# Software-basierte Rekonfiguration in statisch geplanten Mehrkernsystemen zur Behandlung permanenter Fehler

**Von der Fakultät für Mathematik, Naturwissenschaften und Informatik der Brandenburgischen Technischen Universität Cottbus-Senftenberg**

**zur Erlangung des akademischen Grades**

**Doktor der Ingenieurwissenschaften (Dr.-Ing.)**

**genehmigte Dissertation vorgelegt von**

Diplom-Informatiker
Sebastian Müller
geboren am 21.2.1980 in Cottbus

Gutachter: Prof. Dr.-Ing. Heinrich Theodor Vierhaus
Gutachter: Prof. Dr. rer. nat. habil. Petra Hofstedt
Gutachter: Prof. Dr.-Ing. Thomas Holstein

Tag der mündlichen Prüfung: 16.12.2014

# Inhaltsverzeichnis

# 1. Einleitung

## 1.1. Motivation

Digitale Rechensysteme sind aus unserer Gesellschaft nicht mehr wegzudenken und durchziehen die verschiedensten Bereiche unseres Lebens. Wohingegen Desktop-PCs, Laptops und Tablets noch die offensichtlicheren Vertreter darstellen, ist der Großteil der digitalen Systeme als eingebettetes System verborgen in unzähligen Geräten zu finden. Bei Geräten im Endverbrauchermarkt, wie zum Beispiel zur Kommunikation, Informationsbereitstellung oder Geräten zur Erleichterung des täglichen Lebens, bestimmen Anforderungen, wie günstiger Preis und hohe Aktualität, die Eigenschaften des Gerätes. Gerade die Forderung nach hoher Aktualität führt dazu, dass Geräte in immer kürzeren Abständen durch ein Gerät der neueren Generation ersetzt werden, weshalb die Langlebigkeit bzw. die Zuverlässigkeit von Geräten in diesem Marktsegment eine untergeordnete Rolle spielt.

Ein anderes Bild ergibt sich bei der Betrachtung von eingebetteten Systemen im industriellen Bereich, der medizinischen Versorgung oder Geräten in der gesellschaftlichen Infrastruktur, wie Transport und Energieversorgung. In diesen Bereichen hängen von der Zuverlässigkeit der eingesetzten digitalen Systeme hohe volkswirtschaftliche Summen ab bis hin zur Gefährdung von Menschenleben. Solch sicherheitskritische Systeme zeichnen sich darüber hinaus durch weitere Eigenschaften aus, wie zum Beispiel, dass sie langlaufend und gegebenenfalls im Fehlerfall nicht leicht zugänglich sind. In der Vergangenheit gab es immer wieder Zwischenfälle, bei denen solche Systeme versagt haben. Ein prominentes Beispiel aus der jüngeren Zeit ist die Marssonde Curiosity, bei der das System in einen Notlaufmodus wechseln musste, nachdem Defekte in der Elektronik aufgetreten sind. Neben hochzuverlässigen Systemen für Space- und Avionikanwendungen zählt auch die Elektronik im Fahrzeugmarkt zu den sicherheitskritischen Anwendungen. Inzwischen wird im Fahrzeugmarkt von einer steigenden Tendenz an Ausfällen berichtet, die ausschließlich auf den Ausfall elektronischer Baugruppen zurückgeführt werden.

Der stetig steigende Bedarf an leistungsfähigen und effizienten digitalen Systemen führt regelmäßig zu einer Verbesserung in der Halbleitertechnologie. Dadurch wird regelmäßig eine Skalierung der Fertigungsgrößen erreicht, wodurch eine gesteigerte Integrationsdichte an Transistoren bezüglich der Chipfläche ermöglicht wird. Jedoch wird für zukünftige hochintegrierte Systeme im unteren Nanometer-Bereich eine sinkende Zuverlässigkeit vorhergesagt [5, 55]. In der Arbeit von Srinivasan et.al [94] wird dargestellt, dass sich für Prozessoren, welche in der Technologie von 180 nm auf 64 nm skaliert werden, für bestimmte Fehlereffekte eine bedeutsame Erhöhung der Fehleranfälligkeit ergibt.

Eine Skalierung der Fertigungstechnologie bedeutet, dass sich der Querschnitt eines Transistors, das aufgebrachte Gate-Oxid, die Verbindungsstrukturen und die Isolationsschichten zwischen den leitenden Teilen verringern. Eine Verkleinerung der Strukturen auf den nächst kleineren Fertigungsschritt stellt immer hohe Anforderungen an die Fertigungstechnologie. Allerdings ergeben sich für solch kleine Strukturen Schwankungen in der Fertigung bei der Lithographie und der Dotierung, weshalb die idealen Parameter eines Transistors schwer einzuhalten sind und sie somit in ihren Eigenschaften variieren [11]. Darüber hinaus skaliert die Reduzierung der Strukturgrößen nicht ideal für weitere Schaltungsparameter wie Versorgungsspannung und Verlustströme. Dies führt zu einer ungünstigen thermischen Entwicklung und zu einer Verstärkung von bestimmten Alterungseffekten. Erschwerend wirkt, dass in kleiner werdenden Strukturen höhere elektromagnetische Stressbelastungen auftreten, die in Kombination mit der erwähnten gesteigerten thermischen Belastung Fehlereffekte potenzieren. Daraus ergibt sich eine Degradation der Parameter eines Transistors während seiner Lebenszeit bis hin zu einer deutlichen Abweichung von der Spezifikation und dem Ausfall [15]. Insgesamt bedeutet dies, dass zukünftige digitale Systeme Probleme in der Zuverlässigkeit und Lebensdauer haben werden, aufgrund der genannten Effekte durch Verbesserungen in der Halbleitertechnologie. Zusammenfassend kann somit festgehalten werden:

- In unserer Gesellschaft besteht ein steigender Bedarf an hochzuverlässigen Systemen und
- Zukünftige integrierte Schaltkreise werden eine geringere Zuverlässigkeit aufweisen.

Um diesen Umständen Rechnung zu tragen, wird es in Zukunft erforderlich sein Techniken einzusetzen, mit deren Hilfe hochzuverlässige digitale Systeme aus nicht zuverlässigen Komponenten gebaut werden können. Die vorliegende Dissertation beschäftigt sich mit diesem Schwerpunkt und präsentiert software-basierte Techniken, um diese in digitalen Systemen zur Gewährleistung einer höheren Systemzuverlässigkeit einzusetzen.

## 1.2. Zielsetzung und Abgrenzung

In dieser Arbeit werden Techniken entwickelt, mit deren Hilfe ein eingebettetes System auf den dauerhaften Ausfall einer Prozessorbaugruppe reagieren kann. Zur Behandlung der ausgefallenen Komponente erfolgt eine Rekonfiguration des Systems, wobei die ausgeführte Anwendung in der Form angepasst wird, dass die Verwendung der ausgefallenen Baugruppe durch die Anwendung vermieden wird. Als Ersatz kann eine redundante Gruppe dienen oder, falls dies nicht möglich ist, wird eine verminderte Verarbeitungsleistung des Systems akzeptiert. Der Vorgang zur Rekonfiguration des Systems erfolgt durch eine Software-Routine, die selbst auf dem System ausgeführt wird. Zur Anpassung der Anwendung sind daraufhin Techniken erforderlich, wie sie typischerweise in Compilern zur Zielcodeerzeugung eingesetzt werden.

Die Verbesserungen in der Fertigungstechnologie und die dadurch höhere Integrationsdichte für Halbleiterelemente führen langfristig zu massiv parallelen Systemen, die

auch einen heterogenen Charakter haben können [12]. Aus diesem Grund ist der Prozess zur Rekonfiguration hierarchisch organisiert, um die steigende Systemkomplexität moderner eingebetteter Systeme besser beherrschen und ausnutzen zu können. Solche Systeme bieten Redundanz auf verschiedenen Systemebenen, die von einzelnen Prozessorbaugruppen bis hin zu vollständigen Prozessorkernen reicht.

Ein Schwerpunkt der Arbeit liegt in der Organisation und Administration des Rekonfigurationsprozesses. Die Anpassung des Systems erfolgt zum Systemstart und nicht nebenläufig zur normalen Systemausführung. Der Test sowie die Diagnose des Systems sind eng gekoppelt an eine Behandlung ausgefallener Baugruppen. Allerdings zählen diese Punkte nicht zum Umfang der in der Arbeit entwickelten Verfahren. In einem entsprechenden Kapitel zum Stand der Technik werden zur Verfügung stehende Möglichkeiten diskutiert.

Als Basisarchitektur der Untersuchungen dient ein statisch geplantes Prozessormodell. Es bietet die Möglichkeit der parallelen Ausführung von Aufgaben, die allerdings explizit im Programmcode der Anwendung ausgedrückt werden muss. Die Architektur verwendet demnach keine dynamische Aufgabenplanung, im Gegensatz zu einer dynamisch geplanten super-skalaren Architekturen moderner Desktop- und Server-Prozessoren.

## 1.3. Struktur der Arbeit

Im folgenden zweiten Kapitel werden zunächst die relevanten Grundlagen des Themengebiets vorgestellt. Anschließend erfolgt eine Diskussion der verwandten Arbeiten. In Kapitel 3 werden die methodischen Grundlagen definiert und es wird das allgemeine Vorgehen zur Systemrekonfiguration beschrieben. Das vierte Kapitel präsentiert die entwickelten Techniken, die auf lokaler Ebene innerhalb eines Prozessorkerns eingesetzt werden. Die eingesetzten Verfahren in einem globalen Kontext beschreibt das Kapitel 5. Im sich anschließenden Kapitel 6 erfolgt die Präsentation der Untersuchungen und der Ergebnisse. Die Arbeit endet mit einer zusammenfassenden Diskussion im Kapitel 7.

Die folgenden Notationen werden in der Arbeit verwendet. *Kursive Begriffe* heben prägende Eigennamen für Begriffe hervor, die einen entsprechenden Sachverhalt repräsentieren. Darüber hinaus werden allgemeine *Bezeichner* kursiv hervorgehoben. Begriffe, die in breiter **Schrift** stehen, signalisieren, dass der entsprechende Begriff informell eingeführt wird. In den Beschreibungen zu den einzelnen Algorithmen sind bestimmte Notationen aus der Sprache C übernommen, welche nicht den gängigen Formalismen für Pseudocode entsprechen. Dazu zählen die Inkrement- und Dekrementoperation sowie Feldzugriffe und der Zugriffsoperator für Strukturen. Die Verwendung von Feldzugriffen erfolgt für Listen bzw. Folgen, welche implizit als Feld implementiert werden können. Der Punktoperator wird eingesetzt, um einzelne Komponenten eines zusammengesetzten Typs auszuwählen.

# 2. Stand der Technik

Das Gebiet der Zuverlässigkeit digitaler Schaltungen ist seit Jahrzehnten Gegenstand der Forschung. Es existiert eine breite Basis an etablierten Methoden und standardisierten Techniken in dem Themenkomplex, welche im folgenden Kapitel dargestellt werden.

Zunächst werden in diesem Kapitel grundlegende Begrifflichkeiten des Themengebietes vereinbart, analog zu der Herangehensweise in [71]. Anschließend werden Verfahren besprochen, die zur Behandlung kurzzeitig auftretender Fehler eingesetzt werden. Im darauffolgenden Abschnitt werden Fehler diskutiert, die dauerhaft im System vorhanden sind, und es werden Methoden präsentiert, wie diese Fehler kompensiert werden können. Der nächste Abschnitt beschäftigt sich mit der Behandlung von Fehlern in Mehrkern-Systemen und stellt verwandte Arbeiten zum Schwerpunkt der vorliegenden Dissertation vor. Das Kapitel endet mit einem Abschnitt zur Darstellung von Techniken zur Erkennung und der Diagnose von Fehlern.

## 2.1. Grundlegende Begriffe

In der Literatur werden grundsätzlich drei Begriffe unterschieden: Fehler, Störung und Ausfall. Dabei besteht eine Abhängigkeit zwischen den Begriffen bezüglich ihrer Verursachung. Ein Fehler kann eine Störung verursachen, die wiederum einen Ausfall hervorrufen kann. Der Begriff **Fehler** (engl. fault) bezeichnet einen physikalischen Effekt in der vorhandenen Hardware eines Systems. Dieser Effekt kann bewirken, dass die Hardware kurzzeitig oder auch dauerhaft gestört ist. Ein Fehler kann sich in einer **Störung** (engl. error) manifestieren. Darunter versteht man eine Aktivierung des Fehlers in der Form, dass eine Abweichung von der Korrektheit eintritt. Kann eine Störung außerhalb des Systems beobachtet werden, wird dies als **Ausfall** (engl. failure oder malfunction) bezeichnet. Das System verhält sich dann abweichend von dem erwarteten Verhalten und erbringt eine Anforderung zeitverzögert, in geminderter Qualität oder auch gar nicht.

Die eingesetzten Techniken, um den Ausfall eines Systems zu vermeiden, können allgemein in drei Formen unterschieden werden: Fehlervermeidung, Fehlermaskierung und Fehlertoleranz.Die Verfahren kommen auf unterschiedlichen Ebenen zum Einsatz, wie zum Beispiel auf Technologieebene, der Schaltkreisebene oder der Systemebene [21]. Das Ziel der **Fehlervermeidung** ist es, mittels technologischer Mittel dem Auftreten eines Fehlers vorzubeugen durch zum Beispiel Abschirmung eines Schaltkreises gegen Strahlung. In [41] ist eine umfangreiche Übersicht zu Verfahren auf Technologieebene zu finden. Unter dem Begriff **Fehlermaskierung** werden Techniken zusammengefasst, welche die Ausbreitung einer Störung im System vermeiden. Der Begriff **Fehlertoleranz** beschreibt die Fähigkeit eines Systems, beim Auftreten eines Fehlers

ohne Abweichung von der Systemspezifikation weiterzuarbeiten und den Systemausfall zu verhindern. Die in dieser Dissertation entwickelten Verfahren fallen in das Gebiet der Fehlertoleranz. Der allgemeine Begriff der **Zuverlässigkeit** (engl. dependability) umschreibt eine Menge von Qualitätsmerkmalen, die als Anforderung an ein System spezifiziert werden. Zu den Merkmalen zählen unter anderem die Zuverlässigkeitsfunktion, die Verfügbarkeit eines Systems, die Sicherheit, die Leistungsfähigkeit, die Wartbarkeit und die Testbarkeit.

### 2.1.1. Klassifizierung von Fehlern und Redundanzstrategien

Eine wichtige Eigenschaft, nach der Fehler unterschieden werden, ist die Dauer und Art ihres Auftretens, also die Zeit, in der ein Fehler aktiv ist. Ein **permanenter** Fehler ist dauerhaft im System vorhanden. Er wird typischerweise durch einen physikalischen Defekt aufgrund von Herstellungsschwankungen, erhöhter Stressbelastung und dadurch bedingter Alterung verursacht. Das Auftreten **transienter** Fehler erfolgt dynamisch innerhalb kurzer Zeit, wobei der Fehler im Allgemeinen nach dem Verschwinden nicht wieder auftritt. Neben hoch energetischer Strahlung [107, 92, 97] werden auch andere Fehlerquellen als Ursache für transiente Fehler angeführt, wie zum Beispiel Interferenzen zwischen den Leitungen, Schwankungen der Versorgungsspannung, Leitungsrauschen und Schwankungen in der Herstellungsqualität eines Gatters. Die dritte Art sind **intermittierende** Fehler, die wiederholt auftreten und auch wieder verschwinden. Dies kann mit einer hohen Frequenz erfolgen. Es handelt sich dabei um eine Zwischenform der beiden zuvor genannten extremen Erscheinungen. In Abhängigkeit von der Frequenz der wiederholten Aktivität des Fehlers kann dieser dem transienten oder auch dem permanenten Charakter eines Fehlers ähneln. Die Ursachen für intermittierende Fehler sind ähnlich denen für permanente Fehler [72]. Deshalb werden sie im weiteren Verlauf dieser Dissertation nicht weiter gesondert besprochen, da sie darüber hinaus häufig auch zu permanenten Fehlern führen, welche den Schwerpunkt dieser Arbeit darstellen.

Um ein System fehlertolerant auszulegen, ist es erforderlich, eine bestimmte Form von **Redundanz** einzusetzen. Redundanz bedeutet, etwas zusätzlich in ein System einzubringen, was bei einem Verzicht darauf keine Auswirkung auf das Systemverhalten hat. Es werden drei Formen von Redundanz unterschieden. Der Begriff der **Zeit-Redundanz** bedeutet, dass bestimmte Aufgaben wiederholt ausgeführt werden in der Hoffnung, dass der Fehler bei der Wiederholung nicht mehr aktiv ist. Diese Strategie wirkt sich negativ auf die Leistungsfähigkeit eines Systems aus und ist, ungeeignet permanente Fehler zu kompensieren. Die **Hardware-Redundanz** zielt darauf ab, eine ausgefallene Baugruppe durch eine andere Baugruppe zu ersetzen. Dabei handelt es sich häufig um eine Reserve-Gruppe. Daraus resultiert ein gewisser Mehraufwand an notwendiger Chipfläche und auch ein erhöhter Stromverbrauch bei dauerhaft aktiven Reserve-Komponenten. Techniken, die permanente Fehler behandeln, setzen typischerweise eine Form von Hardware-Redundanz ein. Als dritte Form ist die **Informations-Redundanz** bekannt. Hierbei werden die Daten mit zusätzlichen Informationen angereichert, um daraus später Rückschlüsse über einen aufgetretenen Fehler ziehen zu können. Zudem existieren noch software-basierte und hybride Konzepte. Für alle Verfahren

gilt, dass je nach Art des Auftretens eines Fehlers eine bestimmte Redundanzstrategie besser oder weniger gut zur Kompensation geeignet ist.

### Zeit-Redundanz

Die Verfahren zur Zeit-Redundanz nutzen das zeitliche Verhalten transienter Fehler aus, um diese zu erkennen und zu behandeln. Das allgemeine Schema basiert auf der wiederholten Ausführung einer Berechnung mit dem Ziel, dass ein Fehler nur während einer Berechnung aktiv ist. Da dies nicht für permanente Fehler der Fall sein kann, ist diese Redundanzstrategie für sich allein nicht zur Behandlung permanenter Fehler geeignet.

Die Auswahl der zu wiederholenden Aufgabenteile kann unterschiedlich granular erfolgen. Als sehr feingranular ist die Wiederholung auf Befehlswort-Ebene anzusehen. Andere Verfahren arbeiten auf Grundlage von Codesegmenten, wie sie zum Beispiel durch Basisblöcke (vgl. Abschnitt 3.2) beschrieben sind. Eine weitere Granularitätsstufe bilden vollständige Programme, die wiederholt auszuführen sind.

Das allgemeine Vorgehen lässt sich wie folgt beschreiben [36, 39]. Zunächst werden, bezüglich der Programmabarbeitung in bestimmten Abständen Speicherpunkte definiert. Ein solcher Speicherpunkt wird auch als **Checkpoint** bezeichnet. Dabei handelt es sich um ein Abbild des Zustands eines Prozesses $P$ zu einem Zeitpunkt $t$ mit den notwendigen Informationen, um $P$ von $t$ aus erneut starten zu können. Die Berechnungen zwischen zwei Speicherpunkten müssen in geeigneter Form fehlererkennend ausgelegt sein. Dies kann durch wiederholtes Ausführen mit anschließendem Vergleich oder auch mit Coding-Verfahren erfolgen, die im folgenden Abschnitt besprochen werden. Bei einem Unterschied im Ergebnis wird die Berechnung verworfen und der zuletzt gespeicherte Systemzustand geladen. Dieser Vorgang wird als **Rollback** bezeichnet. Die Berechnungen werden nun wiederholt ausgeführt. Es wird davon ausgegangen, dass ein transienter Fehler während der Wiederausführung nicht mehr aktiv ist. Sollten mehrere Wiederholungen fehlschlagen, kann das System gestoppt werden, da davon ausgegangen wird, dass ein permanenter Fehler vorliegt.

Um die Entwicklungszeit eines Systems mit einer zeitbasierten Redundanzstrategie zu beschleunigen, wurden automatisierte Erzeugungen während der Übersetzung einer Anwendung präsentiert [52]. Darüber hinaus wurden erste Integrationsversuche in den Entwurfsprozess in [66, 68] präsentiert. Ein aufbauender Forschungsschwerpunkt beschäftigt sich mit der Optimierung der Checkpointing-Strategie [65]. Dabei steht die Frage im Mittelpunkt, an welchen Stellen der Systemzustand gesichert werden soll, um ein optimales Verhältnis zwischen Speicheraufwand und Häufigkeit der Speicherung zu finden.

In [27] wird ein hardware-basiertes dynamisches Verfahren beschrieben, um Fehler zur Laufzeit zu erkennen. Das Verfahren arbeitet dabei auf einer Granularität von Instruktionswörtern. Der Laufzeit-Scheduler wird so erweitert, dass alle geholten Operationen automatisch dupliziert und in die Warteschlange eingereiht werden. Die duplizierte Operation wird für eine andere als die ursprüngliche Ausführungseinheit geplant. Nach der Berechnung beider Operationen kann durch einen Vergleich der Werte ein möglicher Fehler erkannt werden. Das Verfahren ist beschränkt auf superskalare dynamisch geplante Prozessoren und beschreibt keine Möglichkeit zur Reparatur im Fehlerfall.

Für Systeme, die aus Kostengründen keine komplexen Prozessoren bereitstellen, wird die Administration des Checkpointing auf der Anwendungsebene verbleiben. Mit den Verbesserungen in der Fertigungstechnologie ist es auch in eingebetteten Systemen praktikabel, komplexere Prozessoren einzusetzen, mit denen thread-basierte[1] Multi-Tasking[2] Systeme umsetzbar sind. In solchen Systemen kann auf die eingebauten Mechanismen zur parallelen Ausführung von Aufgaben zurückgegriffen werden, um eine Fehlertoleranzstrategie in der zeitlichen Dimension zu implementieren. Die Organisation kann dabei auf Kernel-Level[3] oder als User-Level Thread erfolgen. In [75, 101] sind entsprechende Verfahren vorgestellt, welche das *Simulteanous Multi-Threading* ausnutzen, um transiente Fehler zu behandeln.

Zusammenfassend lässt sich festhalten, dass eine Strategie auf Basis zeitlicher Redundanz dann von Vorteil ist, wenn der Kostenfaktor Zeit in einem System eine untergeordnete Rolle spielt im Vergleich zu den Hardwarekosten anders implementierter Redundanzstrategien. Eine Redundanzstrategie, die ausschließlich in der zeitlichen Dimension arbeitet, bietet keinen Schutz gegen permanente Fehler. Ein entscheidender Punkt beim Einsatz einer Fehlertoleranzstrategie auf zeitlicher Basis ist die Absicherung der Speichervorgänge des Systemzustands gegen korrupte Daten.

#### Informations-Redundanz

Informations-Redundanz ist eine etablierte Redundanzstrategie zur Absicherung von Speichern gegen Fehler [47, 37]. Diese Form der Redundanz wird auch für Verbindungsstrukturen [80] und Kommunikationsmedien [93] verwendet. Weiterhin können auch Zustandsautomaten, wie zum Beispiel der Kontrollpfad eines Prozessors, auf Fehler überwacht werden [48]. Die Verfahren zur Informations-Redundanz bieten die Möglichkeit, transiente und permanente Fehler zu erkennen. Es gibt Verfahren, die zusätzlich eine Fehlerkorrektur durchführen. Die bekanntesten Techniken zur Informations-Redundanz nutzen fehlererkennende und fehlerkorrigierende Codes. Diese erweitern die Nutzdaten mit zusätzlichen Informationen oder transformieren die Nutzdaten in eine andere Darstellungsform.

Ein **Code** ist eine bestimmte Repräsentation von Informationen oder Daten, die mit Hilfe vordefinierter Regeln gebildet wird. Ein **Codewort** ist eine Folge von Zeichen, die eine Information darstellen, welches mit Hilfe eines Codes erzeugt wurde. Sind die verwendeten Symbole die Zahlen 0 oder 1, dann spricht man auch von einem **Binären Code**. Ein Codewort ist gültig, wenn es alle Regeln eines Codes erfüllt. Andernfalls handelt es sich um ein ungültiges Codewort. Das *Codieren* beschreibt den Vorgang, aus einem Datum ein gültiges Codewort mit den vorhandenen Regeln eines Codes zu bilden. Das Decodieren bezeichnet den inversen Vorgang zum Codieren, also das Rekonstruieren des Datums aus einem Codewort.

[1] Ein Thread fasst einen Steuerfluss und die notwendigen Daten zur Ausführung zusammen.

[2] Multi-Tasking bezeichnet die Illusion von paralleler Aufgabenverarbeitung in einem System mit einem Rechenwerk.

[3] Als Kernel wird der zentrale Bestandteil eines Betriebssystems bezeichnet, der alle notwendigen Informationen, Daten und Aufgaben für die relevanten Betriebssystemdienste organisiert und administriert.

Der allgemeine Begriff der **Hamming-Distanz** für binäre Wörter beschreibt die Anzahl an Bitpositionen, in denen sich zwei Wörter unterscheiden. Ein wichtiger Begriff für binäre Codes ist die **Distanz**. Sie gibt die minimale Hamming-Distanz an, bezüglich derer sich zwei Code-Wörter unterscheiden. Diese Eigenschaft ist entscheidend für die Fehlererkennung und -korrektur bezüglich der Codewörter eines Codes. Eine nächste Eigenschaft kann sein, ob ein Code **seperierbar** ist, geschrieben als $(n, k)$-Code. Ein Codewort der Länge $n$ besteht aus zwei Teilen. Die zwei Teile sind die eigentlichen Daten mit $k$ Bit und die zusätzlichen Zeichen $n - k$ Bit. Die zusätzlichen Zeichen werden auch als Code- oder Check-Bits bezeichnet. Der Vorteil eines $(n, k)$-Codes ist, dass die $k$ Datenbits direkt lesbar sind. Ein nicht separierbarer Code erfordert hingegen eine aufwändigere Decodierung zur Rekonstruktion der ursprünglichen Nutzdaten.

Einer der einfachsten Codes ist der *Parity-Code*. Es handelt sich um einen $(k+1, k)$-Code, der jedes Datenwort um ein Code-Bit erweitert. Das Code-Bit wird für ein Datenwort auf 0 gesetzt, wenn die Anzahl der Einsen im Datenwort gerade ist. Andernfalls wird das Code-Bit auf 1 gesetzt. Der Parity-Code bietet die Möglichkeit, Einzelbitfehler zu erkennen, ohne diese korrigieren zu können. Eine Erkennung von Mehrfachfehlern ist nicht möglich. Mit dem Einführen zusätzlicher Code-Bits ist auch eine Fehlerdiagnose durchführbar, weil dadurch eine überlappende Paritätsbildung bezüglich einer Zerlegung des Datenwortes in verschiedene Bitgruppen ermöglicht wird.

Der *Berger-Code* ist ein $(n, k)$-Code mit einem einfachen Codierungsverfahren. Die Check-Bits werden anhand der vorkommenden Einsen eines jeden Datums gebildet. Die notwendige Anzahl an Checkbits hängt von der Länge $k$ des Datenwortes ab. Der Wert von $k$ ergibt sich aus $k = \lceil \log(n + 1) \rceil$. Mit dem Berger-Code können alle unidirektionalen Mehrfachfehler in einem Codewort erkannt werden.

Einer der bekanntesten Codes ist der *Hamming-Code*. Das Verfahren basiert auf einer Erweiterung der überlappenden Paritätsbildung und bietet die Möglichkeit der Fehlererkennung und -korrektur. Das grundlegende Verfahren geht dabei so vor, dass die Datenbits in Paritätsgruppen unterteilt werden und für jede Gruppe ein Check-Bit eingeführt wird. Die Datenbits werden dabei überlappend den Gruppen zugeteilt. Die Zuteilung ist so zu wählen, dass ein fehlerhaftes Datenbit eindeutig identifizierbar ist anhand der beeinflussten Check-Bits. Das grundlegende Verfahren zum Hamming-Code bietet eine Erkennung und Korrektur für einen aufgetretenen Fehler. Ein solcher Code wird auch **SEC**-Code (engl. Single-Error-Correction) genannt. Bei einem zusätzlichen Bitfehler kann der aufgetretene Fehler nicht mehr sicher erkannt werden. Eine Erweiterung des Hamming-Codes erfordert ein weiteres Check-Bit, um im Falle eines 2-Bit-Fehlers eine Erkennung zu gewährleisten. Ein Code, der einen Einzelfehler korrigieren und 2-Bit-Fehler sicher erkennen kann, wird als **SEC-DED**-Code (engl. Single-Error-Correction and Double-Error-Detection) bezeichnet.

*Arithmetische Codes* werden eingesetzt, um arithmetische Operationen zu überprüfen. Das grundsätzliche Vorgehen dabei ist, die auszuführende arithmetische Operation zweimal durchzuführen in Kombination mit einer Codierung der Ein- bzw. Ausgabe. Am Ende werden die Ergebnisse verglichen. Bei der ersten Berechnung wird das Resultat der arithmetischen Operation codiert. Zur zweiten Berechnung werden zuerst die Eingabedaten codiert und anschließend erfolgt die Ausführung der arithmetische Operation. Die Codierungsfunktion $A$ eines arithmetischen Codes hat die Eigenschaft

$A(a * b) = A(a) * A(b)$, wobei $*$ die arithmetische Operation ist und $a$,$b$ die Operanden sind. Wichtig ist, dass das Verfahren invariant zur arithmetischen Operation ist, damit die geforderte Eigenschaft gilt.

Es existieren noch weitere Code-Verfahren ,die nur erwähnt werden sollen, wie zum Beispiel *m-of-n Codes*, *Check-Summen* und *Cyclic Codes*, auch bekannt als *CRC*-Verfahren. Eine Übersicht zu den verschiedenen Coding-Verfahren ist in [71] zu finden.

### Hardware-Redundanz

Die am häufigsten anzutreffende Redundanz-Strategie ist die Hardware–Redundanz, da sie einerseits einsetzbar, zur Behandlung temporärer sowie permanenter Fehler und andererseits oft auch Bestandteil kombinierter Ansätze ist. Ziel der Hardware-Redundanz ist es, durch Vervielfältigung von Komponenten bzw. Baugruppen Fehler zu kompensieren. Dazu wird ein ausgewählter Teil der Hardware mehrfach ins System integriert. Die Wahl der zu replizierenden Komponenten kann unterschiedlich granular erfolgen. Die möglichen Granularitäten reichen vom Gate-Level über das Modul- und Chip-Level bis hin zum Board-Level. Auf dem Gate- und Modul-Level werden Funktionalitäten einer Schaltung sehr fein-granular repliziert. Auf dem Chip-Level können Prozessoren mehrfach vorhanden sein und auf dem Board-Level werden ganze Platinen mehrfach eingesetzt.

Die drei zu unterscheidenden Formen der Hardware-Redundanz sind die **statische** (auch passive Hardware-Redundanz genannt), die **dynamische** (auch als aktive Hardware-Redundanz bezeichnet) und die **hybride** Hardware-Redundanz. Die statische Hardware-Redundanz zielt darauf ab, Defekte zu maskieren anstelle sie zu erkennen. Dahingegen arbeitet die dynamische Hardware-Redundanz nach dem Prinzip, einen Defekt zunächst zu erkennen, um ihn anschließend zu behandeln. Eine Kombination beider Ansätze ist die hybride Hardware-Redundanz.

Damit die statische Hardware-Redundanz einen Defekt maskieren kann, ist es erforderlich, dass die redundanten Komponenten immer aktiv arbeiten. Eine bestimmte Berechnung ist dadurch in mehreren Ergebnissen vorhanden. Mittels Mehrheitsentscheid wird dann ein Wert aus der Menge an Ergebnissen bestimmt, der zur Verarbeitung an nachfolgende Stufen weitergegeben wird. Dadurch kann ohne Verzögerung das korrekte Ergebnis bereitgestellt werden. Dieses Verfahren ist alternativlos für harte Echtzeitsysteme, bei denen keine Zeit zur Rekonfiguration oder zum wiederholten Ausführen vorhanden ist. Diese Redundanzstrategie ist auch für permanente Fehler verwendbar mit dem Verlust der gleichzeitigen Behandlung transienter Fehler. Zusätzlich entfällt die Eigenschaft zur Erkennung transienter Fehler. Der bekannteste Vertreter der statischen Redundanz ist das *Triple Modular Redundancy* (*TMR*). Für ein Modul eines Systems werden drei identische Instanzen angelegt. Aus den Ergebnissen der drei Instanzen bestimmt ein Voter[4] das endgültige Resultat. Beim Auftreten eines Defektes in einer Instanz wird das falsche Ergebnis durch den Voter maskiert. TMR ist ein Spezialfall des allgemeinen N-Modular Redundancy (NMR)-Schemas, bei dem ein Modul $n$-mal repliziert und über $n$ Ergebnisse eine Mehrheitsentscheidung gebildet wird.

[4]Ein Voter bestimmt per Mehrheitsentscheid aus einer Menge an Eingabewerten den Ausgabewert.

Im allgemeinen Schema der dynamischen Hardware-Redundanz ist zur Ausführung einer Aufgabe nur eine Komponente aktiv. Das Schema zielt nicht darauf ab, einen Defekt zu maskieren, sondern erkennt zunächst, dass ein Fehler aktiv wird. Danach wird die Komponente, die den Fehler verursacht, diagnostiziert. Die fehlerbehaftete Komponente wird anschließend aus dem aktiven Betrieb entfernt und durch eine redundante Komponente ersetzt. Die zugewiesene Aufgabe wird danach auf der alternativen Komponente erneut ausgeführt. Für die Erkennung eines Defektes können unterschiedliche Verfahren eingesetzt werden, was sich direkt auf die behandelbaren Fehlerarten auswirkt. Eine erste Form ist es, in regelmäßigen Abständen zu prüfen, ob die aktive Komponente korrekte Ergebnisse liefert. Offensichtlich ist dieser Ansatz weniger für transiente Fehler geeignet, wenn die Abstände zwischen den Prüfzeitpunkten zu groß gewählt werden. Ein anderer Ansatz ist es, zwei Komponenten aktiv rechnen zu lassen und die Ergebnisse zu vergleichen. Mit dieser Methode ist eine Fehlererkennung möglich, jedoch keine Detektion der fehlerhaften Komponenten. Nachdem der Fehler erkannt wurde, erfolgt eine Diagnose, um das fehlerhafte Modul durch eine Reservekomponente zu ersetzen. Im Falle eines transienten Fehlers ist es zunächst sinnvoll, eine wiederholte Berechnung durchzuführen, bevor das System rekonfiguriert wird [19]. Bei den Reservekomponenten kann es sich um kalte oder aktive Baugruppen handeln. Eine kalte Baugruppe befindet sich in einem Wartemodus, solange sie nicht benötigt wird. Für digitale Baugruppen bedeutet dies typischerweise, dass sie nicht mit der Versorgungsspannung verbunden sind. Nachteilig an kalten Spare-Komponenten ist die zusätzliche Zeit, die benötigt wird, um sie für den Betrieb hochzufahren. Der Vorteil liegt darin, dass kalte Baugruppen einem geringeren Einfluss durch Alterungsprozessen unterworfen sind. Im Gegensatz dazu ist eine aktive Reservebaugruppe von Alterungseffekten betroffen und wirkt sich negativ auf den Stromverbrauch des Systems aus. Allerdings besteht jedoch die Möglichkeit, sie bei Bedarf für Diagnosezwecke sofort einsetzen zu können.

Die hybride Hardware-Redundanz vereint die Konzepte zur statischen und dynamischen Hardware-Redundanz. Dadurch kann ein System Defekte maskieren und des Weiteren anhand einer Detektion mit anschließender Rekonfiguration fehlerhafte Komponenten entfernen. Eine schematische Darstellung eines entsprechenden Systems zeigt die folgende Abbildung 2.1.

Das System ist so aufgebaut, dass Defekte zunächst mittels TMR maskiert werden. Zusätzlich gibt es Reserve-Module und Module zur Fehlerdiagnose und Rekonfiguration. Das Ziel der Rekonfiguration ist es, ein Modul, welches aktiv zur Fehlermaskierung erforderlich ist, im Falle eines permanenten Fehlers durch ein Reservemodul zu ersetzen. Dazu leitet das Rekonfigurationsmodul den Ausgang eines Reservemoduls anstelle des Ausgangs eines regulären Moduls an den Voter weiter. Fehlertoleranz-Strategien, die auf einer hybriden Hardware-Redundanz basieren, erfordern den höchsten Hardware-Mehraufwand im Vergleich zu den anderen Verfahren der Hardware-Redundanz. Dafür wird durch hybride Verfahren eine umfangreichere Absicherung gegen Fehler gewährleistet.

#### Software-basierte und hybride Konzepte

Die auf einem System ausgeführte Software kann nicht von Defekten betroffen sein. Allerdings können Systemausfälle durch Software verursachte Fehler auftreten. Das be-

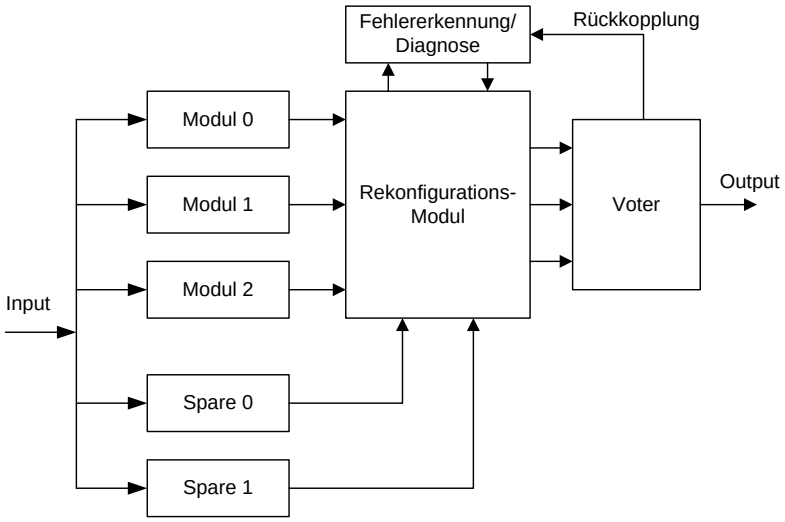


Abbildung 2.1.: Schematische Darstellung eines hybriden Systems mit TMR und 2 Spare-Komponenten (entnommen aus [71])

deutet, dass Fehler im Software-Design bzw. der Implementierung bewirken, dass sich das System entgegen seiner Spezifikation verhält. Eine Möglichkeit, um Implementierungsfehlern zu begegnen, ist das sogenannte *n-Version Programming*. Die Idee dieses Ansatzes ist es, ein bestimmtes Software-Modul mehrfach zu implementieren, was idealerweise durch unterschiedliche Entwicklungsteams erfolgt. Die Software-Module berechnen ihre Ergebnisse zur Laufzeit parallel oder auch sequentiell. Ein Mehrheitsentscheid über die ermittelten Ergebnisse kann Fehler bezüglich der Softwaremodule maskieren.

