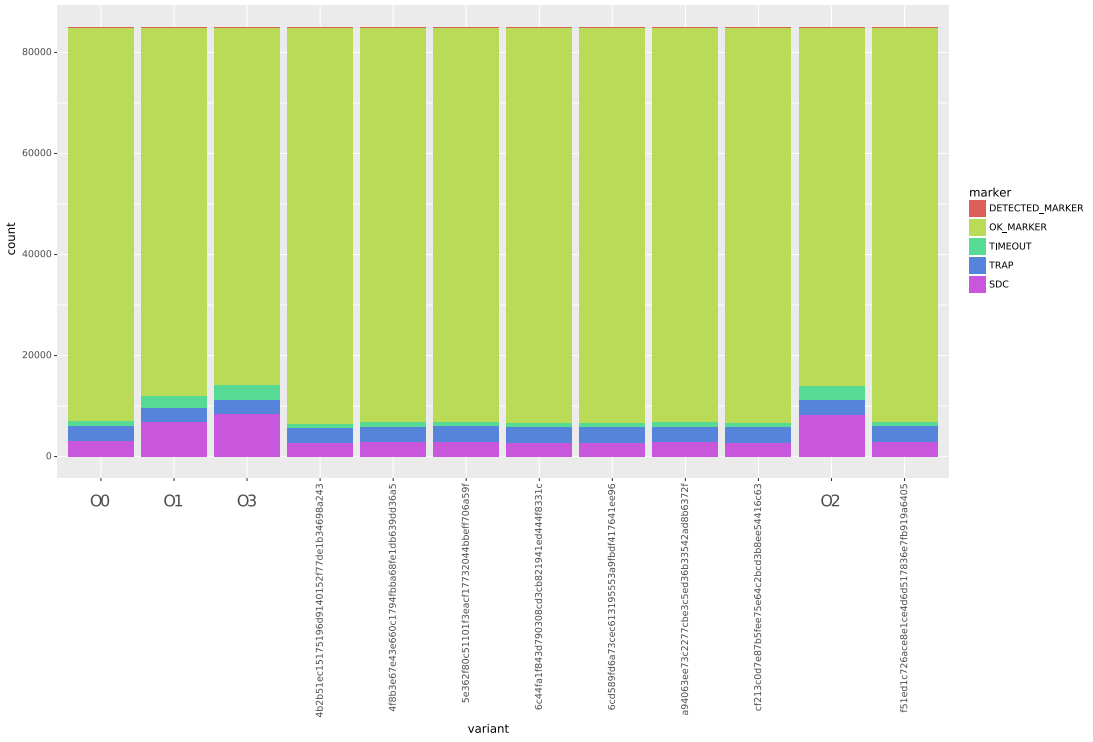

Abbildung 7.11: Vergleich der verschiedenen Resultate pro Individuum.

gebnis der Fitness-Werte ist in Abbildung 7.10 zu sehen. Dort zeigt sich eindeutig, dass alle gefunden Individuen besser als jedes der Standardlevel ist. Offenbar ist `-O0` das toleranteste der Standardlevel gegenüber Fehlern. Auch wenn es nicht genau die drei besten Individuen von zuvor sind, so haben es doch drei erreicht, dass selbst bei 5.000 Stichproben kein SDC bei dijkstra auftrat. Die Optimierungen schaffen es augenscheinlich die Angriffsfläche von dijkstra deutlich zu verringern, so dass SDCs sehr unwahrscheinlich werden.

Hier soll außerdem betrachtet werden, wie häufig die verschiedenen Resultate pro Individuum aufgetreten sind. Dieses ist in Abbildung 7.11 dargestellt. Halten alle gefundenen Individuen und `-O0` ein relativ niedriges Niveau an SDCs, ist die Rate bei den Stufen `-O1`, `-O2`, `-O3` deutlich höher.

Als letztes soll die relative Verbesserung beurteilt werden: Betrachtete man das beste Individuum im Verhältnis zu `-O0`, so ergibt sich eine Verbesserung von $67\,\%$. Das scheint jedoch etwas unwahrscheinlich und hängt maßgeblich damit zusammen, dass dieses Individuum keinen SDC bei dijkstra hatte. Deshalb wird hier das viertbeste Individuum mit `-O0` vergleichen: Dabei erhält man eine relative Verbesserung von ca. $10\,\%$.

Der vorgestellte Entwurf und seine Umsetzung erreichen daher das Ziel, bessere Optimierungsmengen als die Standard-Optimierungslevel zu finden. Die verschiedenen

Mengen der gefundenen Lösungen sind in Tabelle 7.1 zu finden. Sie wurden dabei so angegeben, dass das nächste Standard-Optimierungslevel angegeben wird, dazu die Optimierungen, welche (vom Standardlevel) ausgeschaltet werden müssen, und solche, die noch zusätzlich hinzugefügt werden müssen. Es hat sich aber auch gezeigt, dass einigen Stellen noch Verbesserungsbedarf besteht. Diese Stellen werden nach der folgenden Zusammenfassung im Ausblick näher erläutert.

