Feingranulare Korrektheitsprüfung des Kontrollflusses von Echtzeitsystemen

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Angewandte Informatik
der Universität Augsburg

eingereicht von

Dipl.-Inf. Julian Wolf

im Jahr 2012

Erstgutachter:	Prof. Dr. Theo Ungerer
Zweitgutachter:	Prof. Dr.-Ing. Rudi Knorr

Tag der mündlichen Prüfung: 25. Januar 2013

Kurzfassung

Sicherheit und Zuverlässigkeit sind in vielen Einsatzgebieten entscheidende Anforderungen an eingebettete Systeme. Jede Fehlfunktion kann negative, wenn nicht gar katastrophale Konsequenzen nach sich ziehen. Es genügt nicht, die funktionale Korrektheit im Vorfeld formal zu beweisen oder durch systematische Testläufe zu überprüfen. Vielmehr ist es nötig, auch jedes Fehlverhalten, das durch Umwelteinflüsse wie Strahlung oder Temperaturschwankungen hervorgerufen wird, durch speziell integrierte Mechanismen zur Laufzeit zu erkennen und eine entsprechende Fehlerbehandlung zu ermöglichen. Im Hinblick auf harte Echtzeitsysteme, in welchen eine verspätete Berechnung wertlos ist, spielt dabei neben der logischen auch die zeitliche Korrektheit der Ausführung eine essenzielle Rolle.

Ziel dieser Arbeit ist es, einen Mechanismus zur Erkennung von logischen und zeitlichen Kontrollflussfehlern in Echtzeitsystemen zu entwickeln. Durch eine besonders feingranulare Arbeitsweise soll die Fehlererkennung möglichst früh stattfinden, um eventuelle Gegenmaßnahmen rechtzeitig vor dem Überschreiten der geforderten Zeitschranken einleiten zu können. Die entworfene Technik verfolgt dazu einen hybriden Ansatz: Der Programmcode wird bereits im Vorfeld mit Kontrollpunkten, sog. Checkpoints instrumentiert, welche sowohl logische als auch temporale Informationen zum Kontrollfluss beinhalten. Zur Laufzeit wertet eine spezielle, an den Prozessor angeschlossene Hardware-Einheit diese Daten aus, um die Korrektheit der Ausführung zu prüfen. Auf diese Weise kann eine rasche Erkennung einer Vielzahl transienter Fehler erfolgen.

Neben einer detaillierten Beschreibung der entwickelten Technik präsentiert diese Arbeit auch eine Referenzimplementierung auf Basis eines echtzeitfähigen Prozessormodells. Anhand von Evaluierungen mit künstlich erzeugten Fehlerfällen kann einerseits die Wirksamkeit des Erkennungsmechanismus, andererseits dessen geringfügiger Zusatzaufwand nachgewiesen werden. Abschließend werden verschiedene Optimierungstechniken vorgestellt und evaluiert, mit deren Hilfe es möglich wird, sowohl die Erkennungsrate zu erhöhen als auch den benötigten Aufwand weiter zu reduzieren.

Abstract

Safety and reliability are key requirements in many areas of embedded systems. Any malfunction or faulty system behaviour may have negative, potentially disastrous consequences. It is not sufficient to prove the functional correctness or to perform systematic test runs before using a system under operating conditions. Rather, a specially integrated mechanism is needed to detect and potentially recover at runtime any misbehaviour, which is caused by environmental factors like radiation or overheating. In terms of hard real-time systems, in which a delayed delivery of results is worthless, such a technique must focus on both the logical and temporal correctness of the execution.

This work aims to develop a mechanism to detect logical and temporal control flow errors in real-time systems. The fine-grained operation with very low detection latency enables an immediate reaction to any misbehaviour and possibly the execution of a fall-back solution within the required deadlines. The proposed approach is a hybrid hardware-software technique: The application code is instrumented at compile-time by adding checkpoints, which contain logical and temporal information about the control flow. During runtime, a small hardware check unit connected to the core reads the instrumented data in order to verify the correctness of the application's control flow and timing behavior. By this, a rapid detection of many transient errors is possible.

Besides a detailed description of the developed technology this work presents a reference implementation for a real-time capable processor model. Based on evaluations using fault injection, both the effectiveness of the detection mechanism and its low overhead can be shown. Finally, different optimisation techniques are presented and evaluated to increase the detection rate and to further reduce the required overhead.

Vorwort

Die vorliegende Arbeit entstand in den Jahren 2007 bis 2012 während meiner Tätigkeit als wissenschaftlicher Angestellter am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme der Universität Augsburg. Zunächst war ich im Rahmen des EU-geförderten Forschungsprojekts MERASA beschäftigt, wobei es meine Aufgabe war, die Grundlagen eines echtzeitfähigen Betriebssystems für einen neu entworfenen Multicore-Prozessor zu entwickeln und zu implementieren. Dabei wurde der Fokus der Forschungen immer stärker auch auf Aspekte funktionaler Sicherheit gelegt. Im Zuge verschiedener Diskussionen und Gespräche entstand schließlich eine Idee der dynamischen Fehlererkennung, welche den ersten Grundstein der vorliegenden Arbeit gebildet hat.

Ich möchte mich an dieser Stelle herzlich bei meinem Doktorvater Prof. Dr. Theo Ungerer bedanken. Ohne seine hervorragende Betreuung, die richtungsweisenden Anregungen und Diskussionen sowie nicht zuletzt die angenehme Atmosphäre am Lehrstuhl wäre diese Arbeit nicht möglich gewesen. Weiterer Dank gilt Prof. Dr.-Ing. Rudi Knorr für die Übernahme des Zweitgutachtens. Ich danke auch Prof. Dr. Sascha Uhrig und Dr. Rafael Zalman, die einen wichtigen Beitrag zur Ideenfindung geleistet haben. Besonderer Dank geht zudem an Dr.-Ing. Jörg Mische für die Hilfe bei der Integration der Check-Einheit in den CarCore-Simulator und an Christian Bradatsch für die Unterstützung bei der VHDL-Umsetzung.

Bedanken möchte ich mich auch bei allen meinen Kollegen am Lehrstuhl für die Vielzahl an fruchtbaren Anregungen und Vorschlägen zu meiner Arbeit. Stellvertretend seien meine Kollegen Mike Gerdes, Christian Bradatsch, Sebastian Schlingmann und Michael Roth genannt, die bei Döner, Pizza oder Burger stets volle Diskussionsbereitschaft, aber auch hohen Unterhaltungswert gezeigt haben.

Zu guter Letzt danke ich meinen Eltern, die mich im Laufe der Jahre fortwährend unterstützt und motiviert haben. Meinem Vater gilt dabei auch Dank für das konsequente, wenn auch anstrengende Korrekturlesen der Arbeit.

Augsburg, im November 2012 *Julian Wolf*

Inhaltsverzeichnis

1

Einleitung

Im Automobilbereich stellen neuartige Technologien wie Steer-by-Wire oder Brake-by-Wire sowie Fahrassistenzsysteme z.B. zum automatisierten Spurwechsel und zur Abstandsüberwachung immer höhere Anforderungen an das zugrundeliegende Rechnersystem. Eine Vielzahl verschiedener Sensoren bewirkt hohe Komplexität, da die Werte innerhalb sehr kurzer Zeitintervalle ausgelesen und korrekt verarbeitet werden müssen. Erfolgt eine falsche oder verspätete Reaktion, z.B. beim Auslösen des Airbags, wird dies nicht nur negative, sondern womöglich katastrophale Konsequenzen nach sich ziehen. Ähnliche, wenn nicht gar höhere Anforderungen an Sicherheit und Korrektheit im Zeitverhalten entstehen in anderen Domänen, beispielsweise im Maschinenbau oder in der Luft- und Raumfahrtindustrie.

Komplexe Prozessoren, welche in handelsüblichen Desktop-Rechnern eingesetzt sind, können in Fahrzeugen, Flugzeugen und Maschinen aufgrund der starken Beschränkungen hinsichtlich der Energieversorgung sowie wegen der rauen Einsatzbedingungen nicht verwendet werden. Es ist nötig, je nach Anwendungsbereich eingebettete Systeme zu entwickeln, welche den hohen Anforderungen gerecht werden. Für ein höheres Maß an funktionaler Sicherheit eines Systems müssen potenzielle Fehlerquellen identifiziert werden, um geeignete Mechanismen zur Erkennung von Fehlfunktion zu entwickeln. Wichtig ist hierbei, einen Fehler frühestmöglich festzustellen, um rechtzeitig geeignete Gegenmaßnahmen, z.B. den Start eines Rückfallsystems, einleiten zu können. Auf der anderen Seite steht die Wirtschaftlichkeit im Vordergrund: Eine entscheidende Anforderung ist es, den entstehenden Mehraufwand zur Fehlerdiagnose aus Kostengründen in Grenzen zu halten.

In sicherheitskritischen Echtzeitsystemen gilt es in besonderem Maße, unvorhersehbare Verzögerungen und Latenzen a priori auszuschließen und eine zeitliche

Obergrenze für die Ausführung bestimmter Programmteile zu garantieren. Dabei ist es weniger das Ziel, die durchschnittliche Performanz zu beschleunigen, als die Ausführungsdauer für den schlechtesten Fall zu begrenzen, analysierbar zu machen und im Vorfeld zu bestimmen. Eine statische Analyse der Laufzeit, basierend auf einer Modellierung des Rechnersystems, ermöglicht die Verifikation des Zeitverhaltens, jedoch nur im Modell. Die Ausführung auf realer Hardware weicht potenziell davon ab: Transiente Fehler, also temporäre Zustandsänderungen bedingt durch Strahlung oder externe Störsignale, können das Zeitverhalten sowie das Ausführungsergebnis zur Laufzeit beeinflussen. Somit sind Prüfmechanismen *vor* der Programmausführung wirkungslos. Eine Erkennung der Fehlfunktion kann ausschließlich dynamisch zur Laufzeit erfolgen.

Verschiedene Studien haben mittels künstlich erzeugter Fehler in Rechnersystemen gezeigt, dass sich ein Großteil aller Fehlerfälle negativ auf den Kontrollfluss, d.h. die korrekte zeitliche und logische Abfolge von Programmbefehlen, auswirkt. Somit besteht eine große Notwendigkeit, diese Kontrollflussfehler zuverlässig zu erkennen, um das Sicherheitsniveau eines Systems zu steigern. Ein Mechanismus, der eine solche Korrektheitsprüfung bewerkstelligt, kann unterschiedlich umgesetzt sein, was entsprechende Vor- und Nachteile mit sich bringt. Es ist nötig, entweder eine extra Hardware-Einheit an den Prozessor anzugliedern, welche während der Laufzeit Korrektheitsprüfungen vornimmt, oder die Software durch zusätzliche Instruktionen zu erweitern, welche eine Absicherung veranlassen. Die Bewertung einer solchen Technik muss dabei nach verschiedenen Kriterien erfolgen: Im Mittelpunkt stehen die Abdeckung erkannter Fehlerfälle sowie die Latenz der Fehlererkennung. Doch ebenso wichtig sind der Aufwand und die Umsetzbarkeit des entworfenen Ansatzes.

1.1 Ziele der Arbeit

Das zentrale Ziel dieser Arbeit ist es, einen Mechanismus zur dynamischen Erkennung von Kontrollflussfehlern in sicherheitskritischen Echtzeitsystemen zu entwickeln. Entsprechend der speziellen Anforderungen an solche Systeme steht neben der logischen Korrektheitsprüfung auch die Erkennung von zeitlichen Fehlern im Vordergrund. Der Ansatz basiert auf einer hybriden Hardware-Software-Lösung, welche die Vorteile der jeweiligen Techniken vereinen soll: Der auszuführende sicherheitskritische Programmcode wird im Vorfeld mit Zusatzinformationen auf Software-Basis angereichert; zur Laufzeit nutzt eine spezielle Hardware-Einheit im Prozessorkern diese eingefügten Informationen, um regelmäßig die zeitliche und logische Korrektheit der Anwendung während der Ausführung zu überprüfen. Im Einzelnen soll der entwickelte Mechanismus somit folgende Ziele erfüllen:

- Eine möglichst breite Abdeckung erkannter Fehler soll durch die Überwachung sowohl zeitlicher als auch logischer Aspekte des Kontrollflusses gewährleistet sein.
- Durch eine sehr feingranulare Arbeitsweise des Prüfvorganges soll eine äußerst geringe Latenz der Fehlererkennung erreicht werden.
- Die benötigte Überwachungskomponente soll möglichst einfach in eine Prozessorumgebung integrierbar sein; der Aufwand durch den Mehrfacheinsatz identischer Hardware – wie in sicherheitskritischen Systemen üblich – soll dabei auf den Einsatz einer kostensparenden Check-Einheit innerhalb des Prozessors reduziert werden.
- Es soll nicht nötig sein, Programmcode redundant auszuführen, um Abweichungen festzustellen; der Aufwand hinsichtlich Speicherverbrauch und zusätzlicher Ausführungszeit soll möglichst gering sein, besonders im Vergleich zu bestehenden Fehlererkennungsmechanismen.

Neben einer detaillierten Entwicklung der Einzelschritte des Prüfmechanismus sowie einer Vorstellung der implementierten Werkzeuge befasst sich diese Arbeit auch mit der künstlichen Generierung von Fehlerfällen, um auf Basis einer Simulationsumgebung die Fehlerabdeckung und das Zeitverhalten bei der Erkennung zu evaluieren sowie einen potenziellen Praxiseinsatz zu bewerten. Ferner sollen mögliche Optimierungstechniken vorgestellt werden, mit dem Ziel, sowohl den Aufwand zu verringern als auch das Niveau an Verlässlichkeit weiter zu erhöhen.

1.2 Aufbau der Arbeit

Das folgende Kapitel beginnt mit der Erläuterung einiger Grundlagen: Zunächst sollen der Kontrollfluss eines Programms sowie die Gliederung in Grundblöcke beleuchtet werden. Anschließend werden Grundlagen zur Echtzeitfähigkeit, Anforderungen an Hardware und Software von Echtzeitsystemen sowie Methoden und Werkzeuge zur Analyse des Zeitverhaltens erklärt. Der dritte Teil des Kapitels beschreibt schließlich einige grundlegende Aspekte zu funktionaler Sicherheit, Verlässlichkeit und Fehlertoleranz. Dabei stehen neben Fehlerursachen und deren Auswirkungen auch Mechanismen zur Fehlererkennung und -behandlung im Vordergrund.

In Kapitel 3 folgt eine detaillierte Vorstellung des neu entwickelten Mechanismus zur dynamischen Erkennung von Fehlern in Echtzeitsystemen. Im Wesentlichen besteht diese Technik aus zwei Teilen: Zunächst steht die Erkennung von Fehlern im

Zeitverhalten im Mittelpunkt, anschließend wird diese um die Erkennung logischer Kontrollflussfehler erweitert. An dieser Stelle soll auch anhand eines Fehlermodells diskutiert werden, welche Fehlerarten durch die vorgestellte Technik wirkungsvoll erkannt werden können. Den Abschluss des Kapitels bildet eine Übersicht verwandter und ähnlicher Arbeiten, welche auch vom Ansatz dieser Arbeit abgegrenzt werden sollen.

Eine praxisnahe Implementierung der entwickelten Techniken findet sich in Kapitel 4. Hierbei soll zunächst die Frage verfolgt werden, auf welche Weise die softwareseitig eingefügten (instrumentierten) Sicherheitsinformationen umsetzbar sind, um dann ein Werkzeug vorzustellen, welches die entsprechende Instrumentierung von Applikationen automatisiert durchführt. Außerdem soll die in Hardware realisierte Prüfeinheit vorgestellt werden, wobei die Funktionalität sowohl beim temporalen als auch beim logischen Check-Vorgang erläutert wird. Abschließend ist es das Ziel, den Hardware-Aufwand durch die Eingliederung der zusätzlichen Komponente möglichst genau abzuschätzen.

Auf Basis der vorgestellten Implementierung findet in Kapitel 5 eine Evaluierung verschiedener Aspekte statt. Zentraler Punkt ist hierbei die Abdeckung erkannter Fehlerfälle, was über verschiedene Simulationen mit Erzeugung künstlicher Fehlerfälle abgeschätzt werden kann. Ebenso ist die Latenz bei der Fehlererkennung von Interesse. Auf der anderen Seite soll im Rahmen des Kapitels aber auch eine detaillierte Aufwandsanalyse zeigen, mit welchem Mehrbedarf an Speicher zu rechnen ist und wie hoch die zusätzlich benötigte Ausführungszeit einzuschätzen ist.

Kapitel 6 beschreibt drei Optimierungstechniken, mit der Absicht, einerseits die Fehlererkennung zu verbessern, andererseits die damit verbundenen Kosten zu reduzieren. In der ersten Variante gilt es zu erörtern, inwiefern sich neben der überprüften maximalen Ausführungszeit eine Programmes die zusätzliche Betrachtung der minimalen Ausführungszeit auf eine Fehlererkennung auswirkt. Die zweite Variante untersucht, ob weniger als die zunächst instrumentierten Informationen pro Prüfpunkt ausreichen, um dennoch ein ähnliches Sicherheitsniveau zu erreichen. Im Rahmen einer dritten Optimierungsvariante wird das Ziel verfolgt, die Abstände der instrumentierten Prüfpunkte anzugleichen, um dadurch bei geringerem Aufwand die durchschnittliche Latenz der Fehlererkennung zu reduzieren. Alle Techniken sollen ausführlich nach verschiedenen Kriterien evaluiert und verglichen werden. Außerdem stellt sich die Frage, ob eine Kombination der Optimierungstechniken eine weitere Verbesserung der Ergebnisse erzielen kann.

Den Abschluss dieser Arbeit bildet Kapitel 7 mit einer kurzen Zusammenfassung sowie einem Ausblick auf anknüpfende Forschungsziele.

2

Grundlagen

Um sich detailliert mit einer Erkennung von Kontrollflussfehlern in Echtzeitsystemen befassen zu können, ist es nötig, zuvor einige essenzielle Grundlagen zu den Themen Kontrollfluss, Echtzeitfähigkeit sowie Verlässlichkeit und Fehlertoleranz zu erläutern. Die einzelnen Themengebiete sind natürlich zu umfassend, um hier ein vollständige Einführung zu geben. Daher soll der Fokus auf die im vorliegenden Kontext wichtigen Zusammenhänge gesetzt sein.

2.1 Kontrollfluss

Jedes Computerprogramm besteht aus einer Menge einzelner Befehle. Die Reihenfolge dieser Befehle bei der Programmausführung legt dabei den *Kontrollfluss* bzw. *Steuerfluss* fest. *Kontrollstrukturen* erlauben es, von der sequenziellen Abarbeitung eines Programms abzuweichen und den Steuerfluss eines imperativen oder deklarativen Computerprogramms zu ändern. So kann eine unterschiedliche Reaktion auf Zustände und Eingaben stattfinden. Die verschiedenen Arten von Kontrollstrukturen können nach Hennessy und Patterson (1994, S. 104) wie folgt kategorisiert werden:

- **Bedingte Verzweigungen**: Falls die zu prüfende Bedingung erfüllt ist, wird die Ausführung an einem anderen Programmabschnitt fortgesetzt.
- **Sprünge**: Die Ausführung wird ohne Bedingungen an einem anderen Programmabschnitt fortgesetzt.
- **Prozedurrufe**: Ein Unterprogramm bzw. eine Funktion wird aufgerufen, die Ausführung springt an dessen Beginn.

- **Prozedurrückkehr**: Nach Beendigung eines Unterprogramms springt die Ausführung zurück zum ursprünglichen Abschnitt, von welchem der Aufruf des Unterprogramms initiiert wurde.

Als *Grundblock* oder *Basic Block* (Allen 1970) bezeichnet man eine Sequenz von Befehlen, welche ohne Unterbrechungen oder Verzweigungen hintereinander ausgeführt wird. Jeder Grundblock besitzt also genau einen Eintrittspunkt und genau einen Austrittspunkt. Im gesamten Programm darf kein Sprung existieren, der nicht den Anfang eines Grundblocks zum Ziel hat. Die letzte Instruktion eines Grundblocks ist meist eine Kontrollstruktur, die angibt, welcher Programmteil als Nächstes auszuführen ist.

Maximale Grundblöcke (Grune u. a. 2000, S. 320) sind Grundblöcke, die nicht mit benachbarten Blöcken zusammengefasst werden können, ohne die Definition der Grundblöcke zu verletzen. Die folgende Vorgehensweise beschreibt die Zerlegung von Programmcode in maximale Grundblöcke: Zunächst gilt es, die Startpunkte

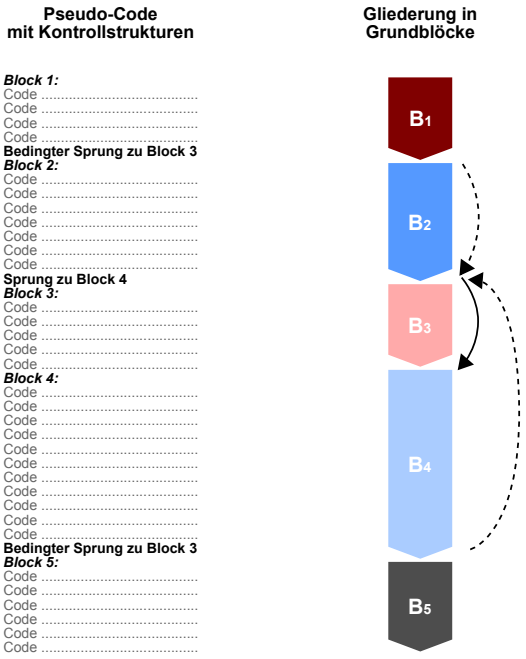


Abbildung 2.1: Grafische Darstellung der Gliederung eines einfachen Programmes mit Kontrollstrukturen in Grundblöcke

der Grundblöcke zu identifizieren. Diese sind die erste Instruktion im Programm, alle Ziele bedingter und unbedingter Sprünge sowie alle Instruktionen, die unmittelbar auf einen bedingten oder unbedingten Sprung folgen. Nun müssen zu jedem Startpunkt die nachfolgenden Instruktionen bis zum Erreichen eines nächsten Startpunktes verfolgt werden. Die Menge dieser Instruktionen ausschließlich des nächsten Startpunktes bildet den jeweiligen maximalen Grundblock. Durch diese Methode ist somit sichergestellt, dass jeder dieser Blöcke genau einen Startpunkt beinhaltet. Wird im Lauf der weiteren Arbeit von Grundblöcken gesprochen, so sind stets maximale Grundblöcke gemeint.

Abbildung 2.1 zeigt auf der linken Seite einen symbolischen Ausschnitt eines imperativen Programmes mit Kontrollstrukturen. Rechts findet sich eine grafische Darstellung des Kontrollflusses: Die farbigen Kästen entsprechen den jeweiligen Grundblöcken, Pfeile repräsentieren Sprünge, gestrichelte Pfeile bedingte Verzweigungen. Ähnliche Grafiken sollen in der weiteren Arbeit bestimmte Szenarien des Kontrollflusses verdeutlichen; die Grundblöcke werden dann aus Platzgründen in horizontaler Abfolge aufgetragen.

2.2 Echtzeitfähigkeit

Ein Rechnersystem ist genau dann *echtzeitfähig* bzw. ein *Echtzeitsystem*, wenn es die Eigenschaft besitzt, auf bestimmte Umgebungsereignisse präzise innerhalb zeitlicher Beschränkungen zu reagieren. Das korrekte Verhalten eines Echtzeitsystems hängt demnach nicht nur vom Ergebniswert der Berechnung ab, sondern auch vom Zeitpunkt, wann dieses Ergebnis vorliegt (Buttazzo 1997, S. 1 ff.). Wird die Berechnung zu spät beendet, kann diese nicht nur nutzlos sein, sondern gefährliche oder gar katastrophale Konsequenzen nach sich ziehen.

Echtzeitsysteme spielen in vielen Bereichen des heutigen Lebens eine große Rolle; besonders *eingebettete Systeme*, d.h. Computer, welche fest in einen speziellen technischen Kontext eingebunden sind, müssen häufig Echtzeitbedingungen erfüllen. Dies reicht von Multimedia- und Kommunikations-Systemen über den Automobilbereich bis hin zur Robotik oder Luft- und Raumfahrtindustrie.

2.2.1 Anforderungen an Echtzeitsysteme

Auf Prozessebene ist für eine Echtzeitanwendung die Existenz einer Zeitschranke (*Deadline*) charakteristisch. Diese symbolisiert den Zeitpunkt, zu dem die Ausführung beendet sein muss (Buttazzo 1997, S. 8). Je nach Ausmaß der Konsequenzen

beim Überschreiten dieser Schranke lassen sich Echtzeitsysteme gewöhnlich in zwei Klassen einteilen (Manacher 1967):

- **Harte Echtzeitsysteme** haben die Anforderung, dass das zu berechnende Ergebnis innerhalb der vorgegebenen Zeitschranke vorliegen *muss*. Wird dies nicht erreicht, sind katastrophale Konsequenzen zu befürchten. Typisches Beispiel ist hier die Steuerung des Airbags im Auto: Wird dieser verzögert ausgelöst, ist die schützende Wirkung bei einem Unfall außer Kraft gesetzt.

- **Weiche Echtzeitsysteme** sind dadurch gekennzeichnet, dass das Ergebnis innerhalb der vorgegebenen Zeitschranke vorliegen *soll*. Im Fall einer Überschreitung leidet der Nutzen, doch das Ergebnis kann in der Regel verwendet werden. Häufig angeführt werden in diesem Zusammenhang Multimedia-Applikationen wie die Dekodierung eines Video-Streams. Kommt es zu einer Verzögerung, so verschlechtert sich die Filmqualität, doch weitere Konsequenzen bleiben aus.

Ausgehend von der Notwendigkeit, das Überschreiten der gegebenen Zeitschranke auszuschließen, muss das System vollständig *vorhersagbar* und damit *analysierbar* sein (Stankovic und Ramamritham 1990). Dabei gilt es, eine Obergrenze für die maximale Ausführungszeit im schlechtesten Fall (*Worst-Case Execution Time, WCET*) zu bestimmen. Diese muss kürzer sein als die Deadline, welche durch den Einsatzzweck vorgegeben ist. Ergibt die Laufzeitanalyse ein potenzielles Überschreiten der Zeitschranke, so muss entweder leistungsstärkere Hardware gewählt werden oder eine Optimierung der Software erfolgen.

Entgegen häufiger Missverständnisse ist es bei Echtzeitsystemen nicht das Ziel, optimale Performanz zu erzielen (Stankovic 1988). Statt möglichst schnell das Ergebnis zu berechnen, gilt es vielmehr, unvorhersehbare Verzögerungen a priori auszuschließen und optimale Analysierbarkeit zu gewährleisten.

2.2.2 Analyse des Zeitverhaltens

Besonders wichtig ist im Zusammenhang mit Echtzeitsystemen eine möglichst optimale Analyse des Zeitverhaltens. Die primäre Schwierigkeit besteht darin, dass die Laufzeit eines Programmes auf einem bestimmten Prozessor sowohl von den Eingabedaten als auch vom Initialzustand der Hardware (vor allem der Speicherbelegung) abhängig ist. Dabei alle möglichen Belegungen zu berücksichtigen, ist in der Regel nicht möglich: Zum Einen ist es schwierig, Werte der Eingabedaten zu finden, welche sämtliche Ausführungspfade abdecken, zum Anderen ist es zu

aufwendig, all diese Pfade zu analysieren und deren Zeitverhalten zu messen. Hinzu kommt, dass sämtliche initialen Hardware-Zustände bei einer Analyse schwer nachzubilden sind. Aus diesen Gründen ist es gängig, den Programmcode bei der WCET-Analyse in kleine Bestandteile (meist Grundblöcke) zu zerlegen und deren Zeitverhalten zu bestimmen. Die WCET-Abschätzung für die Gesamtapplikation wird schließlich aus diesen Teilergebnissen berechnet.

Prinzipiell lässt sich die Art der WCET-Analyse in folgende zwei Klassen einteilen (Wilhelm u. a. 2008):

- Die *statische WCET-Analyse* basiert nicht darauf, Code auf echter Hardware oder einem entsprechenden Simulator auszuführen. Vielmehr wird mit Hilfe von Untersuchungen des Kontrollflusses und des Maschinencodes der längste Pfad eines Programmes ermittelt. Gemeinsam mit einem detaillierten, abstrakten Modell der Hardware kann daraus die maximale Laufzeit errechnet werden. Als Beispiele für diese Art der WCET-Analyse seien hier das kommerzielle Tool *aiT* (Ferdinand und Heckmann 2004) sowie das frei verfügbare Tool *OTAWA* (Ballabriga u. a. 2010) genannt.
- Bei der *messungsbasierten* WCET-Analyse werden Programme oder Programmteile zunächst mit zusätzlichen Instruktionen instrumentiert, um diese dann auf der Zielhardware bzw. einem entsprechenden Simulator auszuführen. Mit Hilfe der Instrumentierung erzeugt die Ausführung Zeitstempel. So kann die minimale und maximale Laufzeit beobachtet und abgeleitet werden, um daraus wiederum die maximale Gesamtlaufzeit zu berechnen. Ein wichtiger Punkt ist hierbei die möglichst optimale Erzeugung von Eingabedaten (Kirner u. a. 2004), um alle Programmpfade bei der Analyse abzudecken. Ein Beispiel für messungsbasierte WCET-Analyse ist das kommerzielle Tool *RapiTime* (RapiTime 2011).

Beide Ansätze haben Vor- und Nachteile: Bei der statischen Analyse ist die Modellierung der jeweiligen Zielplattform mit hohem Aufwand verbunden und problematisch, insbesondere bei modernen Prozessoren mit Caches und komplexen Pipelines. Selbst bei kleinen Änderungen an der Hardware ist oft eine komplett neue Modellierung nötig. Andererseits garantiert die statische Analyse aber ein hohes Maß an Verlässlichkeit, während bei der messungsbasierten Variante stets die Frage bleibt, ob wirklich alle möglichen Ausführungsvarianten berücksichtigt sind. Die Generierung der Eingabedaten und die Betrachtung sämtlicher Hardware-Zustände stellt hierbei eine schwierige Herausforderung dar.

In jedem Fall ist die Anforderung an eine WCET-Analyse nicht nur die Ermittlung einer garantierten zeitlichen Obergrenze, sondern auch eine möglichst geringe Überschätzung. Die *Tightness* der Analyse beschreibt in diesem Zusammenhang

den Abstand zwischen der theoretisch errechneten WCET-Grenze und der (nicht definitiv ermittelbaren) tatsächlich auftretenden Maximallaufzeit. Um ein Maß für die WCET-Analyse zu erhalten, wird bei Wilhelm u. a. (2008) zunächst eine weitere Größe bestimmt, die *Best-Case Execution Time (BCET)*. Damit lässt sich die *Timing Predictability*, also die zeitliche Vorhersagbarkeit eines Systems, als Differenz zwischen analysierter Maximal- und Minimallaufzeit errechnen. Die Hardware eines zeitkritischen Echtzeitsystems wird so dimensioniert, dass die vorgegebene Deadline mit der errechneten WCET-Abschätzung unter allen Umständen eingehalten werden kann. Eine möglichst präzise zeitliche Vorhersagbarkeit spart dabei unnötige Überdimensionierung.

Auf der anderen Seite ergeben sich auch aus Sicht der Analysierbarkeit Anforderungen an Hard- und Software eines Systems. Um die Vorhersagbarkeit möglichst optimal zu halten, gilt es, spekulative Hardware-Elemente und mehrstufige Zwischenspeicher zu vermeiden. Für Entwickler von Software für eingebettete Echtzeitsysteme existieren Richtlinien, sog. *Coding Guidelines*, wie beispielsweise von Bonenfant u. a. (2010). Diese empfehlen, bestimmte Konstrukte und Strukturen bei der Programmierung zu vermeiden, welche zu einer sehr pessimistischen WCET-Abschätzung führen bzw. die WCET-Analyse unmöglich machen würden.

2.3 Verlässlichkeit und Fehlertoleranz

In sicherheitskritischen Rechensystemen steht vor allem eine Eigenschaft im Vordergrund: die Verlässlichkeit. Eine fehlerhafte Berechnung oder ein Ausfall des Systems kann katastrophale Konsequenzen nach sich ziehen. Um derartige Gefahren einzugrenzen, hat das Thema Sicherheit im Laufe der letzten Jahre in Fachkreisen zunehmend an Bedeutung gewonnen (Jones 2000). In Form von Gesetzen oder Standards werden je nach Anwendungsdomäne Verlässlichkeitsanforderungen definiert (Bowen und Stavridou 1993), welche Wahrscheinlichkeiten für kritische Systemausfälle vorschreiben. So darf beispielsweise in einem Verkehrsflugzeug nach dem Standard *DO-178B* (RTCA 1992) nur ein einziger kritischer Fehler pro Milliarde Flugstunden auftreten. Im Rahmen eines Zertifizierungsprozesses muss vor dem Praxiseinsatz sichergestellt werden, dass eine entsprechende Komponente tatsächlich die vom Standard definierten Anforderungen erfüllt.

Im Mittelpunkt dieser Arbeit steht die Fehlererkennung; der vorliegende Abschnitt soll schrittweise an dieses Thema heranführen. Dazu werden im ersten Teil einige Grundkonzepte und Begriffe der Verlässlichkeit erklärt. Es findet eine Klassifizierung potenzieller Fehlerursachen und Auswirkungen statt, um schließlich Mittel

zur Erhöhung der Verlässlichkeit eines Systems zu erläutern. Im zweiten Schritt wird das Gebiet der Fehlertoleranz mit den Teilgebieten der Fehlererkennung, Fehlzustandsbehandlung und Fehlerursachenbehandlung näher beleuchtet. Den Abschluss dieses Kapitels bildet ein Überblick über verschiedene Mechanismen und Implementierungsvarianten der Fehlererkennung.

2.3.1 Grundkonzepte und Terminologie

Als zentraler Überbegriff gilt die *Verlässlichkeit*[1] (*Dependability*): Diese wird definiert als Vertrauenswürdigkeit eines Rechensystems, d.h. es kann Vertrauen in die Leistung, die es erbringt, gesetzt werden. Diese Leistung ist dabei das Systemverhalten, wie es von einem Benutzer beobachtet wird; der Benutzer ist in diesem Zusammenhang ein anderes System (menschlich oder physikalisch), das mit dem vorhergehenden zusammenwirkt (Laprie u. a. 1992).

Die Verlässlichkeit ist durch verschiedene Eigenschaften charakterisiert, die je nach Anwendungsbereich des Systems in unterschiedlichem Maße ausgeprägt sind. Diese werden nach Avizienis u. a. (2004) als Attribute der Verlässlichkeit wie folgt definiert: *Verfügbarkeit* (*Availability*) beschreibt die Bereitschaft zum Gebrauch, während *Zuverlässigkeit* (*Reliability*) auf die Kontinuität der Leistung abzielt. *Sicherheit* (*Safety*) meint das Fernbleiben katastrophaler Folgen für Benutzer und Umgebung, *Integrität* (*Integrity*) das Fernbleiben jeglicher unzulässiger Veränderung des Systems. Als letztes der fünf Attribute beschreibt *Instandhaltbarkeit* (*Maintainability*) die Möglichkeit, dass Änderungen oder Reparaturen durchgeführt werden.

Ausfall, Fehler und Fehlerursache

Auch Begriffe wie Fehler und Ausfall werden bei Laprie u. a. (1992) exakt abgegrenzt: Ein *Ausfall* des Systems (*Failure*) tritt dann auf, wenn die erbrachte Leistung nicht mehr mit der Spezifikation, also der Beschreibung von erwarteter Funktion und Leistung, übereinstimmt. Ein *Fehler* (*Error*) ist dabei der Teil des Systemzustands, der verantwortlich ist, dass im Folgenden ein Ausfall auftritt.

[1] Die Arbeit von Laprie u. a. (1992) gilt als Standardwerk für die Terminologie im vorliegenden Fachbereich. Allerdings wählt die deutsche Übersetzung der Auflage von 1992 für *Dependability* den Begriff *Zuverlässigkeit*. Inzwischen ist es jedoch gängiger und unmissverständlicher, als Übersetzung *Verlässlichkeit* zu benutzen. *Zuverlässigkeit* hingegen beschreibt besser ein Teilgebiet, die *Reliability*.

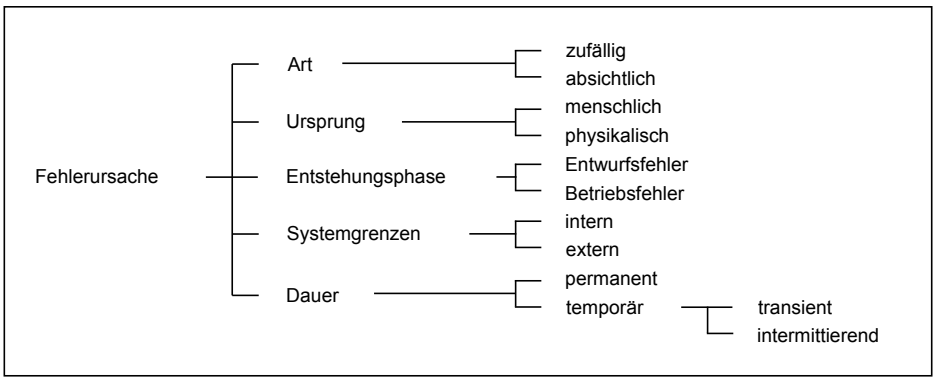


Abbildung 2.2: Klassifizierung von Fehlerursachen

Die (tatsächliche oder hypothetische Ursache) für einen Fehler wird mit dem englischen Wort *Fault* beschrieben und ins Deutsche entsprechend mit *Fehlerursache* übersetzt.

Die Fehlerursachen sind sehr unterschiedlich und können nach vielen Kriterien klassifiziert werden (vgl. Abbildung 2.2): Auf der einen Seite gibt es zufällige Faults, auf der anderen Seite absichtliche Faults, welche durch böswilligen Vorsatz verschuldet sind. Der Ursprung der Fehlerursachen kann nach verschiedenen Gesichtspunkten aufgeteilt werden: Neben menschlichen Ursachen, die durch eine falsche Bedienung ausgelöst sind, kommen physikalische Ursachen vor, z.B. durch hohe Temperatur oder Erschütterungen. Ferner lassen sich Fehlerursachen nach der Entstehungsphase unterscheiden. Diese können auf Mängeln beruhen, welche bereits während der Entwicklung des Systems entstanden sind, oder andererseits auf Betriebsfehlern, die während der Nutzung des Systems auftreten. Fehlerursachen können systemintern verschuldet oder von einer externen Fehlerquelle ausgelöst sein. Besonders wichtig ist auch eine Unterscheidung im Hinblick auf die Dauer der Fehlerursache: *Permanente* Faults sind durchgehend und ohne Bedingungen aktiv, während *temporäre* Fehlerursachen nur während einer bestimmten Zeit vorhanden sind. Temporäre externe Faults, die von der physikalischen Umgebung stammen (z.B. durch elektromagnetische Strahlung), werden oft *transiente* Fehlerursachen genannt, während temporäre interne Faults, beispielsweise durch Überlastung der Hardware-Komponenten, als *intermittierende* Faults bezeichnet werden. In diesem Zusammenhang spricht die Literatur auch häufig von *harten* und *weichen Fehlern* (*Hard Errors* bzw. *Soft Errors*) (Karnik u. a. 2004): Harte Fehler resultieren aus einer permanenten Fehlerursache, weiche Fehler aus einer temporären.

Mittel zur Erhöhung der Verlässlichkeit eines Systems

In den letzten Jahrzehnten wurde eine Vielzahl von Methoden und Mechanismen entwickelt, um Rechensysteme mit den verschiedenen Attributen der Verlässlichkeit auszustatten. Diese können nach Avizienis u. a. (2004) in die folgenden vier Gruppen eingeteilt werden:

- *Fehlerverhinderung* hat das Ziel, das Auftreten bzw. die Einführung von Fehlern abzuwenden,
- *Fehlertoleranz* bezweckt, trotz auftretender Fehler die Leistung zur Verfügung zu stellen, welche mit der Spezifikation des Systems übereinstimmt,
- *Fehlerbeseitigung* strebt an, sowohl die Anzahl als auch den Schweregrad von Fehlern zu reduzieren,
- *Fehlervorhersage* zielt darauf ab, die aktuelle Zahl, das künftige Auftreten sowie die potenziellen Folgen von Fehlern abzuschätzen.

Diese Arbeit fokussiert auf den Bereich der Fehlertoleranz. Daher ist es hilfreich, diesen Begriff im Folgenden etwas näher zu beleuchten sowie die einzelnen Fehlertoleranztechniken gegeneinander abzugrenzen.

2.3.2 Fehlertoleranztechniken

Fehlertoleranz besteht im Wesentlichen aus zwei Komponenten: der *Fehlererkennung* (*Error Detection*) sowie der entsprechenden *Fehlerbehandlung* (*Recovery*). Nach Avizienis u. a. (2004) lassen sich weitere Unterscheidungen treffen (vgl. Abbildung 2.3). Die Fehlererkennung, welche die Präsenz eines Fehlers identifiziert, kann entweder gleichzeitig zur Systemausführung stattfinden oder präemptiv. Im letzteren Fall wird das System im Vorfeld bzw. nach Unterbrechung der Ausführung auf Fehler untersucht.

Neben der Fehlzustandsbehandlung, welche auf die Beseitigung von Fehlern aus dem Rechenzustand abzielt, gibt es die Möglichkeit der Fehlerursachenbehandlung, also der Verhinderung, dass eine Fehlerursache überhaupt erst aktiviert wird. Ein Fehlzustand kann behoben werden, indem ein fehlerfreier Zustand den fehlerhaften ersetzt. Dies kann auf zwei Arten erfolgen (vgl. auch Laprie u. a. 1992): Bei der Rückwärtsfehlerbehebung gilt es, das System in einen Zustand zu bringen, in welchem es vor dem Fehler schon einmal war. Dies setzt Rücksetzpunkte voraus, also Zeitpunkte der Ausführung, zu denen der dann aktuelle Zustand gesichert wird, um später notfalls wiederhergestellt werden zu können. Alternative ist die

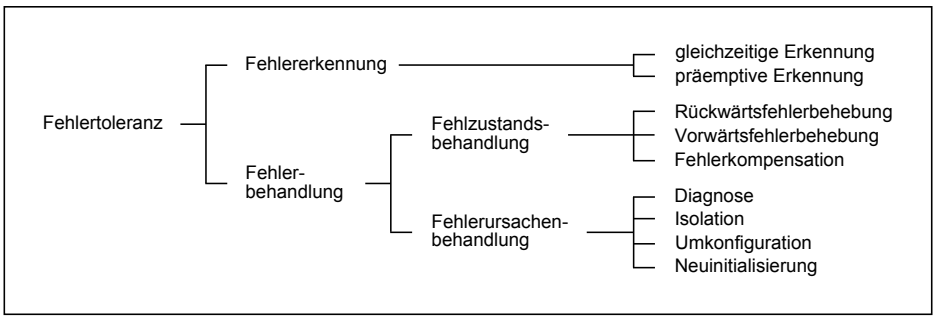


Abbildung 2.3: Fehlertoleranztechniken nach Avizienis u. a. (2004)

Vorwärtsfehlerbehandlung, bei der versucht wird, einen neuen fehlerfreien Zustand zu finden, von dem aus das System (meist in reduziertem Umfang) wieder arbeiten kann. Als weitere Form der Fehlzustandsbehebung gibt es die Fehlerkompensation, bei welcher mit Hilfe von Redundanztechniken eine fehlerfreie Leistungserbringung trotz fehlerhaften internen Zustands ermöglicht wird.

Bei der Fehlerursachenbehandlung ist der erste Schritt die Fehlerursachendiagnose. Hierbei werden Ort und Art der Fehlerursache bestimmt. Anschließend folgen Aktionen, die verhindern sollen, dass diese aktiviert wird. Eine Möglichkeit dafür ist die Isolation, also die Ausgliederung der Fehlerursache. Dies geschieht dadurch, dass die als fehlerhaft identifizierten Komponenten von der weiteren Ausführung ausgeschlossen werden. Ist das System nicht in der Lage, nach einer entsprechenden Isolation die geforderte Leistung zu erbringen, besteht die Möglichkeit einer Umkonfiguration oder Neuinitialisierung, in der Hoffnung, dass die fehlerhaften Komponenten im Anschluss wieder korrekt arbeiten.

Im Rahmen dieser Arbeit steht weniger die Fehlerbehandlung als vielmehr die Fehlererkennung im Mittelpunkt. Daher sollen im Folgenden einige Details zu diesem Teilgebiet der Fehlertoleranz näher beschrieben und erläutert werden.

2.3.3 Fehlererkennung

Für die Erkennung von Fehlern ist es nötig, das Verhalten eines Systems zu überwachen. Anhand einer bestimmten Menge von Regeln gilt es, korrektes Verhalten von fehlerhaftem zu unterscheiden. Scherrer und Steininger (2003) teilen die Erkennungsmechanismen in folgende Kategorien ein:

- **Prüfen der Kontrollflusspfade**: Ziel ist es, die Konsistenz zwischen erwartetem und tatsächlich ausgeführtem Kontrollfluss sicherzustellen. Die Implementierung der entwickelten Ansätze kann hardware- oder softwarebasiert sein (vgl. Abschnitt 3.4). Auch Mischformen sind möglich.

- **Überwachung der Ausführungszeit**: Nach AUTOSAR (2008)[2] fallen in diese Kategorie verschiedene Mechanismen: Applikationen können zu bestimmten Zeitpunkten Lebenszeichen an einen Watchdog senden. Dieser wird aktiv und meldet einen Fehlerfall, sobald die erwarteten Botschaften ausbleiben (*Watchdog Style Monitoring*). Zeitlich überwachen lässt sich aber auch der Kontrollfluss (*Temporal Program Flow Monitoring*) oder andere Aspekte eines Systems (*General Timing Related Checks*).

- **Sicherstellen der Datenintegrität**: Hierbei wird insbesondere die korrekte Speicherung und Übertragung von Daten kontrolliert. Dies kann durch zusätzliche Bits, welche redundante Informationen enthalten, erfolgen. Gängiges Beispiel sind *Error Correcting Codes (ECC)* (Peterson und Weldon 1972), die zur Fehlererkennung und gleichzeitigen -behandlung dienen.

- **Vergleich redundanter Berechnungen**: Eine weit verbreitete Methode zur Fehlererkennung ist die redundante Ausführung von Berechnungen, entweder sequentiell oder parallel (Avizienis u. a. 2004). Basierend auf der Annahme, dass verschiedene Hardware-Komponenten bei physikalischen Fehlerursachen unabhängig voneinander ausfallen, können die redundanten Einheiten identisch aufgebaut sein. Soll gleichzeitig eine Absicherung gegenüber Entwicklungsfehlern erfolgen, können Komponenten eingesetzt werden, welche die gleiche Funktion erfüllen, aber unterschiedlich implementiert sind (Avizienis und Kelly 1984, Avizienis 1985, Lyu 1995).

- **Plausibilitäts-Checks**: Ziel dieser Methode ist es, bei Rechenergebnissen, Sensorwerten oder sonstigen numerischen Daten zu prüfen, ob sich die Werte innerhalb eines erlaubten Bereichs befinden. Die Algorithmen können dabei variieren: Entweder es erfolgt nur ein Vergleich mit den definierten Ober- und Untergrenzen oder der Erwartungswert einer Abfolge von Werten wird auf komplexere Weise ermittelt und entsprechend als Referenz verwendet (AUTOSAR 2008).

[2] AUTOSAR (AUTomotive Open System ARchitecture) ist ein Zusammenschluss von Automobilherstellern, Steuergeräteentwicklern sowie Hard- und Softwarefirmen mit dem Ziel, Austauschbarkeit zwischen unterschiedlichen Hard- und Softwarekomponenten zu erleichtern. Dazu wird eine Vielzahl von Anforderungen definiert, u.a. an die technische Sicherheit und Zuverlässigkeit der eingesetzten Komponenten.

Neben den angeführten Fehlererkennungsmechanismen, welche das Systemverhalten zur Laufzeit beobachten, besteht die Möglichkeit, durch aktive *Selbst-Tests* die korrekte Funktionalität zu prüfen (Scherrer und Steininger 2003). Im Rahmen eines Challenge-Response-Verfahrens werden vor der Programmausführung bestimmte Rechenoperationen mit bereits bekanntem Ergebnis angestoßen. Liefern diese das erwartete Ergebnis, kann auf eine korrekte Funktion des Systems geschlossen werden. Nach AUTOSAR (2008) wird diese Form der Fehlererkennung *Instruction Set Testing* genannt. Nachteil der Methodik ist, dass transiente Fehlerursachen zur Laufzeit nicht erkannt werden können, wenn diese während der initialen Testphase noch nicht aktiv sind.

Es ist offensichtlich, dass nicht alle Erkennungsmechanismen gleich wirkungsvoll arbeiten. Als Maßstab für die Effektivität einer Technik dienen im Wesentlichen zwei Faktoren: Die *Abdeckung* erkannter Fehlerfälle (*Coverage*) gibt an, wie viele Fehler mit Hilfe einer bestimmten Erkennungsmethode identifizierbar sind (Avizienis u. a. 2004). Die *Erkennungslatenz* (*Latency*) beschreibt den Zeitraum zwischen dem Auftreten einer Fehlerursache und dem Augenblick der Erkennung. Während dieser Zeit wird der Fehlzustand auch als *latent* bezeichnet. Ein Fehlzustand kann verschwinden, bevor er erkannt ist. Andererseits kann er – wie in den meisten Fällen – propagieren und erzeugt damit weitere, neue Fehlzustände (Laprie u. a. 1992). Als Erkennungslatenz versteht sich in diesem Fall die Zeitspanne zwischen Auftreten der *ersten* Fehlerursache und der *ersten* Fehlererkennung.

Um Fehlertoleranzeigenschaften eines Rechnersystems formal nachzuweisen, muss eine *Fehlerannahme* (*Fault Assumption*) bzw. ein *Fehlermodell* (*Fault Model*) spezifiziert werden (Gärtner 2001). Dieses formalisiert auf präzise Weise das potenzielle Fehlverhalten einzelner Komponenten eines Systems und kann auf verschiedene Ebenen abgebildet werden: Bekannte Fehlermodelle wie *Fail-Stop* oder *Crash* (Cristian 1985) nehmen einen Absturz auf Prozessebene an, wobei der abgestürzte Prozess im ersten Fall eine Nachricht über den Absturz zurückliefert, im zweiten Fall nicht. Andererseits ist es möglich, nur die Fehlerursache zu spezifizieren, beispielsweise eine temporäre Veränderung eines einzelnen Bitwertes, was häufig durch physikalische Einflüsse entstehen kann. Diese Art des Fehlermodells soll auch für Evaluierungen im Rahmen dieser Arbeit verwendet werden. Näheres dazu folgt in Kapitel 5.

3

Erkennung von Kontrollflussfehlern in Echtzeitsystemen

In sicherheitskritischen Systemen ist eine zuverlässige Programmausführung von besonderer Wichtigkeit. Verschiedene Studien (Schuette und Shen 1987, Ohlsson u. a. 1992) zeigen, dass bis zu 77 % aller Fehler in einem Computersystem den Kontrollfluss unerlaubt verändern. Dies motiviert, den Fokus speziell auf die Korrektheit des Kontrollflusses einer Applikation zu legen. Bei der Umsetzung eines entsprechenden Mechanismus zur Fehlererkennung ist es dabei zunächst nötig, die häufigsten Fehlerursachen eines Systems zu betrachten: Nach Czeck und Siewiorek (1990) werden 80 % - 90 % aller Hardwarefehler (also auch Kontrollflussfehler) von transienten Fehlerursachen ausgelöst. Die Tatsache, dass diese Fehlerursachen zufällig auftreten und nicht reproduzierbar sind, macht eine präemptive Fehlererkennung vor der Programmausführung wirkungslos. Vielmehr ist ein Prüfungsmechanismus nötig, der dynamisch zur Laufzeit den Kontrollfluss auf Fehler überprüft.

Ziel ist eine Garantie, dass verschiedene Teile einer Applikation in der richtigen logischen Abfolge ausgeführt werden und keinerlei plötzliche Abweichungen auftreten. Zusätzlich ergibt sich für Echtzeitsysteme die Anforderung, dass der zeitliche Ablauf gewisse Schranken nicht überschreiten darf. Es ist wesentlich, unerwartete Verzögerungen oder Verklemmungen frühzeitig zu erkennen, um etwaige Konsequenzen zu verhindern bzw. Gegenmaßnahmen ergreifen zu können. Zur Realisierung einer Fehlererkennung müssen daher folgende Voraussetzungen erfüllt sein: Zum Einen muss eine eindeutige Regel zur logischen Abfolge der einzelnen Programmteile existieren, welche eine Vorhersage bzw. Ankündigung erlaubt oder einen Ablauf auf Korrektheit verifizieren lässt. Zum Anderen muss es zu allen Teilstücken möglich sein, Grenzen der Ausführungszeit anzugeben, welche nur im

Fehlerfall überschritten werden. Sind diese Bedingungen erfüllt, so gilt es, beim Entwurf einer entsprechenden Methodik zur Fehlererkennung folgende Faktoren zu beachten:

- **Abdeckung von Fehlerfällen**: Der Anteil von Fehlern, welche durch den konzipierten Mechanismus erkannt werden, soll möglichst hoch sein. Hierbei ist eine Analyse mittels Fehlermodellen hilfreich. Die Erzeugung künstlicher Fehler im Rahmen von Simulationen hilft, den Prozentsatz erkannter Fehler realistisch abzuschätzen.
- **Latenz der Fehlererkennung**: Ziel ist es, den Zeitraum zwischen dem Auftreten eines Fehlers und dessen Erkennung zu minimieren. Die Teilstücke, nach welchen eine Überprüfung stattfindet, sollten daher möglichst feingranular gewählt werden, um eine rasche Erkennung von Fehlern im Kontrollfluss sicherzustellen. Im Hinblick auf das Echtzeitverhalten sollte auch die ermittelte Obergrenze der erlaubten Ausführungszeit nah an der realen Ausführungszeit liegen, um Fehler im Zeitverhalten des Programmablaufs möglichst früh feststellen zu können.
- **Ausführungszeit**: Jeder Prüfvorgang kostet Rechenzeit. Je mehr Prüfpunkte existieren, desto länger wird die Ausführungszeit des sicherheitskritischen Programms. Wichtig ist es daher, einen vertretbaren Kompromiss aus Sicherheitsniveau und Mehraufwand zu ermitteln.
- **Speicherbedarf**: Um die Korrektheit eines Programmablaufs feststellen zu können, muss der reale Zustand schrittweise mit dem zu erwartenden Zustand verglichen werden. Dabei müssen die im Vorfeld ermittelten Referenzwerte in irgendeiner Form abgelegt werden, was zusätzlichen Speicher erfordert. Da Speicher in eingebetteten Systemen nur sehr begrenzt vorhanden und daher kostbar ist, gilt es, den Speicherbedarf für den Mechanismus zur Fehlererkennung möglichst gering zu halten.
- **Komplexität**: Auch die Komplexität des Erkennungsmechanismus spielt eine wichtige Rolle. Es stellt sich die Frage, wie tief in bestehende Hardware eingegriffen werden muss bzw. wie viel zusätzliche Hardware nötig ist, um den Mechanismus zu integrieren.

Rein softwarebasierte Ansätze zur Erkennung von Kontrollflussfehlern führen üblicherweise zu einem enormen Aufwand; durch eine Vielzahl zusätzlich eingefügter Instruktionen, welche die Korrektheit prüfen, wird sowohl die Ausführungszeit als auch der Speicherverbrauch stark erhöht. Zudem machen Software-Lösungen meist Veränderungen am Compiler erforderlich, um den Prüfungsmechanismus in den Programmcode zu integrieren. Ansätze, welche nur zusätzliche Hardware zur

Korrektheitsprüfung des Kontrollflusses verwenden, zeichnen sich typischerweise durch hohe Komplexität aus. Eine Integration in den Prozessorkern erweist sich als schwierig und ist mit hohem Aufwand verbunden.

Der im Folgenden vorgestellte Mechanismus zur dynamischen Erkennung zeitlicher und logischer Fehler im Kontrollfluss von Echtzeitsystemen ist eine Art *hybrider Ansatz*, da er die Vorteile von hardware- und softwarebasierten Mechanismen vereint. Der sicherheitskritische Programmcode wird vor der Ausführung zerlegt und mit Zusatzinformationen zur korrekten Abfolge und dem jeweiligen Zeitverhalten der Teilstücke instrumentiert. Zur Laufzeit prüft eine an den Prozessorkern angeschlossene Hardware-Check-Einheit diese Informationen und garantiert ein höheres Sicherheitsniveau bei der Ausführung. Damit werden die Vorteile von Hardware-Techniken, wie ein geringer Aufwand hinsichtlich Ausführungszeit und Speicherverbrauch, mit der Flexibilität und der einfachen Einsetzbarkeit von Software-Lösungen kombiniert. Die Instrumentierung vor der Ausführung kann durch ein spezielles Werkzeug erfolgen, sodass keinerlei Veränderungen am Compiler notwendig sind.

Dieses Kapitel soll nun ausführlich die Details der entwickelten Technik zur Erkennung von Kontrollflussfehlern in Echtzeitsystemen beschreiben (vgl. Wolf u. a. 2012a). Dazu steht zunächst die Erkennung von Fehlern im Zeitverhalten im Vordergrund, dann die Erkennung logischer Kontrollflussfehler. Anhand eines Fehlermodells soll im Anschluss diskutiert werden, auf welche Weise Kontrollflussfehler im Programmablauf entstehen können und welche davon durch den entworfenen Mechanismus erkennbar sind. Den Abschluss des Kapitels bildet schließlich eine Vorstellung verwandter Arbeiten auf diesem Forschungsgebiet sowie deren Abgrenzung zum vorgestellten Ansatz.