Um mit Hilfe von Software auf Hardware-Fehler schließen zu können, sind weitere Ansätze bekannt. Zur Indikation eines Hardware-Fehlers können regelmäßige Prüfungen auf im Voraus ermittelte Werte durchgeführt werden. Solche Verfahren werden als Akzeptanztest bezeichnet. Eine weitere Möglichkeit bietet das Auswerten von Performanzwerten, die im aktiven Betrieb des Systems erreicht werden. Mit Hilfe von Profilingtechniken[5] wird die wahrscheinliche Rechenleistung des Systems zur Laufzeit abgeschätzt. Anhand dieser Abschätzung kann eine Über- bzw. Unterschreitung der Rechenleistung als Fehlerindiz interpretiert werden.

Neben diesen Konzepten zur Behandlung von Fehlern in der Software und dem Verfahren zur Erkennung eines Hardware-Fehlers existieren Ansätze zur Fehlertoleranz, welche eine Kombination aus den drei zuvor besprochenen Redundanzstrategien darstellen. Häufig werden dazu Verfahren der Hardware-Redundanz um eine Verschiebung in der zeitlichen Dimension erweitert. In modernen digitalen Systemen wird dabei der Fakt ausgenutzt, dass Prozessorbaugruppen generell redundant vorhanden sind. Das bedeutet, dass Komponenten nicht explizit repliziert werden müssen. Dieser Fall wird

[5]Profiling ist die dynamische Analyse eines Programms zur Bestimmung gewisser Eigenschaften.

auch als **natürliche** Hardware-Redundanz bezeichnet. Superskalare Prozessoren bieten genau diese Form der natürlichen Redundanz bezüglich der Ausführungseinheiten im Datenpfad. Die redundanten Einheiten werden zur Erkennung von Fehlern eingesetzt, indem Operationen mindestens parallel mit anschließendem Vergleich berechnet werden [63, 64, 79]. Im Falle eines unterschiedlichen Ergebnisses kann ein weiteres Ergebnis bestimmt werden, was ein wiederholtes Berechnen erfordert. Zusätzlich ist es auch möglich, schon im Voraus ein drittes Ergebnis zu bestimmen. Die Administration zur Fehlerbehandlung kann dabei in Hardware wie auch in Software erfolgen. Wenn die Fehlererkennung und Fehlerbehandlung in den Programmcode der ausgeführten Anwendung integriert wird, wird dies als **software-basiertes** Verfahren bezeichnet. In [76] wird ein entsprechender Ansatz, basierend auf der EPIC-Architektur von Intel, vorgestellt. Dazu wird der Anwendungscode mit duplizierten Operationen und Vergleichsoperationen erweitert. In [10, 83] wird ein ähnlicher Ansatz für eine allgemeine VLIW-Architektur[6] vorgeschlagen.

### 2.1.2. Fehlermodelle und Abgrenzung

Fehlermodelle wurden eingeführt, um von der physikalischen Ursache eines Fehlers abstrahieren zu können. Das Modell beschreibt einen Fehler, reduziert auf die relevante Eigenschaft, welche für den Umgang mit dem Fehler auf einer höheren Systemebene erforderlich ist. Der bekannteste Vertreter ist das Haftfehler-Modell (engl. stuck-at). Mit diesem wird ausgedrückt, dass der logische Wert einer Verbindung fest auf einem Wert bleibt, welcher der 1 bzw. der 0 entsprechen kann. Das Modell ist gut geeignet, um permanente Fehler zu beschreiben. Mit dem *Bridging*-Fehlermodell wird beschrieben, dass zwei Leitungen miteinander koppeln können und dadurch ein abhängiges Verhalten entsteht. Der Wert einer der beiden Leitungen wird dann vom Wert der anderen Leitung bestimmt und nicht vom eigentlichen Treiber. Das *Bit-Flip*-Modell beschreibt, dass der Wert einer Leitung immer invertiert vom eigentlichen Wert ist. Das *Delay-Fault*-Modell modelliert zusätzlich das zeitliche Verhalten einer Schaltungskomponente und beschreibt Abweichungen von der eigentlichen Schaltgeschwindigkeit in Form von Verzögerungen.

In dieser Dissertation wird von einem konkreten Fehlermodell abstrahiert. Die in dieser Arbeit entwickelten Techniken zur Behandlung permanenter Fehler arbeiten auf einem hohen Abstraktionsniveau bezüglich der Hardware-Komponenten des Systems. Die Verfahren betrachten die Baugruppen eines Prozessors bzw. Baugruppen des Systems. Für die eingesetzte Strategie ist es ausreichend zu wissen, ob eine Baugruppe funktioniert oder nicht einsetzbar ist. Die Baugruppen bzw. die zu unterscheidenden Systemkomponenten werden in Kapitel 3.4 identifiziert. Für den eigentlichen Test auf den Defekt einer Baugruppe muss ein geeignetes Modell gewählt werden. Dieser gehört aber nicht zum Umfang der in der Dissertation erstellten Verfahren.

[6] Eine VLIW-Architektur (engl. *Very Long Instruction Word*) bezeichnet eine superskalare statisch geplante Prozessorarchitektur (vgl. Kap 3.1)

### 2.1.3. Der Begriff der Selbst-Reparatur

Im Zusammenhang mit der Behandlung permanenter Fehler wurde der Begriff der **Selbst-Reparatur** (engl. self-repair) geprägt. Der Teilbegriff Reparatur umfasst in diesem Kontext unter anderem die Deaktivierung einer defekten Hardware-Baugruppe und deren Ersatz durch einen redundanten Baustein, gegebenenfalls mit einer verringerten Systemleistung. Letztere kann sich dadurch ergeben, dass nicht ausreichend redundante Bausteine verfügbar sind, um eine vorhandene parallele Abarbeitung im gleichen Umfang zu gewährleisten wie ursprünglich geplant. Eine Degradation der Systemleistung zur Behandlung von Fehlern wird **graceful degredation** genannt. Der Vorgang, eine Hardware-Baugruppe durch eine redundante Gruppe zu ersetzen, wird auch als **Rekonfiguration** bezeichnet. Wenn ein System eine Rekonfiguration ohne ein externes Eingreifen durchführen kann, dann spricht man im Allgemeinen von einer Selbst-Reparatur, bei der die Rekonfiguration autonom durch das System selbst erfolgt.

Eine Selbst-Reparatur erfordert darüber hinaus noch weitere Tätigkeiten neben der eigentlichen Rekonfiguration. Dazu gehören das Erkennen eines Fehlers und die Diagnose der betroffenen Baugruppe. Nachdem eine defekte Baugruppe diagnostiziert ist, muss sie vom System isoliert werden. Erst danach kann eine Rekonfiguration erfolgen. Abschließend sollte getestet bzw. validiert werden, ob der Fehler im System nicht mehr aktiv ist.

### 2.1.4. Getroffene Annahmen und Voraussetzungen

Die vorliegende Dissertation konzentriert sich ausschließlich auf den Teil einer Reparatur, der sich mit einer autonomen Rekonfiguration eines eingebetteten Systems zur Behandlung permanenter Fehler beschäftigt. Die tatsächliche physikalische Quelle des Fehlers wird dabei nicht entfernt. Mit Hilfe einer Rekonfiguration wird lediglich die Verwendung der defekten Baugruppe vermieden. Die Berechnungen der defekten Baugruppen werden dann durch eine redundante Baugruppe ausgeführt. Zusätzlich kann es erforderlich sein eine Aufgabe auch in der zeitlichen Dimension zu verschieben, wenn nicht genügend parallele Baugruppen verfügbar sind. Daraus ergibt sich dann eine degradierte Systemleistung. Für die in der vorliegenden Arbeit entwickelten Reparaturstrategien gelten folgende weitere Annahmen:

- Es wird kein konkretes Fehlermodell gewählt, da es für die eingesetzten Reparaturverfahren unerheblich ist, durch welchen Effekt eine Baugruppe dauerhaft defekt ist. Die Redundanzstrategie arbeitet bezüglich der Granularität auf Komponentenebene eines Prozessors und auf Systemebene.

- Die Schritte Test, Diagnose und Validierung einer Selbst-Reparatur sind nicht Umfang dieser Arbeit. Der zentrale Punkt ist die Organisation und Administration einer autonomen Rekonfiguration eines eingebetteten Systems zur Behandlung eines bereits erkannten und diagnostizierten permanenten Fehlers.

- Es wird davon ausgegangen, dass auftretende Fehler ein gutartiges Verhalten zeigen. Andernfalls wird gefordert, dass alle Komponenten, auf die sich ein Fehler auswirkt, durch den Test als defekt erkannt werden.

Ein Beispiel für den letzten Punkt ist ein Fehler, der die Codierung einer Operation zu einer Sprungoperation verändert. Dadurch kann zunächst der entsprechende Slot nicht verwendet werden. Zusätzlich ist aber auch die Kontrolllogik als defekt zu erkennen, weil diese Sprünge ausführt, die nicht geplant wurden.

## 2.2. Permanente Fehler

Permanente Fehler führen, im Gegensatz zu transienten Fehlern, zum dauerhaften Ausfall einer elektronischen Komponente. Es ist allgemein anerkannt, dass sich der Lebenszyklus einer digitalen Baugruppe anhand der umgangssprachlich bezeichneten Badewannenkurve beschreiben lässt. Abbildung 2.2 stellt die Entwicklung zweier Kurven für den Lebenszyklus dar. Der Zyklus lässt sich in vier Phasen unterscheiden [94]:

1. Zero-Production Faults sind Defekte, die schon während der Produktion entstehen. Dies wird auch als *Infant-Mortality* bezeichnet.

2. Frühzeitige Ausfälle (engl. Early-Live failure (ELF)) sind Defekte, die ein Gerät kurz nach der Produktion ausfallen lassen. Diese Defekte beruhen in der Regel auf den Ursachen der Zero-Production Faults, die nicht sofort als Fehler aktiv wurden.

3. Phase des normalen Lebenszyklus

4. Alterungsphase (engl. wear-out). Hierbei handelt es sich um Abnutzungs- und Alterungsprozesse, welche digitale Schaltungen ausfallen lassen.

Zero-Production Faults und Frühausfälle sind eine Form von permanenten Fehlern, die in der Literatur zusätzlich als *Hard Errors* unterschieden werden. Als ursächlich werden hier Fehler in der Halbleiterfertigung angegeben. Dazu zählen Schwankungen im Fertigungsprozess (engl. Process Variations), wie Schwächen im Litographi-Prozess und Variationen im Dotierungsvorgang [60]. Zero-Production Faults sollen mittels Fertigungstests abgefangen werden. Allerdings können sich solche Defekte noch nicht in einem beobachtbaren Fehler manifestiert haben. Um eine Aktivierung zu provizieren, wird mittels Burn-In Test ein Schaltkreis unter hohen Stress gesetzt, um Geräte mit Frühausfällen abzufangen. Ziel der Hersteller ist es, Geräte auszuliefern, die sich in der nächsten Phase, dem regulären Lebenszyklus, befinden. In der Phase des normalen Lebenszyklus ist ein elektronisches Gerät hauptsächlich von transienten Fehlern betroffen. In der sich anschließenden Alterungsphase kommen zu den transienten Fehler auch permanente Fehler hinzu. In dieser Phase bewirken Alterungsprozesse und Abnutzungserscheinungen, dass sich Eigenschaften von Transistoren und Leitungen außerhalb der Spezifikation verschieben und am Ende einen Ausfall verursachen. Da sich die Eigenschaften über die Zeit verschlechtern, werden dadurch verursachte permanente Fehler auch als *degredation faults* bezeichnet. Die Behandlung von *degredation faults* ist der Schwerpunkt der vorliegenden Dissertation. Wenn im weiteren Verlauf von permanenten Fehlern gesprochen wird, sind immer implizit *degredation faults* gemeint. Der nächste Abschnitt bespricht kurz die möglichen physikalischen Ursachen für *degredation faults*, wie NBTI, HCI, EM, SM, TDDB und Gate Oxid Breakdown.

Die Kurven in Abbildung 2.2 stellen den Lebenszyklus aktueller und zukünftiger elektronischer Systeme gegenüber. Die Kurve für zukünftige Systeme zeigt, dass die Erkennung von *Hard Errors* aufwendiger wird und die Behandlung von *degredation faults* an Bedeutung gewinnt. Das liegt darin begründet, weil mit fortschreitender Fertigungstechnologie Fertigungsschwankungen und Alterungserscheinungen zunehmen und sich der normale Lebenszyklus eines digitalen System verkürzt [55, 15, 11, 49].

Der Umstand, dass *degredation faults* nur im Feldeinsatz auftreten können, erfordert eine andere Behandlungstrategie im Vergleich zum Umgang mit *hard errors*. Die Verfahren zur Behandlung von *degredation faults* haben gemein, dass sie in geeigneter Weise eine Rekonfiguration des Systems vornehmen. Es existieren unterschiedliche Möglichkeiten, ein System zu rekonfigurieren:

1. Rekonfiguration in FPGAs
2. Strategien für festverdrahtete Logik
3. Anpassung der Ablaufplanung in Software und hybride Verfahren.

Die jeweiligen Verfahren werden im Anschluss an den Abschnitt zu den Ursachen für permanente Fehler in jeweils gesonderten Abschnitten besprochen.

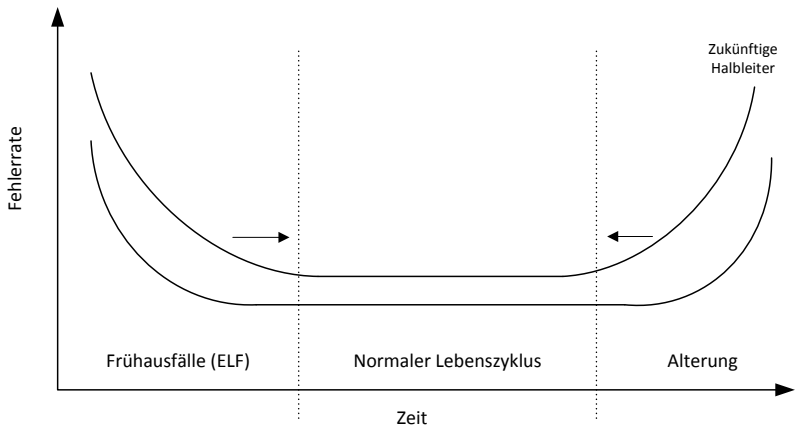


Abbildung 2.2.: Entwicklung des Lebenszyklus für zukünftig hochintegrierte digitale Schaltkreise (entnommen aus [49])

### 2.2.1. Ursachen für permanente Fehler

Es sind verschiedene physikalische Fehlereffekte bekannt, die zu permanenten Ausfällen in Halbleiterstrukturen führen. Diese Effekte treten verstärkt mit der Reduzierung der Fertigungsstrukturen in den unteren Nanometer-Bereich auf. Dabei tritt das Problem auf, dass nicht alle Eigenschaften der Transistoren ideal skalieren [94], wie zum Beispiel die Versorgungs- und Verlustspannung. In Kombination mit den verringerten

Strukturen ergeben sich so höhere Stromdichten, die in Kombination mit einer höheren Temperatur bestimmte Fehlereffekte begünstigen. Die Fehlereffekte, die zu permanenten Defekten führen, können, bezüglich ihres Auftretens in zwei Formen unterschieden werden:

- Veränderung der Eigenschaften von Verbindungsstrukturen,
- Degradation der Transistoreigenschaften und
- Zerstörung eines Transistors.

Die Veränderung der Transistoreigenschaften kann sich auf die Verlustspannung, die Schwellspannung und die Schaltgeschwindigkeit auswirken. Auf den Verbindungsleitungen kann sich zunächst ein geänderter Widerstand einstellen. Darüber hinaus können Durchbrüche von Leitungen und Kurzschlüsse zwischen Leitungen entstehen.

Für die Verbindungsleitungen sind maßgeblich zwei Fehlereffekte relevant, die Elektromigration (EM) [33, 53, 51, 1] und die Stressmigration (SM) [95]. Die Elektromigration tritt verstärkt in Kupferleitungen auf. Bei einem Überschreiten gewisser elektromagnetischer Feldstärken findet ein Abtransport von Metallatomen des Verbindungsmaterials durch den Elektronenfluss statt. Durch das abgetragene Material entstehen so auf der einen Seite Löcher und auf der andere Seite Anlagerungen. Der EM Effekt hängt exponentiell von der Temperatur ab.

Stressmigration hat das gleiche Fehlerbild wie die Elektromigration und zeigt sich durch einen Materialabtransport. Das Modell für SM wird als mechanischer Stress aufgefasst. Der mechanische Stress entsteht zwischen verschiedenen Materialien, wenn diese einen unterschiedlichen Wärmeausbreitungskoeffizienten aufweisen und ein bestimmter Temperaturbereich überschritten wird.

Es sind drei wesentliche Fehlereffekte bezüglich der Degradation der Transistoreigenschaften relevant:

- *Negativ(Positiv) Bias Temperatur Instability* (NBTI/PBTI) [57, 74, 105, 4, 49] führen zu einer veränderten Schaltgeschwindigkeit eines Transistors unter Stress.
- *Hot Carrier Injection* (HCI) [25] ändert dauerhaft die Transistoreigenschaft.
- *Time-Dependent Dielectric Breakdown* (TDDB) bezeichnet den Durchbruch der Isolierschicht [62, 18, 96, 6]

Der Fehlereffekt *Bias Temperatur Instability* kann bei p-Kanal sowie bei n-Kanal Transistoren auftreten in Abhängigkeit von der jeweils anliegenden Gate-Spannung. Der Literatur nach sind p-Kanal Tranistoren stärker von diesem Effekt betroffen. NBTI kommt zum Tragen, wenn an einen p-Kanal Transistor eine negative Gate-Spannung angelegt wird. Das kann dazu führen, dass sich in der Isolierschicht positive Ladungen ansammeln und sich dadurch die Schwellspannung des Transistors erhöht. Der Effekt ist abhängig von der Temperatur und der vorherrschenden Feldstärke. Es existieren mehrere physikalische Modelle, um den Effekt zu erklären[24].

Auch der HCI-Effekt beeinflusst Bereiche eines Transistors, die normalerweise isolierend sein sollen. Durch hohe elektromagnetische Feldstärken werden Partikel beschleunigt und können in das Gate-Oxid gelangen. Dadurch ergeben sich dauerhaft veränderte

Transistoreigenschaften wie eine höhere Schwellspannung und eine geänderte Steilheit der Schaltkurve. Dieser Effekt tritt verstärkt bei hohen Feldstärken auf.

*Time-Dependent Dielectric Breakdown* wird auch als Gate-Oxid Breakdown bezeichnet und beschreibt einen Alterungsprozess, der über die Zeit in MOS-Transistoren statt findet. Im Laufe der Zeit entsteht aus zufällig verteilten leitenden Stellen in der Isolierschicht eine Überlappung bis hin zu einem leitendem Pfad durch die Oxid-Schicht vom Gate zum Substrat. Dieser Effekt verstärkt sich unkontrolliert selbst und kann am Ende zu einem Durchbruch der Isolierschicht mit einem Verlust der Kontrolle über den Transistor führen. *Time-Dependent Dielectric Breakdown* ist stark abhängig von der Spannung und exponentiell von der Temperatur.

### 2.2.2. Behandlung permanenter Fehler in FPGAs

FPGAs (engl. field programmable gate array) bieten eine flexible Möglichkeit, durch die Programmierung ihrer internen Struktur Schaltungs-Designs abzubilden. Da die Programmierung der Logikbausteine änderbar ist, ermöglicht sie einen Rekonfigurationsmechanismus, der zur Behandlung permanenter Fehler ausgenutzt werden kann [59, 61]. Bevor ein FPGA mit einem Design programmiert wird, erfolgt unter anderem ein Abbilden der Funktionen des Designs auf die vorhandenen Schaltungselemente des FPGAs. Dies kann zur Behandlung eines permanenten Fehlers genutzt werden. Nachdem ein Fehler bekannt ist, wird der defekte Logikbaustein vom Abbildungsprozess nicht verwendet und durch einen anderen Bereich auf dem FPGA ersetzt. Die neu erzeugte Abbildung wird anschließend in das FPGA programmiert. Die vorhandenen Lösungsansätze können hinsichtlich der Berechnung der Abbildung unterschieden werden:

1. Ein FPGA wird im Voraus, bezüglich auftretender Defekte, partitioniert. Die notwendigen Abbildungstabellen werden während der Systementwicklung berechnet [46, 35].

2. Die Berechnung einer neuen Abbildung erfolgt im Einsatz des FPGAs unter Berücksichtigung von Reserveflächen [31] .

3. Eine darüber hinausgehende Lösung ist in [104] vorgestellt. In dem Lösungsvorschlag wird zunächst ein Re-Design des Systems vorgenommen mit einer Überführung eines Fehlertoleranzschemas in ein anderes. Es kann zum Beispiel ein TMR-System in ein Duplex-System überführt werden. Als Motivation wird angegeben, dass flächenintensive Fehlertoleranz-Designs kaum Reserve-Bereiche bereitstellen.

Des Weiteren kann unterschieden werden, wer die Berechnung und das Laden der Konfigurationstabellen durchführt. In [31] erfolgt die Berechnung und die Re-Programmierung durch das FPGA selbst. Eine Dual-FPGA-Lösung ist in [56] zu finden. Die beiden im System vorhandenen FPGAs überwachen sich gegenseitig und Rekonfigurieren bei Erkennung eines permanenten Fehlers das jeweils andere FPGA. Für eine entfernte Steuerung einer Rekonfiguration kann die Rekonfiguration auch über ein Netzwerk erfolgen [103].

### 2.2.3. Verfahren für festverdrahtete Logik

Der Begriff festverdrahtete Logik bezeichnet in diesem Kontext Mikroprozessoren und ASICs, die in einem eingebetteten Umfeld eingesetzt werden. Typischerweise bieten sie keine Form der Rekonfiguration, wie sie von den FPGAs bekannt ist. Zur Behandlung permanenten Fehler in ASICs ist es notwendig, interne Baugruppen aktivieren und gegebenenfalls auch abschalten zu können. Darüber hinaus ist es erforderlich, Daten innerhalb des Prozessors an andere Baugruppen umzuleiten. Es können zwei allgemeine Verfahren unterschieden werden, um im Fehlerfall eine Rekonfiguration vorzunehmen:

- Zum Aktivieren und Deaktivieren von Schaltungsbereichen werden Schalter eingesetzt [43].
- Es erfolgt eine Anpassung der Aufgabenplanung für super-skalare Prozessoren, um die Verwendung defekter Baugruppen zu vermeiden. In dynamisch geplanten Architekturen wird der vorhandene Laufzeitscheduler entsprechend erweitert. In [84] ist ein Verfahren präsentiert, welches eine statisch geplante Architektur um eine dynamische Aufgabenplanung erweitert.

Ein wichtiges Kriterium für die Nutzbarkeit einer Reparaturstrategie ist der zusätzliche Hardwareaufwand zur Administration der Reparatur. Für regulär aufgebaute Hardware-Strukturen bleibt der Administrationsaufwand für zusätzliche Controller in einem akzeptablen Verhältnis. In den irregulären Strukturen des Kontrollpfades eines ASICs ist es schwieriger, redundante Strukturen zu identifizieren, weshalb eine Vervielfältigung des gesamten Kontrollpfades erfolgt. Darüber hinaus hat auch die Wahl der Granularität der zu rekonfigurierenden Schaltungsanteile eine Auswirkung auf die allgemeine Nutzbarkeit einer Reparaturstrategie. Es können die folgenden Formen unterschieden werden:

- Auf dem Gate-Level können Transistoren oder auch Gatter sehr fein-granular rekonfiguriert werden [23, 45].
- Auf dem Komponenten-Level erfolgt die Auswahl der zu rekonfigurierenden Hardware-Strukturen anhand von Prozessorbaugruppen wie zum Beispiel ALUs oder Registern [7].
- Auf dem Kern-Level können Prozessor-Kerne gegeneinander ausgetauscht werden.

Eine Rekonfiguration auf dem Gate-Level erfordert einen hohen Administrationsaufwand. Dabei entsteht das Problem, dass unverhältnismäßig viel zusätzliche Hardware zur Organisation der Reparatur erforderlich wird und dann keine Steigerung der Systemzuverlässigkeit erreicht wird. Der Vorteil liegt darin, dass im Fehlerfall wenige Hardware-Strukturen verloren gehen. Außerdem kann es passieren, dass auf dem Kern-Level beim Auftreten von schon einem Fehler eine große Anzahl an weiterhin funktionierenden Transistoren nicht mehr verwendet. Ein entsprechend gutes Verhältnis wird auf dem Komponenten-Level erreicht.

Verfahren für Hardware-Strukturen mit einem regulären Aufbau, wie es Speicherbausteine bieten, sind etabliert. Darauf aufbauend können die Techniken auch für ähnliche

Array-Strukturen des Prozessors angewandt werden. In [14] werden „Array-Strukturen“ wie Re-Orderbuffer und Branch-Table gegen permanente Fehler abgesichert. Hierzu werden Reserve-Einträge in den tabellenartigen Hardwarestrukturen bereitgestellt. In [90] wird ein ähnlicher Ansatz verfolgt, wobei darüber hinaus auch Cache-Strukturen redundant ausgelegt werden. Weiterhin wird die Registerbank als Tabellenstruktur aufgefasst und um Reserve-Einträge, welche dann als Backup-Register fungieren, erweitert.

### 2.2.4. Software-basierte und hybride Ansätze

Die allgemeine Idee der software-basierten und hybriden Ansätze ist es, die auf der Hardware ausgeführte Software in der Form anzupassen, dass sie die Verwendung einer defekten Komponente vermeidet. Die Anpassung der ausgeführten Anwendung kann auf unterschiedlichen Wegen erfolgen und hängt davon ab, ob es sich um eine parallele Architektur mit einer dynamischen oder statischen Ablaufplanung handelt. Die zeitliche Planung und Bindung von Operationen erfolgt für ein dynamisch geplantes System durch einen Laufzeitscheduler im Kontrollpfad. In einem statisch geplanten System erledigt dies der Compiler und die Informationen darüber werden im Programmspeicher codiert. Die folgenden grundsätzlichen Ansätze können ein statisch geplantes System erweitern, um auf den dauerhaften Ausfall von Komponenten reagieren zu können:

1. Ein statisch geplantes System um ein dynamisches Scheduling erweitern [84]
2. Mehrere Instanzen einer Anwendung mit unterschiedlichen Ablaufplänen im Programmspeicher hinterlegen
3. Erweitern der Mikro-Programmablaufsteuerung [9, 8]
4. Rekonfiguration oder Re-Compilierung einer Anwendung im Feld

Der zusätzliche Hardware-Aufwand für die Erweiterung einer statisch geplanten Architektur mit einer dynamischen Ablaufplanung zur Kompensation von permanenten Fehlern steigt mit einer höheren Parallelität im Datenpfad und den Anforderungen an die Qualität der dynamischen Planung.

Um den Hardware-Overhead zu vermeiden, kann eine angepasste Anwendung ausgeführt werden. In [40] werden für einen anwendungsspezifischen Prozessor fehlertolerante Ablaufpläne im Voraus berechnet, um permanente Fehler in einer Ausführungseinheit zu behandeln. Der Fokus der Arbeit liegt auf einer automatischen Generierung eines fehlertoleranten Designs, welches minimal bezüglich des Hardware- und Performance-Overheads ist und eine maximale Anzahl an Fehlern tolerieren kann. Dazu werden mehrere Ablaufpläne während der High-Level-Synthese berechnet, die sich im Ressourcenverbrauch unterscheiden, um eine defekte Komponente zur Laufzeit nicht verwenden zu müssen. Im Fall einer defekten Komponente wird zur Laufzeit des Systems der entsprechende Ablaufplan ausgewählt.

Ein ähnliches Prinzip ist es, den Programmspeicher einer MikroProgrammablaufsteuerung um Sequenzen zu erweitern, die im Fehlerfall alternativ ausgeführt werden. Eine Mikro-Programmsteuerung übernimmt in einem Mikroprozessor anstelle eines festverdrahteten Automaten die Ansteuerung des Datenpfades. Die Signale zur Ansteuerung sind in einem ROM hinterlegt, in welchem durch einen Mikro-Programmsequenzer

die Steuersignale ausgewählt werden. Die decodierten Makro-Operationen der Anwendung dienen als Einsprung in die Mikrobefehlssequenzen, welche durch den Mikro-Programmsequenzer ausgeführt werden. Um dauerhaft ausgefallene Komponenten zu kompensieren, werden zusätzliche, im voraus erzeugte Mikroprogrammsequenzen in den Befehlsspeicher aufgenommen. Die zusätzlichen Sequenzen berechnen komplexe Hardware-Funktionen mit Hilfe einfacherer Hardware-Operationen. Am Beispiel der Multiplikation ergibt sich eine alternative Umsetzung nur mit Additions- und Sprungoperationen. Im Fehlerfall muss durch den Mikro-Programmsequenzer die alternative Sequenz ausgewählt und gestartet werden.

Um die Berechnung im Voraus und den zusätzlichen Speicheraufwand der redundanten Ablaufpläne zu vermeiden, kann die Anpassung der Anwendung zur Laufzeit des Systems erfolgen [84]. Der grundlegende Lösungsansatz ist es, für eine statisch geplante parallele Architektur die Anwendung im Programmspeicher des Systems so zu modifizieren, dass die Verwendung defekter Komponenten im Datenpfad vermieden wird. Dazu werden schrittweise die Instruktionswörter angepasst und die Bindung der Operationen innerhalb der Instruktion verändert. Die so geänderten Instruktionswörter ersetzen die ursprünglichen Instruktionen im Programmspeicher.

Ein erweiterter hybrider Ansatz zur Behandlung in ASIC-basierten Systemen mit parallelen Ausführungseinheiten im Datenpfad ist in [67] und eine Verfeinerung in [16] zu finden. Hybrid bedeutet in diesem Zusammenhang, dass durch Ausnutzung einer zeitlichen Redundanz zusammen mit einer natürlichen Hardware-Redundanz eine graceful degredation bezüglich der Systemleistung akzeptiert wird, um den dauerhaften Ausfall einer Ausführungseinheit zu tolerieren. Um einen permanenten Fehler zu behandeln, erfolgt eine Rekonfiguration des Systems und die ausgeführte Anwendung wird an die Fehlersituation angepasst. Der dauerhafte Ausfall einer Ausführungseinheit bedeutet eine Reduzierung der Parallelität im Datenpfad. Somit ist es notwendig, Operationen, die zuvor gleichzeitig ausgeführt werden konnten, nacheinander auszuführen. Zusätzlich müssen Operationen an eine andere Ausführungseinheit gebunden werden, falls sie ursprünglich an die defekte Einheit gebunden wurden. Die Neuplanung und die Änderung der Bindung wird während der High-Level-Synthese des Systems bestimmt und in der Ablaufplanung berücksichtigt. Die zur Laufzeit erforderlichen Änderungen im Ablauf und der Bindung erfolgen durch angepasste Hardware-Strukturen in Bezug auf die Verbindungsstrukturen und den Controller. Es wird somit ein fehlertoleranter Ablaufplan vorausberechnet, der sich mit wenigen Hardwareänderungen dynamisch zur Laufzeit an eine Fehlersituation anpassen lässt.

## 2.3. Parallele Architekturen und Mehrkern-Systeme

Zur Leistungssteigerung eines Rechensystems kann eine **parallele Berechnung** erfolgen, bei der eine gleichzeitige Nutzung redundanter Ressourcen erfolgt. Parallele Rechnerarchitekturen stellen eine geeignete Hardware zur Verfügung, um eine gleichzeitige Berechnung zu ermöglichen. Die eingesetzten Systemarchitekturen unterscheiden sich dann in der Granularität der redundant vorhandenen Hardware und dem Zugriff auf die redundanten Ressourcen. Die Granularität reicht von mehrfach vorhandenen Recheneinheiten innerhalb eines Kerns bis hin zu verteilten Computersystemen in einem

Netzwerk. Die Struktur des Systems kann dabei hierarchisch aufgebaut sein, wobei sich Komponenten auch gegenseitig enthalten können.

### 2.3.1. Begriffsklärung und Klassifizierung

Aus Sicht der eingebetteten Systemen liegt der Schwerpunkt einer Parallelisierung nicht in der gleichzeitigen Berechnung eines zerlegten Einzelproblems, sondern tendenziell eher darin, verschiedene Aufgaben gleichzeitig abzuarbeiten. In der Fahrzeugelektronik zum Beispiel muss eine Vielzahl von Aufgaben parallel erledigt werden, angefangen bei den Controllern zur Motorsteuerung, ABS, Airbag oder auch Stabilitätskontrolle bis hin zu Assistenzsystemen, welche die Umgebung überwachen und in einer Gefahrensituation den Fahrer warnen und notfalls in den Betrieb eingreifen. Da es sich hier fast ausschließlich um sicherheitskritische Aufgaben handelt, müssen diese immer gleichzeitig arbeiten und es kann schwer eine Priorisierung erfolgen.

Parallele Computerarchitekturen lassen sich unterschiedlich klassifizieren. Eine der bekanntesten Klassifizierungen ist die von Flynn [26]. Die Systeme werden hier bezüglich der abgearbeiteten Instruktions- und Datenströme unterschieden in eine einfache (Single) und mehrfache (Multiple) Abarbeitung. Der Instruktionsstrom wird typischerweise auch als **Steuerfluss** bezeichnet. Die vier Klassifizierungen nach Flynn sind SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instruction Single Data) und MIMD (Multiple Instruction Multiple Data). SISD Architekturen entsprechen dem klassischen Prozessor, der genau einen Steuerfluss und einen Datenstrom verarbeiten kann. Die nächste Architektur SIMD arbeitet auch mit lediglich einem Befehlsstrom, kann aber mehrere Datenströme verarbeiten. Typische Vertreter sind Prozessor-Arrays und Vektormaschinen. In Vektormaschinen bietet die Architektur die Möglichkeit, mit einer Instruktion mehrere Komponenten eines Vektors in einem Ausführungsschritt zu verknüpfen. Die MISD Architektur ist eher untypisch, da hier mehrere Steuerflüsse auf dem gleichen Datum arbeiten. Inzwischen ist die MIMD Architektur die am häufigsten anzutreffende parallele Computerarchitektur. Die in einem System vorhandenen Recheneinheiten können unterschiedliche Steuerflüsse ausführen und diese auf verschiedene Datenströme anwenden.

Ein wesentliches Unterscheidungsmerkmal von parallelen Architekturen ist die Organisation der Speicherstrukturen. Es werden zwei generelle Formen unterschieden:

- Der gemeinsame Speicher (engl. shared memory), bei dem alle Prozessoren Zugriff auf einen globalen Speicher besitzen.
- Der verteilte Speicher (engl. distributed memory), bei dem mehrere lokale Speicher existieren.

Der Vorteil eines gemeinsam genutzten Speichers liegt in dem globalen Adressraum, der für alle Teilnehmer gleich aufgebaut ist, weshalb Änderungen im Speicher für alle Teilnehmer sofort sichtbar sind. Mit steigender Anzahl an Prozessoren im System spielt die Speicherbandbreite eine entscheidende Rolle und stellt den wesentlichen Nachteil dieser Speicherform dar. Traditionell wird der Zugriff auf den gemeinsamen Speicher in UMA (engl. Uniform Memory Access) und NUMA (engl. Non-Uniform Memory Access) unterschieden. In einer UMA-Architektur ist die Speicherzugriffszeit für alle

Prozessoren gleich. In der Praxis kann der tatsächliche Zugriff aufgrund von Caches und Cache-Cohärenz-Protokollen[7] variieren.

Die ersten parallelen Systeme wurden als **Symmetric Multi-Processor** (SMP) bezeichnet. Mehrere homogene Kerne wurden mit einem Bus verbunden und hatten so Zugriff auf einen gemeinsam vorhandenen Hauptspeicher. In der heutigen Zeit vereinen die gängigen Mehrkernprozessoren von Intel und AMD mehrere homogene Kerne auf einem Chip. Diese Prozessoren werden als **Chip-Multi-Processors** (CMP) bezeichnet. Auch hier teilen sich alle Kerne einen Hauptspeicher, allerdings erfolgt dies nicht mehr über einen Bus, sondern über einen integrierten Speichercontroller mit Crossbar-Switch.

Ein System für eine NUMA-Architektur kann mehrere SMP/CMP-Kerne kombinieren, die über spezielle Links miteinander verbunden sind. Es existiert ein logischer globaler Adressraum und typischerweise eine Betriebssysteminstanz, aber mehrere physikalische Speicher. Die Zugriffszeit auf den Speicher kann nun variieren, je nachdem ob der physische Speicher lokal angebunden ist oder der Zugriff über den Link erfolgen muss. Die Server-Prozessoren von AMD und Intel bieten genau diese Möglichkeit, ein entsprechendes System aus mehreren CMP-Kernen aufzubauen.