Hash	String von Optimierungen
4b2b	-O3 -fno-align-functions -fno-align-loops -fno-caller-saves -fno-crossjumping -fno-cse-skip-blocks -fno-dce -fno-delayed-branch -fno-devirtualize -fno-devirtualize-speculatively -fno-dse -fno-expensive-optimizations -fno-gcse -fno-gcse-lm -fno-hoist-adjacent-loads -fno-if-conversion -fno-inline-functions -fno-ipa-profile -fno-ipa-ra -fno-ipa-reference -fno-isolate-erroneous-paths-dereference -fno-move-loop-invariants -fno-optimize-sibling-calls -fno-rerun-cse-after-loop -fno-schedule-insns -fno-schedule-insns2 -fno-shrink-wrap -fno-tree-bit-ccp -fno-tree-ccp -fno-tree-dominator-opts -fno-tree-dse -fno-tree-forwprop -fno-tree-fre -fno-tree-loop-vectorize -fno-tree-partial-pre -fno-tree-pre -fno-tree-pta -fno-tree-sink -fno-tree-slsr -fno-tree-switch-conversion -fno-tree-vrp -fno-unit-at-a-time -fno-unswitch-loops -fconserve-stack -fdata-sections -fdevirtualize-at-ltrans -ffloat-store -ffp-contract=fast -fgcse-sm -finline-limit=0 -fipa-cp-alignment -fipa-pta -fira-algorithm=priority -fira-loop-pressure -fira-region=all -flive-range-shrinkage -flto-compression-level=3 -flto-partition=max -fmodulo-sched -fprefetch-loop-arrays -freciprocal-math -frename-registers -freschedule-modulo-scheduled-loops -fsched-dep-count-heuristic -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns -fsched-stalled-insns-dep -fsched2-use-superblocks -fsection-anchors -fsel-sched-pipelining-outer-loops -fselective-scheduling2 -fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller -ftracer -ftree-loop-distribution -ftree-loop-if-convert -ftree-loop-im -ftree-loop-ivcanon -ftree-loop-optimize -ftree-parallelize-loops=5 -ftree-vectorize -funroll-loops -funsafe-loop-optimizations -funsafe-math-optimizations -fvariable-expansion-in-unroller -fweb
4f8b	-O3 -fno-align-functions -fno-align-jumps -fno-auto-inc-dec -fno-caller-saves -fno-compare-elim -fno-crossjumping -fno-cse-skip-blocks -fno-delayed-branch -fno-devirtualize-speculatively -fno-dse -fno-forward-propagate -fno-gcse -fno-gcse-after-reload -fno-if-conversion2 -fno-indirect-inlining -fno-inline-functions-called-once -fno-inline-small-functions -fno-ipa-icf -fno-ipa-profile -fno-ipa-ra -fno-ipa-reference -fno-lra-remat -fno-merge-constants -fno-peel-loops -fno-predictive-commoning -fno-reorder-blocks-algorithm=stc -fno-reorder-blocks-and-partition -fno-rerun-cse-after-loop -fno-schedule-insns -fno-split-wide-types -fno-strict-aliasing -fno-strict-overflow -fno-tree-ch -fno-tree-copy-prop -fno-tree-dominator-opts -fno-tree-dse -fno-tree-forwprop -fno-tree-fre -fno-tree-loop-vectorize -fno-tree-partial-pre -fno-tree-phiprop -fno-tree-slsr -fno-tree-sra -fno-tree-tail-merge -faggressive-loop-optimizations -fassociative-math -fbranch-probabilities -fbranch-target-load-optimize -fconserve-stack -fcx-limited-range -fdevirtualize-at-ltrans -ffinite-math-only -ffp-contract=fast -finline-limit=6 -fipa-pta -fira-hoist-pressure -fira-loop-pressure -fira-region=all -fisolate-erroneous-paths-attribute -fkeep-static-consts -flto-compression-level=4 -flto-partition=1to1 -fmerge-all-constants -fmodulo-sched-allow-regmoves -fprefetch-loop-arrays -freorder-blocks-algorithm=simple -fsched-critical-path-heuristic -fsched-dep-count-heuristic -fsched-spec-insn-heuristic -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns -fschedule-fusion -fsel-sched-pipelining-outer-loops -fselective-scheduling -fselective-scheduling2 -fsemantic-interposition -fsingle-precision-constant -fsplit-ivs-in-unroller -ftree-loop-distribution -ftree-loop-if-convert-stores -ftree-loop-optimize -ftree-parallelize-loops=10 -funconstrained-commons -funroll-loops -funsafe-loop-optimizations -funsafe-math-optimizations -fuse-linker-plugin -fvariable-expansion-in-unroller
5e36	-O1 -fno-dce -fno-delayed-branch -fno-dse -fno-if-conversion -fno-if-conversion2 -fno-ipa-profile -fno-ipa-reference -fno-move-loop-invariants -fno-shrink-wrap -fno-ssa-phiopt -fno-tree-ccp -fno-tree-ch -fno-tree-coalesce-vars -fno-tree-copy-prop -fno-tree-dse -fno-tree-forwprop -fno-tree-fre -fno-tree-phiprop -falign-labels -fcaller-saves -fconserve-stack -fcse-follow-jumps -fdata-sections -fdelete-null-pointer-checks -fdevirtualize-at-ltrans -ffp-contract=fast -fgcse-sm -findirect-inlining -finline-limit=0 -finline-small-functions -fipa-cp -fipa-cp-alignment -fipa-cp-clone -fipa-icf -fipa-pta -fipa-sra -fira-algorithm=priority -fira-loop-pressure -fira-region=all -fisolate-erroneous-paths-dereference -flive-range-shrinkage -flto-compression-level=3 -flto-partition=max -fmodulo-sched -fmodulo-sched-allow-regmoves -fpartial-inlining -fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -freciprocal-math -free -freorder-functions -freschedule-modulo-scheduled-loops -fsched-critical-path-heuristic -fsched-group-heuristic -fsched-pressure -fsched-rank-heuristic -fsched-spec-insn-heuristic -fsched2-use-superblocks -fsection-anchors -fsel-sched-pipelining -fselective-scheduling -fsemantic-interposition -fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller -fstrict-aliasing -fthread-jumps -ftree-builtin-call-dce -ftree-loop-if-convert -ftree-loop-ivcanon -ftree-loop-optimize -ftree-loop-vectorize -ftree-parallelize-loops=2 -ftree-partial-pre -ftree-pre -funroll-all-loops -funsafe-loop-optimizations -funswitch-loops -fuse-linker-plugin