3.1 Erkennung von Fehlern im Zeitverhalten

Prinzipiell kann ein Fehler im Zeitverhalten eines Programmes entweder durch das Über- oder Unterschreiten erlaubter Zeitschranken entstehen. Da in sicherheitskritischen Echtzeitsystemen allerdings die WCET sowie die Einhaltung der Deadlines im Vordergrund steht, fokussiert der im Folgenden vorgestellte Mechanismus auf Fehler, welche durch die Überschreitung der WCET-Abschätzung entstehen. Eine Erweiterung mit Berücksichtigung der Unterschreitung erlaubter Laufzeiten soll erst im Rahmen der Optimierungsmöglichkeiten in Abschnitt 6.1 näher betrachtet werden.

Der entworfene Mechanismus zur Erkennung zeitlicher Fehler im Kontrollfluss besteht aus folgenden zwei Schritten:

- Zunächst wird der sicherheitskritische Programmcode offline, sprich vor der Ausführung, zerlegt und untersucht. Mit Hilfe der aus der Analyse gewonnenen Zusatzinformationen zum Zeitverhalten wird das Programm anschließend instrumentiert, um bestimmte Prüfpunkte (sog. *Checkpoints*) für den Check während der Ausführung zu setzen. Diese Instrumentierung soll direkt in den Code integriert werden.
- Als zweite Komponente wird eine Überwachungseinheit an den Prozessor angeschlossen. Diese reagiert während der Programmausführung auf die instrumentierten Prüfpunkte, indem sie die jeweiligen Informationen zur WCET-Obergrenze ausliest und weiterverarbeitet. So kann für jedes Codefragment die maximal erlaubte Laufzeit überwacht werden. Bei etwaiger Überschreitung wird die Komponente aktiv, um den aufgetretenen Fehlerfall zu melden.

Die folgenden Abschnitte beschreiben nun im Detail, wie die Instrumentierung sowie die Überprüfung zur Laufzeit funktionieren und ablaufen.

3.1.1 Instrumentierung mit Laufzeitinformationen

Um ein Programm zu analysieren und detaillierte Informationen zum korrekten Ablauf zu gewinnen, ist als erster Schritt eine feingranulare Zerlegung der Applikation nötig. Ziel ist es, ein Programm in Einzelteile aufzuspalten, bei welchen das Zeitverhalten möglichst genau zu bestimmen ist. Abschnitt 2.1 hat gezeigt, dass sich eine Gliederung in Grundblöcke empfiehlt, um eine detaillierte Analyse des Kontrollflusses zu erreichen.

Abbildung 3.1 veranschaulicht nun die Vorgehensweise nach der Zerlegung des Programmcodes: Zunächst wird vor dem Beginn jedes einzelnen Grundblocks ein Checkpoint eingefügt. Aus technischer Sicht besteht ein solcher Checkpoint aus einer oder mehreren Prozessorinstruktionen, welche einen bestimmten Wert für die Überwachungseinheit verfügbar machen (z.B. in ein fest definiertes Register schreiben). Diese zusätzlichen Instruktionen verlängern die Laufzeit des Programms, daher muss eine WCET-Analyse die Laufzeit für die Abarbeitung der eingefügten Checkpoints berücksichtigen. Es ist also nötig, Checkpoints ohne reale WCET-Werte einzufügen, um daraufhin erst die Gesamtlaufzeit der Grundblöcke inklusive der Checkpoints zu ermitteln. Diese Zeiten werden anschließend als reale WCET-Werte in die Checkpoints der einzelnen Grundblöcke geschrieben.

Als Ergebnis liegt instrumentierter Programmcode vor, welcher vor jedem Grundblock einen Checkpoint mit der zugehörigen WCET-Abschätzung enthält. Um dies zur Fehlererkennung zu verwenden, ist ein spezieller Check-Mechanismus im Prozessor erforderlich, welcher im folgenden Abschnitt erläutert wird.

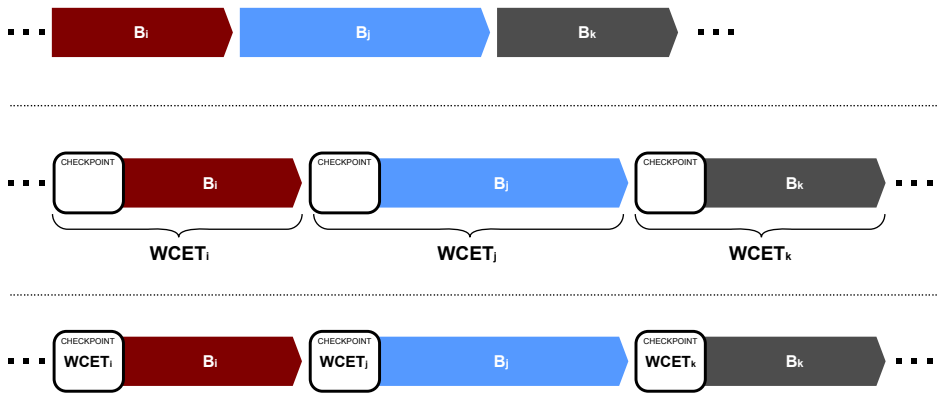


Abbildung 3.1: Einzelschritte bei der Instrumentierung der Grundblöcke (B_x) mit WCET-Werten: Nach der Zerlegung in Grundblöcke werden leere Checkpoints eingefügt und diese daraufhin mit der jeweiligen WCET-Abschätzung befüllt.

3.1.2 Temporaler Check-Mechanismus

An den Prozessor wird eine kleine Hardware-Komponente angeschlossen, welche die WCET-Obergrenzen aus den integrierten Checkpoints ausliest und damit falsches Zeitverhalten im Kontrollfluss erkennt. Sobald bei der Programmausführung ein Checkpoint erreicht ist, schreibt der Prozessor den jeweiligen WCET-Wert beispielsweise in ein definiertes *Checkpoint-Value-* bzw. *CPV*-Register, das mit jeder nachfolgenden Instruktion dekrementiert wird. Durch diesen Mechanismus ist sichergestellt, dass bei korrekter Ausführung das CPV-Register niemals einen negativen Wert erreichen darf. Denn wird ein Grundblock innerhalb der im Vorfeld ermittelten WCET-Grenzen ausgeführt, so kommt der Kontrollfluss bereits vorher an den folgenden Checkpoint und überschreibt den ablaufenden Wert mit einem (in der Regel) höheren Wert. Umgekehrt kann man sagen, dass ein Timing-Fehler vorliegen muss, sobald im CPV-Register ein negativer Wert steht. In diesem Fall wurden im entsprechenden Grundblock mehr Prozessortakte ausgeführt, als in der WCET-Analyse vor der Programmausführung ermittelt wurde. Die Arbeitsweise der Hardware-Komponente ist damit sehr ähnlich zu einem Watchdog-Timer, der in vielen gängigen Mikrocontrollern zu finden ist (vgl. Brinkschulte und Ungerer 2010, S. 79).

3.2 Erkennung logischer Kontrollflussfehler

Neben einer Prüfung des korrekten Timing-Verhaltens ist es auch hilfreich, einen fehlerhaft verzweigten Programmablauf frühzeitig zu identifizieren. Dazu kann der vorliegende Ansatz zur Fehlererkennung – bestehend aus Instrumentierungsmechanismus und hardwareseitiger Check-Komponente – mit nur geringem Mehraufwand erweitert werden, um neben dem Zeitverhalten auch den korrekten logischen Ablauf des Kontrollflusses feststellen zu können.

Das Prüfen einer korrekten logischen Abfolge setzt voraus, dass gewisse Einzelelemente identifizierbar sind. Ist der Programmcode gemäß Abschnitt 2.1 in Grundblöcke zerlegt, so kann deren Abfolge detailliert analysiert werden. Jeder Grundblock wird mit einem eindeutigen Identifikator (ID) versehen; dieser wird in den eingefügten Checkpoints zur jeweiligen WCET-Abschätzung hinzugefügt. Die Hardware-Komponente zur Fehlererkennung wird schließlich so erweitert, dass mit Hilfe dieser instrumentierten Zusatzinformationen überwacht werden kann, ob die Abfolge der Grundblöcke korrekt ist und keine logischen Fehler beim Programmablauf vorliegen.

Anforderung ist, dass später während der Ausführung eine schnelle und einfache Prüfung auf eine korrekte Abfolge ermöglich wird, wobei der Aufwand möglichst begrenzt sein sollte. Dazu werden nun folgende drei Varianten betrachtet und diskutiert:

- **Externe Tabellierung**: Naheliegend ist, die Abfolge der Grundblöcke zu analysieren und die jeweils möglichen Nachfolger zu einem Grundblock tabellarisch festzuhalten. Der Check-Mechanismus könnte dann während der Ausführung den jeweiligen Tabelleneintrag zu einer gegebenen ID auslesen und wüsste über potenzielle Nachfolger im Ablauf Bescheid. Die Instrumentierung und Vergabe der IDs wäre einfach, eine Verteilung ist beliebig möglich. Nachteilig ist zum einen, dass der Check-Mechanismus bei jeder einzelnen ID durch das Auslesen aus einer externen Datenquelle sehr aufwendig, fehleranfällig und zeitintensiv wäre. Zum anderen lässt sich ein hoher Mehraufwand durch das zusätzliche externe Speichern der Tabellendaten nicht verhindern. Unter Beachtung der genannten Anforderungen erweist sich diese Variante somit als impraktikabel.
- **Relation zwischen IDs**: Eine zweite Möglichkeit wäre, die IDs bereits so zu verteilen, dass sie in fester Relation zueinander stehen. Der Check-Mechanismus bei der Ausführung könnte dann durch eine einfache Rechenoperation prüfen, ob ein Grundblock einen korrekten Nachfolger besitzt. Werden beispielsweise die IDs paarweise aufeinanderfolgender Grundblöcke

mit fester Hamming-Distanz[1] in der jeweiligen Binärdarstellung vergeben, könnte die Korrektheit mit Hilfe eines einfachen XOR[2]-Vergleichs festgestellt werden. Bei näherer Betrachtung dieses Ansatzes ist allerdings zu erkennen, dass eine derartige Vergabe der IDs aufgrund der Kontrollstrukturen im Programmablauf sehr hohe Komplexität mit sich bringt. Bei Verzweigungen müssen häufig Zwischenzustände eingefügt werden, um die Relationen korrekt herzustellen. Zusätzlich muss in umgekehrter Richtung sichergestellt sein, dass zwei Grundblöcke, welche nicht aufeinander folgen dürfen, nicht auch (zufällig) bei der Kodierung die festgelegte Hamming-Distanz haben. Sonst würde ein fehlerhafter Sprung fälschlicherweise als korrekt betrachtet werden. Trotz einiger Vorteile bzgl. Aufwands und Einfachheit des Check-Mechanismus erweist sich auch diese Methode aufgrund der hohen Komplexität bei der Instrumentierung als nicht anwendbar.

- **Eingebettete explizite Ankündigung**: Weitaus einfacher erscheint eine dritte Möglichkeit, nämlich die IDs beliebig zu verteilen und die jeweils erlaubten Nachfolger explizit als Instrumentierung innerhalb des Programmcodes anzugeben. Ein Check-Mechanismus während der Ausführung müsste dann vergleichen, ob der tatsächliche Nachfolger mit der Ankündigung potenziell erlaubter Nachfolger übereinstimmt. Die Instrumentierung wäre durch die beliebige Vergabe der IDs einfach und auch der Check-Mechanismus könnte schnell und effektiv arbeiten. Zudem ist ausgeschlossen, dass nicht erlaubte Nachfolger unbeabsichtigt als korrekt gewertet werden, wie es bei der Relation zwischen IDs einfach vorkommen könnte. Einzig kritischer Punkt ist der zusätzliche Aufwand: Da bei näherer Untersuchung (siehe Abschnitt 3.2.1) aber klar wird, dass der Verzweigungsgrad bis auf einige Ausnahmen nicht höher als zwei ist, hält sich der Mehraufwand in Grenzen. Es genügt, bei jedem Grundblock einen oder zwei potenzielle Nachfolger als Ankündigung anzugeben.

Tabelle 3.1 zeigt eine Zusammenfassung der drei vorgestellten Möglichkeiten und bewertet diese hinsichtlich der Komplexität von Instrumentierung und Check-Mechanismus sowie hinsichtlich des entstehenden Mehraufwands. Als Resultat ist zu sehen, dass sich die explizite Ankündigung von Nachfolgern als beste Lösung für die hier beschriebene Problemstellung anbietet, da Instrumentierung sowie Check-Mechanismus einfach zu realisieren sind und der Mehraufwand begrenzt bleibt.

Im Folgenden wird die Methode der expliziten Ankündigung näher erforscht, wobei zunächst die Instrumentierung der Grundblöcke im Vordergrund steht. Es

[1] Die Hamming-Distanz zweier Blöcke fester Länge ist die Anzahl der unterschiedlichen Stellen.
[2] Die *XOR*-Verknüpfung wird auch als *Kontravalenz* bzw. *exklusives Oder* bezeichnet.

	Tabellierung	**Relation**	**Ankündigung**
Instrumentierung	einfach	komplex	einfach
Check-Mechanismus	komplex	einfach	einfach
Mehraufwand	hoch	gering	begrenzt
Bewertung	−	−	+

Tabelle 3.1: Bewertung unterschiedlicher Methoden zur Korrektheitsprüfung der logischen Abfolge von Grundblöcken

ist nötig, auf die möglichen Verzweigungen im Kontrollfluss näher einzugehen und jeweils eine Lösungsmöglichkeit vorzustellen. Anschließend werden der Check-Mechanismus sowie dessen Integration in den Prozessorkern beschrieben.

3.2.1 Instrumentierung durch Ankündigung der Nachfolger

Eine explizite Ankündigung potenzieller Nachfolger ist nur dann sinnvoll, wenn die Liste der Kandidaten überschaubar bleibt. Entscheidend ist es also, den Verzweigungsgrad von Grundblöcken näher zu analysieren, um dadurch die Anzahl der Nachfolger feststellen zu können.

Grundblöcke mit genau einem Nachfolger

Zunächst gibt es Grundblöcke, die am Ende keine Kontrollstruktur beinhalten. In diesem Fall wird definitiv der darauf folgende Block als einzig erlaubter Nachfolger ausgeführt. Abbildung 3.2 zeigt hierzu das Instrumentierungsschema: Vor jeden Grundblock wird ein Checkpoint eingefügt, welcher die ID des jeweiligen Grundblocks selbst sowie die ID des Nachfolgers beinhaltet.

Erfolgt am Ende eines Grundblocks ein (nicht bedingter) Sprung[3], hat dieser ebenfalls genau einen Nachfolger. Abbildung 3.3 zeigt z.B. einen Sprung am Ende von Grundblock B_i zu B_{i+n}. Wie in der Grafik zu sehen ist, wird der Checkpoint von B_i hinsichtlich der ID des Nachfolgers (Block B_{i+n} hat die ID x) entsprechend angepasst.

Neben dem Fall sequentiell aufeinander folgender Grundblöcke und dem Fall eines direkten Sprunges existiert noch die Möglichkeit eines Funktionsaufrufes (*Call*),

[3]Hier werden zunächst nur PC-relative Sprünge betrachtet; indirekte (Register-relative) Sprünge folgen auf Seite 26.

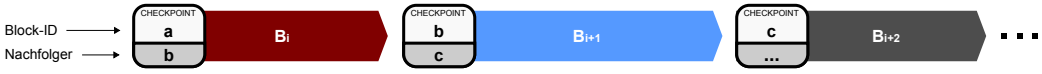


Abbildung 3.2: Instrumentierung sequentiell aufeinander folgender Blöcke

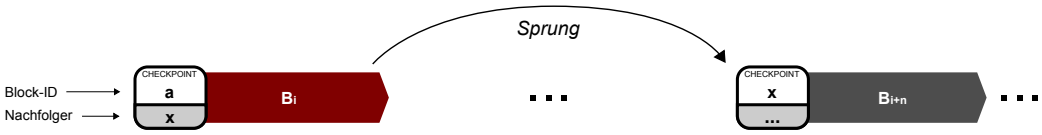


Abbildung 3.3: Instrumentierung der Blöcke bei Sprüngen

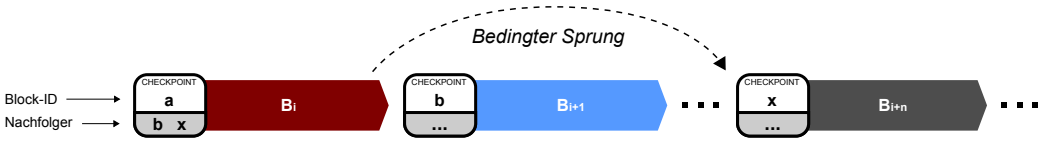


Abbildung 3.4: Instrumentierung der Blöcke bei bedingten Sprüngen

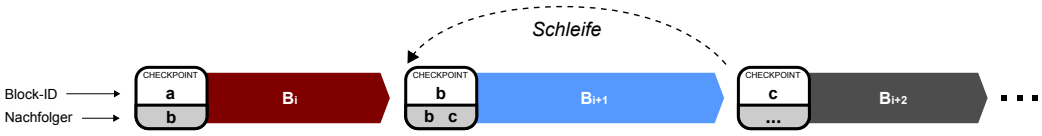


Abbildung 3.5: Instrumentierung der Blöcke bei Schleifen

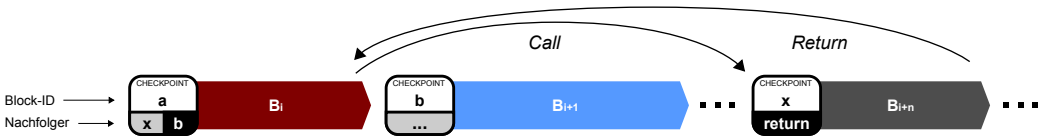


Abbildung 3.6: Instrumentierung der Blöcke bei Funktionen

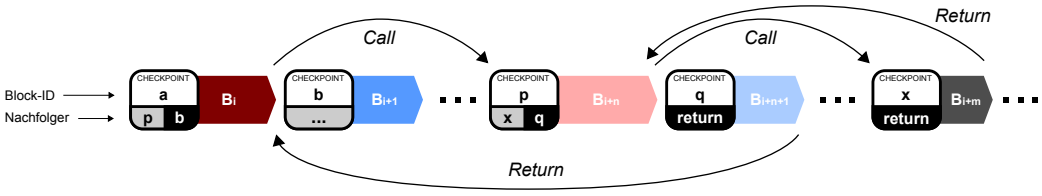


Abbildung 3.7: Instrumentierung der Blöcke bei verschachtelten Funktionen

bei welchem ein Grundblock ebenfalls exakt einen Nachfolger besitzt. Jeder Funktionsaufruf ist allerdings mit einem Rücksprung (*Return*) verbunden, der innerhalb des Funktionskontextes schwieriger zu bewerkstelligen ist. Die Instrumentierung von Call und Return wird deshalb erst im Folgenden näher betrachtet.

Grundblöcke mit zwei möglichen Nachfolgern

Falls ein Grundblock mit einem bedingten Sprung endet, sind im Kontrollfluss zwei Grundblöcke als Nachfolger erlaubt. Abbildung 3.4 zeigt einen bedingten Sprung am Ende von Grundblock B_i zu B_{i+n}. Dies bedeutet, dass nach der Ausführung von B_i entweder der sequentielle Pfad über B_{i+1} oder ein direkter Sprung zu B_{i+n} möglich wären. Instrumentiert wird, wie die Abbildung zeigt, indem als Nachfolger von B_i beide Möglichkeiten (in diesem Fall die IDs b und x) eingetragen werden.

In Abbildung 3.5 ist zu erkennen, dass sich die Instrumentierung bei Schleifen analog verhält. Auch in diesem Fall gibt es genau zwei mögliche Nachfolger zu einem Grundblock. Im Beispiel befindet sich am Ende von B_{i+1} ein bedingter Sprung zurück, um den Grundblock zu wiederholen. Je nach Schleifenbedingung wird also im Anschluss von B_{i+1} entweder erneut B_{i+1} oder sequentiell B_{i+2} ausgeführt. Die Instrumentierung von B_{i+1} beinhaltet daher als potenzielle Nachfolger sowohl die ID b von Block B_{i+1} als auch die ID c von Block B_{i+2}.

Grundblöcke mit mehr als zwei möglichen Nachfolgern

Neben den erläuterten Fällen, in welchen ein Grundblock nur einen oder zwei erlaubte Nachfolger hat, existieren auch Konstellationen, bei denen mehr als zwei Nachfolger möglich sind: Indirekte Sprünge haben kein fest definiertes Sprungziel, sondern machen dieses z.B. vom Wert eines bestimmten Registers abhängig. In dieser Situation kann über die Nachfolger des Grundblocks im Vorfeld keine Aussage getroffen werden. Da indirekte Sprünge auch bei WCET-Analysen Probleme bereiten, ist es in sicherheitskritischen Echtzeitsystemen gängig, Richtlinien zur Programmierung (Bonenfant u. a. 2010) zu definieren, welche den Gebrauch dieser Sprünge verbieten. Eine Lösung zur Ankündigung der Nachfolger bei indirekten Sprüngen kann somit vernachlässigt werden.

Auch bei Funktionsaufrufen und den damit verbundenen Rücksprüngen erweist sich die Ankündigung von Nachfolgern als schwieriger. Da eine Funktion in der Regel von mehreren Stellen im Programmcode aufgerufen wird, ist innerhalb der Funktion nicht bekannt, wohin der Rücksprung erfolgen soll, d.h. welcher Grundblock auf den letzten Grundblock innerhalb des Funktionskontextes folgt. Es muss

bereits zum Zeitpunkt des Funktionsaufrufes eine Information über den aktuellen Punkt der Programmausführung erfasst werden, welche später beim Rücksprung zur Ankündigung des Nachfolgers genutzt werden kann.

Abbildung 3.6 verdeutlicht die Instrumentierung bei Funktionsaufrufen. Im Beispiel sei B_{i+n} eine Funktion, welche am Ende von B_i (und möglicherweise auch an anderen Stellen) aufgerufen wird. Im Checkpoint von B_i wird dazu zunächst x als ID des direkten und einzig erlaubten Nachfolgers angegeben. Zusätzlich wird instrumentiert, welcher Grundblock nach dem Rücksprung von der Funktion auszuführen ist (im Beispiel B_{i+1} mit ID b). Innerhalb der Funktion genügt es somit, im letzten Grundblock (hier B_{i+n}) zu signalisieren, dass ein Rücksprung erfolgt. Der Checker kann in diesem Fall den beim Funktionsaufruf gespeicherten Wert b für die Ankündigung dieses Rücksprungs verwenden.

Dieses System funktioniert analog bei ineinander verschachtelten Funktionsaufrufen, wie Abbildung 3.7 zeigt. B_i ruft die Funktion in B_{i+n} auf, also wird p, die ID von B_{i+n}, als direkter Nachfolger instrumentiert. Für den Rücksprung wird gleichzeitig die ID b von Block B_{i+1} im Checkpoint von B_i vermerkt. Beim Aufruf einer weiteren Funktion innerhalb von B_{i+n} wird dies analog instrumentiert: Als direkter Nachfolger von B_{i+n} wird also x angegeben, für den Rücksprung q. Am Ende jeder Funktion genügt es, ein *return* zu instrumentieren. So kann der Checker jeweils den gespeicherten Wert für den Rücksprung verwenden.

Somit ist der vorliegende Instrumentierungsmechanismus für alle Fälle von Kontrollstrukturen durchführbar. Es müssen bei der Ankündigung in keiner Situation mehr als zwei potenzielle Nachfolge-IDs angegeben werden; auch bei Funktionsaufrufen und Rücksprüngen genügt die Angabe von zwei IDs. Einzige Ausnahme bleiben – wie bereits oben beschrieben – indirekte Sprünge, für welche keine Instrumentierung vorgenommen werden kann.

3.2.2 Logischer Check-Mechanismus

Die Hardware-Komponente zur Korrektheitsprüfung aus Abschnitt 3.1.2 ist direkt an den Prozessor angeschlossen und wird aktiv, sobald ein instrumentierter Checkpoint erkannt wird. Aus technischer Sicht geschieht das z.B. dann, wenn ein Wert in ein bestimmtes fest definiertes Register geschrieben wird. Um zusätzlich die logische Korrektheit zu prüfen, werden bei jeder Aktivität folgende weitere Schritte ausgeführt: Zunächst findet ein Vergleich statt, ob die aktuell ausgelesene ID des Grundblocks der Ankündigung, welche beim vorhergehenden Prüfpunkt gespeichert wurde, entspricht. Im Anschluss wird die aktuelle Ankündigung für den folgenden Checkpoint ermittelt und innerhalb der Check-Einheit

bis zur nächsten Überprüfung gespeichert. Abhängig vom Kontrollfluss muss der Check-Mechanismus unterschiedliche Vergleichsoperationen ausführen. Verschiedene Typen signalisieren dabei, wie die Überprüfung abzulaufen hat:

- **Typ 1 – Ein erlaubter Nachfolger mit ID_x:** In diesem Fall hat der Grundblock nur einen einzigen erlaubten Nachfolger. Die Check-Einheit muss die ID des nächsten Prüfpunktes mit der angekündigten ID_x auf Gleichheit prüfen.
- **Typ 2 – Auswahl zwischen ID_x oder ID_y:** Der Grundblock besitzt zwei potenzielle Nachfolger. Beim nächsten Checkpoint ist zu prüfen, ob die ID entweder identisch zu ID_x oder ID_y ist.
- **Typ 3 – Funktionsaufruf, Sprung zu ID_x mit Rücksprung zu ID_y:** Es wird eine Funktion aufgerufen, welche mit ID_x beginnt. Der nächste Checkpoint muss folglich ID_x beinhalten. Für den Rücksprung wird ID_y vorgemerkt, indem sie innerhalb der Check-Komponente auf einen Stack-Speicher gelegt wird. So können auch ineinander verschachtelte Funktionsaufrufe problemlos behandelt werden.
- **Typ 4 – Rücksprung von einer Funktion:** Um den nachfolgenden Rücksprung von der Funktion auf Korrektheit zu prüfen, muss der oberste Wert vom Stack-Speicher ausgelesen werden. Dieser wird dann mit der ID des folgenden Grundblocks verglichen und vom Stack-Speicher entfernt.

Die Erweiterung des Check-Mechanismus ist relativ einfach umsetzbar und kann während der Programmausführung schnell und effizient arbeiten. Für die Ankündigung muss temporärer Speicher für maximal zwei IDs und den Typ, sowie ein Stack-Speicher für Funktionsaufrufe bereitstehen. Die Größe dieses Speichers richtet sich dabei nach der erlaubten Verschachtelung von Funktionen. Diese ist in der Regel durch die Prozessorarchitektur festgelegt.

Treten bei der Ausführung Interrupts[4] auf, kann ebenfalls der Stack-Speicher verwendet werden, um den bzw. die aktuell angekündigten Nachfolger zu speichern und nach Beendigung der Unterbrechung wiederherzustellen. Das Auftreten eines Interrupt-Signals muss dazu von der Check-Einheit erkannt werden, um die entsprechenden Operationen ausführen zu können. In den Checkpoints können keine Angaben zum Sprung in die Interrupt Service Routine gemacht werden, da zum Zeitpunkt der Instrumentierung keinerlei Aussage über deren Auftreten gemacht werden kann.

[4] Ein *Interrupt* bezeichnet eine kurzfristige Programmunterbrechung, meist durch ein Signal einer Hardware-Komponente. Es wird eine zugeordnete *Interrupt Service Routine* ausgeführt. Nach deren Beendigung kann das Programm an der Unterbrechungsstelle fortgesetzt werden.

3.3 Diskussion erkennbarer Fehler

Entsprechend der beschriebenen Motivation hat der vorgestellte Mechanismus das Ziel, Kontrollflussfehler wirksam und zuverlässig zu erkennen. Der Fokus liegt auf transienten Fehlerursachen, welche zu einem hohen Prozentsatz als Auslöser für Kontrollflussfehler verantwortlich sind. Um die Effektivität der Erkennungstechnik zu analysieren, ist es hilfreich, anhand eines Fehlermodells genauer zu ermitteln, wie Kontrollflussfehler entstehen und welche davon mit Hilfe des vorgestellten Mechanismus erkennbar sind.

3.3.1 Definition eines Fehlermodells

Im vorliegenden Kontext sollen nicht systematische, transiente Fehlerursachen in Form sog. *Single Event Upsets (SEUs)* betrachtet werden. SEUs werden in Halbleitern beim Durchgang hochenergetischer ionisierender Teilchen hervorgerufen und äußern sich in der Regel als *Bitflip* (d.h. als Änderung des Zustands eines Bits) in Speicher- oder Registerelementen (Wang und Agrawal 2008). SEUs können zu einer Fehlfunktion des betroffenen Bauteils führen, richten aber keinen dauerhaften Schaden an. Für das Fehlermodell wird angenommen, dass zu einem bestimmten Zeitpunkt nur jeweils ein Bitflip eine Komponente trifft, zumal mehrfache Bitflips äußerst unwahrscheinlich sind. Die betroffenen Komponenten können prozessorinterne Elemente wie Register oder Befehlszähler sein oder auch der angeschlossene Instruktions- oder Datenspeicher. Zusätzlich sei angenommen, dass die eingesetzte Hardwarekomponente zur Korrektheitsprüfung so robust gebaut ist, dass eine dortige Fehlfunktion sehr unwahrscheinlich und daher zu vernachlässigen ist.

Im Folgenden sollen die Konsequenzen eines Bitflips im Befehlszähler, im Instruktionsspeicher sowie in anderen Speicherelementen und Registern näher beleuchtet werden. Es ist interessant, welche Auswirkungen auf den Kontrollfluss entstehen und inwiefern diese durch den entworfenen Mechanismus zu erkennen sind.

3.3.2 Fehler im Befehlszähler

Der *Befehlszähler* oder auch Programmschrittzähler (engl. *Program Counter*) ist ein spezielles Register innerhalb eines Rechners, das die Speicheradresse der aktuell auszuführenden Instruktion enthält. Diese Adresse wird während der Programmausführung stets ausgelesen und erhöht, ehe der Befehl ausgeführt wird. Kontrollstrukturen, die ein Abweichen der Ausführung von der sequenziellen Reihenfolge bewirken, setzen den Befehlszähler auf den entsprechenden Wert.

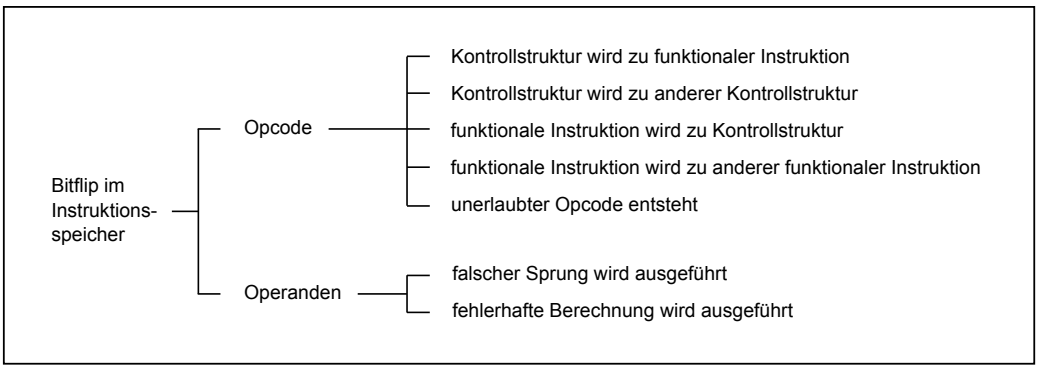


Abbildung 3.8: Kategorisierung der Konsequenzen eines potenziellen Bitflips im Instruktionsspeicher

Trifft ein Bitflip den Befehlszähler und verändert die gesetzte Speicheradresse, so wird infolgedessen ein nicht vorgesehener Befehl ausgeführt. Das Programm wird auf nicht erlaubte Weise fortgeführt, ein Kontrollflussfehler tritt auf. In Bezug auf den vorgestellten Erkennungsmechanismus ist nun zu unterscheiden, ob der fehlerhafte Sprung den aktuellen Grundblock verlassen hat oder nicht. Ist dies der Fall, so kann mit Hilfe des logischen Check-Mechanismus beim Erreichen des folgenden Checkpoints erkannt werden, dass die ID nicht der Ankündigung entspricht. Bewirkt der Sprung kein Verlassen des Grundblocks bzw. einen Sprung in einen ebenfalls erlaubten Nachfolger, so bleibt die Möglichkeit, dass durch die veränderte Ausführung die angegebene WCET-Obergrenze überschritten wurde. In diesem Fall ist eine Fehlererkennung noch vor Erreichen des folgenden Checkpoints möglich. Auch das Beachten einer möglichen Unterschreitung der BCET-Abschätzung könnte die Erkennung derartiger Fehler weiter verbessern.

3.3.3 Fehler im Instruktionsspeicher

Bitflips im Instruktionsspeicher können entweder den *Opcode* oder die *Operanden* eines Befehls treffen. Ein Opcode (bzw. operation code) ist durch einen oder mehrere Zahlenwerte kodiert und spezifiziert einen bestimmten Maschinenbefehl. Die Menge aller Opcodes eines Prozessors bilden dessen Befehlssatz. Auf den Opcode folgen eventuell ein oder mehrere Bytes an zu verarbeitenden Daten, die Operanden des Befehls. Im Folgenden sollen zunächst Fehler an den Opcodes und daraufhin Fehler an den Operanden betrachtet werden. Eine Klassifikation mit möglichen Konsequenzen findet sich in Abbildung 3.8.

	Opcode nach einzelnem Bitflip		
urspr. Opcode	unerlaubt	Kontrollstruktur	funkt. Instruktion
Kontrollstruktur	25.0 %	45.5 %	29.5 %
funkt. Instruktion	25.2 %	7.7 %	67.1 %
beliebiger Befehl	25.1 %	16.1 %	58.7 %

Tabelle 3.2: Wahrscheinlichkeiten für den Übergang eines gültigen Opcodes in einen fehlerhaften Opcode nach einem einzelnen Bitflip am Beispiel des CarCore-Prozessors

Fehler im Opcode eines Befehls

Um diesen Fall näher zu untersuchen, ist es nützlich, die Instruktionen in zwei verschiedene Kategorien einzuteilen: Zum einen gibt es *funktionale* Instruktionen, welche in irgendeiner Form der Datenverarbeitung (z.B. durch arithmetische Berechnungen, Transferoperationen usw.) dienen, zum anderen Befehle, die eine *Kontrollstruktur* repräsentieren (bedingte und nicht bedingte Sprünge, Funktionsaufrufe, Rücksprünge). Im Fehlerfall kann es dazu führen, dass eine Kontrollstruktur durch Bitmodifikation zu einer funktionalen Instruktion wird und umgekehrt. Ebenso ist es möglich, dass sich eine Kontrollstruktur fälschlicherweise in eine andere Kontrollstruktur bzw. sich eine funktionale Instruktion in eine andere funktionale Instruktion verwandelt. Schließlich besteht die Option, dass durch einen Bitflip ein unerlaubter Opcode entsteht, welcher so keiner existierenden Instruktion mehr entspricht.

Betrachtet man konkret den Befehlssatz eines bestimmten Prozessors, so lassen sich Wahrscheinlichkeiten für die jeweils fehlerhaften Opcodes nach einem Bitflip bestimmen. Im vorliegenden Fall bietet es sich an, den echtzeitfähigen CarCore-Prozessor für eine Analyse zu verwenden, der im Rahmen des Kapitels zur Implementierung (Abschnitt 4.1) noch näher vorgestellt wird. Tabelle 3.2 zeigt die Werte für den Befehlssatz dieses Prozessors im Einzelnen: Angenommen es handelt sich um einen Opcode einer Kontrollstruktur, so verwandeln 45.5 % der möglichen einzelnen Bitflips innerhalb des Opcodes diese Instruktion zu einer anderen Kontrollstruktur und 29.5 % zu einer funktionalen Instruktion. Die Wahrscheinlichkeit für den Opcode einer funktionalen Instruktion in eine Kontrollstruktur geändert zu werden, liegt bei 7.7 %, in eine andere funktionale Instruktion bei 67.1 %. Die Wahrscheinlichkeit, dass ein fehlerhafter Opcode vom Prozessor selbst als unerlaubt erkannt wird, ist mit etwa 25 % für alle Befehlsarten gleich.

Es stellt sich nun die Frage, wie viele dieser Opcode-Änderungen durch den vorgestellten Check-Mechanismus erkennbar sind. Leider ist dies auf analytische Weise äußerst komplex zu ermitteln, da sehr viele Einzelfälle zu beachten sind. Besonders wenn durch die Opcode-Änderung die Laufzeit verlängert wird und dadurch die WCET-Abschätzung bis zum folgenden Checkpoint überschritten wird, variieren die Erkennungsmöglichkeiten je nach Anwendungsszenario. Dennoch lassen sich zu den einzelnen Kategorien und den jeweiligen Möglichkeiten einer Fehlererkennung einige Aussagen treffen:

- **Kontrollstruktur wird zu funktionaler Instruktion:** Hierbei ist für die Fehlererkennung die Art der Kontrollstruktur entscheidend. Handelt es sich um einen bedingten Sprung, der fälschlicherweise zu einer funktionalen Instruktion wird, so wäre ja die sequentielle Fortführung des Programms (ohne den Sprung zu nehmen) erlaubt. Somit kann nicht entschieden werden, ob der sequentielle Ablauf die Folge eines nicht genommenen Sprunges oder eines Fehlers ist. Wird jedoch eine andere Art von Kontrollstruktur fälschlicherweise als funktionale Instruktion gewertet, ist die Fehlererkennung nach vorliegender Methode erfolgreich.

- **Kontrollstruktur wird zu anderer Kontrollstruktur:** Auch dieser Fall ist von der Art der Kontrollstruktur abhängig. Entspricht das Sprungziel weiterhin der Ankündigung, kann keine Erkennung stattfinden. Allerdings sind die Ausführungszeiten der Kontrollstrukturen sehr verschieden: Wird ein Sprung fälschlicherweise zu einem Funktionsaufruf, würde dies höchstwahrscheinlich zu einer Überschreitung der angegebenen WCET-Grenze führen. Berücksichtigt man zusätzlich die untere Grenze der Ausführungszeit, würde der umgekehrte Fall vermutlich eine Unterschreitung der BCET-Abschätzung verursachen.

- **Funktionale Instruktion wird zu Kontrollstruktur:** Dieser Fall ändert definitiv den Kontrollfluss, die vorliegende Fehlererkennung würde die Abweichung vom erlaubten Ablauf feststellen. Ausnahme ist nur der unwahrscheinliche Fall, dass das Sprungziel ein als erlaubt deklarierter Nachfolger ist. Aber auch in diesem Fall kann es leicht zu einer Überschreitung der angegebenen WCET-Abschätzung kommen, was wiederum eine erfolgreiche Fehlererkennung zur Folge hätte.

- **Funktionale Instruktion wird zu anderer funktionaler Instruktion:** Da diese Situation prinzipiell keine Kontrollflussänderung bewirkt, ist eine Fehlererkennung nur dann möglich, wenn sich eine höhere Ausführungszeit und damit ein Überschreiten der ermittelten WCET-Grenze des Grundblocks ergibt.

- **Unerlaubter Opcode entsteht:** In der Regel sind nicht alle Opcodes im Befehlssatz eines Prozessors belegt. Somit kann der Fall auftreten, dass durch einen Bitflip ein Opcode entsteht, welcher so keiner Instruktion entspricht. Dieser Fehler wird normalerweise prozessorintern identifiziert, sodass eine Fehlererkennung durch den angeschlossenen Check-Mechanismus nicht weiter nötig ist.

Fehler in den Operanden eines Befehls

Betrifft der Bitfehler nicht den Opcode, sondern den bzw. die Operanden einer Instruktion, muss ebenso zwischen funktionalen Instruktionen und Kontrollstrukturen unterschieden werden. Bei funktionalen Instruktionen führt ein falscher Operand in der Regel zu einem Fehler in der Berechnung, was in der Folge allerdings eher unwahrscheinlich Auswirkungen auf den Kontrollfluss hat. Ändert sich hingegen bei Kontrollstrukturen, insbesondere bei Sprüngen und Funktionsaufrufen der Operand, so wird der Programmablauf an einer nicht vorhergesehenen Stelle fortgeführt. Ähnlich wie bei einem Fehler im Befehlszähler hängen die Erkennungsmöglichkeiten nun davon ab, ob der aktuelle Grundblock durch den falschen Sprung verlassen wurde oder nicht. Im ersteren Fall ist eine Erkennung mit Hilfe der zugewiesenen IDs sehr wahrscheinlich, im zweiten Fall kann eine Erkennung beim Überschreiten der angegebenen WCET-Obergrenze stattfinden.

3.3.4 Fehler im Datenspeicher und in Registern

Kommt es zu einem Fehler im Datenspeicher oder in prozessorinternen Registern, ändern sich die Ergebnisse bei der Datenverarbeitung, was in der Konsequenz ebenso zu Änderungen im Kontrollfluss führen kann. So ist es beispielsweise möglich, dass eine Schleife häufiger als erlaubt durchlaufen wird oder die Sprungbedingungen für bestimmte Kontrollstrukturen fehlerhaft sind. Die vorliegende Methode kann diese semantischen Faktoren allerdings nicht prüfen; derartige Änderungen werden nur als Fehler erkannt, wenn eine zeitliche Überschreitung der angegebenen maximalen Ausführungszeit innerhalb der Grundblöcke die Folge ist.

3.3.5 Fazit

Eine Analyse der modellierten transienten Fehlerursachen ergibt, dass eine Vielzahl von Bitflips besonders im Instruktionsspeicher zu Kontrollflussfehlern führen kann. Die Möglichkeiten, derartige Fehler durch die vorgestellte Methodik zuverlässig und wirkungsvoll zu erkennen, sind vielversprechend. Jedoch reicht

dieser theoretische Ansatz noch nicht aus, um aussagekräftige Werte zu erhalten. Vielmehr ist eine realitätsnahe Simulation mit einer künstlichen Erzeugung verschiedener Fehlerfälle nötig, um die Abdeckung der Fehlererkennung sinnvoll zu evaluieren. In diesem Zusammenhang ist auch die Erkennungslatenz ein wichtiger Faktor: Durch die feingranulare Instrumentierung sind zwar sehr niedrige Werte zu erwarten, allerdings kann erst eine Simulation realistische Ergebnisse liefern.

Neben der Qualität der Fehlererkennung spielen aber, wie zu Beginn des Kapitels erwähnt, noch weitere Kriterien eine wichtige Rolle bei der Bewertung der Methodik. Dies sind die zusätzliche Ausführungszeit, welche aus der Code-Instrumentierung resultiert, sowie der erhöhte Speicherbedarf. Ebenso muss der Hardware-Aufwand bewertet werden, welcher durch die Integration der Check-Einheit entsteht. Im folgenden Kapitel soll eine Implementierung von Instrumentierungs- und Prüfmechanismus beschrieben werden, welche es ermöglicht die genannten Kriterien anhand verschiedener Beispielprogramme zu evaluieren. Gleichzeitig wird diese Implementierung die Grundlage für eine Simulation mit künstlicher Fehlererzeugung sein, was im Rahmen der Evaluierungen in Kapitel 5 weiter ausgeführt wird.

3.4 Abgrenzung zu verwandten Arbeiten

Seit mehreren Jahrzehnten werden Mechanismen zur Korrektheitsprüfung und Überwachung von Computersystemen erforscht. Dieser Abschnitt soll einen Überblick über einige Techniken geben, die einen engen Bezug zur vorliegenden Arbeit aufweisen. Neben einer kurzen Beschreibung der jeweiligen Ideen und Konzepte gilt es, die Vor- und Nachteile der verschiedenen Ansätze zu betonen und die Techniken im Hinblick auf den entworfenen Mechanismus zur Fehlererkennung abzugrenzen.

Die entwickelten Ansätze können prinzipiell in drei Kategorien eingeteilt werden, abhängig von der Ebene der implementierten Erkennungsmechanismen: *Hardwarebasierte* Techniken erweitern einen Prozessor um eine zusätzliche Hardware-Komponente, welche der Überwachung und Fehlererkennung dient. *Softwarebasierte* Ansätze ergänzen eine Applikation um zusätzliche Instruktionen, um durch eine redundante Ausführung bestimmter sicherheitskritischer Programmteile ein höheres Maß an Sicherheit zu erreichen. Ein Anschluss zusätzlicher Hardware ist bei diesen Techniken nicht nötig. Schließlich existieren Mischformen, *hybride* Mechanismen, welche ein Zusammenspiel aus softwareseitiger Absicherung im Programmcode und hardwareseitiger Prüfung realisieren.

3.4.1 Hardwarebasierte Techniken zur Fehlererkennung

Eine der grundlegendsten Techniken, um das Systemverhalten zu überwachen und Programmverklemmungen und Abstürze zu erkennen, ist der Gebrauch eines *Watchdog-Timers* (Brinkschulte und Ungerer 2010, S. 79). Ein Programm muss dazu in bestimmten Zeitintervallen immer wieder Signale als „Lebenszeichen“ an den Watchdog senden. Erhält der Watchdog-Timer kein rechtzeitiges Signal, wird ein Fehler gemeldet. Die Möglichkeiten sind allerdings begrenzt: Einerseits kommt es vor, dass der Timer trotz einer Fehlfunktion des Systems weiterhin ein rechtzeitiges Signal empfängt. Andererseits sind die Auslösezeiten nur in sehr ungenauen Stufen anzugeben[5], was in den meisten Fällen eine hohe Erkennungslatenz zur Folge hat. Es bleibt zudem vollständig dem Programmierer überlassen, an geeigneter Stelle Prüfpunkte zu setzen und die maximal erlaubten Zeiten abzuschätzen.

Höher entwickelte hardwarebasierte Techniken zur Fehlererkennung nutzen meist sog. *Watchdog-Prozessoren*. Nach Mahmood und McCluskey (1988) ist dies ein kleiner und einfach arbeitender Coprozessor, der das Verhalten eines Hauptprozessors zur Laufzeit überwacht. Der Watchdog-Prozessor ist über den Daten- und Adressbus sowohl mit dem Hauptprozessor als auch mit dem Speicher verbunden. Die Arbeitsweise ist verschiedenartig: Mahmood und McCluskey (1988) geben hierzu einen guten Überblick, indem sie verschiedene Mechanismen vergleichen und diskutieren. Die Aufgaben eines Watchdog-Prozessors gehen über das Erkennen von Kontrollflussfehlern hinaus. Nach Rhod u. a. (2008) lassen sich diese in drei Kategorien einteilen:

- *Checks beim Speicherzugriff* stellen sicher, dass keine unerwarteten Lese- und Schreibzugriffe auf den Speicher ausgeführt werden. Als Beispiel gilt hier die Arbeit von Namjoo und McCluskey (1982), in welcher der Watchdog-Prozessor zu jedem Zeitpunkt der Ausführung überwacht, auf welchen Teil des Programmcodes und der Daten zugegriffen wird und bei unerwartetem Verhalten ein Fehlersignal liefert.

- *Checks auf Konsistenz* von Variablen bestehen darin, den Wert einer Variablen auf Plausibilität zu prüfen. Bei Mahmood u. a. (1983) ist dem Watchdog-Prozessor der Ausführungskontext bekannt und dieser kann zu jedem Wert, welchen der Hauptprozessor schreibt oder liest, sicherstellen, dass dieser in einem gewissen zu erwartenden Bereich liegt bzw. in einer bestimmten Relation zu anderen Variablen steht.

[5] Brinkschulte und Ungerer (2010, S. 224) zeigen am Beispiel des Mikrocontrollers ATmega128A, dass die Auslösezeit in 8 möglichen Stufen zwischen 14 Millisekunden und 1,8 Sekunden einstellbar ist.

- In der dritten Kategorie prüfen *Kontrollfluss-Checks*, ob die genommenen Sprünge und Verzweigungen eines Programmes mit dem Kontrollflussgraphen, welcher die erlaubten Abläufe der Applikation beinhaltet, übereinstimmen. Diese Vergleiche erfolgen auf Basis von Signaturen, die bereits beim Kompilieren erzeugt und als Referenzwerte statisch gespeichert werden. Während der Ausführung werden die Signaturen neu berechnet, um durch Vergleiche mit den Referenzwerten eine Aussage über den korrekten Programmablauf treffen zu können. Diese Techniken sollen im Folgenden näher betrachtet werden.

In Abhängigkeit vom Speicherort der Signaturen ergeben sich zwei Typen von Erkennungsmechanismen: Auf der einen Seite ist es möglich, die Signaturen direkt in den Programmcode einzubetten (auch *Embedded Signature Monitoring* bzw. *ESM* genannt), auf der anderen Seite kann ein spezieller Speicher an den Watchdog-Prozessor angeschlossen werden, in welchem die Signaturen abgelegt sind (*Autonomous Signature Monitoring* bzw. *ASM*).

Häufig erwähnt werden ESM-Techniken von Lu (1982), Namjoo (1982), Schuette und Shen (1987), Wilken und Shen (1990) sowie Ohlsson und Rimen (1995). Während der Ausführung werden die eingebetteten Signaturen zur Auswertung an den Watchdog-Prozessor weitergeleitet. In der Zwischenzeit führt der Hauptprozessor *NOP*-Befehle[6] aus.

Beim Ansatz von Lu (1982), genannt *Structural Integrity Checking (SIC)*, werden an bestimmten Stellen Signaturen als explizite Labels im Programm eingefügt. Der Compiler generiert schließlich zwei Programme: die Applikation für den Hauptprozessor sowie ein Check-Programm, das auf dem angeschlossenen Watchdog-Prozessor läuft. Während der Ausführung werden die gelesenen Labels an den Watchdog gesendet und das Check-Programm überprüft die Korrektheit der Ausführung durch den Ablauf der empfangenen Signaturen. Nachteile von SIC sind ein hoher Speicherverbrauch durch das Hinzufügen vieler Signaturen sowie eine entsprechend höhere Ausführungszeit, dadurch dass die Signaturen explizit zum Watchdog-Prozessor geschickt werden müssen.

Namjoo (1982) entwickelte eine andere ESM-Technik: *Path Signature Analysis (PSA)*. Die Signaturen werden dabei aus dem Verlauf des Kontrollflusses abgeleitet, z.B. können diese die XOR-Verknüpfung abgearbeiteter Befehlswörter sein, bestimmte Check-Summen oder ähnliches. An bestimmten Stellen werden statische Zwischenergebnisse als Referenzwerte für die Prüfung gespeichert. Während der Ausführung berechnet der Watchdog-Prozessor parallel zum Hauptprogramm die Signatur aus den jeweils abgearbeiteten Befehlen. Sobald ein Referenzwert

[6] NOP steht für eine *Nulloperation*: ein Befehl für Prozessoren, der effektiv nichts tut.

im Programmcode auftaucht, wird dieser vom Hauptprozessor an den Watchdog übermittelt, sodass durch einen Vergleich mit der aktuell berechneten Signatur die korrekte Ausführung des Programms festgestellt werden kann. *Branch Address Hashing (BAH)* (Shen und Schuette 1983, Schuette und Shen 1987) verbessert PSA, indem die Signaturen mit Adressen von Sprungzielen kombiniert werden. So kann der Speicherbedarf für Signaturen um 50 % reduziert werden (vgl. Mahmood und McCluskey 1988). Den Aufwand herabzusetzen ist auch das Ziel von Wilken und Shen (1990); die eingefügten Signaturen werden auf ein Minimum reduziert, wobei gegenüber PSA dennoch die Abdeckung erkannter Fehlerfälle erhöht und die Latenz bei der Erkennung verringert werden. Ein alternativer Ansatz wird von Ohlsson und Rimen (1995) als *Implicit Signature Checking (ISC)* beschrieben, in welchem der Watchdog-Prozessor implizite Signaturen aus den Instruktionsadressen als Referenzwerte verwendet. Auch hier ist es das Ziel, den Speicherverbrauch gegenüber vorherigen Ansätzen weiter zu vermindern.

Einen anderen Weg gehen ASM-Ansätze wie der von Michel u. a. (1991). Um Overhead im Programmcode durch zusätzliche Instruktionen für den Watchdog-Prozessor zu vermeiden, werden die beim Kompilieren berechneten Signaturen in einem separat an den Watchdog angeschlossenen Speicher abgelegt, die dieser selbstständig auslesen kann. Dadurch müssen auch vorhandene Applikationen vor der Ausführung nicht neu kompiliert werden, um die Signaturen einzufügen. Die Arbeit von Arora u. a. (2006) beinhaltet einen sog. *Intra-Procedural Control Flow Checker*, welcher die Korrektheit anhand einer extern gespeicherten Tabelle überprüft. Ziener und Teich (2009) erweitern diese Ideen, indem sie zum Hauptprozessor einen unabhängigen zweiten Befehlszähler hinzufügen, welcher die Korrektheit des Kontrollflusses überwacht und eine Fehlererkennung mit nur sehr geringer Latenz ermöglicht.

Kein Watchdog-Prozessor zur Erkennung von Fehlern im Kontrollfluss, aber eine häufig genannte hardwarebasierte Technik zur Fehlererkennung kommt bei Austin (1999) als *Dynamic Implementation Verification Architecture (DIVA)* zum Einsatz. Hier wird ein zweiter, kleinerer (Checker-)Core an die Prozessorpipeline angeschlossen, um die Korrektheit aller Berechnungen des Hauptprozessors zu prüfen. Arithmetisch-logische Operationen und Speicherzugriffe werden damit redundant ausgeführt. Nur wenn das Ergebnis der Berechnung durch den Hauptprozessor dem des Checker-Core entspricht, wird dieses an die Commit-Stufe der Pipeline weitergegeben. Zwar ist der Mehrverbrauch an Hardware bei der DIVA-Technik geringer als bei vollständig redundanter Ausführung, doch es kommt durch eine verlängerte Prozessor-Pipeline auch zu Einbußen hinsichtlich Performanz.

Im Hinblick auf Echtzeitsysteme existieren Ansätze, um das Zeitverhalten zur Ausführungszeit zu überwachen. Dabei liegt der Fokus allerdings weniger auf einer

Fehlererkennung zur Laufzeit; vielmehr sollen Möglichkeiten geschaffen werden, um Echtzeitbedingungen einfach analysieren und testen zu können bzw. Systeme auf Programmierfehler zu untersuchen. Die Arbeit von Tsai u. a. (1990) präsentiert eine hardwarebasierte nicht-invasive *Architektur zum Monitoring verteilter Echtzeitsysteme*. Dazu wird eine programmierbare Kontrolleinheit an das Bus-System eines Rechners angeschlossen, um das Zeitverhalten zu überwachen ohne die eigentliche Ausführung zu beeinflussen. Die zu testenden Bedingungen müssen manuell vom Programmierer mit Hilfe von Breakpoints oder durch das Setzen bestimmter Variablen eingefügt werden.

Zusammenfassend lässt sich feststellen, dass es einige vielversprechende Techniken zur Fehlererkennung gibt, die einen rein hardwarebasierten Ansatz verfolgen. Allerdings fokussieren die Mechanismen fast ausschließlich auf logische Korrektheit des Kontrollflusses, weniger auf Fehler im Zeitverhalten. Der Nutzen bei einem Einsatz in harten Echtzeitsystemen ist somit bei den meisten Techniken fraglich.

Ein großer Vorteil der Hardware-Techniken besteht darin, dass geringe bzw. überhaupt keine Modifikationen am Programmcode nötig sind. Dies zahlt sich insbesondere aus, wenn ein erneutes Kompilieren unmöglich ist, da der Quellcode einer Applikation nicht zur Verfügung steht. Auf der anderen Seite beinhalten die Hardware-Techniken aber auch Schwächen: Die Komplexität der zusätzlich benötigten Komponenten ist meist sehr hoch und eine Anpassung muss spezifisch für jeden Prozessor vorgenommen werden, was sowohl zu einem hohem Entwicklungsaufwand als auch zu einer schlechten Portabilität der Techniken führt.

Vergleicht man die Funktionsweise des neu entwickelten Mechanismus zur Erkennung von Kontrollflussfehlern in Echtzeitsystemen mit den vorgestellten Hardware-Techniken, so besteht aufgrund der im Programmcode eingebetteten Checkpoints eine gewisse Ähnlichkeit zu den beschriebenen ESM-Techniken, welche die Signaturen ebenfalls in den Programmcode integrieren. Die Signaturen werden allerdings direkt an den Watchdog übermittelt, während der Hauptprozessor mit NOP-Befehlen pausiert. Im Gegensatz dazu erweitern die Checkpoints als zusätzliche Instruktionen die Software und werden vom Hauptprozessor selbst ausgeführt und an den angeschlossenen Hardware-Checker weitergegeben. Dies erlaubt es, die Funktionalität des Check-Mechanismus auf ein Minimum zu reduzieren. Es ist lediglich eine sehr kleine Erweiterung auf Hardware-Ebene nötig, ohne dabei tief in den bestehenden Prozessor eingreifen zu müssen. Im Hinblick auf Echtzeitsysteme fokussiert lediglich die Arbeit von Tsai u. a. (1990) auf eine Korrektheit im Zeitverhalten. Allerdings werden dabei – wie bereits erwähnt – hauptsächlich Programmierfehler untersucht. Die Zielsetzung, Timing-Fehler ausgelöst durch transiente Fehlerursachen mit möglichst geringer Erkennungslatenz zu identifizieren, fehlt leider völlig.

3.4.2 Softwarebasierte Techniken zur Fehlererkennung

Softwarebasierte Techniken zur Fehlererkennung, in der Literatur auch *Software Implemented Hardware Fault Tolerance (SIHFT)*-Techniken genannt (vgl. Rhod u. a. 2008), nutzen verschiedene Konzepte von Redundanz: Entweder es werden zusätzliche Informationen zur Absicherung eingefügt oder der Programmcode wird mehrfach ausgeführt, entweder parallel oder nacheinander. Die Ansätze von Randell (1975) und Avizienis (1985) zählen zu den ältesten auf diesem Gebiet; hierbei wird die Ausführung einer Applikation einfach repliziert, um ein höheres Maß an Sicherheit zu gewährleisten. Zwar wird dieses Ziel erfüllt, doch da diese Techniken es dem Softwareentwickler überlassen, manuell die Abschnitte für eine doppelte Ausführung und den Zeitpunkt für deren Auswertung zu finden, ist ein Praxiseinsatz nur schwer möglich.