Im Unterschied zum gemeinsamen Speicherkonzept ist für den verteilten Speicher immer eine Form von Verbindungs- bzw. Kommunikationsnetzwerk erforderlich. Die Prozessoren eines solchen Systems besitzen alle einen lokalen Speicher mit einem eigenen lokalen Adressraum und bilden somit keinen globalen Adressraum ab. Die Informationsbeschaffung zwischen den einzelnen Prozessoren muss über ein geeignetes Kommunikationsverfahren erfolgen, da lokale Änderungen nicht automatisch global sichtbar werden. Der Vorteil dieser Speicherarchitektur liegt in der besseren Skalierbarkeit bezüglich Speicherzugriff und Speichergröße. Die Abbildung globaler Datenstrukturen kann sich hingegen als schwierig gestalten. Darüber hinaus kann der Zugriff auf entfernte globale Daten eine signifikante Verzögerung mit sich bringen.

Einen großen Einfluss auf die Leistung eines parallelen Systems haben die Verbindungs- und Kommunikationsstrukturen. Die wesentlichen Eigenschaften der Verbindungsstrukturen sind die Bandbreite, die Skalierbarkeit, die Reaktionsgeschwindigkeit und auch die Ausfallsicherheit. Im eingebetteten Bereich ist der klassische Vertreter die Busstruktur. Ein zentraler Bus verbindet alle relevanten Komponenten eines Systems miteinander. Die Regelung des Buszugriffs kann zentral erfolgen oder über ein Protokoll geregelt sein. Mit steigender Anzahl an Busteilnehmern ergibt sich der Nachteil einer verringerten Bandbreite. Eine nächste Möglichkeit ist der Einsatz eines Schalternetzwerkes, welches einen direkten Kanal zwischen zwei Kommunikationspartnern schaltet. Diese Organisationsform ist eher selten anzutreffen und hat den Nachteil, dass mit steigender Komponentenanzahl der Aufwand für das Schaltnetzwerk exponentiell steigt. Mit der gesteigerten Integrationsdichte digitaler Schaltungen und der dadurch, auch im eingebetteten Bereich, gestiegenen Anzahl an Kernen in einem System etablieren sich Netzwerk-Architekturen zur Kommunikation. Ein weiterer Vorteil ergibt sich daraus, dass die Komponenten des Systems unterschiedlich getaktet sein können und eine exakte Taktverteilung über den gesamten Chip entfallen kann. Die Überbrückung der verschiedenen Takte erfolgt anhand der netzwerkartigen Kommunikationsstrukturen.

[7] Ein Cache-Cohärenz-Protokoll stellt sicher, dass ein Datum konsistent ist, wenn mehrere Kopien in verschiedenen Caches vorhanden sind.

Eingebettete Systeme basierend auf einer Netzwerkkommunikation, heißen **Network on a Chip** (NOC). Mittels der typischen Elemente wie Netzwerk-Adapter, Router und Verbindungsleitungen (Links) zwischen den Routern wird ein Netzwerk im System implementiert. Der gesamte Informations- und Datenaustausch innerhalb des Systems erfolgt über dieses Netzwerk.

In den folgenden Unterkapiteln werden bekannte Strategien zur Zuverlässigkeit und Fehlertoleranz in Mehrkernsystemen diskutiert. Zunächst wird besprochen, wie die Behandlung von transienten Fehlern mit Hilfe von Threads erfolgt. Anschließend liegt der Fokus auf Verfahren zur Behandlung von permanenten Fehlern. Dazu werden zunächst rein hardware-basierte Ansätze präsentiert. Methoden, basierend auf Hardware-Threads und Virtualisierungstechniken, werden im Anschluss präsentiert. Abschließend werden Verfahren diskutiert, welche die Anwendung an eine Fehlersituation in einem Mehrkernsystem anpassen.

### 2.3.2. Behandlung transienter Fehler

Zwei wesentliche Eigenschaften von Mehrkernsystemen erleichtern die Implementierung einer Behandlungsstrategie für transiente Fehler. Zunächst stellen Mehrkernsysteme eine Art von natürlicher Hardware-Redundanz bereit. Darüber hinaus bieten viele der modernen CMP-Systeme das Konzept der Threads an, mit deren Hilfe sich mehrere Instanzen einer Anwendung erzeugen lassen. Die Arbeiten zu diesem Themenkomplex lassen sich anhand der grundlegenden Strategie in folgende drei Bereiche unterteilen:

- hardware-basierte Lösungen, welche die ursprüngliche Prozessorarchitektur erweitern und Komponenten vervielfältigen [100, 2],
- Lösungsstrategien, basierend auf dem Konzept von Threads [30, 73] und
- Verfahren, welche auf Virtualisierung basieren.

In [100] ist ein hardware-basiertes Verfahren beschrieben, basierend auf einem rekonfigurierbarem Mehrkern-System, welches dynamisch die Soft-Error-Rate reduzieren kann. In jedem Kern erfolgt ein Monitoring der Prozessor-Pipeline, um auf transiente Fehler reagieren zu können. Beim Auftreten eines transienten Fehlers wird ein vorheriger Prozessorzustand geladen, um die Berechnungen zu wiederholen. Bestimmte Komponenten, wie zum Beispiel Load/Store-Einheiten, sind doppelt ausgelegt, um eine Fehlerfortpflanzung zu verhindern. Die Fehlerraten eines jeden Prozessors werden in entsprechenden Registern akkumuliert und über das Kommunikationsnetzwerk des Mehrkernsystems in einem zentralen Kern gesammelt. Dieser Kern kann bei Bedarf eine Lastumverteilung im System vornehmen, um auf eine bestimmte Fehlerrate zu reagieren.

Ein weiterer hardware-basierter Ansatz ist in [2] zu finden. Die Grundlage des Mehrkernsystems ist ein Chip-Multiprozessor, der so partitioniert werden kann, dass sich bezüglich der Ressourcennutzung innerhalb der Partitionen ein beliebiges n-faches Redundanzschema ergibt. Viele Komponenten, wie zum Beispiel TLBs, Caches und Speichercontroller, sind dediziert an gewisse Fehlersituationen anpassbar. Die Anpassung an bestimmte Fehlersituationen erfolgt mit Hilfe einer Software auf der System-Ebene.

Der Steuerfluss wird auch in das Konzept der **Threads** gekapselt. Diese bearbeiten unterschiedliche Aufgaben in einem Rechnersystem. Um eine höhere Systemleistung und eine bessere Ressourcen-Auslastung eines einzelnen Kerns zu erzielen, ist es möglich, Threads parallel auf einem Prozessorkern auszuführen. Diese Technik wird als **Simultaneous Multi-Threading** (SMT) bezeichnet. Dazu werden gemeinsam genutzte Komponenten eines Prozessors verdoppelt, wie zum Beispiel der Programmzähler oder die Register. Die Thread-basierten Verfahren lassen sich weiter unterteilen in:

- Methoden, welche die gegebene Architektur um Hardware-Komponenten erweitern [78, 101, 30, 73] und
- rein software-basierte Lösungen [91].

Das kombinierte Verfahren des Thread-Konzepts mit Hardware-Erweiterungen zur Fehlertoleranz erzeugt zu einem gestarten Thread zusätzlich eine Kopie. Die beiden Threads laufen dann zeitlich versetzt. Um Fehler zu erkennen, ist es notwendig, berechnete Werte beider Threads abzugleichen. Dazu wird in [101] die Prozessor-Pipeline erweitert, um während des Ressourcenzugriffs berechnete Ergebnisse zu validieren. Das erfordert teilweise ein Warten des vorauslaufenden Threads auf die Validierung seiner Ergebnisse durch den zweiten Thread. Die Threads können ausschließlich in der zeitlichen Dimension eine Redundanz ausnutzen, um transiente Fehler zu behandeln [58]. Darüber hinaus ist es auch möglich, in der räumlichen Dimension zu arbeiten und die Threads auf getrennten Ressourcen auszuführen [30]. Dabei wird es erforderlich, dass die Prozessor-Pipelines sich bezüglich der Threads kernübergreifend synchronisieren. Ein Rollback erfolgt im Fall eines Fehlers mit dem ursprünglich gesicherten Pipeline-Zustand.

Ein Ansatz, der gänzlich auf Erweiterungen der Hardware verzichtet, ist in [91] präsentiert. Das Verfahren arbeitet nicht auf der Basis von Threads, sondern auf der Basis von Prozessen. Der Unterschied ist, dass ein Prozess mehrere Steuerflüsse enthalten kann. Zu einer Anwendung werden mehrere Prozesse instanziiert, wobei lediglich einer als Master-Prozess fungiert und mit den Slave-Prozessen in Verbindung steht. Das Betriebssystem kann die erzeugten Prozesse beliebig planen und getrennt auf den vorhandenen Ressourcen verteilen. Die Organisation zur Fehlertoleranz erfolgt mittels einer Software-Bibliothek, die durch System-Aufrufe der Prozesse aktiviert wird. Während solch eines Aufrufs werden Ein- und Ausgabedaten der Prozesse abgeglichen, um mögliche Fehler zu erkennen.

Virtualisierung in Mehrkernsystemen kann auf unterschiedliche Weise implementiert sein, wie zum Beispiel durch eine Middleware, die transparent zwischen Betriebssystem und physikalischer Hardware arbeitet, oder durch eine hardware-seitige Implementierung. In [102] wird ein Mehrkernsystem mit einer Virtualisierungslösung in Hardware präsentiert, um ausgeführte Anwendungen gegen transiente Fehler schützen zu können. Der Prozessor stellt dem Betriebssystem eine gewisse Anzahl an virtuellen Kernen zur Verfügung. Die Abbildung bzw. das Scheduling der virtuellen Kerne auf die tatsächlichen physikalisch vorhandenen Kerne erfolgt durch eine entsprechende Hardware-Komponente des Prozessors. Dieser Controller ist darüber hinaus dafür verantwortlich, einen Thread, der fehlertolerant ausgeführt werden soll, auf zwei physikalisch getrennte

Kerne zu spiegeln. Die Kopplung zwischen den Kernen und Organisation des Rollbacks im Fehlerfall obliegt auch dem Hardware-Controller.

### 2.3.3. Permanente Fehler

Viele Lösungsansätze nutzen die in Mehrkern-Systemen vorhandene natürliche Hardware-Redundanz zur Behandlung permanenter Fehler mit einer dadurch tolerierten Degradation der Systemleistung. Wenn eine degradierte Systemleistung inakzeptabel ist, ist es erforderlich, zusätzliche Redundanz in das System einzubringen in Form von Spare-Komponenten. Im Fehlerfall erfolgt dann eine Rekonfiguration, um die Verwendung defekter Komponenten zu vermeiden. Die Granularität einer Rekonfiguration bezüglich der auszutauschenden Komponenten kann folgendermaßen unterschieden werden:

1. Intra-Core Redundanz verwendet ausschließlich Komponenten innerhalb eines Prozessorkerns [14, 89],

2. Inter-Core Redundanz kann kernübergreifend auf Komponenten benachbarter Kerne zugreifen [77] und

3. Redundanz auf Systemebene, bei der Reservekerne bereitgestellt werden [106] oder ein Ausnutzen der natürlichen Redundanz [69] erfolgt.

Intra-Core Redundanz kann auch in einem Mehrkernsystem genutzt werden, kann aber nicht den Vorteil eines Mehrkernsystems in vollem Umfang ausnutzen. Es handelt sich um ein fein-granulareres Schema mit der niedrigsten Verlustrate bezüglich intakter Komponenten. Allerdings kann der Defekt einer nicht fehlertoleranten Komponente eines Kerns einen vollständigen Systemausfall bedeuten. Für die Inter-Core Redundanz werden Komponenten kernübergreifend eingesetzt, zum Beispiel durch die Verwendung einer Baugruppe eines benachbarten Kerns, um den Ausfall einer kritischen Komponente innerhalb eines Kerns zu kompensieren. Auf Systemebene werden Kerne ausgetauscht bzw. deaktiviert mit der höchsten Verlustrate für intakte Schaltungskomponenten. In [106] werden zusätzliche Kerne, als Cold-Spares in das System eingebracht. In [69] werden massiv parallele MPSoCs untersucht. Zur Kompensation permanenter Fehler wird das System rekonfiguriert und defekte Kerne von der Zuweisung einer Aufgabe ausgeschlossen. Das Ziel einer systemweiten Lastumverteilung während der Rekonfiguration ist es dann, die Degradation der Systemleistung zu minimieren.

Die Organisation der Rekonfiguration kann auf unterschiedlichen Systemebenen erfolgen und hat dadurch verschiedene Vor- und Nachteile:

- Hardware-basierte Verfahren [77, 70, 106]
- Eine Kombination aus Architekturanpassungen und Software-Erweiterungen [42]
- Virtualisierung [38, 17]
- Re-Compilierung [54]

Bei den hardware-basierten Verfahren ist es erforderlich, dass die zusätzlichen Komponenten nicht dominierend werden und dadurch die Zuverlässigkeit reduzieren. Aufgrund der Implementierung in Hardware sind die Strategien wenig flexibel und aufwändig in der Anpassung an beliebige Systemkonfigurationen. In [77] wird ein hardwarebasiertes Verfahren präsentiert, welches eine Inter-Core Redundanzstrategie implementiert. Der grundlegende Ansatz ist es, Komponenten eines defekten Kernes auf benachbarten Kernen wiederzuverwenden. Der Vorteil liegt darin, dass intakte Strukturen eines Kerns weiter nutzbar sind, auch wenn der Kern selbst nicht aktiv durch das System genutzt werden kann. Abbildung 2.3 veranschaulicht das Borgen von Ressourcen bei benachbarten Kernen. In dem Beispiel ist Kern 1 deaktiviert und einige Komponenten werden durch die benachbarten Kerne weiter genutzt, um die jeweils defekten eigenen Komponenten zu ersetzen.

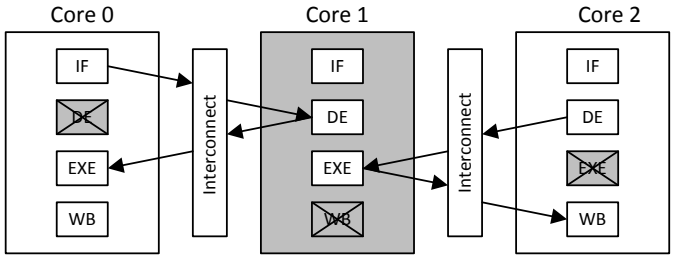


Abbildung 2.3.: Schema zum Borgen von Pipeline-Ressourcen bei benachbarten Kernen [77]

Ein anderer Ansatz, eine hardware-basierte Inter-Core Redundanzstrategie umzusetzen, ist in [70] zu finden. Der grundsätzliche Ansatz, sich Komponenten zu borgen, ist ähnlich dem zuvor präsentierten Verfahren, jedoch erfolgt die Organisation auf eine andere Weise. Hier werden keine Mikro-Operationen (Pipeline-Stufen) zwischen benachbarten Kernen verschoben, sondern der Austausch erfolgt auf einem höheren Abstraktionslevel, dem der Threads. Wenn im Fehlerfall ein Kern eine Instruktion nicht berechnen kann, wird der entsprechende Thread auf einen anderen Kern migriert, dessen entsprechende Hardware-Komponente fehlerfrei ist. Die gesamte Organisation zum Ablauf ist in einem Hardware-Scheduler in den Prozessor integriert und erfolgt völlig transparent für das Betriebssystem bzw. die Anwendung.

### Virtualisierung

Der Einsatz einer Virtualisierungsstrategie erfordert weniger zusätzliche Hardware. Es besteht darüber hinaus die Möglichkeit, fein-granularer auf Fehler zu reagieren. Weiterhin können mit Hilfe von Software komplexe Instruktionen, wie zum Beispiel Multimedia-Operationen, emuliert werden.

Eine Vorstufe zur Virtualisierung ist in [42] zu finden. Die vorgeschlagene Reparaturstrategie erweitert eine bestehende Architektur um Hardware-Komponenten, die in Kombination mit einer Software-Schicht im Fehlerfall eine graceful degredation des

Systems ermöglichen. Das Verfahren arbeitet auf zwei Systemebenen. Es kann eine Rekonfiguration innerhalb eines Kerns vornehmen und darüber hinaus kann es Threads bei Bedarf auf andere Kerne verschieben. Die Organisation erfolgt durch eine transparente Software-Schicht, genannt Microvisor, zwischen der Hardware und dem Betriebssystem bzw. der Anwendung. Der Microvisor befindet sich in einem getrennten Speicherbereich und ist geschützt vor Zugriffen durch das Betriebssystem. Im Fehlerfall kann der Microvisor einen Kern rekonfigurieren. Dies erfolgt durch Anpassung des Microcodes für den vorhandenen Befehlssatz. Komplexe Instruktionen werden durch einfachere Mikrobefehlssequenzen ersetzt. Darüber hinaus kann der Microvisor bestimmte Hardware-Komponenten eines Kerns deaktivieren, um sie von der Benutzung auszuschließen. Um eine optimale Degradation der Systemleistung zu gewährleisten, kann der Microvisor Threads von einem Kern auf einen anderen migrieren. Um dies zu unterstützen, werden zusätzliche Hardware-Strukturen in die Architektur integriert, um die Migration zu beschleunigen und anhand von Performance-Zählern die Notwendigkeit einer Migration erkennen zu können.

In [17] wird eine vorhandene Virtualisierung einer OpenSPARC-Architecture erweitert, um permanente Fehler zu behandeln. Dazu wird eine zusätzliche Software-Schicht zwischen dem bestehenden Hypervisor und der Hardware integriert. Die Software-Schicht übernimmt das Verschieben von Aufgaben zwischen Kernen und emuliert, falls notwendig, Hardware-Funktionen in Software. In der beschriebenen Arbeit fehlen allerdings Implementierungsdetails und konkrete Untersuchungsergebnisse, um die tatsächliche Nutzbarkeit und Vorteile abschätzen zu können.

Die Arbeit von Russ Joseph [38] ist dahingegen fundierter und verfolgt den Ansatz, mit Hilfe einer Virtualisierungslösung aus teilweise defekten Kernen mehrere logische Kerne dem Betriebssystem bzw. der Anwendung zur Verfügung zu stellen. Die logischen Kerne sind aus Sicht der Anwendung in der Lage, jede beliebige Instruktion auszuführen. Allerdings setzt sich ein logischer Kern aus mehreren physikalischen, teilweise auch defekten Kernen zusammen. Die Virtualisierung wird als Systemlevel-Software implementiert und als Firmware in das System integriert. Das Betriebssystem arbeitet oberhalb dieser Abstraktionsebene. Die Virtualisierungssoftware fängt Instruktionen ab, die aufgrund einer defekten Hardware-Komponente nicht ausführbar sind. Die nicht ausführbare Instruktion wird dann entweder vollständig in Software emuliert oder auf einem anderen Kern berechnet, dessen entsprechende Hardware-Komponente intakt ist. Die Wahl hängt von der Höhe der gemessenen Leistungsdegradation des defekten Kerns ab. Der Autor berichtet von einer Leistungsdegradation von bis zu 20% für bestimmte Benchmarkanwendungen, wenn ein Teil der Instruktionen in Software emuliert wird. Zahlen zu Leistungseinbußen durch Migration sind nicht angegeben.

#### De-Touring: Recompilierung im Fehlerfall

Ein dieser Arbeit sehr ähnliches Verfahren wird von Meixner und Sorin in [54] vorgestellt. Die Grundidee ist es, Operationen, die nicht mehr aufgrund defekter Operatoren berechenbar sind, durch Software-Routinen zu ersetzen. Dazu werden die eigentlichen Maschinenbefehle durch Sprunganweisungen in quasi Unterprogramm-Routinen ersetzt. Alternativ dazu erfolgt auch die Ersetzung eines Maschinenbefehls durch eine Sequenz von Befehlen.

Die zugrunde liegende Systemarchitektur setzt sich aus einem Kontrollprozessor und mehreren einfachen In-Order Kernen zusammen. Der Kontrollprozessor ist eine General-Purpose CPU, auf dem das Betriebssystem ausgeführt wird. Der Kontrollprozessor übernimmt die Administration des Systems, verteilt Tasks auf die anderen Kerne und administriert die Reparatur. Die weiteren Kerne werden als statisch geplante single-issue, in-order RISC-Prozessoren beschrieben. Auf den RISC-Kernen läuft ein einfaches Betriebssystem, welches Systemaufrufe zur Behandlung an den Kontrollprozessor weitergibt.

Die eingesetzte Reparaturstrategie basiert auf einer Re-Compilierung der Anwendung im Fehlerfall. Als Compiler wird der GCC zusammen mit dem OPENRisc-Backend verwendet. Die Übersetzung erfolgt auf dem RISC-Kontrollprozessor. Es ist davon auszugehen, dass für die Re-Compilierung der vollständige Quellcode der Anwendung zur Verfügung steht. Die generelle Idee ist es, die Berechnungen von ausgefallenen Baugruppen durch Softwareimplementierungen zu ersetzen. Die Softwareimplementierungen werden als Detoures bezeichnet und in drei Variaten unterschieden. Aufwendigere Detoures (z.B. Multiplikation, Quadratwurzel) werden in die Standardbibliothek des GCC eingefügt und gegen die Anwendung gelinkt. In der Anwendung erfolgt dann lediglich ein Unterprogrammaufruf. Kürzere Detours werden während der Traversierung des Syntaxbaumes in die Zwischensprache eingefügt. Alternativ dazu werden kleinere Detours auch in der Übersetzungsphase von der Zwischensprache in die Assemblersprache in den Ausgabetext eingefügt.

Der vorgestellte Reparaturvorgang ist sehr aufwendig, da eine vollständige Übersetzung der Anwendung im Feld erfolgt. Dieser Vorgang wird nochmals verschärft, wenn auch verwendete Bibliotheken betroffen sind. Es wird darauf verwiesen, dass Fehler in Special-Purpose Register mittels Re-Naming repariert werden. Das hat zur Folge, dass die Calling-Conventions verletzt werden. Deshalb wird es notwendig, auch eingesetzte Bibliotheken erneut zu übersetzen. Dieser Ansatz ist für ein eingebettetes System nur bedingt nutzbar, da die Verwendung einer vollständigen Compiler-Suite mit den notwendigen Bibliotheken und dem Quellcode einen hohen Overhead bedeutet.

## 2.4. Verfahren zur Fehlererkennung

Die Voraussetzung für eine Selbst-Reparatur eines Systems zur Behandlung permanenter Fehler ist die erforderliche Erkennung des jeweiligen Fehlers. Darüber hinaus ist, in Abhängigkeit von der Granularität der gewählten Rekonfigurationsstrategie, eine entsprechende diagnostische Auflösung bezüglich der defekten Komponenten notwendig. Mit der Kenntnis über die ausgefallene Baugruppe kann eine gezielte Rekonfiguration erfolgen. Die Verfahren zur Fehlererkennung können grundsäztlich in:

- on-line Verfahren und
- off-line Verfahren unterschieden werden.

Diese Unterscheidung bezieht sich darauf, ob die eingesetzte Methode zur Fehlererkennung während der aktiven Systemausführung eingesetzt wird oder in Ausführungspausen bzw. dem Systemstart. Die hier betrachteten Ansätze beziehen sich generell auf

den produktiven Einsatz des Systems und nicht auf Fertigungstests beim Hersteller. Darüber hinaus sind nur Verfahren relevant, welche durch das System selbst ausgeführt werden, also ohne einen externen Zugang zum System.

Die on-line Verfahren können nebenläufig oder nicht nebenläufig ausgeführt werden. Die verschiedenen Ansätze gehören typischerweise zu einer Fehlertoleranzstrategie und erfordern einen zusätzlichen Aufwand. Es können die folgenden drei Kategorien unterschieden werden:

- Code-basierte und Hardware-basierte Verfahren [13, 44],
- Duplizierung der ausgeführten Operationen im Programmspeicher [10, 34] und
- Thread-basierte Verfahren [88]

Die hardware-basierten Ansätze integrieren zusätzliche Checker in die Pipeline, welche das zuvor berechnete Ergebnis nochmals verifizieren [13]. Eine weitere Möglichkeit ist es, Informations-Redundanz auszunutzen und mit Hilfe von Codes eine Fehlererkennung bereitzustellen. Mit Hilfe zusätzlicher Hardware können DMR-Systeme eine Fehlererkennung liefern und darüber hinaus mittels TMR eine Maskierung. Es gibt Ansätze zum dynamischen Aktivieren eines TMR-Systems aus einer DMR-Konfiguration, um bei Bedarf eine Diagnose vorzunehmen.

Für statisch geplante super-skalare Architekturen gibt es Verfahren, welche auf Instruktionsebene arbeiten und die Parallelität im Datenpfad ausnutzen. Dazu werden vom Compiler Instruktionen dupliziert. In Kombination mit Vergleichs-Instruktionen erfolgt parallel im Betrieb eine Detektion, ob ein Fehler vorliegt. Auf einer höheren Abstraktionsebene werden Threads dupliziert und entweder zeitlich versetzt oder nebenläufig auf verschiedenen Hardware-Komponenten berechnet. Der Abgleich der Ergebnisse kann nach Berechnung einer Instruktion oder nach willkürlich festgelegten Teilabschnitten erfolgen.

Bei den off-line Verfahren kommen Verfahren zum Einsatz, welche vom klassischen Vorgehen des Chip-Tests bekannt sind. Der Begriff des **Tests** für elektronische Schaltungen bezeichnet den Vorgang des Nachweisens, dass ein System bezüglich seiner aktuellen Eigenschaften der Spezifikation entspricht. Der Testvorgang gestaltet sich in der Form, dass an die zu testende Komponente Eingangswerte angelegt und die ermittelten Werte mit den erwarteten Ergebnissen abgeglichen werden. Der Test wird unterschieden in einen funktionalen bzw. einen strukturellen Test. Die für diese Arbeit relevanten Techniken sind:

1. Der Build-In Self-Test (BIST) [50] und
2. Der software-basierte Selbsttest (SBST) [20, 29, 81].

Die Grundlage für den Build-In Self-Test bildet der weit verbreitete und gut erforschte Scan-Test [22, 28]. Die Idee des Scan-Tests ist es, die in einer Schaltung vorhandenen speichernden Elemente zu Scan-Ketten zusammenzuschalten und dadurch ein logisches Schieberegister zu erzeugen. In diese Scan-Kette wird dann schrittweise ein Testmuster geladen. Durch einen oder mehrere Arbeitstakte wird zu dem Testmuster eine Ausgabe erzeugt, wodurch auch ein Test der kombinatorischen Logik erfolgt. Die Ausgabe wird

anschließend aus der Schaltung geschoben und mit einem Erwartungswert verglichen. Der Build-In Self-Test bietet eine hohe Fehlerüberdeckung[8] und lässt sich automatisiert in ein Design integrieren.

Ein anderer Ansatz ist der software-basierte Selbsttest [99]. Bei diesem Verfahren führt der zu testende Prozessor selbstständig ein Testprogramm aus seinem Programmspeicher aus. Das Testprogramm überprüft anhand von Instruktionen und erwarteten Ergebniswerten, ob eine Komponente Defekt ist. Für software-basierte Selbsttests wird eine gute diagnostische Auflösung und eine hohe Fehlerüberdeckung berichtet [81]. Für die Reparaturverfahren dieser Dissertation wird eine kombinierte Lösung mit einem software-basierten Selbsttest favorisiert. Die Vorteile des software-basierten Selbsttests sind, dass er keine zusätzliche Hardware erfordert und sich gut mit den Verfahren zur Rekonfiguration kombinieren lässt bezüglich der Schnittstellen für die erforderlichen Datenstrukturen. Abschließend ist festzuhalten, dass sich die diagnostische Auflösung eines software-basierter Selbsttests gut an die Granularität einer software-basierten Rekonfiguration anpassen lässt [87].

[8] Die Fehlerüberdeckung gibt für ein Testverfahren das Verhältnis an, zwischen denen durch das Verfahren erkannten Fehlern und allen möglichen Fehlern eines festgelegten Typs.

# 3. Methodische Grundlagen

Das folgende Kapitel beschreibt zunächst den internen Aufbau des Prozessorkerns, der in dieser Arbeit verwendet wird, um die entwickelten Reparaturmethoden zu demonstrieren. Anschließend wird die Programmdarstellung angegeben, so wie sie für die später entwickelten Reparaturverfahren verwendet wird. Im weiteren Verlauf wird dann die betrachtete Systemarchitektur dargestellt, die sich aus mehreren Prozessorkernen und der notwendigen Administration zusammensetzt. Abschließend wird das angestrebte software-basierte Reparaturprinzip eingeführt.

## 3.1. Prozessormodell

Das verwendete Prozessormodell basiert auf einer **VLIW**-Architektur (engl. *Very Long Instruction Word*). Diese Architektur zeichnet sich dadurch aus, dass ein besonders breites Befehlswort, auch als Instruktionswort bezeichnet, verwendet wird. Der Datenpfad eines solchen Prozessors ist superskalar ausgelegt. Dadurch kann der Prozessor im gleichen Zeitschritt mehrere Berechnungen parallel vornehmen. Die dazu notwendige Planung der parallelen Ansteuerung ist im Instruktionswort codiert und wird zur Übersetzungszeit der Anwendung durch den Compiler erzeugt. Diese Form der Steuerung wird auch als **statisch** geplant bezeichnet. Der Prozessor verwendet einen RISC-Befehlssatz (engl. *Reduced Instruction Set Computer*) mit wenigen unterschiedlichen Befehlsarten. Aus Sicht eines fehlertoleranten Systemdesigns bietet eine VLIW-Architektur die folgenden Vorteile:

- Aufgrund der Superskalarität im Datenpfad stellt der Prozessor eine natürliche Hardware-Redundanz zur Verfügung, die durch eine geeignete Reparaturstrategie ausgenutzt werden kann.
- Durch die statisch geplante Architektur ergibt sich ein regulärer struktureller Aufbau des Prozessors bezüglich des Daten- und Kontrollpfades und die Ressourcennutzung ist am Programmcode erkennbar.

Diese Faktoren sind gute Voraussetzungen für den Einsatz einer software-basierten Rekonfiguration zur Behandlung permanenter Fehler in dem beschriebenen Prozessormodell. Für eine Rekonfiguration lässt sich die statische Planung leicht durch eine Anpassung der Instruktionsworte verändern. Unter Berücksichtigung der natürliche Hardware-Redundanz und des regulären Aufbaus können dadurch Berechnungen von defekten Baugruppen auf redundante Komponenten verschoben werden.

Eine schematische Darstellung eines VLIW-Prozessors mit Steuer- und Datenpfad zeigt Abbildung 3.1. Der Datenpfad des Prozessors setzt sich aus mehreren **Funktionseinheiten** (kurz FU) ($FU1$, ..., $FUn$) zusammen. Für einen gegebenen Prozessor

gibt $maxFU$ die verfügbare Anzahl von Funktionseinheiten an. Dieser Wert entspricht auch dem maximalen Grad an parallel ausführbaren Operationen. In dem Beispiel hat $maxFU$ den Wert vier. Die Berechnungen werden in einer FU durch Operatoren ausgeführt. Ein Operator repräsentiert die notwendige Hardware, um eine Operation eines bestimmten Typs berechnen zu können. Für eine Operation $v$ liefert die Funktion $type(v)$ den jeweiligen Operationstyp. Die Menge der Operationstypen ist $O$, wobei gilt $O = \{0, ..., n\}$. Die leere Operation, auch als $Nop$ bezeichnet, kann von jeder FU ausgeführt werden. Sie hat keine Nebeneffekte und konsumiert lediglich einen Zeitschritt auf der ausführenden FU.

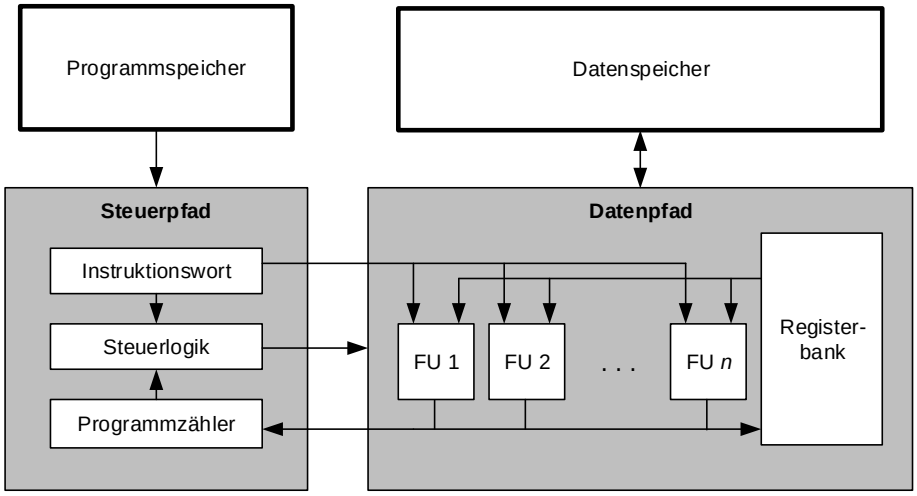


Abbildung 3.1.: Schematische Darstellung eines VLIW-Prozessors mit vier Funktionseinheiten und den getrennten Speichern für Programmcode und Daten

Die Funktionseinheiten im Datenpfad arbeiten parallel, allerdings wird je FU zu einem bestimmten Zeitpunkt immer nur eine Operation ausgeführt. Aufgrund der parallelen Ausführung benötigt jede FU einen separaten Zugriff auf die Registerbank. Die Registerbank stellt eine bestimmte Anzahl an allgemeinen Registern zur Zwischenspeicherung von Ergebnissen zur Verfügung. Für einen konkreten Prozessor beschreibt $maxReg$ die vorhandene Anzahl an Registern. Der Zugriff auf die Register der Registerbank erfolgt über Lese- und Schreibports. Für jede FU sind je zwei Lese- und ein Schreibport vorhanden. Alle Ports arbeiten parallel auf den Registern. Die Leseports stellen die notwendigen Operanden für die Berechnung einer Operation bereit. Über den Schreibport kann das berechnete Ergebnis durch das jeweilige Zielregister übernommen werden. Es handelt sich um eine Register-Register-Architektur. Damit Operationen auf Werte im Datenspeicher angewendet werden können, müssen diese zuvor in ein Register geladen werden.

Es gibt weiterhin Operationen zur Programmverzweigung. Eine Programmverzweigung kann bedingt, geknüpft an bestimmte Bedingungen, oder unbedingt erfolgen. Das Ausführen einer Verzweigungsoperation bewirkt, dass der Programmzähler mit dem

Wert für eine Zieladresse geladen wird. Speicheroperationen sowie Verzweigungsoperationen können prinzipiell von jeder FU ausgeführt werden. Eine parallele Ausführung auf mehreren FUs innerhalb eines Zeitschrittes kann unvorhersehbare Auswirkungen haben und es obliegt dem Compiler bzw. dem Programmierer, dies zu vermeiden.

Die Steuerung des Datenpfades erfolgt in jedem Zeitschritt durch ein **Instruktionswort**. Ein Instruktionswort setzt sich aus mehreren Operationen $(v_0, ..., v_{n-1})$ zusammen. Die Position einer Operation $v_i$ innerhalb eines Instruktionswortes gibt an, dass $v_i$ auf der Funktionseinheit $f_i$ ausgeführt wird. Abbildung 3.2 zeigt ein Instruktionswort für einen Datenpfad mit vier Funktionseinheiten. Das Instruktionswort setzt sich demzufolge aus vier Operationen zusammen. Jeder FU ist eine Operation zugewiesen. Das gesamte Instruktionswort ist 104 Bit breit. Eine einzelne Operation wird in 26 Bit codiert. Eine Operation enthält die Informationen zum auszuführenden Operationstyp, den beiden Quelloperanden ($src1$ und $src2$) und dem Zielregister ($dst$). Zum Codieren der Registerinformationen werden jeweils 6 Bit verwendet. Der Operationstyp ist mit 8 Bit codiert.

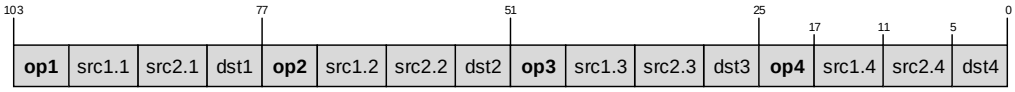


Abbildung 3.2.: VLIW-Instruktionswort mit 4 Operationen und entsprechender Codierungsgröße im Programmspeicher

#### Fließbandverarbeitung und Bypass im Prozessormodell

Zur Leistungsteigerung wird eine einfache 4-stufige Befehls-Pipeline verwendet. Die Pipelinestufen sind so aufgebaut, dass in jedem Zeitschritt die Abarbeitung des jeweiligen Instruktionswortes abgeschlossen wird. Insgesamt befindet sich jede Instruktion 4 Takte in der Pipeline. Wenn die Pipeline gefüllt ist, verlässt in jedem Zeitschritt ein berechnetes Instruktionswort die Pipeline. Die vier Phasen der Pipeline haben folgende Aufgaben:

1. In der **Fetch**-Phase (FE) wird ein Instruktionswort aus dem Programmspeicher in den Prozessor geholt.
2. Die **Decode**-Phase (DE) dekodiert die Operationen einer Instruktion und stellt die Registerwerte der Operanden bereit.
3. Während der **Execute**-Phase (EXE) erfolgt die Berechnung der Operationen auf den zugehörigen FUs.
4. In der **Write-Back**-Phase (WB) erfolgt das Zurückschreiben der Berechnungsergebnisse in die Zielregister.

Es ist nicht vorgesehen, dass die FUs selbst auch über eine Fließbandverarbeitung verfügen. Die erforderliche Ausführungszeit einer Operation $op$ auf einer FU wird als

**Latenz** bezeichnet und hat für alle Operationen den Wert 1. Operationen, die eine höhere Latenz benötigen, werden in Teiloperationen zerlegt, die dann jeweils eine Latenz von 1 haben. Dabei ist es erforderlich, dass die Teiloperationen nacheinander auf der gleichen FU ausgeführt werden.