6c44	-O3 -fno-align-labels -fno-crossjumping -fno-cse-follow-jumps -fno-dce -fno-devirtualize -fno-dse -fno-forward-propagate -fno-gcse -fno-if-conversion -fno-indirect-inlining -fno-inline-functions-called-once -fno-inline-small-functions -fno-ipa-icf -fno-ipa-profile -fno-ipa-pure-const -fno-ipa-ra -fno-ipa-reference -fno-lra-remat -fno-merge-constants -fno-predictive-commoning -fno-reorder-blocks-algorithm=stc -fno-rerun-cse-after-loop -fno-schedule-insns -fno-split-paths -fno-split-wide-types -fno-strict-aliasing -fno-strict-overflow -fno-tree-copy-prop -fno-tree-dce -fno-tree-dominator-opts -fno-tree-dse -fno-tree-forwprop -fno-tree-loop-distribute-patterns -fno-tree-loop-vectorize -fno-tree-slsr -fno-tree-switch-conversion -fno-tree-tail-merge -fno-tree-vrp -fno-unit-at-a-time -fno-vect-cost-model -fassociative-math -fbranch-probabilities -fbranch-target-load-optimize -fbtr-bb-exclusive -fconserve-stack -fearly-inlining -ffat-lto-objects -ffinite-math-only -ffloat-store -finline-limit=2 -fipa-cp-alignment -fira-region=one -fisolate-erroneous-paths-attribute -fkeep-static-consts -flive-range-shrinkage -flto-compression-level=4 -flto-partition=1to1 -fprefetch-loop-arrays -frounding-math -fsched-critical-path-heuristic -fsched-group-heuristic -fsched-pressure -fsched-rank-heuristic -fsched-spec-load-dangerous -fsched2-use-superblocks -fschedule-fusion -fsel-sched-pipelining-outer-loops -fselective-scheduling -fselective-scheduling2 -fsemantic-interposition -fsingle-precision-constant -ftree-loop-distribution -ftree-loop-if-convert-stores -ftree-loop-im -ftree-parallelize-loops=10 -funroll-all-loops -funsafe-loop-optimizations -fuse-linker-plugin -fvariable-expansion-in-unroller
6cd5	-O0 -falign-functions -falign-jumps -falign-labels -fauto-inc-dec -fbranch-probabilities -fbranch-target-load-optimize -fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcombine-stack-adjustments -fcompare-elim -fconserve-stack -fcprop-registers -fcse-skip-blocks -fcx-limited-range -fdelayed-branch -fdelete-null-pointer-checks -fdevirtualize -fdevirtualize-speculatively -fearly-inlining -fexpensive-optimizations -ffat-lto-objects -ffinite-math-only -ffloat-store -fgcse-lm -fhoist-adjacent-loads -findirect-inlining -finline-limit=2 -fipa-cp-alignment -fipa-cp-clone -fipa-pure-const -fipa-sra -fira-algorithm=CB -fira-hoist-pressure -fira-loop-pressure -fira-region=mixed -fisolate-erroneous-paths-dereference -fkeep-inline-functions -fkeep-static-consts -fkeep-static-functions -flive-range-shrinkage -flto-compression-level=3 -flto-partition=max -fmerge-all-constants -fmodulo-sched -fpredictive-commoning -freciprocal-math -frename-registers -freorder-blocks-algorithm=stc -freorder-functions -freschedule-modulo-scheduled-loops -fsched-critical-path-heuristic -fsched-group-heuristic -fsched-pressure -fsched-spec-load-dangerous -fsched-stalled-insns-dep -fsched2-use-superblocks -fschedule-fusion -fschedule-insns2 -fsel-sched-pipelining-outer-loops -fselective-scheduling -fselective-scheduling2 -fsemantic-interposition -fshrink-wrap -fsingle-precision-constant -fsplit-ivs-in-unroller -fsplit-paths -fssa-backprop -fssa-phiopt -fthread-jumps -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-dce -ftree-dominator-opts -ftree-fre -ftree-loop-distribute-patterns -ftree-loop-distribution -ftree-loop-if-convert-stores -ftree-loop-im -ftree-loop-optimize -ftree-parallelize-loops=10 -ftree-sink -ftree-switch-conversion -funit-at-a-time -funroll-all-loops -funroll-loops -funsafe-math-optimizations -funswitch-loops -fuse-linker-plugin -fvariable-expansion-in-unroller -fvect-cost-model -fweb
a940	-O1 -fno-dce -fno-delayed-branch -fno-dse -fno-if-conversion -fno-if-conversion2 -fno-ipa-profile -fno-ipa-reference -fno-move-loop-invariants -fno-shrink-wrap -fno-split-wide-types -fno-tree-ccp -fno-tree-ch -fno-tree-coalesce-vars -fno-tree-copy-prop -fno-tree-dse -fno-tree-forwprop -fno-tree-fre -fno-tree-phiprop -falign-labels -fcaller-saves -fconserve-stack -fcse-follow-jumps -fdata-sections -fdelete-null-pointer-checks -fdevirtualize-at-ltrans -ffp-contract=fast -fgcse-sm -findirect-inlining -finline-limit=0 -finline-small-functions -fipa-cp -fipa-cp-alignment -fipa-cp-clone -fipa-icf -fipa-pta -fipa-sra -fira-algorithm=priority -fira-loop-pressure -fira-region=all -fisolate-erroneous-paths-dereference -flive-range-shrinkage -flto-compression-level=3 -flto-partition=max -fmodulo-sched -fmodulo-sched-allow-regmoves -fpartial-inlining -fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -freciprocal-math -free -freorder-functions -fsched-dep-count-heuristic -fsched-group-heuristic -fsched-pressure -fsched-rank-heuristic -fsched-spec-insn-heuristic -fsched-spec-load-dangerous -fsched-stalled-insns -fsched2-use-superblocks -fschedule-insns2 -fsection-anchors -fselective-scheduling -fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller -fstrict-aliasing -fthread-jumps -ftree-loop-distribute-patterns -ftree-loop-if-convert -ftree-loop-if-convert-stores -ftree-loop-ivcanon -ftree-loop-optimize -ftree-loop-vectorize -ftree-parallelize-loops=2 -ftree-partial-pre -ftree-pre -funroll-all-loops -funroll-loops -funsafe-math-optimizations -funswitch-loops -fuse-linker-plugin -fvariable-expansion-in-unroller -fvect-cost-model