In jüngerer Zeit wurde von Oh u. a. (2002c) ebenfalls eine reine Software-Lösung, *Error Detection by Duplicated Instructions (EDDI)*, entwickelt, welche die Instruktionen eines Programms dupliziert und an geeigneten Stellen automatisierte Prüfungen integriert. Die zusätzlichen Instruktionen werden vom Compiler generiert und in das Originalprogramm eingeflochten. Jede duplizierte Ausführung erfolgt dabei unter Nutzung anderer Register und Speicherstellen, um die Originale nicht zu beeinflussen. An bestimmten Synchronisationspunkten werden schließlich Instruktionen eingefügt, um zu prüfen, ob das Originalprogramm und die duplizierten Teile zum selben Ergebnis gekommen sind. *Error Detection by Diverse Data and Duplicated Instructions (ED^{4}I)* (Oh u. a. 2002a) geht noch einen Schritt weiter: Das Originalprogramm wird in eine modifizierte Version mit gleicher Funktionalität transformiert, welche gleichzeitig zum Original ausgeführt wird, um die Zwischenergebnisse auf Korrektheit zu vergleichen. Diese Methodik verbessert die Möglichkeiten, transiente und permanente Fehler zu erkennen, allerdings ist der enorme Mehraufwand hinsichtlich Ausführungszeit in vielen Einsatzgebieten nicht realisierbar.

Weitere Techniken, die innerhalb der letzten Jahre entwickelt wurden, schützen vor Fehlern, indem nur bestimmte Instruktionen zur Absicherung eines korrekten Kontrollflusses automatisiert in den Programmcode eingefügt werden. Dies senkt vor allem die zusätzliche Ausführungszeit, da nicht mehr die vollständige Anwendung, sondern nur noch bestimmte sicherheitskritische Teile redundant ausgeführt werden. Diese Mechanismen, auch *Control Flow Checking*-Techniken genannt, basieren auf einer Aufteilung des Programmes in Grundblöcke. Der Ansatz *Control-Flow Checking Using Assertions (CCA)*, der bei Kanawati u. a. (1996) evaluiert wird, fügt sowohl am Beginn als auch am Ende von Grundblöcken bzw. Intervallen ohne Verzweigungen eine Absicherung ein. Implementiert wird CCA

als Präprozessor eines Compilers; dabei wird keine detaillierte Analyse des Kontrollflusses und auch keine Erzeugung des Kontrollflussgraphen nötig. Interessant ist auch die Erweiterung zu *Enhanced Control-Flow Checking Using Assertions (ECCA)* von Alkhalifa u. a. (1999): Im Hinblick auf einen Einsatz in verteilten Echtzeitsystemen wird sowohl der Speicherbedarf der Technik verringert als auch werden Instruktionen für Sprünge und Verzweigungen in die Absicherung integriert. ECCA weist zunächst den einzelnen Grundblöcken eindeutige Zahlenwerte zu. Während der Ausführung wird eine Integer-Variable mitgeführt, deren Wert beim Durchlaufen der Grundblöcke verändert wird. Durch schrittweises Testen und Erhöhen dieser Variable kann somit sichergestellt werden, dass die Applikation in einer erlaubten Form abgearbeitet wird und der Kontrollfluss korrekt ist. Experimente von Venkatasubramanian u. a. (2003) zeigen allerdings, dass ECCA zu enorm hohem Aufwand führt: Der Speicherbedarf sei demnach um mehr als 150 % erhöht. Goloubeva u. a. (2003) entwickelten die Idee von ECCA unter dem Titel *Yet Another Control-Flow Checking using Assertions (YACCA)* weiter, besonders im Hinblick auf eine noch vollständigere Fehlerabdeckung bei gleichzeitiger Reduktion von Speicherplatz und zusätzlicher Ausführungszeit. Am Anfang und Ende jedes Grundblocks werden gemäß eines im Vorfeld erstellten Kontrollflussgraphen zusätzliche Instruktionen eingefügt, welche die korrekte Abfolge des Kontrollflusses überprüfen.

Eine andere softwarebasierte Technik wurde von Miremadi u. a. (1992) als *Block Signature Self-Checking (BSSC)* entwickelt. Hier wird ein Programm in Grundblöcke aufgeteilt, um jedem einzelnen eine Signatur zuzuweisen. Zur Laufzeit wird diese Signatur am Ende jedes Blocks innerhalb einer Subroutine überprüft, ohne dass zusätzliche Hardware nötig ist. Nachteil dieser Technik ist, dass der Programmcode vom Speicherort abhängt, weil die Signaturen aus einer absoluten Speicheradresse bestehen. *Control-Flow Checking by Software Signatures (CFCSS)* von Oh u. a. (2002b) basiert auf einer ähnlichen Vorgehensweise: Auch hier werden den einzelnen Grundblöcken eindeutige statische Signaturen sowie die jeweilige *XOR*-Differenz zum folgenden Block zugewiesen. Während der Ausführung wird eine globale Variable mit der Signatur des ersten Blocks initialisiert. Sobald der Kontrollfluss in einen anderen Grundblock läuft, wird die Signatur dieses nachfolgenden Blocks aus der Signatur des Vorgängers sowie der gespeicherten Differenz berechnet und mit der statisch gespeicherten verglichen. Die Evaluierungen bei Oh u. a. (2002b) zeigen, dass CFCSS etwa 97 % der auftretenden transienten Fehler erfolgreich erkennen kann, wobei der zusätzlich geforderte Speicherbedarf zwischen 30 % und 60 % liegt. Nach Bernardi u. a. (2006) hat CFCSS allerdings folgende Schwäche: Falls mehrere Blöcke wiederum mehrere gleiche Nachfolger im Kontrollfluss haben, so kann mit Hilfe von CFCSS ein Fehler nicht erkannt werden.

Control-flow Error Detection through Assertions (CEDA) von Vemu und Abraham

(2006) verfolgt eine ähnliche Idee von Signaturen mit der Zielsetzung, den Aufwand auf ein Minimum zu reduzieren. So wird bei dieser Technik nur eine Instruktion pro Grundblock nötig, um die Signatur zur Laufzeit anzupassen. Ein Check, ob die Signatur mit dem statisch gespeicherten Erwartungswert übereinstimmt, findet nur noch vereinzelt explizit statt. Im Rahmen einer ausführlichen Evaluierung mit Erzeugung künstlicher Fehler wird bei Vemu und Abraham (2006) gezeigt, dass die Effizienz, d.h. das Verhältnis von erkannten Fehlern und Zusatzaufwand, bei CEDA deutlich besser ist als bei vergleichbaren Ansätze wie CFCSS (Oh u. a. 2002b) oder YACCA (Goloubeva u. a. 2003).

Eine präemptive Prüfmethodik speziell für den Kontrollfluss von Controllern in drahtlosen Telefonnetzen beschreiben Bagchi u. a. (2001) als *PreEmptive COntrol Signature (PECOS)*. Vorteil dieser präemptiven Technik ist, dass ein fehlerhafter Sprung bereits vor der Ausführung erkannt wird und somit rechtzeitig verhindert werden kann. Dazu werden spezielle Operationen zur Absicherung in den Assemblercode eines Programms eingefügt. Auf der anderen Seite verursacht die Technik allerdings enormen Mehraufwand, sowohl was den zusätzlichen Speicherbedarf betrifft, als auch hinsichtlich der Ausführungszeit der Applikationen.

Venkatasubramanian u. a. (2003) folgen einer alternativen Idee: Jedem Grundblock wird ein Paritätsbit zugewiesen, welches bei erfolgreicher Ausführung des Blocks gesetzt wird. Am Ende einer Applikation kann somit geprüft werden, ob all diese Bits entsprechend einer korrekten Ausführung des Programms den richtigen Wert haben. Bei Schleifen und bestimmten Programmteilen, die mehrfach durchlaufen werden, müssen nach jeder Iteration eine Prüfung und ein Rücksetzen des bzw. der entsprechenden Bits stattfinden. Vorteil der Methode ist v.a. der vergleichsweise geringe Mehraufwand hinsichtlich Speicher (nach deren Angaben ca. 30 %) und Laufzeit (ca. 47 %). Nachteilig ist, dass ein eventueller Fehler erst spät bei der Prüfung der Paritätsbits erkannt wird sowie dass die Abdeckung der Fehlerfälle nicht so hoch wie bei vergleichbaren Mechanismen ist.

Als Kombination und Erweiterung verschiedener Software-Techniken kann man den Ansatz *Software Implemented Fault Tolerance (SWIFT)* von Reis u. a. (2005) betrachten. Zum einen kommt hier der oben beschriebene EDDI-Mechanismus von Oh u. a. (2002c) zum Einsatz, bei welchem Instruktionen durch den Compiler dupliziert werden. Zum anderen wird eine erweiterte Form der CFCSS-Technik von Oh u. a. (2002b) (siehe oben) integriert – mit dem Unterschied, dass eine Prüfung nicht mehr am Ende sondern zu Beginn jedes Grundblocks erfolgt. Somit ist nicht nur sichergestellt, dass ein Sprung das richtige Sprungziel hat, sondern auch dass dieses korrekte Sprungziel wirklich erreicht wird. SWIFT verbindet diese Techniken außerdem mit gängigen ECC-Mechanismen (vgl. Peterson und Weldon 1972),

um Fehler im Speicher zu korrigieren. Somit entsteht eine optimierte Kombination, welche die Vorteile der einzelnen Fehlererkennungstechniken vereint.

Eine ähnliche Motivation zur bereits vorgestellten Hardware-Technik von Tsai u. a. (1990) verfolgt der softwarebasierte Ansatz von Chodrow u. a. (1991): Auch hier gilt es, das temporale Verhalten eines Echtzeitsystems zur Laufzeit zu überwachen, weniger im Hinblick auf Fehlertoleranz, sondern um die Korrektheit der Programmierung zu testen und zu verifizieren. Über eine C-Bibliothek wird dem Programmierer ein Toolkit angeboten, um im Programm Annotationen zum geforderten Zeitverhalten einzufügen. Diese Bedingungen können zur Laufzeit entweder *synchron*, d.h. zu einem bestimmten Zeitpunkt vom Echtzeit-Task selbst, oder *asynchron*, also von einem separaten Task, ausgewertet werden. Bei Mok und Liu (1997) wird diese Technik unter der Zielsetzung, ein Fehlverhalten möglichst frühzeitig festzustellen, erweitert. Den Prüfvorgang übernimmt dabei der sog. *Java Run-time Timing-constraint Monitor*, der entweder auf demselben Rechner parallel zum überwachten System oder alternativ auf einem eigenständigen Rechner läuft. Dieser wertet zur Laufzeit die eingebetteten Annotationen aus, um mögliche Abweichungen vom geforderten Zeitverhalten zu melden. Eine zusätzliche Erweiterung dieser Technik findet sich bei Mok u. a. (2002): Da es häufig schwierig ist, einen exakten Zeitpunkt eines bestimmten Systemereignisses anzugeben, können nun auch Zeitintervalle in den Annotationen vermerkt werden, welche vom Monitor entsprechend bei der Auswertung berücksichtigt werden.

Softwarebasierte Techniken zur Fehlererkennung bringen einige Vorteile mit sich, zumal keine Anpassung der Hardware nötig ist, um ein höheres Sicherheitslevel zu erreichen. Die Mechanismen arbeiten sehr effektiv bei der Erkennung von logischen Fehlern im Kontrollfluss, doch fehlt auch hier der Fokus auf Fehler im Zeitverhalten. Eine stark erhöhte Ausführungszeit der Applikationen beschränkt die Verwendung auf Einsatzgebiete, in welchen Performanz vernachlässigbar ist. Gerade in Echtzeitsystemen ist die verlängerte Ausführungszeit natürlich ein kritischer Punkt, da stets zu garantieren ist, dass alle Zeitschranken erreicht werden. Auch der Bedarf an zusätzlichem Speicher zum Einfügen duplizierter Informationen und zusätzlicher Instruktionen ist in eingebetteten Systemen von Nachteil.

Vergleicht man die Software-Techniken mit dem in dieser Arbeit vorgestellten Mechanismus zur Fehlererkennung, so kann letzterer durch geringen Aufwand hinsichtlich Ausführungszeit und Speicherverbrauch überzeugen (vgl. Kapitel 5). Ein weiterer positiver Aspekt ist, dass die Instrumentierung nicht unbedingt den Quellcode einer Applikation benötigt. Für die beschriebenen Software-Lösungen ist es hingegen essenziell, den Quellcode zum erneuten Kompilieren verfügbar zu haben, was einen Einsatz bei kommerziellen Applikationen erschwert.

3.4.3 Hybride Techniken zur Fehlererkennung

Hybride Techniken zur Erkennung von Kontrollflussfehlern kombinieren eine kostensparende SIHFT-Technik mit zusätzlich angeschlossener Hardware. Kritische Befehle der Applikation werden dupliziert und redundant ausgeführt; außerdem werden Instruktionen eingefügt, welche mit dem angebundenen Hardware-Checker kommunizieren, um Informationen zum aktuell genommenen Kontrollfluss zu übermitteln. Diese Prüfeinheit arbeitet parallel zum Hauptprozessor und stellt die Konsistenz der Programmbefehle sicher, um die Korrektheit des Kontrollflusses zu garantieren.

Der hybride Ansatz von Bernardi u. a. (2006) verbindet den Software-Ansatz YACCA von Goloubeva u. a. (2003) mit einer Technik von Cheynet u. a. (2000), welche durch Duplikation von Variablen die Datenintegrität absichert. Zusätzlich kommt ein sog. *Infrastructure-IP-Core (I-IP-Core)* als Hardware-Erweiterung zum Einsatz. Diese Art von IP-Core[7] wird in der Regel als Ein-/Ausgabegerät an den Bus eines Systems angeschlossen und kann nicht nur – wie in diesem Fall – der Erhöhung von Sicherheit dienen, sondern auch für Test-, Debugging- oder Diagnose-Zwecke eingesetzt werden (vgl. Zorian 2002). YACCA fügt – wie in Abschnitt 3.4.2 beschrieben – am Anfang und Ende jedes Grundblocks zusätzliche Instruktionen ein, um zur Laufzeit eine Signatur nach bestimmten Vorgaben zu modifizieren und jeweils zu prüfen, ob diese dem statisch gespeicherten Referenzwert entspricht. Die Idee des hybriden Ansatzes ist nun, die Vergleichsoperation in die angegliederte Hardware, also den I-IP-Core zu verlagern. Ebenso ist es Ziel, die von Cheynet u. a. (2000) entwickelte Technik zur Prüfung von Datenkonsistenz auf Hardware-Ebene zu übertragen. Dies führt, verglichen zu den ursprünglichen Software-Ansätzen, zu deutlich reduziertem Aufwand hinsichtlich Ausführungszeit und Speicherverbrauch. Gleichzeitig ist die Verwendung des I-IP-Cores mit wenig Integrationsaufwand verbunden, so dass eine Verwendung bestehender Prozessoren ohne größere Modifikationen ermöglicht wird. Nachteilig ist allerdings, dass die auszuführenden Applikationen stets im Quellcode vorliegen müssen, um die Signaturen überhaupt einfügen zu können.

Einen alternativen hybriden Erkennungsmechanismus entwickelten Ragel und Parameswaran (2011). Der Befehlssatz eines Prozessors wird dabei um jeweils eine Prüfinstruktion pro Kontrollstrukturtyp erweitert. Vor der Ausführung wird die Binärdatei einer Applikation instrumentiert, indem vor jeder auftretenden Kontrollstruktur zusätzlich eine entsprechende neu definierte Prüfinstruktion eingefügt wird. Während der Ausführung wertet die Hardware nun bei jedem Sprung und

[7] Als *IP-Core (Intellectual Property Core)* wird ein wiederbenutzbarer Teil eines Chipdesigns bezeichnet, welcher das geistige Eigentum des Entwicklers enthält.

jeder Verzweigung im Kontrollfluss beide Instruktionen aus und kann durch diese Redundanz die Korrektheit sicherstellen. Die Performanz dieser redundanten Prüfoperation ist hoch, da die Check-Mechanismen hardwareseitig als Routinen in Mikrocode implementiert sind. Der Aufwand dieser hybriden Technik wurde anhand von Simulationen mit künstlicher Fehlererzeugung evaluiert: Demzufolge ergibt sich ein geringer Mehrbedarf an Speicher von bis zu 13,5 % und eine um maximal 2,8 % erhöhte Ausführungszeit. Die zusätzlich integrierte Hardware wird mit 2,7 % im Verhältnis zum Original-Prozessor abgeschätzt. Ein weiterer Vorteil ist die Tatsache, dass eine Instrumentierung auf Basis der Binärdatei erfolgt. Somit können auch Bibliotheksfunktionen und Programme, zu welchen kein Quellcode verfügbar ist, abgesichert werden. Nachteilig ist, dass für die Integration des vorliegenden Mechanismus ein tiefer Eingriff in die Prozessor-Hardware nötig ist; der Fokus liegt daher eher auf einem Einsatz bei der Entwicklung neuer Prozessoren. Ebenfalls eine Schwäche ist die nicht vollständige Abdeckung von Fehlerfällen: Wird beispielsweise eine Instruktion fälschlicherweise, z.B. durch ein vertauschtes Bit im Opcode, zu einem Sprungbefehl, so ist der resultierende Fehler im Kontrollfluss durch die vorliegende Technik nicht erkennbar.

Im Gegensatz zu den beiden vorgestellten hybriden Ansätzen, welche nicht auf Fehler im Zeitverhalten fokussieren, betrachtet die Arbeit von Paolieri und Mariani (2011) speziell die zeitliche Korrektheit in Echtzeitsystemen. Mit Hilfe einer WCET-Analyse werden hierbei Informationen zum Laufzeitverhalten der Applikation gewonnen und im Rahmen einer Instrumentierung in den Quellcode eingefügt. Eine speziell entwickelte Hardware-Einheit, genannt *Timing-aware Coverage Monitor Unit (TaCMU)*, wird als IP-Core angeschlossen und prüft das Zeitverhalten während der Programmausführung. Die Granularität der Prüfung kann vom Entwickler selbst bestimmt werden: So ist es möglich, für jeden Grundblock Timing-Checks zu veranlassen oder nur für vollständige Funktionen. Der Ansatz zielt besonders auf einen Einsatz in einer echtzeitfähigen Multicore-Umgebung ab, in welcher mehrere Kontrollfäden parallel ausgeführt werden. Die im Fehlermodell der Arbeit abgebildeten „Timing Faults“ bezeichnen daher hauptsächlich Verzögerungen, welche durch die mehrfädige Nutzung gemeinsamer Ressourcen entstehen. Prinzipiell sollten derartige Verzögerungen aber bereits in einer korrekten WCET-Analyse berücksichtigt sein, so dass dadurch keine Fehlerursachen entstehen dürften. Das Verhalten bei transienten Fehlern wie SEUs wird im Rahmen der Arbeit leider nicht evaluiert. Von Nachteil erscheint bei der Technik außerdem, dass *ausschließlich* die zeitliche Korrektheit betrachtet wird, nicht aber die logische Korrektheit im Kontrollfluss.

Als Fazit für hybride Techniken lässt sich feststellen, dass diese eine sehr effektive Fehlererkennung ermöglichen. Ein hohes Maß an Zuverlässigkeit steht geringem Aufwand beim Einsatz gegenüber, sowohl hinsichtlich Speicherverbrauchs als auch

im Hinblick auf zusätzliche Ausführungszeit. Im Unterschied zu bestehenden Arbeiten verbindet der neu entwickelte und im Rahmen dieser Arbeit vorgestellte Mechanismus zur Erkennung von Kontrollflussfehlern in Echtzeitsystemen allerdings zwei bisher unterschiedliche Teilgebiete: die Erkennung von Fehlern im Zeitverhalten und die Erkennung logischer Kontrollflussfehler.

4

Implementierung des Erkennungsmechanismus

Um die Stärken und Schwächen des in Kapitel 3 vorgestellten Mechanismus zur Fehlererkennung besser bewerten zu können, ist eine möglichst realitätsnahe Implementierung nützlich. Dabei soll insbesondere das Zusammenspiel zwischen der Offline-Instrumentierung und der Prüfung während der Programmausführung näher betrachtet werden. Zum einen ist es also nötig, ein Werkzeug zur Code-Instrumentierung bereitzustellen, zum anderen muss die Prozessorsimulation um eine Check-Einheit erweitert werden, welche bei der Ausführung die instrumentierten Zusatzinformationen ausliest und im Fehlerfall entsprechende Reaktionen liefert.

Prinzipiell ist es möglich, das Instrumentierungswerkzeug so zu implementieren, dass es auf Basis der Binärdatei einer Applikation arbeitet. Weitaus weniger komplex ist allerdings eine Umsetzung, die Assembler-Dateien als Eingabeparameter einliest. Diese Variante bietet identische Funktionalität, setzt aber voraus, dass die zu instrumentierenden Applikationen im Quellcode vorliegen. Da sämtliche für eine Evaluierung vorgesehenen Benchmark-Programme auf diese Weise verfügbar sind, ist es naheliegend, im Rahmen dieser Arbeit die einfachere Variante zu realisieren.

Abbildung 4.1 zeigt die Integration der zu implementierenden Komponenten: Der Quellcode der Applikation wird zunächst zu Assembler-Code kompiliert. Im Anschluss sollen die erzeugten Assembler-Dateien vom Instrumentierungs-Tool eingelesen und mit Zusatzinformationen versehen werden, um daraufhin mit Assembler und Linker eine ausführbare Binärdatei zu erzeugen. Im zweiten Schritt wird diese

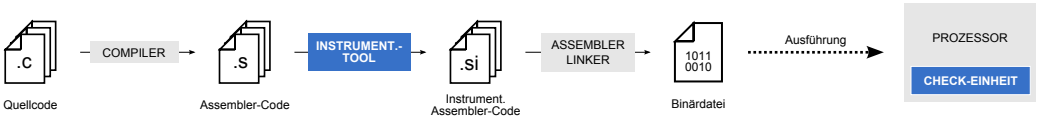


Abbildung 4.1: Integration von Instrumentierungs-Tool und Check-Einheit

Binärdatei in der Simulationsumgebung ausgeführt, wobei eine Check-Einheit zur Auswertung der Instrumentierung an den Prozessor angeschlossen ist.

Der folgende Abschnitt erläutert etwas näher die Ausführungsplattform; im Anschluss werden der darauf abgestimmte Instrumentierungsmechanismus sowie das implementierte Tool beschrieben. Schließlich folgt ein Einblick in die Funktionsweise der integrierten Check-Einheit und eine Abschätzung des benötigten Hardware-Aufwands. Die Implementierung soll eine Grundlage für umfangreiche Evaluierungen bilden, welche Inhalt des folgenden Kapitels 5 sein werden.

4.1 Ausführungsplattform

Als Basis für die Ausführung dient der echtzeitfähige CarCore-Prozessor (Uhrig u. a. 2005, Mische u. a. 2010), ein zweifach superskalarer SMT-Prozessor[1]. Der Befehlssatz des CarCore ist binärkompatibel zum Infineon TriCore, einem häufig eingesetzten Mikrocontroller in Echtzeitsystemen (vgl. TriCore 2008). Die Superskalarität resultiert aus zwei Prozessor-Pipelines (einer Adress- und einer Daten-Pipeline), welche jeweils aus den Stufen *Decode*, *Execute* und *Write Back* bestehen. Gemeinsam genutzt werden die Stufen *Instruction Fetch* und *Schedule* im vorderen Teil des Prozessors. Der Prozessor arbeitet *in-order*, d.h. Instruktionen durchlaufen in der Reihenfolge des Befehlsstroms die Pipelines. Eine parallele Ausführung zweier Instruktionen eines Threads ist möglich, falls eine Adressoperation direkt auf eine Datenoperation folgt. Ansonsten wird die ungenutzte Pipeline mit einer Instruktion aus einem ggf. vorhandenen weiteren Thread befüllt. Die im vorliegenden Zusammenhang verwendeten Anwendungen sind allerdings einfädig, so dass die Fähigkeiten zum Multithreading im Folgenden vernachlässigbar sind.

Simuliert wird auf einem SystemC-Modell des CarCore, welches das Timing-Verhalten taktgenau nachbildet. Dies ermöglicht es, realitätsnahe Laufzeiten verschie-

[1] *Superskalarität* beschreibt die Eigenschaft eines Prozessors, mehrere Instruktionen gleichzeitig verarbeiten zu können (Brinkschulte und Ungerer 2010, S. 296 ff.). *Simultane Mehrfädigkeit* (Simultaneous Multithreading, SMT) verbindet Superskalarität und Mehrfädigkeit: In einem Takt können mehrere Instruktionen unterschiedlicher Programmfäden gleichzeitig ausgeführt werden (Brinkschulte und Ungerer 2010, S. 406).

dener Applikationen zu messen und zu vergleichen. Aufgrund der Binärkompatibilität zwischen CarCore und TriCore wird im Folgenden der *HighTec*[2] *GNU C/C++-Compiler* in Version 3.3 zum Kompilieren von Anwendungen verwendet. Dabei ist eine kompilerseitige Optimierung in Stufe O2 gewählt, was gängige Code-Optimierungen, aber kein Funktions-Inlining und Loop-Unrolling beinhaltet.

4.2 Umsetzung der Checkpoints

Um die Funktionalität des in Kapitel 3 beschriebenen Mechanismus zu implementieren, sind bei einem Checkpoint folgende Informationen nötig:

- **Typ**: Gemäß Abschnitt 3.2.2 muss die Check-Einheit zwischen vier verschiedenen Checkpoint-Typen unterscheiden: Entweder ein Grundblock hat genau einen Nachfolger, es gibt zwei mögliche Nachfolger, es folgt ein Funktionsaufruf oder es folgt ein Rücksprung (Return) aus einer Funktionsdefinition. Da es sich um genau vier mögliche Typen handelt, wird für diese Information eine Breite von 2 Bit benötigt.
- **ID**: Jeder Grundblock besitzt eine ID, welche innerhalb der gesamten Applikation nur ein einziges Mal auftritt. Je nach der Anzahl der Grundblöcke im Programm muss die Bitbreite für die ID festgelegt werden, sodass alle Blöcke eindeutig identifizierbar sind.
- **Nachfolge-ID 1**: Um die logische Abfolge der IDs im Programmfluss anzukündigen, ist es nötig, die ID des Nachfolgers im Checkpoint anzugeben. Im Fall eines Funktionsaufrufes steht hier die ID des Grundblocks, in welchem die Funktion definiert ist. Die Bitbreite muss ebenso groß wie für die Angabe der ID gewählt werden.
- **Nachfolge-ID 2**: Bei bedingten Sprüngen ist die Ankündigung zweier potenzieller Nachfolger nötig. Ebenso ist es nötig, wie in Abschnitt 3.2.1 beschrieben, bei Funktionsaufrufen die ID zum Rücksprung anzugeben. Auch hier ist dieselbe Bitbreite wie für die Angabe der ID zu wählen.
- **WCET**: Zur Ermöglichung einer zeitlich korrekten Ausführung wird die vorher ermittelte WCET-Abschätzung des folgenden Grundblocks im Checkpoint gespeichert. Die Bitbreite muss so festgelegt werden, dass die höchste vorkommende WCET-Grenze im Programm darstellbar bleibt. Hier ist eine Skalierung möglich, was zwar mit einer gewissen Ungenauigkeit bei der Prüfung verbunden ist, allerdings in gewissem Rahmen akzeptabel wäre.

[2]HighTec EDV-Systeme GmbH, http://www.hightec-rt.com/

Typ	ID	Nachfolge-ID 1	Nachfolge-ID 2	WCET
2 Bit	x Bit	x Bit	x Bit	$(30 - 3 * x)$ Bit

Abbildung 4.2: Bitmaske mit expliziter Angabe zweier Nachfolge-IDs

4.2.1 Abbildung eines Checkpoint auf einen 32-Bit-Wert

Voraussetzung für die Instrumentierung ist zunächst, ein spezifisches Format für Checkpoints zu definieren. Im Hinblick auf eine Minimierung des Aufwands, empfiehlt es sich, das Format so zu wählen, dass dieses eine Länge von 32 Bit nicht überschreitet. Nur dann ist es möglich, den Checkpoint in ein 32-Bit-Register des CarCore zu schreiben, welches ja von der angebundenen Check-Einheit auszulesen ist. Würden für einen Checkpoint mehr als 32 Bit benötigt, müssten mehrere Register genutzt werden, was zu einer Vervielfachung des Aufwands führt.

Die Bitmaske in Tabelle 4.2 zeigt eine mögliche Darstellungsform. Für die Angabe des Grundblock-Typs sind 2 Bit nötig, für die Kodierung der ID sowie der beiden Nachfolger muss dieselbe Länge von x Bit reserviert sein. Die Darstellung der WCET-Abschätzung ergibt sich aus den verbleibenden Bit.

Problematisch ist hier die Begrenzung auf 32 Bit: Der Wert von x darf höchstens 9 sein, um die Gesamtlänge von 32 Bit nicht zu überschreiten. In diesem Fall verbleiben nur noch 3 Bit zur Darstellung der WCET-Abschätzung. Um eine aussagekräftigere Instrumentierung der WCET-Informationen zu erreichen, sollten für die IDs maximal 8 Bit verwendet werden. Dies erlaubt allerdings nur 256 instrumentierte Grundblöcke in der Applikation und macht somit nur die Instrumentierung sehr kleiner Anwendungen möglich. Zusammenfassend lässt sich feststellen, dass diese Menge an Informationen schlecht innerhalb von 32 Bit darstellbar ist, sodass dieser Ansatz für einen Praxiseinsatz relativ wertlos ist.

Abhilfe schafft folgende Einschränkung bei der Vergabe der Grundblock-IDs: Aufeinander folgende Blöcke seien per Definition mit inkrementell aufsteigenden IDs zu instrumentieren. Diese Tatsache erlaubt es, bei bedingen Sprüngen und Funktionsaufrufen nur noch eine statt bisher zwei Nachfolge-IDs explizit anzugeben. Denn auf jeden Grundblock, der mit einem bedingten Sprung endet, folgt am Ende entweder die ID des (explizit als Nachfolger angegebenen) Sprungziels oder des unmittelbar nachfolgenden Blocks, welcher die inkrementierte eigene ID besitzt. Es genügt also, den zweiten Nachfolger implizit anzugeben. Auch bei Funktionsaufrufen, bei denen als zweite Nachfolge-ID der nach dem Rücksprung ausgeführte Grundblock übergeben wird, ist eine explizite Angabe zweier Nachfolge-IDs unnötig. Der Rücksprung erfolgt an die Adresse, von welcher aus der Funktionsaufruf stattfand, d.h. die ID für den Rücksprung ist ebenfalls die inkrementierte eigene

Typ	ID	Nachfolge-ID	WCET
2 Bit	x Bit	x Bit	$(30 - 2 * x)$ Bit

Abbildung 4.3: Bitmaske mit expliziter Angabe einer Nachfolge-ID

ID. Im Zusammenhang mit der Erläuterung der Check-Einheit wird in Abschnitt 4.4 der Kontext zu expliziten und impliziten Nachfolgern nochmals aufgegriffen und detaillierter erklärt.

Dadurch dass es nicht mehr nötig ist, eine zweite Nachfolge-ID anzugeben, verändert sich die Bitmaske der Checkpoints. Tabelle 4.3 zeigt, dass bei einer Begrenzung auf 32 Bit nun bis zu $x = 14$ Bit für die IDs zur Verfügung stehen – abhängig davon, wie genau der WCET-Wert angegeben werden soll. Für eine erste Implementierung wird für x nun der Wert 10 gewählt, sodass 1024 Grundblöcke instrumentiert werden können, die eine maximale WCET-Abschätzung von 1024 Prozessortakten haben dürfen. Zur Evaluierung anhand verschiedener Benchmarks ist diese Variante gut einsetzbar.

4.2.2 Integration in die Programmausführung

Um die Auswertung der im Checkpoint gespeicherten Informationen während der Ausführung möglich zu machen, müssen die Checkpoints in einer bestimmten Form von der angeschlossenen Check-Einheit erkannt und gelesen werden. Es ist nötig, bestimmte Befehle einzufügen, welche beim Ausführen auf dem Prozessor den Check-Mechanismus auslösen. Ziel ist hierbei, möglichst wenige Prozessorinstruktionen für einen Checkpoint zu verwenden, um die zusätzliche Ausführungszeit zu minimieren. Andererseits soll eine Integration möglichst ohne große Eingriffe in die Prozessorarchitektur geschehen. Es ist nicht die Absicht, den Fokus auf eine möglichst optimale CarCore-spezifische Lösung zu legen, sondern eine gute Implementierung zu finden, die einfach auch auf andere Prozessoren übertragbar ist.

Diskussion prinzipieller Möglichkeiten

Eine Integration des Instrumentierungs- und Check-Mechanismus ist in folgenden Varianten denkbar:

- **Definition einer eigenen Instruktion**: Möglich ist, einen nicht benutzten Opcode zu wählen und eine eigene Prozessorinstruktion zu definieren. Im CarCore stehen bei einem solchen Befehl bis zu 24 freie Bit zur Verfügung,

um den Checkpoint-Wert zu übergeben. Für einen 32-Bit-Wert sind damit nur zwei Instruktionen nötig, was zwar einen geringen Mehraufwand bzgl. der Laufzeit zur Folge hat, allerdings einen aufwendigen Eingriff in die Hardware erfordert. Zum einen muss erst ein freier Opcode gefunden werden, zum anderen muss die gesamte Pipeline entsprechend angepasst werden.

- **Verwendung eines Coprozessor-Befehls**: Im CarCore gibt es – wie in den meisten Prozessoren – Coprozessor-Befehle, welche bereits für die Verwendung von Erweiterungen reserviert sind. Auch hier ist eine Integration nicht einfach, zudem sind die Befehle so eingeschränkt, dass weniger als 16 Bit zur Verfügung stehen. Für das Übertragen eines 32-Bit-Wertes sind damit mehr als zwei Instruktionen nötig, was wiederum hohen Aufwand zur Folge hat.

- **Schreiben an eine festgelegte Speicherstelle**: Relativ einfach zu integrieren ist eine Check-Einheit, welche eine bestimmte Adresse im Speicher überwacht. Sobald ein Schreibzugriff vorliegt, wird eine Prüfung angestoßen. Für einen einzelnen Checkpoint sind drei Instruktionen nötig: zwei zum Schreiben des 32-Bit-Wertes in ein Adressregister und ein Befehl zum Schreiben in den Speicher. Durch die Speicherlatenz kostet die Ausführung allerdings mehrere zusätzliche Prozessortakte, sodass der Aufwand bei dieser Variante recht hoch wird.

- **Überwachen eines bestimmten Adress- oder Datenregisters**: Denkbar ist auch, ein Register direkt zu überwachen. Zwei Adressregister werden bereits gemäß CarCore- bzw. TriCore-Spezifikation nicht vom Compiler verwendet. Daher wäre es möglich, die Check-Einheit so zu integrieren, dass diese bei jedem Zugriff auf ein bestimmtes dieser Register aktiv wird. Der Aufwand bleibt gering, da nur zwei Instruktionen zum Schreiben der 32 Bit nötig sind. Allerdings ist auch bei dieser Variante ein tieferer Eingriff in die Hardware nötig, da das Signal zum Registerzugriff erst von der Pipeline nach außen gelegt werden muss.

- **Verwendung eines Spezialregisters**: Neben 16 Adress- und 16 Datenregistern verfügt der CarCore – wie der Infineon TriCore und in ähnlicher Weise auch die meisten anderen Prozessoren – über spezielle *Core Special Function Registers* (*CSFRs*), auf welche über die Instruktionen `MTCR` (move to core register) bzw. `MFCR` (move from core register) schreibend bzw. lesend zugegriffen wird. Die Check-Einheit kann einfach mit dem Prozessor verbunden werden, sodass diese aktiv wird, sobald die Instruktion `MTCR` auf dem spezifischen Checkpoint-Register ausgeführt wird. Pro Check sind drei Instruktionen nötig: zwei zum Schreiben des Checkpoint-Wertes in ein

	Aufwand pro Check	HW-Integration
Eigene Instruktion	ca. 2 Takte	komplex
Coprozessor-Befehl	> 2 Takte	komplex
Speicherstelle	> 5 Takte	einfach
Adr. / Daten-Register	ca. 2 Takte	komplex
Spezialregister	< 5 Takte	einfach

Tabelle 4.1: Bewertung unterschiedlicher Integrationsvarianten im CarCore hinsichtlich Aufwand pro Check (in Prozessortakten) und Komplexität der Hardware-Integration

Datenregister und ein `MTCR`-Befehl zum Kopieren des Wertes in das Spezialregister. Dadurch, dass `MTCR` ein Adressbefehl ist, der hier unmittelbar auf einen Datenbefehl folgt, können die beiden Befehle im CarCore parallel ausgeführt werden. Dies wirkt sich bei dieser Variante positiv auf den entstehenden Mehraufwand hinsichtlich Ausführungszeit aus.

Tabelle 4.1 fasst die fünf vorgestellten Varianten nochmals zusammen. Die Definition einer eigenen Instruktion, die Verwendung eines Coprozessor-Befehls sowie das Überwachen eines Adress- oder Datenregisters erfordern einen tiefen Eingriff in die Hardware. Gerade im Hinblick auf eine Lösung, die relativ einfach auch auf andere Prozessoren übertragbar ist, erscheint eine Realisierung dieser Varianten im CarCore als wenig sinnvoll. Unter den verbleibenden Möglichkeiten gilt es nun, diejenige zu wählen, welche mit dem wenigsten Aufwand verbunden ist. Es bietet sich also die Verwendung eines Spezialregisters an, da diese Variante in der Regel weniger Aufwand als das Schreiben an eine Speicherstelle verursacht.

Der folgende Abschnitt beschreibt die Verwendung eines Spezialregisters näher. Dabei kommen wiederum zwei Implementierungsvarianten in Betracht, welche sich im Hinblick auf den Aufwand unterscheiden.

Implementierungsvarianten

Der Zugriff auf ein Spezialregister mittels `MTCR` benötigt den zu schreibenden 32-Bit-Wert in einem Datenregister. Da der Instrumentierungsvorgang nach dem Kompilieren auf Basis des Assemblercodes ausgeführt wird, muss sichergestellt sein, dass keine Werte von Datenregistern überschrieben werden, die für den folgenden Programmablauf wichtig sind. Andernfalls würde dies fatale Fehler als Konsequenz haben. Um ein Überschreiben zu vermeiden, ergeben sich folgende Strategien bei der Verwendung der Spezialregister:

- **Variante 1 – Sicherung des Datenregisterinhalts**: Die Adressregister A[8] und A[9] werden gemäß CarCore- bzw. TriCore-Spezifikation nicht vom Compiler verwendet. Daher besteht die Möglichkeit, die Inhalte eines Datenregisters in eines dieser Adressregister zu sichern und nach der Check-Operation wieder zurückzuschreiben. Ein einzelner Checkpoint besteht auf Assemblerebene dann aus folgenden fünf Instruktionen: `MOV.A` zum Sichern des Datenregisterinhalts, `MOVH` zum Schreiben der oberen 16 Bit, `ADDI` zum Schreiben der unteren 16 Bit, `MTCR` zum Aktivieren des Checkers und `MOV.D` zum Zurückschreiben des ursprünglichen Datenregisterinhaltes.

- **Variante 2 – Freihalten eines Datenregisters beim Kompilieren**: Durch eine spezielle Anweisung beim Kompilieren kann ein bestimmtes Datenregister freigehalten werden. Somit entfällt die Notwendigkeit die Registerinhalte vor dem Beschreiben zu Sichern. Ein Checkpoint besteht dann aus drei Instruktionen: `MOVH` zum Schreiben der oberen 16 Bit, `ADDI` zum Schreiben der unteren 16 Bit und `MTCR` zum Aktivieren des Checkers.

Es erscheint offensichtlich, dass eine Instrumentierung nach erster Variante einen höheren Aufwand bei der Programmausführung verursacht. Im Gegensatz dazu kann bei der zweiten Variante die Tatsache, dass der Compiler ein Datenregister freihalten muss, im Hinblick auf die Laufzeit völlig vernachlässigt werden. Verschiedene Tests, bei welchen Benchmarks ohne Instrumentierung sowohl mit als auch ohne freigehaltene Datenregister kompiliert und ausgeführt wurden, zeigen keinerlei Laufzeitunterschied. Zwar ist es schwierig diese Beobachtung zu verallgemeinern, doch auf Basis der durchgeführten Messungen kann die zweite Instrumentierungsvariante bedingungslos überzeugen. Im weiteren Verlauf der Arbeit wird demnach die erste Variante nicht weiter betrachtet. Wird von Instrumentierung gesprochen, so ist im Folgenden stets eine Instrumentierung nach zweiter Variante gemeint.

4.3 Tool zur Instrumentierung von Applikationen

Wie bereits in Abbildung 4.1 gezeigt, erfolgt die Instrumentierung auf Basis des Assembler-Codes. Dies hat gegenüber einer Instrumentierung im Quellcode den Vorteil, dass der Beginn und das Ende von Grundblöcken einfach identifizierbar sind. Außerdem ist die Funktionsweise der Instrumentierung dadurch unabhängig von Optimierungsoptionen beim Kompilieren. Vorteile gegenüber einer Instrumentierung der gelinkten Binärdatei sind ebenfalls naheliegend: Das Einfügen von

Checkpoints (mit anderen Worten zusätzlicher Instruktionen) würde alle nachfolgenden Adressen verschieben. Daher müssten sämtliche Verweise und Sprungziele angepasst werden, was zwar funktionieren würde, aber sehr komplex ist.

Bei der Instrumentierung werden, wie bereits erläutert, Informationen zur WCET-Obergrenze sowie zu nachfolgenden Grundblock-IDs eingefügt. Besteht eine Anwendung aus mehreren Quellcode-Dateien, müssen für die Instrumentierung der logischen Programmabfolge die IDs sämtlicher Dateien bekannt sein. Da Funktionsaufrufe möglicherweise zu einem Sprung in andere Assemblerdateien führen, muss in diesem Fall als Nachfolger die ID des Grundblocks der zugehörigen Funktionsdefinition zugeordnet werden können. Daher wird der Instrumentierungsvorgang einmalig aufgerufen und beinhaltet als Parameter sämtliche zu instrumentierende Dateien, um die IDs über alle Dateien hinweg global zu vergeben.

Zur Instrumentierung des Programmcodes kommt das im Folgenden beschriebene Tool zum Einsatz. Dieses umfasst sowohl eine Komponente zur Analyse des logischen Ablaufs als auch ein einfach gehaltenes Werkzeug zur Abschätzung der WCET. Als Erweiterung ist denkbar, ein gängiges statisches WCET-Tool zu verwenden, welches neben den einzelnen Instruktionen auch Instruktionsabläufe und die detaillierte Prozessorarchitektur in die Berechnung mit einbezieht. Bedingung ist nur, dass dieses externe Tool auf Basis von Grundblöcken arbeitet und die jeweiligen WCET-Obergrenzen der Grundblöcke als Ausgabe vorliegen.

Abbildung 4.4 verdeutlicht den Ablauf des implementierten Tools: Als Eingabe wird der bereits kompilierte Assembler-Code der Applikation erwartet; dieser kann entweder aus einer einzelnen oder aus mehreren Dateien bestehen. Außerdem ist eine tabellarische Übersicht der WCET-Obergrenzen sämtlicher Prozessorinstruktionen nötig. In der aktuellen Implementierung liegt diese Information als einfache Textdatei vor. Durch den Aufruf des Tools werden die Komponenten *Basic Block Parser*, *Logical Analyzer*, *WCET Calculator*, *Checkpoint Generator* und *File Writer* durchlaufen. Im Folgenden soll jeder Einzelschritt näher erläutert werden.

4.3.1 Basic Block Parser

Jede Assemblerdatei wird zunächst von einem Parser durchlaufen. Dabei findet eine Zerlegung in maximale Grundblöcke (vgl. Abschnitt 2.1) statt, welche intern tabellarisch gespeichert werden. In diesem Zusammenhang werden den Grundblöcken inkrementell aufsteigende IDs vergeben, welche ebenfalls in der Tabelle vermerkt werden. Zwar wird für jede Eingabedatei eine eigene Tabelle angelegt, doch die IDs werden bereits global eindeutig zugewiesen. So kann ein globales Funktionsverzeichnis, sprich eine Zuordnung aller vorkommenden Funktionen zu

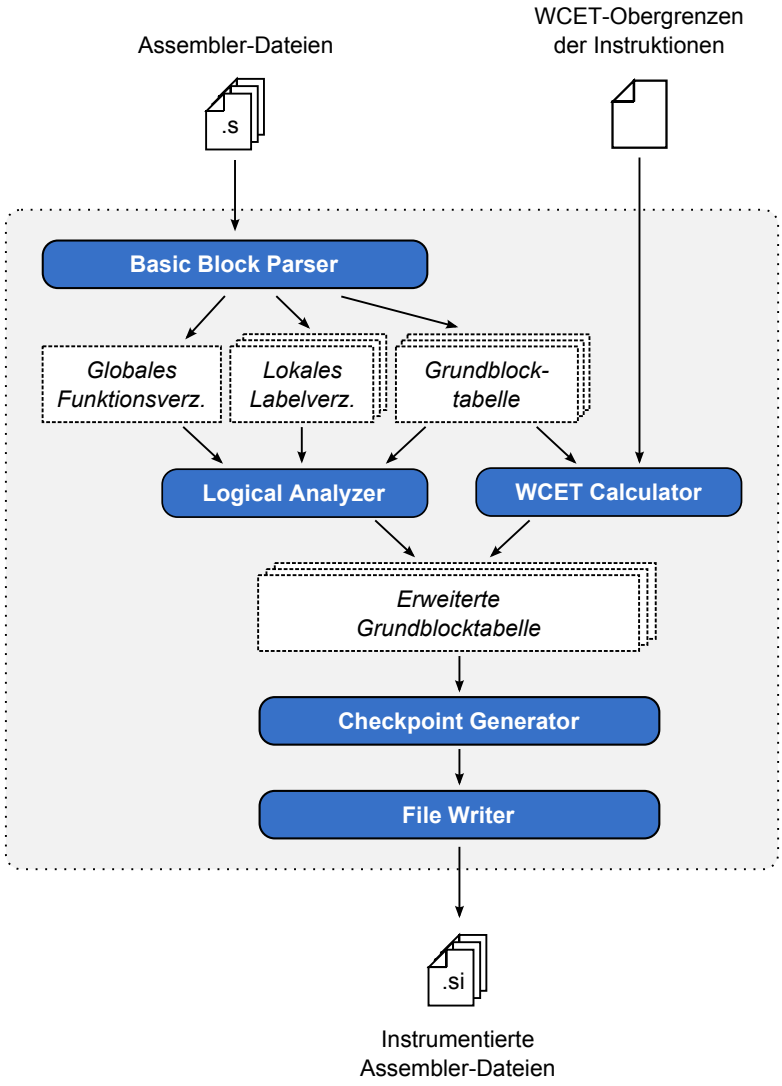


Abbildung 4.4: Ablauf der Instrumentierung

den jeweiligen IDs, angelegt werden, was für den nächsten Arbeitsschritt nötig wird. Außerdem werden alle genutzten Labels sowie statische (d.h. nur lokal sichtbare) Funktionen mit den jeweiligen IDs in einem für jede Assembler-Datei lokalen Labelverzeichnis abgelegt.

4.3.2 Logical Analyzer

Auf Basis der Grundblocktabelle und unter Zuhilfenahme der erzeugten Verzeichnisse wird der logische Ablauf der Grundblöcke analysiert. Dabei gibt insbesondere die jeweils letzte Instruktion eines Grundblocks Aufschluss über den weiteren Programmablauf. Entsprechend werden die Blöcke klassifiziert: Jedem Block wird ein Typ zugewiesen, der anzeigt, ob es sich um eine Sequenz, einen bedingten oder unbedingten Sprung, einen Funktionsaufruf oder einen Rücksprung handelt. Bei einem Sprung wird das entsprechende Ziel mit Hilfe des Labelverzeichnisses in eine ID übersetzt und als Nachfolger in der Tabelle vermerkt. Im Fall eines Funktionsaufrufes gilt es zunächst zu überprüfen, ob es sich um eine statische Funktion handelt. Ist dies der Fall, so befindet sich im lokalen Labelverzeichnis ein entsprechender Eintrag. Falls nein, wird die ID der Funktionsdefinition über das globale Funktionsverzeichnis ermittelt, um diese schließlich in der Grundblocktabelle zu vermerken. Als Ergebnis erhält man eine erweiterte Grundblocktabelle, welche alle notwendigen Informationen über die Nachfolger der jeweiligen Blöcke enthält.

4.3.3 WCET Calculator

Als ein einfach konzipiertes WCET-Tool berechnet der WCET Calculator eine Obergrenze für die maximale Ausführungszeit der einzelnen Grundblöcke. Dazu werden aus einer externen Tabelle die WCET-Grenzen der einzelnen Prozessorinstruktionen ausgelesen. Im Anschluss durchläuft der WCET Calculator auf Basis der vom Parser erzeugten Grundblocktabelle jeden einzelnen Block und summiert die WCET-Werte der enthaltenen Instruktionen. Als Ergebnis erhält man eine WCET-Obergrenze die wohl weit höher geschätzt ist, als das Ergebnis gängiger WCET-Tools. Allerdings kann garantiert werden, dass die reale Ausführungszeit die berechnete WCET-Grenze niemals überschreiten wird. Um die „Tightness“ des Ergebnisses, d.h. den Abstand zwischen maximaler auftretender Ausführungszeit und berechneter WCET, zu verbessern, wäre eine Erweiterung mit Einsatz eines externen statischen WCET-Tools nützlich. Im vorliegenden Zusammenhang ist dieser Aspekt jedoch weniger im Fokus, daher wird im Rahmen der Arbeit auf diese Erweiterung verzichtet.

4.3.4 Checkpoint Generator

Basierend auf der erweiterten Grundblocktabelle müssen nun die einzelnen Werte der Checkpoints generiert werden. Die Aufteilung der verfügbaren 32 Bit erfolgt dabei gemäß der Bitmaske in Tabelle 4.3. Typ 1 beschreibt eine sequentielle Abfolge oder einen unbedingten Sprung; neben der eigenen ID wird die inkrementierte ID bzw. das Sprungziel als Nachfolger übergeben. Typ 2 steht für einen bedingten Sprung mit zwei möglichen Nachfolgern. Dabei wird das Sprungziel als expliziter Nachfolger übergeben. Die inkrementierte eigene ID muss nicht zusätzlich angegeben werden, da dies anhand des Typs für die Check-Einheit ersichtlich wird. Einen Funktionsaufruf beschreibt Typ 3: Als Nachfolger steht der Grundblock der Funktionsdefinition. Als Ziel für den Rücksprung wird die Check-Einheit später die inkrementierte eigene ID verwenden, so dass diese ebenfalls nicht explizit anzugeben ist. Schließlich steht Typ 4 für einen Rücksprung, wobei in diesem Fall keine weitere Information als die eigene ID nötig ist. Unabhängig vom Typ des Grundblocks wird die WCET-Abschätzung aus der erweiterten Grundblocktabelle verwendet und in die entsprechenden Bits des Checkpoint geschrieben. Eine Ausnahme bilden Aufrufe von Funktionen, welche nicht innerhalb des instrumentierten Codes definiert sind, sondern z.B. aus einer gelinkten Bibliothek stammen. Da in diesem Fall nicht instrumentierter Code ausgeführt wird, bis ein neuer Checkpoint erreicht wird, ist die Angabe einer WCET-Grenze zwischen den beiden Checkpoints nicht möglich. Falls es indirekte Sprünge oder indirekte Funktionsaufrufe im Code gibt, werden diese ausschließlich mit der ID des jeweiligen Grundblocks instrumentiert, die Felder zu Nachfolgern und WCET-Abschätzung bleiben leer.

4.3.5 File Writer

Als letzter Schritt werden die instrumentierten Assemblerdateien erzeugt. Der File Writer schreibt zu jeder Eingabedatei eine entsprechende instrumentierte Ausgabedatei. Vor jeden Grundblock werden Assembler-Anweisungen gestellt, welche den generierten Checkpoint-Wert gemäß Abschnitt 4.2 in ein Spezialregister schreiben und während der Ausführung bei der Check-Einheit eine Überprüfung veranlassen.

4.4 Hardware-Check-Einheit

Zur Überprüfung der instrumentierten Informationen während der Programmausführung wird in den SystemC-Simulator des CarCore-Prozessors eine spezielle Check-Einheit integriert. Diese Komponente ist direkt mit der Prozessor-

Pipeline verbunden und erhält sowohl das Taktsignal als auch die dekodierten `MTCR`-Instruktionen als Eingabe. So erfolgt in jedem Takt eine Prüfung, ob im Programm ein Checkpoint vorliegt, um mittels temporalem und logischem Check-Vorgang entsprechend zu reagieren. Programmunterbrechungen in Form von Interrupts (vgl. Abschnitt 3.2.2) werden im Rahmen der Implementierung nicht berücksichtigt.

4.4.1 Temporaler Check-Vorgang

Algorithmus 4.1 veranschaulicht die Implementierung des temporalen Check-Vorganges innerhalb der Check-Einheit. Im Wesentlichen basiert der Vorgang auf der internen Variable *Countdownwert*, welche zunächst mit dem Wert 0 initialisiert wird. Sobald während der Ausführung der Applikation ein Checkpoint erkannt wird, erfolgt eine Zuweisung des im Checkpoint angegebenen WCET-Wertes (siehe Zeile 7). In jedem Prozessortakt wird der *Countdownwert* um den Wert 1 dekrementiert (Z. 3) und gleichzeitig überprüft, ob dieser bereits im negativen Bereich liegt (Z. 4). Ist dies der Fall, wurde die angegebene WCET-Abschätzung überschritten und es muss ein Fehler gemeldet werden. Die Überprüfung in Zeile 2 sichert ab, dass der erste Checkpoint bereits erreicht wurde. Zuvor ist keine Fehlererkennung möglich, da der Initialwert dekrementiert würde und damit einen negativen Wert hätte. Dies würde fälschlicherweise einen Fehler melden.

Algorithmus 4.1 Arbeitsweise der temporalen Check-Einheit

```
 1: wiederhole
 2:   wenn Check-Mechanismus gestartet dann
 3:     dekrementiere Countdownwert
 4:     prüfe ob Countdownwert ≥ 0
 5:   wenn_ende
 6:   wenn Checkpoint erkannt dann
 7:     Countdownwert ← Checkpoint[WCET]
 8:     starte Check-Mechanismus
 9:   wenn_ende
10:   warte auf folgendes Taktsignal
11: wiederhole_ende
```

Als Fazit lässt sich feststellen, dass für den temporalen Check-Vorgang eine sehr einfache Hardware-Implementierung möglich ist. Etwas aufwendiger wird es für die Überprüfung der logischen Korrektheit, was im Folgenden näher erläutert ist.

4.4.2 Logischer Check-Vorgang

Bei der Implementierung des logischen Check-Vorganges, wie in Algorithmus 4.4.2 gezeigt, spielt die Unterscheidung nach dem Checkpoint-Typ eine zentrale Rolle. Dazu wird die Variable *Typ* jeweils am Ende eines Prüfvorganges auf den Wert des aktuell gelesenen Checkpoint-Typs gesetzt (Zeile 15), um zu wissen, welche Prüfung beim Erreichen des folgenden Checkpoints abzuarbeiten ist. Als Initialwert erhält die Variable *Typ* den Wert 0, sodass beim Erreichen des ersten Checkpoints keine Prüfung stattfindet. Neben dem Checkpoint-Typ werden beim Erreichen aller Checkpoints folgende Variablen gesetzt: Als expliziter Nachfolger *ExplNachfolger* wird der im Checkpoint angegebene Nachfolger ausgelesen (Z. 16), als impliziter Nachfolger *ImplNachfolger* wird die inkrementierte eigene ID (Z. 17) gespeichert.

Algorithmus 4.2 Arbeitsweise der logischen Check-Einheit

```
 1: wiederhole
 2:   wenn Checkpoint erkannt dann
 3:     wenn Typ = 1 dann # Sequenz oder unbedingter Sprung erwartet
 4:       prüfe ob Checkpoint[ID] = ExplNachfolger
 5:     sonst wenn Typ = 2 dann # bedingter Sprung erwartet
 6:       prüfe ob Checkpoint[ID] = ExplNachfolger ∨ ImplNachfolger
 7:     sonst wenn Typ = 3 dann # Funktionsaufruf erwartet
 8:       prüfe ob Checkpoint[ID] = ExplNachfolger
 9:       lege ImplNachfolger auf Stack
10:     sonst wenn Typ = 4 dann # Rücksprung nach Funktion erwartet
11:       hole Nachfolger vom Stack
12:       prüfe ob Checkpoint[ID] = Nachfolger
13:       lösche obersten Stack-Eintrag
14:     wenn_ende
15:     Typ ← Checkpoint[Typ]
16:     ExplNachfolger ← Checkpoint[Nachfolger]
17:     ImplNachfolger ← Checkpoint[ID] + 1
18:   wenn_ende
19:   warte auf folgendes Taktsignal
20: wiederhole_ende
```

Abhängig vom aktuellen Wert der Variable *Typ* wird beim Erreichen eines Checkpoints einer der folgenden Prüfvorgänge ausgeführt. Falls eine der Prüfungen zu einem negativen Ergebnis kommt, so muss von einem Fehler im Programmfluss ausgegangen werden.

- ***Typ* = 1**: Handelt es sich um eine Sequenz oder einen unbedingten Sprung, findet ein Vergleich statt, ob die ID des aktuell gelesenen Checkpoints dem zuletzt gespeicherten expliziten Nachfolger entspricht (Z. 4).
- ***Typ* = 2**: Im Fall eines bedingten Sprungs wird verglichen, ob die ID des aktuellen Checkpoints entweder mit dem expliziten oder impliziten Nachfolger identisch ist (Z. 6).
- ***Typ* = 3**: Bei einem Funktionsaufruf gilt es zunächst zu testen, ob der Checkpoint der Funktionsdefinition erreicht wurde. Die aktuelle ID muss also dem expliziten Nachfolger entsprechen (Z. 8). Um einen Testvorgang für den Rücksprung zu ermöglichen, wird im zweiten Schritt der implizite Nachfolger auf den internen Stack-Speicher gelegt (Z. 9).
- ***Typ* = 4**: Wird ein Rücksprung am Ende einer Funktionsdefinition erwartet, so muss als Nachfolgewert der oberste Eintrag des Stack-Speichers mit der aktuell ausgelesenen Checkpoint-ID auf Gleichheit überprüft werden (Z. 11f.). Im Anschluss wird der gelesene Wert vom Stack entfernt (Z. 13).