Aufgrund der Pipeline-Organisation werden berechnete Werte erst in der letzten Phase in das Zielregister geschrieben. Eine sich anschließende Operation, die diesen Wert benötigt, muss zwei Takte warten, bis sie den erforderlichen Wert aus der Registerbank lesen kann. Dieses Problem ist als **Datenhazard** bekannt und wird mit dem Einsatz einer Forwarding-Technik behoben. Dazu werden bei Bedarf mit Hilfe eines Bypass-Netzwerks Ergebnisse aus einer Pipeline-Stufe direkt in eine andere Stufen geleitet. Abbildung 3.3 zeigt den Aufbau des Bypass-Netzwerks. Das Netzwerk setzt sich aus den notwendigen Multiplexern für die Quelloperanden, den Verbindungsleitungen und einer Kontrolllogik zur Multiplexersteuerung zusammen. An den Eingangsmultiplexern kann gewählt werden, ob ein Operand aus der Registerbank, von einem beliebigen FU-Ausgang oder aus einer WriteBack-Stufe bereitgestellt wird. Die Auswahl wird durch eine Kontrolllogik getroffen, die dazu die Information bezüglich Quell- und Zielregister zweier aufeinander geplanten Operationen auswertet. Stimmt die Registernummer eines Quelloperanden einer Operation $v$ aus Takt $i$ mit der Zielregisternummer einer Operation $u$ aus Takt $i-1$ überein, so ist als Operandeneingang für $v$ der ALU-Ausgang der FU zu wählen, auf der $u$ ausgeführt wurde. Liegen die Operationen zwei Takte auseinander, dann ist der Wert aus der jeweiligen Write-Back-Stufe zu wählen.

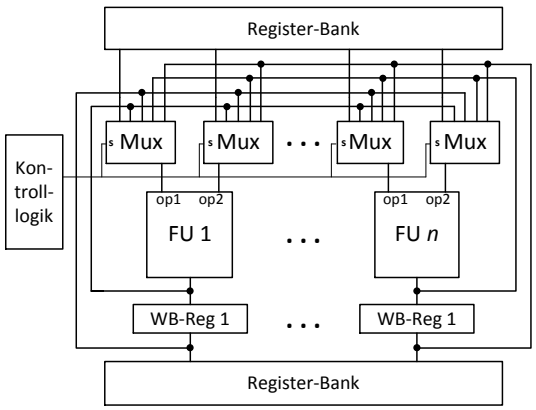


Abbildung 3.3.: Bypass im VLIW-Kern zur Behandlung von Datenkonflikten

Durch den Einsatz einer Fließbandverarbeitung entstehen darüber hinaus Struktur- und Kontrollhazards. Einem Strukturhazard wird dadurch begegnet, dass das System getrennte Speicher für Daten und Programmcode verwendet. Ein Kontrollhazard entsteht bei der Ausführung einer Sprungoperation in Kombination mit einer Fließbandverarbeitung. In der aktuellen Implementierung wird der Programmzähler während der Execute-Stufe geladen. Das bedeutet, dass noch zwei weitere Befehle in die Pipeline geladen werden, bevor zum Sprungziel verzweigt wird. Beim Auftreten eines Kontrollhazards werden die bereits geholten Instruktionen aus der Pipeline entfernt. Dieses

Vorgehen ist transparent für die ausgeführte Software und erleichtert eine softwarebasierte Rekonfiguration.

Die statische Planung zusammen mit dem einfachen Befehlssatz ermöglicht eine reguläre Struktur der Architektur und eine Integration von Teilen der Kontrolllogik in den Datenpfad. Im speziellen ist es möglich, Baugruppen zur Pipeline-Organisation in den Datenpfad zu integrieren. Die Komponenten im Datenpfad können weiter gruppiert werden bezüglich der Komponenten, die jeweils für die Ausführung einer Operation auf einer FU erforderlich sind. Eine solche Gruppierung wird als **Slot** bezeichnet. Abbildung 3.4 zeigt den detaillierten Aufbau eines VLIW-Prozessors mit 2 Slots. Für jeden Slot ist eine FU vorhanden, welche die Berechnungen der Operationen während der Execute-Phase vornimmt. Darüber hinaus zählen zu den Komponenten eines Slots die jeweiligen Pipeline-Register, je Slot zwei Leseports für die Registerbank und ein Schreibport. Darüber hinaus können die Teile des Bypass-Netzwerks auch den einzelnen Slots zugeordnet werden.

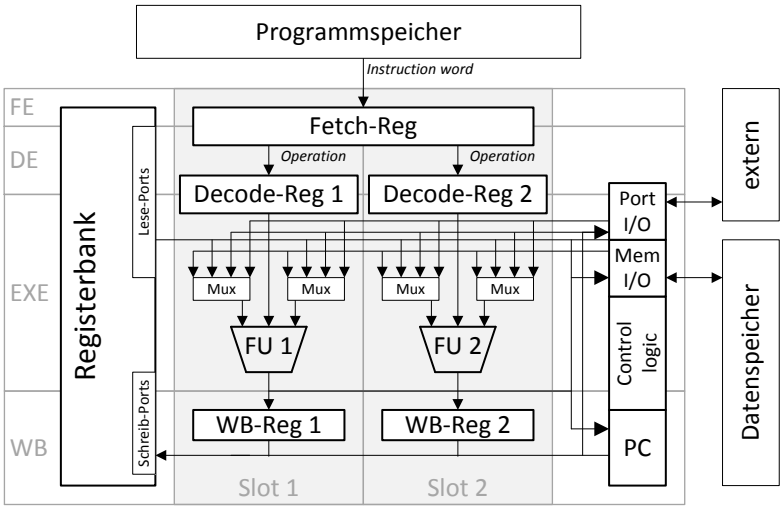


Abbildung 3.4.: Detailierter Aufbau eines VLIW-Kerns mit 2 Slots und den integrierten Komponenten der Kontrolllogik in den Datenpfad

Für die entwickelte Rekonfigurationsstrategie ist diese reguläre Struktur der Architektur von entscheidender Bedeutung. Durch eine Anpassung der statischen Planung wird es möglich, die Verwendung vieler Hardware-Komponenten durch die ausgeführte Software zu verändern, was für andere Architekturen nicht möglich ist. Daraus ergibt sich ein hierarchischer Reparaturansatz, der in Abhängigkeit von der betroffenen Komponente sehr fein-granular vorgehen kann bis hin zu grob-granularen Ersetzungsstrategien.

## 3.2. Aufbau des Programmspeichers

Die entwickelte Reparaturstrategie basiert darauf, die statische Ablaufplanung eines VLIW-Prozessors zu verändern, um die Verwendung defekter Komponenten zu vermeiden. Dazu erfolgt eine Modifikation der Instruktionswörter des Programmspeichers. Die Darstellung des Programmspeichers erfolgt aus Sicht der Reparaturstrategie. Für die Reparaturstrategie ist es relevant, wie ein Programm im Programmspeicher kodiert ist, wie es in einer temporären Darstellung einfach zu repräsentieren ist und wie es in die ursprüngliche Repräsentation überführt werden kann.

Eine **Operation** $v$ beschreibt sich als ein 4-Tupel $v = (type, src1, src2, dst)$. In der ersten Komponente steht der Operationstyp der Operation. Die Komponenten $src1$ und $src2$ spezifizieren die Quellregister der Operation und $dst$ gibt die Nummer des Zielregisters an. Um das Zielregister einer Operation $v$ zu erhalten, wird die Funktion $dst(v)$ verwendet. Die Funktionen $src1(v)$ und $src2(v)$ liefern die Registernummer des ersten bzw. des zweiten Quelloperanden.

Ein **Instruktionswort** $w$, kurz als Instruktion bezeichnet, ist ein $n$-Tupel der Form $w = (v_0, v_1, ..., v_{n-1})$, wobei $n$ der Anzahl an Ausführungsslots eines konkreten VLIW-Prozessors entspricht. Die Elemente einer Instruktion $w$ sind Operationen. Eine an der $i$-ten Stelle stehende Operation $v_i$ wird durch die korrespondierende FU $f_i$ bzw. den Slot $s_i$ ausgeführt. Die Ausführung der Operationen einer Instruktion erfolgt parallel im gleichen Zeitschritt.

Der Programmspeicher $P$ eines konkreten VLIW-Prozessor ist eine Folge von Instruktionen gleicher Länge. Der Index $i$ einer Instruktion $P(i)$ wird auch als Adresse aufgefasst. Für eine Operation $v$ liefert die Funktion $addr(v)$ die Adresse $a$ bezüglich der Instruktion $w_i$ die $v$ enthält, so dass gilt $a = i$. In einem Programmspeicher $P$ ist ein **Basisblock** $b$ maximaler Länge eine Folge von Instruktionen $(w_0, w_1, ..., w_{m-1})$ mit den Eigenschaften, dass:

- die Abarbeitung der Instruktionen sequentiell, beginnend bei $w_0$, erfolgt,
- alle Instruktionen ausgeführt werden und
- lediglich die letzte Instruktion $w_{m-1}$ eine Verzweigungsoperation enthalten darf.

Die Länge eines Basisblocks $b$ ist durch $m$ gegeben, wobei es keinen Basisblock der Länge 0 gibt. Um eine Instruktion $w_i$ an der $i$-ten Position eines Basisblocks $b$ auszuwählen, wird auch $b(i)$ geschrieben. Bezüglich der Instruktionen von $b$ gilt, dass ausschließlich b(0) Ziel einer Verzweigungsoperation aus $P$ sein darf.

Abbildung 3.5 zeigt schematisch die Strukturierung des Programmspeichers. Der dargestellte Programmspeicher bezieht sich auf einen VLIW-Prozessor mit vier Verarbeitungsslots. Die Spalten repräsentieren die Operationen und die Operationen einer Zeile ergeben eine Instruktion. In jedem Ausführungsschritt wird eine Instruktion in den Datenpfad des Prozessors geholt. Die Instruktionen des dargestellten Basisblocks werden nacheinander durch den Prozessor ausgeführt, beginnend bei der Instruktion $w_i$.

Zwischen den Operationen eines Basisblocks $b$ können **Datenabhängigkeiten** bestehen. Datenabhängigkeiten entstehen durch eine zeitlichen Ausführungsreihenfolge

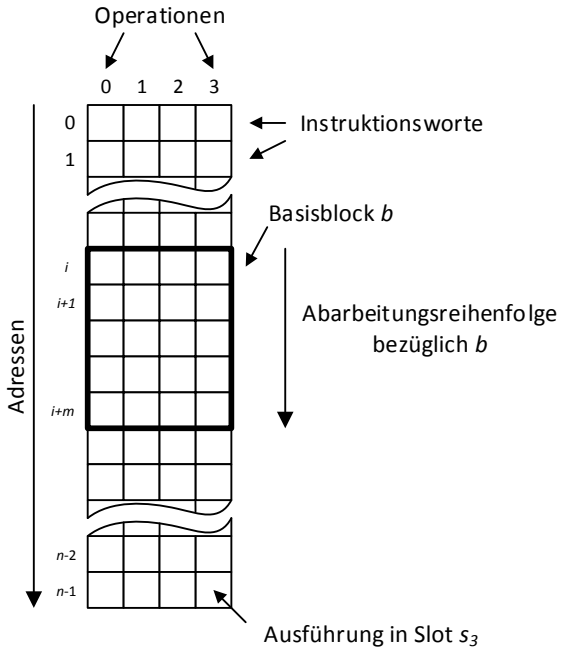


Abbildung 3.5.: Strukturierung des Programmspeichers bezüglich der Operationen und deren Gruppierung in Instruktionen

von Operationen, welche auf das gleiche Datum zugreifen und mindestens ein Zugriff erfolgt schreibend. Die drei Arten der Datenabhängigkeit sind die Flussabhängigkeit (engl. *true dependency*), die Gegenabhängigkeit (engl. *anti dependency*) und die Ausgabeabhängigkeit (engl. *output dependency*). Zwischen zwei Operationen $u$ und $v$ besteht eine Flussabhängigkeit, wenn $u$ ein Datum definiert, welches $v$ zu einem späterem Zeitpunkt liest und zwischenzeitlich keine weitere Definition des Datums erfolgt. Eine Gegenabhängigkeit besteht, wenn eine Operation $u$ ein Datum liest und $v$ zu einem späteren Zeitpunkt das Datum schreibt, ohne dass es zuvor erneut definiert wurde. Eine Ausgabeabhängigkeit besteht, wenn zwei Operationen schreibend auf das gleiche Datum zugreifen.

Abbildung 3.6 zeigt das Layout des Programmspeichers für Funktionen und eine Anwendung. Eine **Funktion** $f$ ist eine Folge von Basisblöcken $(b_0, b_1, ..., b_{k-1})$. Der Einsprung in eine Funktion erfolgt an die erste Instruktion des Basisblocks $b_0$. Die Grenzen einer Funktion $f$ sind durch die Adresse der ersten Instruktion von $b_0$ und die Adresse der letzten Instruktion von $b_{n-1}$ gegeben. Das Verlassen einer Funktion erfolgt mit der letzten Instruktion von $b_{n-1}$ anhand einer speziellen *return*-Operation. Basisblöcke die zu verschiedenen Funktionen $f$ und $g$ gehören, dürfen nur zu $f(0)$ bzw. $g(0)$ verzweigen. Ein Basisblock $b$ einer Funktion $f$ darf zu allen Basisblöcken von $f$ verzweigen.

Wie in Abbildung 3.6 gezeigt, werden die Funktionen in einer **Anwendung** *app* zusammengefasst. Diese setzt sich aus mehreren Funktionen $(func_0, func_1, ..., func_{l-1})$ zusammen. Wie in der Darstellung zu sehen, muss eine Anwendung nicht zwingend an der ersten Adresse des Programmspeichers beginnen. Allerdings muss dann der Einsprung in die Anwendung in geeigneter Weise erfolgen. In der Darstellung ist weiterhin angedeutet, dass eine Anwendung nicht den gesamten Programmspeicher belegen muss. Es wird vereinbart, dass es in einem Programmspeicher nur eine Anwendung gibt. Daraus ergibt sich, dass ein VLIW-Prozessor nur eine Anwendung ausführt.

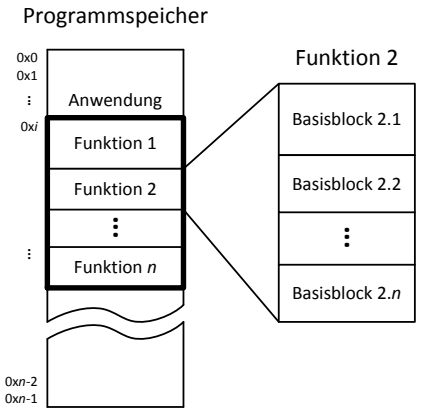


Abbildung 3.6.: Strukturierung des Programmspeichers bezüglich der Anwendung, Funktionen und Basisblöcken

## 3.3. Systemmodell

Das in dieser Arbeit verwendete Systemmodell stellt sich als Multiprozessor-System-on-a-Chip dar, kurz als **MPSoC** bezeichnet. Die im System eingesetzten Prozessoren basieren auf einer VLIW-Architektur, analog zu dem bereits beschriebenen Prozessormodell. Abbildung 3.7 zeigt eine schematische Darstellung des verwendeten Systemmodells. Das System besteht aus einer Menge von VLIW-Prozessorkernen $(c_1, ..., c_n)$. Alle Komponenten des Systems werden durch einen systemweiten Zeitgeber gleich getaktet. Zu den $n$ Prozessorkernen existieren $n$-viele Programmspeicher $(p_1, ..., p_n)$. Die Programmspeicher entsprechen der in Abschnitt 3.2 beschriebenen Struktur. Ein Kern $c_i$ liest sein Programm aus dem Programmspeicher $p_i$. Ein Speicher $p_i$ ist an die Architektur des entsprechenden Kerns angepasst und jeder Programmspeichereintrag enthält genau so viele Operationen wie $c_i$ Slots besitzt. In dem System ist ein gemeinsamer **Datenspeicher** vorhanden. Der Datenspeicher ist eine Folge von Speichereinträgen, die alle die gleiche Anzahl an Bits aufweisen. Für verschiedene Systeme kann die Bitbreite des Datenspeichers variieren. Alle Kerne des Systems haben Zugriff auf die Speicherinhalte des Datenspeichers. Dadurch besteht die Möglichkeit, dass in einem Zeitschritt

mehrere Zugriffe auf den Datenspeicher erfolgen. Die Synchronisation gleichzeitiger Datenspeicherzugriffe erfolgt durch das Verbindungsnetzwerk, weil die Speicher nur einen Zugriff je Takt verarbeiten können.

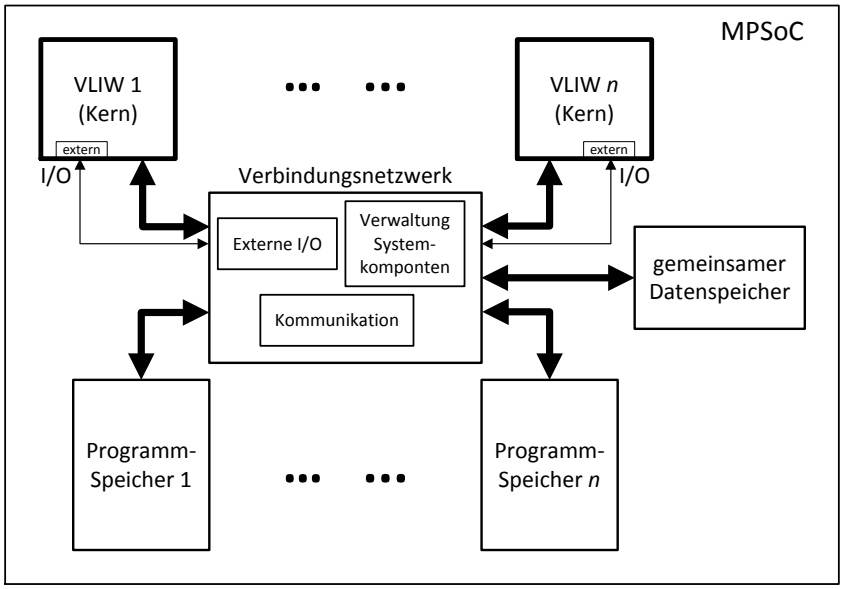


Abbildung 3.7.: Schematische Darstellung eines MPSoC mit VLIW-Kernen, Programm- und Datenspeicher und Verbindungsnetzwerk

Die VLIW-Kerne des Systems sind identisch bezüglich des Befehlssatzes, können allerdings heterogen bezüglich der Konfiguration des Datenpfades sein. Die Datenpfade der Kerne können sich in der Anzahl der allgemeinen Register, der Funktionseinheiten und deren Operatorkonfiguration und der Anzahl der Slots unterscheiden. In jedem Programmspeicher $p_i$ befindet sich genau eine Anwendung, die durch den zugehörigen Prozessorkern $c_i$ ausgeführt wird. Die Anwendungen können Informationen über den gemeinsamen Datenspeicher oder über I/O-Ports austauschen. Die Verwaltung der I/O-Ports erfolgt über das Verbindungsnetzwerk.

Auf die Modellierung von Caches und Speicherhierarchien wurde verzichtet, weil dies keine Auswirkung auf die eingesetzte Reparaturstrategie hat. Somit werden die Zugriffszeiten auf die Speicher als ideal betrachtet, wobei eventuelle Verzögerungen fixiert sind und sich durch eine Priorisierung bei gleichzeitigen Speicherzugriffen erhöhen. Das Lesen einer Instruktion aus dem Programmspeicher dauert einen Zeitschritt. Der Datenspeicher benötigt zwei Zeitschritte zur Bearbeitung von lesenden und auch von schreibenden Zugriffen.

Das Verbindungsnetzwerk bildet einen zentralen Punkt der Systemarchitektur und stellt Funktionen zur Kommunikation und Synchronisation bereit. Die Anforderungen an das Verbindungsnetzwerk sind:

- Bereitstellung einer I/O-Anbindung der Kerne, um mit Systemkomponenten und externen Schnittstellen kommunizieren zu können,
- Verwaltung des Zugriffs auf den Programm- und den Datenspeicher und
- Transfer von Inhalten zwischen dem Programm- und dem Datenspeicher.

Diese Punkte können als Schnittstelle aufgefasst werden, die von einer beliebigen Implementierung des Verbindungsnetzwerkes zu erbringen sind, unabhängig davon, ob es sich um eine netzwerk- oder eine bus-basierte Implementierung handelt. Für die in dieser Arbeit entwickelte software-basierte Rekonfiguration ist der letzte Punkt von entscheidender Bedeutung. Die Rekonfiguration führt Anpassungen an der Anwendung im Programmspeicher durch. Die vorgestellte Systemarchitektur entspricht einer Harvard-Architektur mit getrennten Programm- und Datenspeichern, die einen solchen Zugriff nicht unterstützt. Der erforderliche Transfer von Speicherinhalten wird durch einen Controller des Verbindungsnetzwerks bereitgestellt.

Eine schematische Darstellung des in dieser Arbeit implementierte Verbindungsnetzwerkes zeigt Abbildung 3.8. Es handelt sich dabei um eine Bus-basierte Implementierung, in der das Verbindungsnetzwerk als Bus-Master fungiert. Das Verbindungsnetzwerk setzt sich aus den notwendigen Leitungen, zwei Controller-Bausteinen und einer Register-Bank zum Zwischenspeichern von Daten zusammen.

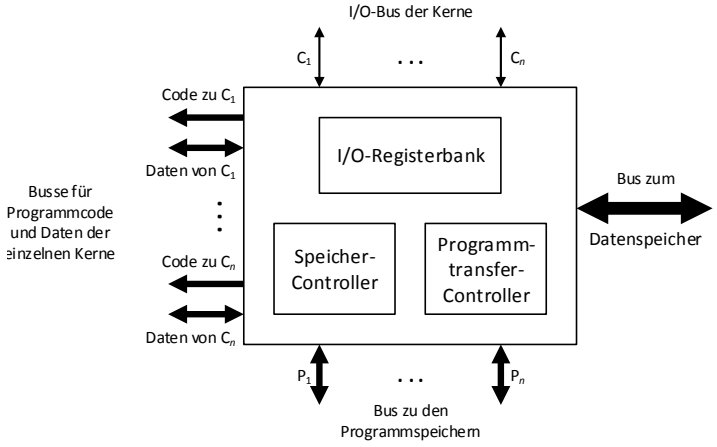


Abbildung 3.8.: Verfeinerte Darstellung zum Verbindungsnetzwerk, mit angeschlossenen Bussen und internen Komponenten

Die Registerbank wird zur systeminternen Kommunikation und zum systemexternen Datenaustausch verwendet. Jedem Prozessor steht eine I/O-Port-Anbindung zur Verfügung, mit der auf die Registerbank des Verbindungsnetzwerkes zugegriffen wird. Die Register sind frei verfügbar und können von jedem Kern wahlweise gelesen oder geschrieben werden. Im Prozessorkern erfolgt die Ansteuerung mit Hilfe von I/O-Operationen. Die Operation sind `out src,dst` und `in src,dst` und diese transferieren den Registerinhalt eines internen Registers in die externe Registerbank oder umgekehrt.

Die externen Register werden auch zur Steuerung von Controllerbausteinen eingesetzt. Der Programmstransfer-Controller des Verbindungsnetzwerkes wird über die externen Register gesteuert. Die Aufgabe des Programmtransfer-Controllers ist es, Speichertransfers zwischen dem Programmspeicher und dem Datenspeicher bereitzustellen. Dadurch wird Zugriff auf die Instruktionsworte ermöglicht, damit diese durch die Rekonfiguration verändert werden können. Der Programmtransfer-Controller kann in zwei Modi arbeiten. Im ersten Modus werden Instruktionen vom Programmspeicher in den Datenspeicher überführt. Der zweite Modus schreibt Inhalte aus dem Datenspeicher in den Programmspeicher. Die Ansteuerung erfolgt über drei externe Register. Zwei dieser Register spezifizieren die Adressen für den Programm- bzw. Datenspeicher. Mit dem Schreiben in das dritte Register wird der jeweilige Modus gewählt und der Transfervorgang initiiert. Während der Transferoperation werden alle anderen Ressourcenzugriffe blockiert und alle Kerne setzen mit der Abarbeitung aus. Der Transfer dauert mehrere Takte, weil das Instruktionswort in mehrere Teile zerlegt und nacheinander in den Datenspeicher geschrieben wird. Für das Zurückschreiben werden die einzelnen Teile einer Instruktion schrittweise aus dem Datenspeicher gelesen und im Controller gepuffert. Befindet sich das Instruktionswort vollständig im Puffer, wird es am Ende in den Programmspeicher geschrieben.

Das Versetzen der Kerne in einen Pausemodus erfolgt mit dem Setzen eines entsprechenden *stall*-Signals. Dieses Signal wird auch verwendet, wenn der Speichercontroller bei gleichzeitigen Datenspeicherzugriffen eine Synchronisation vornehmen muss. Die Synchronisation erfolgt anhand einer Priorisierung der Kerne, basierend auf der vergebenen $ID$. Die $ID$ ergibt sich aus einer Durchnummerierung der vorhandenen Kerne. Durch die Priorisierung wird dem Kern mit der kleinsten $ID$ der exklusive Speicherzugriff gewährt, während die anderen Kerne warten müssen. Dabei handelt es sich um keine faire Strategie mit der Gefahr, dass Kerne dauerhaft warten müssen.

## 3.4. Das Prinzip der software-basierten Rekonfiguration

Das in dieser Dissertation entwickelte hierarchische Verfahren zur software-basierten Rekonfiguration basiert auf dem Prinzip der software-basierten Reparatur statisch geplanter VLIW-Prozessoren [84]. Die Idee dabei ist es, die Instruktionswörter einer VLIW-Architektur in der Form anzupassen, dass sie die Verwendung defekter Prozessorbaugruppen vermeiden. Die Anpassung der Instruktionsworte erfolgt im Programmspeicher und wird dort dauerhaft hinterlegt. Die Berechnungen zur Rekonfiguration werden durch eine Software-Routine vorgenommen. Die Ausführung der Reparaturroutine erfolgt durch den defekten Prozessor während des Systemstarts. Abbildung 3.9 zeigt beispielhaft das Vorgehen für einen VLIW-Prozessor mit 4 FUs im Datenpfad.

Abbildung 3.9(a) zeigt das ursprüngliche System mit 3 Instruktionsworten im Programmspeicher. In der Abbildung 3.9(b) wird ein Fehler in der dritten FU angenommen. Die Operationen, die ursprünglich auf der defekten Komponente ausgeführt werden sollen, werden durch das Reparaturverfahren auf eine redundante FU verschoben. Dazu werden in jeder Instruktion die FUs genutzt, die eine leere Operation ausführen. Diese Form der Redundanz kann natürlich vorhanden sein, weil die Anwendung nicht genügend Parallelität auf Instruktionswortebene bereitstellt oder

sie wird zusätzlich in Form redundanter Slots in das fehlertolerante System eingebracht. Das Verfahren in Abbildung 3.9(b) verändert die Bindung der Operationen innerhalb einer Instruktion und wird dementsprechend als Rebinding bezeichnet [84, 86].

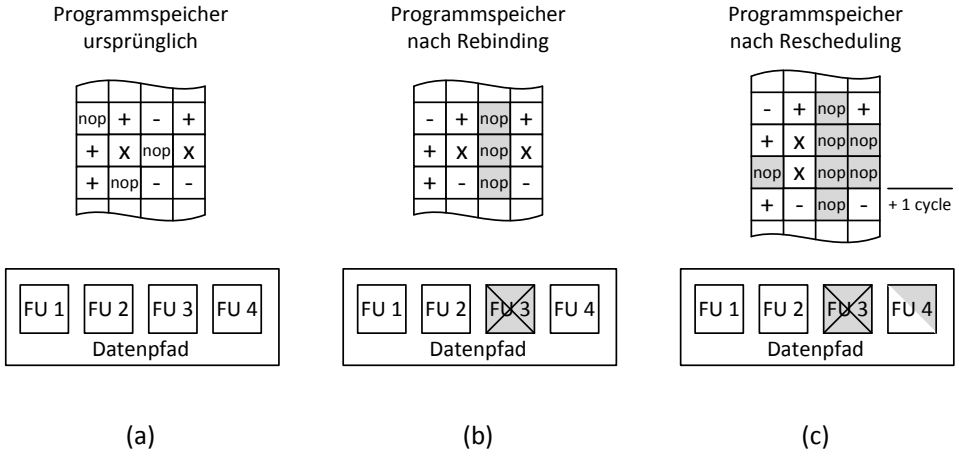


Abbildung 3.9.: Anpassung der originalen Instruktionswörter (a) an einen defekten Datenpfad mit Hilfe einer Hardware-Redundanz (b) und zusätzlich durch eine zeitliche Redundanz (c)

Eine Verbesserung stellt das Rescheduling dar [86], welches in Abbildung 3.9(c) angewandt wurde, um einen weiteren Fehler im Multiplikations-Operator von FU 4 zu behandeln. Es ist zu beachten, dass FU 4 weiterhin andere Operationen berechnen kann. Das nun eingesetzte Rescheduling arbeitet zusätzlich in der zeitlichen Dimension und verschiebt Operationen notfalls in andere Instruktionen. Dabei können weitere Instruktionen entstehen, welche die Ablaufplanung eines Basisblocks verlängern können. Das Verfahren arbeitet auf der Granularität von Basisblöcken und erzeugt für diese eine neue Ablaufplanung. Im Beispiel aus 3.9(c) wird die Mulitplikation der zweiten Instruktion in eine weitere Instruktion geplant. Zur Laufzeit erfordert die Ausführung der so geänderten Ablaufplanung einen zusätzlichen Zeitschritt. Durch eine Degradation der Systemleistung konnte somit ein weiterer Fehler in der vierten FU behandelt werden.

Im Vergleich zu einem hardware-basierten Verfahren, welches dynamisch die Anpassung der Bindung im Datenpfad einer VLIW-Architektur vornimmt [84], hat das zuvor beschriebene Vorgehen die folgenden Vorteile:

- Eine Behandlung permanenter Fehler zur Laufzeit ist nicht erforderlich, da die Instruktionen dauerhaft geändert werden.
- Der notwendige Mehraufwand an Hardware zur Administration einer Reparatur kann minimal gehalten werden.

- Eine software-basierte Reparatur ist flexibel einsetzbar und einfach zu skalieren in Bezug auf sich ändernde Systemparametern.

Für hardware-basierte Verfahren kann es erforderlich sein, zusätzliche Reparaturtakte durchzuführen, um eine Operation $v$ auf einer fehlerfreien Komponente berechnen zu lassen. Dieser Vorgang muss jedes Mal erneut ausgeführt werden, falls $v$ wiederholt zur Ausführung kommt, wenn beispielsweise Schleifen durchlaufen werden. Beim Einsatz einer software-basierten Rekonfiguration ist eine erneute Anpassung zur Laufzeit nicht notwendig, da hier die Instruktionen statisch im Programmspeicher geändert werden.

Der zweite Punkt begründet sich dadurch, dass der benötigte Mehraufwand an Hardware-Strukturen sich lediglich auf eine Möglichkeit zum Transfer von Daten zwischen Programm- und Datenspeicher beschränkt.

Ein flexibles Einsatzspektrum der software-basierten Rekonfiguration ergibt sich dadurch, dass die Reparaturverfahren auf unterschiedlichen Ebenen einsetzbar sind und sich nicht auf die Behandlung des Fehlers einer einzigen Prozessorbaugruppe beschränken. Eine Skalierbarkeit wird deshalb leicht unterstützt, weil die Reparaturroutinen in Abhängigkeit von Systemparametern, wie zum Beispiel der Anzahl an Slots oder FUs, entwickelt werden und eine Änderung der Parameter einfach zu berücksichtigen ist.

Durch den Einsatz einer software-basierten Reparatur ergeben sich die folgenden relevanten Konsequenzen:

- Wenn nicht genügend natürlich Redundanz vorhanden ist, muss diese zusätzlich in Form von Spare-Slots oder Programmspeichereinträgen bereit gestellt werden.
- Es muss in geeigneter Weise ein Zugriff auf den Programmspeicher gewährt werden (vgl. Abschnitt 3.3).
- Die Auswirkungen auf den Programmspeicher durch eine verlängerten Ablaufplanung von Basisblöcken müssen beachtet und behandelt werden.

In Abbildung 3.9(c) wurde bereits darauf hingewiesen, dass eine Verschiebung von Operationen in der zeitlichen Dimension zusätzliche Instruktionen erfordern kann, was zu Verschiebungen im Programmspeicher und geänderten Adressbezügen führt. Die sich dadurch geänderten Ziele von Sprunganweisungen müssen durch die software-basierte Rekonfiguration behandelt werden. Darüber hinaus muss dafür Sorge getragen werden, dass Codebereiche nicht überschrieben werden. Die notwendigen Maßnahmen zur Behandlung solcher Effekte durch den Einsatz einer software-basierten Rekonfiguration stellen einen wesentlichen Schwerpunkt der vorliegenden Arbeit dar.

#### Die hierarchische software-basierte Rekonfiguration

Die in dieser Dissertation entwickelten Methoden für die software-basierte Rekonfiguration basieren auf den zuvor präsentierten Ideen des Rebindungs und des Reschedulings. Die entwickelten Verfahren integrieren die vorhandenen Ansätze in eine hierarchisch organisierte kernübergreifende Reparatur eines Mehrkernsystems, welches mit zusätzlichen Verfahren zur Behandlung weiterer Prozessorbaugruppen kombiniert wird. Zunächst werden zwei hierarchische Ebenen unterschieden:

- die lokale Kernebene, die sich auf die Rekonfiguration eines einzelnen VLIW-Kerns beschränkt, und
- die globale Systemebene, welche das gesamte Mehrkernsystem mit mehreren Prozessorkernen umfasst.

Die auf der Kernebene eingesetzten Reparaturverfahren werden autonom durch jeden Kern selbst ausgeführt. Zur Behandlung permanenter Fehler verwenden die Verfahren redundante Prozessorbaugruppen innerhalb eines Kerns und darüber hinaus kann eine Kombination in der zeitlichen Dimension mit einer degradierten Systemleistung erfolgen. Das Ziel der eingesetzten Verfahren auf der Systemebene ist es, Redundanzen kernübergreifend auszunutzen unter Wiederverwendung der lokalen Reparaturverfahren.

Die grundlegende Idee des entwickelten Reparaturverfahrens ist es, die Anwendung eines Systems erneut zu übersetzen und während des Übersetzungsvorgangs die sich geänderten verwendbaren Systemressourcen zu berücksichtigen. Die Übersetzung erfolgt im Feld durch das System selbst. Dabei werden Techniken verwendet, die für die Zielcodeerzeugung in der Compilertechnik eingesetzt werden. Die Struktur des Reparaturverfahrens ist in Abbildung 3.10 gezeigt.

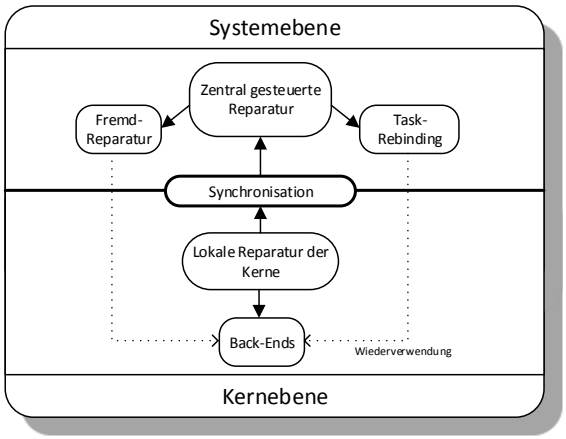


Abbildung 3.10.: Ablauf zur Organisation der software-basierten Reparatur in einem Mehrkernsystem

Auf der untersten Ebene, der Kernebene, erfolgt eine lokale Reparatur die sich auf einen VLIW-Kern beschränkt. Die Reparaturroutinen werden vom jeweiligen VLIW-Kern selbst ausgeführt. Auf der lokalen Ebene stehen unterschiedlich mächtige **Backends**[1] zur Verfügung. In dieser Arbeit bezeichnet ein Backend die verschiedenen Softwareroutinen, die den Programmcode einer Anwendung gezielt an eine bestimmte Art einer ausgefallenen Prozessorbaugruppe anpassen können. In Abbildung 3.9 wurden

[1] In der Compilertechnik bezeichnet ein Backend den Teil eines Compilers, der aus einer Zwischencodedarstellung den Programmcode für die jeweilige Zielarchitektur erzeugt.

bereits zwei Backends vorgestellt, das Rebindung und das Rescheduling. Das Rescheduling ist mächtiger als das Rebindung, da es zusätzlich in der zeitlichen Dimension arbeitet und dadurch andere Kombinationen von Fehlern behandeln kann. Beide Verfahren können allerdings nicht den Ausfall eines Registers der Registerbank behandeln. Dazu werden die Backends Register-Renaming und Register-Allokation eingesetzt.

Die obere Ebene der Reparatur ist die Systemebene, die auch als globale Reparatur bezeichnet wird. Die Durchführung einer globalen Reparatur ist dann erforderlich, wenn die lokale Reparatur eines Kerns nicht erfolgreich war. Eine lokale Reparatur kann aus den folgenden Gründen fehlschlagen:

- Der Reparaturalgorithmus ist durch den defekten Kern nicht ausführbar und
- es konnte keine gültige Ablaufplanung für alle Basisblöcke einer Anwendung bestimmt werden.

Da es sich auch bei der lokalen Reparatur um eine Selbstreparatur handelt, kann es passieren, dass der Reparaturalgorithmus selbst eine fehlerhafte Prozessorbaugruppe verwendet. In diesem Fall kann nicht garantiert werden, dass eine korrekte Übersetzung der Anwendung erzeugt wird. Der zweite Fall kann eintreten, wenn zu viele Fehler vorhanden sind. Dadurch können sich zu lange Ablaufpläne ergeben, die zeitliche Beschränkungen verletzen oder nicht im verfügbaren Programmspeicher hinterlegbar sind, da die erforderliche Größe nicht ausreicht.