cf21	-O1 -fno-auto-inc-dec -fno-combine-stack-adjustments -fno-dce -fno-delayed-branch -fno-dse -fno-inline-functions-called-once -fno-merge-constants -fno-move-loop-invariants -fno-ssa-backprop -fno-ssa-phiopt -fno-tree-coalesce-vars -fno-tree-dse -fno-tree-forwprop -fno-tree-fre -fno-tree-phiprop -fno-tree-pta -fno-tree-slsr -fno-tree-sra -faggressive-loop-optimizations -falign-jumps -fassociative-math -fbranch-probabilities -fbranch-target-load-optimize -fcaller-saves -fconserve-stack -fcse-follow-jumps -fcse-skip-blocks -fcx-fortran-rules -fcx-limited-range -fdata-sections -fdevirtualize -fdevirtualize-at-ltrans -fdevirtualize-speculatively -ffloat-store -ffp-contract=off -fgcse -fgcse-las -fgcse-sm -fhoist-adjacent-loads -finline-functions -finline-limit=7 -finline-small-functions -fipa-cp-alignment -fipa-cp-clone -fipa-ra -fipa-sra -fira-algorithm=CB -fira-hoist-pressure -fira-loop-pressure -fira-region=mixed -fisolate-erroneous-paths-dereference -fkeep-static-consts -fkeep-static-functions -flto-compression-level=3 -flto-partition=one -fmerge-all-constants -fmodulo-sched -fpeel-loops -fprefetch-loop-arrays -frename-registers -freorder-blocks-algorithm=simple -freorder-blocks-and-partition -freorder-functions -freschedule-modulo-scheduled-loops -frounding-math -fsched-critical-path-heuristic -fsched-group-heuristic -fsched-last-insn-heuristic -fsched-spec-insn-heuristic -fsched-spec-load -fsched-stalled-insns-dep -fsched2-use-superblocks -fschedule-fusion -fschedule-insns -fsel-sched-pipelining -fsemantic-interposition -fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller -fstrict-overflow -ftracer -ftree-loop-distribute-patterns -ftree-loop-distribution -ftree-loop-if-convert -ftree-loop-if-convert-stores -ftree-loop-ivcanon -ftree-parallelize-loops=5 -ftree-reassoc -ftree-tail-merge -ftree-vectorize -ftree-vrp -funsafe-loop-optimizations -fvariable-expansion-in-unroller -fvect-cost-model -fvpt
f51e	-O2 -fno-cprop-registers -fno-crossjumping -fno-cse-skip-blocks -fno-dce -fno-delayed-branch -fno-devirtualize -fno-devirtualize-speculatively -fno-dse -fno-expensive-optimizations -fno-gcse -fno-hoist-adjacent-loads -fno-if-conversion -fno-if-conversion2 -fno-inline-small-functions -fno-ipa-profile -fno-ipa-ra -fno-ipa-reference -fno-lra-remat -fno-move-loop-invariants -fno-optimize-sibling-calls -fno-reorder-blocks-algorithm=stc -fno-reorder-blocks-and-partition -fno-schedule-insns -fno-shrink-wrap -fno-ssa-backprop -fno-tree-builtin-call-dce -fno-tree-coalesce-vars -fno-tree-dse -fno-tree-forwprop -fno-tree-fre -fno-tree-phiprop -fno-tree-pre -fno-tree-pta -fno-tree-slsr -fno-tree-sra -fno-tree-switch-conversion -faggressive-loop-optimizations -fassociative-math -fconserve-stack -fdata-sections -fdevirtualize-at-ltrans -ffp-contract=fast -fgcse-sm -finline-limit=0 -fipa-cp-alignment -fipa-cp-clone -fipa-pta -fira-algorithm=priority -fira-loop-pressure -fira-region=one -flive-range-shrinkage -flto-compression-level=3 -flto-partition=max -fmodulo-sched -fmodulo-sched-allow-regmoves -fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -freciprocal-math -free -fsched-dep-count-heuristic -fsched-pressure -fsched-rank-heuristic -fsched-spec-insn-heuristic -fsched-spec-load-dangerous -fsched-stalled-insns -fsched2-use-superblocks -fsection-anchors -fselective-scheduling -fsingle-precision-constant -fsplit-ivs-in-unroller -fstdarg-opt -ftracer -ftree-loop-distribute-patterns -ftree-loop-distribution -ftree-loop-if-convert -ftree-loop-if-convert-stores -ftree-loop-ivcanon -ftree-parallelize-loops=5 -ftree-reassoc -ftree-vectorize -funsafe-loop-optimizations -fvariable-expansion-in-unroller -fvect-cost-model -fvpt