Zusammenfassend ist die Implementierung der logischen Check-Einheit zwar etwas umfassender als die der temporalen, die Vergleichsoperationen können allerdings sehr einfach und effizient ausgeführt werden. Um eine fundierte Abschätzung zu erhalten, soll im folgenden Abschnitt der Hardware-Aufwand durch die integrierte Check-Einheit möglichst genau analysiert werden.

4.4.3 Abschätzung des Hardware-Aufwands

Um den Hardware-Aufwand der Check-Einheit besser abschätzen zu können, wurde das implementierte SystemC-Modell in ein VHDL-Modell[3] umgewandelt, um damit eine logische Synthese für ein Altera Stratix II Field Programmable Gate Array (FPGA) durchführen zu können. Eine solche Synthese kann den Bedarf an sog. *Adaptive Look-Up Tables (ALUTs)* sowie *Dedicated Logic Registers* ermitteln. Als Absolutwerte sind diese Angaben zwar wenig hilfreich, doch in Relation zum Gesamtprozessor ergibt sich eine gute Abschätzung für den Hardware-Verbrauch (vgl. Tabelle 4.2).

Als kritischer Punkt bei der Implementierung erweist sich der Stack-Speicher, welcher für die Umsetzung der Funktionsaufrufe und Rücksprünge benötigt wird. Die Größe dieses Stacks richtet sich nach der Tiefe der Funktionsaufrufe (*Call*

[3] *Very High Speed Integrated Circuit Hardware Description Language*, kurz *VHDL*, ist eine Hardware-Beschreibungssprache, die es ermöglicht, digitale Systeme textbasiert, ähnlich zu einer Programmiersprache, zu beschreiben.

	Stack in logischen Registern			**Stack im On-Chip-Speicher**
Tiefe der Funktionsaufrufe	16	32	64	beliebig
ALUTs	239 (0.7 %)	338 (1.0 %)	600 (1.7 %)	163 (0.5 %)
Dedic. Logical Registers	221 (2.2 %)	382 (3.7 %)	703 (6.9 %)	102 (1.0 %)

Tabelle 4.2: Hardware-Verbrauch der Check-Einheit im Verhältnis zum CarCore-Prozessor, abhängig von der Stack-Implementierung

Depth) eines Programmes. Tabelle 4.2 zeigt die Resultate bei einer Implementierung des Stack-Speichers als interne logische Register: Nimmt man eine Funktionsaufruftiefe von 16, so benötigt die Check-Einheit 239 ALUTs (0.7 % im Vergleich zum vollständigen CarCore-Processor) und 221 Dedicated Logic Registers (2.2 %). Bei Vergrößerung des Stack-Speichers wachsen diese Werte fast linear. Bei einer Funktionsaufruftiefe von 64 erhält man somit 600 ALUTs (1.7 %) und 703 (6.9 %) Dedicated Logic Registers. Ein reduzierter Hardware-Aufwand lässt sich erzielen, indem der Stack innerhalb eines Onchip-Speichers definiert wird, der weniger kostbar ist als logische Register. In diesem Fall benötigt die Check-Einheit nur noch 163 ALUTs (0.5 %) und 102 (1.0 %) Dedicated Logic Registers, unabhängig von der Tiefe der Funktionsaufrufe. Allerdings kommt in diesem Fall natürlich der Speicherbedarf für den Stack hinzu: Bei einer 10-Bit-Darstellung der ID und Funktionsaufruftiefe f ergibt sich eine Speichergröße von $(f * 10)/8$ Byte.

Als Fazit lässt sich somit feststellen, dass die implementierte Check-Komponente durch ihren sehr geringen Hardware-Aufwand überzeugen kann. Die Umsetzung profitiert insbesondere von der geringen Komplexität der entworfenen Einheit sowie der Möglichkeit, diese einfach in die vorhandene Prozessorumgebung zu integrieren. Damit ist eine wichtige Anforderung, die bereits am Anfang von Kapitel 3 definiert wurde, erfüllt. Um weitere Faktoren wie die Abdeckung von Fehlerfällen, die Latenz der Fehlererkennung sowie eine Aufwandsanalyse hinsichtlich Speicherbedarf und Ausführungszeit der Applikationen messen zu können, ist eine Evaluierung anhand verschiedener Benchmark-Programme nötig. All dies soll Inhalt des folgenden Kapitels sein.

5
Evaluierung

Ziel dieses Kapitels ist es, den entworfenen Mechanismus zur Erkennung von Kontrollflussfehlern in Echtzeitsystemen auf Basis der vorgestellten Implementierung umfassend zu testen und anhand verschiedener Kriterien zu evaluieren (vgl. Wolf u. a. 2012b,c). So soll nicht nur die Wirksamkeit der Fehlererkennung sondern auch der damit verbundene Aufwand gemessen und bewertet werden. Zu diesem Zweck findet eine Simulation typischer Applikationen aus dem Automotive-Bereich statt, welche im Folgenden näher vorgestellt werden sollen. Dabei wird eine Vielzahl künstlicher Fehler erzeugt, um deren Auswirkungen zu protokollieren und die Resultate anschließend zu analysieren.

Die Aspekte, welche evaluiert werden sollen, richten sich nach den in Kapitel 3 definierten Anforderungen an den Erkennungsmechanismus:

- **Abdeckung erkannter Fehlerfälle**: Es ist interessant zu bestimmen, welche bzw. wie viele Fehlerfälle durch die eingesetzten Technik erfolgreich identifizierbar sind. Außerdem soll verglichen werden, welche Folgen die künstlich erzeugten Fehler ohne Erkennungstechnik hätten.

- **Latenz der Fehlererkennung**: Ziel ist es stets, Fehler möglichst frühzeitig zu erkennen. Im Rahmen der Evaluierung soll der Zeitraum zwischen Auftreten und Erkennung eines Fehlers wiederum am Beispiel künstlich erzeugter Fehlerfälle genauer untersucht werden.

- **Aufwandsanalyse**: Durch die Code-Instrumentierung werden zusätzliche Instruktionen in eine Applikation eingefügt, was zu längerer Ausführungszeit und zu mehr Bedarf an Instruktionsspeicher führt. Diese beiden Größen sollen anhand von Beispielprogrammen analysiert und evaluiert werden.

Der nachfolgende Abschnitt beginnt mit einer Beschreibung der Evaluierungsumgebung, der verwendeten Benchmark-Programme und des implementierten Fehlermodells. Anschließend werden die verschiedenen Aspekte der Evaluierung einzeln präsentiert.

5.1 Evaluierungsumgebung

Die Evaluierung erfolgt auf Basis des in Abschnitt 4.1 vorgestellten CarCore-Prozessors. Mit Hilfe eines SystemC-Modells wird das Verhalten des Prozessors taktgenau nachgebildet, um so auch Aussagen zum Timing-Verhalten der Ausführung zu ermöglichen. Das Prozessormodell beinhaltet die in Abschnitt 4.4 beschriebene Hardware-Check-Einheit zur Fehlererkennung. Zudem ist eine Erweiterung in den Simulator integriert, welche es erlaubt, auf systematische Weise künstliche Fehler zu injizieren.

Als Beispielapplikationen dienen verschiedene Benchmark-Programme, welche mit dem in Abschnitt 4.3 präsentierten Tool instrumentiert wurden. Diese Benchmarks sollen im Folgenden näher vorgestellt werden. Im Anschluss folgen einige Erläuterungen des zur Evaluierung verwendeten Fehlermodells sowie der Simulatorerweiterung zur künstlichen Fehlererzeugung.

5.1.1 Verwendete Benchmark-Programme

Die Benchmarksuite EEMBC[1] AutoBench 1.1 beinhaltet verschiedene Beispielprogramme, welche im C-Standard implementiert sind und typische Eigenschaften und Anforderungen von Software für eingebettete Systeme im Automotive-Bereich repräsentieren. Ziel des herausgebenden Konsortiums ist es, einen Industriestandard zu etablieren, um die Leistung eingebetteter Systeme gemäß objektiver und klar definierter Maßstäbe anwendungsbasiert evaluieren zu können.

Da im CarCore-Simulator keine Gleitkommaeinheit (engl. *Floating Point Unit*) integriert ist, führen Benchmarks mit Gleitkommaberechnungen durch ständige Aufrufe von Bibliotheksfunktionen zu einer enormen Laufzeitverlängerung und damit zu weniger realistischen Ergebnissen. Aus diesem Grund werden für die Evaluierung folgende sieben Applikationen der Benchmarksuite ausgewählt, welche keine Gleitkommaoperationen nutzen (vgl. EEMBC 2011):

[1] Embedded Microprocessor Benchmark Consortium

aifirf – Finite Impulse Response Filter Der Algorithmus dieses Benchmarks simuliert das Filtern mit endlicher Impulsantwort (auch FIR-Filter oder Transversalfilter genannt), wie es in eingebetteten Applikationen aus dem Automotive- bzw. Industrial-Bereich vorkommt. Auf die Daten des Eingabesignals werden dabei verschiedene Hochpass- und Tiefpass-FIR-Filter angewandt.

bitmnp – Bit Manipulation Dieser Benchmark repräsentiert eine rechenintensive Applikation, in welcher eine große Anzahl an Bits verändert werden muss, viele Entscheidungen aufgrund von Bitwerten getroffen werden und eine Reihe von arithmetische Operationen auf Basis von Bitwerten stattfindet. Dies hat besondere Ähnlichkeit zu Teilen eines Displays mit verschiedener Pufferung der anzuzeigenden Daten.

canrdr – CAN Remote Data Request Hierbei wird eine Applikation simuliert, in welcher ein Knoten mit CAN-(Controller Area Network-)Schnittstelle existiert, um Nachrichten im System auszutauschen. Insbesondere ist die Situation implementiert, die auftritt, wenn eine RDR-(Remote Data Request-) Nachricht von allen Knoten empfangen wird.

pntrch – Pointer Chasing Dieser Benchmark beinhaltet eine Applikation, die eine Vielzahl an Zeigerveränderungen durchführt. In einer doppelt verketteten Liste werden bestimmte Einträge gesucht, die dem Eingabewert entsprechen. Dabei wird die Anzahl der Schritte, welche zum Auffinden des Eintrages nötig sind, aufgezeichnet.

puwmod – Pulse Width Modulation Der Algorithmus dieses Benchmarks simuliert die Steuerung eines Aktuators über ein Signal, das mit Hilfe von Pulsweitenmodulation proportional zum Eingabesignal ist. Dabei wird angenommen, dass der eingebettete Prozessor über eine H-Brückenschaltung den Treiber eines Motors ansteuert.

rspeed – Road Speed Calculation In diesem Benchmark aus dem Automotive-Bereich berechnet die CPU wiederholt die Fahrgeschwindigkeit, basierend auf bestimmten Differenzen zwischen Zeitgeberwerten. Die Werte müssen dazu gefiltert werden, um Fehler durch Rauschen zu vermeiden. Außerdem muss sichergestellt sein, dass bei einem stehenden Fahrzeug nicht unendlich lange auf eine Inkrementierung des Zeitgeberwertes gewartet wird.

ttsprk – Tooth To Spark Dieser EEMBC-Benchmark simuliert eine Applikation, in welcher die CPU die Benzineinspritzung und Zündung eines Verbrennungsmotors steuert. Von besonderer Wichtigkeit sind dabei die Regulierung des Benzin-/Luftgemisches und die Zeitsteuerung der Zündung, was unter Echtzeitbedingungen erfolgen muss.

Benchmark	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Anzahl Inst.	573	1 315	256	473	777	455	1 601
Anzahl Grundb.	132	411	76	132	195	87	240
⌀ Inst./Grundb.	4,34	3,20	3,37	3,58	3,98	5,23	6,67

Tabelle 5.1: Vergleich der verwendeten EEMBC-Benchmarks nach Anzahl der Instruktionen, Anzahl der Grundblöcke und durchschnittlicher Grundblocklänge (in Instruktionen)

Die Benchmarks werden zur Evaluierung, wie in Abschnitt 4.1 erläutert, mit Hilfe des *HighTec GNU C/C++-Compiler* für den Infineon TriCore Prozessor kompiliert (vgl. TriCore 2008) und dabei entsprechend Abbildung 4.1 mit Checkpoints instrumentiert.

Im Hinblick auf die entworfene Technik zur Fehlererkennung, welche bei der Instrumentierung jeden Grundblock mit Zusatzinformationen versieht, ist es interessant, die Granularität der Grundblöcke am Beispiel der verwendeten Benchmarks etwas näher zu untersuchen. Tabelle 5.1 zeigt hierzu Details der vorgestellten EEMBC-Benchmarks[2]: Die Codelänge der Applikationen ist sehr verschieden und variiert zwischen 256 und 1 601 Instruktionen, die Länge der Grundblöcke ist dabei im Durchschnitt ähnlich – zwischen etwa 3,20 und 6,67 Instruktionen pro Grundblock. Betrachtet man die Laufzeiten der einzelnen Grundblöcke bei einer Ausführung auf dem CarCore-Prozessor, so ergibt sich die Verteilung, welche in Abbildung 5.1 zu sehen ist. Dargestellt sind alle Grundblöcke aus den sieben oben genannten EEMBC-Benchmarks, aufgetragen nach der Anzahl der Takte in einem Grundblock. Es ist zu sehen, dass die meisten Blöcke weniger als 30 Takte zur Ausführung benötigen. Die Tatsache, dass viele Grundblöcke im Bereich von 60-70 Takten liegen, ist durch die Call- und Return-Behandlung des CarCore-Prozessors zu erklären, welche etwas mehr Laufzeit als bei anderen Architekturen in Anspruch nimmt. Wichtig ist auch die Tatsache, dass kein Block eine längere Ausführungszeit als rund 160 Takte hat.

5.1.2 Fehlermodell

Entsprechend der in Abschnitt 3.3 erläuterten Motivation stehen bei der Evaluierung transiente Fehlerursachen im Vordergrund. Diese äußern sich in der Regel als Bitflips (vgl. Abschnitt 3.3.1), also als (temporäre) Änderung des Zustands eines Bits. Interessant ist nun, solche Bitflips auf systematische Weise künstlich

[2]Betrachtet wird hierbei nur die Benchmark-Implementierung, beim Kompilieren eingebundene Standardbibliotheken werden vernachlässigt.

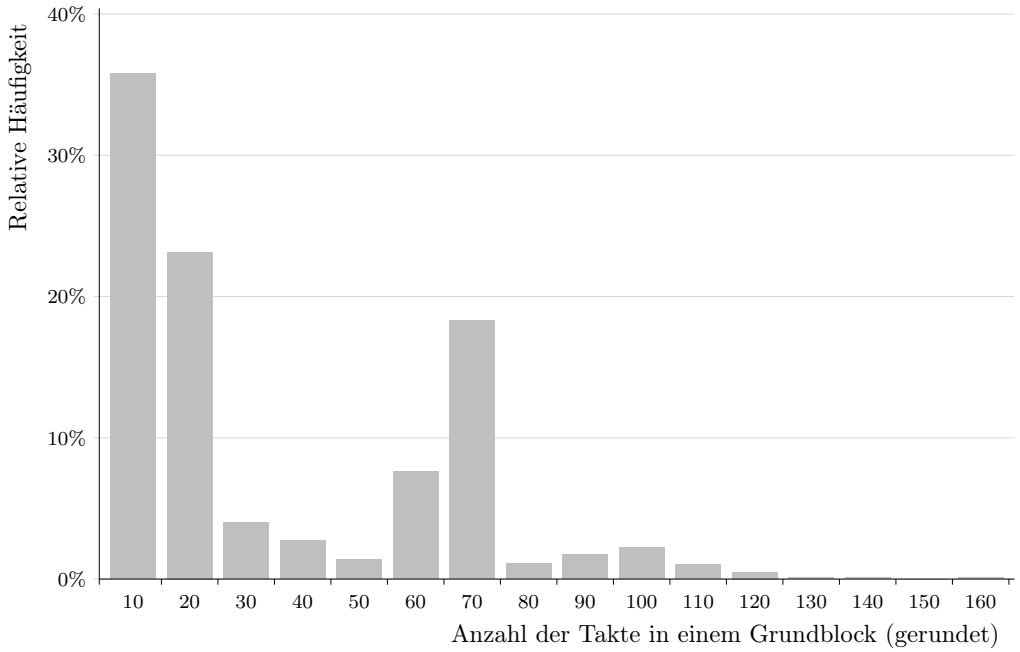


Abbildung 5.1: Häufigkeit der Laufzeiten einzelner Grundblöcke der verwendeten EEMBC-Benchmarks

zu erzeugen und die jeweiligen Auswirkungen zu beobachten. So können nach einer ausreichend großen Anzahl an Testläufen Aussagen getroffen werden, welcher Anteil der Bitflips überhaupt zu einem Fehlzustand führt und wie viele dieser Fehler durch den Erkennungsmechanismus identifizierbar sind. Auch die Latenz der Fehler lässt sich durch diese Art der Simulation gut ermitteln. Da ein gleichzeitiges Auftreten mehrerer Bitänderungen extrem unwahrscheinlich ist, wird in diesem Kontext nur ein einzelner Bitflip pro Ausführung betrachtet, um dessen Konsequenzen zu verfolgen.

Konkret sollen Bitflips im Instruktionsspeicher modelliert werden; dies hat verschiedene Gründe: Zum einen sind deren Auswirkungen extrem vielfältig (vgl. Abschnitt 3.3.3) und ohne konkrete Modellierung und Simulation schwer zu erforschen. Zum anderen erweist sich eine zuverlässige Erkennung dieser Fehler durch bestehende Techniken oftmals als schwierig und stellt damit eine zentrale Aufgabe für einen neu entworfenen Mechanismus dar.

Zum Zweck der künstlichen Fehlererzeugung ist es hilfreich, die Befehlsbereitstellung des verwendeten CarCore-Simulators näher zu betrachten: Die Befehlshol-

aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
18 745	38 748	2 557	15 291	19 455	8 766	40 127

Tabelle 5.2: Anzahl injizierter Bitflips innerhalb der Hauptfunktion des jeweiligen Benchmarks

stufe (engl. *Fetch Stage*) der Prozessorpipeline liest stets von einer angegebenen Adresse des Instruktionsspeichers ein Befehlswort mit einer Länge von 64 Bit ein. Aufgrund der variablen Instruktionslänge der TriCore-Architektur (Instruktionen können 16 oder 32 Bit umfassen), beinhaltet dieses Befehlswort zwischen zwei und vier Instruktionen mit den zugehörigen Opcodes und Operanden. Eine Integration der Fehlerinjektion erfolgt nun in zwei Schritten: Zunächst wird der Simulator um zwei Eingabeparameter, eine Adresse des Instruktionsspeichers sowie einen Wert zwischen 0 und 63, erweitert. Im zweiten Schritt wird die Arbeitsweise der Befehlsholstufe angepasst: Erfolgt ein Zugriff auf die Adresse, welche mit dem Eingabeparameter übereinstimmt, so wird im geholten Befehlswort das im zweiten Parameter spezifizierte Bit umgekippt. Ist die bewusste Manipulation erfolgt, so wird dies intern vermerkt und auf eine Erzeugung weiterer Bitflips innerhalb der aktuell laufenden Simulation verzichtet.

Mit Hilfe der vorliegenden Modellierung ist es nun möglich, iterativ Simulationen mit verschiedenen Eingabeparametern auszuführen, d.h. Simulationen, welche systematisch an einer festgelegten Speicherstelle genau einen Bitflip verursachen. Die Untersuchung der Konsequenzen wird dabei umso aussagekräftiger, je mehr Simulationen mit unterschiedlichen Bitflips ausgeführt werden. Bei einer Verwendung der in Abschnitt 5.1.1 vorgestellten EEMBC-Benchmarks bietet es sich aufgrund der relativ knappen Speichergröße an, eine Änderung sämtlicher Bits innerhalb der Hauptfunktionen zu simulieren. Tabelle 5.2 gibt hierzu einen Überblick, wie viele Bitflips bei den einzelnen Benchmark-Applikationen injiziert, d.h. wie viele unterschiedliche Simulationsläufe bei der Evaluation ausgeführt werden. Bei den Angaben ist die Instrumentierung der Benchmarks bereits berücksichtigt.

5.2 Abdeckung erkannter Fehlerfälle

Unter Verwendung des beschriebenen Fehlermodells gilt es nun zu ermitteln, welcher Anteil an Fehlern durch den entworfenen Erkennungsmechanismus identifizierbar ist. Dazu werden Simulationen mit künstlich erzeugten Bitflips ausgeführt und deren Konsequenzen überwacht. Für die Auswertung wird einerseits protokolliert, ob ein Fehler durch die integrierte Technik erkannt wird, andererseits

aber auch die Folgen, falls keine Erkennung stattfindet. Interessant ist in diesem Zusammenhang auch der Vergleich zu Simulationen ohne Instrumentierung und angeschlossene Check-Einheit.

Im Folgenden soll zunächst geklärt werden, auf welche Weise die Simulationsläufe ausgewertet werden, um daraufhin die Ergebnisse im Detail zu diskutieren.

5.2.1 Kategorisierung der Simulationsläufe

Das Verhalten eines Simulationslaufes mit injiziertem Bitflip lässt sich zunächst in drei offensichtliche Gruppen einteilen:

- **Terminierung:** Die Ausführung terminiert an einem vorgesehenen Punkt. Dabei bleibt ohne nähere Untersuchung unklar, ob der Bitflip einen Fehler bei der Simulation und damit möglicherweise ein falsches Ergebnis ausgelöst hat. Sicher ist jedoch, dass ein eventuell aufgetretener Fehler zur Laufzeit unerkannt geblieben ist.
- **Abbruch:** Alternativ kann es sein, dass zur Laufzeit ein Fehler entdeckt und die Simulation abgebrochen wird. Für die Fehlererkennung können dabei einerseits der vorgestellte Mechanismus verantwortlich sein, andererseits auch verschiedene prozessorseitige Ausnahmebehandlungen. Die Evaluierung eines solchen Testdurchlaufs endet zum Zeitpunkt des Abbruchs, da weitere Maßnahmen zur Fehlerbehandlung in diesem Kontext nicht berücksichtigt werden sollen.
- **Endlosschleife:** Die dritte Möglichkeit besteht darin, dass ein durch die Injektion ausgelöster Fehler zu einer Endlosschleife bei der Simulation führt. In einem solchen Fall wird der Durchlauf nach einer gewissen maximalen Laufzeit abgebrochen.

Um die Abdeckung der durch den integrierten Mechanismus erkannten Fehlerfälle ermitteln zu können, ist eine genauere Unterscheidung nötig: Es muss am Ende jeder terminierten Simulation analysiert werden, ob die injizierte Fehlerursache überhaupt ein Fehlverhalten bei der Simulation ausgelöst hat. Um dies festzustellen, wird im Vorfeld ein definitiv korrekter Durchlauf als Referenz gespeichert. Die Anzahl der benötigten Takte sowie die Registerbelegung am Ende des Durchlaufs stellen ein wichtiges Indiz dar, ob ein künstlich erzeugter Bitflip Konsequenzen hervorgerufen hat. Stimmen diese Werte nicht mit den Referenzwerten überein, so wird die Simulation im Rahmen der vorliegenden Evaluierungen als fehlerhaft betrachtet.

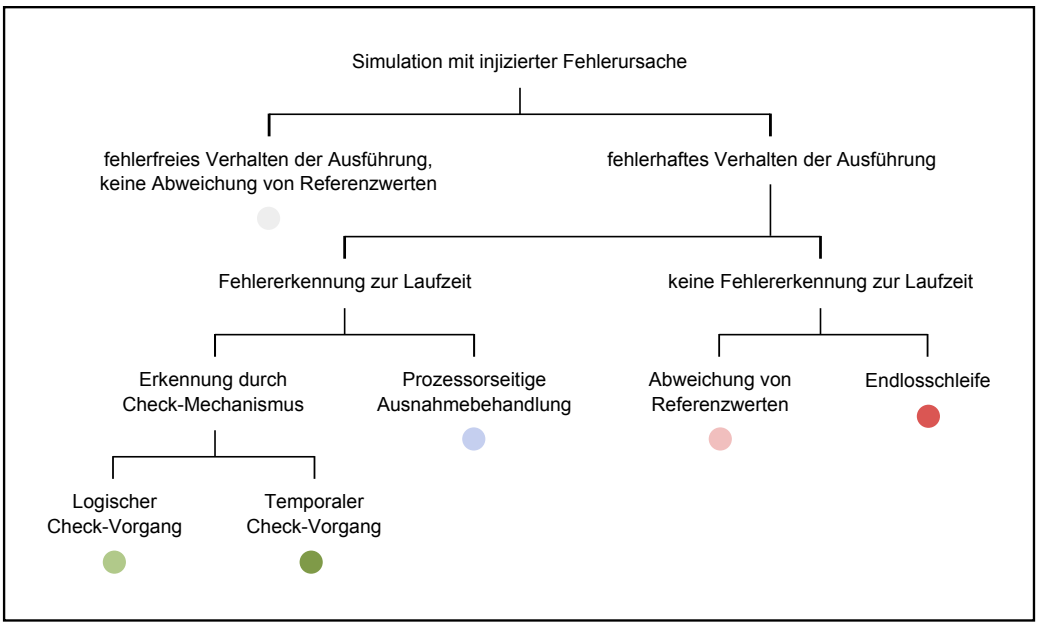


Abbildung 5.2: Kategorisierung der Simulationsläufe

Für die weitere Evaluierung ist es somit hilfreich, die Kategorien der Simulationsläufe von einem aufgetretenen Fehlverhalten bzw. der entsprechenden Erkennung abhängig zu machen. Es ergibt sich damit die Einteilung aus Abbildung 5.2: Zunächst wird unterschieden, ob es bei der Ausführung zu fehlerfreiem oder fehlerhaftem Verhalten kommt. In letzterem Fall gilt es zu trennen, ob zur Laufzeit eine erfolgreiche Fehlererkennung – entweder durch einen logischen bzw. temporalen Check-Vorgang oder durch eine Ausnahmebehandlung des Prozessors – stattfindet oder nicht. Falls fehlerhaftes Verhalten zur Laufzeit unerkannt geblieben ist, zeigt sich dies entweder durch eine Terminierung mit falschen Ergebnisse (d.h. Abweichungen von den im Vorfeld ermittelten Referenzwerten) oder durch eine Endlosschleife bei der Ausführung.

5.2.2 Auswertung der Ergebnisse

Dieser Abschnitt soll nun das Verhalten der simulierten EEMBC-Benchmarks bei Erzeugung künstlicher Fehler näher beleuchten. Fasst man die Konsequenzen aus allen möglichen Bitflips (entsprechend Tabelle 5.2 also insgesamt 143 689 Simulatorläufe mit jeweils einem Bitflip) zusammen, ergibt sich die in Abbildung 5.3, Zeile A dargestellte Verteilung.

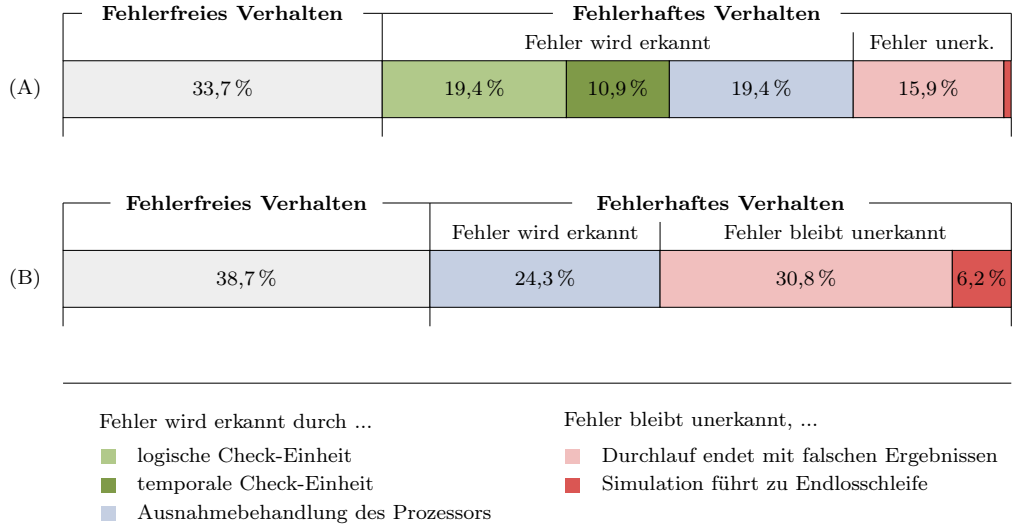


Abbildung 5.3: Zusammenfassung der Ergebnisse aus allen Simulationsdurchläufen mit injiziertem Bitflip bei (A) aktiviertem oder (B) nicht aktiviertem Check-Mechanismus (ohne Software-Instrumentierung)

Das Diagramm zeigt: In 66,3 % der Simulationsläufe bewirkt die Injektion ein fehlerhaftes Verhalten, während in 33,7 % keine Abweichung von einer korrekten Ausführung zu erkennen ist. Dieser relativ hohe Anteil, in welchem Bitflips keinerlei Auswirkungen zeigen, lässt sich auf verschiedene Ursachen zurückführen:

- Dadurch dass die Injektion in der Befehlsholstufe der Prozessorpipeline erfolgt, ist nicht sichergestellt, dass die modifizierte Instruktion überhaupt ausgeführt wird. Da stets 64 Bit (d.h. bis zu vier Instruktionen) bei jedem einzelnen Fetch-Vorgang eingelesen werden, ist es bei Kontrollstrukturen häufig der Fall, dass die nachfolgenden Instruktionen verworfen werden.
- Der verwendete Befehlssatz des TriCore-Prozessors beinhaltet einige ungenutzte Bits in verschiedenen Opcodes. Ein Bitflip an einer solchen Stelle wird somit ebenfalls kein fehlerhaftes Verhalten auslösen, da der Wert keine weitere Verwendung hat.
- Die Bitflips werden natürlich auch in Instruktionen injiziert, welche einen instrumentierten Checkpoint repräsentieren. Dabei kommt es häufig vor, dass der in die Instrumentierung integrierte WCET-Wert verändert wird. Im Falle einer Erhöhung wird die Zeitschranke definitiv weiterhin eingehalten und es findet keine Fehlererkennung bzw. weitere Auswirkung des Fehlers statt.

Von den übrigen 66,3 % der Simulationen, in welchen die Injektion ein fehlerhaftes Verhalten bewirkt, wird in insgesamt 49,7 % der Fälle ein Fehler zur Laufzeit erkannt. 19,4 % sind dabei auf die logische Check-Einheit zurückzuführen, d.h. resultieren aus logischen Überprüfungen der IDs der Checkpoints, während 10,9 % über die temporale Check-Einheit aufgrund einer Überschreitung der instrumentierten WCET-Werte ermittelt werden. In den übrigen 19,4 % der Ausführungen meldet eine prozessorseitige Ausnahmebehandlung den Fehler, davon in 13,6 % der Fälle wegen ungültiger Opcodes und in 5,8 % wegen fehlerhaft lesender oder schreibender Speicherzugriffe. In 15,9 % der Simulationsdurchläufe terminiert die Ausführung der entsprechenden Applikation zunächst regulär und ohne Fehlererkennung zur Laufzeit. Überprüft man das Simulationsergebnis allerdings mit den Referenzwerten, so sind Abweichungen hinsichtlich Ausführungsdauer und Registerbelegung zu erkennen. Schließlich bleiben 0,8 % der Ausführungen, welche die insgesamt erlaubte Simulationszeit aufgrund einer Endlosschleife überschreiten. Auch hier verursacht der Bitflip ein Fehlverhalten, welches zur Laufzeit unerkannt bleibt.

Um die Arbeitsweise der Fehlererkennung besser bewerten zu können, ist ein Vergleich zu Simulationsläufen ohne Check-Mechanismus interessant. Zeile B aus Abbildung 5.3 veranschaulicht hierzu eine Gegenüberstellung der Ergebnisse zu Ausführungen der ursprünglichen Benchmark-Programme ohne Software-Instrumentierung. Hierbei verkürzt sich durch das Fehlen der Checkpoints der Applikationscode, sodass nur eine geringere Anzahl unterschiedlicher Bitflips (insgesamt 83 782 statt 143 673) injizierbar ist.

Zunächst fällt auf, dass der Anteil an fehlerfreien Simulationen bei den Benchmark-Programmen ohne Instrumentierung etwas höher bei 38,7 % liegt. Dies lässt sich wie folgt erklären: Bei der Instrumentierung werden zusätzliche Instruktionen eingefügt, welche den Checkpoint repräsentieren. Da exakt diese Informationen vom Hardware-Checker zur Laufzeit ausgewertet werden, ist die Korrektheit dieser Instruktionen von großer Wichtigkeit. Wird also ein Bitflip innerhalb der instrumentierten Checkpoints injiziert, verursacht dies fast immer eine Fehlererkennung aufgrund falscher IDs oder einer unterschrittenen WCET-Grenze. Das heißt im Umkehrschluss: Bei der Ausführung der Benchmarks ohne Instrumentierung fehlen diese „sensiblen“ Instruktionen und es kommt bei Bitflips häufiger zu fehlerfreien Simulationsdurchläufen.

Betrachtet man nun die übrigen 61,3 % der Simulationen ohne Instrumentierung, so ist fehlerhaftes Verhalten zu beobachten. Davon wird in 24,3 % der Fälle eine Ausnahmebehandlung des Prozessors ausgelöst (15,2 % ungültige Opcodes, 9,1 % fehlerhafter Speicherzugriff), d.h. der Fehler wird zur Laufzeit erkannt. Durch das Fehlen des Check-Mechanismus steigt dieser Anteil verglichen zu den oben

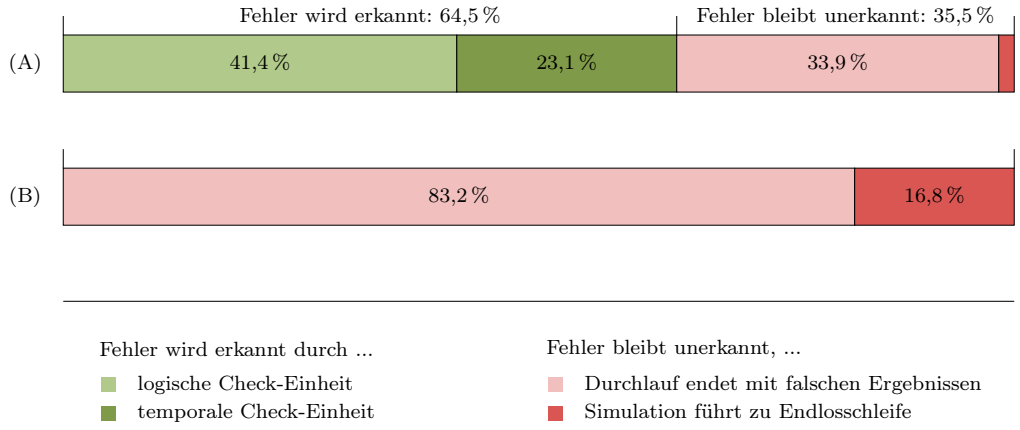


Abbildung 5.4: Zusammenfassung der Simulationsergebnisse mit fehlerhaftem Verhalten, das nicht durch prozessorseitige Ausnahmebehandlungen erkennnbar ist (bei (A) aktiviertem oder (B) nicht aktiviertem Check-Mechanismus)

beschriebenen Messungen etwas an: Verschiedene Bitflips wären bereits mit niedrigerer Erkennungslatenz vom Check-Mechanismus als Fehler erkannt worden, ehe diese nun in weiterer Konsequenz eine prozessorseitige Ausnahmebehandlung auslösen. Schließlich verbleiben 37,0 % der Simulationen, bei welchen fehlerhaftes Verhalten entsteht, jedoch keine Erkennung zur Laufzeit stattfindet. In 30,8 % dieser Fälle endet der Durchlauf mit Ergebnissen, die nicht den im Vorfeld ermittelten Referenzwerten entsprechen, während 6,2 % der Simulationen zu einer Endlosschleife führen.

Die in Abbildung 5.3 dargestellte Verteilung stellt zwar eine anschauliche Zusammenfassung der Simulationsergebnisse dar, allerdings ist das Potenzial der Fehlererkennung der vorgestellten Technik noch nicht offensichtlich: Zum einen gibt es keinerlei Motivation, Bitflips zu erkennen, welche *keinen* Fehler verursachen. Zum anderen ist es nicht nötig, Fehlverhalten zu erkennen, das bereits eine prozessorseitige Ausnahmebehandlung auslöst. Vielmehr ist es wichtig, auf diejenigen Simulationsläufe zu fokussieren, in welchen ein fehlerhaftes Verhalten vorliegt, das *nicht* durch eine Ausnahmebehandlung erkannt wird. Abbildung 5.4 zeigt dazu ausschließlich die Verteilung der Simulationsdurchläufe, in welchen ein (prozessorseitig) unerkannter Fehler vorliegt. Hierbei ist zu sehen, dass die integrierte Technik in 64,5 % der Fälle eine erfolgreiche Fehlererkennung veranlasst: 41,4 % auf Basis logischer und 23,1 % mit Hilfe temporaler Checks. Nur noch 35,5 % der Fehlerfälle bleiben unerkannt, wobei 33,9 % zu falschen Ergebnissen

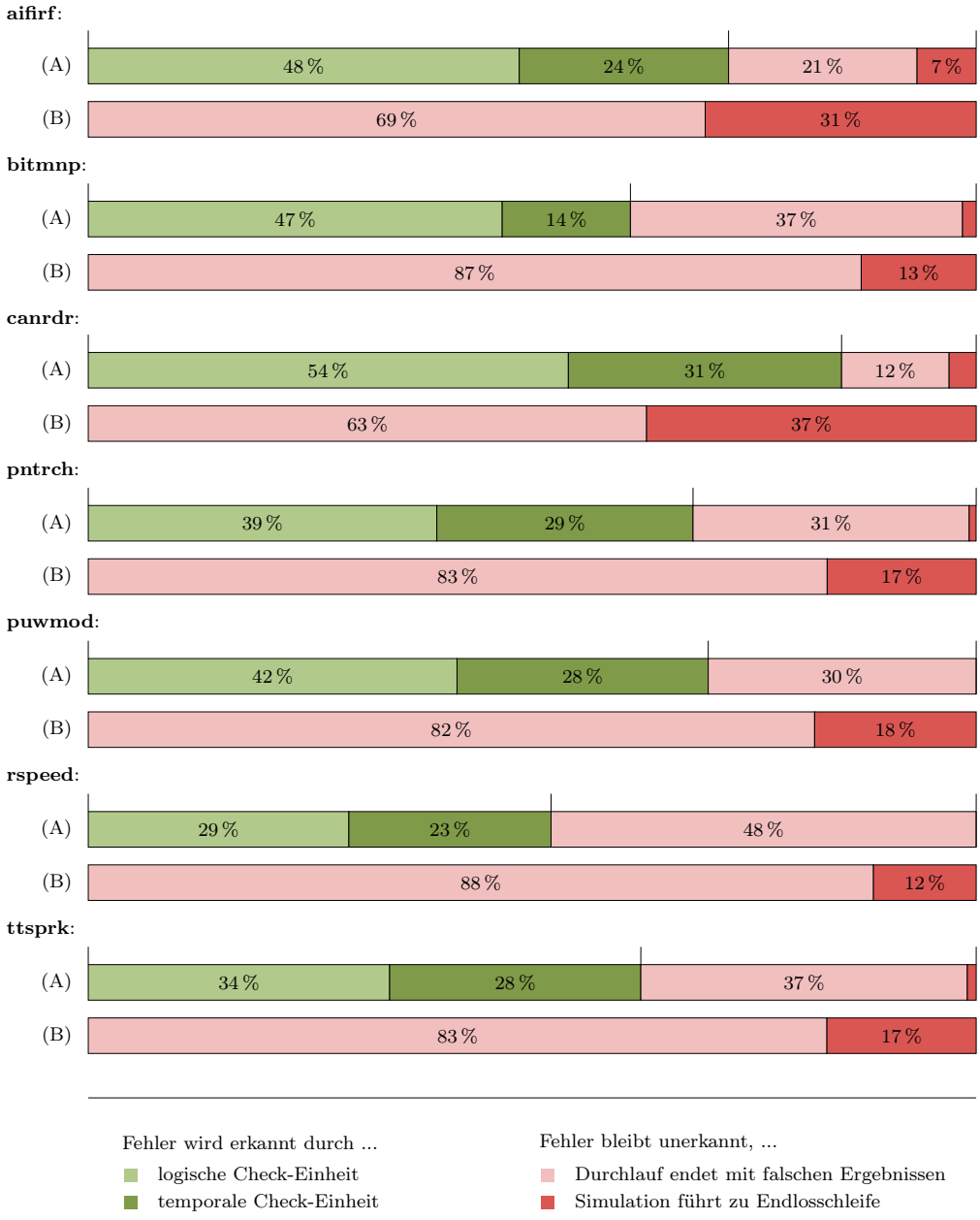


Abbildung 5.5: Ergebnisse der einzelnen Benchmarks mit fehlerhaftem Verhalten, das nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist (bei (A) aktiviertem oder (B) nicht aktiviertem Check-Mechanismus)

am Ende des Durchlaufes führen und lediglich 1,6 % eine Endlosschleife auslösen. Vergleicht man diese Zahlen mit den Simulationsergebnissen ohne Check-Einheit, so ist zu beobachten, dass 83,2 % der fehlerhaften Durchläufe ohne prozessorseitige Erkennung zu falschen Endergebnissen führen, während die restlichen 16,8 % eine Endlosschleife hervorrufen. Damit lässt sich direkt der positive Effekt der vorgestellten Technik ablesen: Der Anteil an fehlerhaften Simulationen, die mit falschen Ergebnissen terminieren, reduziert sich um fast 60 % (von 83,2 % auf 33,9 % mit integriertem Check-Mechanismus), der Anteil an unerkannten Endlosschleifen um über 90 % (von 16,8 % auf 1,6 %).

Abbildung 5.5 zeigt das Verhalten der einzelnen EEMBC-Benchmarks: Dabei wurden zunächst sämtliche möglichen Bitflips entsprechend Tabelle 5.2 in jeweils einem Durchlauf simuliert. Die Auswertung beschränkt sich aber analog zur Darstellung in Abbildung 5.4 auf diejenigen Simulationsläufe, in welchen die Injektion ein fehlerhaftes Verhalten auslöst, das nicht durch eine Ausnahmebehandlung des Prozessors erkennbar ist. Zudem ist für jeden einzelnen Benchmark der Vergleich zu sehen, wie sich die Simulationen ohne Check-Einheit verhalten. Im Hinblick auf die vorgestellte Erkennungstechnik ist zu beobachten: Der Anteil erfolgreich identifizierter Fehlerfälle variiert zwischen etwa 52 % bei Benchmark *rspeed* und 85 % bei *canrdr*. In der Regel werden etwas mehr Fehler mit Hilfe der logischen als der temporalen Prüfungen erkannt. Die Anzahl der Fälle, in welchen ein falsches Simulationsergebnis unerkannt bleibt, wird ebenso wie auftretende Endlosschleifen bei allen Benchmarks deutlich reduziert.

Als Fazit lässt sich feststellen, dass der vorgestellte Mechanismus zur Erkennung von Kontrollflussfehlern in Echtzeitsystemen eine hohe Abdeckung erkannter Fehlerfälle aufweist. Mehr als 64 % der injizierten Bitflips, welche ein ansonsten unerkanntes fehlerhaftes Verhalten verursachen, können erfolgreich als Fehler identifiziert werden. Der Anteil unerkannter Fehler verringert sich drastisch, wobei hier besonders die Reduktion der Endlosschleifen um mehr als 90 % hervorzuheben ist.

5.3 Latenz der Fehlererkennung

Neben der Abdeckung erkannter Fehlerfälle gilt es, die zugehörige Latenz, also die Zeitspanne zwischen dem Auftreten einer Fehlerursache und der Erkennung des ersten dadurch verursachten Fehlers, zu evaluieren. Dazu sollen nun alle Simulationsläufe mit Bitflip aus Abschnitt 5.2 betrachtet werden, bei welchen eine erfolgreiche Erkennung durch den implementierten Mechanismus stattfindet (in Abbildung 5.3 in Grüntönen dargestellt). Insgesamt sind dies 43 516 Fälle, davon

Latenz	≤ 10	$\leq 10^2$	$\leq 10^3$	$\leq 10^4$	$\leq 10^5$	$\leq 10^6$
Häufigkeit	33,91 %	94,76 %	96,76 %	97,68 %	98,96 %	99,90 %

Tabelle 5.3: Größenordnung der gemessenen Latenzen (in Prozessortakten)

27 911 mit logischer und 15 605 mit temporaler Fehlererkennung. Gemessen wird dabei stets die Anzahl an Taktzyklen zwischen Injektion des Bitflips und Reaktion der zugehörigen Fehlererkennung. Tabelle 5.3 veranschaulicht die Größenordnung der ermittelten Latenzen: Es ist zu sehen, dass die meisten Erkennungsfälle bereits nach weniger als 100 Taktzyklen auftreten; nur etwa 5 % der Simulationen haben eine höhere Latenz. Mehr als 1 000 Takte Latenzzeit tritt bei etwa 3 % der Ausführungen, mehr als 100 000 Takte nur noch bei etwa 1 % der Ausführungen auf.

Eine detaillierte Untersuchung dieser Fälle mit extrem hohen Latenzen zeigt, dass es sich dabei in der Regel um Bitflips innerhalb der Typ-Angabe eines Checkpoints handelt. Beispielsweise kann ein Checkpoint, der einen bedingten Sprung anzeigt (Typ 2, vgl. Seite 60), durch einen Bitflip zu einem Checkpoint vom Typ 3, der einen Funktionsaufruf repräsentiert, modifiziert werden. In der Folge bedeutet dies, dass die Hardware-Check-Einheit bei der Ausführung des Checkpoint den impliziten Nachfolger fälschlicherweise auf den Stack-Speicher legt. Da der Vergleich mit dem explizit angegebenen Nachfolger bei beiden Checkpoint-Typen analog stattfindet, erkennt die Check-Einheit somit erst einen Fehler, sobald die fehlerhaft gespeicherte ID vom Stack geholt und für den folgenden Vergleich verwendet wird – dies kann Tausende Takte später der Fall sein. Ein ähnliches Verhalten entsteht, falls durch einen Bitflip ein Checkpoint vom Typ 3 zu Typ 2 modifiziert wird. Dabei wird der implizite Nachfolger *nicht* auf den Stack-Speicher gelegt, wie eigentlich beabsichtigt. Somit kommt es ebenfalls bei einem nachfolgenden Stack-Zugriff und dem damit verbundenen logischen Vergleich zu einer extrem späten Fehlererkennung. Da die Simulation sämtliche möglichen Bitflips auch innerhalb der Checkpoints berücksichtigt, kommt es für jeden einzelnen Checkpoint im Programm einmal zu den beiden oben genannten Fällen. Eine mögliche Lösung dieses Programms bestünde darin, die Typen-Repräsentation der Checkpoints zu verändern: Unterscheidet sich die Kodierung der unterschiedlichen Typen um mehr als ein Bit (die Hamming-Distanz ist größer 1), kann ein einzelner Bitflip keinen veränderten gültigen Typen erzeugen. Nachteilig ist dabei, dass weniger Bit zur Darstellung der restlichen Informationen im Checkpoint verfügbar bleiben. Aus diesem Grund wird diese Lösung in der vorgestellten Implementierung und Evaluierung vernachlässigt. Vielmehr soll der Fokus der folgenden Analysen auf den Simulationsfällen mit Latenzzeiten unter 100 Takten liegen, welche mehr als 94,7 % ausmachen.

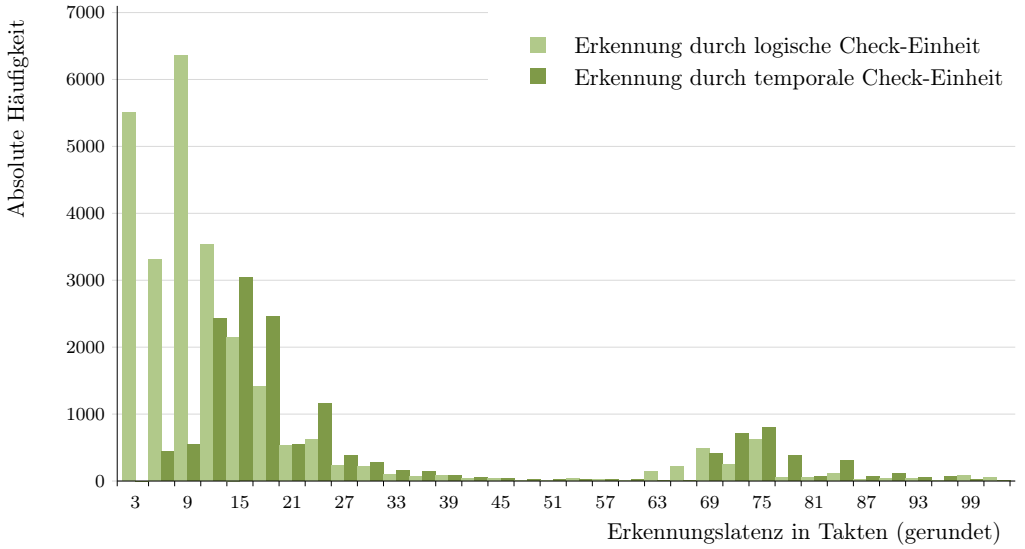


Abbildung 5.6: Häufigkeiten der Erkennungslatenzen mit Vergleich zwischen logischer und temporaler Fehlererkennung

Abbildung 5.6 zeigt detailliert die Häufigkeiten der Erkennungslatenzen, summiert über alle Benchmarks. Hierbei ist eine große Ähnlichkeit zu Abbildung 5.1, welche die Größe der Grundblöcke nach deren Häufigkeiten darstellt, festzustellen. Unterschieden wird nun zwischen logischer und temporaler Fehlererkennung: Es ist zu sehen, dass besonders die logische Erkennung meist mit äußerst niedrigen Latenzen von weniger als 15 Takten verbunden ist. Mehr als 30 Takte sind seltener zu beobachten. Die temporale Fehlererkennung hat demgegenüber ein Maximum bei etwa 15-20 Takten Latenzzeit. Zudem gibt es ein verstärktes Vorkommen höherer Latenzen im Größenbereich von 60-70 Takten. Dieses Phänomen ist durch die relativ lange Ausführungszeit von Call- und Return-Befehlen im CarCore zu erklären: In den betreffenden Grundblöcken wird die WCET-Abschätzung recht hoch angesetzt, sodass ein auftretender Fehler erst nach einem etwas längeren Zeitraum vom Erkennungsmechanismus identifiziert werden kann. Im Allgemeinen lässt sich der leichte Unterschied zwischen den Latenzen bei logischer und temporaler Erkennung einfach veranschaulichen: Während ein logischer Kontrollflussfehler unmittelbar beim Erreichen des nachfolgenden Grundblocks und der Auswertung der entsprechenden Checkpoint-ID festzustellen ist, muss für einen Fehler im Zeitverhalten zunächst der angegebene WCET-Wert ablaufen, damit das Fehlverhalten offensichtlich und erkennbar wird.

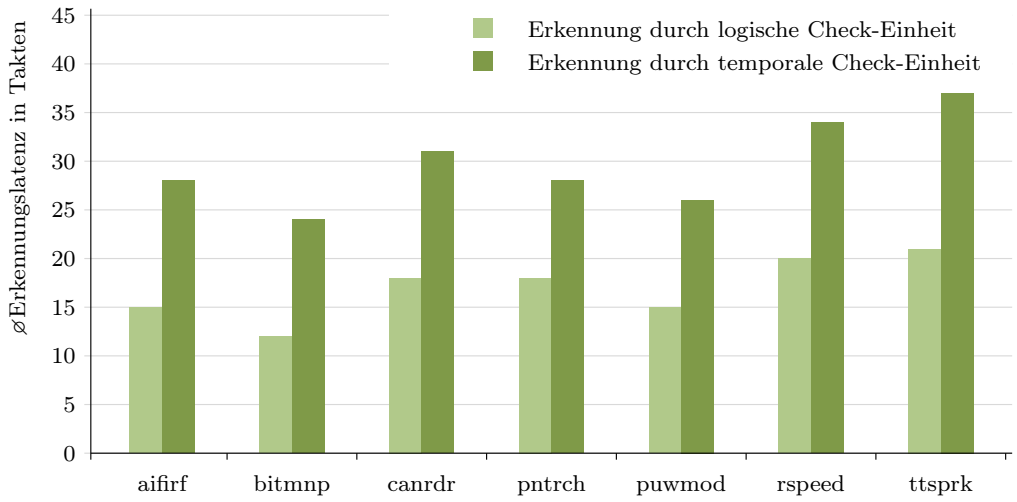


Abbildung 5.7: Vergleich der durchschnittlichen Erkennungslatenzen bei Fehlern in den einzelnen Benchmark-Programmen

Betrachtet man die Latenzen der ausgeführten Benchmark-Applikationen im Einzelnen, so ergeben sich die in Abbildung 5.7 dargestellten Durchschnittswerte. Dabei werden wiederum nur die Fälle berücksichtigt, in welchen die Latenzzeit kürzer als 100 Takte ist. Es ist zu sehen, dass sich die Werte bei der logischen Fehlererkennung recht wenig unterscheiden (zwischen etwa 12 Takten bei *bitmnp* und etwa 21 Takten bei *ttsprk*). Geringfügig stärkere Abweichungen gibt es bei der Erkennung mit Hilfe der temporalen Check-Einheit: Hier liegen die Durchschnittswerte der Benchmarks zwischen 24 Takten bei *bitmnp* und 37 Takten bei *ttsprk*. Diese Tatsache erklärt sich durch die unterschiedliche Größe der Grundblöcke: Die Applikation *bitmnp* beinhaltet beispielsweise viele Grundblöcke mit sehr kurzer Ausführungszeit. Diese werden folglich mit niedrigen WCET-Werten instrumentiert, so dass die temporale Fehlererkennung zur Laufzeit mit kürzeren Latenzen als bei anderen Benchmarks arbeitet, bei welchen Grundblöcke durchschnittlich höhere Laufzeiten haben und entsprechend mit höheren WCET-Abschätzungen instrumentiert sind.

Zusammenfassend lässt sich feststellen, dass sich der vorgestellte Mechanismus durch sehr geringe Latenzen bei der Fehlererkennung auszeichnet. Die Evaluierung ergibt zwar zunächst bei Berücksichtigung aller Simulationen einen sehr hohen Mittelwert von 4300 Takten zwischen der Injektion des Bitflips und der Identifikation des Fehlers; lässt man jedoch ca. 5 % nicht repräsentativer Fälle (wie oben beschrieben) außer Acht, so verringert sich der Durchschnitt auf nur

noch 21,4 Takte. Die Erkennung mit Hilfe der logischen Check-Einheit ist dabei im Mittel etwas frühzeitiger mit nur 16,4 Takten Latenz, während die Erkennung auf Basis temporaler Check-Vorgänge durchschnittlich mit 30,3 Takten Latenzzeit verbunden ist.

Setzt man die Ergebnisse in Relation zu einem gewöhnlichen Watchdog-Timer (wie in Abschnitt 3.4.1 erläutert), so wird die niedrige Latenz des vorgestellten Erkennungsmechanismus noch deutlicher: Bei dem von Brinkschulte und Ungerer (2010, S. 224) beschriebenen Mikrocontroller ATmega128A liegt die niedrigste zu setzende Watchdog-Auslösezeit bei 14 Millisekunden. Unter Annahme einer Taktfrequenz von 16 MHz ergibt sich damit im Fehlerfall eine Latenz von bis zu 224 000 Prozessortakten.

5.4 Aufwandsanalyse

Der vorgestellte Mechanismus zur Prüfung des Kontrollflusses in Echtzeitsystemen erhöht die Verlässlichkeit eines Systems, verursacht jedoch Zusatzaufwand. Im Rahmen der Code-Instrumentierung auf Software-Basis werden zusätzliche Instruktionen in den Applikationen eingefügt, was sowohl zu einer längeren Ausführungszeit als auch zu einem höheren Bedarf an Instruktionsspeicher führt. Ziel ist es, einen guten Kompromiss zu finden: Einerseits soll der Mehraufwand möglichst gering gehalten werden, andererseits ist ein gewisses Maß akzeptabel, solange das Ergebnis dadurch entscheidend verbessert wird.

Ziel dieses Abschnitts ist es, die einzelnen zur Evaluierung verwendeten Benchmark-Applikationen hinsichtlich ihres Mehraufwands näher zu untersuchen. Dazu soll zunächst die Ausführungszeit und daraufhin der Speicherbedarf betrachtet werden.