Für die beiden Fälle stehen der globalen Reparatur die Backends Fremdreparatur und Task-Rebinding zur Verfügung. Aufgabe der Fremdreparatur ist es, die Software-Routine der Selbstreparatur eines fehlerhaften Kerns an die ausgefallenen Baugruppen anzupassen. Anschließend führt der fehlerhafte Kern die so angepassten Reparaturroutinen selbst aus. Das zweite Backend ändert die Zuordnungen zwischen den Anwendungen und den Kernen eines Systems mit dem Ziel, eine weniger ressourcenlastige Anwendung einem fehlerhaften Kern zuzuweisen. Durch die globalen Reparaturverfahren erfolgt eine Wiederverwendung von den Backends der lokalen Ebene. Der Übergang von der lokalen in die globale Reparatur erfolgt durch eine Synchronisation der Kerne des Systems. Der wesentliche Bestandteil der Synchronisation liegt darin, dass die Kerne des Systems einen Kern bestimmen, der den globalen Reparaturvorgang administriert. Somit erfolgt die globale Reparatur zentral gesteuert durch lediglich einen Kern.

### Die Backends und der Defektspeicher

Die Backends des Reparaturverfahrens können unterschiedliche Anpassungen der Anwendung an eine vorliegende Fehlersituation leisten. Eine Übersicht zu den zur Verfügung stehenden Backends zeigt Tabelle 3.1. Die ersten vier Backends arbeiten auf der lokalen Ebene und nehmen Anpassung der Anwendung bezüglich eines Kerns vor. Die letzten beiden Backends werden auf der Systemebene eingesetzt. Für jedes Backend ist weiterhin angegeben, ob es ausschließlich eine Hardware-Redundanz einsetzt oder diese zusätzlich mit einer Degradation der Systemleistung kombiniert.

Einige Backends sind vielseitig einsetzbar und kommen beim Ausfall einer breiten Anzahl an Prozessorbaugruppen zum Einsatz. Abbildung 3.11 visualisiert die hierarchische Aufteilung der Prozessorbaugruppen in die verschiedenen Ebenen und die redundanten

| Backend | Ebene | Verwendete Redundanz |
|---|---|---|
| Rebinding | lokal | Hardware-Redundanz |
| Rescheduling | lokal | Hardware-Redundanz und Degradation |
| Renaming | lokal | Hardware-Redundanz |
| Registerallokation | lokal | Hardware-Redundanz und Degradation |
| Fremdreparatur | global | Hardware-Redundanz |
| Task-Rebinding | global | Hardware-Redundanz und Degradation |

Tabelle 3.1.: Übersicht über die Backends, deren Einsatzgebiet und die verwendete Art der Fehlerkompensation

Komponenten, die bei einer Rekonfiguration berücksichtigt werden. Die feinste Granularität bietet eine Rekonfiguration auf Operator-Level. Mit steigender Granularität ist auf Systemebene die gröbste Granularität erreicht. Die grauen Boxen in Abbildung 3.11 repräsentieren die redundanten Baugruppen einer Ebene, für die eine bestimmte Reparaturstrategie eingesetzt wird. Die regulären Boxen fassen diese Baugruppen als eine Ebene zusammen. Ist eine Anwendungsrekonfiguration auf einer Ebene nicht erfolgreich, kann dies auf dem nächst höheren Level mit einem anderen Backend wiederholt werden.

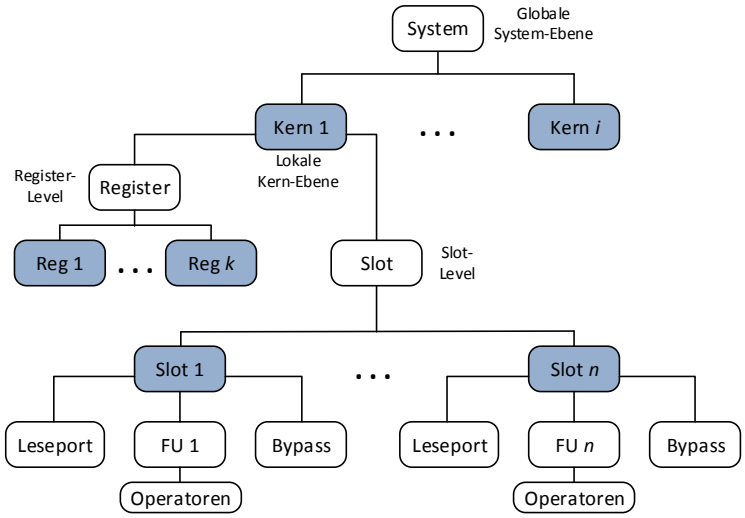


Abbildung 3.11.: Hierarchie der Prozessorbaugruppen mit redundanten Gruppen auf den einzelnen Ebenen

Tabelle 3.2 zeigt eine Zuordnung zwischen der Komponentenhierarchie und dem jeweiligen Backend, das beim Ausfall einer entsprechenden Baugruppe verwendet wird. Die Backends Rebinding und Rescheduling sind flexibel auf mehreren Levels einsetzbar. Fehler in den Registern sind nur durch die Backends Renaming und Registerallokation behandelbar. Auf der Systemebene erfolgt eine Wiederverwendung der lokalen Backends durch die Backends Fremdreparatur und Task-Rebinding.

| Komponente/Level | Bedeutung | Reparaturmethode |
|---|---|---|
| Operator | Operator/FU fehlerhaft | Rebinding, Rescheduling |
| Funktionseinheit | FU/Slot fehlerhaft | Rebinding, Rescheduling |
| Bypass | Bypass/Slot fehlerhaft | Rebinding, Rescheduling |
| Portanbindung | Port/Slot fehlerhaft | Rescheduling |
| Register | Register fehlerhaft | Renaming, RegAllok |
| Slot | Slot fehlerhaft | Rebinding, Rescheduling |
| Kern | Kern fehlerhaft | Remote-Repair, Task-Rebinding |

Tabelle 3.2.: Komponenten eines VLIW-Prozessors bzw. Mehrkernsystem, die Konsequenz bei Ausfall und die passende mögliche Reparaturstrategie

Damit während der Reparatur entschieden werden kann, welches Backend einzusetzen ist, muss die vorliegende Fehlersituation innerhalb des Systems in geeigneter Form abgebildet werden. Dazu kommt ein **Fehlerspeicher** zum Einsatz, der als Datenstruktur im gemeinsamen Datenspeicher des Systems hinterlegt ist. Der Aufbau des Fehlerspeichers leitet sich davon ab, welche Informationen bezüglich des Hardware-Zustandes relevant sind, damit das jeweils eingesetzte Backend fehlerhafte Komponenten von der Planung ausschließt und durch redundante ersetzen kann. Der Fehlerspeicher beschreibt somit die vorliegende Fehlersituation und stellt eine Repräsentation des Zustands der Systemkomponenten dar. Der Zustand einer Komponente kann fehlerfrei oder fehlerhaft sein. Der Fehlerspeicher dient darüber hinaus als Schnittstelle zwischen der Fehlererkennung und der Rekonfiguration.

Der Fehlerzustand $fss$ des Systems setzt sich aus den Zuständen der einzelnen Kerne zusammen $(fsc_1, ..., fsc_n)$. Den Zustand eines Kerns $c_i$ beschreibt das 5-Tupel

$$fsc_i = (fsRegs_i, fsSlots_i, fsFUs_i, fsPorts_i, fsBypass_i)$$

anhand der Zustände der Register, der Slots, der FU-Konfiguration, der Leseports und des Bypasses. Die Funktion $fsRegs_i : \{0, ..., k-1\} \rightarrow \{0, 1\}$ gibt an, ob ein Register $k$ verwendet werden kann. Für einen Slot $s_n$ bestimmt $fsSlots_i : \{0, ..., n-1\} \rightarrow \{0, 1\}$ den Fehlerzustand.

Für die Konfiguration einer FU werden die folgenden Operationstypen unterschieden:

- Arithmetische Operationen, einzeln nach Addition, Subtraktion, Multiplikation und Division
- Logische und Bitoperationen
- Speicherzugriffsoperationen
- Sprung- und Verzweigungsoperationen

Die maximale Anzahl an unterstützten Operationstypen durch eine FU ist auf $types = 7$ festgelegt. Für eine FU $f_n$ gibt

$$fsFUs_i : \{0, ..., n-1\} \times \{0, .., types-1\} \rightarrow \{0, 1\}$$

an, ob die FU einen fehlerfreien Operator für den entsprechenden Operationstyp bereitstellt. Jede FU verfügt über zwei Leseports, um ein Datum aus der Registerbank zu lesen. Für die Bestimmung der Zustände der Leseports wird die Funktion

$$fsPorts_i : \{0, ..., n-1\} \times \{0, 1\} \rightarrow \{0, 1\}$$

angegeben. Mit ihr wird für eine FU $f_n$ ermittelt, ob der linke beziehungsweise der rechte Leseport defekt ist. Für jeden Operanden, den eine FU verarbeiten kann, existiert ein Bypass. Ein Forwarding kann aus dem Takt $i-1$ oder $i-2$ aus einem beliebigen Slot des jeweiligen Kerns erfolgen. Mit der Funktion

$$fsBypass_i : \{0, .., n-1\} \times \{0, .., m-1\} \times \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$$

wird angegeben, ob für einen Slot $s_n$ der jeweilige Bypass aus dem entsprechenden vorangegangenen Takt ein gültiges Ergebnis von der FU $f_m$ bereitstellen kann.

# 4. Lokale Rekonfiguration innerhalb eines Kerns

In diesem Kapitel werden die Techniken beschrieben, mit denen eine Anpassung der Anwendung auf der lokalen Kernebene erfolgt. Die Anpassung erfolgt durch eine Rekonfiguration der Anwendung im Programmspeicher. Die erforderlichen Schritte zur Rekonfiguration überschneiden sich mit den eingesetzten Techniken zur Zielcodeerzeugung, wie sie aus der Compilertechnik bekannt sind. Aufgrund gewisser Randbedingungen sind jedoch die bekannten Methoden nicht in vollem Umfang auf die vorliegende Problemstellung übertragbar. Die sich daraus ergebenden Besonderheiten in den Backends sowie zur Organisation und Administration der software-basierten Rekonfiguration bilden den Schwerpunkt des folgenden Kapitels.

Im weiteren Verlauf wird zunächst der Ablauf einer lokalen Rekonfiguration dargestellt. Dazu werden die Randbedingungen präsentiert, unter denen diese Strategie Anwendung findet. Aus den Randbedingungen ergeben sich Konsequenzen und notwendige Teilschritte für die Abarbeitung der Rekonfiguration. Begleitend werden die Details der Backends zur Reparatur einer ausgefallenen Komponente herausgearbeitet. Aus dem Einsatz des jeweiligen Backends ergeben sich gewisse Konsequenzen für die geplante Reparaturstrategie. Mit den sich daraus ergebenden Problemstellungen beschäftigt sich der letzte Abschnitt dieses Kapitels.

## 4.1. Ablauf während der lokalen Reparatur

Im Idealfall würde die eingesetzte Reparaturstrategie für eine Anwendung eine vollständigen Neuübersetzung durchführen, um diese optimal an den vorliegenden Defektzustand anzupassen. Aufgrund des eingebetteten Systemumfelds ist dies nicht anwendbar und es sind einige Rahmenbedingungen zu beachten. Zunächst ist zu berücksichtigen, dass die Rechenleistung der einzelnen Kerne an die Systemspezifikation angepasst ist und somit weit entfernt von der Leistung von Desktoprechnern liegt. Ähnlich verhält es sich mit der verfügbaren Menge an Programm- und Datenspeichern, die im eingebetteten Umfeld signifikant geringer sind. Dadurch muss für die Planungsverfahren eine Abwägung zwischen Rechenleistung, Speicherplatz und der Qualität der Rekonfiguration getroffen werden. Dabei wird unter anderem auf die Möglichkeit zurückgegriffen, den Reparaturverfahren im Voraus berechnete Informationen bereitzustellen, um einen Teil der Komplexität von der Reparaturzeit in die Systementwicklungszeit zu verschieben. Zusammenfassend ergeben sich die folgenden Anforderungen an die Algorithmen der Reparaturstrategie:

1. Der Programmcode der Reparaturroutinen soll gering gehalten werden, damit der notwendige Mehraufwand nicht den Programmspeicher dominiert.

2. Der Einsatz komplexer Datenstrukturen (z.Bsp. Graphen oder Bäume) durch die Reparaturalgorithmen ist nicht möglich, weil die Datenspeichergröße beschränkt ist.

3. Unter Berücksichtigung der Rechenleistung und Einhaltung von gewissen Zeitanforderungen können lediglich Planungsverfahren mit einfachen Heuristiken eingesetzt werden, die unter Umständen auch auf im Voraus berechnete Daten zurückgreifen.

Weitere Einschränkungen ergeben sich aufgrund von globalen Optimierungen, die durch den Compiler während der Systemübersetzungszeit vorgenommen werden. In einem typischen Übersetzungsvorgang werden in einer späten Phase globale Optimierungen durchgeführt. Diese betreffen beispielsweise Schleifenzähler, die in global verwendeten Registern gehalten werden. Diese Register besitzen dann eine basisblockübergreifende Gültigkeit. Den Algorithmen des Compilers stehen dazu globale Informationen in entsprechenden Datenstrukturen zur Verfügung. Die Reparaturalgorithmen der vorliegenden Dissertation können diese Informationen jedoch nicht rekonstruieren, da sie dazu den Eingabeprogrammtext als Ganzes betrachten müssen. Aufgrund der Ressourcenbeschränkung ist das nicht möglich, und es erfolgt lediglich eine schrittweise Verarbeitung des Binärcodes einer Anwendung. Um dennoch eine umfangreiche Rekonfiguration an einer Anwendung vornehmen zu können, werden Informationen bezüglich eines globalen Kontext im Voraus berechnet und den Reparaturverfahren als Meta-Daten zur Verfügung gestellt.

Der grobe Ablauf, um eine Anwendung zu rekonfigurieren, erfolgt anhand der folgenden Vorgehensweise. Zunächst wird die Anwendung schrittweise aus dem Programmspeicher in den Datenspeicher überführt. Weil die kopierten Instruktionsworte nur in codierter Form vorliegen, werden sie im nächsten Schritt dekodiert und zwischengespeichert. Anschließend erfolgt die Anpassung der Instruktionen an den Fehlerzustand mit Hilfe eines der vorhandenen Planungsverfahren. Die Auswahl des passenden Backends erfolgt anhand des Fehlerzustands. Abschließend werden die Operationen wieder in Instruktionswörter codiert und in den Programmspeicher zurückgeschrieben. Im letzten Schritt erfolgt ein Überschreiben der ursprünglichen Instruktionen durch die neu geplanten Instruktionsworte. Solange die ursprüngliche Anzahl an Instruktionsworten für einen bearbeiteten Abschnitt gleich bleibt, ergeben sich keine Probleme durch die Überschreibungen im Programmspeicher.

Allerdings können die Planungsverfahren Operationen in spätere Instruktionen verschieben, wodurch sich zusätzliche Instruktionsworte ergeben können. In der Konsequenz bewirkt ein so verlängerter Ablaufplan, dass sich Adressen in den nachfolgenden Speicherbereichen verändern. Dadurch werden Sprungbefehle ungültig, die zu Basisblöcken anhand der ursprünglichen Adresse verzweigen. Darüber hinaus führen verlängerte Ablaufpläne dazu, dass sich Code-Bereiche ohne entsprechende Gegenmaßnahmen überschreiben. Damit die Planungsverfahren eine gültige Umplanung gewährleisten können, erfolgt eine Rekonfiguration immer innerhalb der Grenzen von Basisblöcken. Somit muss die zu rekonfigurierende Anwendung in ihre Basisblöcke zerlegt werden. Die Schwerpunkte zur Organisation einer Rekonfiguration sind demnach:

- Rekonstruktion der Information zu den Basisblöcken,

- Anpassung von Sprungzielen für ungültige Sprungoperationen und
- Bereitstellen und Administration von zusätzlichen Programmspeicherbereichen zur Vermeidung von Überschreibungen.

Unter Berücksichtigung dieser drei Punkte zeigt Abbildung 4.1 eine Verfeinerung der abzuarbeitenden Schritte während der lokalen Rekonfiguration. In der Abbildung stellen eckig dargestellte Boxen Eingabedaten dar, wohingegen abgerundete Boxen Abarbeitungsteilschritte repräsentieren. Gestrichelte Pfeile signalisieren, dass Informationen weitergegeben werden und normale Pfeile verzweigen zum Nachfolger bezüglich des logischen Flusses in der Abarbeitungsreihenfolge.

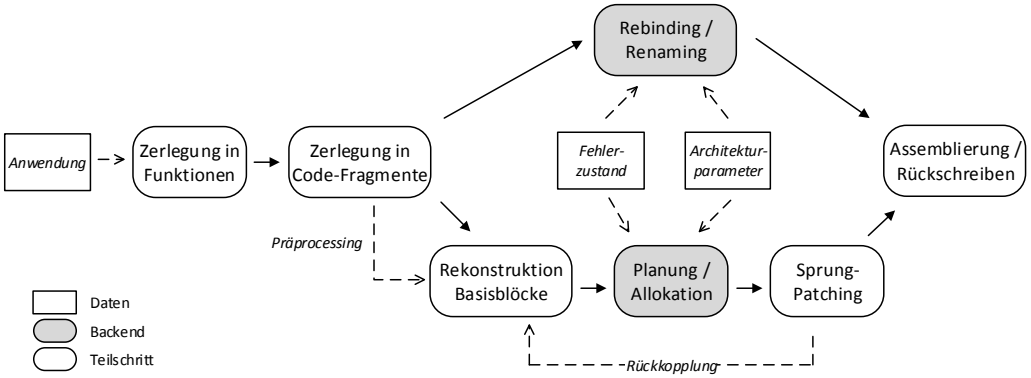


Abbildung 4.1.: Schematische Darstellung der Abläufe zur Steuerung der lokalen Reparatur

Der Prozess der Rekonfiguration beginnt mit der Zerlegung der Anwendung in ihre Funktionen. Dazu wird eine Liste aufgebaut, welche die Start- und Endadressen der vorhandenen Funktionen enthält. Aus Effizienzgründen wird für diesen Vorgang nicht der Programmspeicher nach den entsprechenden Speicheradressen durchsucht. Statt dessen werden im Voraus generierte Daten verwendet, die lediglich in die Liste einzulesen sind.

Anschließend erfolgt eine schrittweise Verarbeitung der Funktionen. Bevor die Bearbeitung des eigentlichen Programmcodes der Funktion stattfindet, erfolgt eine Vorverarbeitung von globalen Informationen zur aktuellen Funktion. Dabei handelt es sich um Meta-Daten bestimmter Basisblockadressen, für die eine Bestimmung zur Laufzeit zu zeitintensiv und aufgrund der Ressourcenbeschränkung nicht möglich ist. Danach wird der Speicherbereich einer jeden Funktion in Codefragmente zerlegt, die im weiteren Verlauf auch als Codefenster bezeichnet werden. Der aktuelle Programmspeicherbereich eines Codefensters wird dann ausgelesen und in den Datenspeicher überführt.

In Abhängigkeit vom Fehlerzustand wird nun von der Steuerung entschieden welches Rekonfigurationsbackend eingesetzt wird, um den Programmtext des aktuellen Codefensters anzupassen. Dazu kann einmal ein einfaches Verfahren zum Einsatz kommen, welches lediglich innerhalb eines Instruktionswortes Anpassungen vornimmt. Andererseits können auch mächtigere Backends eingesetzt werden, wenn dies eine komplizierte

Fehlersituation erforderlich macht. Im Falle eines einfachen Backends (Rebinding/Renaming in Abb. 4.1) kann direkt eine Anpassung für jedes Instruktionswort des aktuellen Codefensters erfolgen. Ist hingegen der Einsatz eines umfangreicheren Verfahrens notwendig, erfordert dies die Ausführung weiterer Teilschritte.

Ein erster Teilschritt ist es, eine Rekonstruktion der Information zu den Basisblöcken für das aktuelle Codefenster durchzuführen. Zur Bestimmung der Basisblöcke gehört auch ein Preprocessing, welches zu Beginn der Funktionsabarbeitung aufgerufen wird. Hierbei werden die Adressen von Basisblöcken aus den Meta-Daten gelesen, die a priori bekannt sein müssen. Dabei handelt es sich um Basisblöcke, zu denen Sprungoperationen verzweigen, die sich in späteren Adressbereichen befinden. Im eigentlichen Schritt zur Basisblockrekonstruktion wird die Basisblockliste um die Blöcke erweitert, die sich im aktuellen Codefenster befinden. Dazu wird eine Analyse auf der Operationsliste durchgeführt. Wenn die Informationen zu den Basisblöcken rekonstruiert wurden, kann mit dem Schritt der eigentlichen Ablaufplanung bzw. Ressourcenallokation begonnen werden. Hier erfolgt eine Umplanung des Programmcodes innerhalb der Grenzen eines Basisblocks unter Berücksichtigung der Architekturparameter und des Defektzustands.

Zur Behandlung von verlängerten Ablaufplänen erfolgt im letzten Teilschritt ein Anpassen von Sprungadressen. Hier werden Sprungoperationen des aktuellen Codefensters direkt aktualisiert, wenn die gültigen Sprungadressen bereits bekannt sind. Andernfalls werden sich Sprungoperationen gemerkt, um diese zu einem späteren Zeitpunkt anzupassen. Die Teilschritte zur Basisblockrekonstruktion, Scheduling bzw. Allokation und Sprungzielanpassung werden miteinander verzahnt ausgeführt, da einige Aktualisierungen nicht immer sofort vorgenommen werden können und erst zum Tragen kommen, wenn sich ein verlängerter Ablaufplan ergeben hat.

## 4.2. Backends

In diesem Kapitel werden die verschiedenen Backends vorgestellt, mit deren Hilfe die Anpassung der Anwendung an eine Defektsituation erfolgt. In Abbildung 4.1 handelt es sich dabei um die grau unterlegten Boxen.

### 4.2.1. Anpassung der Bindung von Operationen

Beim Rebinding handelt es sich um ein einfaches Backend bezüglich der Komplexität der vorgenommenen Anpassungen. Dadurch ist das Verfahren wenig rechenintensiv und erfordert im Vergleich zu den anderen Backends einen geringeren Platz im Datenspeicher. Darüber hinaus entfällt der Mehraufwand zur Administration (vgl. Abbildung 4.1 Basisblockrekonstruktion und Sprungpatching), da das Rebinding keine Änderungen an der zeitlichen Ablaufplanung vornimmt. Das Rebinding beschränkt sich auf Anpassungen innerhalb einer Instruktion und nutzt vorhandene Hardware-Redundanzen, um die Bindung von Operationen zu vertauschen. Durch dieses Vorgehen können die folgenden Komponentenausfälle behandelt werden:

- Fehler in den Operatoren von Funktionseinheiten,
- Nicht mehr verwendbare FUs oder Slots und

- Fehler im Forwardingnetzwerk des Bypass.

Typischerweise sind zu einem Operationstyp mehrere Operatoren im Datenpfad auf unterschiedlichen FUs vorhanden. Durch ein Vertauschen von Operationen lassen sich ausgefallene Operatoren leicht behandeln. Für den Ausfall von FUs oder auch Slots muss im gleichem Instruktionswort eine ungenutzte Ressource des auszutauschenden Typs vorhanden sein. Zur Behandlung von Ausfällen im Bypass muss das Verfahren zusätzlich die zwei zurückliegenden Instruktionen betrachten, um entsprechende Änderungen an der Bindung vornehmen zu können.

Für die Erläuterungen des Rebindings wird das folgende Beispiel aus Abbildung 4.2 betrachtet. In Abbildung 4.2(a) ist ein Datenpfad mit 5 FUs und einem Instruktionswort $w$ dargestellt. Auf der rechten Seite ist der gleiche Datenpfad mit einem Ausfall des Operators für die Multiplikation der dritten FU angegeben. Ziel des Rebindings ist es dann die Ressourcenbindung der Operationen einer Instruktion so zu modifizieren, dass eine gültige Instruktion entsteht. Im betrachteten Beispiel ist die Instruktion $w'$ in der Form geändert, dass der defekte Multiplikationsoperator nicht verwendet wird.

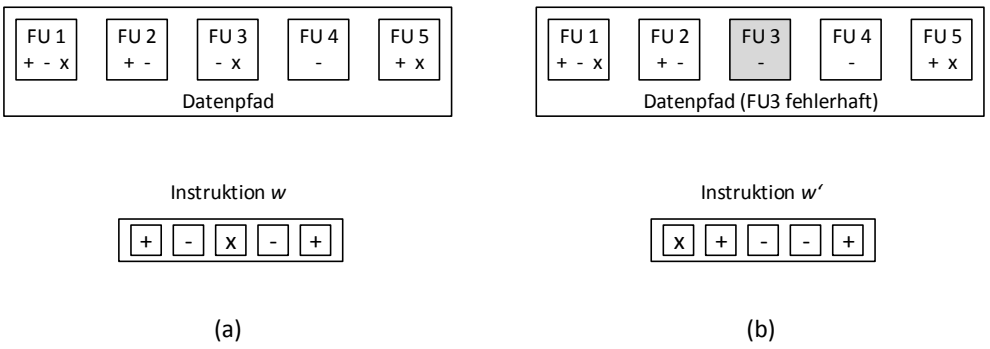


Abbildung 4.2.: (a) Datenpfad mit 5 FUs und einem Instruktionswort $w$ (b) Ausfall der Multiplikation auf FU3 mit entsprechend angepasstem Instruktionswort $w'$

Das Rebinding konstruiert eine modizifierte Instruktion $w'$ durch schrittweises Vertauschen der Operationen des ursprünglichen Instruktionswortes $w$. Die auszuführenden Schritte, bis aus $w$ die Instruktion $w'$ entsteht, werden durch eine Folge von Funktionseinheiten $\{x_0, ..., x_{n-1}\}$ beschrieben. Diese Folge wird auch als Pfad $p$ der Länge $n$ bezeichnet. Eine Verschiebung erfolgt zwischen zwei aufeinanderfolgenden Elementen $x_i$ und $x_{i+1}$ und bedeutet, dass eine Operation, die ursprünglich an die FU $x_i$ gebunden wurde, nun an $x_{i+1}$ gebunden wird. Dementsprechend wird die Operation der FU $x_{i+1}$ an die FU $x_{i+2}$ gebunden. Mit dem Durchlaufen aller Schritte einer Folge wird ein neues Instruktionswort konstruiert, welches den aktuellen Fehlerzustand berücksichtigt.

Abbildung 4.3 zeigt für das vorrangegangene Beispiel die erforderlichen Schritte zum Vertauschen der Operationen um $w'$ zu erzeugen. In diesem Beispiel sind 3 Schritte erforderlich, um das modifizierte Instruktionswort zu konstruieren. Begonnen wird bei

der Multiplikation die der dritten FU zugeordnet ist. Diese wird im ersten Schritt an die erste FU gebunden. Die Addition wird dann auf die zweite FU verschoben. Im letzten Schritt wird der Subtraktion die Ausgangsfunktionseinheit FU 3 zugeordnet. Der Pfad für dieses Beispiel ist dann beschrieben durch die Folge $\{f_3, f_1, f_2, f_3\}$.

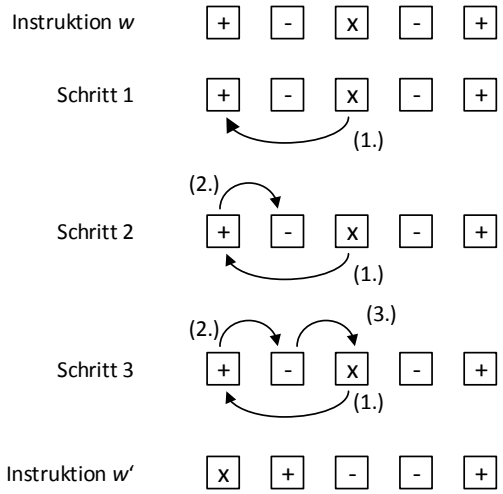


Abbildung 4.3.: Schrittweises Verschieben der Operationen zur Bindungsänderung innerhalb einer Instruktion

Zur allgemeinen Berechnung einer Folge wird in einer Breitensuche die Menge aller möglichen Pfade erzeugt. Die Erzeugung erfolgt dabei schrittweise und beginnt mit einem einelementigen Pfad, welcher lediglich den Startknoten enthält. Beim Startknoten handelt es sich um eine Operation, die auf ihrer ursprünglich zugewiesenen Hardware-Ressource nicht ausführbar ist. In jedem Iterationsschritt wird für den jeweils letzten Knoten $x_n$ eines Pfades $p$ geprüft, auf welche anderen Funktionseinheiten $f'$ dieser Knoten verschoben werden kann. Ist eine entsprechende Verschiebung möglich, wird ein neuer Pfad $p'$ in die Menge $P$ aufgenommen. Der neue Pfad $p'$ enthält $p$ und ist um $f'$ erweitert. Damit eine Verschiebung einer Operation von einer FU auf eine andere möglich ist, müssen folgende Bedingungen erfüllt sein:

- $f'$ muss einen funktionierenden Operator für die Operation von $x_n - 1$ bereitstellen,
- $f'$ muss einen funktionierenden Bypass besitzen, falls dieser von der Operation benötigt wird, und
- $f'$ muss verschieden von $x_n - 1$ sein.

Für das Beispiel vom Beginn des Abschnitts wird in Abbildung 4.4 ausführlich eine Breitensuche für die Instruktion $w := (+, -, *, -, +)$ zur Konstruktion des Pfades $\{f_3, f_1, f_2, f_3\}$ dargestellt. Ausgehend von der Startfunktionseinheit $f_3$ kann die Multiplikation im ersten Schritt auf die FUs $f_1$ und $f_5$ verschoben werden. Dadurch ergeben

sich die beiden Pfade $\{f_3, f_1\}$ und $\{f_3, f_5\}$. Im nächsten Schritt wird geprüft wie die beiden zuvor erzeugten Pfade erweitert werden können. Die Addition der ersten FU kann auf die zweite und fünfte FU verschoben werden. Der Pfad $\{f_3, f_1\}$ kann dadurch zu den Pfaden $\{f_3, f_1, f_2\}$ und $\{f_3, f_1, f_5\}$ erweitert werden. Weiterhin wird in Schritt 2 der Pfad $\{f_3, f_5\}$ betrachtet. Die Addition der fünften FU kann auf die FUs $f_2$ und $f_1$ verschoben werden. Die sich ergebenden Pfade sind demnach $\{f_3, f_5, f_1\}$ sowie $\{f_3, f_5, f_2\}$. Im nächsten Schritt wird für den Pfad $\{f_3, f_1, f_2\}$ festgestellt, dass sich die Subtraktion der zweiten FU auf die FU $f_3$ verschieben lässt. Da es sich bei dieser FU um die Ausgangsfunktionseinheit handelt, wird die Konstruktion beendet. Als Lösung konnte der Pfad $\{f_3, f_1, f_2, f_3\}$ bestimmt werden.

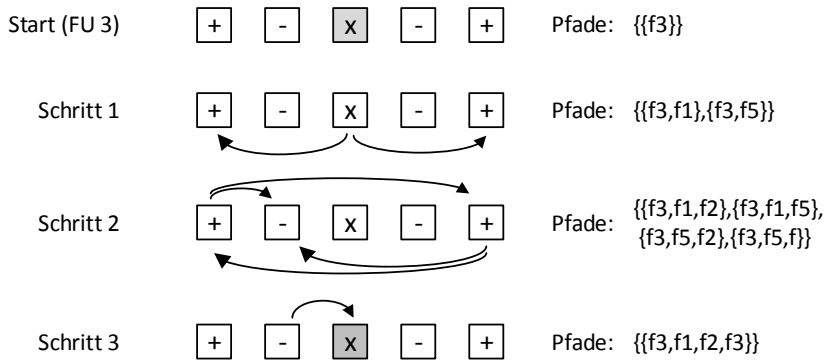


Abbildung 4.4.: Schrittweise Konstruktion der Pfade in einer Breitensuche

Der folgende Algorithmus 4.1 beschreibt formal das Vorgehen zur Modifizierung eines übergebenes Instruktionswort $w$ bezüglich des aktuellen Fehlerzustands. Algorithmus 4.1 kopiert zunächst das eingegebene Instruktionswort $w$ in eine temporäre Instruktion $w'$ (Zeile 3). Mit der äußeren Schleife (Zeile 4 bis 22) wird versucht, die Instruktion so lange anzupassen, bis alle Operationen einer gültigen Ressource zugeordnet sind. Dazu wird in jedem Durchlauf nach der ersten nicht ausführbaren Operation gesucht (Zeile 6), welche gleichzeitig der Startknoten für die aktuelle Pfadkonstruktion ist. Die innere Schleife (Zeile 7 bis 16) führt die Breitensuche durch, bis keine Pfade erweitert werden können. Die einzelnen Schritte zum Verschieben werden in der innersten Schleife (Zeile 9 bis 14) gebildet. Für den aktuellen Schritt wird jeweils geprüft, ob sich die bereits bekannten Pfade erweitern lassen (Zeile 10). Dazu werden die Zeilen der Matrix des vorangegangenen Schrittes betrachtet. Die Überprüfung, ob eine Operation verschoben werden kann, erfolgt anhand des aktuellen Defektzustands des Datenpfades (vgl. Kapitel 3.4 Fehlerspeicher). Kann ein Pfad erweitert werden (Zeile 12), erfolgt ein entsprechender Eintrag in der Matrix und in der Kategorie wird die FU eingetragen, von der die Verschiebung erfolgt. Wird im aktuellen Schritt der Startknoten erreicht oder wird eine FU gefunden, die eine Nop-Operation ausführt (Zeile 13), ist ein gültiger Pfad vorhanden und die beiden inneren Schleifen können verlassen werden.

Damit das Verfahren Fehler im Bypass behandeln kann, werden zusätzlich die beiden zuletzt verarbeiteten Instruktionen zwischengespeichert, was in der Algorithmus-

**Algorithm 4.1** Rebinding

1: **Eingabe:**Instruktionswort $w$
2: **Ausgabe:**Instruktionswort $w'$
3: $w' := w$
4: **repeat**
5: $P := \emptyset$
6: - bestimmte Start-FU $s$ bezüglich $w'$, und füge einen neuen Pfad beginnend mit $s$ in $P$ ein
7: **repeat**
8: $P' := \emptyset$
9: **for all** Pfade $p \in P$ **do**
10: - betrachte den letzten Knoten $k$ des Pfades $p$
11: - prüfe, ob die Operation $v$ von FU $k$ auf eine andere FU $f$ verschiebbar ist
12: - für jede mögliche Verschiebung erweitere $P'$ um einen neuen Pfad $p'$, der $p$ enthält und um $f$ verlängert ist
13: - wurde $s$ oder eine Nop-Operation erreicht, beende den aktuellen Schritt und setzte $p'$ als Lösung
14: **end for**
15: $P := P'$
16: **until** $P$ bleibt unverändert
17: **if** keine gültige Lösung gefunden **then**
18: - breche mit Fehlermeldung ab
19: **else**
20: - verschiebe die Operationen in $w'$, bezüglich des gefundenen Pfades $p'$
21: **end if**
22: **until** alle Operation von $w'$ sind ausführbar

beschreibung 4.1 nicht extra erwähnt ist. Zur Überprüfung, ob eine Operation auf eine FU verschiebbar ist, muss dann der Zustand des Bypasses berücksichtigt werden, wenn ein Quellregister in einer der vorherigen Instruktionen definiert wurde. Mit der Funktion $fsBypass$ kann dann überprüft werden, ob ein entsprechendes Forwarding möglich ist. Ferner wird eine Vereinbarung mit dem Compiler der Systemübersetzung getroffen, dass die ersten zwei Instruktionen eines jeden Basisblocks keine Datenabhängigkeiten aufweisen dürften, die über den Bypass behandelt werden. Da das Rebinding keine Basisblöcke kennt, würde es andernfalls überflüssige Datenabhängigkeiten (aufeinanderfolgende Basisblöcke) behandeln und bestimmte Datenabhängigkeiten gar nicht behandeln (Rückwärtssprünge). Diese Vereinbarung muss auch von den Backends beachtet werden, die eine Umplanung der Instruktionen auf Basisblockebenen durchführen.

Das Rebinding ist ein einfaches Verfahren, das schnell und mit geringem Organisationsaufwand eine Anpassung der Bindung innerhalb von Instruktionen vornehmen kann. Dabei kann es Fehler in den Komponenten des Operator-Levels, FU-Level, Slot-Level und den Bypässen behandeln. Allerdings kann es nicht für Operationen eingesetzt werden, die mehrere Takte auf einer FU ausgeführt werden. Das Verfahren ist weiter-

hin auf eine Hardware-Redundanz angewiesen, da es keine Freiheitsgrade in der zeitlichen Dimension ausnutzen kann. Das im nächsten Abschnitt beschriebene Rescheduling überwindet diese Einschränkungen auf Kosten einer degradierten Systemleistung und einem intensiveren Rechenaufwand für die Rekonfiguration.

### 4.2.2. Anpassung der Ablaufplanung für Basisblöcke

Um die Einschränkungen des Rebinding zu überwinden, nutzt das Rescheduling zur Hardware-Redundanz auch eine Verschiebung in der zeitlichen Dimension [86]. Die Idee des Verfahrens basiert auf dem Scoreboarding-Algorithmus zur Befehlsplanung in der DLX-Fließkomma-Pipeline [32]. Das Rescheduling Backend erzeugt für einen Basisblock einen neuen Ablaufplan unter Berücksichtigung des aktuellen Fehlerzustands. Ein **Ablaufplan** ist eine Folge von Instruktionen der Länge $l$ für einen Basisblock $b$. Die Berechnung eines Ablaufplans wird als **Ablaufplanung** bezeichnet. Die Berechnung erfolgt durch eine Funktion $\alpha : \{0, ..., l-1\} \rightarrow \wp(V_b)$, wobei $V_b$ die Menge der Operationen des Basisblocks $b$ ist. Die Berechnung einer neuen Ablaufplanung muss folgenden Forderungen genügen:

1. Alle Operationen von $b$ müssen einer Instruktion und somit einem Zeitschritt zugeordnet werden.
2. Jeder Operation $v$ muss eine Hardware-Ressource zur Ausführung zugewiesen werden. Die zugewiesene FU muss einen Operator bereitstellen, der den Operationstyp $type(v)$ unterstützt. Jede FU kann die leere Operation ausführen. Diese Zuordnung erfolgt implizit durch die Position einer Operation in einer Instruktion.
3. Die ursprünglichen Datenabhängigkeiten zwischen den Operationen sind durch die neue zeitliche Planung zu beachten.
4. Jeder FU wird in jedem Zeitschritt höchstens eine Operation $v$ zugewiesen. Wird keine Operation zugewiesen, wird automatisch eine leere Operation geplant.
5. Teiloperationen müssen in aufeinanderfolgenden Zeitschritten auf der gleichen FU geplant sein.
6. In einer Instruktion dürfen maximal so viele Operation geplant werden, wie FUs im Datenpfad vorhanden sind.