Tabelle 7.1: Auflistung der Optimierungsmengen für die gefundenen Individuen. Der Hash wurde dabei auf die ersten vier Zeichen gekürzt, welches jedes Individuum dennoch eindeutig identifiziert. Die Individuen wurden relativ zu dem nächsten Standard-Optimierungslevel angegeben, um Gemeinsamkeiten besser erkennbar zu machen.

8 Zusammenfassung und Ausblick

Zuerst wird eine Zusammenfassung der Arbeit gegeben, danach werden im Ausblick weitere mögliche Arbeiten an diesem Konzept bzw. in diesem Feld vorgestellt.

8.1 Zusammenfassung

Ziel der Arbeit war es ein Verfahren zu entwickeln, mit dem Optimierungsmengen gefunden werden können, die besser bzgl. der Fehlertoleranz sind als die Standard-Optimierungslevel. Dafür wurde DyFIGA entworfen, ein genetischer Algorithmus, welcher an mehreren Stellen spezifisch zur Lösung des Problems angepasst wurde. Zur Abbildung von parametrisierten Optimierungen wurde eine ganzzahlige Codierung der Individuen genutzt. Entsprechende Verfahren für die Mutation und Rekombination wurden entworfen, welche diese Besonderheit beachten. Für die Selektion konnte dagegen die auf der Fitness basierende Tunier-Selektion verwendet werden. Besonders auszeichnend für DyFIGA ist das dynamische Resampling. Dieses soll einerseits die Anzahl der Benchmarks verringern, welche keine SDCs erzeugen. Andererseits soll dadurch sichergestellt werden, dass die Rangfolge eindeutig ist und keine Unsicherheiten bei der Sortierung bestehen.