5.4.1 Zusätzlich benötigte Ausführungszeit

Um einen Vergleich der Laufzeiten der verwendeten Benchmarks zu ermitteln, wird jede Applikation unter gleichen Bedingungen mit und ohne Software-Instrumentierung ausgeführt. Eine Fehlerinjektion findet bei diesen Simulationen nicht statt. Abbildung 5.8 veranschaulicht die Ergebnisse: Im günstigsten Fall (bei Benchmark *rspeed*) verlängert sich die Ausführungszeit um 4,5 %, im schlechtesten Fall (*bitmnp*) um 21,7 %. Der Mittelwert über alle verwendeten Benchmarks liegt bei 12,2 % an zusätzlicher Laufzeit. Setzt man diesen Aufwand mit der Anzahl durchlaufener Checkpoints ins Verhältnis, so ergibt sich eine durchschnittliche Ausführungszeit von 1,925 Takten pro Checkpoint. Wie in Abschnitt 4.2.2 beschrieben,

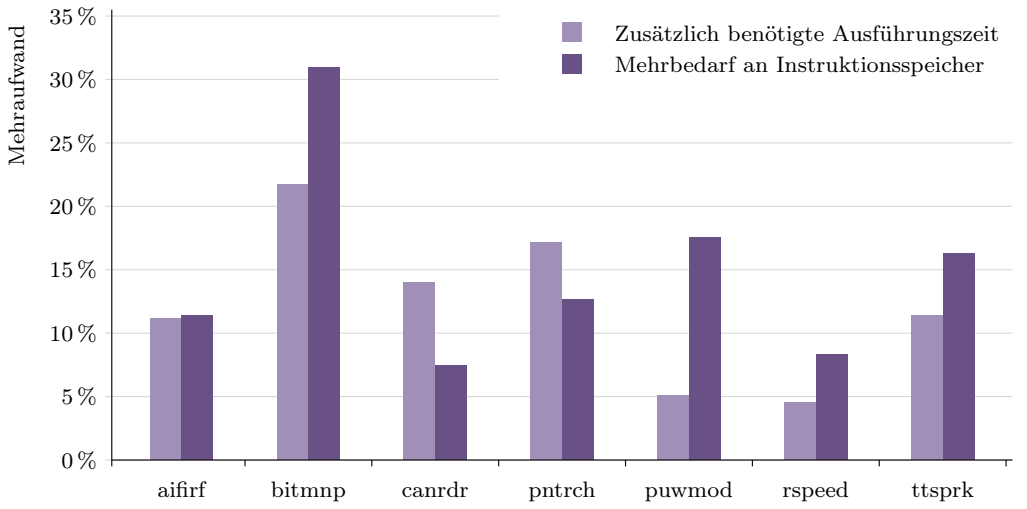


Abbildung 5.8: Mehraufwand der einzelnen Benchmark-Programme

sind für die Umsetzung eines Checkpoints drei Prozessorinstruktionen nötig, von welchen der CarCore-Prozessor zwei parallel ausführen kann. Es kostet also in der Regel nur zwei Takte an Laufzeit, um einen Checkpoint abzuarbeiten. Dass der gemessene Mittelwert nur 1,925 beträgt, lässt sich durch die veränderte Instruktionsabfolge nach der Instrumentierung erklären: Es existiert nun eine höhere Anzahl an Fällen, in welchen die Möglichkeiten paralleler Ausführungen im CarCore genutzt werden können.

Bekanntlich liegt in Echtzeitsystemen der Fokus weniger auf einer Steigerung der Performanz, als auf der definitiven Garantie einer maximalen Ausführungszeit. Da der Instrumentierungsprozess natürlich die WCET einer Applikation erhöht, stellt sich die Frage, wie einfach eine neue Obergrenze der Laufzeit zu ermitteln ist. Auf der einen Seite kann garantiert werden, dass die Ausführungszeit eines Checkpoints nie länger als zwei Takte dauert. Andererseits kommt es durch das Einfügen eines Checkpoints zu Verschiebungen bei bisher aufeinanderfolgenden Instruktionen. So könnten beispielsweise zwei Instruktionen nicht mehr parallel ausgeführt werden, obwohl dies vor der Instrumentierung der Fall war. Auch dazu lässt sich eine Konstante ermitteln, um eine maximale Ausführungszeit angeben zu können. Somit ist es möglich, auch ohne eine vollständig neue WCET-Analyse eine garantierte Obergrenze anzugeben. Andererseits würde eine erneute Laufzeitanalyse der entsprechenden Applikation mit Berücksichtigung aller Faktoren natürlich genauere Werte und eine weitaus geringere Überschätzung ergeben.

5.4.2 Mehrbedarf an Instruktionsspeicher

Neben der zusätzlich benötigten Ausführungszeit ist es interessant, den Mehrbedarf an Instruktionsspeicher zu evaluieren. Die gemessenen Ergebnisse beziehen sich dabei auf die Anzahl der im Instruktionsspeicher enthaltenen Befehle[3]. Ein Vergleich der vollständig kompilierten Binärdateien wäre ebenfalls möglich, allerdings würde das Einbeziehen des Datenspeichers, in welchem durch die Instrumentierung keinerlei Veränderungen vorgenommen werden, zu einer positiven Verfälschung des Ergebnisses führen. Zudem finden sich in den Binärdateien teilweise Fülldaten (engl. *Padding*), die einen Größenvergleich zusätzlich erschweren.

Abbildung 5.8 zeigt den zusätzlichen Speicherbedarf der einzelnen Benchmark-Programme: Dieser variiert zwischen 7,4 % bei Benchmark *canrdr* und 31,0 % bei *bitmnp*. Durchschnittlich wächst die Anzahl der Instruktionen der Benchmarks durch die Instrumentierung um 15,0 %. Offensichtlich besteht ein enger Zusammenhang zwischen dem gemessenen Speicherbedarf und der Anzahl an Grundblöcken aus Tabelle 5.1. Je mehr Grundblöcke eine Applikation umfasst, desto mehr Checkpoints werden natürlich im Rahmen der Instrumentierung eingefügt und desto stärker wächst die Größe des geforderten Instruktionsspeichers.

Zusammenfassend lässt sich aus der Aufwandsanalyse feststellen, dass im Durchschnitt 12,2 % an zusätzlicher Laufzeit und 15,0 % an Instruktionsspeicher durch den entworfenen Sicherheitsmechanismus benötigt werden. Verglichen zu einer vollständig redundanten Ausführung, wie bei sicherheitskritischen Systemen üblich, sind diese Werte sehr niedrig. Dennoch stellt sich die Frage, ob es nicht Optimierungsmöglichkeiten geben kann, welche sowohl die Leistungsfähigkeit als auch den Aufwand weiter verbessern können. All dies soll aber Gegenstand des folgenden Kapitels sein.

[3] Hierbei wird der vollständige Instruktionsspeicher der kompilierten Benchmark-Programme incl. eingebundener Standardbibliotheken betrachtet.

6

Optimierungstechniken

Die Evaluierungen im vorhergehenden Kapitel haben gezeigt, dass der entworfene Mechanismus zur feingranularen Erkennung von Fehlern im Kontrollfluss von Echtzeitsystemen die am Anfang von Kapitel 3 definierten Anforderungen und Ziele erfüllt. Dennoch stellt sich die Frage, ob bzw. welche Möglichkeiten zur Verbesserung existieren.

Die Motivation für potenzielle Optimierungen kommt dabei grundsätzlich aus unterschiedlichen Richtungen: Auf der einen Seite steht die Absicht, die Fehlererkennung zu verbessern. Eine noch umfassendere Abdeckung erkannter Fehlerfälle wäre wünschenswert. Gleichzeitig könnte möglicherweise die Latenz der Fehler verringert werden, d.h. eine Erkennung frühzeitiger als bisher stattfinden. Andererseits trägt es entscheidend zur Anwendbarkeit einer Technik bei, wenn der entstehende Mehraufwand gering bleibt. Somit ist es ebenso motivierend, die zusätzlich benötigte Ausführungszeit der Programme sowie deren erhöhten Speicherbedarf einzugrenzen.

Im Rahmen der vorliegenden Arbeit sollen nun die drei folgenden Optimierungstechniken näher betrachtet und untersucht werden, jeweils im Hinblick auf die oben genannten Ziele:

- **Berücksichtigung der Untergrenze der Ausführungszeit**: Im Hinblick auf eine Verbesserung der Erkennung von Fehlern im Zeitverhalten erscheint es sinnvoll, neben der maximalen auch eine minimale Laufzeit anzugeben und beim Erreichen eines Checkpoints zu überprüfen, ob diese tatsächlich überschritten wurde.

- **Reduktion der Checkpoint-Informationen**: Um Speicherverbrauch und Ausführungszeiten zu verringern, wäre es hilfreich, die Bitbreite eines Checkpoints zu reduzieren. Es stellt sich die Frage, welche Informationen verkürzt dargestellt werden könnten, ohne die Funktionalität der Fehlererkennung weit einschränken zu müssen.

- **Angleichung der Checkpoint-Abstände**: Durch die sehr unterschiedliche Länge der Grundblöcke sind die Checkpoints unregelmäßig im Applikationscode verteilt. Eine Angleichung der Abstände könnte zwei Ziele erfüllen: eine Verbesserung der Erkennung durch die Vermeidung langer Abschnitte ohne Checkpoints sowie eine Reduktion des Aufwands durch ein Zusammenfügen kurzer Passagen.

Die folgenden Abschnitte sollen diese Optimierungstechniken einzeln vorstellen. Dabei gilt es jeweils auch, die nötigen Anpassungen zur Integration in die bestehende Implementierung zu beschreiben. Zudem findet eine Evaluierung der verschiedenen Optionen statt, um die resultierenden Vor- und Nachteile zu bewerten. Auch Kombinationsmöglichkeiten sollen näher untersucht und evaluiert werden. Am Ende des Kapitels gilt es schließlich, die vorgestellten Optimierungstechniken untereinander zu vergleichen und hinsichtlich der Verbesserung der Fehlererkennung sowie der Reduktion des Mehraufwands zu bewerten.

6.1 Berücksichtigung der Untergrenze der Ausführungszeit

Wie in Abschnitt 2.2.1 erwähnt, existiert in Echtzeitsystemen neben der WCET eine weitere Größe, die das Zeitverhalten charakterisiert: Die *Best-Case Execution Time (BCET)* beschreibt die kürzeste Ausführungszeit im optimalen, bestmöglichen Fall. Mit Hilfe von Analysewerkzeugen lässt sich für eine gegebene Ausführungsumgebung eine untere Schranke der BCET eines Programmes ermitteln, d.h. eine minimale Laufzeit, welche unter keinen Umständen unterschritten wird. Gemeinsam mit der errechneten WCET-Abschätzung lässt sich damit das Zeitverhalten eines Programmes in beide Richtungen eingrenzen.

Eine Idee ist nun, die BCET-Schranke im Rahmen der vorgestellten Technik mit einzubeziehen: Damit wäre es möglich, nicht nur Fehler zu erkennen, die zu einer Überschreitung der ermittelten WCET-Abschätzung führen, sondern auch solche, die im Gegensatz dazu eine Unterschreitung des berechneten BCET-Wertes verursachen.

Typ	ID	Nachfolge-ID	WCET	BCET
2 Bit	x Bit	x Bit	y Bit	$(30 - 2 * x - y)$ Bit

Abbildung 6.1: Bitmaske mit Angabe der BCET-Abschätzung

6.1.1 Vorgehensweise

Die Prüfung der BCET soll analog zur Prüfung der WCET auf Basis der einzelnen Grundblöcke stattfinden. Dazu ist es nötig, sowohl die Instrumentierung als auch die an den Prozessor angeschlossene Hardware-Komponente zu erweitern. In der Offline-Phase wird für den sicherheitskritischen Programmcode zusätzlich eine BCET-Abschätzung ermittelt und in die Checkpoints eines jeden Grundblocks integriert. Dabei ist es – wie bereits in Abschnitt 3.1.1 erwähnt – wichtig, die Ausführung der Checkpoints selbst bei der Laufzeitanalyse zu berücksichtigen. Auf Hardware-Seite muss die Check-Komponente nun zur Laufzeit einen weiteren Zähler mitführen, der beim Erreichen eines Checkpoints zurückgesetzt wird. Findet zu jedem ausgeführten Takt eine Inkrementierung dieses Zählers statt, so muss der Wert beim Erreichen des folgenden Checkpoints garantiert über der ermittelten BCET des Grundblocks liegen. Bei jedem Erreichen eines Checkpoints erfolgt ein Vergleich zwischen dem aktuellen Wert des Zählers und den offline ermittelten und in den Checkpoints abgelegten BCET-Werten. Im Fall einer Unterschreitung dieser Werte war die Ausführung fehlerhaft.

6.1.2 Integration in die Implementierung

Um die BCET-Werte in die Implementierung zu integrieren, ist es zunächst nötig, das in Abschnitt 4.2.1 definierte Format der Checkpoints zu verändern. Ziel soll es sein, die Gesamtlänge eines Checkpoints von 32 Bit nicht zu erhöhen, zumal dies einen enormen Anstieg des Aufwands hinsichtlich Ausführungszeit und Speicherbedarf zur Folge hätte. Abbildung 6.1 zeigt die gewünschte Bitmaske eines Checkpoints mit Angabe der BCET-Abschätzung; die Herausforderung besteht nun darin, geeignete Werte für x und y zu finden.

Wird die Bitbreite zur Darstellung der WCET- und BCET-Werte kleiner gewählt als für die Angabe nötig, so besteht die Möglichkeit, diese skaliert im Checkpoint anzugeben. In diesem Fall ist natürlich eine entsprechende Anpassung des Hardware-Checkers nötig, um die Skalierung bei der Prüfung zu berücksichtigen. Eine gewisse Einschränkung bleibt jedoch bestehen: Durch die Skalierung muss die

WCET aufgerundet und die BCET abgerundet werden, was wiederum die Erkennungsmöglichkeiten und besonders die damit verbundenen Latenzzeiten negativ beeinflusst.

Enthält ein Programm mehr Grundblöcke, d.h. eine höhere Anzahl von IDs, als maximal im Checkpoint darstellbar, so bestünde die Möglichkeit, mehreren Blöcken dieselbe ID zuzuweisen und die angegebenen Nachfolger entsprechend anzupassen. Im Rahmen der Evaluierungen in diesem Abschnitt ist dies noch nicht nötig. Die Technik soll aber im Zusammenhang mit der nachfolgenden Optimierungstechnik in Abschnitt 6.2 näher erläutert und evaluiert werden.

Erweiterung des Instrumentierungstools

Das entwickelte Tool zur Instrumentierung von Applikationen wird um den sog. *BCET Calculator* erweitert (vgl. Abbildung 6.2). Ähnlich zum bereits vorgestellten WCET Calculator berechnet diese zusätzliche Komponente die minimale Ausführungszeit der einzelnen Grundblöcke. Dies erfolgt auf Basis der BCET-Abschätzungen der einzelnen Prozessorinstruktionen, welche in einer externen Tabelle gespeichert sind. Die Ergebnisse werden schließlich in der erweiterten Grundblocktabelle zur Weiterverarbeitung abgelegt.

Ähnlich zur WCET-Analyse liegt auch bei der Berechnung der BCET-Abschätzung der Fokus weniger auf der „Tightness“ der Ergebnisse, d.h. einer Abschätzung, die möglichst eng an der realen (nicht definitiv ermittelbaren) BCET liegt. Vielmehr ist es Ziel, eine garantierte Untergrenze der Ausführung anzugeben, um die Arbeitsweise des Check-Mechanismus grundsätzlich erforschen zu können.

Ergänzung der Hardware-Check-Einheit

Algorithmus 6.1 (Seite 88) zeigt die ergänzte Implementierung der temporalen Check-Einheit. Nach dem Start des Check-Mechanismus wird zu jedem Prozessortakt der intern gepeicherte $BCET\_Counter$ um den Wert 1 inkrementiert (Zeile 5). Sobald ein Checkpoint erreicht wird, erfolgt eine Prüfung, ob dieser $BCET\_Counter$ größer oder gleich der vorhergesagten $BCET$ ist (Z. 9). Ist dies nicht der Fall, wurde die Applikation offensichtlich falsch ausgeführt und ein Fehler im Zeitverhalten ist zu melden. Im korrekten Fall wird der interne $BCET\_Counter$ auf null zurückgesetzt (Z. 10) und die Vorhersage der BCET aus dem aktuell gelesenen Checkpoint intern für die nachfolgende Überprüfung gespeichert (Z. 11).

Abbildung 6.2: Ablauf der Instrumentierung mit Berücksichtigung einer zeitlichen Untergrenze der Ausführung

Algorithmus 6.1 Arbeitsweise der optimierten temporalen Check-Einheit

```
1: wiederhole
2:    wenn Check-Mechanismus gestartet dann
3:       dekrementiere WCET_Countdown
4:       prüfe ob WCET_Countdown ≥ 0
5:       inkrementiere BCET_Counter
6:    wenn_ende
7:    wenn Checkpoint erkannt dann
8:       WCET_Countdown ← Checkpoint[WCET]
9:       prüfe ob BCET_Counter ≥ BCET
10:      BCET_Counter ← 0
11:      BCET ← Checkpoint[BCET]
12:      starte Check-Mechanismus
13:   wenn_ende
14:   warte auf folgendes Taktsignal
15: wiederhole_ende
```

6.1.3 Evaluierung

Um die Wirksamkeit der vorgestellten Optimierungstechnik mit Berücksichtigung einer zeitlichen Untergrenze der Ausführungszeit (nachfolgend auch *Optimierungstechnik 1* genannt) näher zu untersuchen, sollen im Folgenden verschiedene Evaluierungen durchgeführt werden. Die Kriterien entsprechen dabei Kapitel 5: Zunächst stehen die Abdeckung der Fehlerfälle und die Latenz der Fehlererkennung im Vordergrund, im Anschluss eine Aufwandsanalyse. Dabei kommen erneut die vorgestellten EEMBC-Benchmarks zum Einsatz; die Evaluierungsumgebung und die Art der künstlich erzeugten Fehler entsprechen vollständig den Ausführungen in Abschnitt 5.1.

Für die Evaluierung werden die Checkpoints gemäß der Bitmaske in Abbildung 6.3 instrumentiert. Die höchste Anzahl an Grundblöcken findet sich nach Tabelle 5.1 bei Benchmark *bitmnp* mit 383 Grundblöcken; somit sind 9 Bit zur eindeutigen Darstellung aller IDs ausreichend. Die WCET-Abschätzung wird, wie zuvor beschrieben, skaliert im Checkpoint abgespeichert. Hierfür eignet sich der Faktor $1/8$, um sämtliche Werte mit bestmöglicher Genauigkeit darstellen zu können. Die BCET-Grenzen werden analog um den Faktor $1/5$ skaliert.

Da sich durch die modifizierte Bitmaske das gesamte Verhalten der Erkennungstechnik bei der Erzeugung künstlicher Fehlerfälle etwas verändert, ist ein direkter Vergleich zu den Ergebnissen ohne Optimierungstechnik möglicherweise ungenau. Besonders durch die Skalierung der WCET kann es aufgrund von Rundungsfehlern zu einer verspäteten temporalen Fehlererkennung kommen. Aus diesem Grund

Typ	ID	Nachfolge-ID	1/8 WCET	1/5 BCET
2 Bit	9 Bit	9 Bit	6 Bit	6 Bit

Abbildung 6.3: Bitmaske zur Evaluierung der Optimierungstechnik mit Berücksichtigung einer zeitlichen Untergrenze der Ausführung (Opt. 1)

sollen als Referenz zusätzlich Simulationen ohne Optimierungen, jedoch mit der veränderten Bitmaske ausgeführt werden[1]. Damit kann explizit die Verbesserung gezeigt werden, die ausschließlich aus der hier vorgestellten Optimierungstechnik resultiert.

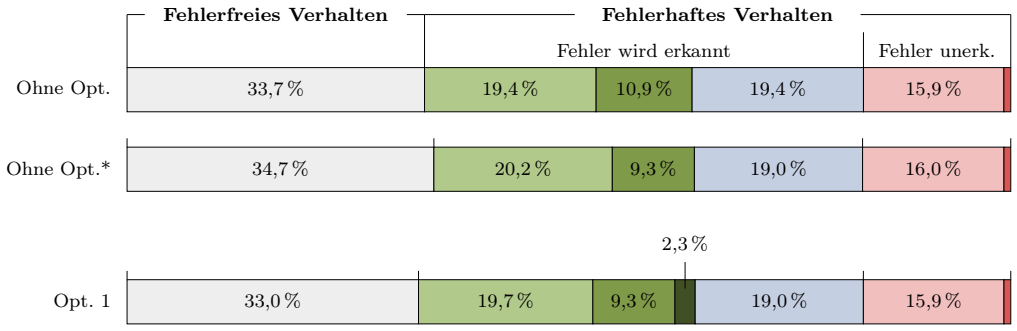
Abdeckung erkannter Fehlerfälle

In Abbildung 6.4(a) ist das Verhalten der Simulationen mit injizierten Bitflips zu sehen. Die erste Zeile zeigt die Verteilung ohne Optimierungen, die bereits in Kapitel 5.2.2 näher erläutert wurde.

In der zweiten Zeile (Ohne Opt.*) ist die Veränderung ersichtlich, welche sich ausschließlich durch die modifizierte Bitmaske ergibt: Der Anteil an Simulationen, in welchen das Verhalten fehlerfrei bleibt, steigt um etwa 1 % an; dies ist dadurch zu erklären, dass Injektionen innerhalb der im Checkpoint reservierten Bits zur Darstellung der BCET-Abschätzung aktuell keinerlei Auswirkungen auf den weiteren Programmablauf haben. Ebenfalls steigt der Anteil an Fällen, in welchen ein Fehler mit Hilfe der logischen Check-Einheit erkannt wird, von 19,4 % auf etwa 20,2 %, während sich der Anteil erkannter Fehler auf Basis der temporalen Prüfeinheit durch die ungenauere Angabe der WCET-Werte von 10,9 % auf etwa 9,3 % verringert. Diese Umverteilung erklärt sich wie folgt: Durch die Angabe der aufgerundeten und damit höheren WCET-Werte wird in manchen Grundblöcken trotz fehlerhafter Ausführung noch ein nachfolgender Block erreicht und damit eine Inkonsistenz der angegebenen IDs festgestellt. In der ursprünglichen Konstellation würde bereits der (niedrigere) WCET-Wert überschritten werden, bevor ein nachfolgender Block erreicht ist und damit eine Fehlererkennung mit Hilfe des temporalen anstelle des logischen Check-Mechanismus stattfinden würde. In der Summe nehmen die erfolgreich erkannten Fehlerfälle durch die veränderte Bitmaske um etwa 0,8 % ab. Der Anteil aller weiteren Erkennungsfälle durch prozessorseitige Ausnahmebehandlungen aufgrund von fehlerhaften Opcodes oder falschen Speicherzugriffen verändert sich durch die modifizierte Bitmaske nur minimal. Ebenso bleibt der Anteil an unerkannten Fehlerfällen etwa auf demselben Niveau.

[1] Diese Messreihe ist in den folgenden Diagrammen mit einem Stern (*) gekennzeichnet

(a) Zusammenfassung aller Simulationsergebnisse:



(b) Zusammenfassung der Simulationsergebnisse mit fehlerhaftem Verhalten, das nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist:

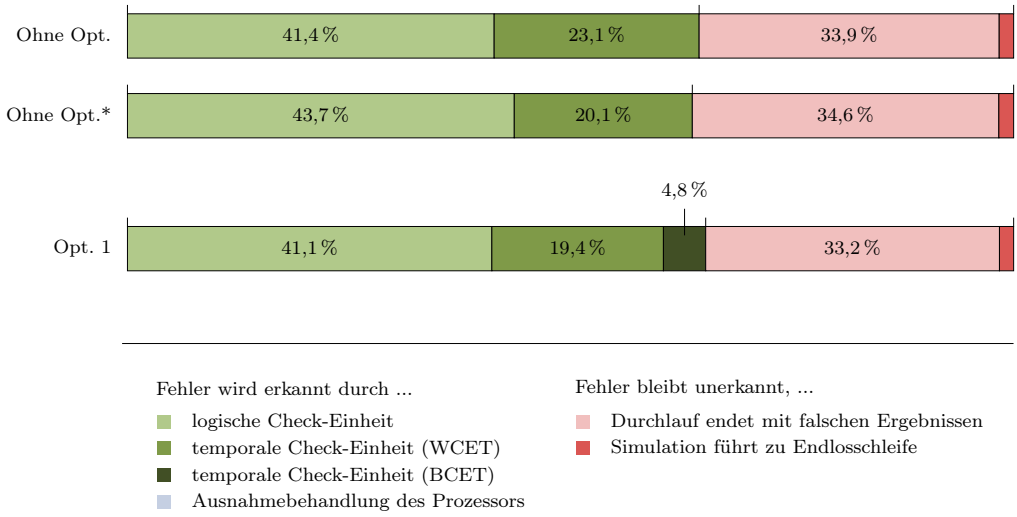


Abbildung 6.4: Abdeckung erkannter Fehlerfälle bei Berücksichtigung einer zeitlichen Untergrenze der Ausführung (Opt. 1) mit Vergleich zu Simulationen ohne Optimierungen (Ohne Opt.) sowie nur mit modifizierter Bitmaske (Ohne Opt.*)

Interessant ist nun zu beobachten, welche Auswirkung die Aktivierung der BCET-Prüfung auf die Simulationsläufe hat. Die dritte Zeile von Abbildung 6.4(a) zeigt zunächst, dass sich der Anteil an Simulationen, in denen das Ergebnis einem Durchlauf ohne Fehlerinjektion entspricht, auf 33,0 % reduziert. Wie bereits erwähnt, ist dies auf Injektionen innerhalb der im Checkpoint reservierten Bits zur Darstellung der BCET zurückzuführen: Während diese Fälle bei der vorigen Messreihe keine weiteren Auswirkungen hatten, entstehen aus diesen Bitflips bei Aktivierung von Optimierungstechnik 1 meist erkennbare Fehlerfälle. Insgesamt führen mit Hilfe der Optimierung etwa 2,3 % der simulierten Fälle zu einer erfolgreichen Fehlererkennung mit Hilfe des BCET-Wertes. Die Erkennung auf Basis logischer Checks nimmt demgegenüber um etwa 0,5 % ab, weil es häufig bereits zu einer erfolgreichen Fehlererkennung mit Hilfe der BCET-Abschätzung kommt, ehe eine Prüfung auf korrekte Abfolge der IDs stattfindet. Der Anteil erkannter Fehler mit Hilfe der temporalen Prüfeinheit bleibt etwa gleich. Ebenso verändert sich die Zahl der prozessorseitigen Ausnahmebehandlungen kaum und auch der Anteil an Simulationen mit unerkannt fehlerhaften Ergebnissen oder Endlosschleifen ist etwa auf gleichem Niveau bei 15,9 % bzw. 0,8 %.

Wie bereits in Abschnitt 5.2.2 erläutert, wird das Potenzial der Fehlererkennung erst bei einer Betrachtung derjenigen Fälle sichtbar, in welchen der Bitflip fehlerhaftes Verhalten auslöst, welches nicht durch bereits integrierte Ausnahmebehandlungen erkennbar ist. Abbildung 6.4(b) zeigt hierzu wiederum die Verteilung ohne Optimierungen, die Verteilung bei veränderter Bitmaske (Ohne Opt.*) sowie die Verteilung bei aktivierter Optimierungstechnik 1: Zunächst ist zu beobachten, dass die Gesamtzahl an erkannten Fehlerfällen durch die veränderte Bitmaske von 64,5 % auf etwa 63,8 % sinkt, jedoch bei Aktivierung der Optimierungstechnik auf 65,3 % ansteigt. Dabei wird ein Anteil von 41,1 % durch die logische und ein Anteil von 24,2 % durch die temporale Check-Einheit erkannt – davon wiederum 19,4 % auf Basis der WCET- und 4,8 % mit Hilfe der BCET-Abschätzung. Die Zahl unerkannter Fehlerfälle sinkt leicht von ursprünglich 35,5 % auf etwa 34,8 %, wobei der Anteil an Endlosschleifen mit 1,6 % nahezu identisch bleibt. Diejenigen Durchläufe, welche keine Erkennung zur Laufzeit auslösen, jedoch mit falschen Ergebnissen enden, verringern sich durch den Einsatz von Optimierungstechnik 1 von 33,9 % auf 33,2 %.

Als Fazit zur Abdeckung erkannter Fehlerfälle lässt sich zusammenfassen, dass die vorgestellte Optimierungstechnik in der Summe einen leichten Anstieg erfolgreicher Erkennungsfälle bewirkt. Der Anteil an Simulationen mit fehlerhaftem Endergebnis sowie Endlosschleifen wird im Gegenzug etwas verringert. Durch die modifizierte Bitmaske, welche nötig ist, um den Aufwand gering zu halten, und die damit verbundenen Skalierung der instrumentierten WCET-Werte sinkt jedoch auch der Anteil der Erkennungsfälle auf Basis der maximal erlaubten Ausführungszeit.

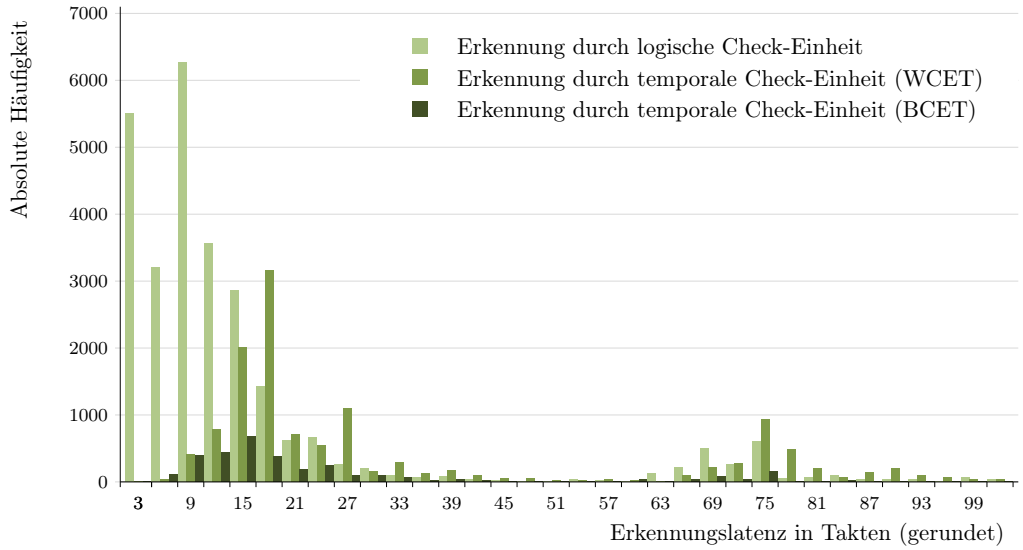


Abbildung 6.5: Häufigkeitsverteilung der Erkennungslatenzen bei Berücksichtigung einer zeitlichen Untergrenze der Ausführung

Latenz der Fehlererkennung

Durch die Integration der vorgestellten Optimierungstechnik ergeben sich Veränderungen bei der Latenz der Fehlererkennung. Abbildung 6.5 veranschaulicht hierzu die Häufigkeitsverteilung: Im Vergleich zu Abbildung 5.6 sind kaum Unterschiede bei der logischen Erkennungsmethode festzustellen. Die temporale Fehlererkennung auf Basis der WCET-Werte weist jedoch – verursacht durch die ungenauere Angabe in der modifizierten Bitmaske – ein etwas nach rechts verschobenes Maximum bei ca. 20 Takten auf. In der Abbildung ist zusätzlich die Verteilung der Erkennungslatenzen auf Basis der BCET-Angaben ersichtlich: Die meisten Messwerte liegen hier im Bereich zwischen 10 und 30 Takten Latenz, wobei das Maximum bei etwa 15 Takten zu sehen ist. Ähnlich zu den anderen Methoden finden sich auch bei der BCET-basierten Fehlererkennung einige Latenzen zwischen ca. 65 und 75 Takten, was wiederum auf die CarCore-spezifische Call- und Return-Behandlung zurückzuführen ist.

Im Durchschnitt über alle ausgeführten Simulationen ergibt sich eine Erkennungslatenz von 4062 Takten. Beschränkt man die Betrachtung aber analog zu Abschnitt 5.3 auf Latenzen unter 100 Takten, welche ca. 95 % der gemessenen Fälle ausmachen, so ergibt sich ein Mittelwert von 22,1 Takten. Gegenüber der Evaluierung

ohne Optimierung ist dieser Wert um 0,7 erhöht. Dieser Anstieg geht dabei vor allem auf die Erkennungsfälle auf Basis der WCET-Abschätzung zurück, bei welchen sich der Durchschnittswert durch die veränderte Bitmaske von 30,3 auf etwa 34,1 Takte verlängert hat. Die Erkennungslatenz der Fehlerfälle, welche durch eine falsche Abfolge der IDs feststellbar sind, ist nur um etwa 0,1 Takte angestiegen. Neu ist der Blick auf Fehler, die durch eine Unterschreitung der BCET-Werte erkannt werden: Diese haben eine durchschnittliche Latenz von etwa 120 Takten; bei einer Beschränkung auf Fälle unter 100 Takte liegt der Mittelwert bei 25,4 Takten.

Aufwandsanalyse

Da sich bei der vorgestellten Optimierungstechnik nur die Bitmaske, jedoch nicht die Anzahl und Länge der Checkpoints verändert, gibt es hinsichtlich Speicherbedarf und Ausführungszeit keinerlei Veränderungen zu den Aufwandsabschätzungen in Abschnitt 5.4. Im Rahmen der Code-Instrumentierung fällt natürlich eine BCET-Analyse der Applikationen als zusätzlich benötigter Schritt an. Dies ist bei Verwendung des integrierten *BCET Calculator* von geringer Bedeutung; soll jedoch eine vollständige Timing-Analyse der Programme mit Hilfe externer Tools vorgenommen werden, ist dieser Zusatzaufwand nicht zu vernachlässigen.

6.1.4 Fazit

Ziel der vorgestellten Optimierungstechnik ist es, die Fehlererkennung durch Prüfung einer zeitlichen Untergrenze der Ausführung zu verbessern. Die Evaluierungen zeigen jedoch, dass die Abdeckung erkannter Fehlerfälle dadurch nur minimal zunimmt. Im Gegenzug verringern sich hauptsächlich diejenigen Fälle, welche bei der ursprünglichen Evaluierung ohnehin unbedenklich waren. Die kritischen Fehlerfälle, d.h. der Anteil an Simulationen mit unerkannt fehlerhaftem Ergebnis oder Endlosschleifen bleibt nahezu gleich hoch. Ungünstig ist auch die Notwendigkeit, für die Speicherung der BCET-Werte die Bitmaske der Checkpoints zu verändern. Dies hat zur Folge, dass die Fehlererkennung mit Hilfe der WCET-Werte an Genauigkeit verliert.

Abschließend bleibt zu wiederholen, dass eine BCET-Analyse in der Praxis mit hohem Aufwand verbunden ist. Der Mehrwert im Hinblick auf die Fehlererkennung wird diesen gemäß den hier durchgeführten Evaluierungen wohl nicht rechtfertigen.

6.2 Reduktion der Checkpoint-Informationen

Der vorgestellte Mechanismus zur Erkennung von Kontrollflussfehlern in Echtzeitsystemen vergibt für jeden Grundblock einer Applikation eine eindeutige ID. Diese wird gemeinsam mit der Vorhersage des Nachfolgers und der zugehörigen WCET in einem Checkpoint gespeichert. Problematisch wird dieser Ansatz bei der Instrumentierung größerer Applikationen: Bei einer hohen Anzahl von Grundblöcken wird es nötig, die Bitbreite der Checkpoints zu vergrößern, um ausreichend viele unterschiedliche IDs darstellen zu können. Dies führt zu einem enormen Anstieg des Aufwands, sowohl was den Speicherbedarf als auch was die Ausführungszeit des Programmes betrifft.

Die Idee der vorliegenden Optimierungstechnik (nachfolgend auch *Optimierungstechnik 2* genannt) ist nun, IDs für Grundblöcke mehrfach zu vergeben. Dies ermöglicht zum einen die Instrumentierung umfangreicher Applikationen ohne weitere Einschränkungen. Zum anderen könnte die bei der bisherigen Implementierung eingesetzte Bitbreite verringert werden. Interessant ist dabei zu beobachten, inwiefern sich das Verhältnis zwischen Aufwand und Qualität der Fehlererkennung verändert. Zudem sollen Ideen untersucht werden, um die in den Checkpoints angegebenen WCET-Werte kompakter, d.h. mit weniger verwendeten Bits, darzustellen.

6.2.1 Vorgehensweise

Konkret bekommt nun jeder Grundblock statt einer ID eine *Kennzahl* der Bitlänge n zugewiesen. Ziel ist zwar, dass sich die Kennzahlen der Grundblöcke nach Möglichkeit unterscheiden; ein mehrfaches Auftreten ist jedoch erlaubt. Analog zur bisherigen Vorhersage einer nachfolgenden ID wird nun die Kennzahl des folgenden Grundblocks im Checkpoint gespeichert. Vorteil dieser Methode ist, dass die Länge n je nach Bedarf beliebig gewählt werden kann, da keine Abhängigkeit von der Gesamtzahl der Grundblöcke einer Applikation besteht. Nachteilig ist im Gegenzug, dass der Kontrollfluss bei einer fehlerhaften Ausführung zufällig in einen Block springen könnte, der dieselbe Kennung wie der zu erwartende besitzt. Der Fehlerfall wäre damit nicht erkennbar. Durch eine geschickte Vergabe der Kennzahlen im Rahmen der Instrumentierung, kann diese Wahrscheinlichkeit allerdings verringert werden.

An der Funktionsweise der Instrumentierung ändert sich im Prinzip nur wenig: Es ist nötig, eine Funktion zu integrieren, welche jeder ID auf eindeutige Weise eine Kennzahl zuordnet. Wird diese Funktion auf sämtliche IDs und Nachfolge-IDs

Typ	Kennzahl	Nachfolge-Kennzahl	WCET
2 Bit	x Bit	x Bit	$(14 - 2 * x)$ Bit

Abbildung 6.6: Bitmaske bei Reduktion der Checkpoint-Informationen (Opt. 2)

angewandt, so bleiben die Checkpoints in sich konsistent. Eine grundsätzliche Änderung der Hardware-Check-Einheit ist nicht nötig; es muss lediglich die Bitmaske der Checkpoints bekannt sein.

Auch die Angabe der WCET-Werte innerhalb der Checkpoints kann auf aufwandsreduzierte Weise erfolgen. Wird der Wert skaliert im Checkpoint gespeichert, sind zur Darstellung weniger Bits nötig. Die Wahl der Skalierungsfunktion hängt somit von der Anzahl an Bits ab, welche im Checkpoint für die Speicherung der WCET-Abschätzung zur Verfügung steht. In der Check-Einheit muss die Skalierung zur Laufzeit bekannt sein, um die Werte korrekt verarbeiten zu können. Ein Nachteil dieser skalierten Angabe der WCET-Werte ist jedoch, dass der temporale Erkennungsmechanismus an Genauigkeit verliert, zumal sich die Latenz der Fehlererkennung durch Informationsverluste bei der Skalierung erhöhen kann.

6.2.2 Integration in die Implementierung

Die Integration dieser Optimierungstechnik in die bestehende Implementierung aus Kapitel 4 ist recht einfach möglich. Unter dem Hauptziel, den Aufwand zu verringern, wird die Größe der Checkpoints pauschal auf 16 Bit reduziert. Damit ergibt sich die Bitmaske aus Abbildung 6.6: Die Kennzahl der Grundblöcke sowie die nachfolgende Kennzahl werden mit x Bit dargestellt; für die Angabe der WCET-Grenze verbleiben damit die restlichen Bit. Feste Werte werden noch nicht gewählt, da im Rahmen der Evaluierungen verschiedene Konstellationen analysiert werden sollen. Abhängig von der Bitbreite der WCET-Abschätzung kommen dabei auch verschiedene Skalierungsfunktionen zum Einsatz.

Das Instrumentierungstool wird um eine kleine Komponente, den *ID Converter*, ergänzt (vgl. Abbildung 6.7). Dessen Aufgabe ist es, vor der Generierung der Checkpoints mit Hilfe einer Konvertierungsfunktion die IDs und Nachfolge-IDs sämtlicher Grundblöcke in die entsprechenden Kennzahlen mit reduzierter Bitbreite umzuwandeln. Mit der Absicht, eine weite Verteilung identischer Kennzahlen zu erreichen und den Einsatz von expliziten und impliziten Nachfolgern zu bewahren, wird in der eingesetzten Implementierung die Modulo-Funktion $f(i) = i \; mod \; n$ auf jede ID i angewandt, wobei n die zur Darstellung der Kennzahlen verfügbare Bitbreite ist.

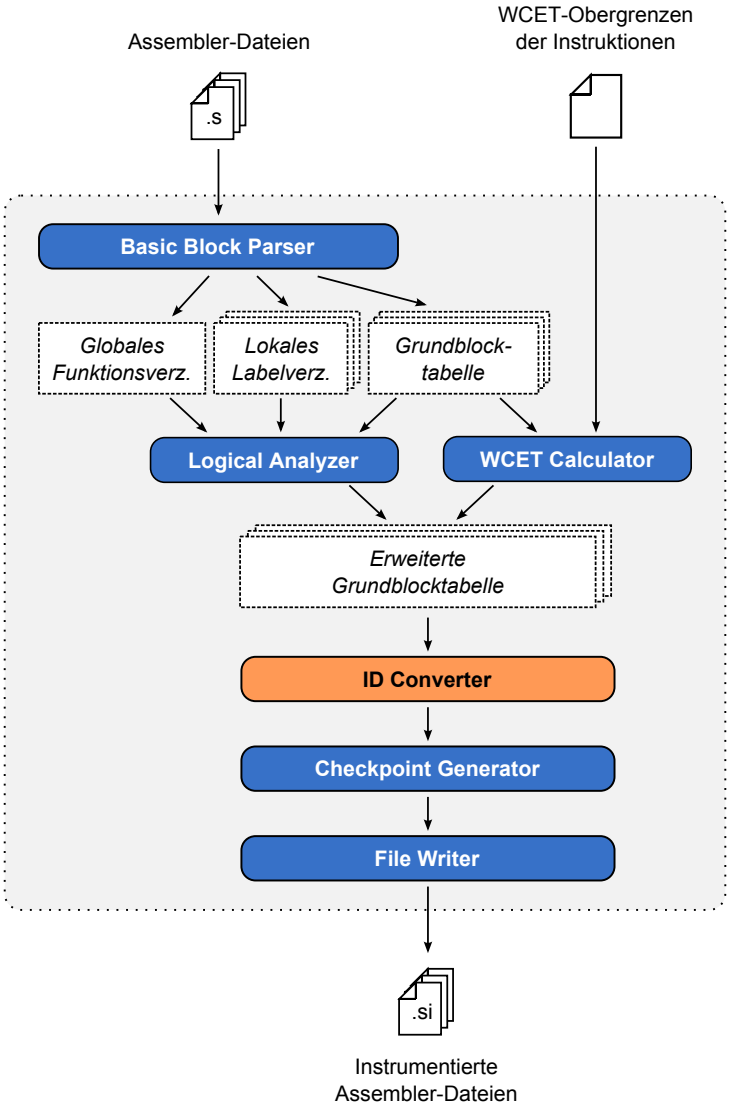


Abbildung 6.7: Ablauf der Instrumentierung bei Reduktion der Checkpoint-Informationen

Auch in der hardwareseitigen Check-Einheit sind geringfügige Veränderungen nötig. Die Bitmaske der Checkpoints muss bekannt sein, um die relevanten Daten korrekt auslesen zu können. Zudem muss das System der impliziten Nachfolger angepasst werden: Bisher ist ja in bedingten Sprüngen und Funktionsaufrufen der zweite mögliche Nachfolger stets die inkrementierte eigene ID. Bei der Umstellung auf Kennzahlen, die mit Hilfe der Modulo-Funktion erzeugt werden, ist dies weiterhin möglich – mit dem Zusatz, dass für die maximale Kennzahl $2^n - 1$ die niederwertigste Kennzahl 0 als impliziter Nachfolger gilt.

6.2.3 Evaluierung

Um die Stärken und Schwächen bewerten zu können, soll die vorgestellte Optimierungstechnik in verschiedenen Varianten evaluiert werden. Die Kriterien entsprechen dabei wiederum Kapitel 5: Mit Hilfe einer Injektion künstlicher Fehlerfälle wird die Abdeckung erkannter Fehler sowie die Erkennungslatenz ermittelt; anschließend gilt es, den Aufwand der Technik abzuschätzen.

Maßgeblichen Einfluss auf die Evaluierungsergebnisse hat in diesem Zusammenhang die verwendete Bitmaske der Checkpoints. Besonders interessant ist es, hierbei zu betrachten, inwiefern sich die Bitbreite der Kennzahlen sowie die Skalierung der WCET-Angabe im Checkpoint bei den Simulationsläufen bemerkbar machen. Konkret sollen nun sechs verschiedene Varianten mit 16 Bit breiten Checkpoints evaluiert werden; die jeweiligen Bitmasken sind in Abbildung 6.8 dargestellt.

Bei Optimierungsvariante 2a wird die Angabe der Kennzahl eines Grundblocks auf lediglich 2 Bit reduziert. Somit ergeben sich vier unterschiedliche Werte, welche über die Grundblöcke eines Benchmarks verteilt werden. Vorteil dieser Variante ist, dass für die Angabe der WCET-Abschätzung weiterhin volle 10 Bit zur Verfügung stehen. Bei den verwendeten Benchmarks ist dadurch keine Skalierung nötig. Etwas anders verhält es sich bei den Varianten 2b und 2c: Durch die Integration einer 3 Bit / 4 Bit breiten Kennzahl verbleiben nur noch 8 bzw. 6 Bit für die Angabe der WCET. Damit wird eine Skalierung um den Faktor $1/2$ bzw. $1/8$ notwendig. In der hardwareseitigen Check-Einheit kann ein solcher skalierter Wert sehr einfach mit Hilfe einer bitweisen Verschiebung zum ursprünglichen WCET-Wert zurücktransformiert werden.

Nun stellt sich die Frage, ob eine lineare Skalierung der WCET-Werte ideal ist, zumal sehr häufig kleine Grundblöcke mit kurzen WCET-Zeiten auftreten. Es wäre eine Variante zu bevorzugen, welche kleinere Werte mit höherer Genauigkeit abbildet. Optimierungstechnik 2d verwendet daher die Wurzelfunktion bei der Angabe des WCET-Wertes. Um die verfügbaren 6 Bit im Hinblick auf die evaluierten

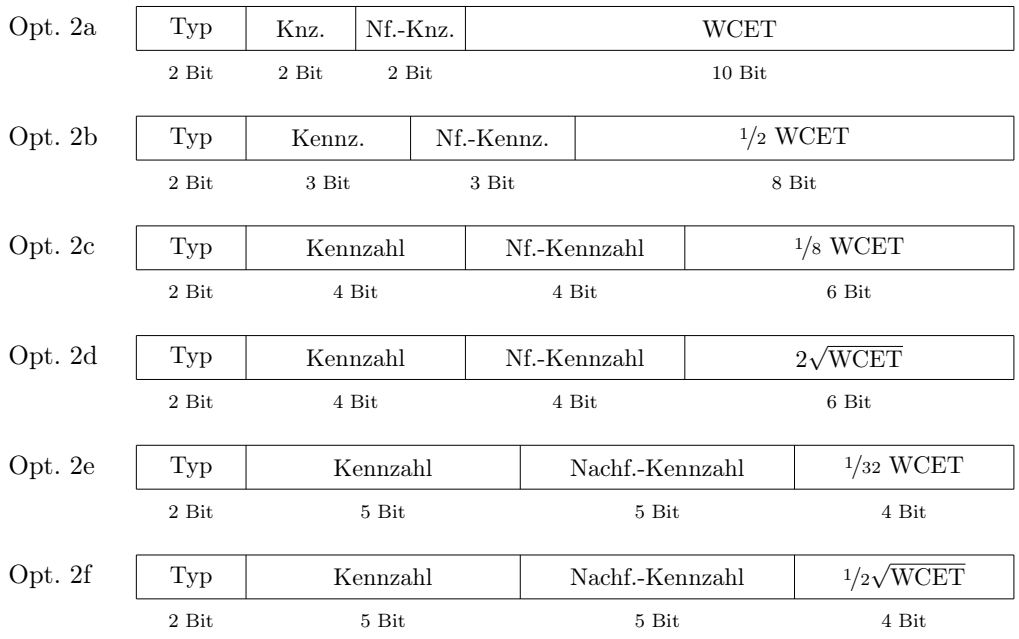


Abbildung 6.8: Bitmasken zur Evaluierung verschiedener Varianten bei Reduktion der Checkpoint-Informationen auf 16 Bit

Benchmarks mit höherer Genauigkeit zu nutzen, wird zusätzlich der Faktor 2 eingeführt: Anstelle des expliziten WCET-Wertes wird nun der Wert $2\sqrt{\text{WCET}}$ im Checkpoint gespeichert. Nachteilig ist, dass auf Hardware-Seite höherer Aufwand bei der Berechnung des Quadrats nötig wird. Dies könnte bei einer Synthese eine Umsetzung mit mehrtaktiger Multiplikation erfordern, was wiederum zu Verzögerungen bei der Fehlererkennung führen würde. Im Rahmen der Evaluierung sei jedoch weiterhin nur ein Takt zur Bereitstellung des Ergebnisses angenommen.

Die Bitmasken der Optimierungsvarianten 2e und 2f stellen für die Angabe der Kennzahl und Nachfolge-Kennzahl jeweils 5 Bit bereit. In den verbleibenden 4 Bit zur Speicherung der WCET-Werte soll nun einerseits eine lineare Skalierung um den Faktor $1/32$ zum Einsatz kommen (Variante 2e), andererseits eine Transformation zu $1/2\sqrt{\text{WCET}}$ (2f).

Abdeckung erkannter Fehlerfälle

Auch wenn es das primäre Optimierungsziel ist, den Aufwand der Fehlererkennung zu verringern, gilt es zunächst, die Wirksamkeit der Technik im Hinblick

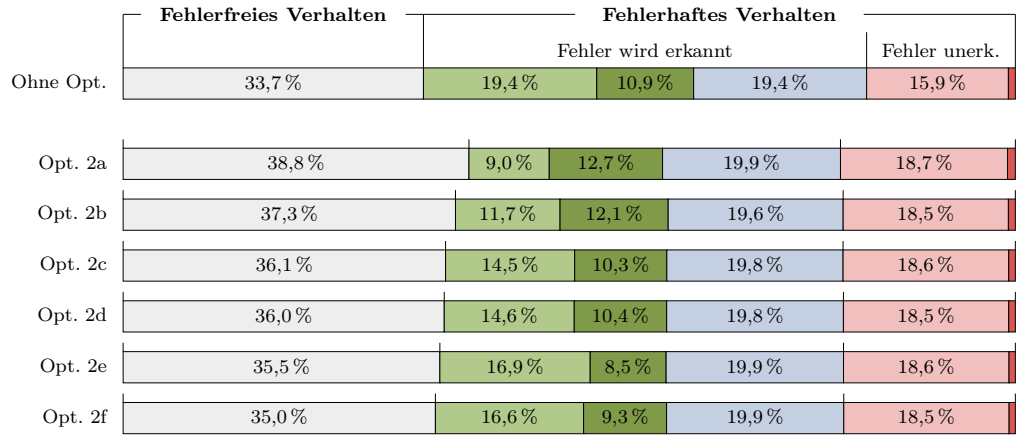
auf die Abdeckung erkannter Fehlerfälle zu untersuchen. Analog zu den Beschreibungen in Kapitel 5 sollen systematisch künstliche Fehler erzeugt und deren Folgen beobachtet werden. Da die Anzahl der Instruktionen in den ausgeführten Benchmark-Programmen durch die verkürzte Darstellung der Checkpoints kleiner wird, kann folglich nur eine geringere Zahl unterschiedlicher Bitflips im Instruktionsspeicher der Hauptfunktionen simuliert werden: Anstelle von 143 689 Simulationsläufen werden im Folgenden 123 676 Simulationen pro Optimierungsvariante ausgeführt.

Abbildung 6.9(a) zeigt das Verhalten der unterschiedlichen Varianten: Es fällt auf, dass der Anteil an fehlerfreien Simulationen bei Optimierungstechnik 2a am größten ist und bei allen weiteren Varianten kontinuierlich abnimmt. Dies ist maßgeblich die Konsequenz aus den jeweils eingesetzten Bitmasken: Bei Variante 1 stehen 10 Bit zur Darstellung der WCET-Abschätzung zur Verfügung. Ein Bitflip innerhalb dieses Bereiches führt häufig zu einer Erhöhung der WCET, was im Folgenden keinerlei Fehler bei der Ausführung bewirkt. Auch eine leichte Erniedrigung dieses Werts durch einen Bitflip bleibt unerkannt, solange die reale Ausführungszeit noch darüber liegt. Im Gegensatz dazu bewirken Bitflips innerhalb der ID-Angaben der Checkpoints stets eine Fehlererkennung; je breiter die Darstellung der IDs in den Bitmasken also ist, desto seltener tritt fehlerfreies Verhalten bei Bitflips innerhalb der Checkpoints auf.

Betrachtet man die Anteile der logischen und temporalen Fehlererkennung, so sind ebenfalls die Konsequenzen der eingesetzten Bitmasken ersichtlich. Da bei Optimierungstechnik 2a lediglich 2 Bit für die Repräsentation der Kennzahl verwendet werden, bleiben viele Fehlerfälle durch ein zufälliges Springen in einen falschen Block mit identischer Kennzahl unentdeckt. Es führen lediglich 9,0 % der fehlerhaften Simulationsläufe zu einer Erkennung mit Hilfe des logischen Check-Mechanismus. Im Gegenzug vergrößert sich der Anteil der temporalen Erkennungsfälle auf 12,7 %: Dies ist vor allem dadurch zu erklären, dass nun Fehler, welche nicht mehr über eine falsche Kennzahl identifizierbar sind (was in der Regel mit kürzeren Latenzzeiten verbunden ist), nun eine Überschreitung der angegebenen Maximallaufzeit verursachen. Insgesamt liegt die Anzahl erkannter Fehlerfälle bei Variante 2a bei lediglich 21,7 %, während der Anteil korrekter Durchläufe auf 38,8 % und der Anteil fehlerhafter Durchläufe, die zur Laufzeit als solche unerkannt bleiben, auf 18,7 % steigt. Bei einer Betrachtung der Optimierungsvarianten 2b, 2c und 2e ist zu sehen, dass sich bei einer Kennzahl mit 3, 4 oder 5 Bit der Anteil deutlich zugunsten der logischen Erkennungsfälle verschiebt. Im Gegenzug nimmt durch die stärkere Skalierung und damit ungenauere Angabe der WCET-Werte das Potenzial der temporalen Fehlererkennung immer weiter ab.

In der Summe wächst der Anteil erfolgreich erkannter Fehlerfälle auf 23,8 % bei

(a) Zusammenfassung aller Simulationsergebnisse:



(b) Zusammenfassung der Simulationsergebnisse mit fehlerhaftem Verhalten, das nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist:

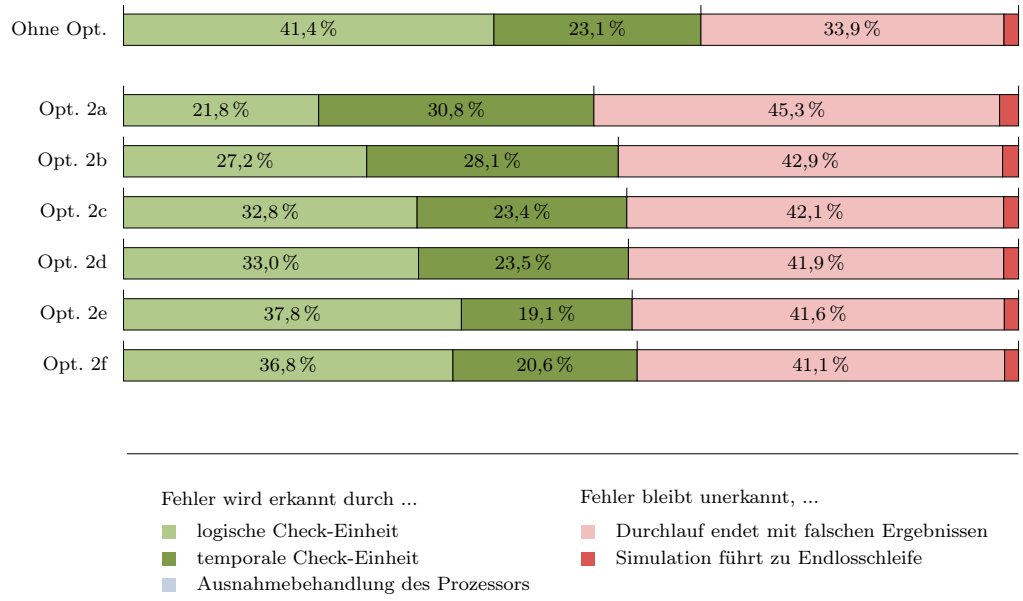


Abbildung 6.9: Abdeckung erkannter Fehlerfälle bei Reduktion der Checkpoint-Informationen auf 16 Bit (unter Verwendung der unterschiedlichen Bitmasken aus Abb. 6.8)

Variante 2b, auf 24,8 % bei 2c und auf 25,4 % bei 2e. Insgesamt reduzieren sich damit hauptsächlich diejenigen Simulationsläufe, in welchen das Ergebnis einem korrekten Durchlauf entspricht. Der Anteil an prozessorseitigen Ausnahmebehandlungen bleibt bei allen evaluierten Varianten auf einem gleichen Niveau von etwa 19,8 %. Auch der Anteil unerkannter Fehler unterscheidet sich nur minimal: Bei etwa 18,6 % der Fälle endet der Durchlauf mit falschen Ergebnissen, während es bei etwa 0,8 % zu einer Endlosschleife kommt.

Zur Veranschaulichung der Abdeckung erkannter Fehlerfälle zeigt Abbildung 6.9(b) wiederum eine Zusammenfassung der Simulationsergebnisse mit Fehlverhalten, welches nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist. In der Summe werden bei Optimierungstechnik 2a 52,6 % der Fehler durch den integrierten Check-Mechanismus erkannt. Dieser Wert nimmt bei den weiteren Optimierungsvarianten stetig zu und erreicht ein Maximum von 57,4 % bei Variante 2f. Verglichen zu den Simulationen ohne Optimierung ist dieser Wert um 7,1 % verringert – allerdings zugunsten eines deutlich niedrigeren Aufwands, wie nachfolgend beschrieben. Der Anteil an unerkannten Fehlern, welche ein falsches Simulationsergebnis bewirken, liegt bei Variante 2a mit 45,3 % am höchsten und bei Variante 2f mit 41,1 % am niedrigsten. Die Anzahl an Endlosschleifen reduziert sich von 2,1 % bei Variante 2a auf etwa 1,6 % bei Variante 2f.