Das im Folgenden entwickelte Verfahren implementiert einen einfachen List-Scheduling-Algorithmus, der die Idee zur Verwaltung der Datenabhängigkeiten des Scoreboarding-Algorithmus einsetzt. Die Datenabhängigkeiten werden dabei nicht explizit durch einen Abhängigkeitsgraphen beschrieben, sondern durch tabellenartige Datenstrukturen, in denen die Informationen während der Ablaufplanung erzeugt und aktualisiert werden.

Der Algorithmus erhält als Eingabe eine Liste mit den Operationen des zu verarbeitenden Basisblocks. Die Liste wird auch als **Prioritätsliste** bezeichnet und vom Planungsverfahren als Heuristik eingesetzt zur Auswahl der nächsten Operationen als Kandidaten zur Planung. Die Reihenfolge der Operationen innerhalb der Liste entspricht für Operationen unterschiedlicher Instruktionen dem Ausführungszeitpunkt im

ursprünglichen Basisblock. Operationen der gleichen Instruktion werden anhand ihres Index innerhalb der Instruktion in die Prioritätsliste einsortiert. Die Erzeugung der Prioritätsliste wird in Abschnitt 4.3.1 im Teil zur Steuerung der lokalen Reparatur erläutert.

Das Verfahren erzeugt schrittweise eine neue Ablaufplanung für die Operationen aus der Prioritätsliste. In jedem Schritt wird ein Instruktionswort geplant. Ziel in jedem Schritt ist es, eine maximale Anzahl an Operationen auf die Funktionseinheiten des Datenpfades zu verteilen. Eine Operation $v$ kann im aktuellen Zeitschritt $t$ geplant werden, falls:

1. in der aktuellen Instruktion eine freie FU $f$ vorhanden ist, die einen Operator für $type(v)$ bereitstellt,
2. das Zielregister $dst(v)$ nicht in schreibender Benutzung durch eine ungeplante Operation $u$ mit höherer Priorität ist,
3. die Quellregister $src1(v)$ und $src2(v)$ durch bereits geplante Operationen definiert sind und
4. das Zielregister $dst(v)$ nicht Quelloperand einer ungeplanten Operation $u$ mit höherer Priorität ist.

Um diese Punkte zu gewährleisten, werden zur Verwaltung der Datenabhängigkeiten die Datenstrukturen $defs$, $uses$, $regs$ und $fus$ vereinbart. Die Datenstruktur $defs$ ist eine Liste, deren Einträge Operationsnummern sind. Für jedes Register ist ein Eintrag vorhanden. Ist in $defs(r)$ eine bestimmte Operationsnummer hinterlegt, dann bedeutet dies, dass das Register $r$ durch die entsprechende Operation mit der hinterlegten Nummer definiert wurde. Darüber werden die Flussabhängigkeiten verwaltet. Die Liste $uses$ stellt für jedes Register einen Eintrag bereit. Die Einträge entsprechen einem numerischen Zähler, der für jedes Register die noch offenen Lesezugriffe verwaltet. Wird eine Operation geplant, die ein Register $r$ definiert, so wird der Lesezähler für $r$ in $uses$ mit einem im Vorfeld ermittelten Wert initialisiert. Für jede Operation, die im weiteren Verlauf geplant wird, wird der Lesezähler der jeweiligen Quellregister um Eins dekrementiert. Solange der Lesezähler eines Registers nicht den Wert Null hat, darf das Register nicht neu definiert werden. Mit diesem Vorgehen werden die Lese-Schreib-Abhängigkeiten beachtet.

Die Datenstruktur $regs$ ist eine Liste, die für jedes Register ein Flag speichert. Wird das Flag für ein Register $r$ gesetzt, dann darf dieses Register durch keine weitere Operation geschrieben werden, bis das Flag zurückgenommen wird. Durch diese Datenstruktur werden die Ausgabeabhängigkeiten behandelt. Mit der Datenstruktur $fus$ wird die Ressourcenvergabe der Funktionseinheiten verwaltet. In der Liste $fus$ ist für jede FU ein Tupel der Form $(lock, opNr, cycle)$ vorhanden. Das Flag $lock$ zeigt an, dass eine FU in Benutzung ist durch die Operation mit der Nummer $opNr$. Die Ausführungszeit ist in $cycle$ gespeichert. In der Liste zu den Operationen wird zusätzlich gespeichert, zu welchem Zeitpunkt eine Operation geplant wurde und welche Ressource ihr zugewiesen wurde.

Mit dem Wissen über das allgemeine Verhalten und die verwendeten Datenstrukturen wird der nachfolgende Algorithmus 4.2 zum Rescheduling angegeben. Der Algorithmus

erhält als Eingabe eine Prioritätsliste, als *ops* bezeichnet. Vor der Ausführung des Algorithmus erfolgt eine Voranaylse der Operationen der Prioritätsliste. Aus Platzgründen ist der Analysevorgang nicht in Algorithmus 4.2 berücksichtigt. Mit der Voranalyse werden die Werte der Lesezähler ermittelt, mit denen ein Eintrag in *uses* initialisiert wird, wenn eine Operation geplant wird, die ein Register definiert. Dazu erfolgt eine Rückwärtsanalyse der Operationen bezüglich ihrer ursprünglichen Startzeitpunkte. Während der Analyse wird zu jedem Register die Anzahl an Lesezugriffen gezählt. Wird eine Operation $v$ gefunden, die ein Register $r$ definiert, dann wird sich zu $v$ der Wert des Lesezählers von $r$ gemerkt. Der Zähler wird für $r$ wieder auf 0 gesetzt. Alle Register in *uses*, die am Ende des Analysevorganges einen Wert größer 0 haben, sind globale Register und können in $defs$ initial auf definiert gesetzt werden.

**Algorithm 4.2** Rescheduling

```
1:  EINGABE: Operations-Liste ops
2:  AUSGABE: Ablaufplan für ops
3:
4:  t:=0
5:  repeat
6:      for f := 1 to n do
7:          if fus(f).busy = true then
8:              continue
9:          end if
10:         for all op in ops do
11:             if op ist ungeplant and regs(dst(op)) = frei then
12:                 regs(dst(op)) := belegt //WAW
13:             else
14:                 continue
15:             end if
16:             if type(op) ausführbar auf f and uses(dst(op)) = 0 and
                defs(src_1(op)) ist gültig and defs(src_2(op)) ist gültig then
17:                 plane op in t auf f
18:             end if
19:         end for
20:     end for
21:     t:=t+1
22:     for f := 1 to n do
23:         fus(f).cycle := fus(f).cycle − 1
24:         if fus(f).cycle = 0 then
25:             regs(dst(fus(f).op)) := frei // beende lock auf r
26:             defs(dst(fus(f).op)) := Nummer von fus(f).op
27:             uses(dst(fus(f).op)) := Lesezähler von fus(f).op
28:             fus(f).busy := false
29:         end if
30:     end for
31: until Alle Operation in ops sind geplant
```

Die äußere Schleife (Zeile 5 bis 31) in Algorithmus 4.2 versucht, alle Operation der Prioritätsliste zu verplanen. Dazu wird in der ersten inneren Schleife (Zeile 6 bis 20) versucht, ein Maximum an Operationen im aktuellen Zeitschritt $t$ auf den FUs zu planen. Zunächst wird geprüft, ob die untersuchte FU überhaupt frei ist (Zeile 7). Anschließend wird in der Prioritätsliste nach dem nächsten planbaren Kandidaten mit höchster Priorität gesucht (Zeile 10 bis 19). Ist eine Operation *op* ungeplant und keine ungeplante Operation mit höherer Priorität belegt das Zielregister $dst(op)$, dann wird vermerkt, dass *op* selbst die nächste Operation mit höchster Priorität bezüglich des Registers $dst(op)$ ist (Zeile 11 und 12). Anschließend wird versucht, *op* auf der aktuell untersuchten FU $f$ zu planen (Zeile 16). Dazu muss $f$ einen Operator passenden Typs bereitstellen. Weiterhin muss geprüft werden, dass keine offenen Lesezugriffe auf des Register $dst(op)$ bestehen. Als letztes ist abzufragen, ob die Quelloperanden von *op* in einem Register bereitstehen. Nachdem die maximale Anzahl an Operationen für den aktuellen Schritt geplant wurden, werden die Datenstrukturen aktualisiert (Zeile 22 bis 30). Ist im nächsten Zeitschritt die Berechnung einer Operation $v$ beendet (Zeile 24), wird das Zielregister $dst(v)$ freigegeben (Zeile 25) und vermerkt, dass $v$ nun $dst(v)$ definiert hat. Es wird weiterhin der Lesezähler für $dst(v)$ gesetzt und die FU wird als verfügbar markiert.

Bisher wurden nur Datenabhängigkeiten bezüglich Registern besprochen. Zusätzlich ist es auch notwendig, Datenabhängigkeiten für Speicherzugriffe und Zugriffe auf die I/O-Ports in der richtigen Reihenfolge durchzuführen. Der Einfachheit halber werden diese Operationen in genau der gleichen Reihenfolge geplant wie im ursprünglichen Ablaufplan, da eine Analyse der Speicher- und Portzugriffe (Zeigerdereferenzierung, Steuerung externer Bausteine über Port-Register) zu aufwendig ist.

Einen nächsten Sonderfall stellen Datenabhängigkeiten zwischen Operationen der selben Instruktion dar. Operationen der selben Instruktion werden nacheinander in die Operationsliste eingelesen. Die Abarbeitung der Liste erfolgt sequentiell, weshalb der Algorithmus die parallele Ausführung nicht erkennt. Statt dessen wird eine Datenabhängigkeit erkannt, die den Algorithmus zu einer falschen Planungsreihenfolge zwingt. Das Problem tritt auf, wenn eine Operation $v_i$ ein Register schreibt und eine Operation $v_j$ das Register lesend verwendet. Ist der Index $i$ kleiner als $j$ wird der Algorithmus bei der Abarbeitung der Operationsliste eine Lese-Schreib-Abhängigkeit feststellen und $v_j$ später als $v_i$ planen. Diese Konstellation muss durch den Compiler während der Systemübersetzungszeit ausgeschlossen werden. Ferner darf auch das Rescheduling keine Operationen in der Form planen.

Das Rescheduling ist ein komplexes software-basiertes Reparaturverfahren, welches zur Rekonfiguration die Freiheitsgrade der redundanten Hardware und eine Verschiebung von Operationen in spätere Instruktionen kombiniert einsetzt. Der Vorteil des Verfahrens liegt darin, dass es eine Anwendung auch dann an einen Fehlerzustand anpassen kann, wenn nicht ausreichend Hardware-Redundanz verfügbar ist. Die akzeptierte degradierte Systemleistung entsteht durch das Verschieben von Operationen in spätere Startzeitpunkte. Mit diesem Verfahren können viele Komponentenausfälle behandelt werden (vgl. Tabelle 3.2), außer defekte in Registern. Für den Fall eines ausgefallenen Registers werden in den nächsten beiden Abschnitten entsprechende Backends vorgestellt.

### 4.2.3. Umbenennung von Registern zur Behandlung defekter Registern

Das Renaming ist ein Backend, um Registerausfälle zu behandeln. Die eingesetzte Strategie basiert, wie das Rebinding, auf einer reinen Hardware-Redundanz. Die Idee dabei ist es, ungenutzte Reserve-Register vorzuhalten, um diese bei Bedarf als Ersatz für ein ausgefallenes Register einzusetzen [85]. Zur Reparatur ändert das Renaming im Programmcode die Registerbezeichnung eines fehlerhaften Registers in den Bezeichner eines Reserve-Registers. Der Vorteil des Renaming ist seine Einfachheit, weil alle Register gleich behandelt werden. Dadurch ist keine aufwendige Unterscheidung zwischen globalen, lokalen und Spezialregistern notwendig und der administrative Aufwand bleibt gering.

Voraussetzung für den Einsatz des Verfahrens ist, dass die Reserve-Register durch die Anwendung nicht verwendet werden. Dies abzusichern, ist Aufgabe des Compilers zur Übersetzungszeit des Systems. Hier wird festgelegt, wie viele von den durch die Architektur bereitgestellten allgemeinen Registern als Reserve verwendet werden. Der Compiler nutzt dann zur Registerallokation diese festgelegte Anzahl an frei verwendbaren Registern. Die Zahl der Reserve-Register ergibt sich aus der Anzahl an $k$-Registerausfällen, die durch das System tolerierbar sein sollen.

Unter Berücksichtigung, dass $k$ Register zur Übersetzungszeit reserviert wurden, wird der folgende Algorithmus 4.3 für das Renaming angegeben. Als Eingabe erhält der Algorithmus eine Liste mit Operationen. Dabei handelt es sich nicht um eine Prioritätsliste wie für das Rescheduling, da die Reihenfolge der Operationen für das Renaming keine Bedeutung hat. Als temporäre Datenstruktur wird eine Liste $regs$ mit $n$ numerischen Einträgen verwendet. Die Liste $regs$ repräsentiert eine Abbildung zwischen der originalen Registernummer und dem Register, welches neu verwendet werden soll. Soll ein altes Register $r$ weiterhin verwendet werden, so enthält $regs[r]$ die Nummer von $r$. Die Registernummern der Quelloperanden und des Zielregisters werden als Index in $regs$ verwendet, um die neue Registernummer nachzuschlagen.

Zunächst erzeugt das Renaming in einer Schleife eine Abbildung zwischen den originalen Registern und den neu zu verwendenden (Zeile 4 bis 11). Zum Überprüfen, ob ein Register fehlerhaft ist (Zeile 5), wird der Registerzustand aus dem Defektspeicher des Kerns abgefragt (vgl. Kap 3.4). Ist ein Register $r$ defekt, wird in $regs$ der entsprechende Eintrag mit der Nummer des nächsten Reserve-Registers $spare$ initialisiert. Anschließend wird $spare$ auf das nächste Reserve-Register gesetzt. Ist nach dem Erzeugen der Abbildung der Wert von $spare - 1$ größer als die Anzahl an Reserve-Registern, dann liegen zu viele fehlerhafte Register vor (Zeile 12). In der nächsten Schleife (Zeile 15 bis 19) werden alle verwendeten Register einer Operation auf die jeweilige Registernummer aus $regs$ gesetzt.

Das Renaming hat den Vorteil, keine Laufzeiteinbußen zu verursachen, da es ausschließlich redundante Hardware-Komponenten einsetzt. Aufgrund der einfachen Ersetzungsstrategie kann der organisatorische Aufwand gering gehalten werden. Nachteilig sind die Kosten für die zusätzliche Hardware und den dadurch höheren Stromverbrauch. Um den Aufwand zwischen Reserve-Registern und tolerierbaren Registerfehlern besser ausbalancieren zu können, wird im nächsten Abschnitt ein Verfahren präsentiert, welches auch in der zeitlichen Dimension arbeitet. Dadurch können auf Kosten einer

**Algorithm 4.3** Renaming

```
1: Eingabe: Liste mit Operationen ops
2: Ausgabe: ops'
3: spare := Nummer des ersten Reserveregisters
4: for r := 0 to n − 1 do
5:     if Register r ist defekt then
6:         regs(r) := spare
7:         spare := spare + 1
8:     else
9:         regs(r) := r
10:    end if
11: end for
12: if (spare > k + 1) then
13:    Abbruch - Anzahl an Spare-Registern unzureichend
14: end if
15: for all op in ops do
16:    dst(op) := regs(dst(op))
17:    src1(op) := regs(src1(op))
18:    src2(op) := regs(src2(op))
19: end for
```

degradierten Systemleistung Registerausfälle behandelt werden, falls die Anzahl der Reserve-Register nicht ausreichend ist.

### 4.2.4. Anpassung der Registervergabe

Im zurückliegenden Abschnitt wurden defekte Register dadurch kompensiert, dass vom Compiler während der Systemübersetzungszeit ein Teil des Registersatzes zurückgehalten wurde. Die entstandenen Reserveregister konnten dann zur Reparaturzeit durch ein entsprechendes Renaming-Verfahren defekte Register ersetzen. Der Nachteil einer solchen Strategie ist, dass die volle Leistungsfähigkeit eines Prozessors nicht verfügbar ist. In diesem Kapitel soll ein komplexeres Verfahren vorgestellt werden, das eine Anpassung der Registerallokation mit wenigen oder gar keinen Reserveregistern vornehmen kann. Somit steht der Anwendung im fehlerfreien Fall der gesamte Registersatz zur Verfügung, um die gesamte Leistungsfähigkeit eines Prozessors nutzen zu können.

Zur Behandlung eines ausgefallenen Registers erfolgt die Integration einer Registerallokation in ein Verfahren zur Ablaufplanung, wie es in Abschnitt 4.2.2 beschrieben wurde. Eine Neuvergabe der Registerallokation birgt zwei hauptsächliche Schwierigkeiten:

- aufgrund der limitierten Ressourcen besteht nur eine eingeschränkte Analysemöglichkeit bezüglich des Programmcodes der zu rekonfigurierenden Anwendung und
- die globale Registervergabe muss von dem Verfahren beachtet werden.

Der erste Punkt bedeutet, dass die eigentliche Allokationsstrategie keine komplexen Datenstrukturen und Algorithmen einsetzen kann. Darüber hinaus kann, im Hinblick auf den zweiten Punkt, keine Analyse bezüglich der globalen Registervergabe des Programmcodes erfolgen. Die Strategie zur Registervergabe muss demnach berücksichtigen, dass Werte in Registern eine basisblockübergreifende Gültigkeit besitzen können.

Um die Problematik mit der globalen Registervergabe zu umgehen, wird eine Strategie eingesetzt, die auf der Kenntnis über die eingesetzten globalen Register basiert. Die Idee dabei ist es, die zur Verfügung stehenden Register $(r_0, ..., r_{n-1})$ in zwei Teile zu partitionieren. Die Register $(r_0, ..., r_{k-1})$ des ersten Teils speichert ausschließlich global genutzte Werte. Der zweite Teil mit den Registern $(r_k, ..., r_{n-1})$ darf nur lokale Werte speichern. Die Unterteilung erfolgt statisch und wird zur Übersetzungszeit des Systems getroffen. Während der Reparatur kann nun das Allokationsverfahren anhand der Nummer eines Registers entscheiden, ob es sich um ein lokales oder ein globales Register handelt. Der Algorithmus zur Registerallokation arbeitet als zweistufiges Verfahren. In Abhängigkeit davon, ob ein lokales oder globales Register fehlerhaft ist, werden unterschiedliche Maßnahmen ergriffen:

- Ist ein lokales Register defekt, wird für jeden Basisblock eine neue lokale Registervergabe berechnet unter Ausschluss der fehlerhaften Register.

- Ist ein globales Register defekt, wird es durch das erste fehlerfreie lokale Register ersetzt. Dazu wird mit dem Algorithmus 4.3 eine Anpassung der Registernummern durchgeführt. Da im lokalen Registersatz nun ein Register fehlt, wird dies so behandelt, als wäre ein lokales Register defekt und es wird eine neue Registerallokation für die lokalen Register bestimmt.

#### Anpassung der lokalen Registervergabe

Die Anpassung der lokalen Registervergabe erfolgt, wenn ein lokal genutztes Register defekt ist oder ein lokales Register als Ersatzregister für ein defektes globales eingesetzt wird. Als Grundlage für den Algorithmus der Registerallokation dient das Planungsverfahren aus Abschnitt 4.2.2. Das Planungsverfahren wird in der Form erweitert, dass es einer Operation, die geplant wird, nun auch ein Zielregister zuordnen kann. Das erweiterte Verfahren arbeitet weiterhin auf der Granularität von Basisblöcken.

Bevor die eigentliche Registerallokation für die Operationen eines Basisblocks erfolgen kann, sind zwei Schritte im Vorfeld durchzuführen:

- Die Anzahl an Verwendungen für ein Datum muss in einer Voranalyse ermittelt und

- die ursprüngliche Zuordnung zwischen den Operationen und den vergebenen Quell- und Zielregistern muss entkoppelt werden.

Eine wichtige Information für eine Registervergabe ist, ab welchem Zeitpunkt der Wert in einem Register keine nächste Verwendung hat. Wenn ein Wert keine nächste Verwendung hat, kann das assoziierte Register freigegeben werden. Um den Registerdruck auf eine Allokationsstrategie zu verringern, ist eine frühzeitige Freigabe von Vorteil. Das

Entkoppeln von der alten Registervergabe ist erforderlich, um während der Registerallokation mehrdeutige Bezüge auf das gleiche Register zu vermeiden.

Die beiden Punkte werden in einer Voranalyse bezüglich der Operationen eines Basisblocks behandelt. Dazu wird die Anzahl der Zugriffe auf jeden Wert ermittelt. Darüber hinaus wird aus der alten Registervergabe für jeden definierten Wert eine temporäre Variable erzeugt. Für das Verfahren werden zwei Datenstrukturen eingesetzt, einmal der Variablendeskriptor *varDescr* und eine Look-Up-Tabelle *lookup*. Die Lesezugriffe für einen Wert werden anhand des Variablendeskriptors gezählt. Weiterhin kann in *varDescr* zu einer temporären Variable das zugewiesene Register gemerkt werden. Dieser Punkt wird allerdings erst durch die eigentliche Registerallokation genutzt. Die Look-Up-Tabelle wird während der Voranalyse als Indizierung in den Variablendeskriptor verwendet. Die vorhandenen Einträge der Tabelle entsprechen der Anzahl an lokalen Registern. Erfolgt die Definition eines Registers $r$ im Programmcode, dann wird eine neue temporäre Variable *temp* erzeugt. Im Eintrag $lookup[r]$ wird die Nummer von *temp* vermerkt, um spätere Verwendungen von $r$ durch *temp* zu ersetzen.

Der nachfolgende Algorithmus 4.4, verarbeitet alle Operationen in der Reihenfolge, in der sie ursprünglich geplant wurden. Wird durch eine Operation $v$ ein Register $r$ definiert, dann wird hierfür ein temporärer Wert $k$ vergeben. Die Nummer von $k$ wird in der Look-Up-Tabelle an der Position $r$ eingetragen. Im Variablendeskriptor werden nun im Eintrag $k$ alle nachfolgenden Lesezugriffe auf $r$ gezählt, bis $r$ erneut definiert wird. Zusätzlich werden in den Operationen die ursprünglich vergebenen Registernummern durch den temporären Wert $k$ ersetzt.

**Algorithm 4.4** Konstruktion Variablendescriptor

```
1:  Eingabe: Operationsliste ops
2:  Ausgabe: Variablendescriptor vars
3:
4:  k := maxGlobRegs
5:  for all op aus ops do
6:      if op besitzt ein Zielregister and dst(op) ≥ maxGlobReg then
7:          lookup[dst(op)] := k
8:          vars[k].uses := 0
9:          - ersetze Zielregister von op durch k
10:         k := k + 1
11:     end if
12:     if op besitzt linkes Quellregister and src1(op) ≥ maxGlobReg then
13:         vars[lookup[src1(op)]].uses := vars[lookup[src1(op)]].uses + 1
14:         - ersetze linkes Quellregister von op mit dem Wert lookup[src1(op)]
15:     end if
16:     if op besitzt rechtes Quellregister and src2(op) ≥ maxGlobReg then
17:         vars[lookup[src2(op)]].uses := vars[lookup[src2(op)]].uses + 1
18:         - ersetze rechtes Quellregister von op mit dem Wert lookup[src2(op)]
19:     end if
20: end for
```

Der Algorithmus erhält als Eingabe eine Liste *ops* mit den Operationen des bearbeiteten Basisblocks. Die Sortierung der Operation entspricht für Operationen aus unterschiedlichen Instruktionen der ursprünglichen Planungsreihenfolge. Operationen der gleichen Instruktionen werden anhand der Position aus ihrer Instruktion sortiert. In der Hauptschleife (Zeile 5 bis 20) werden die Operationen aus der Liste *ops* nacheinander abgearbeitet. Zunächst wird für eine Operation $v$ die Untersuchung für das Zielregister durchgeführt (Zeile 6 bis 11). Wenn es sich um kein globales Register handelt, wird die Nummer des dadurch definierten temporären Wertes $k$ in der Look-Up-Tabelle eingetragen und zusätzlich auch in $v$ vermerkt. Weiterhin wird der Lesezähler von $k$ mit 0 initialisiert. In den Zeilen 12 bis 19 werden die Quellregister, insoweit diese vorhanden und lokale Register sind, behandelt. In der Look-Up-Tabelle wird nachgeschlagen, welcher temporäre Wert mit der entsprechenden Registernummer verknüpft ist. Anhand der Nummer des temporären Wertes kann der Lesezähler im Variablendeskriptor erhöht werden. Der Variablendeskriptor ist neben der Operationsliste die wesentliche Eingabe für den nachfolgenden Algorithmus zur Registerallokation.

Das Allokationsverfahren erweitert das List-Scheduling-Verfahren, das in Abschnitt 4.2.2 vorgestellt wurde. Es unterscheidet sich vom Rescheduling in der eingesetzten Heuristik und in der Verwaltung der Datenabhängigkeiten, weil die Gegen- und Ausgabeabhängigkeiten mit der Vergabe der temporären Variablen aufgehoben wurden. Der Algorithmus versucht in jedem Takt ein Maximum an Operationen auf die freien Funktionseinheiten zu verteilen. Die Auswahl der Kandidaten erfolgt anhand der eingegebenen Prioritätsliste *ops*. Der Algorithmus verwendet einen Registerdeskriptor *regs* zur Verwaltung der Registernutzung. Für jedes lokale Register ist ein Eintrag in *regs* vorhanden zur Markierung ob ein Register belegt ist. Ein Register wird freigegeben, wenn für einen temporären Wert alle Lesezugriffe erfolgt sind. Die Zuordnung zwischen einem temporären Wert und einem Register wird im Variablendeskriptor gemerkt. Dadurch kann direkt anhand der Nummer der temporären Variable das zugewiesene Register ermittelt werden. Der folgende Algorithmus 4.5 führt eine Registerallokation mit den zuvor beschriebenen Datenstrukturen durch.

Die äußere Schleife versucht, alle Operationen aus der Operationsliste *ops* zu planen (Zeile 4 bis 32). Mit der ersten inneren For-Schleife (Zeile 5 bis 23) soll für jede Funktionseinheit eine Operation zur Planung gefunden werden. Wenn eine freie FU vorhanden ist, wird ein Kandidat zur Planung bestimmt (Zeile 7). Es wird nach einer Operation $v$ mit höchster Priorität gesucht. Dabei werden Operationen bevorzugt, die durch ihre Planung mehr Register frei geben als sie definieren. Zum Beispiel wird eine Operation, die eine Konstante lädt, ein Register belegen, wohingegen eine Additionsoperation im besten Fall ein freies Register erzeugt. Für einen gefundenen Kandidaten $v$ wird weiterhin geprüft, ob die FU $f$ einen Operator für $type(v)$ bereitstellt und ob die Quelloperanden in einem Register verfügbar sind. Anschließend werden in den Zeilen 9 bis 18 die möglichen Quelloperanden von $v$ behandelt. Handelt es sich dabei um ein lokales Register, wird anhand der temporären Variablen $k$ das zugeordnete Register $r$ ermittelt. Für $k$ wird der Lesezähler dekrementiert und falls $k$ keine nächste Verwendung hat wird $r$ freigegeben. Weiterhin wird in $v$ die Nummer von $k$ mit der Registernummer von $r$ ersetzt. Danach wird in Zeile 19 ein freies Register $r'$ als Zielregister für $v$ gesucht. Bei der Suche wird der lokale Registerbereich beachtet und zusätzlich der Defektzustand

**Algorithm 4.5** Registerallokation

```
1: EINGABE: Operationsliste ops, Variablendeskriptor vars
2: AUSGABE: Operationsliste ops'
3: t:=0
4: repeat
5:     for f := 1 to n do
6:         if fus(f) ist verfügbar then
7:             - suche Operation op mit höchster Priorität
8:             - prüfe, ob die Quelloperanden von op definiert sind
9:             if src1(op) ist gültig and src1(op) ≥ maxGlobReg then
10:                - vars[src1(op)].uses − −
11:                - ersetze src1 in op durch vars[src1(op)].reg
12:                - ist vars[src1(op)].uses 0, markiere Register als frei
13:            end if
14:            if src2(op) ist gültig and src2(op) ≥ maxGlobReg then
15:                - vars[src2(op)].uses − −
16:                - ersetze src2 in op durch vars[src2(op)].reg
17:                - ist vars[src2(op)].uses 0, markiere Register als frei
18:            end if
19:            - wähle nächstes freies Register r und vermerke, dass r für op reser-
               viert ist und setze r auf belegt
20:            - plane op in t auf f
21:        end if
22:    end for
23:    t:=t+1
24:    for f := 1 to n do
25:        fus(f).cycle := fus(f).cycle − 1
26:        if fus(f).cycle = 0 then
27:            - setze für vars[dst(op)].reg das reservierte Register von dst(op)
28:            - markiere f als verfügbar
29:        end if
30:    end for
31: until Alle Operation aus ops sind geplant
```

der Register. Anschließend wird vermerkt, dass $r'$ in Benutzung ist und für die temporäre Variable $k$ verwendet wird. Im Variablendeskriptor kann dies noch nicht vermerkt werden, da sonst Operationen im gleichen Instruktionswort geplant werden könnten, die den temporären Wert von $v$ lesend verwenden.

Nachdem ein Maximum an Operationen im aktuellen Zeitschritt $t$ geplant wurde, werden die Datenstrukturen aktualisiert (Zeile 25 bis 31). Ist die Berechnung einer Operation beendet, kann die entsprechende FU als verfügbar markiert werden. Darüber hinaus wird nun im Variablendeskriptor vermerkt, dass einer temporären Variablen das entsprechende Register zugeordnet ist. Der Lesezähler besitzt bereits den passenden Wert, weil der Variablendeskriptor aus der Voranalyse unverändert bleibt.

Was weiterhin beachtet wird, aber nicht explizit im Algorithmus aufgeführt ist, sind die Datenabhängigkeiten zwischen Operationen mit Speicherzugriffen und Portzugriffen. Die Behandlung dieser Abhängigkeiten erfolgt analog zum Scheduling-Algorithmus 4.2 aus Abschnitt 4.2.2.

Damit der Algorithmus in der Lage ist, eines oder mehrere ausgefallene Register zu kompensieren, wird die Anzahl an parallel geplanten Operationen je Instruktion verringert. Dies erfolgt, indem der Wert von $n$ für die Schleife aus Zeile 5, nicht auf die maximale Anzahl an verfügbaren FUs im Datenpfad gesetzt wird. Die Reduzierung der gleichzeitig geplanten Operationen hat zum Ziel, weniger lebendige Register je Instruktion zu erzeugen. Dadurch können für den neu erzeugten Ablaufplan weniger Register erforderlich sein, wodurch sich eine Kompensation von defekten Registern ergibt.

#### Behandlung eines fehlerhaften globalen Registers

Für den Fall, dass ein globales Register $g$ defekt ist, erfolgt eine 2-stufige Behandlung. Das Register $g$ wird durch ein Register $r$ aus dem Bereich der lokalen Register ersetzt. Als $r$ wird dazu das lokale Register mit der kleinsten Nummer gewählt. Für eine vorliegende Registeraufteilung $(r_0, ..., r_{k-1}, r_k, ..., r_{n-1})$ wird der Wert von $maxGlobRegs$ um 1 erhöht und die Aufteilung der Register ändert sich zu $(r_0, ..., r_k, r_{k+1}, ..., r_{n-1})$. Das Ersetzen von $g$ durch $r$ erfolgt mit Hilfe des Algorithmus 4.3 für das Renaming aus Abschnitt 4.2.3. Da $r$ nun im lokalen Registersatz fehlt, wird dieser Fall wie der Defekt eines lokalen Registers gehandhabt und es wird eine Registervergabe mit Algorithmus 4.5 durchgeführt.

Der nachfolgende Algorithmus 4.6 beschreibt den Ablauf zur allgemeinen Registerallokation. Zunächst erfolgt eine Registervergabe mit der Angabe, dass $r$ defekt ist, damit $r$ durch den Algorithmus der lokalen Registerallokation nicht verwendet wird. Erst anschließend kann $r$ den globalen Registern hinzugefügt werden. Nach der lokalen Registerallokation wird $r$ als fehlerfrei markiert und dem Satz an globalen Registern hinzugefügt werden. Nun kann die Umbenennung von $g$ in $r$ durchgeführt werden.

**Algorithm 4.6** Globale Registerallokation

```
1:  Eingabe: Basisblock b
2:  Ausgabe: Basisblock b'
3:
4:  if Register r ist defekt and r ≥ maxGlobRegs then
5:      - führe lokale Registerallokation für b durch
6:  else
7:      - Reserveregister spare := maxGlobRegs
8:      - markiere Register spare als defekt
9:      - führe lokale Registerallokation für b durch
10:     - markiere Register spare als fehlerfrei
11:     maxGlobRegs := maxGlobRegs + 1
12:     - führe Renaming für b durch und ersetze r mit spare
13: end if
```

## 4.3. Organisation der lokalen Reparatur

Der folgende Abschnitt beschreibt detailliert die Organisation der lokalen Rekonfiguration. Die behandelten Schwerpunkte sind in Abbildung 4.5 hervorgehoben, wohingegen die bereits besprochenen Teile ausgeblendet sind. Der erste Schwerpunkt ist die Zerlegung der Anwendung in Funktionen und Codefragmente, um diese schrittweise umzuplanen. Dazu gehört auch die Administration von Reservebereichen im Programmspeicher. Diese sind erforderlich, da sich durch die Ablaufplanung verlängerte Basisblöcke ergeben. Der nächste Schwerpunkt ist die Rekonstruktion der Basisblöcke. Die Analyse bezüglich der Basisblöcke erfolgt für jedes Codefragment getrennt. Der letzte Schwerpunkt dieses Abschnitts ist das Sprung-Patching zur Behandlung von sich geänderten Sprungzielen durch eine Ablaufplanverlängerung.

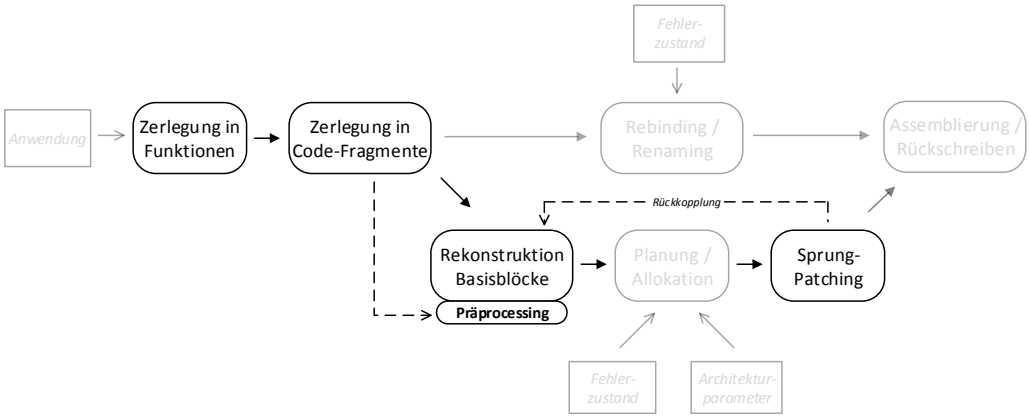


Abbildung 4.5.: Detailansicht zur Organisation der lokalen Rekonfiguration

### 4.3.1. Administration des Programmspeichers

Aufgrund der Ressourcenbeschränkungen in einem eingebetteten System ist ein angepasster Umgang mit dem Code der Anwendung erforderlich. Während der Rekonfiguration umfasst die Administration des Programmspeichers die folgenden 3 wesentlichen Punkte:

- das Verwalten des Reservespeichers als Reservebereich,
- das Zerlegen der Anwendung in Funktionen und Codefragmente und
- der Umgang mit den Metadaten zur Anwendung.

Der Reservespeicher wird verwendet, wenn ein Planungsverfahren Operationen einem späteren Zeitpunkt zuweist. Ein dadurch verlängerter Ablaufplan belegt mehr Platz im Programmspeicher. Durch den Reservespeicher werden Überschreibungen vermieden. Wegen der eingeschränkten Größe des Datenspeichers ist die Verarbeitung des Binärcodes als Ganzes nicht durchführbar, weshalb eine schrittweise Zerlegung erfolgt. Die

Zerlegung erfolgt dabei zunächst auf der Granularität von Funktionen, wie es auch typisch für das Vorgehen eines Compilers ist. Der Code der einzelnen Funktionen wird dann in Fragmente zerlegt, für die dann nacheinander die Ablaufplanung erfolgt. Bei den Metadaten handelt es sich um Informationen zur Anwendung, die zur Übersetzungszeit des Systems berechnet werden und im Programmspeicher hinterlegt sind. Dadurch können Analysevorgänge zur Reparaturzeit vereinfacht werden, die andernfalls zu aufwendig sind.

Die Metadaten umfassen Informationen zu:

1. den Startadressen aller Funktionen einer Anwendung und
2. die Zieladressen von Sprungoperationen, die rückwärts verzweigen.