DyFIGA läuft in Verbindung mit FAIL*, welches für die Fehlerinjektion genutzt wird. Dabei mussten einige Änderungen vorgenommen werden: Eine eigene Kampagne wurde erstellt, diese fiel jedoch recht simpel aus. Für den Injektionsclient wurde das *generic-experiment* angepasst, dem die Funktion hinzugefügt wurde, dass Binaries zur Laufzeit heruntergeladen und anschließend verwendet werden können. Dabei wurde auch einige Arbeit auf die Synchronisation der Downloads verwendet, um Konflikte zu vermeiden und Ressourcen (Webserver und Netzwerk) zu schonen. Zur Beschleunigung des Golden Run wurde außerdem das *generic-tracing* angepasst, damit dieses nur die Laufzeit eines Benchmarks misst anstatt aller Details der Instruktionen und Speichernutzungen.

Zur Abbildung verschiedener Arbeitslasten wurden 17 Benchmarks verwendet und angepasst. Sie wurden für FAIL* kompatibel mit eCos gemacht, die Eingabe in das Binary eingebettet und die Ausgabe durch Prüfzahlen minimiert.

Zum Testen des Verfahrens wurde eine aktuelle Version (7.3.0) der GCC verwendet, speziell der `gcc` und `g++`. Diese verfügen über insgesamt 210 Optimierungen, wovon 177 DyFIGA zur Auswahl standen. Das Verfahren kann aber auch einfach auf andere Compiler angewandt werden.

In der Evaluation hat sich gezeigt, dass DyFIGA einige Schwächen hat, wovon ein Teil bereits behoben wurde. Trotzdem war das Verfahren erfolgreich, denn es wurden Optimierungsmengen gefunden, die das Ziel erfüllen. Dabei konnte eine Verbesserung von ca. $10\,\%$ sicher erreicht werden. Von den Fällen, wo ein Benchmark kein SDC erzeugt hat, wird sogar eine Verbesserung von etwa $67\,\%$ suggeriert.

8.2 Ausblick

Eine der Schwächen von DyFIGA ist, dass das Resampling praktisch nicht funktioniert hat, auch wenn der Gedanke dahinter richtig ist. Hier könnte man verbessern, dass für einzelne Benchmarks weitere Stichproben gezogen werden können und so über ein Resampling pro Benchmark entscheiden wird. Dadurch könnten auch viele Stichproben der anderen Benchmarks eingespart werden, die bereits bei 200 Samples ausreichend unterschiedlich sind, und für Benchmarks, die keinen SDC erfahren, würden automatisch deutlich mehr Samples gezogen werden.

Weiter können die Abhängigkeiten zwischen den Optimierungen betrachtet und umgesetzt werden. Es könnte neben dem Genom und dem sich daraus direkt ergebenen String von Optimierungen auch der effektive Optimierungsstring berechnet werden, welcher der kürzest mögliche String ist, der die gleichen Optimierungen aktiviert. Implizierte Optimierungen können dabei entfernt werden und ebenso Optimierungen, deren notwendigen Optimierungen nicht vorhanden sind. Damit würde sich der Raum aller möglichen Optimierungsmengen verringern und einige Individuen könnten zusammen gelegt werden.

Es könnte untersucht werden, warum die Standardlevel neben `-O1` bis `-O3` nicht übersetzt werden konnten. Auch andere Compilerfehler könnten genauer inspiziert werden, um so eine möglichst große Menge an Optimierungen zu nutzen bzw. sicher festzustellen, dass einige Kombinationen nicht im Rahmen der Fehlertoleranz genutzt werden können.

Eine weitere Möglichkeit ist, einen noch aktuelleren Compiler (GCC hat zum Abschluss dieser Arbeit eine Version 8.2 veröffentlicht) zu nutzen. Natürlich besteht die Möglichkeit auch noch mehr Benchmarks hinzuzufügen, um die Menge der möglichen Programme, auf welche die Optimierungen angewendet werden, noch besser abzudecken.

Außerdem könnte der Injektionsclient weiter überarbeitet werden. Die Synchronisation kann mit Sicherheit noch performanter gestaltet werden und trotzdem die Ressourcen schonen. Dabei wäre es optimal den gesamten Downloadprozess auszulagern, so dass je nach Bedarf die Implementierung einfach angepasst werden kann, ohne den kompletten Client umzuschreiben.