Betrachtet man nun speziell die Optimierungsvarianten 2d und 2f, welche eine nichtlineare Skalierung der WCET-Abschätzung beinhalten, so ist besonders bei Variante 2f eine leichte Erhöhung der Erkennungsrate gegenüber der linearen Skalierung in 2e zu verzeichnen. Im Rahmen der Evaluierungen zeigt Variante 2f somit insgesamt die beste Abdeckung erkannter Fehlerfälle, zum einen aufgrund des höchsten Anteils erkannter Fehlerfälle (57,4 %), zum anderen wegen einer niedrigen Zahl von Durchläufen mit unerkanntem Fehlverhalten (42,6 %).

Latenz der Fehlererkennung

Als zweiter Aspekt der Evaluierung soll die Latenz der Fehlererkennung betrachtet werden. Abbildung 6.10 zeigt die Durchschnittswerte der logischen und temporalen Fehlererkennung in Abhängigkeit von der jeweils eingesetzten Optimierungsvariante. Dabei wird wiederum nur die Mehrheit der Fälle mit Latenzen bis 100 Takte berücksichtigt (wie bereits in Abschnitt 5.3 erläutert).

Es ist zu sehen, dass die Erkennungslatenzen auf Basis der logischen Vergleiche bei allen Varianten etwa auf gleichem Niveau (zwischen ca. 15,8 und 17,5 Takten) bleiben. Auch der Unterschied zum Durchschnittswert der Evaluierungen ohne

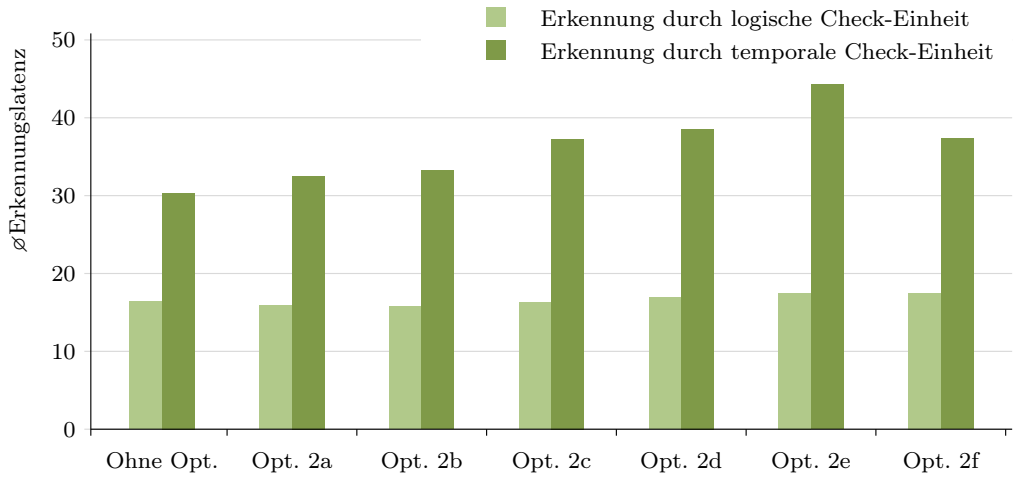


Abbildung 6.10: Durchschnittliche Latenzzeiten der Fehlererkennung bei Reduktion der Checkpoint-Informationen auf 16 Bit (unter Verwendung der unterschiedlichen Bitmasken aus Abb. 6.8)

Optimierung (16,4 Takte) ist sehr gering. Dies ist wenig überraschend, da das Erkennungsprinzip durch Veränderung der Bitbreite der Kennzahlen gleich bleibt. Sobald ein folgender Checkpoint erreicht wird, erfolgt eine Überprüfung des erlaubten Nachfolgers und ein Fehler kann sofort erkannt werden – unabhängig von der eingesetzten Bitbreite.

Im Hinblick auf die temporale Fehlererkennung ist ein Anstieg der durchschnittlichen Latenz von ca. 30,4 auf bis zu 44,3 Takte bei Variante 2e zu beobachten. Dies ist auf die skalierten WCET-Werte in den Checkpoints zurückzuführen, welche eine überschätzte Angabe der maximalen Laufzeit zur Folge haben. Eine Fehlererkennung findet erst statt, wenn diese Angabe überschritten wurde – in diesen Fällen entsprechend mit höheren Latenzzeiten. Bei Betrachtung der Werte in Abbildung 6.10 ist auch zu sehen, dass die Skalierung um den Faktor $1/2$ (Variante 2b) nur wenig Auswirkungen auf die Latenz verursacht (verglichen zur direkten Angabe der Werte in Variante 2a). Erst die Skalierung um die Faktoren $1/8$ (Variante 2c) und $1/32$ (Variante 2e) lässt die Latenzzeiten sehr deutlich ansteigen. Die Verwendung einer nichtlinearen Skalierung bringt bei Variante 2d nicht den gewünschten Erfolg: Entgegen der Erwartungen ist ein leichter Anstieg der Latenz zu verzeichnen. Anders verhält es sich hingegen bei Optimierungsvariante 2f: Die Transformation zu $1/2\sqrt{\text{WCET}}$ verkürzt die Latenzzeiten der temporalen Erkennungsfälle um etwa 7 Takte (verglichen zur Skalierung um Faktor $1/32$ in Variante 2e).

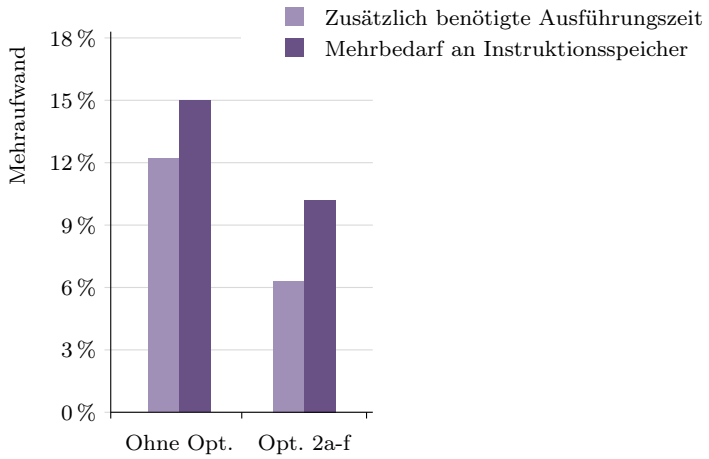


Abbildung 6.11: Mehraufwand bei Reduktion der Checkpoint-Informationen

Aufwandsanalyse

Eine zentrale Motivation der vorgestellten Optimierungstechnik ist es, den Aufwand der Fehlererkennung zu reduzieren. Da sämtliche Optimierungsvarianten 2a bis 2f Checkpoints in einer Breite von 16 Bit einsetzen, welche sich nur durch die Bitmasken unterscheiden, ist der Aufwand für all diese Varianten identisch: Wie Abbildung 6.11 zeigt, reduziert sich die zusätzlich benötigte Ausführungszeit bei Verwendung von 16 Bit breiten Checkpoints durchschnittlich von 12,2,% auf 6,3 %, der Mehrbedarf an Instruktionen von 15,0 % auf 10,2 %. Der Rückgang des Speicherbedarfs um etwa ein Drittel lässt sich dadurch erklären, dass nicht mehr drei, sondern nur noch zwei Instruktionen pro Checkpoint benötigt werden. In der ursprünglichen Version ohne Optimierungen können zwei der drei Instruktionen im CarCore-Prozessor parallel ausgeführt werden, sodass in der Regel zwei Takte Ausführungszeit pro Checkpoint nötig sind. Bei der vorgestellten Optimierungstechnik kann die zusätzliche Ausführungszeit nun um etwa die Hälfte reduziert werden, da die beiden verbleibenden Instruktionen, die einen Checkpoint repräsentieren, bei paralleler Ausführung nur noch einen Prozessortakt benötigen.

6.2.4 Fazit

Zusammenfassend lässt sich feststellen, dass die Reduktion der Checkpoint-Information eine durchaus nützliche Optimierungstechnik darstellt. Besonders unter

der Zielsetzung den Mehraufwand minimal zu halten, erweist sich eine mehrfache Vergabe von Checkpoint-IDs sowie eine Skalierung der angegebenen WCET-Werte als sinnvoll. Die Abdeckung erkannter Fehlerfälle nimmt erwartungsgemäß ab: Je nach eingesetzter Bitmaske sinkt der Gesamtanteil erkannter Fehlerfälle um 7,1 % - 11,9 %. Die Latenz der Fehlererkennung mit Hilfe der logischen Check-Einheit bleibt etwa gleich, während sich die Latenz der temporalen Erkennungsfälle durch die skalierte Angabe der WCET etwas erhöht. Der große Vorteil der Optimierungstechnik – und damit die Rechtfertigung für einen Einsatz unter bestimmten Rahmenbedingungen – ist die starke Verringerung des Aufwands: Der Speicherbedarf für zusätzlich eingefügte Instruktionen wird um etwa ein Drittel, die zusätzliche Ausführungszeit sogar um etwa die Hälfte reduziert.

Bei den evaluierten Varianten hat sich gezeigt, dass es im Hinblick auf den Anteil erkannter Fehlerfälle nützlicher ist, die Bitbreite der Kennzahl etwas höher zu wählen und die WCET-Werte dafür in stärkerem Maß zu skalieren. In dieser Hinsicht erzielt eine nichtlineare Skalierung die besten Ergebnisse. Allerdings bleibt abzuwägen, ob dies durch den entstehenden Hardware-Aufwand, d.h. die steigende Komplexität der Check-Einheit gerechtfertigt wird. Eine lineare Skalierung könnte hingegen sehr viel einfacher durch bitweises Verschieben realisiert werden.

6.3 Angleichung der Checkpoint-Abstände

Bei der vorgestellten Technik zur Erkennung von Fehlern im Kontrollfluss von Echtzeitsystemen sind die instrumentierten Checkpoints fest mit den jeweiligen Grundblöcken verbunden. Dies hat einerseits den Vorteil einer gut durchführbaren WCET-Analyse auf Grundblock-Ebene und ermöglicht andererseits eine einfache Handhabung der Vorhersagen im Kontrollfluss. Eine Schwachstelle ist jedoch, dass bei größeren Grundblöcken über längere Zeit kein Checkpoint durchlaufen wird, während hingegen bei sehr kleinen Grundblöcken innerhalb kürzester Zeit mehrere Checks stattfinden. Im ersten Fall könnte ein Fehler erst sehr spät erkannt werden, im zweiten Fall werden durch die häufige Ausführung der Checkpoints möglicherweise mehr Ressourcen verbraucht, als für die angestrebte Fehlererkennung nötig wären.

Eine Optimierungsidee im Hinblick auf diese Problematik besteht darin, Checkpoints nicht mehr starr mit jedem einzelnen Grundblock zu verbinden. Bei einer Abfolge von kurzen Grundblöcken könnte ein Prüfvorgang erst nach mehreren Blöcken ausreichen, während bei langen Blöcken wiederholte Checks nach einer bestimmten Anzahl ausgeführter Instruktionen stattfinden sollten. Kurz gefasst

ist es das Ziel, die Intervalle zur Prüfung auf eine ähnliche Größe zu bringen, um damit die Fehlererkennung zu verbessern und gleichzeitig den Aufwand zu reduzieren. Im Folgenden sollen die nötigen Rahmenbedingungen analysiert sowie eine darauf angepasste Implementierung vorgestellt werden. Abschließend gilt es, die Optimierungstechnik in verschiedenen Varianten hinsichtlich Abdeckung erkannter Fehlerfälle, Erkennungslatenz und Zusatzaufwand zu evaluieren.

6.3.1 Vorgehensweise

Diese Optimierungstechnik besteht prinzipiell aus zwei Teilen, einerseits der Aufspaltung und andererseits der Kombination von Grundblöcken. Als Optimierungsziel sei ein bestimmter Schwellenwert vorgegeben. Überschreitet die errechnete WCET-Abschätzung eines Grundblocks diesen Wert, so wird ein zusätzlicher Checkpoint in die Mitte des Blocks eingefügt. Diese Mitte soll dabei so definiert sein, dass die resultierenden Teilblöcke möglichst ähnliche WCET-Werte erhalten. Eine Anpassung der Checkpoints zur Prüfung der logischen Korrektheit des Kontrollflusses ist recht einfach machbar: Der in der Mitte des Grundblocks eingefügte Checkpoint erhält eine bisher ungenutzte ID; Checkpoint-Typ und Nachfolge-IDs werden – ebenso wie beim Checkpoint am Anfang des Blockes – angepasst. Gemäß der Definition von Grundblöcken kann es im Programm keinen Sprung geben, der den neuen Checkpoint in der Mitte des Blocks als Ziel hat. Eine Anpassung der logischen Nachfolger weiterer Checkpoints ist damit nicht nötig. Was den temporalen Check betrifft, muss lediglich eine WCET-Analyse für die beiden entstandenen Teilbereiche erfolgen, um diese Angaben zur Maximallaufzeit in die entsprechenden Checkpoints zu integrieren. Das Einfügen von Checkpoints soll solange rekursiv auf allen entstehenden Teilblöcken stattfinden, bis die WCET keines Checkpoints mehr den vorgegebenen Schwellenwert überschreitet.

Die Kombination von Grundblöcken, d.h. das Eliminieren von Checkpoints, erweist sich als etwas komplizierter, besonders im Hinblick auf die Konsistenz der angegebenen IDs und Nachfolge-IDs. Grundsätzlich soll versucht werden, einen Checkpoint zwischen zwei aufeinanderfolgenden Grundblöcken zu löschen, wenn die Summe der WCETs dieser Blöcke kleiner als ein vorgegebener Schwellenwert ist. Dies ist im Rahmen der vorliegenden Technik jedoch nur dann möglich, wenn jeder Grundblock im Kontrollfluss weiterhin maximal zwei Nachfolger hat. Ebenso ist es wichtig, das System der Checkpoints bei Funktionsaufrufen und Rücksprüngen durch das Eliminieren zu bewahren. Ob ein Checkpoint entfernt werden kann, hängt damit von verschiedenen Kriterien ab, welche im Kontext der Implementierung im folgenden Abschnitt einzeln diskutiert werden sollen.

6.3.2 Integration in die Implementierung

Um diese Optimierungstechnik in die Implementierung zu integrieren, ist keinerlei Veränderung der hardwareseitigen Check-Einheit nötig. Die Erweiterungen beschränken sich ausschließlich auf die Instrumentierung. Konkret wird das Tool um folgende zwei Komponenten ergänzt: Der *Block Splitter* teilt lange Grundblöcke auf und fügt zusätzliche Checkpoints in den Applikationscode ein, während der *Block Combiner* für das Zusammenfassen von Blöcken, also das Löschen von Checkpoints zuständig ist (vgl. Abbildung 6.12).

Der Block Splitter prüft zunächst für jeden Grundblock, ob die errechnete WCET-Abschätzung höher als der zur Optimierung angegebene Schwellenwert ist. Ist dies der Fall, wird die Position zum Einfügen des zusätzlichen Checkpoint ermittelt; idealerweise so, dass die WCET-Abschätzung des Blockes dadurch annähernd halbiert wird. Um eine logische Prüfung zu ermöglichen, welche konsistent zur bestehenden Implementierung mit einer expliziten und impliziten Angabe von Nachfolgern ist, erhält der eingefügte Checkpoint Y die inkrementierte ID des Checkpoints X am Anfang des geteilten Grundblocks. Alle folgenden IDs müssen daraufhin entsprechend erhöht werden. Wichtig ist dabei auch eine Anpassung der in sämtlichen Checkpoints angegebenen Nachfolger. Für die Checkpoints X und Y sind zudem folgende Schritte nötig: Der Typ von X wird nach dem Einfügen auf den Wert *1* (Sequenz / unbedingter Sprung, vgl. Abschnitt 3.2.2) gesetzt, als expliziter und einzig erlaubter Nachfolger gilt die neue ID von Checkpoint Y. Im Anschluss werden bei Checkpoint Y die ehemaligen Typ- und Nachfolgerwerte von X eingetragen, um schließlich die WCET-Werte für X und Y neu zu ermitteln und entsprechend einzufügen. Somit ist nach rekursiver Anwendung des Block Splitters weiterhin eine uneingeschränkte zeitliche und temporale Prüfung des Kontrollflusses möglich.

Die Funktionsweise des Block Combiners ist prinzipiell ähnlich: Zunächst wird mit Hilfe der WCET-Abschätzung der Grundblöcke ermittelt, welche Checkpoints unter Berücksichtigung des angegebenen Schwellenwertes zu eliminieren wären. Ist ein entsprechender Kandidat gefunden, muss anhand der Konstellation der beteiligten Checkpoints entschieden werden, ob eine Optimierung durchführbar ist oder nicht. Abbildung 6.13 zeigt die folgende Situation symbolisch: Der Checkpoint von Grundblock C soll eliminiert werden. Grundblock B sei der Block, welcher sich im Programmcode direkt vorderhalb befindet; Block D sei im Anschluss. Außerdem gebe es einen oder mehrere Grundblöcke A, welche den Checkpoint von Block C als expliziten Nachfolger beinhalten. Abhängig von den Varianten des Kontrollflusses ergeben sich nun verschiedene Situationen, die im Folgenden näher erläutert werden sollen. Eine Zusammenfassung findet sich in Tabelle 6.1.

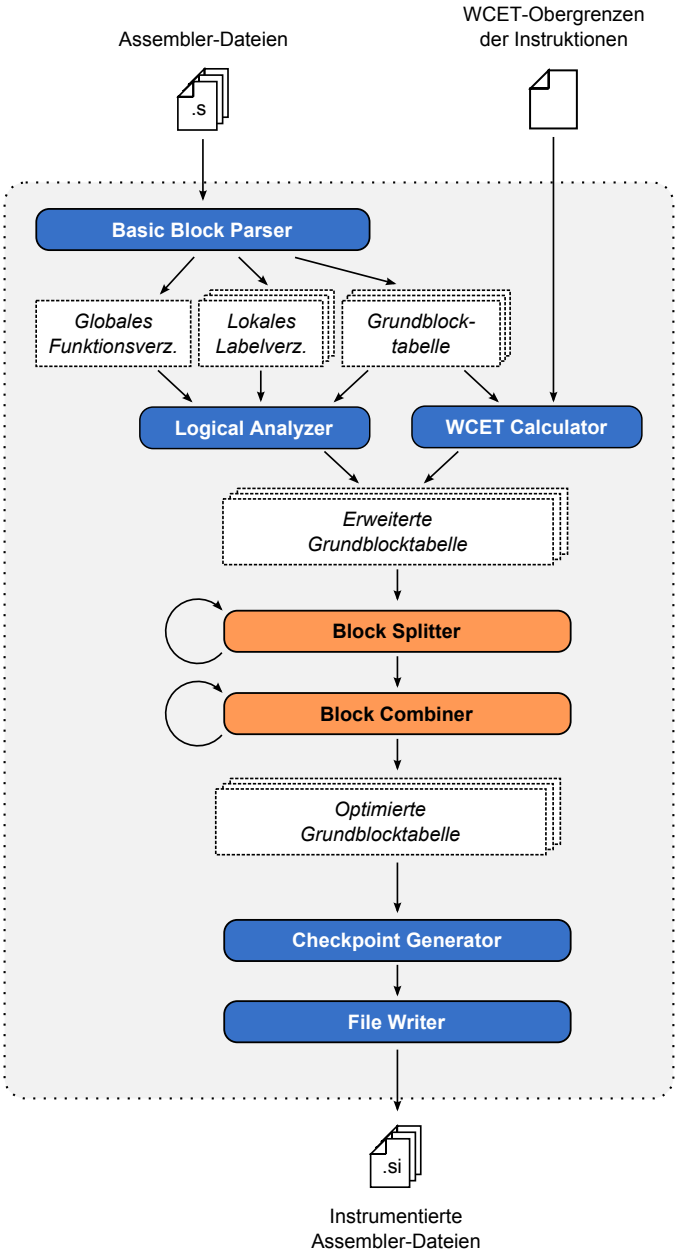


Abbildung 6.12: Ablauf der Instrumentierung bei Angleichung der Checkpoint-Abstände auf Basis eines angegebenen Schwellenwerts

C.Typ	***B.Typ***	**Maßnahmen zum Eliminieren**
Sequenz	Sequenz	$A.Suc \leftarrow C.Suc$; $B.Suc \leftarrow C.Suc$
	Sprung / Return	$A.Suc \leftarrow C.Suc$
	Bed. Sprung	$A.Suc \leftarrow C.Suc$; Bedingung:
	Call	IDs werden wieder inkrementell aufsteigend vergeben
Sprung	Sequenz	$A.Suc \leftarrow C.Suc$; $B.Suc \leftarrow C.Suc$; $B.Typ \leftarrow$ Sprung
	Sprung / Return	$A.Suc \leftarrow C.Suc$
	Bed. Sprung	Eliminieren **nicht möglich**, da bei A eine explizite Angabe
	Call	von zwei Sprungzielen nötig wäre
Bed. Sprung	Sequenz	wenn $\neg\exists A$ mit $A.Suc = C$: $\{B.Typ \leftarrow$ Bed. Sprung; $B.Suc \leftarrow C.Suc\}$, sonst Eliminieren **nicht möglich**, da bei A eine explizite Angabe von zwei Sprungzielen nötig wäre
	Sprung / Return	Eliminieren **nicht möglich**, da bei A bzw. B eine explizite Angabe von zwei Sprungzielen nötig wäre
	Bed. Sprung	
	Call	
Call	Sequenz	wenn $(\neg\exists A$ mit $A.Suc = C) \vee (\forall A$ mit $A.Suc = C$: $A.Typ =$ Sprung): $\{B.Typ \leftarrow$ Call; $B.Suc \leftarrow C.Suc$; $A.Typ \leftarrow$ Call; $A.Suc \leftarrow C.Suc\}$, sonst Eliminieren **nicht möglich**, da bei A eine explizite. Angabe von zwei Sprungzielen nötig wäre
	Sprung / Return	wenn $\forall A$ mit $A.Suc = C : A.Typ =$ Sprung: $\{A.Typ \leftarrow$ Call; $A.Suc \leftarrow C.Suc\}$, sonst El. **nicht möglich**, da bei A eine explizite Angabe von zwei Sprungzielen nötig wäre
	Bed. Sprung	Eliminieren **nicht möglich**, da bei B eine explizite Angabe von zwei Sprungzielen nötig wäre
	Call	Eliminieren **nicht möglich**, da es zu einer falschen Verschachtelung der Calls und Returns führen würde
Return	Sequenz	wenn $(\neg\exists A$ mit $A.Suc = C) \vee (\forall A$ mit $A.Suc = C$: $A.Typ =$ Sprung): $\{B.Typ \leftarrow$ Return; $A.Typ \leftarrow$ Return$\}$, sonst Eliminieren **nicht möglich**, da es zu einer falschen Verschachtelung der Calls und Returns führen würde
	Sprung / Return	wenn $\forall A$ mit $A.Suc = C : A.Typ =$ Sprung: $\{A.Typ \leftarrow$ Return$\}$, sonst Eliminieren **nicht möglich**, da es zu einer falschen Verschachtelung der Calls und Returns führen würde
	Bed. Sprung	Eliminieren **nicht möglich**, da es zu einer falschen Verschachtelung der Calls und Returns führen würde
	Call	
	alle	wenn $\forall A$ mit $A.Suc = C : A.Typ =$ Call: $\{A.Typ \leftarrow$ Sprung; $A.Suc = C.Suc\}$

Tabelle 6.1: Maßnahmen zum Eliminieren des Checkpoints bei Grundblock C (Abb. 6.13) in Abhängigkeit von den Checkpoint-Typen der Blöcke B und C (Erläuterung der Notation: $X.Typ$ bezeichnet den Typ des Checkpoints von Grundblock X, $X.Suc$ den im Checkpoint von Grundblock X angegebenen expliziten Nachfolger.)

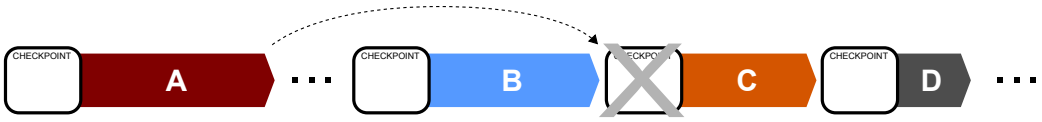


Abbildung 6.13: Beteiligte Grundblöcke beim Eliminieren eines Grundblocks C

Liegt bei Block C ein sequenzieller Ablauf vor, d.h. D ist einzig möglicher Nachfolger, so ist ein Eliminieren des Checkpoints möglich. Die zu ergreifenden Maßnahmen hängen in diesem Fall von Block B ab: Ist hier ebenfalls ein sequenzieller Ablauf zu sehen, so wird der Nachfolger von C zum (expliziten) Nachfolger von Block A und B. Liegt am Ende von B ein Sprung oder Funktionsrücksprung vor, genügt es, den bisherigen Nachfolger von C als Nachfolger von A einzutragen. Handelt es sich am Ende von B um einen bedingten Sprung oder Funktionsaufruf, so muss ebenso der Nachfolger von C als Nachfolger von A eingetragen werden. In diesen Fällen ist es außerdem zwingend nötig, die IDs der Checkpoints nach dem Löschen wieder inkrementell aufsteigend zu vergeben, damit D als impliziter Nachfolger gelten kann.

Falls im Grundblock des zu optimierenden Checkpoints ein (nicht bedingter) Sprung vorliegt, ist ein Eliminieren nur unter bestimmten Bedingungen möglich. Wird die Ausführung am Ende von B sequenziell weitergeführt, so erhalten A und B als expliziten Nachfolger den bisherigen Nachfolger von C; der Sprung findet nun im Anschluss an den Checkpoint B statt. Handelt es sich am Ende von B bereits vorher um einen Sprung oder Funktionsrücksprung, so wird – ähnlich wie oben – der Nachfolger von C als Nachfolger von A eingetragen. Liegt bei B allerdings ein bedingter Sprung oder ein Funktionsaufruf vor, so ist ein Eliminieren von Checkpoint C nicht möglich. Für diesen Fall müssten bei A zwei explizite Nachfolger angegeben werden, was dem Grundprinzip der Implementierung widerspricht.

Ähnlich verhält es sich, wenn sich am Ende von C ein bedingter Sprung befindet. Auch dann wäre es beim Eliminieren dieses Checkpoints stets nötig, bei A zwei explizite Nachfolger anzugeben. Einzige Ausnahme ist folgende: Der Ablauf am Ende des Blockes B ist sequenziell und es gibt in der gesamten Applikation keinen Grundblock A, der Block C als expliziten Nachfolger hat. In diesem Fall müsste Checkpoint B den Nachfolger von C übernehmen und als Checkpoint-Typ einen bedingten Sprung anzeigen.

Befindet sich am Ende von Block C ein Funktionsaufruf, so ist ein Löschen ebenfalls nur unter ganz bestimmten Bedingungen denkbar. Handelt es sich am Ende von Block B um einen sequenziellen Ablauf, so gilt: Nur falls es keinen Block A gibt, der C als expliziten Nachfolger hat, oder falls alle Blöcke A unbedingte

Sprünge mit Sprungziel C sind, ist eine Optimierung wie folgt möglich: Sowohl bei Block A als auch bei Block B muss der Checkpoint-Typ einen Funktionsaufruf signalisieren und als expliziter Nachfolger muss jeweils der bisherige explizite Nachfolger von C angegeben werden. Liegt bei B ein Sprung oder Funktionsrücksprung vor, kann Block C ebenfalls nur eliminiert werden, wenn alle Blöcke A, die C als Nachfolger haben, mit einem unbedingten Sprung enden. Der Checkpoint von A signalisiert in diesem Fall bereits den Funktionsaufruf und gibt als expliziten Nachfolger den Nachfolger von Block C an. Handelt es sich am Ende von C um einen Funktionsaufruf und am Ende von B um einen bedingten Sprung, ist ein Löschen des Checkpoints unmöglich, da wieder die explizite Angabe von zwei Sprungzielen nötig wäre. Falls am Ende von B ein weiterer Funktionsaufruf steht, würde die Optimierung zu einer falschen Verschachtelung der Calls und Returns führen, was ebenso nicht erlaubt sein darf.

Abschließend soll noch der Fall betrachtet werden, dass sich am Ende des zu eliminierenden Grundblocks ein Funktionsrücksprung befindet. Unabhängig von Block B gilt dabei: Sofern alle Grundblöcke A, deren expliziter Nachfolger C ist, mit einem Funktionsaufruf enden, kann der Checkpoint bei C gelöscht werden. Der Typ der Checkpoints bei allen Blöcken A muss dann einen unbedingten Sprung signalisieren und als Nachfolger den bisherigen Nachfolger von Block C angeben. Liegt bei Block B ein sequenzieller Ablauf vor, so ist folgende Konstellation zu prüfen: Falls es entweder keinen Block A gibt, der C als expliziten Nachfolger beinhaltet oder für alle diese Blöcke A gilt, dass am Ende ein nicht bedingter Sprung zu C erfolgt, kann der Checkpoint bei C optimiert werden. In dieser Situation erhalten die Checkpoints der Blöcke A und B die Aufgabe, den Rücksprung zu signalisieren. Handelt es sich bei Block B andernfalls um einen (unbedingten) Sprung oder ebenso um einen Funktionsrücksprung, so könnte der Checkpoint bei C nur dann eliminiert werden, wenn alle Blöcke A mit einem Sprung enden. Die Checkpoints in A müssten dann statt einem Sprung einen Funktionsrücksprung ankündigen. In sämtlichen anderen Konstellation ist ein Löschen des Checkpoints von C unmöglich, da es zu einer falschen Verschachtelung der Funktionsaufrufe und -rücksprünge führen würde, was so vom Hardware-Checker nicht mehr prüfbar wäre.

6.3.3 Evaluierung

Zur näheren Untersuchung der vorgestellten Optimierungstechnik (nachfolgend auch *Optimierungstechnik 3* genannt) sollen nun verschiedene Varianten simuliert und die jeweiligen Ergebnisse untersucht werden. Wie bereits erwähnt, hängt der Grad der Optimierung maßgeblich vom Schwellenwert der WCET-Abschätzung

ab: Ist dieser niedrig gewählt, wird der Block Splitter bei einer Vielzahl von längeren Blöcken zusätzliche Checkpoints einfügen. Im Gegenzug wird bei einem hohen Wert der Block Combiner häufig versuchen, Checkpoints zu eliminieren.

Im Rahmen dieser Evaluierungen sollen beide Komponenten, also zunächst Block Splitter und im Anschluss Block Combiner, stets mit identischem Schwellenwert auf die verschiedenen Benchmarks angewandt werden. Betrachtet werden fünf Varianten mit den Schwellenwerten 50, 75, 100, 125 und 150. Tabelle 6.2 fasst die Anzahl der Optimierungsoperationen zusammen: Zu jeder Variante ist dabei sowohl die Anzahl eingefügter und eliminierter Checkpoints als auch die daraus resultierende Gesamtzahl an Checkpoints eines Benchmarks zu sehen. Die oberste Zeile zeigt zum Vergleich die Anzahl an Checkpoints ohne Optimierungstechniken.

Erwartungsgemäß werden bei niedrigen Schwellenwerten deutlich mehr zusätzliche Checkpoints eingefügt als bei höheren Schwellenwerten. Dadurch ergibt sich für den Block Combiner die Ausgangssituation, dass der erste Teil eines bereits geteilten Grundblocks – der nun garantiert kürzer als der angegebene Schwellenwert ist – häufig mit dem jeweiligen Vorgängerblock kombiniert werden kann. Somit entsteht auch bei niedrigen Schwellenwerten eine hohe Zahl eliminierter Checkpoints, was zunächst widersprüchlich erscheint. Würde der Block Combiner ohne vorhergehenden Block Splitter eingesetzt werden, so könnte man natürlich erst mit Erhöhung des Schwellenwerts für alle Benchmarks einen Anstieg eliminierter Checkpoints beobachten.

Betrachtet man Optimierungsvariante 3a, so ist zu sehen, dass sich nach dem Einsatz von Block Splitter und Block Combiner bei fast allen eingesetzten Benchmark-Applikationen die Anzahl der Checkpoints erhöht. Nur bei Benchmark *bitmnp*, welcher aus sehr vielen kurzen Grundblöcken besteht, werden die Checkpoints deutlich reduziert. Bei den Varianten 3b-3e ist mit höherem Schwellenwert bei allen Benchmarks eine Verringerung der Checkpoints erkennbar.

Es stellt sich nun die Frage, wie sich die Optimierungen auf die Möglichkeiten der Fehlererkennung auswirken. Dazu sollen analog zu Abschnitt 5.1.2 Simulationen mit künstlichen Fehlern durchgeführt werden, um das Verhalten exakt zu analysieren. Da sich je nach Optimierungsvariante die Anzahl der eingefügten Checkpoints unterscheidet, variiert auch der genutzte Instruktionsspeicher. Unter der Zielsetzung, sämtliche möglichen Bitflips innerhalb der Hauptfunktionen zu injizieren, ergibt sich je nach Benchmark und Schwellenwert der Optimierung eine verschiedene Anzahl an Simulationsdurchläufen. Tabelle 6.3 gibt hierzu einen Überblick über die Anzahl: Während bei Optimierungsvariante 3a noch 142 974 mögliche Bitflips evaluiert werden können, sind es bei Variante 3e durch die deutlich niedrigere Anzahl an Instruktionen nur noch 122 644.

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Checkpoints	132	411	76	132	195	87	240
Optimierungsvariante 3a (Schwellenwert 50):							
Block Splitter	+80	+81	+48	+62	+74	+56	+225
Block Combiner	−64	−233	−39	−51	−72	−36	−133
Checkpoints	148	259	85	143	197	107	332
Optimierungsvariante 3b (Schwellenwert 75):							
Block Splitter	+17	+6	+8	+11	+12	+11	+74
Block Combiner	−30	−231	−16	−27	−48	−22	−61
Checkpoints	119	186	68	116	159	76	253
Optimierungsvariante 3c (Schwellenwert 100):							
Block Splitter	+2	+2	+2	+2	+6	+4	+25
Block Combiner	−25	−246	−12	−24	−47	−17	−40
Checkpoints	109	167	66	110	154	74	225
Optimierungsvariante 3d (Schwellenwert 125):							
Block Splitter	+0	+0	+0	+0	+4	+1	+9
Block Combiner	−24	−257	−11	−23	−47	−15	−36
Checkpoints	108	154	65	109	152	73	213
Optimierungsvariante 3e (Schwellenwert 150):							
Block Splitter	+0	+0	+0	+0	+2	+1	+7
Block Combiner	−24	−257	−11	−23	−46	−15	−35
Checkpoints	108	154	65	109	151	73	212

Tabelle 6.2: Anzahl der Optimierungsoperationen pro Benchmark bei Angleichung der Checkpoint-Abstände nach verschiedenen Schwellenwerten

Ohne Opt.	Opt. 3a	Opt. 3b	Opt. 3c	Opt. 3d	Opt. 3e
143 689	142 974	128 400	124 566	122 840	122 644

Tabelle 6.3: Anzahl injizierter Bitflips in Abhängigkeit von der Variante der Abstandsangleichung

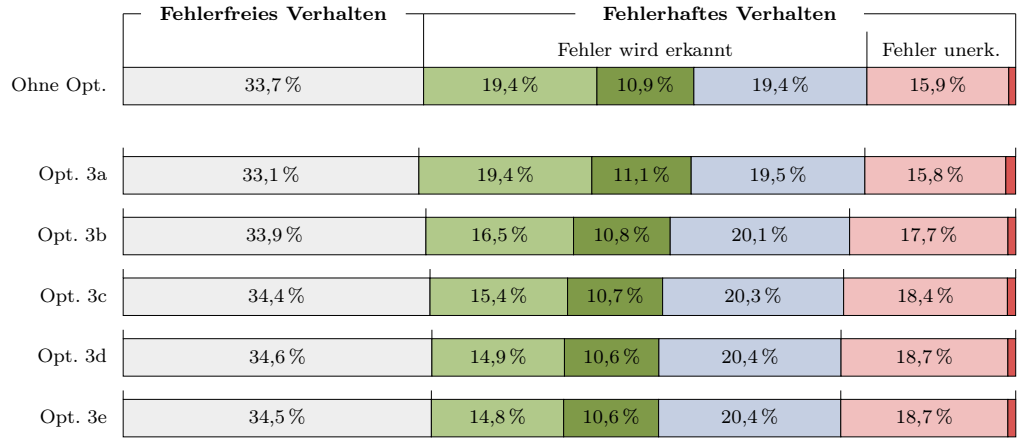
Abdeckung erkannter Fehlerfälle

Abbildung 6.14(a) zeigt die Ergebnisse aller Simulationsdurchläufe mit den injizierten Bitflips aus Tabelle 6.3: Zunächst ist zu sehen, dass sich der Anteil an Simulationen mit fehlerfreiem Verhalten bei Optimierungsvariante 3a auf 33,1 % verringert, während bei allen weiteren Varianten eine leichte Zunahme auf bis zu 34,5 % bei 3e zu verzeichnen ist. Die Anzahl erkannter Fehler durch die Check-Einheit nimmt bei Variante 3a im Vergleich zu den Simulationen ohne Optimierung um insgesamt etwa 0,2 % zu und erreicht einen Wert von 30,5 %. Mit steigendem Schwellenwert der Optimierungsvarianten und der damit reduzierten Anzahl an Checkpoints verringert sich der Anteil der erkannten Fehler, sodass bei den Varianten 3d und 3e nur noch etwa 25,5 % der Simulationen eine erfolgreiche Fehlererkennung auslösen. Vergleicht man Variante 3e direkt mit der Evaluierung ohne Optimierungen, so ist zu sehen, dass die Erkennungsrate auf Basis logischer Checks um beinahe 5 % zurückgeht, während die Erkennung mit Hilfe der WCET-Grenze fast keine Einbußen verzeichnet. Im Gegenzug wächst auch der Anteil an Erkennungsfällen durch prozessorseitige Ausnahmebehandlungen mit steigendem Schwellenwert um etwa 1 % an. Der Anteil unerkannter Fehlerfälle, welche ein falsches Simulationsergebnis verursachen, verringert sich bei Optimierungsvariante 3a etwas, um dann mit steigendem Schwellenwert von 15,8 % auf bis zu 18,7 % bei den Varianten 3d und 3e anzuwachsen. Die Zahl der Endlosschleifen ist allerdings bei Variante 3a mit 1,1 % etwas höher als bei allen anderen Varianten (ca. 0,9 %).

In Abbildung 6.14(b) ist eine Zusammenfassung der Simulationsergebnisse zu sehen, bei welchen fehlerhaftes Verhalten auftritt, das nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist. Bei Optimierungsvariante 3a ist demnach die Summe erkannter Fehler mit 64,4 % etwa genauso hoch wie bei den Simulationen ohne Optimierung (64,5 %). Bei allen weiteren evaluierten Optimierungsvarianten nimmt dieser Wert kontinuierlich ab und erreicht bei den Varianten 3d und 3e mit jeweils 56,5 % sein Minimum. Allerdings ist auch hier zu erkennen, dass die Erkennungsrate auf Basis temporaler Checks stets bei etwa 23,5 % bleibt, während sich der Anteil logischer Checks mit steigenden Schwellenwerten deutlich von 40,9 % auf 32,9 % verringert. Betrachtet man den Anteil unerkannt bleibender Fehlerfälle, so ist ein Anstieg von 35,6 % bei Variante 3a (davon 33,3 % mit fehlerhaften Ergebnissen, 2,3 % mit Endlosschleifen) auf bis zu 43,5 % bei Variante 3e (41,5 % mit fehlerhaften Ergebnissen, 2,0 % mit Endlosschleifen) zu sehen.

Zusammenfassend lässt sich feststellen, dass die Zielsetzung, die Abdeckung erkannter Fehlerfälle zu verbessern, mit keiner Variante der Optimierungstechnik erreicht wird. Selbst bei klein gewählten Schwellenwerten ist im Durchschnitt keine

(a) Zusammenfassung aller Simulationsergebnisse:



(b) Zusammenfassung der Simulationsergebnisse mit fehlerhaftem Verhalten, das nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist:

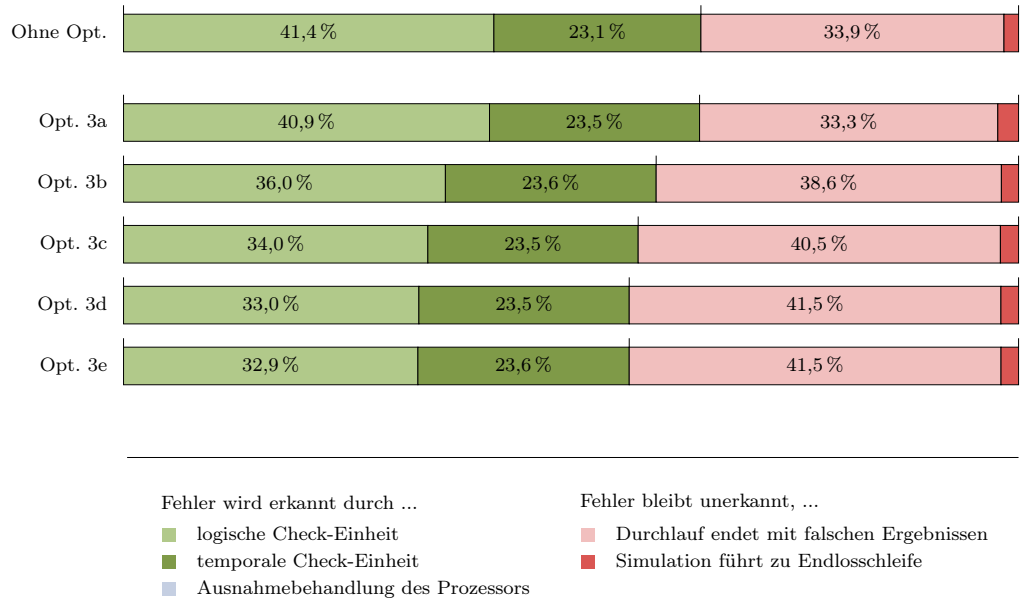


Abbildung 6.14: Abdeckung erkannter Fehlerfälle bei Angleichung der Checkpoint-Abstände (unter Verwendung der Schwellenwerte aus Tab. 6.2)

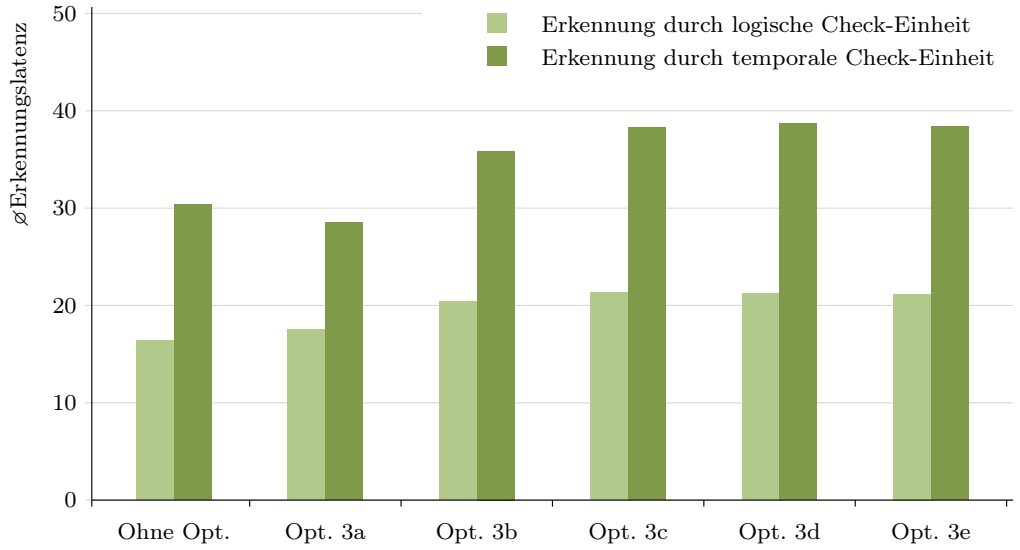


Abbildung 6.15: Durchschnittliche Latenzzeiten der Fehlererkennung bei Angleichung der Checkpoint-Abstände

Verbesserung gegenüber den Ausführungen ohne Optimierungen zu erkennen. In allen anderen Varianten werden weniger der injizierten Fehler erkannt. Interessant ist nun der Blick auf die anderen Aspekte der Evaluierung: Falls der Aufwand hinsichtlich Ausführungszeit und Speicherverbrauch merklich reduziert werden kann, ist möglicherweise ein Kompromiss auf Kosten einer geringeren Abdeckung von Fehlerfällen sinnvoll.

Latenz der Fehlererkennung

Als zweiter Aspekt der Evaluierung soll die Latenz der Fehlererkennung betrachtet werden. Abbildung 6.15 zeigt die Durchschnittswerte bei der logischen und temporalen Fehlererkennung in Abhängigkeit von den analysierten Optimierungsvarianten. Vernachlässigt werden auch hier diejenigen Einzelfälle der Fehlererkennung, welche mit einer Latenz von mehr als 100 Takten verbunden sind.

Verglichen zu den Evaluierungen ohne Optimierung zeigt sich bei Variante 3a zunächst ein leichter Rückgang der Latenz auf Basis der temporalen Fehlererkennung von 30,3 auf 28,5 Takte. Mit Erhöhung der Schwellenwerte ist ein Anstieg dieser Erkennungslatenzen auf bis zu 38,7 Takte bei Variante 3d zu sehen. Dies ist so zu

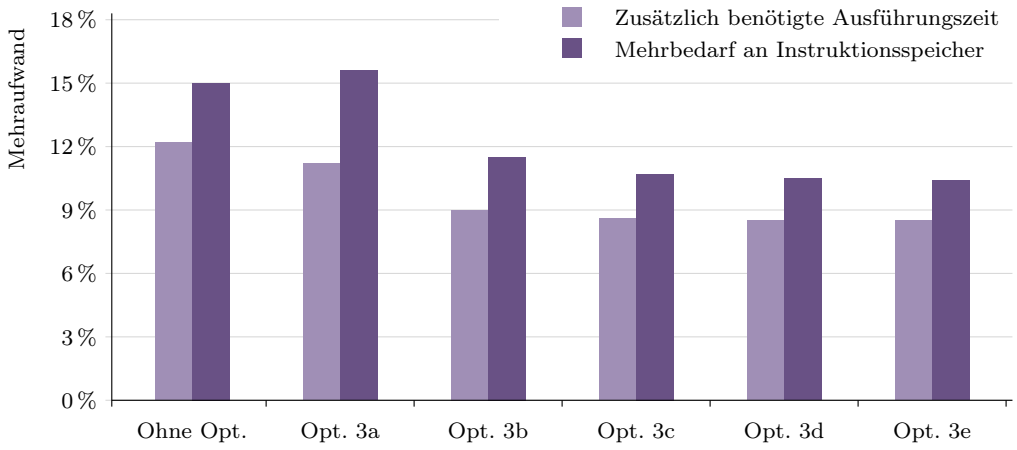


Abbildung 6.16: Mehraufwand bei Angleichung der Checkpoint-Abstände

erwarten, da sich die Abstände zwischen den einzelnen Checkpoints und damit die instrumentierten WCET-Werte vergrößern. Im Fehlerfall kann es somit deutlich länger dauern, bis eine Überschreitung der instrumentierten WCET-Abschätzung vorliegt.

Betrachtet man die logische Fehlererkennung auf Basis der instrumentierten IDs, so ist die durchschnittliche Latenz bei Optimierungsvariante 3a mit 17,6 Takten ein wenig höher als bei Durchläufen ohne Optimierungen (16,4 Takte). Bei allen weiteren Varianten 3b bis 3e liegt der Durchschnittswert etwa 3-4 Takte höher, maximal bei 21,3 Takten bei Variante 3c. Mit ansteigendem Schwellenwert ist hier kein deutlicher Anstieg mehr zu beobachten.

Aufwandsanalyse

Abschließend gilt es, den Aufwand der vorgestellten Optimierungen zu analysieren. Abbildung 6.16 veranschaulicht dazu sowohl die zusätzlich benötigte Ausführungszeit als auch den Mehrbedarf an Instruktionsspeicher.

Im Mittel über alle gemessenen Benchmarks ist bei Optimierungsvariante 3a zunächst ein Anstieg des Speicherbedarfs festzustellen. Durch den niedrig gewählten Schwellenwert erhöht sich die Anzahl eingefügter Checkpoints, was natürlich Speicher kostet. Gleichzeitig sinkt jedoch die Ausführungszeit im Vergleich zur Evaluierung ohne Optimierungen. Dies ist dadurch zu erklären, dass eine Vielzahl

sehr häufig ausgeführter kurzer Grundblöcke nun mit den jeweiligen Nachbarblöcken kombiniert ist und damit zur Laufzeit weniger Checkpoints durchlaufen werden müssen. Erweitert man den Vergleich auf Variante 3b, so ist hier ein deutlicher Rückgang des Aufwands zu verzeichnen: Die Ausführungszeit sinkt von etwa 11,2 % bei Variante 3a auf 9,0 %, während sich der Speicherbedarf von etwa 15,6 % auf 11,5 % verringert. Bei allen weiteren betrachteten Optimierungsvarianten sind kaum mehr Änderungen zu sehen. Die zusätzliche Ausführungszeit sinkt nur noch minimal auf Werte um 8,5 %, während sich der Aufwand hinsichtlich Speicherverbrauch bei etwa 10,5 % einpendelt.

6.3.4 Fazit

Motivation der vorgestellten Optimierungstechnik ist es, durch die Angleichung der Checkpoint-Abstände einerseits die Erkennungsmöglichkeiten zu verbessern und andererseits den damit verbundenen Aufwand zu reduzieren. Betrachtet man die Ergebnisse der Evaluierung, so ist diese Zielsetzung nur bedingt erfüllt: Weder die Abdeckung erkannter Fehlerfälle noch die Latenz der Erkennung kann – verglichen zu den Simulationen ohne Optimierungstechnik – verbessert werden. Der Mehraufwand hinsichtlich Ausführungszeit und benötigtem Instruktionsspeicher kann zwar deutlich verringert werden, allerdings auf Kosten der Erkennungsmöglichkeiten. Je nach Einsatzgebiet kann diese Kompromisslösung sinnvoll sein.

Es stellt sich die Frage nach einem direkten Vergleich zu Optimierungstechnik 2, der Reduktion der Checkpoint-Information: Auch dabei wird der Mehraufwand reduziert, während sich die Erkennungsmöglichkeiten etwas verschlechtern. Möglicherweise ist auf der Suche nach einem optimalen Kompromiss auch eine Kombination der Optimierungstechniken interessant. Dies soll im Folgenden näher untersucht werden.

6.4 Kombination der Optimierungstechniken

Im Rahmen der vorhergehenden Abschnitte wurden drei Optimierungstechniken vorgestellt, mit der Zielsetzung, das Verhältnis zwischen Erkennungsmöglichkeiten und Zusatzaufwand zu verbessern. Anhand zahlreicher Evaluierungen konnte die Wirksamkeit der einzelnen Techniken näher gezeigt werden. Nun stellt sich die Frage, wie sich eine Kombination der unterschiedlichen Optimierungen im Hinblick auf Fehlererkennung und Aufwand verhält und ob sich dadurch womöglich weitere Verbesserungen ergeben können.

Zunächst gilt es dabei zu untersuchen, welche Varianten überhaupt für eine mögliche Kombination geeignet sind:

- Die Berücksichtigung einer zeitlichen Untergrenze (Optimierungstechnik 1) erbringt bei den Evaluierungen ein relativ schlechtes Ergebnis. Die Abdeckung erkannter Fehlerfälle wird unwesentlich verbessert, während der Aufwand hinsichtlich Ausführungszeit und Speicherbedarf nicht gesenkt werden kann. Im Gegensatz kommt sogar ein hoher Aufwand durch die Notwendigkeit einer BCET-Analyse zum Zeitpunkt der Instrumentierung hinzu. Eine Kombination dieser Optimierungstechnik mit anderen Techniken erscheint daher wenig sinnvoll und soll im Folgenden vernachlässigt werden.

- Durch die Reduktion der Checkpoint-Informationen auf 16 Bit bei Optimierungstechnik 2 ergibt sich eine starke Verringerung des Aufwands. Im Gegenzug verschlechtern sich allerdings die Abdeckung erkannter Fehlerfälle sowie die Erkennungslatenz. Da der Aufwand hinsichtlich Ausführungszeit und Speicherbedarf für alle evaluierten Varianten 2a bis 2f identisch ist, genügt es, bei einer weiteren Kombination mit anderen Techniken diejenige Variante mit der besten Erkennungsrate zu betrachten – in diesem Fall Variante 2f.

- Auch durch die Angleichung der Checkpoint-Abstände (Optimierungstechnik 3) lässt sich hauptsächlich der Aufwand der Erkennungstechnik reduzieren. Die Evaluierung der Varianten 3a bis 3e zeigt eine unterschiedliche Auswirkung der Optimierungsparameter auf das Verhältnis zwischen Erkennungsrate, Latenz und Aufwand. Bei einer Kombination mit anderen Techniken ist es daher interessant, sämtliche Varianten zu betrachten.

Somit ergeben sich Kombinationen aus Optimierungstechnik 2 (Reduktion der Checkpoint-Informationen) und Optimierungstechnik 3 (Angleichung der Checkpoint-Abstände), welche im Folgenden in verschiedenen Varianten evaluiert werden sollen.

6.4.1 Evaluierung

Die Evaluierung der kombinierten Optimierungstechniken erfolgt nach den bekannten Kriterien: Zunächst wird die Abdeckung erkannter Fehlerfälle ermittelt, um anschließend die Erkennungslatenz sowie den Mehraufwand genauer zu analysieren. Wie bereits erwähnt, wird zur Reduktion der Checkpoint-Informationen

die Bitmaske von Variante 2f (vgl. Abbildung 6.8, S. 98) eingesetzt. Die Schwellenwerte zur Angleichung der Checkpoint-Abstände entsprechen Abschnitt 6.3.3[2].

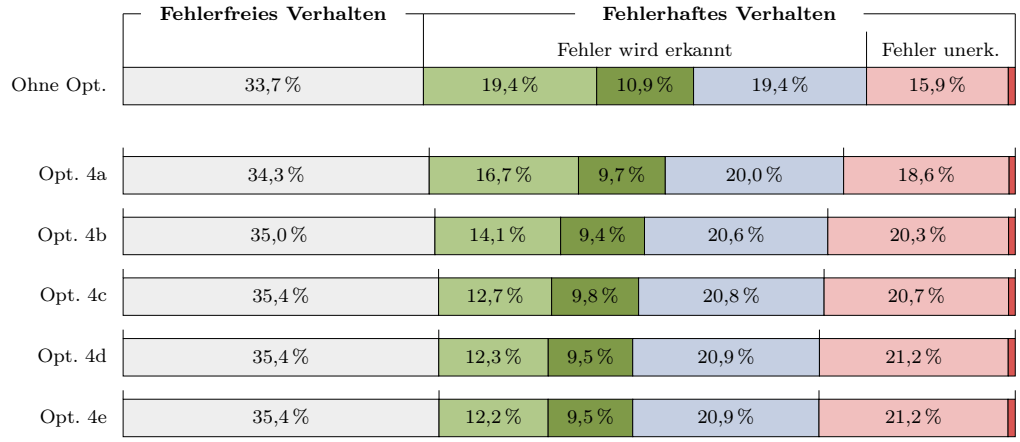
Abdeckung erkannter Fehlerfälle

Abbildung 6.17(a) zeigt die Abdeckung erkannter Fehlerfälle der vorgestellten Kombinationsmöglichkeiten: Zunächst ist zu sehen, dass der Anteil an Simulationsläufen, in welchen trotz des injizierten Bitflips ein fehlerfreies Verhalten vorliegt, von 34,3 % bei Optimierungsvariante 4a mit steigendem Schwellenwert auf bis zu 35,4 % bei den Varianten 4c, 4d und 4e ansteigt. Verglichen zu den Simulationen ohne Optimierungen ist dieser Wert nur minimal erhöht. Der Anteil, in welchem fehlerhaftes Verhalten zur Laufzeit erfolgreich erkannt wird, liegt bei Variante 4a zunächst bei insgesamt 46,4 % und ist damit um 3,3 % niedriger als bei den Durchläufen ohne Optimierungstechniken. Dabei werden 16,7 % der Fehler durch die logische und 9,7 % durch die temporale Check-Einheit erkannt, während 20,0 % der Erkennungsfälle auf prozessorseitige Ausnahmebehandlungen zurückzuführen sind. Mit steigenden Schwellenwerten bei der Angleichung der Checkpoint-Abstände nimmt vor allem die Erkennungsrate auf Basis logischer Checks ab (12,2 % bei Optimierungsvariante 4e), während der Anteil der Erkennungsfälle durch WCET-Checks etwa auf demselben Niveau bleibt. Die Zahl der Ausnahmebehandlungen nimmt im Gegenzug leicht zu und erreicht bei Variante 4e einen maximalen Wert von 20,9 %. Der Prozentsatz unerkannt bleibender Fehler, welche zu einem falschen Ergebnis der Simulation führen, liegt bei Variante 4a zunächst bei 18,6 % und steigt auf bis zu 21,2 % bei den Varianten 4d und 4e. Der Anteil an Endlosschleifen hat bei sämtlichen Varianten einen Wert zwischen 0,7 % und 0,8 % und ist damit fast identisch zu den Messungen ohne Optimierungstechniken.

Zur besseren Bewertung der Abdeckung erkannter Fehlerfälle zeigt Abbildung 6.17(b) wiederum nur diejenigen Fälle, in welchen die Simulation ein fehlerhaftes Verhalten aufweist, welches nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist. Die Summe erkannter Fehlerfälle durch den integrierten Check-Mechanismus liegt demnach bei Optimierungsvariante 4a bei etwa 57,9 % und ist damit um 6,6 % niedriger als bei Simulationen ohne Optimierungen. Mit steigendem Schwellenwert bei der Angleichung der Checkpoint-Abstände nimmt die Erkennungsrate stetig ab und erreicht bei Variante 4e schließlich noch einen Wert von 49,7 %. Bemerkenswert ist jedoch, dass der Anteil der Erkennungsfälle mit Hilfe der temporalen Check-Einheit fast gleich bleibt und bei höheren Schwellenwerten

[2] Im Folgenden werden die Kombinationsvarianten auch als Optimierungstechnik 4a, 4b, 4c, 4d und 4e bezeichnet, wobei zur Angleichung der Checkpoint-Abstände die Schwellenwerte 50, 75, 100, 125 und 150 zum Einsatz kommen.