Anhand der Startadressen der Funktionen kann die Reparatur die Anwendung in ihre Funktionen zerlegen. Eine **Sprungoperation** kann bedingt oder unbedingt verzweigen. Die Sprungzieladresse eines **Rückwärtssprungs** $v$ ist kleiner als die Adresse der Instruktion, zu der $v$ gehört. Analog dazu ist die Zieladresse bei einem **Vorwärtssprung** größer. Zur Unterstützung der Basisblockrekonstruktion sind die Zieladressen von Rückwärtssprüngen in den Metadaten hinterlegt. Die genaue Funktionsweise wird in den dazugehörigen Abschnitten erläutert.

#### Verwaltung des Reservespeichers

Der erforderliche Speicherplatz für die verlängerten Basisblöcke nach einer Reparatur wird zur Systementwicklungszeit festgelegt und der Programmspeicher wird um die entsprechende Anzahl an Reserveeinträgen größer dimensioniert. Der Bereich mit den Reserveadressen wird an das Ende des Programmspeichers, also hinter die eigentliche Anwendung gelegt. Um die freien Speicherbereiche für den jeweils in der Rekonfiguration befindlichen Basisblock nutzen zu können, wird die folgende Strategie angewandt. Vor der eigentlichen Rekonfiguration wird der gesamte Programmcode der Anwendung an das Ende des Programmspeichers verschoben. Der Bereich mit den Reserveeinträgen verlagert sich dadurch vor die Anwendung. Während der nun folgenden schrittweisen Umplanung der Anwendung werden fertig geplante Basisblöcke wieder in den vorderen Speicherbereich zurückgeschrieben. Durch dieses Vorgehen wird erreicht, dass

- zur Umplanung eines jeden Basisblocks der maximale Reservespeicher zur Verfügung steht und
- ein Überschreiben von ungeplanten Code durch rekonfigurierte Basisblöcke vermieden wird.

Ein weiterer Vorteil ist, dass sich diese Strategie zur Verwaltung der Reservebereiche einfach implementieren lässt im Vergleich zu Verfahren, die zusätzliche Instruktionen mit Nop-Operationen in die Basisblöcke des Programmcodes integrieren. Die Integration zusätzlicher Instruktionen hat den Nachteil, dass der Code entweder zur Laufzeit generell mit ausgeführt wird oder die Ausführung in geeigneter Form vermieden werden muss. Diese Probleme entfallen für die eingangs vorgestellte Strategie und es ergibt sich der wesentliche Vorteil, dass

- im fehlerfreien Fall keine Leistungseinbußen durch die Reserveadressen entstehen.

Abbildung 4.6 zeigt die Verwendung der Reserveeinträge durch das Reparaturverfahren. Es sind drei Programmspeicherinhalte während der unterschiedlichen Phasen einer Anwendungsrekonfiguration dargestellt. Die erste Abbildung stellt das Layout des Programmspeichers vor der Rekonfiguration da. Das Layout nach dem Verschieben der Anwendung ist in der Mitte zu erkennen. Dabei ist zu sehen, dass der Code für den Systemtest und die Reparatur nicht verschoben wurde. Auf der rechten Seite ist der Programmspeicher dargestellt, nachdem die ersten zwei Funktionen umgeplant wurden. Diese wurden in den vorderen Bereich zurückgeschrieben und konnten dadurch im Fall einer Verlängerung des Ablaufplans den Reservebereich ausnutzen.

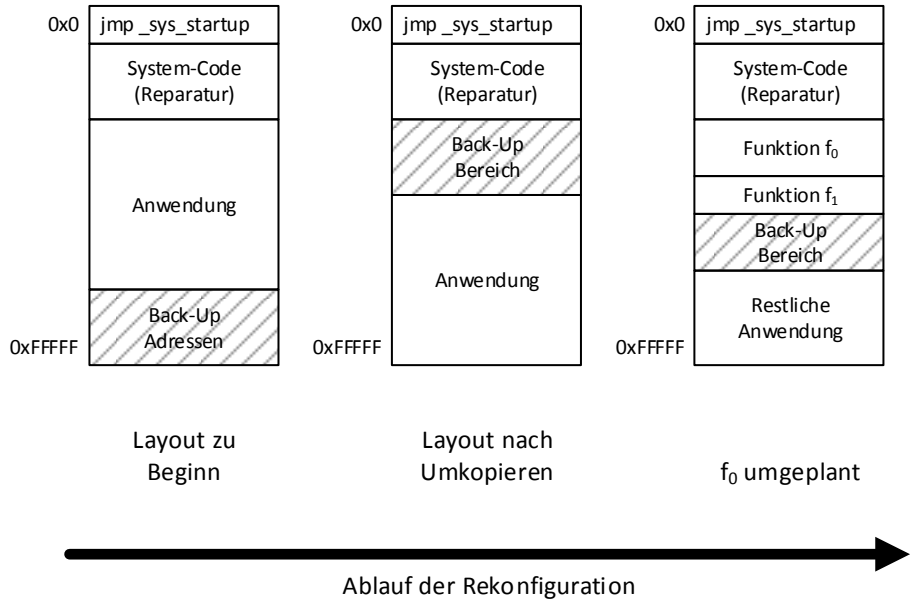


Abbildung 4.6.: Ablauf der schrittweisen Umplanung der einzelnen Funktionen einer Anwendung

### Zerlegung des Programmspeichers

Die Zerlegung einer Anwendung zur Rekonfiguration erfolgt in der ersten Stufe auf der Granularität von Funktionen. Für die einzelnen Funktionen erfolgt eine Anpassung in der zweiten Stufe, basierend auf Codefragmenten. Für die vorliegenden Arbeit ergibt sich dabei der Vorteil, dass gesammelte temporäre Informationen nur für eine Funktion vorgehalten werden müssen und nicht für die gesamte Anwendung. Dadurch verringert sich die Größe der notwendigen Datenstrukturen entsprechend. Die zweite Granularitätsstufe wurde gewählt, weil davon auszugehen ist, dass für eine beliebige Funktion nicht der gesamte Programmcode zur Rekonfiguration im Datenspeicher vorgehalten

werden kann. Statt dessen ist dies nur für Teile des Codes möglich, die als Fragmente bezeichnet werden.

Eine Analyse der Anwendung bezüglich der Start- und Endadressen zu den einzelnen Funktionen wird als zu aufwendig angesehen. Deshalb werden die Adressen dem Reparaturverfahren als Metadaten zur Verfügung gestellt. Die Metadaten zu den Funktionen werden in die Liste $funcs$ eingelesen. In $funcs$ werden zu jeder Funktion zwei Adressen eingetragen. Zum einen ist das die ursprüngliche Adresse aus den Metadaten und zum anderen die neue Startadresse, die sich später anhand möglicher Programmspeicherverschiebungen ergibt. Für das initiale Erzeugen der Liste ist zunächst nur der ursprüngliche Adresseintrag relevant. Der genaue Aufbau der Metadaten wird im Anschluss an diesen Abschnitt besprochen.

Die Funktionen aus der Funktionsliste werden schrittweise abgearbeitet und einer Rekonfiguration unterzogen. Als Erstes werden zu jeder Funktion weitere Metadaten eingelesen, die sich ausschließlich auf die jeweilige Funktion beziehen. Diese Metadaten enthalten die Zieladressen von Rücksprüngen, die sich auf Adressen beziehen, die innerhalb der aktuell bearbeiteten Funktion liegen. Mit diesen Metadaten wird die Basisblockliste *bbs* initialisiert. Die Liste *bbs* enthält Tupel der Form $(old, new)$. Ein Eintrag $bbs(i)$ speichert für einen Basisblock $b$ in $old$ die alte Anfangsadresse von $b$ und in $new$ eine später neu zugewiesene Anfangsadresse. Die ursprünglichen Adressen der Basisblöcke können aus den Metadaten stammen oder werden während der Basisblockrekonstruktion generiert. Die maximale Anzahl an Einträgen von *bbs* wird während der Übersetzungszeit des Systems festgelegt anhand des Wertes der Funktion mit den meisten Basisblöcken. Nachdem die Rekonfiguration einer Funktion abgeschlossen ist, werden die Informationen bezüglich *bbs* verworfen.

Zur Rekonfiguration der aktuellen Funktion wird diese in Codefragmente, auch als Codefenster bezeichnet, zerlegt. Die maximale Größe des Codefensters ist statisch fixiert und wird zur Übersetzungszeit festgelegt. Das Codefenster besteht aus einer Startadresse und einer Endadresse. Die maximale Fenstergröße ist so festgelegt, dass der längste Basisblock (aus der Übersetzungszeit) zuzüglich einer festgelegten Toleranz für eine Verlängerung des Ablaufplans mit seiner Start- und Endadresse in das Fenster passt. Das Codefenster wird während der Rekonfiguration schrittweise über den Speicherbereich der Funktion bewegt und die eigentliche Anpassung des Programmcodes erfolgt innerhalb der Grenzen des durch das Codefensters beschriebenen Speicherbereichs. Der Programmcode, der durch das Fenster überdeckt wird, wird in den Datenspeicher kopiert. Der kopierte Programmcode wird dekodiert und die dekodierten Operationen werden in die Operationsliste *ops* eingetragen.

### Die Metadaten für Funktionen und Basisblöcke

Die eingesetzte Reparaturstrategie basierte darauf, bestimmte Information im Voraus zu kennen. Die erforderlichen Informationen sind Metadaten bezüglich der Anwendung. Die Metadaten umfassen zunächst die Startadressen von den Funktionen einer Anwendung. Weiterhin werden zu jeder Funktion die Zieladressen von Rückwärtssprüngen erfasst, die innerhalb des Speicherbereichs einer jeweiligen Funktion liegen. Jede dieser Adressen ist der Beginn eines Basisblocks.

Anhand des beschriebenen Prozessormodells besteht nur im Programmspeicher die Möglichkeit, vorinitialisierte Daten zu hinterlegen. Um den Speicherplatz effizient zu nutzen, werden im Programmspeicher die Metadaten kompakt kodiert. Das Auslesen erfolgt mit Hilfe des Programmtransfer-Controllers, mit dem Speicherbereiche zwischen Programm- und Datenspeicher kopiert werden können. Im Vergleich zum Laden von Konstanten über entsprechende Befehlssequenzen, kann ab einer gewissen Speicherbreite eine Ersparnis erreicht werden.

Abbildung 4.7 zeigt das Layout des Programmspeichers mit den hinterlegten Metadaten. Der vordere Bereich des Programmspeichers bis zum Beginn der eigentlichen Anwendung ist der Code, der nicht durch die Rekonfiguration verschoben wird. Dabei handelt es sich um den Programmcode der Reparatur-Routinen und die Metadaten mit den Adressen zu den Funktionen. Der Programmcode der eigentlichen Anwendung setzt sich aus den einzelnen Funktionen zusammen. Zu Beginn einer jeden Funktion sind die Metadaten bezüglich der Sprungadressen eingetragen. Der auszuführende Code einer jeden Funktion beginnt nach den Metadaten.

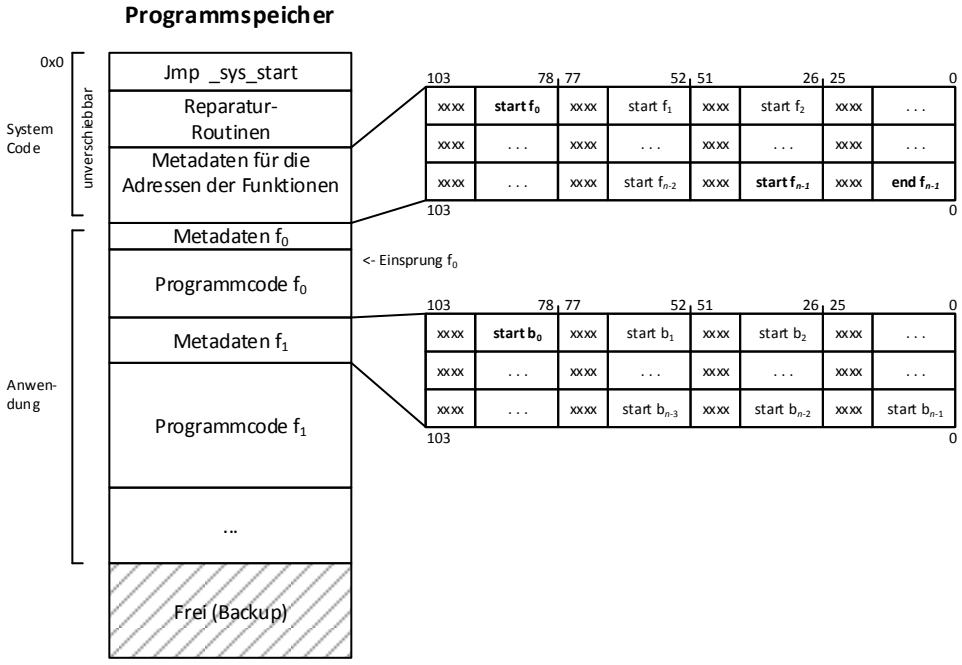


Abbildung 4.7.: Layout des Programmspeichers bezüglich der Metadaten und deren Aufteilung

Der Aufbau der Metadaten ist in Abbildung 4.7 im rechten Teil dargestellt. Es ist einmal der Metadatenbereich für die Startadressen der Funktionen und für die Basisblockadressen der Funktion $f_1$ gezeigt. Das Layout der Metadaten ist für einen Programmspeicher, der ein Instruktionswort mit 4 Operationen speichert, angegeben. Ein Instruktionswort für 4 FUs ist 104 Bit breit. Bei einer 16 Bit Adressierung können 6

Adressen pro Instruktionswort gespeichert werden. Die restlichen Bits werden mit Nullen aufgefüllt. Damit die 6 Adressen sequentiell in den Datenspeicher überführt werden können, wird der Programmtransfer-Controller um einen Modus erweitert, der ein Instruktionswort ohne zusätzliche Ausrichtung in den Datenspeicher überführt. Dadurch kann eine effizientere Nutzung erfolgen im Vergleich zum Transfer-Modus für Instruktionswörter, denn dieser richtet transferierte Daten anders im Datenspeicher aus. Die Größe der jeweiligen Metadaten ist statisch fixiert. Im Metadatenbereich für die Startadressen der Funktionen wird der Beginn des jeweiligen Metadatenbereichs eingetragen. Der Beginn des tatsächlichen Programmcodes einer jeden Funktion wird anhand der Metadatengröße und der hinterlegten Adresse errechnet.

### 4.3.2. Rekonstruktion der Basisblöcke

Um die Steuerflussabhängigkeiten im ursprünglichen Programm zu erhalten, arbeiten das Rescheduling und die Registerallokation auf der Grundlage von Basisblöcken. Das im Folgenden vorgestellte Verfahren wird dazu eingesetzt, die notwendigen Informationen zu den Basisblöcken aus dem Code einer Anwendung zu rekonstruieren. Bekannte Algorithmen für diese Problematik sind in [98] und [3] zu finden. Allerdings basieren diese Verfahren darauf, dass sie den tatsächlichen Eingabeprogrammtext im Ganzen analysieren können. Aufgrund der bereits erwähnten eingeschränkten Ressourcen wird ein angepasstes Verfahren benötigt, welches auch dann alle Basisblöcke erkennt, wenn der Binärcode nur in Teilausschnitten analysiert werden kann.

Aus dem zurückliegenden Abschnitt ist bekannt, dass die Basisblockrekonstruktion für jedes Codefragment durchgeführt wird. Die Nutzungsdauer der rekonstruierten Informationen beschränkt sich auf die jeweils in Abarbeitung befindliche Funktion. Das Verfahren zur Rekonstruktion muss nun sicherstellen, dass es, nachdem ein Codefragment analysiert wurde, alle zurückliegenden und im Codefragment befindlichen Basisblöcke erkannt hat, ohne den nachfolgenden Code betrachtet zu haben. Das Verfahren untersucht die Operationen der Operationsliste, bei der ersten Operation beginnend. Durch diese Vorwärtsanalyse entsteht das Problem, dass Basisblöcke, auf die Rücksprünge aus dem noch nicht analysierten Code existieren, nicht immer erkannt werden können. Würde nun eine Rekonfiguration, basierend auf den unvollständig erkannten Basisblöcken, erfolgen, würden unter Umständen Operationen in andere Basisblöcke vorschoben werden. Aus diesem Grund werden dem Rekonstruktionsverfahren solche Rücksprungadressen als Metadaten zur Verfügung gestellt. Durch die Kombination der Vorwärtsanalyse mit im Vorfeld bekannten Rücksprungadressen können die Basisblöcke korrekt erkannt werden.

Die Basisblöcke werden durch den Algorithmus zur Rekonstruktion in die Basisblockliste *bbs* eingetragen. Da es sich um ein statisches Feld handelt, kann es vorkommen, dass bereits bekannte Blöcke im Feld umkopiert werden müssen, damit ein neu erkannter Block eingefügt werden kann. Somit wird während der Rekonstruktion die Basisblockliste durch Zerlegen und Verschieben bereits erkannter Blöcke schrittweise verfeinert. Zu Beginn der Funktionsabarbeitung ist nur ein Basisblock in der Basisblockliste eingetragen mit der Anfangsadresse der Funktion. Mit dem Einlesen der Metadaten wird dieser eine Block in weitere Blöcke verfeinert. Während der Rekonstruktion durchläuft

der Algorithmus die Operationsliste sequentiell. Wird eine Sprunoperation gefunden, dann gilt:

- Die Sprungoperation selbst befindet sich in der letzten Instruktion eines Basisblocks und die nachfolgende Instruktion ist die Anfangsadresse eines weiteren Basisblocks.
- Die Zieladresse der Sprungoperation kann einen noch nicht bekannten Basisblock definieren. Allerdings ist diese Untersuchung nur für Vorwärtssprünge relevant, weil Basisblöcke, die durch eine Rückwärtsverzweigung angesprungen werden, durch die Metadaten bekannt sind.

Die Behandlung der beiden Fälle ist in Algorithmus 4.7 dargestellt. Der Algorithmus beginnt mit dem Eintragen des ersten Basisblocks in *bbs* (Zeile 4). Dabei handelt es sich um die Adresse der ersten Operation der übergebenen Operationsliste *ops*. Mit der Hauptschleife in Zeile 5 wird einmal über die gesamte Liste iteriert. In jedem Schritt wird geprüft, ob es sich bei der gerade untersuchten Operation *op* um eine Sprungoperation oder einen Unterprogrammaufruf handelt(Zeile 6). Liegt der entsprechende Operationstyp vor, so beendet die Instruktion, die *op* enthält, einen Basisblock. Die Adresse $a + 1$ bezüglich *op* definiert einen neuen Block. Um dies in der Basisblockliste zu vermerken, wird in *bbs* der erste Eintrag gesucht, dessen Adresse größer oder gleich $a + 1$ ist (Zeile 7). Zur Auswertung werden drei Fälle unterschieden:

1. Es wurde kein entsprechender Basisblock gefunden (Zeile 8), der sich an der Adresse $a$ befindet. Der Basisblock mit der Adresse $a + 1$ wird an das Ende von *bbs* angefügt.
2. Ein entsprechender Basisblock mit der Adresse $a + 1$ ist schon vorhanden und es kann fortgefahren werden (Zeile 10).
3. Es wurde ein Eintrag $b$ mit $b > a$ gefunden (Zeilen 12 bis 15). Somit muss der neue Basisblock innerhalb des Feldes eingefügt werden und alle Einträge ab $b$ müssen verschoben werden.

In den Zeilen 17 bis 25 erfolgt die Behandlung der Sprungzieladresse von *op*. Die Behandlung der Zieladresse muss lediglich für Sprungoperationen, nicht aber für Unterprogrammaufrufe erfolgen. Die Suche nach einem passenden Eintrag und die Unterscheidung der Fälle ist dabei grundsätzlich gleich zur Behandlung der Adresse von *op* selbst.

Damit Algorithmus 4.7 sicher alle Basisblöcke erkennen kann, müssen die folgenden Forderungen bei der Codeerzeugung während der Systemübersetzung eingehalten werden:

- Die Adressen von Basisblöcken, die nur durch Rückwärtssprünge erreicht werden, müssen in den Metadaten hinterlegt werden,
- Startadressen von Basisblöcken, die durch Sprünge mittels Registerinhalt erfolgen, müssen durch eine Operation des selben Basisblocks in das Register geladen werden und

**Algorithm 4.7** Basisblock-Rekonstruktion

```
1: Eingabe: Operationsliste ops, Basisblock-Liste bbs
2: Ausgabe: Basisblock-Liste bbs'
3:
4: - füge neuen Block in bbs ein, mit der Adresse von ops(0)
5: for all op aus ops do
6:     if type(op) ist Vorwärtssprung then
7:         - suche in bbs den ersten Block b dessen Anfangsadresse größer oder gleich
           der Adresse addr(op) + 1 ist
8:         if kein b gefunden then
9:             - erweitere bbs um neuen Block, mit addr(op) + 1 als Adresse
10:        else if Adresse von b entspricht addr(op) + 1 then
11:            continue // Basisblock bereits bekannt
12:        else
13:            - kopiere die Einträge in bbs ab der Position von b um einen Eintrag
14:            - füge neuen Block in bbs ein, mit addr(op) + 1 als Adresse
15:        end if
16:
17:        - suche in bbs den ersten Block b dessen Anfangsadresse größer oder gleich
           der Sprungadresse von op ist
18:        if kein b gefunden then
19:            - erweitere bbs um neuen Block, mit Sprungadresse von op
20:        else if Adresse von b gleich der Zieladresse von op then
21:            continue // Basisblock bereits bekannt
22:        else
23:            - umkopieren der Einträge in bbs ab b
24:            - füge neuen Block in bbs ein, mit Sprungadresse von op
25:        end if
26:    end if
27: end for
```

- Sprungoperationen über Registerinhalte, die zuvor berechnet wurden, sind nicht zulässig.

Der letzte Punkt soll den Umfang der Metadaten reduzieren und dem Rekonstruktionsverfahren die Möglichkeit bieten, Sprünge über Registerinhalte zu erkennen, ohne auf eine aufwendige Rückwärtsanalyse angewiesen zu sein. Eine Datenflussanalyse bezüglich berechneter Sprungziele erfolgt nicht.

### 4.3.3. Sprunganpassung aufgrund verlängerter Ablaufpläne

Die beiden komplexen Planungsverfahren, das Rescheduling und die Registerallokation, können Operationen einem späteren Startzeitpunkt zuweisen als ursprünglich geplant. Die dadurch verlängerten Basisblöcke erzeugen Verschiebungen des Codes im Programmspeicher. Aufgrund der Adressänderung von Instruktionen sind Sprungope-

rationen ungültig, die zu einer Instruktion anhand der alten Adresse verzweigen. Um dieses Problem zu vermeiden, ist es erforderlich, die Zieladressen von Sprungoperationen an die veränderte Situation im Programmspeicher anzupassen.

Die Anpassung der Sprungoperationen erfolgt, nachdem ein Basisblock geplant wurde, weil dann auf eine am Ende vorhandene Sprungoperation einfach zugegriffen werden kann. Das Verfahren zum Sprung-Patching arbeitet in zwei Stufen. Für Rückwärtssprünge bezüglich der aktuellen Position kann die geänderte Zieladresse sofort ermittelt und geändert werden. Dagegen können Vorwärtssprünge nicht sofort angepasst werden, weil der nachfolgende Code noch nicht rekonfiguriert wurde und eine mögliche Offset-Verschiebung noch nicht bekannt ist. Solche Operationen werden sich vom Verfahren gemerkt und die Adressanpassung erfolgt zu einem späteren Zeitpunkt. Für das Sprung-Patching zu einem späteren Zeitpunkt werden zwei Fälle unterschieden im Hinblick auf den Zugriff auf die zu patchende Operation. Befindet sich eine Sprungoperation im aktuellen Codefragment, kann auf sie über die Operationsliste *ops* zugegriffen werden. Operationen die außerhalb des aktuellen Fragments liegen befinden sich im bereits rekonfigurierten Teil des Programmspeichers und müssen zum Patchen erneut aus dem Programmspeicher in den Datenspeicher eingelesen werden.

Um die beiden Fälle zu unterscheiden, merkt sich das Verfahren die Operationen in zwei verschiedenen Listen. Die Einträge in der lokalen Patchliste *lpl* sind Paare $(a, i)$, wobei $a$ das Sprungziel einer Sprungoperation $v$ ist und $i$ der Index der Operationsliste, damit gilt $ops(i) = v$. In der globalen Patchliste *gpl* werden Paare $(a, target)$ gespeichert, wobei $a$ die neue Adresse einer Operation $v$ ist und $target$ das ursprüngliche Sprungziel von $v$.

Bezüglich der alten Anfangsadresse *old* eines Basisblocks $b$ und der neuen Adresse *new* untersucht das Verfahren nun zwei Fälle:

1. Es wird in der globalen Patch-Liste nach allen Einträgen $gpl(i)$ gesucht für die gilt $target = old$. Wird ein solcher Eintrag gefunden, dann befindet sich die referenzierte Operation im Programmspeicher, und zwar in dem bereits rekonfigurierten Codebereich. Als Sonderfall ist der erste Basisblock einer Funktion $f$ zu behandeln. Ein gefundener Eintrag in *gpl* bedeutet dann, dass es sich bei der entsprechenden Operation um einen Funktionsaufruf bezüglich $f$ handelt.

2. In der lokalen Patch-Liste werden alle Einträge $lpl(i)$ gesucht, für welche die gespeicherte Sprungadresse $a$ der von *old* entspricht. Zu einem Eintrag ist dann in *lpl* ein Verweis in die Operationliste hinterlegt. Für die darüber zu findende Operation wird sich der Wert *new* als neues Sprungziel gemerkt.

Nachdem gespeicherte Sprungoperationen gepatcht sind, behandelt das Verfahren eine mögliche Sprungoperation *op* am Ende des gerade verarbeiteten Basisblocks $b$. Ist *op* ein Rückwärtssprung mit einer Zieladresse $a$ innerhalb der aktuellen Funktion, muss ein Eintrag $bbs(i)$ existieren mit $a = old$. Bei einem Rückwärtssprung über den Bereich der aktuellen Funktion hinaus kann über die Funktionsliste die jeweilige neue Adresse gefunden werden. Führt *op* einen Sprung zu höheren Adressen aus, wird anhand der Größe des Codefensters entschieden, ob *op* in *lpl* oder in *gpl* zu speichern ist.

**Algorithm 4.8** Sprung-Patching

```
1:  Eingabe: Operationsliste ops, Basisblock-Liste bbs, Basisblock b
2:               globale Patch-Liste gpl, Funktionsliste funcs
3:
4:  for j := 1 to n do
5:      if Sprungadresse a von lpl(j) gleich der Adresse old von bbs(b) then
6:          - setze die Adresse new von bbs(b) als Sprungziel der Operation ops(i), für
            den Index i von lpl(j)
7:          - entferne den Eintrag j aus lpl
8:      end if
9:  end for
10: for k := 1 to n do
11:     if Sprungadresse target von gpl(k) entspricht der Adresse old von bbs(b) then
12:         - lade Instruktion w der Adresse a des Eintrags gpl(k) in den Speicher
13:         - setze das Sprungziel in w auf die neue Adresse new von bbs(b)
14:         - schreibe w zurück und entferne den Eintrag k aus gpl
15:     end if
16: end for
17: if b besitzt eine Sprungoperation u then
18:     if u verwendet Register r für Sprung then
19:         - suche in ops die Operation v die r mittels Konstante lädt
20:         if kein solches v gefunden then
21:             - continue // Rücksprung aus Funktion gefunden
22:         end if
23:     end if
24:     if Zieladresse a von v bzw. u ist kleiner der Adresse old von bbs(b) then
25:         - suche Basisblock d dessen Adresse a entspricht
26:         if kein solches d gefunden then
27:             - suche in funcs nach der zu verwendenden Adresse für v bzw. u
28:         else
29:             - aktualisiere v bzw. u mit der Adresse new von bbs(d)
30:         end if
31:     else if Zieladresse a von v bzw. u liegt im aktuellen Code-Fenster then
32:         - füge Eintrag (a, i) in lpl ein, mit ops(i) = v bzw. ops(i) = u
33:     else
34:         - füge (addr, a) in gpl ein, wobei addr die neue Adresse von v bzw. u ist
35:     end if
36: end if
```

Algorithmus 4.8 beschreibt das Verfahren zum Sprung-Patching ausführlicher. Der Algorithmus prüft zunächst, ob in der lokalen Patchliste *lpl* Verweise auf den übergebenen Basisblock *b* vorhanden sind (Zeile 4 bis 9). Ist dies der Fall, dann wird über den Querverweis aus *lpl* die Operation in *ops* gepatcht und der Eintrag wird aus *lpl* entfernt. Anschließend untersucht der Algorithmus, ob in der globalen Patch-Liste Einträge mit der alten Adresse von *b* vorhanden sind (Zeile 11 bis 17). Wird ein Eintrag gefunden,

dann gibt die im Eintrag hinterlegte Adresse die Instruktion im Programmspeicher an, welche die zu patchende Operation $v$ enthält. Um $v$ anzupassen, muss die Instruktion in den Datenspeicher eingelesen werden. Nachdem $v$ mit der neuen Anfangsadresse von $b$ gepatcht wurde, wird die Instruktion in den Programmspeicher zurückgeschrieben und der Eintrag wird aus *gpl* gelöscht.

In den Zeilen 19 bis 38 erfolgt die Behandlung einer Sprungoperation $u$ in der letzten Instruktion von $b$. Verwendet $u$ den Inhalt eines Registers, um den Sprung durchzuführen, muss nach einer Operation $v$ gesucht werden, die das entsprechende Register mit einer Konstanten lädt (Zeile 20 bis 25). Wird keine solche Operation gefunden, wird davon ausgegangen, dass es sich um einen Rücksprung aus einer Funktion handelt. Ist die Zieladresse von $u$ bzw. von $v$ kleiner der alten Anfangsadresse von $b$, wird nach der neuen Adresse entweder in *bbs* gesucht oder aber in $funcs$, wenn es sich um einen Funktionsaufruf handelt. Handelt es sich um einen Vorwärtssprung, wird die Operation entweder in *lpl* gespeichert oder in *gpl*, falls die Zieladresse außerhalb des Bereichs des Codefragments liegt.

Das Verfahren funktioniert für Sprungoperationen, die einen Immediate-Wert verwenden, problemlos. Für Sprung- bzw. Verzweigungsoperationen, die über den Inhalt eines Registers $r$ den Programmzähler laden, sind weitere Unterscheidungen zu treffen:

- Das Register $r$ muss in einer vorherigen Operation $v$ des selben Basisblocks mit einer Konstanten geladen werden. Die Operation $v$ ist dann die Operation, die gepatcht werden muss. Wenn der aktualisierte Wert der Zieladresse noch nicht bekannt ist, wird sich $v$ gemerkt und zu einem späteren Zeitpunkt angepasst.

- Sollte keine Operation gefunden werden, die $r$ mit einer Konstanten lädt, wird die Sprungoperation nicht vom Verfahren beachtet. Es wird davon ausgegangen, dass es sich dann um einen Rücksprung aus einer Funktion handelt, dessen Ziel dynamisch während der Programmausführung ermittelt wurde.

Die genannten Punkte stellen gleichzeitig Anforderungen an die Codeerzeugung der Anwendung. Das Verfahren zum Sprungpatching erwartet die genannten Muster, um Zieladressen anpassen zu können. Sollten jedoch Sprungziele berechnet werden, dann wird dies nicht vom Verfahren behandelt. Ein entsprechender Code muss vermieden und gegebenenfalls während der Übersetzungszeit in die genannten Muster transformiert werden.

Das folgende Beispiel verdeutlicht das Vorgehen zur Anpassung der Sprungbefehle und stellt die Sonderfälle heraus. Abbildung 4.8(a) zeigt einen Ausschnitt aus dem Programmspeicher. In diesem Beispiel ist das aktuelle Code-Fenster durch die gestrichelte Box gekennzeichnet. Der Code des Fensters enthält vier Basisblöcke. Die Einträge nach der Basisblockrekonstruktion sind in der Liste *bbs* gezeigt. Der Basisblock mit der Adresse $0x15$ wurde bereits rekonfiguriert. Es wurde eine neue Adresse mit dem Wert 16 vergeben, da sich eine Verlängerung um eine Instruktion ergab. Aus vorangegangenen Planungsteilschritten ist ein Vorwärtssprung in den Code des aktuellen Fensters bekannt. Die entsprechende Adresse $0x17$ ist in einer Liste (als global bezeichnet) eingetragen. Abbildung 4.8(b) zeigt die entsprechende Startbelegung der restlichen Datenstrukturen. Es ist eine weitere Liste (als lokal bezeichnet) verzeichnet, in der die Adressen von Sprungoperationen bezüglich des aktuellen Fensters gemerkt

werden. Weiterhin ist auch die Operationsliste *ops* dargestellt mit einem Eintrag für den Index 25.

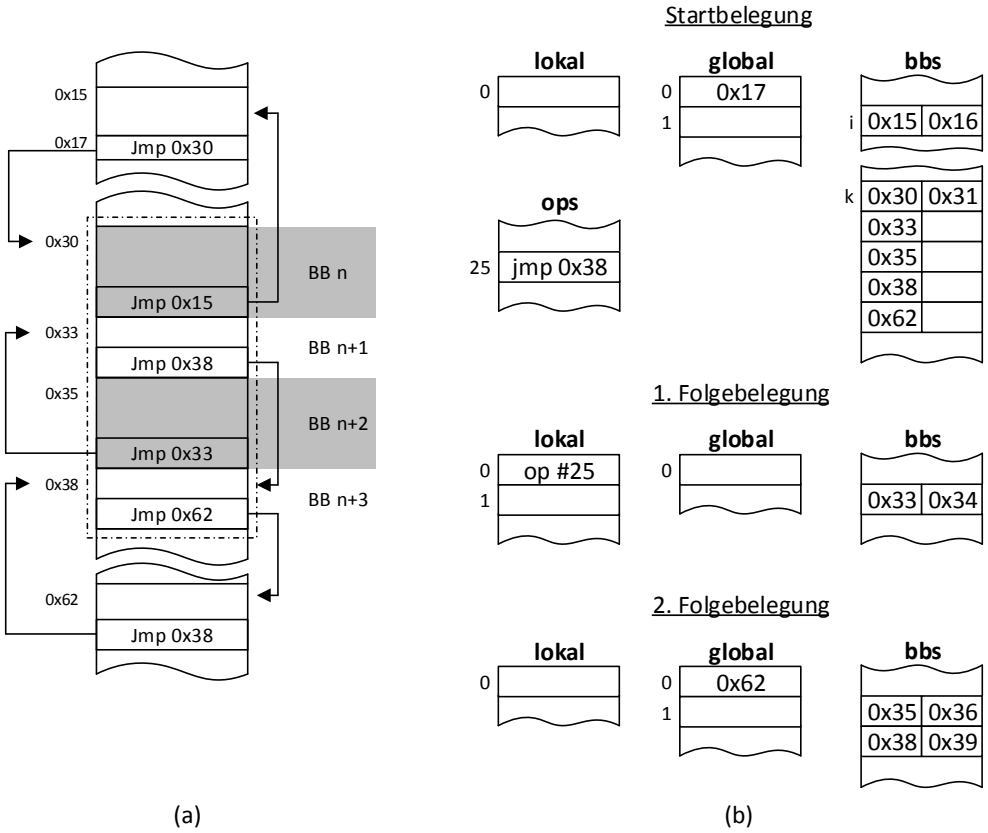


Abbildung 4.8.: (a) Beispiel-Code im Programmspeicher zur Sprungbefehlanpassung (b) Entwicklung der Einträge in den Listen während des Sprung-Patching

Es können nun die Basisblöcke durch die entsprechenden Planungsverfahren angepasst werden. Nachdem Basisblock $b_n$ geplant wurde, kann der Einsprung zu $b_n$ angepasst werden sowie der Sprungbefehl *op*, der $b_n$ verlässt. Die aktualisierte Zieladresse von *op* kann über die Basisblockliste *bbs* abgefragt werden, da dieser Code-Bereich schon geplant wurde. Anhand der globalen Sprungzielliste kann der Sprung gefunden werden, der noch nicht über die aktualisierte Startadresse von $b_n$ verfügt. Danach kann der Basisblock $b_{n+1}$ bearbeitet werden. Es wird die neue Startadresse $0x34$ von $b_{n+1}$ zusammen mit der alten gemerkt. Die Adresse ist um eine Instruktion erhöht, da sich vorher schon eine Verlängerung in der Planung ergab. Der Block $b_{n+1}$ wird mit einem Vorwärtssprung verlassen, welcher innerhalb des aktuellen Code-Fensters endet. Der Sprungbefehl mit dem Index 25 wird in der lokalen Patch-Liste gemerkt und zu einem späteren Zeitpunkt aktualisiert. Da sich der Sprung im aktuellen Code-Fenster befindet, sind auch die detaillierten Informationen in der Operationsliste *ops* vorhanden.

Somit ist es ausreichend, einen Verweis in die Liste *ops* zu speichern. Die aktuelle Belegung der Listen zeigt Abbildung 4.8(b) unter dem Punkt der ersten Folgebelegung. Der Basisblock $b_{n+2}$ weist keine Besonderheiten auf. Die Startadresse von $b_{n+2}$ wird in *bbs* aktualisiert und der verlassende Sprungbefehl kann mit Hilfe von *bbs* entsprechend aktualisiert werden. Der letzte Block $b_{n+3}$ enthält einen Sonderfall, nämlich den letzten Sprungbefehl, welcher in ein nachfolgendes Code-Fenster verzweigt. Um diesen zu behandeln, wird die Adresse der Sprungoperation in der globalen Patch-Liste hinterlegt. Die entsprechend geänderte Belegung der Datenstrukturen zeigt Abbildung 4.8(b) als zweite Folgebelegung.