Es wäre möglich dieses entworfene Verfahren auch mit anderen Optimierungsverfahren zu vergleichen, um zu überprüfen, wie gut die erreichten Ergebnisse im Verhältnis sind. Narayanamurthy, Pattabiraman und Ripeanu haben ihrem entwickelten Verfah-

ren Simulated Annealing gegenüber gestellt. Genauso könnte hier selbiges oder ein beliebiger anderer Algorithmus zur Optimierung getestet und mit den hier gewonnen Ergebnissen verglichen werden.

Potentiell könnte nicht nur die Fehlertoleranz optimiert sondern dabei auch die Anzahl der genutzten Optimierungen möglichst gering gehalten werden. Damit wäre die Ausfilterung von Optimierungen möglich, die nur eine sehr marginale Verbesserung bzgl. der Fehlertoleranz bewirken.

Eine sinnvolle Erweiterung wäre auch, dass zwischen den Benchmarks unterschieden wird, ob diese FPU-Register nutzen oder nicht. Denn werden sie nicht genutzt, dann ist schon vor der Durchführung eines Experiments, welches in diese Register injiziert, klar, dass daraus ein OK_MARKER resultieren wird. Dies könnte noch weiter geführt werden, so dass nicht nur zwischen Allzweck- und FPU-Registern unterschieden wird, sondern die Nutzung jedes einzelnen Registers betrachtet wird. Damit wird auch die EAFC-Metrik unerlässlich, die die unterschiedliche Menge an Registern abbilden kann.

Literatur

[1] C. Bienia. „Benchmarking Modern Multiprocessors“. Diss. Princeton University, Jan. 2011.

[2] C. Blackmore, O. Ray und K. Eder. „Automatically Tuning the GCC Compiler to Optimize the Performance of Applications Running on the ARM Cortex-M3“. In: *CoRR* abs/1703.08228 (2017). arXiv: 1703.08228. URL: http://arxiv.org/abs/1703.08228.

[3] S. Borkar. „Designing reliable systems from unreliable components: the challenges of transistor variability and degradation“. In: *IEEE Micro* 25.6 (Nov. 2005), S. 10–16. ISSN: 0272-1732. DOI: 10.1109/MM.2005.110.

[4] H. Cha u. a. „A gate-level simulation environment for alpha-particle-induced transient faults“. In: *IEEE Transactions on Computers* 45.11 (Nov. 1996), S. 1248–1256. ISSN: 0018-9340. DOI: 10.1109/12.544481.

[5] M. Demertzi, M. Annavaram und M. Hall. „Analyzing the effects of compiler optimizations on application reliability“. In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2011, S. 184–193. DOI: 10.1109/IISWC.2011.6114178.

[6] M. Demertzi, M. Annavaram und M. Hall. „Analyzing the effects of compiler optimizations on application reliability“. In: *2011 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2011, S. 184–193. DOI: 10.1109/IISWC.2011.6114178.

[7] European Space Agency. *OPS-SAT*. Zugriffsdatum: 23.12.2018. Apr. 2017. URL: https://www.esa.int/Our_Activities/Operations/OPS-SAT.

[8] R. Fisher. *Statistical methods for research workers*. Edinburgh Oliver & Boyd, 1925.

[9] F.-A. Fortin u. a. „DEAP: Evolutionary Algorithms Made Easy“. In: *Journal of Machine Learning Research* 13 (Juli 2012), S. 2171–2175.

[10] T. Funke u. a. „Eigenschaften und Entwicklung von Kleinstsatelliten“. In: *Deutscher Luft- und Raumfahrtkongress 2016, Braunschweig* (2016).

[11] Google LLC. *Google Protocol Buffers*. http://code.google.com/apis/protocolbuffers/.

[12] M. R. Guthaus u. a. „MiBench: A Free, Commercially Representative Embedded Benchmark Suite“. In: *Proceedings of the IEEE International Workshop on Workload Characterization (WWC '01)*. Washington, DC, USA: IEEE Press, 2001, S. 3–14. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.15.

[13] A. Höller u. a. „Evaluation of diverse compiling for software-fault detection“. In: *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*. März 2015, S. 531–536. DOI: 10.7873/DATE.2015.0118.

[14] R. K. Iyer und D. J. Rossetti. „A Measurement-Based Model for Workload Dependence of CPU Errors“. In: *IEEE Transactions on Computers* C-35.6 (Juni 1986), S. 511–519. ISSN: 0018-9340. DOI: 10.1109/TC.1986.5009428.

[15] D. Li u. a. „Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach“. In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2013, S. 1–12. DOI: 10.1145/2503210.2503226.

[16] Q. Lu u. a. „LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults“. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. Aug. 2015, S. 11–16. DOI: 10.1109/QRS.2015.13.

[17] S. Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 9780080558325, 9780123695291.