(a) Zusammenfassung aller Simulationsergebnisse:



(b) Zusammenfassung der Simulationsergebnisse mit fehlerhaftem Verhalten, das nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist:

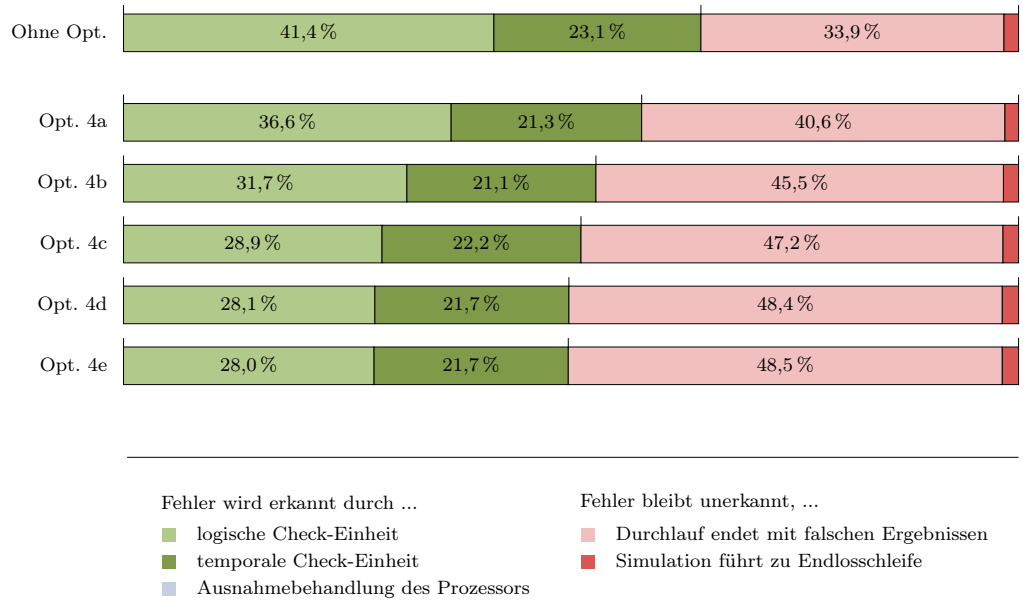


Abbildung 6.17: Abdeckung erkannter Fehlerfälle bei gleichzeitiger Reduktion der Checkpoint-Informationen auf 16 Bit und Angleichung der Abstände anhand unterschiedlicher Schwellenwerte

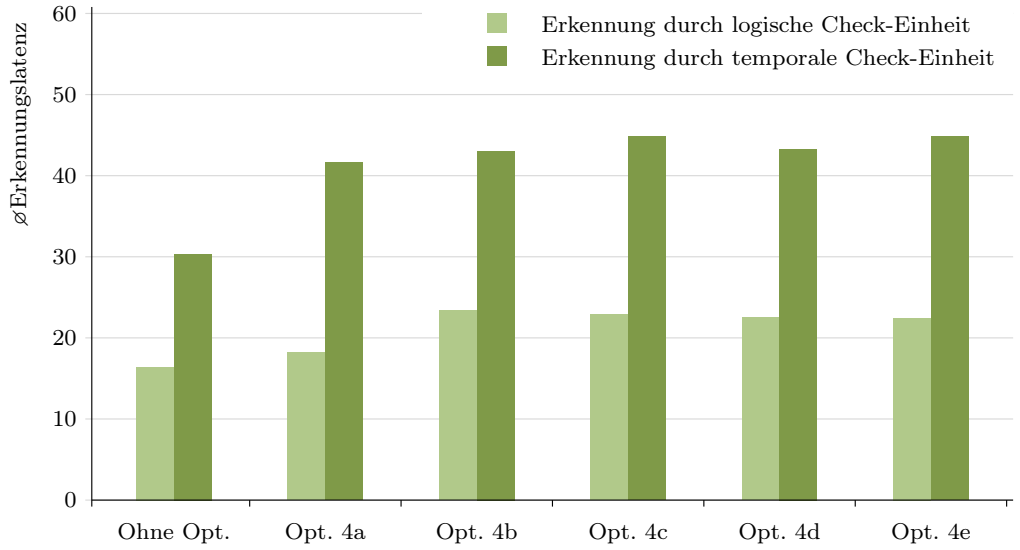


Abbildung 6.18: Durchschnittliche Latenzzeiten der Fehlererkennung bei gleichzeitiger Reduktion der Checkpoint-Informationen auf 16 Bit und Angleichung der Abstände

sogar minimal ansteigt. Der Anteil unerkannter Fehler, welche ein falsches Simulationsergebnis auslösen, wächst von 33,9 % ohne Optimierungen auf 40,6 % bei Optimierungsvariante 4a. Bei allen weiteren Varianten vergrößert sich der Anteil stetig, um schließlich bei Variante 4e ein Maximum von 48,5 % zu erreichen. Die Zahl der Endlosschleifen steigt hingegen nur minimal von etwa 1,5 % bei Variante 4a auf etwa 1,8 % bei den Varianten 4d und 4e.

Wie zu erwarten war, sinkt bei der Kombination der Optimierungstechniken die Abdeckung erkannter Fehlerfälle noch etwas weiter ab. Nun ist es interessant, die anderen Aspekte der Evaluierung zu betrachten, um bewerten zu können, ob diese Verschlechterung anderweitig kompensiert werden kann.

Latenz der Fehlererkennung

In Abbildung 6.18 sind die Erkennungslatenzen der kombinierten Optimierungstechniken in unterschiedlichen Varianten aufgetragen: Wie in den vorhergehenden Betrachtungen sind auch hier diejenigen Einzelfälle vernachlässigt, welche eine höhere Latenzzeit als 100 Takte aufweisen.

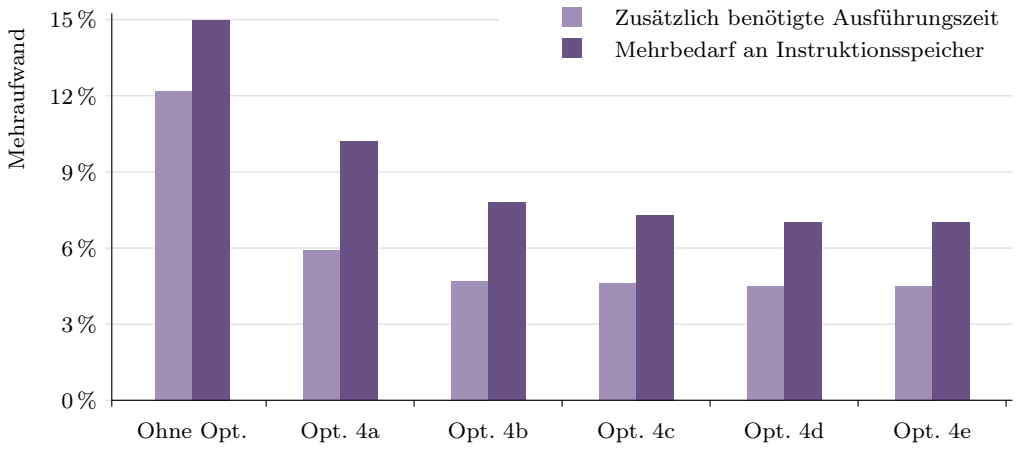


Abbildung 6.19: Mehraufwand bei gleichzeitiger Reduktion der Checkpoint-Informationen auf 16 Bit und Angleichung der Abstände

Betrachtet man die Erkennung auf Basis der logischen Check-Einheit, so ist bei Variante 4a zunächst nur ein leichter Anstieg auf 18,2 Takte gegenüber den Simulationen ohne Optimierungen zu sehen. Bei den weiteren Optimierungsvarianten steigt dieser Durchschnittswert auf etwa 22-23 Takte an. Mit steigendem Schwellenwert ist allerdings eher ein minimaler Rückgang als ein weiteres Anwachsen zu beobachten.

Im Gegensatz dazu ist bei den Erkennungsfällen auf Basis der temporalen Check-Einheit ein starker Anstieg zu sehen: Während die Latenz ohne Optimierungen noch bei 30,3 Takten liegt, ist bei Variante 4a ein Anwachsen auf 41,6 Takte zu verzeichnen. Bei allen weiteren Varianten liegt der Wert zwischen etwa 43 und 45 Takten.

Aufwandsanalyse

Im Hinblick auf den Mehraufwand ist durch die Kombination der Optimierungstechniken eine deutliche Reduktion zu erkennen, wie Abbildung 6.19 zeigt: Bei Variante 4a liegt die zusätzlich benötigte Ausführungszeit bei 5,9 % und ist damit etwa halb so groß wie bei Simulationen ohne Optimierungstechniken. Mit größeren Schwellenwerten nimmt dieser Wert weiter ab und erreicht etwa 4,5 % bei den Varianten 4d und 4e.

Beobachtet man den Mehrbedarf an Instruktionsspeicher, so ist ein ähnliches Verhalten zu sehen: Bei Optimierungsvariante 4a reduziert sich der Anteil zusätzlich eingefügter Befehle auf 10,2 % und sinkt bei den weiteren evaluierten Varianten auf bis zu 7,1 % ab.

6.4.2 Fazit

Als Fazit lässt sich feststellen, dass die Kombination der Optimierungstechniken die offensichtlichen Erwartungen erfüllt: Gegenüber den einzelnen Techniken kann der Aufwand der Erkennungstechnik weiter gesenkt werden. Auf der anderen Seite verschlechtern sich jedoch die Abdeckung erkannter Fehlerfälle sowie die Latenz der Fehlererkennung. Somit scheint ein Einsatz hauptsächlich in Bereichen nützlich, wo es gilt, den Mehraufwand besonders gering zu halten.

Interessant ist nun ein direkter Vergleich der Ergebnisse zu den Evaluierungen der anderen Optimierungstechniken – was Inhalt des folgenden Abschnitts ist.

6.5 Vergleich und Bewertung

Im Rahmen der Evaluierungen wurden die einzelnen Aspekte der Optimierungstechniken und deren Varianten ausführlich beleuchtet. Nun ist es interessant, in einem direkten Vergleich eine Übersicht über die wichtigsten Evaluierungsergebnisse aller Optimierungstechniken sowie deren Kombinationen zu betrachten. Gerade im Hinblick auf das Verhältnis zwischen Erkennungspotenzial, Latenz und Mehraufwand scheint eine solche Gegenüberstellung hilfreich zu sein.

Der Vergleich soll in drei Schritten erfolgen: Zunächst gilt es, die durchschnittliche Erkennungslatenz für alle Optimierungsvarianten ins Verhältnis zur jeweiligen Abdeckung erkannter Fehlerfälle zu setzen. Im Anschluss sollen die beiden Aspekte der Aufwandsanalyse, d.h. die zusätzlich benötigte Ausführungszeit sowie der Mehrbedarf an Speicher, ebenfalls in Relation zur Abdeckung erkannter Fehlerfälle dargestellt und bewertet werden.

6.5.1 Abdeckung erkannter Fehlerfälle vs. Erkennungslatenz

Abbildung 6.20 zeigt den ersten Vergleich der Optimierungstechniken: Auf der x-Achse ist dabei die Abdeckung erkannter Fehlerfälle dargestellt, d.h. der Anteil erfolgreich erkannter Fehler durch die logische und temporale Check-Einheit. Berücksichtigt werden dabei nur Simulationen, welche ein fehlerhaftes Verhalten

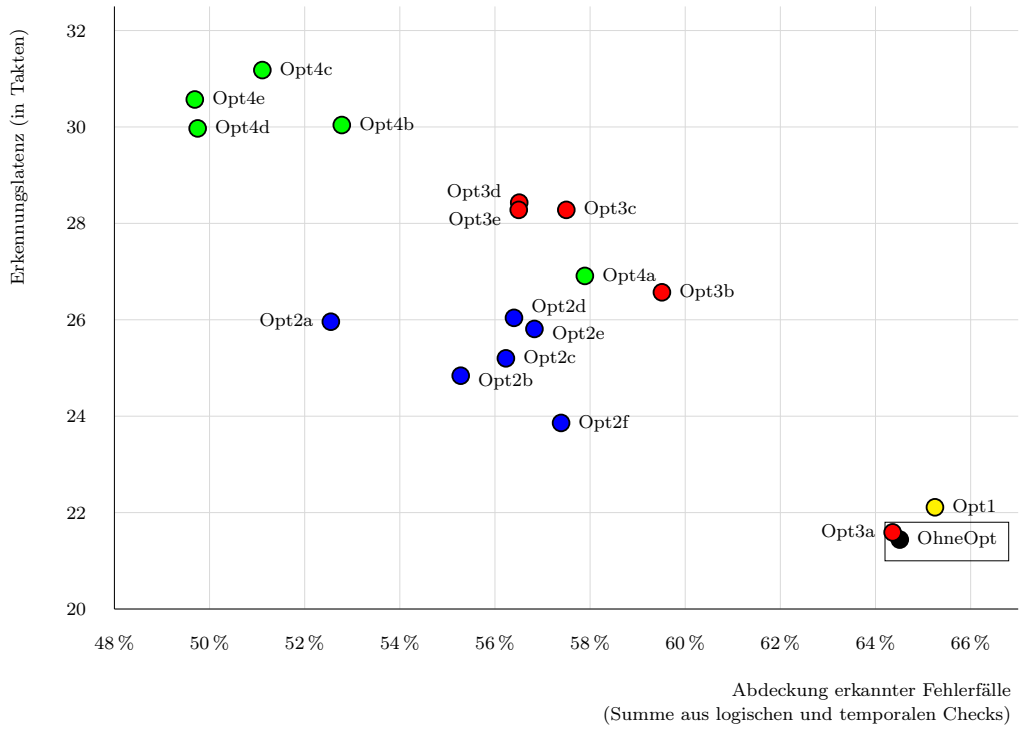


Abbildung 6.20: Vergleich zwischen Abdeckung erkannter Fehlerfälle und Erkennungslatenz der evaluierten Optimierungstechniken

zeigen, das nicht durch prozessorseitige Ausnahmebehandlungen erkennbar ist. Auf der y-Achse ist die durchschnittliche Erkennungslatenz zu sehen, wobei auch hier nicht zwischen logischer und temporaler Fehlererkennung unterschieden wird. Die Optimierungstechniken sind dabei in verschiedenen Farben aufgetragen, wobei mehrere Varianten einer Technik dieselbe Farbe haben.

Zu beobachten ist in Abbildung 6.20, dass die ursprüngliche Technik ohne Optimierungen die besten Werte hinsichtlich Fehlerabdeckung und Latenz liefert. Die Angleichung der Checkpoint-Abstände mit Schwellenwert 50 (Opt3a) zeigt ähnlich gute Ergebnisse. Bei der Angabe der BCET-Werte (Opt1) wird zwar die Abdeckung erkannter Fehlerfälle leicht erhöht, doch auch die Erkennungslatenzen steigen etwas an. Bei den Optimierungsvarianten mit Reduktion der Checkpoint-Information (Opt2a - Opt2f) ist ein starker Abfall beider Werte zu erkennen, wobei Opt2f von diesen Varianten die besten Ergebnisse hinsichtlich Fehlerabdeckung und Latenz erzielen kann. Bei der Angleichung der Checkpoint-Abstände

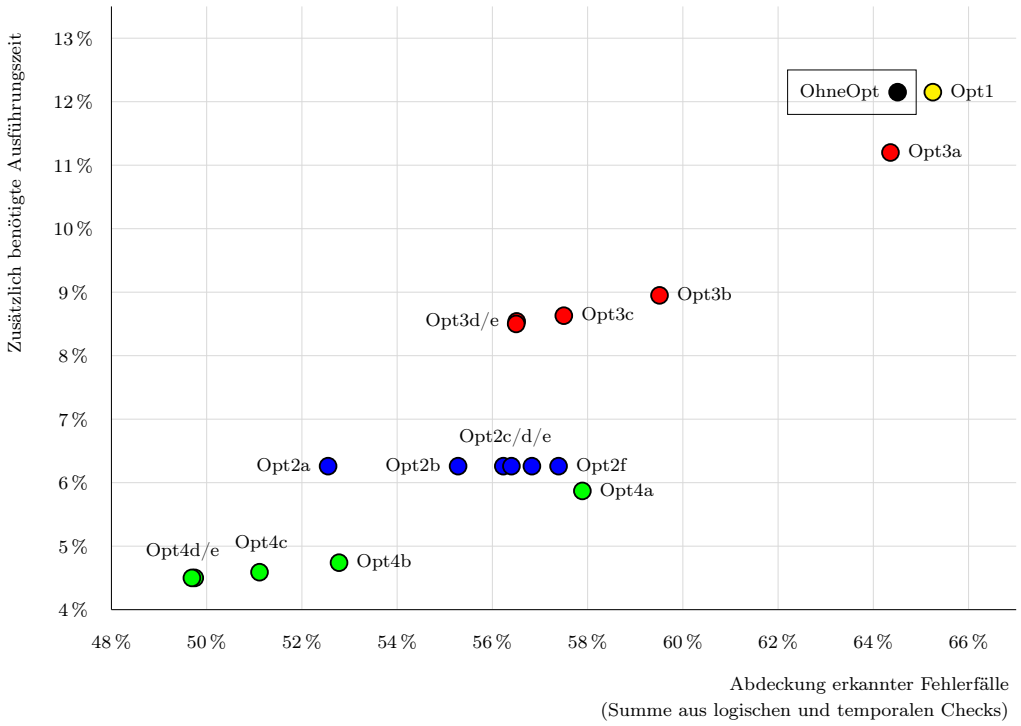


Abbildung 6.21: Vergleich zwischen Abdeckung erkannter Fehlerfälle und Ausführungszeit der evaluierten Optimierungstechniken

mit Schwellenwerten größer als 50 (Opt3b - Opt3e) ist eine deutliche Verschlechterung der Erkennungslatenz zu verzeichnen, wobei auch die Abdeckung erkannter Fehlerfälle mit zunehmenden Schwellenwerten stärker abnimmt. Die Varianten, welche die Angleichung der Checkpoint-Abstände mit der Reduktion der Information kombinieren (Opt4a - Opt4e) können in diesem Vergleich nicht überzeugen: Lediglich die Variante mit Schwellenwert 50 (Opt4a) kann mit den anderen Techniken vergleichbare Ergebnisse liefern. Alle anderen Varianten schneiden sowohl hinsichtlich der Abdeckung erkannter Fehler als auch hinsichtlich der Erkennungslatenz deutlich schlechter ab.

6.5.2 Abdeckung erkannter Fehlerfälle vs. Ausführungszeit

An zweiter Stelle soll nun das Verhältnis zwischen Erkennungsrate und zusätzlicher Ausführungszeit der unterschiedlichen Techniken verglichen werden. Abbil-

dung 6.21 veranschaulicht auf der x-Achse wiederum die Abdeckung erkannter Fehlerfälle, auf der y-Achse den prozentualen Mehraufwand hinsichtlich der Laufzeit der evaluierten Benchmark-Applikationen. Auch hier soll nicht zwischen der logischen und temporalen Erkennungstechnik unterschieden werden.

Abbildung 6.21 zeigt: Die Technik ohne Optimierungen erzielt zwar die bestmögliche Abdeckung erkannter Fehlerfälle, benötigt aber am meisten zusätzliche Ausführungszeit. Optimierungstechnik 1 mit Angabe der zeitlichen Untergrenze bewirkt noch einen minimalen Anstieg der Erkennungsrate, während die Ausführungszeit gleich hoch bleibt wie bei Simulationen ohne Optimierungstechniken. Die Angleichung der Checkpoint-Abstände mit Schwellenwert 50 (Opt3a) kann bei fast identischer Abdeckung den Mehraufwand um etwa ein Prozent senken. Die Angleichung der Abstände mit Schwellenwerten größer als 50 (Opt3b - Opt3e) reduziert den Mehraufwand deutlich, schmälert aber auch die Abdeckung erkannter Fehlerfälle. Im direkten Vergleich der Varianten fällt auf, dass der Einsatz von Opt3c, Opt3d und Opt3e völlig nutzlos erscheint, da andere Techniken bei niedrigerem Aufwand eine höhere Erkennungsrate erzielen können. Die Optimierungstechnik mit Reduktion der Checkpoint-Information (Opt2a - Opt2f) zeigt in allen Varianten denselben Mehraufwand an Ausführungszeit. Im direkten Vergleich erweist sich deshalb Opt2f als zunächst optimale Variante mit der besten Erkennungsrate. Durch die Kombination der Optimierungstechniken entsteht allerdings mit Variante Opt4a eine weitere Verbesserung, welche bei weiter reduziertem Aufwand eine noch bessere Abdeckung erkannter Fehlerfälle bietet. Schließlich bleiben noch die Kombinationen Opt4b - Opt4e, welche – verglichen zu Opt4a – die zusätzlich benötigte Ausführungszeit weiter senken, jedoch auch Einbußen hinsichtlich der Erkennungsrate zu verzeichnen haben. Da diese vier Varianten bzgl. Mehraufwands etwa auf demselben Niveau liegen, ist im direkten Vergleich wohl die Kombination Opt4b unter der Zielsetzung eines minimalen Aufwands zu bevorzugen.

6.5.3 Abdeckung erkannter Fehlerfälle vs. Speicherbedarf

Auch das Verhältnis zwischen Erkennungsrate und zusätzlichem Bedarf an Instruktionsspeicher ist ein wichtiges Kriterium, um die unterschiedlichen Techniken zu vergleichen. Abbildung 6.22 zeigt analog zu den vorhergehenden Abschnitten auf der x-Achse die Abdeckung erkannter Fehlerfälle, während auf der y-Achse nun der zusätzliche Speicherbedarf dargestellt ist. Wie bereits erwähnt, werden bei der Abdeckung erkannter Fehler nur diejenigen Durchläufe berücksichtigt, welche bei der Ausführung ein fehlerhaftes Verhalten hervorrufen, das nicht durch eine prozessorseitige Ausnahmebehandlung erkannt wird.

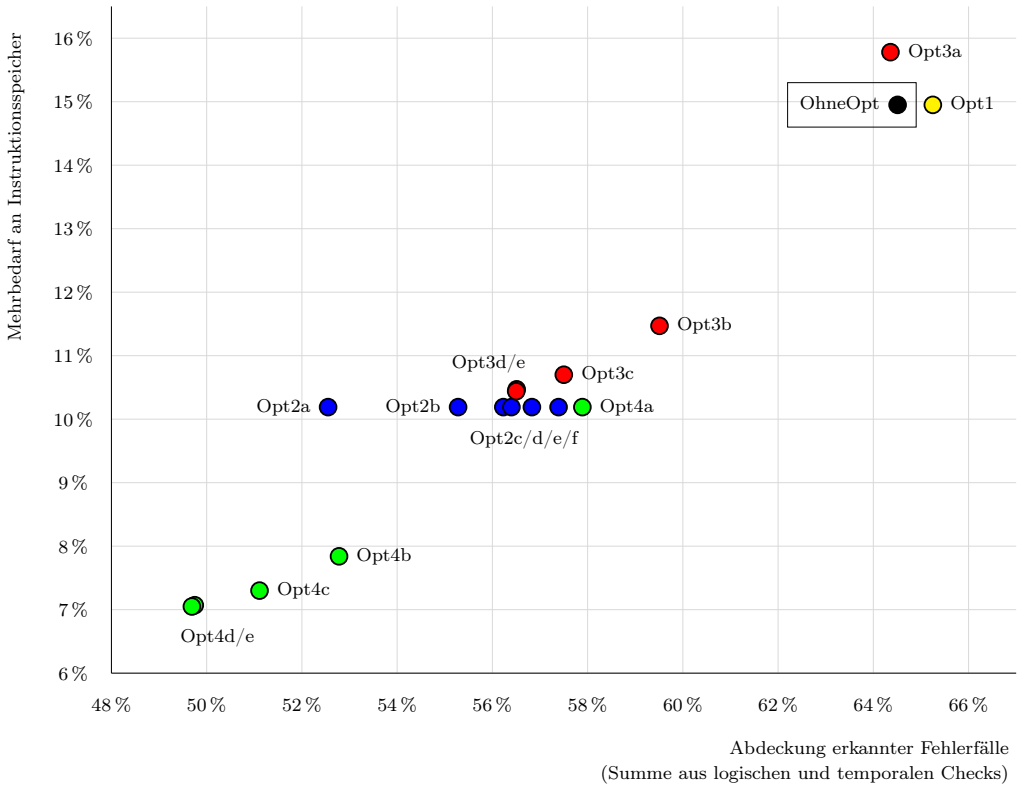


Abbildung 6.22: Vergleich zwischen Abdeckung erkannter Fehlerfälle und Speicherbedarf der evaluierten Optimierungstechniken

In Diagramm 6.22 ist natürlich eine starke Ähnlichkeit zu Abbildung 6.21 zu sehen, doch einige Aspekte sind verschieden: So erzeugt die Angleichung der Checkpoint-Abstände bei Schwellenwert 50 (Opt3a) in diesem Fall einen noch höheren Mehraufwand als die Technik ohne Optimierungen. Der Einsatz der Optimierungsvariante ist damit in dieser Hinsicht nutzlos. Die Angabe einer zeitlichen Untergrenze (Opt1) kann – analog zu den vorhergehenden Betrachtungen – bei gleichem Bedarf an Instruktionsspeicher eine minimale Verbesserung der Erkennungsrate gegenüber Durchläufen ohne Optimierungen verzeichnen. Ein positiver Effekt zeigt sich bei der Angleichung der Checkpoint-Abstände mit Schwellenwert 75 (Opt3b): Hierbei wird zwar die Erkennungsrate um etwa 2,5 % verschlechtert, doch der Mehrbedarf an Speicher reduziert sich von etwa 15 % auf weniger als 12 %. Im Gegensatz dazu sind alle weiteren Optimierungsvarianten mit Angleichung des Checkpoint-Abstands (Opt3c, Opt3d und Opt3e) nutzlos, da die kombinierte Op-

timierung Opt4a bei niedrigerem Aufwand eine höhere Abdeckung erkannter Fehlerfälle erzielen kann. Auch den direkten Vergleich zu den Varianten mit Reduktion der Checkpoint-Information (Opt2a - Opt2f) kann Opt4a für sich entscheiden: Bei gleichem Aufwand ist die Erkennungsrate dort am höchsten. Unter der Zielsetzung, den Mehrbedarf an Instruktionsspeicher auf ein Minimum zu reduzieren, erscheinen die kombinierten Optimierungen Opt4b - Opt4e günstig. Allerdings ist mit höheren Schwellenwerten kaum noch eine Reduktion des Speicherbedarfs zu beobachten, wohl aber ein Rückgang hinsichtlich der Abdeckung erkannter Fehlerfälle. Unter diesem Gesichtspunkt dürfte die Variante Opt4b mit knapp 8 % Speicherbedarf und einer Erkennungsrate von über 52 % im direkten Vergleich unter diesen vier Varianten optimal sein.

7

Zusammenfassung und Ausblick

Dieses Kapitel soll einerseits die Motivation und Konzepte dieser Arbeit, andererseits die entwickelten und implementierten Techniken sowie deren Ergebnisse zusammenfassen. Im Anschluss folgt ein kurzer Ausblick auf mögliche anknüpfende Forschungsthemen, welche in diesem Zusammenhang bisher nicht näher betrachtet wurden.

7.1 Zusammenfassung

Der Einsatz von sicherheitskritischen eingebetteten Systemen im Automobilbereich, in der Luft- und Raumfahrtindustrie sowie im Maschinenbau bringt hohe Anforderungen mit sich. Neben der logischen Korrektheit der Ausführung ist meist auch die zeitliche Korrektheit von essenzieller Bedeutung: In harten Echtzeitsystemen ist ein verspätetes Ergebnis nicht nur wertlos, sondern verursacht möglicherweise katastrophale Konsequenzen. Auch die Verlässlichkeit eines Systems ist in diesen Domänen ein entscheidender Faktor. Im Falle eines auftretenden Fehlzustands darf es nicht zu einem völligen Absturz kommen, sondern der Fehler muss selbstständig erkannt und entsprechend behoben werden, um eine sichere Fortsetzung der Ausführung zu gewährleisten.

Aus diesen Anforderungen ergibt sich die Notwendigkeit, einen Mechanismus zur möglichst frühen Erkennung von logischen und zeitlichen Fehlern zu entwerfen, um rechtzeitig vor Überschreiten einer kritischen Zeitschranke Gegenmaßnahmen

bzw. eine Fehlerbehandlung einleiten zu können. Bei der Entwicklung und Umsetzung eines solchen Erkennungsmechanismus sind einige grundsätzliche Kriterien zu bedenken:

- **Zeitpunkt der Erkennung:** Ein hoher Anteil an Fehlerfällen entsteht durch transiente, d.h. temporäre, nicht systematisch auftretende Fehlerursachen, ausgelöst durch Umwelteinflüsse, beispielsweise Strahlung oder Überhitzung. Um diese Fehler identifizieren zu können, muss ein Erkennungsmechanismus zur Laufzeit des Programms arbeiten. Eine Prüfung im Vorfeld ist sinnlos, da die Fehlerursache zu diesem Zeitpunkt in der Regel noch nicht auftritt.
- **Basis der Korrektheitsprüfung:** Verschiedene Studien zeigen, dass sehr viele Fehler eine Veränderung des Kontrollflusses eines Programms verursachen. Eine logische und temporale Korrektheitsprüfung der Abfolge der Grundblöcke eines Programms stellt daher eine erfolgversprechende Basis für einen Erkennungsmechanismus dar.
- **Art der Umsetzung:** Ein Vergleich mit verwandten Arbeiten zeigt, dass ein hybrider Ansatz bestehend aus softwareseitiger Code-Instrumentierung und hardwareseitiger Prüfung ein hohes Maß an Flexibilität bei relativ niedrigem Zusatzaufwand bieten kann.

Unter Berücksichtigung dieser Gesichtspunkte wurde im Rahmen der Arbeit ein Mechanismus zur feingranularen Online-Erkennung von Fehlern im Kontrollfluss von Echtzeitsystemen vorgestellt. Dabei kommt einerseits eine softwareseitige Instrumentierung der Programme mit Hilfe von Checkpoints zum Einsatz, andererseits eine hardwareseitige Check-Einheit, welche zur Laufzeit die Informationen der Checkpoints auswertet und damit die Korrektheit des Programmablaufs überprüft.

Die Erkennung temporaler Fehler erfolgt dabei auf Basis der WCET-Abschätzungen einzelner Grundblöcke eines Programms. Diese Werte werden im Vorfeld ermittelt und in den entsprechenden Checkpoints gespeichert. Zur Laufzeit werden die Informationen von der Check-Einheit ausgelesen und an einen modifizierten Watchdog-Timer übermittelt, der einen Fehler meldet, sobald die maximal erlaubte Ausführungszeit überschritten ist. Um auch logische Kontrollflussfehler erkennen zu können, werden den einzelnen Grundblöcken zunächst IDs zugeordnet und ebenfalls in den Checkpoints gespeichert. Zusätzlich werden bereits im Vorfeld die erlaubten Pfade im Kontrollfluss analysiert und die IDs der erlaubten Nachfolger in den Checkpoints ergänzt. Aufgabe des Hardware-Checkers beim Erreichen eines Checkpoint ist es somit, die aktuelle ID mit dem im vorhergehenden Checkpoint explizit angekündigten Nachfolger zu vergleichen. Wie in der Arbeit gezeigt wurde,

kann dieser Mechanismus sämtliche Abweichungen vom sequentiellen Kontrollfluss bewältigen. Lediglich indirekte Sprünge stellen ein Hindernis dar, was jedoch im Hinblick auf den Einsatz in harten Echtzeitsystemen zu vernachlässigen ist.

Sucht man auf diesem Gebiet nach verwandten Arbeiten, so findet sich eine ganze Reihe von Mechanismen, um logische Kontrollflussfehler im Programmablauf zu erkennen. Allerdings sind diese Techniken nicht auf die Anforderungen für einen Einsatz in eingebetteten Echtzeitsystemen ausgerichtet. Die zeitliche Korrektheit wird vernachlässigt. Auf der anderen Seite existieren Mechanismen, welche ausschließlich das Zeitverhalten einer Anwendung prüfen, die logische Korrektheit jedoch nicht mit einbeziehen. Die Neuheit an der vorliegenden Arbeit ist dabei, den Kontrollfluss sowohl nach logischen als auch nach temporalen Aspekten zu prüfen – unter der Zielsetzung, eine äußerst geringe Erkennungslatenz und möglichst wenig Zusatzaufwand zu erreichen.

Um sowohl die Wirksamkeit als auch den zusätzlichen Aufwand der entworfenen Technik analysieren und evaluieren zu können, wurde im Rahmen der Arbeit eine realitätsnahe Implementierung auf Basis des echtzeitfähigen CarCore-Prozessors vorgestellt. Zum einen war es Ziel, ein Instrumentierungstool mit WCET-Abschätzung zu entwerfen und in den Kompilierprozess zu integrieren, zum anderen wurde eine hardwareseitige Prüfeinheit in SystemC modelliert und an das CarCore-Modell angeschlossen. Mit Hilfe einer VHDL-Synthese konnte in diesem Zusammenhang der sehr geringe Hardware-Aufwand dieser Einheit nachgewiesen werden.

Im Zuge einer umfassenden Evaluierung wurde der entworfene Mechanismus anhand der vorgestellten Implementierung untersucht. Eine Erzeugung künstlicher Fehlerfälle hilft dabei, einerseits die Abdeckung erkannter Fehlerfälle und andererseits die Latenz der Fehlererkennung zu ermitteln. Die Resultate können von der Wirksamkeit der eingesetzten Technik überzeugen: Mehr als 64 % der simulierten transienten Fehler, ausgelöst durch einzeln erzeugte Bitflips im Instruktionsspeicher, können zuverlässig erkannt werden. Die durchschnittliche Erkennungslatenz beträgt dabei nur etwas mehr als 20 Prozessortakte. Als positiver Effekt ist auch hervorzuheben, dass sich die Anzahl unerkannter Endlosschleifen, welche durch das Fehlverhalten ausgelöst werden, durch Einsatz des Erkennungsmechanismus um mehr als 90 % reduziert. Da bei der Instrumentierung zusätzliche Instruktionen eingefügt werden, ist es ebenso interessant, den dadurch entstehenden Mehraufwand zu analysieren: Bei den untersuchten Benchmark-Programmen zeigt sich im Durchschnitt eine um 12 % erhöhte Laufzeit, während der Bedarf an Instruktionsspeicher um etwa 15 % wächst.

Beim Betrachten der Evaluierungsergebnisse kommt die Frage auf, ob sich die Resultate durch Modifikation des Erkennungsmechanismus weiter verbessern las-

sen. Zielsetzung möglicher Optimierungstechniken ist dabei auf der einen Seite, die Fehlererkennung zu verbessern, d.h. die Erkennungsrate zu erhöhen und die Latenz zu senken. Auf der anderen Seite steht die Motivation, den Zusatzaufwand weiter zu reduzieren. Im Rahmen der vorliegenden Arbeit wurden daher drei unterschiedliche Optimierungstechniken entwickelt und evaluiert: Die erste Technik beruht auf der Idee, neben der zeitlichen Obergrenze auch die zeitliche Untergrenze (BCET-Abschätzung) der Grundblöcke als Referenz für die Korrektheitsprüfung zu verwenden. Dazu wird einerseits das Instrumentierungstool um ein Modul zur BCET-Abschätzung erweitert, andererseits die hardwareseitige Prüfeinheit um einen zusätzlichen Zähler ergänzt. Die Evaluierung zeigt, dass die Abdeckung erkannter Fehlerfälle etwas zunimmt, die Erkennungslatenz sich aber deutlich verschlechtert. Insgesamt erscheint der minimale Anstieg der Erkennungsrate damit im Vergleich zum Aufwand als nicht gerechtfertigt.

Eine zweite Optimierungstechnik verfolgt die Motivation, die Checkpoint-Informationen zu verringern und damit den Aufwand durch die Instrumentierung stark zu reduzieren. Anstelle von 32 Bit breiten Checkpoints werden nur noch 16 Bit verwendet. Bei Programmen mit einer höheren Anzahl von Grundblöcken ist es dabei nicht mehr möglich, eindeutige IDs zu vergeben; stattdessen werden den Grundblöcken (möglichst unterschiedliche) Kennzahlen zugeordnet und deren Abfolge überprüft. Auch die WCET-Abschätzung wird aus Platzgründen skaliert im Checkpoint gespeichert. Bei der Evaluierung dieser Technik ist zu sehen, dass sich Erkennungsrate und -latenz zwar (wie etwartet) etwas verschlechtern, der Aufwand jedoch im Gegenzug massiv gesenkt werden kann.

Die Idee der dritten Optimierungstechnik besteht schließlich darin, die Abstände zwischen aufeinanderfolgenden Checkpoints anzugleichen: Bei sehr kurzen Grundblöcken könnten Checkpoints eliminiert werden, während es bei langen Grundblöcken sinnvoll wäre, zusätzliche Checkpoints einzufügen. Dieses Angleichen der Abstände verspricht auf der einen Seite eine Verbesserung der Erkennungsrate und -latenz, auf der anderen Seite eine Aufwandsreduktion. Im Rahmen der Evaluierungen zeigt sich, dass es nicht möglich ist, beide Zielsetzungen gleichzeitig zu erfüllen. Allerdings lässt sich bei größer gewählten Checkpoint-Abständen der Aufwand deutlich reduzieren, während die Abdeckung erkannter Fehlerfälle nur geringfügig abnimmt. Den Abschluss des Kapitels über Optimierungstechniken bildet die Untersuchung der Kombination der vorgestellten Ideen sowie ein umfassender Vergleich: Dabei ist zu erkennen, dass durch den Einsatz der Optimierungstechniken eine deutliche Reduktion des zusätzlichen Aufwands möglich ist, allerdings stets auf Kosten einer Verschlechterung von Erkennungsrate und -latenz. Ein Einsatz der Optimierungstechniken erscheint daher hauptsächlich für diejenigen Bereiche nützlich, in welchen es gilt, den Mehraufwand besonders gering zu halten.

7.2 Ausblick

Anknüpfend an die vorliegende Arbeit ergeben sich verschiedene Themen, welche ein interessantes weiteres Forschungsfeld bilden. Eine erste Idee wäre die Zielsetzung, die Fehlererkennung weiter zu verbessern. Dies könnte einerseits durch eine Kombination der vorgestellten Technik mit gängigen Erkennungs- und Korrekturmechanismen wie ECC erfolgen. Andererseits wurden bei der Implementierung und Evaluierung des vorgestellten Mechanismus zur Erkennung von Kontrollflussfehlern die Abhängigkeiten von der Prozessorarchitektur weitgehend ausgeklammert. Ein Einfluss des verwendeten Prozessors und Befehlssatzes auf die Abdeckung erkannter Fehlerfälle ist allerdings zweifellos vorhanden: Ein auftretender transienter Fehler, welcher den Opcode eines Befehls modifiziert, kann je nach Prozessor völlig unterschiedliche Auswirkungen auf den Kontrollfluss haben. Anders betrachtet stellt sich damit folgende Frage: Wie müsste der Befehlssatz eines Prozessors entworfen werden, damit es bei Einsatz der gegebenen Erkennungstechnik durch Bitflips in möglichst häufigen Fällen und möglichst rasch zu einer Änderung des Kontrollflusses kommt?

Ein weiterer Bereich, der im Rahmen dieser Arbeit größtenteils vernachlässigt wurde, ist die Fehlerbehandlung in harten Echtzeitsystemen. Das zuverlässige Erkennen eines fehlerhaften Zustands bedeutet noch nicht, dass sich das System wieder in einem sicheren Zustand befindet. Vielmehr ist es nötig, einen Mechanismus zur Vorwärts- oder Rückwärtsfehlerbehandlung bzw. zur Fehlerkompensation einzusetzen. Gerade im Hinblick auf harte Echtzeitsysteme ergeben sich dabei einige offene Fragen: In welchen Fällen kann ein fehlerhafter Programmcode erneut ausgeführt werden, ohne die Zeitschranken zu überschreiten? Wie würde das Rücksetzen des Systemzustands funktionieren? Kann gegebenenfalls ein neuer fehlerfreier Zustand gefunden werden, von welchem aus das System die Arbeit fortsetzen kann? Wie müsste ein Rückfallsystem konzipiert sein, welches bei der Erkennung eines Fehlers die Arbeit des Hauptsystems übernimmt, um die geforderten Zeitschranken dennoch zu erreichen? Für ein Rücksetzen des Systemzustandes auf einen früheren, fehlerfreien Zustand der Ausführung könnten möglicherweise die integrierten Checkpoints genutzt bzw. erweitert werden. Ein Ansatz von Yalcin u. a. (2011) verfolgt die Idee, die Konzepte transaktionaler Speicher zur Fehlerbehandlung zu verwenden. Eine nähere Betrachtung dieser Techniken mit Fokus auf Echtzeitsysteme wäre wohl ein interessanter Anknüpfpunkt an die vorliegende Arbeit.

Neben der Fehlzustandsbehandlung bleibt schließlich die Fehlerursachenbehandlung ein potentielles Forschungsfeld, um die vorgestellte Technik zur Erkennung von Kontrollflussfehlern in Echtzeitsystemen fortzuführen. Gerade im Hinblick auf

permanente Fehler ist eine zuverlässige Diagnose der Fehlerquelle nötig, um das defekte Bauteil nach Möglichkeit zu isolieren. Auch eine Umkonfiguration oder Neuinitialisierung sind potentielle Varianten der Fehlerursachenbehandlung, welche im Kontext harter Echtzeitsysteme noch ein recht offenes Forschungsthema darstellen.

Literaturverzeichnis

[Alkhalifa u. a. 1999] Alkhalifa, Z. ; Nair, V.S.S. ; Krishnamurthy, N. ; Abraham, J.A.: Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. In: *IEEE Transactions on Parallel and Distributed Systems* 10 (1999), Nr. 6, S. 627–641. – ISSN 1045-9219 40

[Allen 1970] Allen, F.E.: Control Flow Analysis. In: *ACM SIGPLAN Notices* 5 (1970), Nr. 7, S. 1–19. – ISSN 0362-1340 6

[Arora u. a. 2006] Arora, D. ; Ravi, S. ; Raghunathan, A. ; Jha, N.K.: Hardware-Assisted Run-Time Monitoring for Secure Program Execution on Embedded Processors. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14 (2006), Nr. 12, S. 1295–1308. – ISSN 1063-8210 37

[Austin 1999] Austin, T.M.: DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In: *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*. Washington, DC, USA : IEEE Computer Society, 1999, S. 196–207. – ISBN 0-7695-0437-X 37

[AUTOSAR 2008] *AUTOSAR: Concept for Dual-Microcontroller Support*. 2008. – Version 1.1 15, 16

[Avizienis 1985] Avizienis, A.: The N-Version Approach to Fault-Tolerant Software. In: *IEEE Transactions on Software Engineering* 11 (1985), Nr. 12, S. 1491–1501. – ISSN 0098-5589 15, 39

[Avizienis und Kelly 1984] Avizienis, A. ; Kelly, J.P.J.: Fault Tolerance by Design Diversity - Concepts and Experiments . In: *Computer* 17 (1984), Nr. 8, S. 67–80. – ISSN 0018-9162 15

[Avizienis u. a. 2004] Avizienis, A. ; Laprie, J.-C. ; Randell, B. ; Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE Transactions on Dependable and Secure Computing* 1 (2004), Nr. 1, S. 11–33. – ISSN 1545-5971 11, 13, 14, 15, 16

[Bagchi u. a. 2001] Bagchi, S. ; Liu, Y. ; Whisnant, K. ; Kalbarczyk, Z. ; Iyer, R. ; Levendel, Y. ; Votta, L.: A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller. In: *Proceedings of the International Conference on Dependable Systems and Networks.*

Los Alamitos, CA, USA : IEEE Computer Society, 2001, S. 225–234. – ISBN 0-7695-1101-5 41

[Ballabriga u. a. 2010] Ballabriga, Clément ; Cassé, Hugues ; Rochange, Christine ; Sainrat, Pascal: OTAWA: An Open Toolbox for Adaptive WCET Analysis. In: *Proceedings of the 8th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS)* Bd. LNCS 6399. Berlin, Heidelberg : Springer-Verlag, 2010, S. 35–46. – ISBN 978-3-642-16255-8 9

[Bernardi u. a. 2006] Bernardi, P. ; Bolzani, L.M.V. ; Rebaudengo, M. ; Reorda, M. S. ; F.L.Vargas ; Violante, M.: A New Hybrid Fault Detection Technique for Systems-on-a-Chip. In: *IEEE Transactions on Computers* 55 (2006), Nr. 2, S. 185–198. – ISSN 0018-9340 40, 43

[Bonenfant u. a. 2010] Bonenfant, A. ; Broster, I. ; Ballabriga, C. ; Bernat, G. ; Cassé, H. ; Houston, M. ; Merriam, N. ; Michiel, M. de ; Rochange, C. ; Sainrat, P.: Coding Guidelines for WCET Analysis Using Measurement-Based and Static Analysis Techniques / IRIT - Institut de recherche en informatique de Toulouse. 2010. – Forschungsbericht 10, 26

[Bowen und Stavridou 1993] Bowen, J. ; Stavridou, V.: Safety-critical systems, formal methods and standards. In: *Software Engineering Journal* 8 (1993), Nr. 4, S. 189–209. – ISSN 0268-6961 10

[Brinkschulte und Ungerer 2010] Brinkschulte, Uwe ; Ungerer, Theo: *Mikrocontroller und Mikroprozessoren*. Berlin, Heidelberg : Springer-Verlag, 2010. – ISBN 978-3-642-05397-9 21, 35, 48, 79

[Buttazzo 1997] Buttazzo, G.C.: *Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications*. Norwell, Massachusetts, USA : Kluwer Academic Publishers, 1997. – ISBN 0-79-239994-3 7

[Cheynet u. a. 2000] Cheynet, P. ; Nicolescu, B. ; Velazco, R. ; Rebaudengo, M. ; Sonza Reorda, M. ; Violante, M.: Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors. In: *IEEE Transactions on Nuclear Science* 47 (2000), Nr. 6, S. 2231–2236. – ISSN 0018-9499 43

[Chodrow u. a. 1991] Chodrow, S.E. ; Jahanian, F. ; Donner, M.: Run-Time Monitoring of Real-Time Systems. In: *Proceedings of the 12th Real-Time Systems Symposium (RTSS)*. Los Alamitos, CA, USA : IEEE Computer Society, 1991, S. 74–83. – ISBN 0-8186-2450-7 42

[Cristian 1985] Cristian, F.: A Rigorous Approach to Fault-Tolerant Programming. In: *IEEE Transactions on Software Engineering* SE-11 (1985), Nr. 1, S. 23–31. – ISSN 0098-5589 16

[Czeck und Siewiorek 1990] Czeck, E. W. ; Siewiorek, D. P.: Effects of Transient Gate-Level Faults on Program Behavior. In: *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS)*. Los Alamitos, CA, USA : IEEE Computer Society, 1990, S. 236–243. – ISBN 0-8186-2051-X 17

[EEMBC 2011] *EEMBC AutoBench 1.1 Data Book*. 2011. – http://eembc.org/ 64

[Ferdinand und Heckmann 2004] Ferdinand, C. ; Heckmann, R.: aiT: Worst-Case Execution Time Prediction by Static Program Analysis. In: *Building the Information Society* Bd. 156. Springer Boston, 2004, S. 377–383. – ISBN 978-1-4020-8156-9 9

[Goloubeva u. a. 2003] Goloubeva, O. ; Rebaudengo, M. ; Reorda, M. S. ; Violante, M.: Soft-Error Detection Using Control Flow Assertions. In: *Proceedings of the 18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT)*. Los Alamitos, CA, USA : IEEE Computer Society, 2003, S. 581–588. – ISBN 0-7695-2042-1 40, 41, 43

[Gärtner 2001] Gärtner, F.C.: *Formale Grundlagen der Fehlertoleranz in verteilten Systemen*, Technische Universität Darmstadt, Dissertation, 2001 16

[Grune u. a. 2000] Grune, D. ; Bal, H.E. ; Jacobs, C.J.H. ; Langendoen, K.G.: *Modern Compiler Design*. Chichester, England : John Wiley & Sons, Ltd, 2000. – ISBN 0-471-97697-0 6

[Hennessy und Patterson 1994] Hennessy, J.L. ; Patterson, D.A.: *Rechnerarchitektur – Analyse, Entwurf, Implementierung, Bewertung*. Braunschweig/Wiesbaden : Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1994. – ISBN 3-528-05173-6 5

[Jones 2000] Jones, A.: Guest Editor's Introduction: The Challenge of Building Survivable Information-Intensive Systems. In: *IEEE Computer* 33 (2000), S. 39–43. – ISSN 0018-9162 10

[Kanawati u. a. 1996] Kanawati, G.A. ; Nair, V.S.S. ; Krishnamurthy, N. ; Abraham, J.A.: Evaluation of Integrated System-Level Checks for On-Line Error Detection. In: *Proceedings of IEEE International Computer Performance and Dependability Symposium*. Los Alamitos, CA, USA : IEEE Computer Society, 1996, S. 292–301. – ISBN 0-8186-7484-9 39

[Karnik u. a. 2004] Karnik, T. ; Hazucha, P. ; J., Patel: Characterization of Soft Errors Caused by Single Event Upsets in CMOS Processes. In: *IEEE Transactions on Dependable and Secure Computing* 1 (2004), Nr. 2, S. 128–143. – ISSN 1545-5971 12

[Kirner u. a. 2004] Kirner, R. ; Puschner, P. ; Wenzel, I.: Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation. In: *Proceedings of the 4th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Rennes Cedex, France : IRISA, Campus Universitaire de Beaulieu, 2004, S. 67–70. – ISSN 1166-8687 9

[Laprie u. a. 1992] Laprie, J.C. C. ; Avizienis, A. ; Kopetz, H.: *Dependability: Basic Concepts and Terminology*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1992. – ISBN 0-387-82296-8 11, 13, 16

[Lu 1982] Lu, D.J.: Watchdog Processors and Structural Integrity Checking. In: *IEEE Transactions on Computers* 31 (1982), Nr. 7, S. 681–685. – ISSN 0018-9340 36

[Lyu 1995] Lyu, M.R.: *Software Fault Tolerance*. Hoboken, NJ, USA : John Wiley & Sons Ltd, 1995. – ISBN 0-4719-5068-8 15

[Mahmood u. a. 1983] Mahmood, A. ; Lu, D.J. ; McCluskey, E.J.: Concurrent Fault Detection Using a Watchdog Processor and Assertions. In: *Proceedings of the IEEE International Test Conference (ITC)*. Los Alamitos, CA, USA : IEEE Computer Society, 1983, S. 622–628. – ISBN 0-8186-0502-2 35

[Mahmood und McCluskey 1988] Mahmood, A. ; McCluskey, E.J.: Concurrent Error Detection Using Watchdog Processors–A Survey. In: *IEEE Transactions on Computers* 37 (1988), Nr. 2, S. 160–174. – ISSN 0018-9340 35, 37

[Manacher 1967] Manacher, G.K.: Production and Stabilization of Real-Time Task Schedules. In: *Journal of the ACM* 14 (1967), Nr. 3, S. 439–465. – ISSN 0004-5411 8

[Michel u. a. 1991] Michel, T. ; Leveugle, R. ; Saucier, G.: A New Approach to Control Flow Checking without Program Modification. In: *Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS)*. Los Alamitos, CA, USA : IEEE Computer Society, 1991, S. 334–341. – ISBN 0-8186-2150-8 37

[Miremadi u. a. 1992] Miremadi, G. ; Harlsson, J. ; Gunneflo, U. ; Torin, J.: Two Software Techniques for On-Line Error Detection. In: *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS)*. Los

Alamitos, CA, USA : IEEE Computer Society, 1992, S. 328–335. – ISBN 0-8186-2875-8 40

[Mische u. a. 2010] Mische, J. ; Guliashvili, I. ; Uhrig, S. ; Ungerer, T.: How to Enhance a Superscalar Processor to Provide Hard Real-Time Capable In-Order SMT. In: *Proceedings of the 23rd International Conference on Architecture of Computing Systems (ARCS)* Bd. LNCS 5974. Berlin, Heidelberg : Springer-Verlag, 2010, S. 2–14. – ISBN 978-3-642-11949-1 48

[Mok u. a. 2002] Mok, A.K. ; Lee, Chan-Gun ; Woo, Honguk ; Konana, P.: The Monitoring of Timing Constraints on Time Intervals. In: *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS)*. Los Alamitos, CA, USA : IEEE Computer Society, 2002, S. 191–200. – ISBN 0-7695-1851-6 42

[Mok und Liu 1997] Mok, A.K. ; Liu, Guangtian: Efficient Run-time Monitoring Of Timing Constraints. In: *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*. Los Alamitos, CA, USA : IEEE Computer Society, 1997, S. 252–262. – ISBN 0-8186-8016-4 42

[Namjoo 1982] Namjoo, M.: Techniques for Concurrent Testing of VLSI Processor Operation. In: *Proceedings of the IEEE International Test Conference (ITC)*. Los Alamitos, CA, USA : IEEE Computer Society, 1982, S. 461–468 36

[Namjoo und McCluskey 1982] Namjoo, M. ; McCluskey, E.J.: Watchdog Processors and Capability Checking. In: *Proceedings of the 12th International Symposium on Fault-Tolerant Computing (FTCS)*. Los Alamitos, CA, USA : IEEE Computer Society, 1982, S. 245–248. – ISBN 9-9936-2909-X 35

[Oh u. a. 2002a] Oh, N. ; Mitra, S. ; McCluskey, E.J.: ED^{4}I: Error Detection by Diverse Data and Duplicated Instructions. In: *IEEE Transactions on Computers* 51 (2002), Nr. 2, S. 180–199. – ISSN 0018-9340 39

[Oh u. a. 2002b] Oh, N. ; Shirvani, P.P. ; McCluskey, E.J.: Control-flow Checking by Software Signatures. In: *IEEE Transactions on Reliability* 51 (2002), Nr. 1, S. 111–122. – ISSN 0018-9529 40, 41

[Oh u. a. 2002c] Oh, N. ; Shirvani, P.P. ; McCluskey, E.J.: Error Detection by Duplicated Instructions in Super-Scalar Processors. In: *IEEE Transactions on Reliability* 51 (2002), Nr. 1, S. 63–75. – ISSN 0018-9529 39, 41

[Ohlsson und Rimen 1995] Ohlsson, J. ; Rimen, M.: Implicit Signature Checking. In: *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS)*. Los Alamitos, CA, USA : IEEE Computer Society, 1995, S. 218–227. – ISBN 0-8186-7079-7 36, 37

[Ohlsson u. a. 1992] Ohlsson, J. ; Rimen, M. ; Gunneflo, U.: A Study of the Effects of Transient Fault Injection into a 32-bit RISC with Built-in Watchdog. In: *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS)*. Los Alamitos, CA, USA : IEEE Computer Society, 1992, S. 316–325. – ISBN 0-8186-2875-8 17

[Paolieri und Mariani 2011] Paolieri, Marco ; Mariani, Riccardo: Towards Functional-Safe Timing-Dependable Real-Time Architectures. In: *Proceedings of the 17th IEEE International On-Line Testing Symposium (IOLTS)*. Los Alamitos, CA, USA : IEEE Computer Society, 2011, S. 31–36. – ISBN 978-1-4577-1053-7 44

[Peterson und Weldon 1972] Peterson, W.W. ; Weldon, E.J.: *Error-Correcting Codes*. MIT Press, 1972. – ISBN 0-26-216039-0 15, 41

[Ragel und Parameswaran 2011] Ragel, R.G. ; Parameswaran, S.: A Hybrid Hardware–Software Technique to Improve Reliability in Embedded Processors. In: *ACM Transactions on Embedded Computing Systems* 10 (2011), Nr. 3, S. 36:1–36:16. – ISSN 1539-9087 43

[Randell 1975] Randell, B.: System Structure for Software Fault Tolerance. In: *ACM SIGPLAN Notices* 10 (1975), Nr. 6, S. 437–449. – ISSN 0362-1340 39

[RapiTime 2011] Rapita Systems Ltd.: *RapiTime White Paper*. http://www.rapitasystems.com/. 2011 9

[Reis u. a. 2005] Reis, G.A. ; Chang, J. ; Vachharajani, N. ; Rangan, R. ; August, D.I.: SWIFT: Software Implemented Fault Tolerance. In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. Washington, DC, USA : IEEE Computer Society, 2005, S. 243–254. – ISBN 0-7695-2298-X 41

[Rhod u. a. 2008] Rhod, E. ; Lisbôa, C. ; Carro, L. ; Sonza Reorda, M. ; Violante, M.: Hardware and Software Transparency in the Protection of Programs Against SEUs and SETs. In: *Springer Journal of Electronic Testing* 24 (2008), S. 45–56. – ISSN 0923-8174 35, 39

[RTCA 1992] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. 1992. – Radio Technical Commission for Aeronautics (RTCA) 10

[Scherrer und Steininger 2003] Scherrer, C. ; Steininger, A.: Dealing With Dormant Faults in an Embedded Fault-Tolerant Computer System. In: *IEEE Transactions on Reliability* 52 (2003), Nr. 4, S. 512–522. – ISSN 0018-9529 14, 16

[Schuette und Shen 1987] Schuette, M.A. ; Shen, J.P.: Processor Control Flow Monitoring Using Signatured Instruction Streams. In: *IEEE Transactions on Computers* 36 (1987), Nr. 3, S. 264–276. – ISSN 0018-9340 17, 36, 37

[Shen und Schuette 1983] Shen, J.P. ; Schuette, M.A.: On-Line Self-Monitoring Using Signatured Instruction Streams. In: *Proceedings of the IEEE International Test Conference (ITC)*. Los Alamitos, CA, USA : IEEE Computer Society, 1983, S. 275–282. – ISBN 0-8186-0502-2 37

[Stankovic 1988] Stankovic, J.A.: Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. In: *Computer* 21 (1988), Nr. 10, S. 10–19. – ISSN 0018-9162 8

[Stankovic und Ramamritham 1990] Stankovic, J.A. ; Ramamritham, K.: What is Predictability for Real-Time Systems? In: *Journal of Real-Time Systems* 2 (1990), S. 247–254. – ISSN 0922-6443 8

[TriCore 2008] Infineon Technologies AG: *Tricore 1 User's Manual*, January 2008. – V1.3.8 48, 66

[Tsai u. a. 1990] Tsai, J.J.P. ; Fang, K.-Y. ; Chen, H.-Y.: A Noninvasive Architecture to Monitor Real-Time Distributed Systems. In: *Computer* 23 (1990), Nr. 3, S. 11–23. – ISSN 0018-9162 38, 42

[Uhrig u. a. 2005] Uhrig, S. ; Maier, S. ; Ungerer, T.: Toward a Processor Core for Real-time Capable Autonomic Systems. In: *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*. Los Alamitos, CA, USA : IEEE Computer Society, 2005, S. 19–22. – ISBN 0-7803-9313-9 48

[Vemu und Abraham 2006] Vemu, R. ; Abraham, J.A.: CEDA: Control-flow Error Detection through Assertions. In: *Proceedings of the 12th IEEE International On-Line Testing Symposium (IOLTS)*. Los Alamitos, CA, USA : IEEE Computer Society, 2006, S. 151–158. – ISBN 0-7695-2620-9 40, 41

[Venkatasubramanian u. a. 2003] Venkatasubramanian, R. ; Hayes, J.P. ; Murray, B.T.: Low-Cost On-Line Fault Detection Using Control Flow Assertions. In: *Proceedings of the 9th IEEE On-Line Testing Symposium (IOLTS)*. Los Alamitos, CA, USA : IEEE Computer Society, 2003, S. 137–143. – ISBN 0-7695-1968-7 40, 41

[Wang und Agrawal 2008] Wang, F. ; Agrawal, V.D.: Single Event Upset: An Embedded Tutorial. In: *Proceedings of the 21st International Conference on VLSI Design*. Los Alamitos, CA, USA : IEEE Computer Society, 2008, S. 429–434. – ISBN 0-7695-3083-4 29

[Wilhelm u. a. 2008] Wilhelm, R. ; Engblom, J. ; A., Ermedahl ; Holsti, N. ; Thesing, S. ; Whalley, D. ; Bernat, G. ; Ferdinand, C. ; Heckmann, R. ; Mitra, T. ; Mueller, F. ; Puaut, I. ; Puschner, P. ; Staschulat, J. ; Stenström, P.: The Worst-case Execution Time Problem—Overview of Methods and Survey of Tools. In: *ACM Transactions on Embedded Computing Systems* 7 (2008), Nr. 3, S. 1–53. – ISSN 1539-9087 9, 10

[Wilken und Shen 1990] Wilken, K. ; Shen, J.P.: Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 9 (1990), Nr. 6, S. 629–641. – ISSN 0278-0070 36, 37

[Wolf u. a. 2012a] Wolf, Julian ; Fechner, Bernhard ; Uhrig, Sascha ; Ungerer, Theo: Fine-Grained Timing and Control Flow Error Checking for Hard Real-Time Task Execution. In: *Proceedings of the 7^{th} International Symposium on Industrial Embedded Systems (SIES)*. Los Alamitos, CA, USA : IEEE Computer Society, 2012, S. 257–266. – ISBN 978-1-4673-2685-8 19

[Wolf u. a. 2012b] Wolf, Julian ; Fechner, Bernhard ; Ungerer, Theo: Fault Coverage of a Timing and Control Flow Checker for Hard Real-Time Systems. In: *Proceedings of the 18^{th} IEEE International On-Line Testing Symposium (IOLTS)*. Los Alamitos, CA, USA : IEEE Computer Society, 2012, S. 127–129. – ISBN 978-1-4673-2082-5 63

[Wolf u. a. 2012c] Wolf, Julian ; Fechner, Bernhard ; Ungerer, Theo: Fault Detection Capabilities of an Enhanced Timing and Control Flow Checker for Hard Real-Time Systems. In: *Proceedings of the 4^{th} International Conference on Advances in System Testing and Validation Lifecycle (VALID)*, IARIA, 2012, S. 57–62. – ISBN 978-1-4673-2082-5 63

[Yalcin u. a. 2011] Yalcin, G. ; Unsal, O.S. ; Cristal, A. ; Hur, I. ; Valero, M.: SymptomTM: Symptom-Based Error Detection and Recovery Using Hardware Transactional Memory. In: *Proceedings of the 20^{th} International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Los Alamitos, CA, USA : IEEE Computer Society, 2011, S. 199–200. – ISBN 978-1-4577-1794-9 133

[Ziener und Teich 2009] Ziener, D. ; Teich, J.: Concepts for run-time and error-resilient control flow checking of embedded RISC CPUs. In: *International Journal of Autonomous and Adaptive Communications Systems* 2 (2009), Nr. 3, S. 256–275. – ISSN 1754-8632 37

[Zorian 2002] Zorian, Y.: Guest Editor's Introduction: What is Infrastructure IP? In: *IEEE Design and Test of Computers* 19 (2002), Nr. 3, S. 5–7. – ISSN 0740-7475 43

Abkürzungsverzeichnis

ALUT	Adaptive Look-Up Table (S. 61)
ASM	Autonomous Signature Monitoring (S. 36)
BCET	Best-Case Execution Time (S. 10)
CCA	Control-Flow Checking using Assertions (S. 39)
CEDA	Control-Flow Error Detection through Assertions (S. 40)
CFCSS	Control-Flow Checking by Software Signatures (S. 40)
CPV-Register	Checkpoint Value Register (S. 21)
DIVA	Dynamic Implementation Verification Architecture (S. 37)
ECC	Error Correcting Codes (S. 15)
ECCA	Enhanced Control-Flow Checking using Assertions (S. 40)
EDDI	Error Detection by Duplicated Instructions (S. 39)
EEMBC	Embedded Microprocessor Benchmark Consortium (S. 64)
ESM	Embedded Signature Monitoring (S. 36)
FIR	Finite Impulse Response (S. 65)
IP-Core	Intellectual Property Core (S. 43)
I-IP-Core	Infrastructure Intellectual Property Core (S. 43)
MFCR	Move from Core Register (S. 52)
MTCR	Move to Core Register (S. 52)
PSA	Path Signature Analysis (S. 36)
SEU	Single Event Upset (S. 29)
SIC	Structural Integrity Checking (S. 36)
SIHFT	Software Implemented Hardware Fault Tolerance (S. 39)
SMT	Simultaneous Multithreading (S. 48)
SWIFT	Software Implemented Fault Tolerance (S. 41)
VHDL	Very High Speed Integrated Circuit Hardware Description Language (S. 61)
WCET	Worst-Case Execution Time (S. 8)
YACCA	Yet Another Control-Flow Checking using Assertions (S. 40)

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

A

Messergebnisse

Im Folgenden sollen die einzelnen Messergebnisse der evaluierten Benchmarks *aifirf*, *bitmnp*, *canrdr*, *pntrch*, *puwmod*, *rspeed* und *ttsprk* (vgl. Abschnitt 5.1.1) in ausführlicher Form tabellarisch aufgelistet werden. Eine Auswertung und grafische Darstellung der Resultate findet sich in den entsprechenden Kapiteln der Arbeit.