[18] N. Narayanamurthy, K. Pattabiraman und M. Ripeanu. „Finding Compiler Optimizations that Preserve Resilience using Meta-Heuristic Search Techniques“. Entwurf von den Autoren erhalten.

[19] N. Narayanamurthy, K. Pattabiraman und M. Ripeanu. „Finding Resilience-Friendly Compiler Optimizations Using Meta-Heuristic Search Techniques“. In: *2016 12th European Dependable Computing Conference (EDCC)*. Sep. 2016, S. 1–12. DOI: 10.1109/EDCC.2016.26.

[20] A. Norouzi u. a. „An enhanced integer coded genetic algorithm to optimize PWRs“. In: *Progress in Nuclear Energy* 53.5 (2011), S. 449–456. ISSN: 0149-1970. DOI: https://doi.org/10.1016/j.pnucene.2011.03.005. URL: http://www.sciencedirect.com/science/article/pii/S0149197011000394.

[21] P. Rakshit, A. Konar und S. Das. „Noisy evolutionary optimization algorithms – A comprehensive survey“. In: *Swarm and Evolutionary Computation* 33 (2017), S. 18–45. ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2016.09.002. URL: http://www.sciencedirect.com/science/article/pii/S221065021630308X.

[22] B. Sangchoolie u. a. „A Study of the Impact of Bit-Flip Errors on Programs Compiled with Different Optimization Levels“. In: *2014 Tenth European Dependable Computing Conference*. Mai 2014, S. 146–157. DOI: 10.1109/EDCC.2014.30.

[23] H. Schirmeier, C. Borchert und O. Spinczyk. „Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors“. In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Juni 2015, S. 319–330. DOI: 10.1109/DSN.2015.44.

[24] H. Schirmeier. „Efficient fault-injection-based assessment of software-implemented hardware fault tolerance“. Diss. Jan. 2016. DOI: 10.17877/DE290R-17222.

[25] F. Siegmund, A. H. C. Ng und K. Deb. „A comparative study of dynamic resampling strategies for guided Evolutionary Multi-objective Optimization“. In: *2013 IEEE Congress on Evolutionary Computation*. Juni 2013, S. 1826–1835. DOI: 10.1109/CEC.2013.6557782.

[26] M. Srinivas und L. M. Patnaik. „Genetic algorithms: a survey“. In: *Computer* 27.6 (Juni 1994), S. 17–26. ISSN: 0018-9162. DOI: 10.1109/2.294849.

[27] R. M. Stallman und the GCC Developer Community. *Using the GNU Compiler Collection. For gcc version 7.3.0*. GNU Press, URL: https://gcc.gnu.org/onlinedocs/gcc-7.3.0/gcc.pdf (besucht am 05. 10. 2018).

[28] I. Steinke und T. C. Stocker. *Statistik. Grundlagen und Methodik*. De Gruyter Oldenburg, Nov. 2016. ISBN: 978-3-11-035389-1.

[29] J. A. Stratton u. a. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Techn. Ber. University of Illinois, März 2012.

[30] B. Sumit, P. Praveen und P. G.N. „SVM Based Decision Support System for Heart Disease Classification with Integer-Coded Genetic Algorithm to Select Critical Features“. In: *Lecture Notes in Engineering and Computer Science* 2173 (Okt. 2008).

[31] A. Syberfeldt u. a. „Evolutionary optimisation of noisy multi-objective problems using confidence-based dynamic resampling“. In: *European Journal of Operational Research* 204.3 (2010), S. 533–544. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2009.11.003. URL: http://www.sciencedirect.com/science/article/pii/S0377221709008534.

[32] J. Wei u. a. „Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults“. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Juni 2014, S. 375–382. DOI: 10.1109/DSN.2014.2.

[33] D. S. Weile und E. Michielssen. „Integer coded Pareto genetic algorithm design of constrained antenna arrays“. In: *Electronics Letters* 32.19 (Sep. 1996), S. 1744–1745. ISSN: 0013-5194. DOI: 10.1049/el:19961174.

Abbildungsverzeichnis

Tabellenverzeichnis

Eidesstattliche Versicherung
(Affidavit)

Name, Vorname
(Last name, first name)

Matrikelnr.
(Enrollment number)

Ich versichere hiermit an Eides statt, dass ich die vorliegende Bachelorarbeit/Masterarbeit* mit dem folgenden Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present Bachelor's/Master's* thesis with the following title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution.

Titel der Bachelor-/Masterarbeit*:
(Title of the Bachelor's/ Master's* thesis):

*Nichtzutreffendes bitte streichen
(Please choose the appropriate)

Ort, Datum
(Place, date)

Unterschrift
(Signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird gfls. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to €50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, section 63, subsection 5 of the North Rhine-Westphalia Higher Education Act (*Hochschulgesetz*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:**

Ort, Datum
(Place, date)

Unterschrift
(Signature)

**Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.