A.1 Abdeckung erkannter Fehlerfälle

Zur Ermittlung der Abdeckung erkannter Fehlerfälle wurde eine Vielzahl von Simulationen mit jeweils einem injizierten Bitflip ausgeführt (vgl. Abschnitt 5.1.2). Die Tabellen A.1 bis A.20 zeigen dabei die Häufigkeiten des unterschiedlichen Simulationsverhaltens in Abhängigkeit von der jeweils aktivierten Optimierungstechnik. Die Durchläufe sind in folgende Kategorien zusammengefasst:

- `IDCheck`: Die logische Check-Einheit erkennt einen Fehler in der Abfolge der instrumentierten IDs. Die Simulation wird abgebrochen.
- `WCETCheck`: Die temporale Check-Einheit erkennt auf Basis der instrumentierten WCET-Abschätzung eine Überschreitung der zulässigen Laufzeit. Die Simulation wird abgebrochen.
- `BCETCheck`: Die temporale Check-Einheit erkennt auf Basis der instrumentierten BCET-Abschätzung eine Unterschreitung der zulässigen Laufzeit. Die Simulation wird abgebrochen.

- `RegCor/CycCor`: Die Simulation terminiert an der vorgesehenen Stelle. Der Vergleich zu den Referenzwerten bei korrekter Ausführung zeigt keine Abweichungen bei der Gesamtzahl an benötigten Prozessortakten und volle Übereinstimmung der Registerbelegungen am Schluss.

- `RegFal/CycCor`: Die Simulation terminiert an der vorgesehenen Stelle. Der Vergleich zu den Referenzwerten bei korrekter Ausführung zeigt keine Abweichungen bei der Gesamtzahl an benötigten Prozessortakten, jedoch bei den Registerbelegungen am Schluss.

- `RegCor/CycFal`: Die Simulation terminiert an der vorgesehenen Stelle. Der Vergleich zu den Referenzwerten bei korrekter Ausführung zeigt Abweichungen bei der Gesamtzahl an benötigten Prozessortakten, jedoch volle Übereinstimmung der Registerbelegungen am Schluss.

- `RegFal/CycFal`: Die Simulation terminiert an der vorgesehenen Stelle. Der Vergleich zu den Referenzwerten bei korrekter Ausführung zeigt Abweichungen sowohl bei der Gesamtzahl an benötigten Prozessortakten als auch bei den Registerbelegungen am Schluss.

- `ExceptionUOP`: Eine prozessorseitige Ausnahmebehandlung meldet einen unbekannten Instruktions-Opcode (`Unknown Opcode`). Die Simulation wird abgebrochen.

- `ExceptionNOF`: Eine prozessorseitige Ausnahmebehandlung meldet einen unerlaubten Aufbau eines Instruktions-Opcodes (`No Function Found`). Die Simulation wird abgebrochen.

- `ExceptionHWA`: Eine prozessorseitige Ausnahmebehandlung meldet einen fehlerhaft ausgerichteten Zugriff auf ein Speicherwort (`Half Word Access`). Die Simulation wird abgebrochen.

- `ExceptionIRA`: Eine prozessorseitige Ausnahmebehandlung meldet einen unerlaubten Lesezugriff auf ein Speicherwort (`Illegal Read Access`). Die Simulation wird abgebrochen.

- `ExceptionUMR`: Eine prozessorseitige Ausnahmebehandlung meldet einen fehlerhaft ausgerichteten Zugriff beim Lesen aus dem Speicher (`Unaligned Memory Read`). Die Simulation wird abgebrochen.

- `ExceptionUMW`: Eine prozessorseitige Ausnahmebehandlung meldet einen fehlerhaft ausgerichteten Zugriff beim Schreiben in den Speicher (`Unaligned Memory Write`). Die Simulation wird abgebrochen.

- `ExceptionRSP`: Eine prozessorseitige Ausnahmebehandlung meldet das Erreichen eines Stop-Befehls (`Reached Stop PC`). Die Simulation wird abgebrochen.
- `MaxCycles`: Die Gesamtdauer der Simulation (in Prozessortakten) hat eine vorher festgelegte Grenze überschritten. Höchstwahrscheinlich befindet sich das Programm in einer Endlosschleife. Die Simulation wird abgebrochen.

Neben den einzelnen Varianten des Simulationsverhaltens wird in den nachfolgenden Tabellen auch in der Zeile `Injections` jeweils die Summe der Simulationsläufe mit injiziertem Fehler angegeben. Die Zeile `NoInjections` zeigt demgegenüber die Anzahl an Simulationen, in welchen zwar ein Bitflip im Instruktionsspeicher erzeugt wird, die betreffende Instruktion aber im Programmverlauf niemals zur Ausführung kommt. In der Zeile `SimulationRuns` ist schließlich die Summe aller Simulationsläufe dargestellt.

Tabelle A.1 betrachtet das Verhalten bei aktivierter Erkennungstechnik ohne Optimierungen, Tabelle A.2 den Vergleich zu Durchläufen ohne aktiven Check-Mechanismus. In Tabelle A.3 ist das Verhalten mit Berücksichtigung einer zeitlichen Untergrenze zu sehen, während Tabelle A.4 nur eine Gegenüberstellung zu Resultaten zeigt, welche ausschließlich aus der veränderten Bitmaske ohne Einsatz der Optimierungstechnik entstehen. Die Tabellen A.5 bis A.10 veranschaulichen das Verhalten bei Reduktion der Checkpoint-Informationen in verschiedenen Varianten. Schließlich sind in den Tabellen A.11 bis A.15 die Ergebnisse aus der Angleichung der Checkpoint-Abstände und in den Tabellen A.16 bis A.20 die Resultate aus der Kombination verschiedener Optimierungstechniken in unterschiedlichen Varianten zu sehen.

Tabelle A.1: Häufigkeiten des unterschiedlichen Simulationsverhaltens bei aktiviertem Erkennungsmechanismus ohne Optimierungen

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	3546	10190	710	3251	3435	981	5798
WCETCheck	1722	3156	404	2390	2340	761	4832
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6630	10163	731	3814	8164	3772	15094
RegFal/CycCor	522	336	55	188	280	372	277
RegCor/CycFal	485	6771	73	1597	1580	729	4968
RegFal/CycFal	544	1067	31	791	636	499	1037
ExceptionUOP	1642	3013	198	1216	1510	766	3709
ExceptionNOF	1157	2405	130	739	789	582	1633
ExceptionHWA	32	12	16	22	15	4	41
ExceptionIRA	1598	790	145	723	377	159	1650
ExceptionUMR	219	335	16	376	258	70	661
ExceptionUMW	162	174	8	119	71	71	256
ExceptionRSP	2	0	0	1	0	0	0
MaxCycles	484	336	40	64	0	0	171
Injections	18745	38748	2557	15291	19455	8766	40127
NoInjections	2503	21668	3	1797	9985	322	1345
SimulationRuns	21248	60416	2560	17088	29440	9088	41472

Tabelle A.2: Häufigkeiten des unterschiedlichen Simulationsverhaltens ohne Instrumentierung und Erkennungsmechanismus

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	0	0	0	0	0	0	0
WCETCheck	0	0	0	0	0	0	0
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	4525	4856	421	2112	5552	2902	12086
RegFal/CycCor	501	282	58	201	263	372	300
RegCor/CycFal	637	6827	99	1592	1656	747	5869
RegFal/CycFal	667	1455	70	1157	885	677	1534
ExceptionUOP	1084	1443	73	628	883	619	2684
ExceptionNOF	811	1613	77	476	467	507	1354
ExceptionHWA	29	10	15	25	10	4	28
ExceptionIRA	1526	577	116	684	308	135	1557
ExceptionUMR	191	268	9	334	205	61	612
ExceptionUMW	174	188	13	132	91	77	259
ExceptionRSP	0	0	0	1	0	0	0
MaxCycles	793	1274	134	594	624	235	1557
Injections	10938	18793	1085	7936	10944	6336	27840
NoInjections	1606	8407	3	832	4608	0	384
SimulationRuns	12544	27200	1088	8768	15552	6336	28224

Tabelle A.3: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Berücksichtigung der zeitlichen Untergrenze (Opt. 1)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	3412	10281	724	3621	3616	918	5781
WCETCheck	1597	2435	338	1722	1919	950	4388
BCETCheck	398	1168	100	393	428	113	686
RegCor/CycCor	6522	9768	702	3706	8005	3749	14933
RegFal/CycCor	526	339	55	188	280	372	277
RegCor/CycFal	475	6796	74	1620	1605	633	4993
RegFal/CycFal	545	1108	33	823	650	399	1066
ExceptionUOP	1592	2896	190	1155	1450	758	3616
ExceptionNOF	1134	2281	121	710	744	575	1558
ExceptionHWA	30	10	16	22	11	4	28
ExceptionIRA	1659	899	142	788	451	157	1774
ExceptionUMR	192	251	11	357	218	64	600
ExceptionUMW	170	179	9	118	71	71	257
ExceptionRSP	2	0	0	1	0	0	0
MaxCycles	489	337	40	64	0	0	171
Injections	18743	38748	2555	15288	19448	8763	40128
NoInjections	2505	21668	5	1800	9992	325	1344
SimulationRuns	21248	60416	2560	17088	29440	9088	41472

Tabelle A.4: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit veränderter Bitmaske (gemäß Abb. 6.3), jedoch ohne Optimierungstechniken

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	3488	10386	750	3707	3690	959	6024
WCETCheck	1597	2435	338	1723	1919	950	4388
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6840	10729	772	4008	8343	3821	15376
RegFal/CycCor	526	339	55	188	280	372	277
RegCor/CycFal	476	6882	78	1620	1608	633	4993
RegFal/CycFal	547	1124	33	826	663	399	1066
ExceptionUOP	1592	2896	190	1155	1450	758	3616
ExceptionNOF	1134	2281	121	710	744	575	1558
ExceptionHWA	30	10	16	22	11	4	28
ExceptionIRA	1659	899	142	789	451	157	1774
ExceptionUMR	192	251	11	357	218	64	600
ExceptionUMW	170	179	9	118	71	71	257
ExceptionRSP	2	0	0	1	0	0	0
MaxCycles	490	337	40	64	0	0	171
Injections	18743	38748	2555	15288	19448	8763	40128
NoInjections	2505	21668	5	1800	9992	325	1344
SimulationRuns	21248	60416	2560	17088	29440	9088	41472

Tabelle A.5: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Reduktion der Checkpoint-Informationen (Opt. 2a, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	1487	4268	356	1288	1265	369	2101
WCETCheck	1725	3141	351	2337	2356	773	5025
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6644	10285	759	3673	7961	3747	14873
RegFal/CycCor	517	350	58	205	275	373	299
RegCor/CycFal	513	6733	77	1512	1547	722	5201
RegFal/CycFal	532	1138	38	765	605	519	1175
ExceptionUOP	1525	2667	174	1110	1411	748	3464
ExceptionNOF	879	1746	86	520	468	527	1408
ExceptionHWA	24	10	15	32	10	4	43
ExceptionIRA	1539	734	132	709	388	152	1632
ExceptionUMR	188	264	9	328	205	58	589
ExceptionUMW	172	171	8	114	72	68	256
ExceptionRSP	0	0	0	1	0	0	0
MaxCycles	505	405	43	78	13	4	30
Injections	16250	31912	2106	12672	16576	8064	36096
NoInjections	2182	17240	6	1408	8192	64	960
SimulationRuns	18432	49152	2112	14080	24768	8128	37056

Tabelle A.6: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Reduktion der Checkpoint-Informationen (Opt. 2b, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	1954	5480	436	1749	1743	452	2681
WCETCheck	1677	2902	346	2174	2250	746	4900
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6322	9630	698	3480	7663	3704	14574
RegFal/CycCor	511	345	57	204	271	373	299
RegCor/CycFal	497	6645	77	1502	1541	725	5192
RegFal/CycFal	516	1121	35	742	588	510	1159
ExceptionUOP	1523	2678	177	1119	1412	759	3447
ExceptionNOF	865	1668	82	503	456	527	1366
ExceptionHWA	22	10	15	23	11	4	31
ExceptionIRA	1512	684	125	675	360	138	1589
ExceptionUMR	188	253	9	314	205	58	585
ExceptionUMW	173	172	7	116	72	68	256
ExceptionRSP	0	0	0	1	0	0	0
MaxCycles	490	324	42	70	4	0	17
Injections	16250	31912	2106	12672	16576	8064	36096
NoInjections	2182	17240	6	1408	8192	64	960
SimulationRuns	18432	49152	2112	14080	24768	8128	37056

Tabelle A.7: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Reduktion der Checkpoint-Informationen (Opt. 2c, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2221	6606	480	2413	2254	522	3396
WCETCheck	1562	2311	321	1712	1779	745	4348
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6165	9058	667	3284	7579	3643	14302
RegFal/CycCor	510	340	57	201	276	372	302
RegCor/CycFal	489	6602	81	1506	1546	728	5193
RegFal/CycFal	511	1146	38	748	621	509	1190
ExceptionUOP	1503	2630	171	1067	1371	740	3412
ExceptionNOF	856	1647	82	505	443	521	1364
ExceptionHWA	21	10	15	32	10	4	28
ExceptionIRA	1566	819	136	716	415	153	1711
ExceptionUMR	188	268	9	310	206	59	583
ExceptionUMW	173	173	7	113	71	68	256
ExceptionRSP	0	0	0	1	0	0	0
MaxCycles	485	302	42	64	5	0	11
Injections	16250	31912	2106	12672	16576	8064	36096
NoInjections	2182	17240	6	1408	8192	64	960
SimulationRuns	18432	49152	2112	14080	24768	8128	37056

Tabelle A.8: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Reduktion der Checkpoint-Informationen (Opt. 2d, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2258	6527	477	2329	2200	523	3701
WCETCheck	1559	2441	324	1769	1884	749	4089
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6134	9009	667	3259	7551	3634	14261
RegFal/CycCor	507	339	57	201	276	372	301
RegCor/CycFal	489	6590	81	1509	1531	727	5194
RegFal/CycFal	511	1138	38	748	607	511	1190
ExceptionUOP	1503	2664	172	1094	1405	750	3413
ExceptionNOF	859	1655	84	522	443	521	1369
ExceptionHWA	23	10	15	23	10	4	57
ExceptionIRA	1555	794	133	724	387	145	1645
ExceptionUMR	193	268	9	316	206	59	609
ExceptionUMW	174	173	7	113	71	69	256
ExceptionRSP	0	0	0	1	0	0	0
MaxCycles	485	304	42	64	5	0	11
Injections	16250	31912	2106	12672	16576	8064	36096
NoInjections	2182	17240	6	1408	8192	64	960
SimulationRuns	18432	49152	2112	14080	24768	8128	37056

Tabelle A.9: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Reduktion der Checkpoint-Informationen (Opt. 2e, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2502	7356	514	2638	2778	593	4453
WCETCheck	1365	1710	308	1552	1454	711	3430
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6071	8886	639	3165	7400	3600	14142
RegFal/CycCor	511	354	57	200	273	372	304
RegCor/CycFal	495	6569	80	1506	1543	727	5198
RegFal/CycFal	515	1143	38	749	611	513	1189
ExceptionUOP	1513	2692	180	1099	1390	748	3482
ExceptionNOF	864	1692	85	517	459	524	1386
ExceptionHWA	22	18	15	30	10	4	29
ExceptionIRA	1536	754	132	715	378	145	1631
ExceptionUMR	192	284	9	322	206	59	588
ExceptionUMW	179	180	7	113	74	68	255
ExceptionRSP	0	0	0	1	0	0	0
MaxCycles	485	274	42	65	0	0	9
Injections	16250	31912	2106	12672	16576	8064	36096
NoInjections	2182	17240	6	1408	8192	64	960
SimulationRuns	18432	49152	2112	14080	24768	8128	37056

Tabelle A.10: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Reduktion der Checkpoint-Informationen (Opt. 2f, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2468	7190	515	2721	2660	590	4400
WCETCheck	1427	2269	312	1524	1636	716	3612
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6043	8560	636	3108	7342	3593	14003
RegFal/CycCor	511	338	57	200	273	372	302
RegCor/CycFal	495	6570	80	1506	1540	727	5194
RegFal/CycFal	515	1135	38	749	612	513	1191
ExceptionUOP	1513	2687	178	1097	1392	748	3484
ExceptionNOF	864	1693	85	517	459	524	1386
ExceptionHWA	22	18	15	30	10	4	29
ExceptionIRA	1536	721	132	717	371	149	1640
ExceptionUMR	192	284	9	323	206	59	590
ExceptionUMW	179	176	7	114	74	69	256
ExceptionRSP	0	0	0	1	0	0	0
MaxCycles	485	271	42	65	1	0	9
Injections	16250	31912	2106	12672	16576	8064	36096
NoInjections	2182	17240	6	1408	8192	64	960
SimulationRuns	18432	49152	2112	14080	24768	8128	37056

Tabelle A.11: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Angleichung der Checkpoint-Abstände (Opt. 3a, Schwellenwert 50)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	4019	6292	606	3778	3534	1530	7955
WCETCheck	2041	2352	378	2254	2283	841	5758
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6926	7489	713	3789	8135	3743	16578
RegFal/CycCor	529	313	54	193	287	375	300
RegCor/CycFal	502	6750	73	1577	1622	730	4830
RegFal/CycFal	569	1147	38	799	661	512	720
ExceptionUOP	1742	2284	178	1279	1490	852	4031
ExceptionNOF	1212	2144	110	746	827	631	1978
ExceptionHWA	27	14	16	23	13	5	35
ExceptionIRA	1581	618	134	758	398	173	1735
ExceptionUMR	236	330	15	362	258	68	653
ExceptionUMW	167	169	8	122	72	74	258
ExceptionRSP	1	0	0	2	0	0	0
MaxCycles	472	275	41	60	0	0	722
Injections	20024	30177	2364	15742	19580	9534	45553
NoInjections	2568	16287	4	1282	8772	130	1167
SimulationRuns	22592	46464	2368	17024	28352	9664	46720

Tabelle A.12: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Angleichung der Checkpoint-Abstände (Opt. 3b, Schwellenwert 75)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2908	4975	496	3170	3080	802	5804
WCETCheck	1820	1859	378	2153	2045	733	4917
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6686	6625	647	3481	7485	3283	15340
RegFal/CycCor	529	285	55	182	282	376	298
RegCor/CycFal	511	6663	78	1554	1602	735	5079
RegFal/CycFal	547	1128	37	777	619	514	924
ExceptionUOP	1560	2035	160	1164	1410	739	3713
ExceptionNOF	1130	1912	102	668	757	572	1739
ExceptionHWA	24	12	16	24	11	5	29
ExceptionIRA	1584	628	135	739	378	164	1678
ExceptionUMR	214	297	17	367	241	68	628
ExceptionUMW	179	175	7	117	72	73	261
ExceptionRSP	2	0	0	0	0	0	0
MaxCycles	475	260	42	63	0	0	292
Injections	18169	26854	2170	14459	17982	8064	40702
NoInjections	2567	12634	6	1221	8066	64	770
SimulationRuns	20736	39488	2176	15680	26048	8128	41472

Tabelle A.13: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Angleichung der Checkpoint-Abstände (Opt. 3c, Schwellenwert 100)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2582	4391	487	2920	2840	796	5184
WCETCheck	1744	1846	387	2091	1861	735	4608
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6651	6450	646	3409	7537	3288	14851
RegFal/CycCor	549	273	55	191	283	375	296
RegCor/CycFal	523	6668	76	1567	1619	734	5090
RegFal/CycFal	543	1109	37	774	653	514	926
ExceptionUOP	1518	1944	160	1102	1359	739	3553
ExceptionNOF	1146	1818	104	653	745	572	1698
ExceptionHWA	25	19	17	33	13	5	48
ExceptionIRA	1570	610	136	729	373	165	1636
ExceptionUMR	212	311	15	361	242	67	669
ExceptionUMW	176	190	8	119	73	74	257
ExceptionRSP	1	0	0	2	0	0	0
MaxCycles	482	265	42	64	0	0	287
Injections	17722	25894	2170	14015	17598	8064	39103
NoInjections	2502	11482	6	1217	8258	64	769
SimulationRuns	20224	37376	2176	15232	25856	8128	39872

Tabelle A.14: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Angleichung der Checkpoint-Abstände (Opt. 3d, Schwellenwert 125)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2555	3847	487	2869	2866	795	4826
WCETCheck	1703	1837	387	2055	1856	733	4423
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6600	6394	646	3421	7537	3288	14599
RegFal/CycCor	546	291	55	188	283	375	296
RegCor/CycFal	515	6719	76	1568	1620	734	5077
RegFal/CycFal	544	1146	37	774	620	514	982
ExceptionUOP	1521	1887	161	1102	1397	742	3476
ExceptionNOF	1157	1845	103	659	717	573	1644
ExceptionHWA	24	13	17	32	15	5	29
ExceptionIRA	1557	600	137	737	373	164	1640
ExceptionUMR	217	305	15	363	243	66	664
ExceptionUMW	170	174	7	118	71	75	257
ExceptionRSP	2	0	0	1	0	0	0
MaxCycles	483	262	42	64	0	0	230
Injections	17594	25320	2170	13951	17598	8064	38143
NoInjections	2502	11288	6	1217	8258	64	769
SimulationRuns	20096	36608	2176	15168	25856	8128	38912

Tabelle A.15: Häufigkeiten des unterschiedlichen Simulationsverhaltens mit Angleichung der Checkpoint-Abstände (Opt. 3e, Schwellenwert 150)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2555	3825	487	2869	2833	795	4813
WCETCheck	1703	1852	387	2055	1869	733	4448
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6600	6254	646	3421	7538	3288	14596
RegFal/CycCor	546	285	55	188	283	375	296
RegCor/CycFal	515	6707	76	1568	1616	734	5079
RegFal/CycFal	544	1132	37	774	652	514	979
ExceptionUOP	1521	1899	161	1102	1356	742	3455
ExceptionNOF	1157	1827	103	659	738	573	1655
ExceptionHWA	24	12	17	32	18	5	29
ExceptionIRA	1557	598	137	737	379	164	1646
ExceptionUMR	217	301	15	363	241	66	661
ExceptionUMW	170	172	7	118	71	75	256
ExceptionRSP	2	0	0	1	0	0	0
MaxCycles	483	264	42	64	0	0	230
Injections	17594	25128	2170	13951	17594	8064	38143
NoInjections	2502	11480	6	1217	8262	64	769
SimulationRuns	20096	36608	2176	15168	25856	8128	38912

Tabelle A.16: Häufigkeiten des unterschiedlichen Simulationsverhaltens bei Kombination der Optimierungen (Opt. 4a, Schwellenwert 50)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2901	4704	417	2904	2717	1033	5883
WCETCheck	1566	1854	324	1654	1573	675	4311
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	6122	6525	609	3261	7283	3465	14905
RegFal/CycCor	516	287	57	201	270	373	305
RegCor/CycFal	505	6569	83	1504	1545	732	5166
RegFal/CycFal	515	1082	39	738	648	511	1150
ExceptionUOP	1611	2119	163	1137	1393	812	3833
ExceptionNOF	876	1657	85	510	463	536	1432
ExceptionHWA	22	10	16	23	11	4	28
ExceptionIRA	1562	620	128	730	386	173	1690
ExceptionUMR	191	280	9	336	210	59	580
ExceptionUMW	160	181	7	123	72	74	261
ExceptionRSP	1	0	0	1	0	0	0
MaxCycles	468	264	43	62	5	1	8
Injections	17016	26152	1980	13184	16576	8448	39552
NoInjections	2248	12632	4	1152	6976	64	768
SimulationRuns	19264	38784	1984	14336	23552	8512	40320

Tabelle A.17: Häufigkeiten des unterschiedlichen Simulationsverhaltens bei Kombination der Optimierungen (Opt. 4b, Schwellenwert 75)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	2011	3693	344	2494	2376	548	4525
WCETCheck	1508	1648	321	1473	1460	599	3657
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	5818	6043	580	3109	6840	3247	14028
RegFal/CycCor	514	287	57	205	269	376	302
RegCor/CycFal	517	6596	81	1521	1555	733	5231
RegFal/CycFal	525	1095	39	743	639	512	1200
ExceptionUOP	1449	1898	147	1065	1322	724	3448
ExceptionNOF	855	1626	78	548	433	518	1380
ExceptionHWA	23	12	18	25	10	7	31
ExceptionIRA	1542	596	123	713	364	160	1639
ExceptionUMR	208	277	9	337	208	58	582
ExceptionUMW	166	200	12	117	74	70	255
ExceptionRSP	1	0	0	2	0	0	0
MaxCycles	473	259	43	64	2	0	10
Injections	15610	24230	1852	12416	15552	7552	36288
NoInjections	2246	10906	4	1088	6720	0	576
SimulationRuns	17856	35136	1856	13504	22272	7552	36864

Tabelle A.18: Häufigkeiten des unterschiedlichen Simulationsverhaltens bei Kombination der Optimierungen (Opt. 4c, Schwellenwert 100)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	1767	3247	347	2333	2125	533	3798
WCETCheck	1488	1614	324	1479	1385	606	3996
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	5693	5927	578	2951	6974	3254	14120
RegFal/CycCor	519	288	57	203	270	376	321
RegCor/CycFal	510	6642	81	1521	1579	736	5249
RegFal/CycFal	520	1100	39	742	613	527	1208
ExceptionUOP	1406	1883	146	1027	1306	721	3417
ExceptionNOF	863	1620	78	524	438	515	1422
ExceptionHWA	22	12	15	22	10	4	32
ExceptionIRA	1524	589	123	709	372	145	1666
ExceptionUMR	204	280	9	333	213	59	583
ExceptionUMW	167	203	13	122	72	75	276
ExceptionRSP	2	0	0	1	0	0	0
MaxCycles	477	250	43	64	3	1	8
Injections	15162	23655	1853	12031	15360	7552	36096
NoInjections	2182	10329	3	1089	6720	0	1023
SimulationRuns	17344	33984	1856	13120	22080	7552	37119

Tabelle A.19: Häufigkeiten des unterschiedlichen Simulationsverhaltens bei Kombination der Optimierungen (Opt. 4d, Schwellenwert 125)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	1741	2799	347	2302	2122	530	3611
WCETCheck	1476	1680	324	1435	1398	605	3468
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	5676	5771	578	2939	6975	3254	13600
RegFal/CycCor	517	286	57	202	271	376	300
RegCor/CycFal	510	6655	81	1521	1577	736	5291
RegFal/CycFal	519	1121	39	742	644	528	1227
ExceptionUOP	1400	1819	146	1020	1276	720	3314
ExceptionNOF	868	1616	78	528	443	515	1403
ExceptionHWA	23	11	15	22	11	4	30
ExceptionIRA	1524	579	123	740	361	148	1616
ExceptionUMR	202	281	9	332	207	60	605
ExceptionUMW	163	207	13	119	73	75	255
ExceptionRSP	1	0	0	1	0	0	0
MaxCycles	478	252	43	64	1	1	32
Injections	15098	23077	1853	11967	15359	7552	34752
NoInjections	2182	10139	3	1089	6721	0	576
SimulationRuns	17280	33216	1856	13056	22080	7552	35328

Tabelle A.20: Häufigkeiten des unterschiedlichen Simulationsverhaltens bei Kombination der Optimierungen (Opt. 4e, Schwellenwert 150)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
IDCheck	1741	2785	347	2302	2128	530	3582
WCETCheck	1476	1700	324	1435	1393	605	3464
BCETCheck	0	0	0	0	0	0	0
RegCor/CycCor	5676	5774	578	2939	6972	3254	13642
RegFal/CycCor	517	286	57	202	271	376	302
RegCor/CycFal	510	6665	81	1521	1581	736	5305
RegFal/CycFal	519	1122	39	742	643	528	1232
ExceptionUOP	1400	1807	146	1020	1280	720	3295
ExceptionNOF	868	1612	78	528	437	515	1392
ExceptionHWA	23	11	15	22	13	4	28
ExceptionIRA	1524	577	123	740	360	148	1620
ExceptionUMR	202	280	9	332	208	60	601
ExceptionUMW	163	205	13	119	72	75	255
ExceptionRSP	1	0	0	1	0	0	0
MaxCycles	478	253	43	64	0	1	34
Injections	15098	23077	1853	11967	15358	7552	34752
NoInjections	2182	10139	3	1089	6721	0	576
SimulationRuns	17280	33216	1856	13056	22079	7552	35328

A.2 Latenz der Fehlererkennung

Die nachfolgenden Tabellen A.21 bis A.39 zeigen die durchschnittlichen Erkennungslatenzen der einzelnen Benchmarks (in Prozessortakten) abhängig von der jeweiligen Optimierungstechnik. In der Zeile `Latency` ist dabei die durchschnittliche Latenz *sämtlicher* Erkennungsfälle dargestellt, während sich die Zeile `IDLatency` nur auf diejenigen Simulationen beschränkt, in welchen eine Erkennung auf Basis der logischen Check-Einheit stattfindet. Die Erkennungsfälle mit Hilfe des temporalen Check-Mechanismus sind auf die Zeilen `WCETLatency` und `BCETLatency` verteilt, abhängig vom Überschreiten der WCET- oder Unterschreiten der BCET-Abschätzung.

Ergänzend dazu sollen, wie bereits in Abschnitt 5.3 erläutert, nur Erkennungsfälle mit Latenzen bis zu 100 Takten betrachtet werden. Die Zeile `Latency100` zeigt dabei wiederum die durchschnittliche Erkennung mit Hilfe von logischer und temporaler Check-Einheit, während sich die Zeilen `IDLatency100`, `WCETLatency100` und `BCETLatency100` auf die entsprechenden Teilmengen beschränken.

Analog zum vorhergehenden Abschnitt wird in Tabelle A.21 zunächst der Mechanismus ohne Optimierungen betrachtet. Die Tabellen A.22 und A.23 zeigen die Verwendung einer zeitlichen Untergrenze bzw. die dazu benötigte veränderte Bitmaske. In den Tabellen A.24 bis A.29 wird schließlich die Reduktion der Checkpoint-Informationen, in den Tabellen A.30 bis A.34 die Angleichung der Checkpoint-Abstände betrachtet. Die Tabellen A.35 bis A.39 präsentieren abschließend die Kombination der Optimierungstechniken.

Tabelle A.21: Durchschnittliche Erkennungslatenzen bei aktiviertem Erkennungsmechanismus ohne Optimierungen

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
`Latency`	8633	1904	1136	10835	1354	765	4203
`IDLatency`	9251	2072	1756	14060	1898	1314	5445
`WCETLatency`	7361	1362	47	6449	557	56	2713
`BCETLatency`	0	0	0	0	0	0	0
`Latency100`	19	15	23	22	20	26	29
`IDLatency100`	15	12	18	18	15	20	21
`WCETLatency100`	28	24	31	28	26	34	37
`BCETLatency100`	0	0	0	0	0	0	0

Tabelle A.22: Durchschnittliche Erkennungslatenzen mit Berücksichtigung der zeitlichen Untergrenze (Opt. 1)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	7303	1978	1241	9957	1360	2993	3977
IDLatency	9099	2287	1955	12141	1920	1568	5714
WCETLatency	5214	1586	53	7564	601	4721	2302
BCETLatency	289	72	87	328	26	48	61
Latency100	20	15	24	23	21	28	29
IDLatency100	15	12	19	17	16	21	21
WCETLatency100	31	28	35	34	30	35	39
BCETLatency100	25	19	25	27	26	36	32

Tabelle A.23: Durchschnittliche Erkennungslatenzen mit veränderter Bitmaske (gemäß Abb. 6.3), jedoch ohne Optimierungstechniken

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	7764	2136	1324	10696	1444	3104	4145
IDLatency	8931	2265	1896	11860	1882	1503	5487
WCETLatency	5214	1586	53	8193	601	4721	2302
BCETLatency	0	0	0	0	0	0	0
Latency100	20	15	24	23	21	27	29
IDLatency100	16	12	18	17	16	20	21
WCETLatency100	31	28	35	34	30	35	39
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.24: Durchschnittliche Erkennungslatenzen mit Reduktion der Checkpoint-Informationen (Opt. 2a, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	18482	9043	1223	18643	2801	1764	5453
IDLatency	31316	10534	2364	23332	5991	5339	13910
WCETLatency	7420	7016	65	16059	1088	58	1917
BCETLatency	0	0	0	0	0	0	0
Latency100	24	19	26	25	24	30	33
IDLatency100	17	13	17	16	15	19	18
WCETLatency100	30	27	34	30	28	34	38
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.25: Durchschnittliche Erkennungslatenzen mit Reduktion der Checkpoint-Informationen (Opt. 2b, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
`Latency`	15746	4686	1092	13858	2304	419	3978
`IDLatency`	24451	4918	1914	19292	3811	926	7907
`WCETLatency`	5604	4249	57	9487	1136	111	1829
`BCETLatency`	0	0	0	0	0	0	0
`Latency100`	23	18	25	24	23	29	32
`IDLatency100`	16	12	17	16	16	19	20
`WCETLatency100`	31	28	34	30	29	35	39
`BCETLatency100`	0	0	0	0	0	0	0

Tabelle A.26: Durchschnittliche Erkennungslatenzen mit Reduktion der Checkpoint-Informationen (Opt. 2c, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
`Latency`	10447	2849	1066	12411	2474	658	4138
`IDLatency`	15773	3309	1741	15222	3324	1434	6440
`WCETLatency`	2874	1536	57	8449	1398	114	2339
`BCETLatency`	0	0	0	0	0	0	0
`Latency100`	23	17	26	25	25	31	33
`IDLatency100`	16	12	17	17	17	21	20
`WCETLatency100`	34	31	38	36	35	38	42
`BCETLatency100`	0	0	0	0	0	0	0

Tabelle A.27: Durchschnittliche Erkennungslatenzen mit Reduktion der Checkpoint-Informationen (Opt. 2d, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
`Latency`	10070	2801	1067	12066	2280	637	4092
`IDLatency`	15405	3349	1752	14961	3431	1465	5637
`WCETLatency`	2343	1338	60	8256	936	59	2694
`BCETLatency`	0	0	0	0	0	0	0
`Latency100`	24	18	27	25	26	33	34
`IDLatency100`	17	12	18	17	17	21	22
`WCETLatency100`	34	32	41	36	35	41	44
`BCETLatency100`	0	0	0	0	0	0	0

Tabelle A.28: Durchschnittliche Erkennungslatenzen mit Reduktion der Checkpoint-Informationen (Opt. 2e, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	9505	2310	645	12262	1529	509	3754
IDLatency	13592	2503	990	15009	1676	1035	5357
WCETLatency	2014	1483	69	7592	1247	70	1674
BCETLatency	0	0	0	0	0	0	0
Latency100	26	18	29	27	24	35	32
IDLatency100	17	13	18	17	17	22	23
WCETLatency100	43	44	48	43	40	46	45
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.29: Durchschnittliche Erkennungslatenzen mit Reduktion der Checkpoint-Informationen (Opt. 2f, vgl. Abb. 6.8)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	9436	2256	640	12076	1547	509	3638
IDLatency	13779	2560	988	14556	1816	967	5347
WCETLatency	1924	1292	65	7648	1109	131	1556
BCETLatency	0	0	0	0	0	0	0
Latency100	23	16	23	25	23	31	31
IDLatency100	17	13	18	17	17	22	23
WCETLatency100	35	27	34	40	36	40	43
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.30: Durchschnittliche Erkennungslatenzen mit Angleichung der Checkpoint-Abstände (Opt. 3a, Schwellenwert 50)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	8707	2009	1274	11100	1301	541	2692
IDLatency	11331	2043	2050	12706	1694	823	3624
WCETLatency	3540	1918	31	8408	693	27	1405
BCETLatency	0	0	0	0	0	0	0
Latency100	20	19	24	21	20	20	24
IDLatency100	16	14	20	18	16	16	20
WCETLatency100	27	30	30	27	26	27	29
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.31: Durchschnittliche Erkennungslatenzen mit Angleichung der Checkpoint-Abstände (Opt. 3b, Schwellenwert 75)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	7278	2317	1429	10809	1461	813	3384
IDLatency	7733	2497	2484	11453	1869	1512	4614
WCETLatency	6551	1835	43	9860	847	48	1932
BCETLatency	0	0	0	0	0	0	0
Latency100	24	24	27	24	22	30	31
IDLatency100	20	17	23	19	17	22	24
WCETLatency100	32	43	33	31	30	39	38
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.32: Durchschnittliche Erkennungslatenzen mit Angleichung der Checkpoint-Abstände (Opt. 3c, Schwellenwert 100)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	6006	2557	1429	9977	1593	814	4120
IDLatency	5367	2729	2528	10336	2017	1525	5621
WCETLatency	6953	2148	45	9476	946	44	2431
BCETLatency	0	0	0	0	0	0	0
Latency100	24	27	28	24	25	31	33
IDLatency100	19	19	23	19	19	22	25
WCETLatency100	31	46	33	32	34	39	41
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.33: Durchschnittliche Erkennungslatenzen mit Angleichung der Checkpoint-Abstände (Opt. 3d, Schwellenwert 125)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	6102	2721	1429	10088	1583	815	3921
IDLatency	5423	3212	2528	10519	1982	1526	5223
WCETLatency	7120	1691	45	9487	965	44	2501
BCETLatency	0	0	0	0	0	0	0
Latency100	24	28	28	24	25	30	33
IDLatency100	19	20	23	18	19	22	24
WCETLatency100	31	47	33	32	34	39	43
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.34: Durchschnittliche Erkennungslatenzen mit Angleichung der Checkpoint-Abstände (Opt. 3e, Schwellenwert 150)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	6102	2693	1429	10088	1594	815	3974
IDLatency	5423	2904	2528	10519	2054	1526	5457
WCETLatency	7120	2257	45	9487	895	44	2370
BCETLatency	0	0	0	0	0	0	0
Latency100	24	26	28	24	25	30	34
IDLatency100	19	20	23	18	19	22	24
WCETLatency100	31	41	33	32	34	39	44
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.35: Durchschnittliche Erkennungslatenzen bei Kombination der Optimierungen (Opt. 4a, Schwellenwert 50)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	10477	2782	700	13131	1365	373	4062
IDLatency	14253	3193	1209	15340	1671	562	5138
WCETLatency	3484	1740	45	9252	836	84	2593
BCETLatency	0	0	0	0	0	0	0
Latency100	25	23	26	26	26	26	30
IDLatency100	17	15	16	18	18	17	20
WCETLatency100	39	43	39	41	39	39	42
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.36: Durchschnittliche Erkennungslatenzen bei Kombination der Optimierungen (Opt. 4b, Schwellenwert 75)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	12252	3033	787	13669	1397	639	3471
IDLatency	20397	3651	1455	16610	1558	1258	3881
WCETLatency	1390	1650	72	8689	1135	74	2963
BCETLatency	0	0	0	0	0	0	0
Latency100	28	26	25	26	26	37	35
IDLatency100	22	18	19	21	20	26	29
WCETLatency100	39	47	34	38	38	51	46
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.37: Durchschnittliche Erkennungslatenzen bei Kombination der Optimierungen (Opt. 4c, Schwellenwert 100)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	6741	2681	780	11298	1678	680	3152
IDLatency	11357	3190	1446	12385	1893	1372	4503
WCETLatency	1260	1655	67	9584	1349	72	1868
BCETLatency	0	0	0	0	0	0	0
Latency100	29	28	26	26	29	38	36
IDLatency100	20	21	20	21	22	27	26
WCETLatency100	41	49	34	38	42	51	47
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.38: Durchschnittliche Erkennungslatenzen bei Kombination der Optimierungen (Opt. 4d, Schwellenwert 125)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	6821	2906	780	11422	1657	683	4304
IDLatency	10597	3582	1446	12551	1853	1379	6217
WCETLatency	2367	1778	67	9610	1359	72	2313
BCETLatency	0	0	0	0	0	0	0
Latency100	27	27	26	25	28	38	34
IDLatency100	19	23	20	20	21	27	24
WCETLatency100	39	40	34	37	43	51	48
BCETLatency100	0	0	0	0	0	0	0

Tabelle A.39: Durchschnittliche Erkennungslatenzen bei Kombination der Optimierungen (Opt. 4e, Schwellenwert 150)

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Latency	6821	2906	780	11422	1774	683	4977
IDLatency	10597	3216	1446	12551	2066	1379	7517
WCETLatency	2367	2397	67	9610	1327	72	2351
BCETLatency	0	0	0	0	0	0	0
Latency100	27	27	26	25	28	38	37
IDLatency100	19	22	20	20	21	27	25
WCETLatency100	39	40	34	37	43	51	53
BCETLatency100	0	0	0	0	0	0	0

A.3 Mehraufwand

Abschließend sollen die Einzelergebnisse zur Ermittlung des benötigten Mehraufwands tabellarisch dargestellt werden. Tabelle A.40 zeigt dazu die Ausführungszeiten der einzelnen Benchmarks (bei fehlerfreier Ausführung) unter Berücksichtigung der jeweiligen Erkennungstechnik und Optimierung. Tabelle A.41 präsentiert die Anzahl der Instruktionen der kompilierten Benchmark-Programme.

Tabelle A.40: Ausführungszeit in Prozessortakten bei fehlerfreier Ausführung

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Ohne Opt.	872554	1389646	175807	1074885	245409	153817	1244664
Ohne Instr.	784744	1141412	154205	917578	233479	147169	1117638
Opt. 1	872554	1389646	175807	1074885	245409	153817	1244664
Opt. 2a bis 2f	828849	1266675	165106	1006690	239057	150493	1182411
Opt. 3a	877640	1300213	175707	1024112	250953	157413	1259152
Opt. 3b	834442	1253034	175007	1025162	245369	153791	1248928
Opt. 3c	834142	1231450	175007	1025162	245321	153791	1245293
Opt. 3d	834142	1224545	175007	1025162	245321	153791	1244846
Opt. 3e	834142	1221925	175007	1025162	245321	153791	1244846
Opt. 4a	831545	1215817	165205	989256	241741	152491	1190090
Opt. 4b	809746	1196068	164706	989046	239040	150481	1184265
Opt. 4c	809146	1187068	164706	989006	239016	150481	1182407
Opt. 4d	809146	1180468	164706	989006	239016	150481	1182047
Opt. 4e	809146	1180468	164706	989006	239016	150481	1182047

Tabelle A.41: Anzahl der Instruktionen der kompilierten Benchmark-Programme incl. eingebundener Standard-Bibliotheken

	aifirf	bitmnp	canrdr	pntrch	puwmod	rspeed	ttsprk
Ohne Opt.	3508	5189	2976	3421	3968	3214	4896
Ohne Instr.	3149	3962	2770	3036	3375	2967	4210
Opt. 1	3508	5189	2976	3421	3968	3214	4896
Opt. 2a bis 2f	3403	4778	2919	3291	3777	3138	4681
Opt. 3a	3568	4837	3006	3438	4013	3279	5215
Opt. 3b	3480	4516	2949	3347	3871	3165	4957
Opt. 3c	3472	4437	2943	3337	3862	3163	4861
Opt. 3d	3468	4423	2941	3336	3859	3161	4824
Opt. 3e	3468	4423	2941	3336	3854	3161	4823
Opt. 4a	3403	4778	2919	3291	3777	3138	4681
Opt. 4b	3375	4328	2903	3259	3696	3113	4710
Opt. 4c	3359	4290	2899	3247	3686	3109	4654
Opt. 4d	3357	4264	2897	3245	3682	3107	4630
Opt. 4e	3357	4264	2897	3245	3680	3107	4628

B

Eigene Veröffentlichungen

Veröffentlichungen als Erstautor

2010 Wolf, J.; Gerdes, M.; Kluge, F.; Uhrig, S.; Mische, J.; Metzlaff, S.; Rochange, C.; Cassé, H.; Sainrat, P.; Ungerer, T.: RTOS Support for Parallel Execution of Hard Real-Time Applications on the MERASA Multi-Core Processor. In: *Proc. of the 13th IEEE Int. Symposium on Object/Component/Service-Oriented Real-time Distributed Computing (ISORC)*, S. 193-201. IEEE Computer Society, 2010. – ISBN 978-0-7695-4037-5

Wolf, J.; Kluge, F.; Guliashvili, I.: Final System-Level Software for the MERASA Processor. *Technical Report 2010-08*. Institute of Computer Science, University of Augsburg, October 2010.

2011 Wolf, J.; Gerdes, M.; Kluge, F.; Uhrig, S.; Mische, J.; Metzlaff, S.; Rochange, C.; Cassé, H.; Sainrat, P.; Ungerer, T.: RTOS Support for Execution of Parallelized Hard Real-Time Tasks on the MERASA Multi-Core Processor. In: *Int. Journal of Computer Systems, Science and Engineering (CSSE)*, Vol. 26, No. 6, 2011. – ISSN 0267-6192

2012 Wolf, J.; Fechner, B.; Uhrig, S.; Ungerer, T.: Fine-Grained Timing and Control Flow Error Checking for Hard Real-Time Task Execution. In: *Proc. of the 7th Int. Symposium on Industrial Embedded Systems (SIES)*, S. 257-266. IEEE Computer Society, 2012. – ISBN 978-1-4673-2685-8

Wolf, J.; Fechner, B.; Ungerer, T.: Fault Coverage of a Timing and Control Flow Checker for Hard Real-Time Systems. In: *Proc. of the 18th Int. On-Line Testing Symposium (IOLTS)*, S. 127-129. IEEE Computer Society, 2012. – ISBN 978-1-4673-2082-5

Wolf, J.; Fechner, B.; Ungerer, T.: Fault Detection Capabilities of an Enhanced Timing and Control Flow Checker for Hard Real-Time Systems. In: *Proc. of the 4th Int. Conference on Advances in System Testing and Validation Lifecycle (VALID)*, S. 57-62. IARIA, 2012. – ISBN 978-1-4673-2082-5

2013 Wolf, J.; Ungerer, T.: An Optimized Timing and Control Flow Checker for Hard Real-Time Systems. In: *Proc. of the 9th Workshop on Dependability and Fault Tolerance (VERFE)*, zur Veröffentlichung akzeptiert, 2013.

Weitere Veröffentlichungen

2008 Kluge, F.; Wolf, J.: Basic System-Level Software for a Single-Core MERASA Processor. *Technical Report 2008-06*. Institute of Computer Science, University of Augsburg, April 2008.

Gerdes, M.; Wolf, J.; Zhang, J.; Uhrig, S.; Ungerer, T.: Multi-Core Architectures for Hard Real-Time Applications. In: *ACACES 2008 Poster Abstracts*, S. 195-198. Academia Press, Ghent, 2008. – ISBN 978-9-0382-1288-3

Kluge, F.; Wolf, J.: Refined System-Level Software for a Single-Core MERASA Processor. *Technical Report 2008-15*. Institute of Computer Science, University of Augsburg, October 2008.

2009 Bagci, F.; Wolf, J.; Ungerer, T.; Bagherzadeh, N.: Mobile Agents for Wireless Sensor Networks. In: *Proc. of the 2009 Int. Conference on Wireless Networks (ICWN)*, S. 502-508. CSREA Press, 2009. – ISBN 1-60132-113-9

Kluge, F.; Wolf, J.: System-Level Software for a Multi-Core MERASA Processor. *Technical Report 2009-17*. Institute of Computer Science, University of Augsburg, October 2009.

2010 Bagci, F.; Wolf, J.; Satzger, B.; Ungerer, T.: UbiMASS – Ubiquitous Mobile Agent System for Wireless Sensor Networks. In: *Proc. of the 3rd Int. Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, S. 245-252. IEEE Computer Society, 2010. – ISBN 978-0-7695-4049-8

Rochange, C.; Bonenfant, A.; Sainrat, P.; Gerdes, M.; Wolf, J.; Ungerer, T.; Petrov, Z.; Mikulu, F.: WCET Analysis of a Parallel 3D Multigrid Solver Executed on the MERASA Multi-Core. In: *Proc. of the 10th Int. Workshop on Worst-Case Execution Time Analysis (WCET)*, S. 90-100. OASICS, Schloss Dagstuhl, 2010. – ISBN 978-3-939897-21-7

Ungerer, T.; Cazorla, F. J.; Sainrat, P.; Bernat, G.; Petrov, Z.; Cassé, H.; Rochange, C.; Quinones, E.; Uhrig, S.; Gerdes, M.; Guliashvili, I.; Houston, M.; Kluge, F.; Metzlaff, S.; Mische, J.; Paolieri, M.; Wolf, J.: MERASA: Multi-Core Execution of Hard Real-Time Applications Supporting Analysability. In: *IEEE Micro 2010, Special Issue on European Multicore Processing Projects*, Vol. 30, No. 5, Sept./Okt. 2010. IEEE Computer Society, 2010. – ISSN 0272-1732

2011 Paolieri, M.; Quinones, E.; Cazorla, F. J.; Wolf, J.; Ungerer, T.; Uhrig, S.; Petrov, Z.: A Software-Pipelined Approach to Multicore Execution of Timing Predictable Multi-threaded Hard Real-Time Tasks. In: *Proc. of the 14th IEEE Int. Symposium on Object/Component/Service-Oriented Real-time Distributed Computing (ISORC)*, S. 233-240. IEEE Computer Society, 2011. – ISBN 978-0-7695-4368-0

Gerdes, M.; Wolf, J.; Guliashvili, I.; Ungerer, T.; Houston, M.; Bernat, G.; Schnitzler, S.; Regler, H.: Large Drilling Machine Control Code – Parallelisation and WCET Speedup. In: *Proc. of the 6th Int. Symposium on Industrial Embedded Systems (SIES)*, S. 91-94. IEEE Computer Society, 2011. – ISBN 978-1-61284-818-1

Weis, S.; Garbade, A.; Wolf, J.; Fechner, B.; Mendelson, A.; Giorgi, R.; Ungerer, T.: A Fault Detection and Recovery Architecture for a Teradevice Dataflow System. In: *Proc. of the 1st Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*, S. 38-44. IEEE Computer Society, 2011. – ISBN 978-1-4673-0709-3

C

Lebenslauf

Persönliche Daten

Name:	Julian Christoph Wolf
Geburtsdatum:	7. April 1982
Geburtsort:	Landsberg am Lech

Ausbildung

1988 - 1992	Albert-Schweitzer-Grundschule Ettringen
1992 - 2001	Joseph-Bernhart-Gymnasium Türkheim, Abschluss: Abitur
2001 - 2007	Studium der Angewandten Informatik mit Nebenfach Betriebswirtschaftslehre an der Universität Augsburg, Abschluss: Diplom-Informatiker
	Thema der Diplomarbeit: Entwicklung und Implementierung eines mobilen Agentensystems für Sensornetze

Wissenschaftliche Tätigkeit

seit 2007	Wissenschaftlicher Angestellter am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme an der Universität Augsburg