### 4.3.4. Steuerung der lokalen Reparatur

Algorithmus 4.9 beschreibt den organisatorischen Ablauf zur lokalen Rekonfiguration einer Anwendung. Mit dem Zeiger $globalOffset$ wird die aktuelle Position des Programmspeichers referenziert, an die das nächste fertig geplante Codefragment zurückgeschrieben wird. Entsprechend der sich ergebenden Fragmentlänge wird der Zeiger erhöht. Zu Anfang wird $globalOffset$ auf die Startadresse der ersten Funktion gesetzt. Danach wird die gesamte Anwendung an des Ende des Programmspeichers kopiert. Anschließend werden die Metadaten zu den Funktionen eingelesen und $funcs$ initialisiert. Zur Bearbeitung einer Funktion $f$ werden zu Beginn die Metadaten bezüglich der Sprungadressen eingelesen und die Basisblockliste wird neu initialisiert. Danach wird das Codefenster $cWnd$ auf die erste Instruktion von $f$ ausgerichtet und der Code des dadurch beschriebenen Fragments wird in einen Zwischenspeicher im Datenspeicher eingelesen. Ist die Ausführung eines komplexen Backends erforderlich, werden zunächst die Informationen bezüglich der Basisblöcke rekonstruiert. Die Basisblöcke werden einzeln rekonfiguriert mit einer anschließenden Anpassung der Sprunginformationen. Am Ende kann das angepasste Codefragment in den Programmspeicher zurückgeschrieben werden.

Das Umkopieren der Anwendung (Zeile 6) erfolgt rückwärts, von der Endadresse der letzten Funktion bis zur Startadresse der ersten Funktion. Das Umkopieren erfolgt mit I/O-Operationen, die den Programmtransfercontroller entsprechend ansteuern (vgl. Kap 3.3). Es wird ein spezieller Modus genutzt, der das Umkopieren innerhalb des Programmspeichers erlaubt, ohne dass ein Zwischenspeichern im Datenspeicher erforderlich ist. Mit der äußeren Schleife (Zeile 7 bis 22) werden nacheinander die Funktionen aus $funcs$ bearbeitet. Der aktuelle Wert von $globalOffset$ wird als neue Startadresse zu $f$ gespeichert (Zeile 10), um später Funktionsaufrufe mit genau diesem Wert zu aktualisieren. Zum Initialisieren des Codefensters wird die Startadresse von $cWnd$ auf die Anfangsadresse der ersten Funktion gesetzt (Zeile 11), die nach dem Verschieben an das Programmspeicherende einen zusätzlichen Offset erhält. Des Weiteren ist zu beachten, dass das Codefenster nicht über den Bereich der Funktion hinaus geht. Für diesen Fall wird das Ende des Codefensters auf die letzte Adresse von $f$ ausgerichtet.

Die innere Schleife (Zeile 12 bis 21) bewegt das Codefenster $cWnd$ über den Adressbereich der aktuell bearbeiteten Funktion $f$. In Zeile 13 erfolgt das Einlesen des Programmcodes in den Puffer im Datenspeicher durch Ansteuerung des Programm-Transfer Controllers. Dieser Vorgang beinhaltet auch das Decodieren des Befehlscodes aus

**Algorithm 4.9** Steuerung der lokalen Reparatur

```
1: Eingabe: Start- und Endadressen der Anwendung app
2: Ausgabe: keine
3:
4: - globalOffset := Startadresse von app
5: - einlesen der Metadaten und initialisieren von funcs
6: - verschieben von app an das Ende des Programmspeichers
7: for all f aus funcs do
8:     - einlesen der Metadaten von f und initialisieren von bbs
9:     - schreibe Metadaten von f zurück an Adresse globalOffset
10:    - vermerke in funcs den Wert von globalOffset als neue Startadresse von f
11:    - initialisieren von cWnd mit der ersten Codeadresse von f
12:    repeat
13:        - lese Code von cWnd in ops ein
14:        - führe Basisblockrekonstruktion mit Algorithmus 4.7 durch
15:        for all Basisblöcke b gefunden in cWnd do
16:            - rekonfiguriere b basierend auf dem Defektzustand
17:            - führe Sprungpatching mit Algorithmus 4.8 durch
18:        end for
19:        - schreibe rekonfigurierten Code zurück und aktualisiere globalOffset
20:        - setze cWnd weiter
21:    until Endadresse von f erreicht
22: end for
```

dem Puffer in die Operationsliste *ops*, die als Eingabe für die verschiedenen Backends dient. Für den Code erfolgt dann die Basisblockrekonstruktion (Zeile 14). Anschließend wird für jeden erkannten Basisblock eine Rekonfiguration mit dem passenden Backend vorgenommen (Zeile 16). Zur Rekonfiguration kann ein Wechsel der Backends in Abhängigkeit vom Defektzustand des Prozessors erfolgen. Die Auswahl der Backends ist nicht explizit im Algorithmus angegeben und soll im Folgenden genauer besprochen werden.

Tabelle 4.1 zeigt eine Übersicht zu den relevanten Kriterien, anhand derer die Auswahl des Backends erfolgt. Zu den Kriterien zählt, ob ein lokal oder global genutztes Register defekt ist und ob Reserveregister vorhanden sind. Darüber hinaus wird berücksichtigt, ob weitere Komponenten defekt sind. Anhand der drei zuerst genannten Kriterien wird entschieden, welches Backend zur Behandlung eines defekten Registers ausgeführt wird. Wenn ein globales Register defekt ist, wird dies immer durch ein lokales Register ersetzt (Renaming). Das fehlende lokale Register kann dann durch ein Reserve-Register kompensiert (Renaming) oder durch eine Neuvergabe der Registerallokation behandelt werden. Die Behandlung eines fehlenden lokalen Registers ist identisch mit der Behandlung eines defekten lokalen Registers. Sind andere Komponenten außer einem Register defekt, erfolgt zunächst immer ein Rebinding. Erst wenn das Rebinding eine Instruktion eines Basisblocks $b$ nicht erfolgreich anpassen konnte, wird $b$ erneut durch das Rescheduling rekonfiguriert. Wird eine Register-Allokation durchge-

führt, wird diese auch genutzt, um Defekte in anderen Komponenten zu behandeln. Dies ist möglich, weil der Algorithmus zur Register-Allokation auf dem Rescheduling basiert und dadurch auch eine Ablaufplanung vornehmen kann. In Tabelle 4.1 ist in der ersten Zeile ein Sonderfall angegeben für den Fall dass Reserve-Register vorhanden sind. Bei dieser Konstellation wird keine Register-Allokation durchgeführt. Statt dessen wird das defekte globale Register durch ein lokales ersetzt und das lokale Register durch ein Reserve-Register. Der Ausfall weiterer Komponenten muss dann durch das Rebinding behandelt werden. Sind in einer Zeile Sterne vorhanden, dann ist das jeweilige Kriterium nicht relevant für die Entscheidungsfindung.

Nach der Rekonfiguration des Codes werden die Sprunginformationen aktualisiert, die sich aus einer verlängerten Ablaufplanung ergeben (Zeile 17). Die neue Anfangsadresse von $b$ ergibt sich aus dem Wert von $globalOffset$ und wird in $bbs$ vermerkt. Ist in der letzten Instruktion von $b$ eine Sprungoperation enthalten, dann muss deren Sprungziel angepasst werden. Falls eine Anpassung nicht sofort möglich ist, weil die Operation zu nachfolgenden Adressen verzweigt, muss die Anpassung zu einem späteren Zeitpunkt erfolgen und die Operation muss sich gemerkt werden. Nachdem alle Basisblöcke des aktuellen Fragments geplant sind, wird der Programmcode erzeugt und in den Programmspeicher zurückgeschrieben. Das Codefenster wird auf die Startadresse des nächsten Basisblocks ausgerichtet, der noch nicht umgeplant wurde. Bezüglich der Endadresse von $cWnd$ muss beachtet werden, dass diese nicht über den Speicherbereich der Funktion hinaus geht (Zeile 20).

| lok. Reg. defekt | glob. Reg. defekt | Spare-Reg. vorhanden | sonstige Defekte | auszuführende Backends |
|---|---|---|---|---|
| * | 1 | 1 | 1 | Reg-Allok. + Rebinding |
| * | 1 | 0 | * | Reg-Allok. |
| 1 | 0 | 0 | * | Reg-Allok. |
| 1 | 0 | 1 | 0 | Renaming |
| 0 | 0 | * | 1 | Rebinding |
| 1 | 0 | 1 | 1 | Renaming + Rebinding |

Tabelle 4.1.: Wahrheitstabelle für die Entscheidungsfindung welches Backend zur Rekonfiguration auszuführen ist

# 5. Globale Reparatur im Mehrkernsystem

Die bisher präsentierten Rekonfigurationsmöglichkeiten beschränken sich auf die lokale Reparatur eines einzelnen VLIW-Kerns. Dazu wurden Techniken vorgestellt, die fein-granular die durch die Anwendung verwendeten Prozessorbaugruppen an eine Fehlersituation anpassen. Zur Rekonfiguration ist die Planung von Operationen in spätere Instruktionen zulässig und eine degradierte Systemleistung wird für einen rekonfigurierten Kern akzeptiert. Aufgrund einer zu hohen Degradation können sich Probleme ergeben und eine lokale Rekonfiguration kann fehlschlagen. Es gibt zwei wesentliche Gründe, weshalb die lokale Rekonfiguration eines Kerns fehlschlagen kann:

1. Der lokale Reparaturalgorithmus kann auf dem defekten Kern nicht ausgeführt werden, weil der Programmcode des Algorithmus die Verwendung einer defekten Komponente vorsieht und

2. durch den Algorithmus kann keine gültige Ablaufplanung für die Anwendung bestimmt werden, weil die neue Ablaufplanung gegebene Ressourcenbeschränkungen verletzt.

Der erste Punkt ergibt sich, weil das vorgestellte Konzept auf der Idee einer Selbstreparatur basiert und die Ressourcennutzung der Reparaturalgorithmen zur Systemübersetzungszeit festgelegt wird. Die im zweiten Punkt genannte Verletzung der Ressourcenbeschränkung entsteht, wenn die Summe der einzelnen verlängerten Basisblöcke den bereitgestellten Reservebereich im Programmspeicher übersteigt.

Die lokale Reparatur hat keinen Zugriff auf die systemweiten Reservekomponenten und es kann die Situation entstehen, dass

- eine fehlgeschlagene lokale Reparatur einen Systemausfall bedingt, obwohl auf anderen Kernen ungenutzte Spare-Komponenten bereit stehen und

- die systemweiten Reservekomponenten nicht optimal genutzt und minimal gehalten werden können.

Im folgenden Kapitel wird eine erweiterte globale Reparaturstrategie vorgestellt, welche die Beschränkungen der lokalen Verfahren überwindet und Lösungen für die genannten Probleme bereitstellt. Das Verfahren ist hierarchisch organisiert und setzt die lokalen Reparaturmethoden in einem globalen Systemkontext erneut ein. Durch eine systemweite Rekonfiguration können sich Kerne gegenseitig anpassen und es besteht die Möglichkeit, Reservekomponenten nicht nur kernübergreifend einzusetzen, sondern auch deren Anzahl zu reduzieren.

Die globale Reparaturstrategie stellt zwei Verfahren bereit, mit denen die zuvor genannten Fälle behandelt werden. Die erste Methode, die Fremdreparatur, passt die Reparaturroutinen eines defekten Kerns an die Fehlersituation an, damit der defekte

Kern anschließend eine Selbstreparatur durchführen kann. Das zweite Verfahren kommt dann zum Einsatz, wenn eine lokale Reparatur aufgrund von Verletzungen der Ressourcenbeschränkung nicht erfolgreich war. In dem Fall wird die Zuordnung zwischen den Anwendungen und den Kernen geändert. Ziel dabei ist es, durch das Vertauschen von Anwendungen diese erfolgreich an einen neu zugewiesenen Kern anpassen zu können, wenn die Anwendungen unterschiedliche Leistungsanforderungen haben. Der Vorgang wird als Task-Rebinding bezeichnet.

Das folgende Kapitel ist so organisiert, dass im nächsten Abschnitt die Administration der globalen Reparatur besprochen wird. Anschließend wird das Verfahren zur Fremdreparatur präsentiert und das Kapitel endet mit der Diskussion des Task-Rebinding.

## 5.1. Organisation der globalen Reparatur

Der Reparaturvorgang des vorgestellten Mehrkernsystems setzt sich aus zwei Phasen zusammen. Zunächst führt jeder Kern selbstständig die lokale Reparatur durch, die im vorherigen Kapitel vorgestellt wurde. Anschließend erfolgt der Übergang in die globale Reparaturphase, welche durch einen zur Laufzeit bestimmten Kern administriert wird. Abbildung 5.1 zeigt schematisch den Ablauf und die durchgeführten Aktionen während des Systemstarts. Im linken Bereich ist die autonom durchgeführte lokale Reparaturphase dargestellt, wie sie von jedem Kern durchlaufen wird. Jeder Kern führt zunächst einen Selbsttest durch und bestimmt seinen Defektzustand, der dann im Datenspeicher hinterlegt wird. Anschließend führt ein defekter Kern eine Rekonfiguration durch, wie sie im Kapitel zur lokalen Reparatur diskutiert wurde.

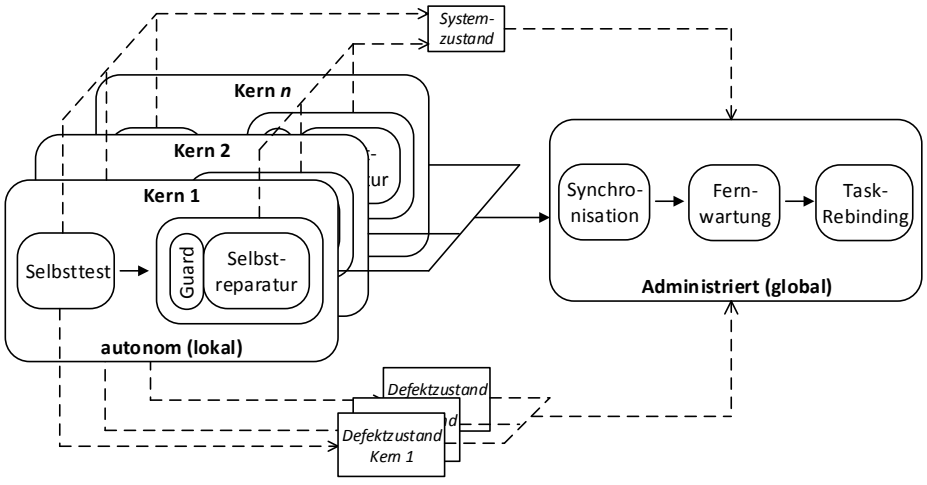


Abbildung 5.1.: Von den Kernen durchlaufene Teilphasen während des Systemstarts im Mehrkernsystem

Jeder Kern dokumentiert während der Ausführung der lokalen Phase des Systemstarts die durchlaufenen Teilphasen. Die Protokollierung erfolgt im Datenspeicher und wird für alle Kerne zusammengefasst als Systemzustand bezeichnet. Anhand des Systemzustands kann der Kern, der die globale Reparatur administriert, den Zustand der einzelnen Kerne bestimmen.

Der globale Teil der Reparaturstrategie ist in Abbildung 5.1 rechts dargestellt. Die globale Phase wird durch einen zur Laufzeit ermittelten Master-Kern gesteuert. Die Auswahl des Master-Kerns erfolgt während der Synchronisation der Kerne. Zusätzlich sorgt die Synchronisation dafür, dass sich alle Kerne in der gleichen Teilphase des Systemstarts befinden. Nach dem Abschluss der Synchronisation übernimmt der Master die Steuerung für den restlichen Teil des Systemstarts. Auf der Grundlage des Systemzustands und der Defektsituation auf den einzelnen Kernen entscheidet der Master über das weitere Vorgehen. Zunächst besteht die Möglichkeit, die Reparaturroutine eines defekten Kerns an die jeweilige Defektsituation anzupassen. Falls eine Fremdreparatur nicht erfolgreich ist oder eine lokale Reparatur fehlschlug, führt der Master-Kern eine Rekonfiguration der Task-Bindung im System durch.

Die beschriebene Reparaturstrategie hat Auswirkungen auf die lokale Reparaturphase und erfordert einige Erweiterungen, die bisher nicht berücksichtigt wurden. Die zusätzlich zu beachtenden Anforderungen sind:

- Zentrale Kenntnis über die Zustände- und Defektsituation der einzelnen Kerne,
- Ein Schutzmechanismus, damit ein unkontrolliert rechnender Kern keine Systemressourcen modifiziert und
- Eine Möglichkeit zur Synchronisation für den Übergang in die globale Phase.

Im nächsten Abschnitt wird die Bildung des Systemzustands besprochen. Im darauffolgenden Abschnitt wird dargelegt, wie die Absicherung gegen ein Überschreiben von Systemressourcen erfolgt. Dazu schützt sich einmal ein Kern selbst mit Hilfe eines *guards* und darüber hinaus muss der schreibende Zugriff auf Systemressourcen explizit freigeschaltet werden. Die Synchronisation und die Bestimmung des Kerns zur Administration sind Gegenstand des letzten Abschnitts.

### 5.1.1. Abbildung des Systemzustands

Damit der Master-Kern während der globalen Reparaturphase entscheiden kann, welche globalen Backends auszuführen sind, wird eine entsprechende Datenstruktur verwendet. Dadurch können die Probleme umgangen werden:

- dass von außen nicht erkennbar ist, welcher Teil der Test- und Reparaturroutinen von einem defekten Kern schon abgearbeitet wurde und
- ob die ausgeführten lokalen Reparaturmaßnahmen erfolgreich waren.

Während der lokalen Reparaturphase protokolliert jeder Kern die durchgeführten Schritte und gegebenenfalls, ob sie fehlschlugen. Die Protokollierung erfolgt für jeden Kern in getrennten Zustandsvariablen im gemeinsamen Datenspeicher. Zusammengefasst bilden

die einzelnen Variablen den **Systemzustand**. Tabelle 5.1 zeigt die zu unterscheidenden Zustände bezüglich eines Kerns und die jeweilige Bedeutung. Nach dem Starten des Systems hat die Variable den initialen Wert 0. Die nächsten beiden Zustände zeigen an, dass der Selbsttest gestartet bzw. beendet wurde. Der Zustand *guard passed* wird gesetzt, wenn ein Kern überprüft hat, dass sein Defektzustand die Ausführung der lokalen Selbstreparatur zulässt. Wenn die lokale Reparaturroutine ausgeführt wurde, kann dies mit zwei Zuständen enden. Zunächst kann die Reparatur erfolgreich sein, was bedeutet, dass die Routine in der Lage war, die Anwendung an die Defektsituation anzupassen. Der gleiche Zustand wird auch für den fehlerfreien Fall gesetzt. Im Fall, dass eine lokale Reparatur nicht erfolgreich war, wird der Zustand *self-repair failed* gesetzt.

| Zustand | Bedeutung |
|---|---|
| Startup | Initialer Wert |
| Self-Test | Selbsttest wurde gestartet |
| Self-Test Done | Selbsttest wurde beendet |
| Guard Passed | Guard konnte passiert werden |
| Guard Declined | Guard konnte nicht passiert werden |
| Self-Repair Done | Lokale Selbstreparatur erfolgreich abgeschlossen |
| Self-Repair Failed | Lokale Selbstreparatur schlug fehl |

Tabelle 5.1.: Mögliche lokale Zustände eines Kerns während des Systemstarts

Die Zustände eines jeden Kerns werden durch die globale Reparaturstrategie folgendermaßen ausgewertet. Der Fall *Self-Test Done* und *Guard declined* wird so interpretiert, dass auf einem Kern Defekte festgestellt wurden und diese eine Ausführung der Reparaturroutine unmöglich machen. In solch einem Fall kann die Fremdreparatur als globale Strategie zum Einsatz kommen. Im Fall, dass der Zustand *SelfRepair failed* gesetzt ist, wird als globale Reparaturstrategie das Task-Rebinding verwendet.

Abbildung 5.2 zeigt den Aufbau des Systemzustands im Datenspeicher. In diesem Beispiel ist jede Zustandsvariable 16 Bit breit und die untersten Bits repräsentieren jeweils einen Zustand aus Tabelle 5.1. Ist ein entsprechendes Bit auf den Wert 1 gesetzt, dann wurde der entsprechende Zustand erreicht.

### 5.1.2. Ressourcenschutz und der Guard

Eine Voraussetzung für die Reparatur eines defekten Kerns mit Hilfe einer globalen Reparaturstrategie ist, dass Systemressourcen vor Manipulation durch einen unkontrolliert rechnenden Kern geschützt werden. Dies kann passieren, wenn der lokale Reparaturalgorithmus unter Verwendung defekter Komponenten ausgeführt wird. Dabei kann der Programmspeicher geändert werden ohne eine Möglichkeit der Rekonstruktion. Um dieses Problem zu vermeiden, werden zwei Mechanismen in das System integriert:

1. In das Verbindungsnetzwerk wird eine Kontrollinstanz integriert, welche den Zugriff auf Ressourcen freischaltet.

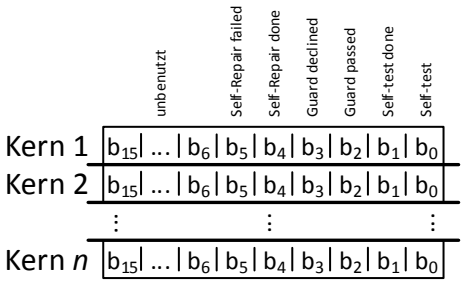


Abbildung 5.2.: Layout des Systemszustands im Datenspeicher

2. Jeder Kern prüft in einer Softwareroutine, als **guard** bezeichnet, ob sein Defektzustand die Ausführung seiner Reparaturalgorithmen zulässt.

Der Zugriff auf Systemressourcen ist initial nach dem Starten des Systems gesperrt. Die Ausnahmen sind der Defektspeicher und die Zustandsvariable eines jeden Kerns. Jeder Kern hat schreibenden Zugriff nur auf seinen eigenen Zustand. Der Speichercontroller überwacht die angelegten Adressen und ignoriert gegebenenfalls eine Schreibanweisung auf unzulässige Adressen. Um Zugriff auf weitere Ressourcen zu erlangen, muss ein Kern über zwei I/O-Befehle dem Verbindungsnetzwerk mitteilen, dass er bereit für einen erweiterten Ressourcenzugriff ist. Die Aufrufe der Befehle werden in die Abarbeitung des Guards integriert, um den Zugriff nur anzufordern, wenn der eigene Defektzustand es zulässt. Dabei ist zu beachten, dass die Befehle in der richtigen Reihenfolge ausgeführt werden, da ansonsten der Zugriff verwehrt wird. Weil sich defekte Komponenten auch auf die Code-Sequenz des Guards auswirken können, wird ein zweistufiges Verfahren verwendet, das auf disjunkten Prozessorbaugruppen ausgeführt wird und die Aktivierungsreihenfolge einhält.

Abbildung 5.3 zeigt den schematischen Aufbau der Guard-Sequenz. Die Aufteilung der Aktivierungssequenz in zwei Teile soll den Einfluss defekter Baugruppen auf die Codesequenz zur Freischaltung des Ressourcenzugriffs verringern. Im ungünstigsten Fall hat der *guard* keinen Effekt und der Schreibzugriff wird gewährt, obwohl der Defektzustand die Ausführung der lokalen Reparaturalgorithmen nicht zulässt. Das zweistufige Verfahren verringert Einflüsse durch Defekte in der Sprungeinheit, der Speicheranbindung, den Slots und den Registern. Die Sprungsequenzen sind so gewählt, dass bei einer fehlerhaften Sprungeinheit die Aktivierungssequenz in der falschen Reihenfolge oder nicht vollständig ausgeführt wird. Für den Fall, dass gar nicht gesprungen wird, wird zuerst der zweite Teil ausgeführt. Sollte die Sprungeinheit immer springen, wird die Sequenz vorzeitig verlassen. Wenn im Fehlerfall zu den Adressen ein Offset addiert wird, kann der Aktivierungsbefehl nicht genau angesprungen werden.

Für die Prüfsequenzen wird die natürliche Hardwareredundanz des VLIW-Konzepts ausgenutzt. Der Code der einzelnen Teilsequenzen wird auf verschiedenen Slots ausge-

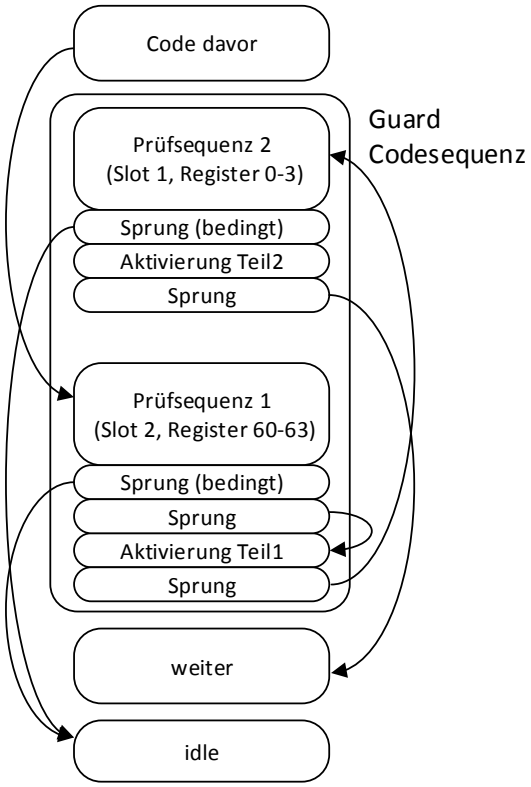


Abbildung 5.3.: Steuerfluss innerhalb der Befehlssequenz des Guard-Codes

führt und verwendet unterschiedliche Register. Die Prüfsequenz verwendet ein Minimum an Befehlen für die Berechnungen. Die folgende Assemblersequenz wird im ersten Slot ausgeführt und verwendet Register $r_{16}$, um den Defektzustand der Register $r_0$ bis $r_{15}$ zu überprüfen. Der Zustand der Register ist in einem 16-Bit-Datenwort invertiert gespeichert. Ist ein Register $r_i$ defekt, dann ist an der i-ten Bitposition in dem Datenwort eine 1 gesetzt. Mit einem bedingten Sprung wird, falls ein Bit auf 1 gesetzt ist, die Ausführung der Aktivierungssequenz vermieden.

```
ldc #regFaultState0to15 -> r16;
nop;
nop;
load [r16] -> r16;
nop;
nop;
jnz r16, #idle;
```

Die zweite Prüfsequenz führt den Code auf einem anderen Slot mit einem anderem Register durch. Zur Überprüfung der Speicheranbindung werden von vorinitialisierten

Speicheradressen einmal der Wert 0 und ein Datum, in dem jedes Bit auf 1 gesetzt ist, gelesen und ausgewertet.

Durch bestimmte Defektkonstellationen kann der Mechanismus des *guards* ausgehebelt werden und es ist nicht sicher gestellt, dass ungewollte Änderungen an Systemressourcen erfolgen. In diesem Fall wird das gesamte System als Defekt betrachtet. In die Ergebnisse (Kapitel 6.3) zu den tolerierbaren Fehlern und der Lebensdauer fließen diese Kombinationen mit ein.

### 5.1.3. Eintritt in die globale Phase

Nachdem die lokale Reparaturphase abgeschlossen ist, erfolgt der Übergang in die globale Phase. Die globale Phase wird durch einen zur Laufzeit gewählten Kern administriert. Dies geschieht während der Synchronisationphase zu Beginn der globalen Reparatur. Diese Teilphase wird als Barriere aufgefasst, auf welcher die einzelnen lokalen Steuerflüsse auflaufen. Ziel dieser Phase ist es, den Kern zur Administration der globalen Reparatur zu bestimmen. Anschließend wird zentral gesteuert die globale Phase durchlaufen.

Die grundlegende Idee zur Bestimmung des Masterkerns ist, dass der erste Kern, welcher erfolgreich seine lokale Reparaturphase abschließt, sich selbst als Masterkern für die globale Phase deklariert. Die Deklaration geschieht durch setzen der eigenen Kernnummer in einer globalen Variablen *master*. Der Schreibzugriff auf die Adresse der Variablen ist erst nach dem erfolgreichen Abschluss der Aktivierungssequenz möglich. Bezüglich der Variablen ist ein gegenseitiger Ausschluss erforderlich, damit aufgrund von asynchronen Zugriffen keine Unstimmigkeit bezüglichen des Kerns zur Administration entsteht.

Um keine zusätzlichen Hardwarestrukturen zur Synchronisation in das System integrieren zu müssen, wird eine Strategie zur freiwilligen Abgabe gewählt. Da es grundsätzlich egal ist, welcher Kern die globale Reparatur administriert, wird das selbst gewährte Recht unter Umständen freiwillig abgegeben, damit weiterhin Einigkeit bestehen kann. Bevor ein Kern $k$ sich selbst als Masterkern in die Zustandsvariable einträgt, überprüft er, ob nicht schon ein anderer Kern die Variable gesetzt hat. In dem Fall würde $k$ in einen Wartezustand übergehen. Andernfalls trägt $k$ seine eigene Kernnummer in die Variable *master* ein. Nach einer kurzen Pause überprüft $k$, ob in *master* immer noch die eigene Kernnummer gesetzt ist. Ist dies der Fall, dann administriert $k$ die globale Reparatur. Andernfalls gab es einen überlappenden Zugriff auf *master* und $k$ betrachtet sich nicht weiter als Masterkern und geht in einen Wartezustand über.

## 5.2. Fremdreparatur

Die Fremdreparatur behandelt den Fall, dass ein Kern seine eigenen Reparaturalgorithmen nicht ausführen kann, weil der Programmcode der Reparaturalgorithmen die Verwendung einer fehlerhaften Prozessorbaugruppe vorsieht. Die Strategie sieht nun vor, dass durch einen anderen Kern des Systems, konkret durch den zuvor bestimmten Masterkern, lediglich der Programmcode der Reparaturalgorithmen an die Fehlersituation angepasst wird. Die Anpassung der Reparaturalgorithmen greift ausschließlich auf

redundante Hardware zurück und verwendet keine Verschiebung der Operation in der zeitlichen Dimension.

Um der Fremdreparatur mehr Freiheitsgrade zur Anpassung des Systemscodes zu bieten, wird die parallele Ressourcennutzung durch den Code der Reparatur beschränkt. Dies wird erreicht, indem durch den Compiler während der Übersetzungszeit des Systems keine Operationen parallel geplant werden. Erfolgt die Entwicklung der Reparaturalgorithmen in Assembler, dann ist der Entwickler für eine entsprechende Ressourcennutzung verantwortlich.

Damit eine Anpassung der Reparaturroutinen erfolgen kann, ist eine genaue Kenntnis über die Ressourcennutzung der Reparaturroutinen erforderlich. Dazu wird vereinbart, dass die Ursprungsversion immer den ersten Slot eines Kerns verwendet und darüber hinaus einen festgelegten Satz an Registern, der mit dem Register $r_0$ beginnt. Der Satz an Registern, der einen Teilsatz der gesamten Register $(r_0, ..., r_{n-1})$ darstellt, ist durch die erforderliche Anzahl $k$ an Registern, die für den Programmcode der Reparatur maximal allokiert sind, gegeben. Typischerweise ist $k$ deutlich kleiner als $n$ und der gesamte Registersatz kann in Gruppen, die $k$-viele Register enthalten, unterteilt werden. Die einzelnen Gruppen werden durch die Fremdreparatur als redundanter Registersatz verwendet.

Abbildung 5.4 zeigt schematisch, wie sich die Ressourcennutzung für die ursprüngliche Übersetzung der Reparaturroutinen darstellt. Der Code ist in der Form übersetzt, dass er Operationen im ersten Slot ausführt und die Register des ersten Teilsatzes $(r_0, ..., r_{15})$ verwendet.

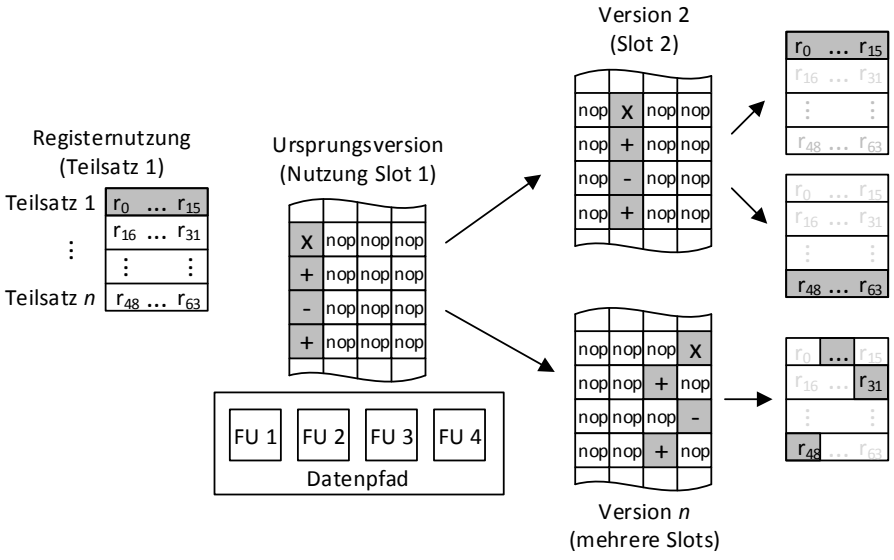


Abbildung 5.4.: Schematische Darstellung der Ressourcennutzung der Reparaturroutinen vor und nach einer Anpassung durch den Masterkern

In Abhängigkeit vom Defektzustand verschiebt der Masterkern die Operationen vom ursprünglichen in einen redundanten fehlerfreien Slot oder er verwendet das Rebinding, um den Code anzupassen. In Abbildung 5.4 zeigt Version 2 eine Anpassung durch Verschieben der Operationen in den zweiten Slot. Darunter ist eine Anpassung auf mehrere Slots für das Rebinding dargestellt. Die Entscheidung, ob ein einfaches Verschieben oder ein Rebinding erfolgt, wird vor der Codeanpassung getroffen. Steht ein fehlerfreier Slot zur Verfügung, der alle erforderlichen Operationstypen unterstützt, ist ein einfaches Verschieben zwischen den Slots ausreichend.

In Abbildung 5.4 ist weiterhin eine Anpassung der Registernutzung dargestellt. Zur Rekonfiguration der Registervergabe wird das Renaming verwendet. Die Wahl eines Ersatzregisters $k$ für ein defektes Register $r$ erfolgt anhand der redundanten Registersätze. Im Beispiel aus Abbildung 5.4 können für $r_0$ 3 Reserveregister verwendet werden, die Register $r_{16}$,$r_{32}$ und $r_{48}$, wobei diese aus drei redundanten Registersätzen stammen.

Nachdem eine Reparatur durchgeführt wurde, müssen die folgenden zwei organisatorischen Probleme behandelt werden:

1. Umgehung des Guards und korrektes Setzen des Zustands und

2. die Reaktivierung eines fehlerhaften Kerns aus dem Wartezyklus zur Ausführung der eigenen Reparaturroutinen.

Das Problem ist, das auch nach einer Anpassung des Systemscodes der fehlerhafte Kern den Guard nicht durchlaufen würde, da der Defektzustand weiterhin der gleiche ist. Zur Umgehung des Problems setzt der Masterkern den Systemzustand des defekten Kerns auf den richtigen Wert, aktiviert den Schreibzugriff für den defekten Kern und manipuliert die Codeausführung auf dem defekten Kern in der Form, dass dieser direkt mit der Ausführung der Reparaturroutinen fortfährt.

Abbildung 5.5 zeigt, wie die Reaktivierung aus dem Wartezyklus erfolgt. Der Wartezyklus ist eine Schleife, in welcher der defekte Kern ein aktives Warten durchführt. Durch eine Manipulation der Sprungadresse kann indirekt der Steuerfluss auf dem defekten Kern beeinflusst werden. Als Sprungziel wird in diesem Fall der Einsprung in die Reparaturroutinen gewählt, nachdem vom Master-Kern der gültige Systemzustand hergestellt wurde.

## 5.3. Task Rebinding

Das Task-Rebinding führt für den Fall, dass eine lokale Rekonfiguration bestehende Ressourcenbeschränkungen verletzt, ein Vertauschen der Anwendungen durch mit dem Ziel, die jeweilige Anwendung erfolgreich an die Architektur des neu zugewiesen Kerns anzupassen. Mit dieser Vorgehensweise entsteht die Möglichkeit, die lokal vorhandene Hardware-Redundanz kernübergreifend zu verwenden. Allerdings wird das Task-Rebinding für eine auf die Anwendung optimierte Systemkonfiguration, in der jeder Kern minimale Ressourcen besitzt, für den allgemeinen Fall nicht erfolgreich einsetzbar sein. Deshalb muss zusätzliche Hardware-Redundanz in das System eingebracht werden, die durch die kernübergreifende Reparatur global administriert wird und sich dadurch

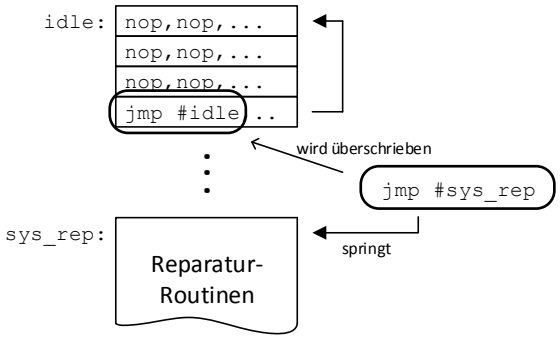


Abbildung 5.5.: Re-aktivieren eines fehlerhaften Kerns aus dem Wartezyklus

optimiert verteilen lässt. Eine grundsätzliche Idee, wie sich Spare-Komponenten günstig in ein Mehrkernsystem integrieren lassen und welche Anforderungen dies beeinflussen, wird im nächsten Abschnitt diskutiert.

Der Vorgang zum eigentlichen Task-Rebinding wird in Abbildung 5.6 schematisch dargestellt. Es ist ein Mehrkernsystem mit 3 Kernen und unterschiedlichen Datenpfadkonfigurationen dargestellt. Während der globalen Reparaturphase prüft zunächst der Masterkern, ob ein anderer Kern seine lokale Reparatur nicht erfolgreich beenden konnte. Ist dies der Fall, dann ermittelt der Masterkern eine neue Task-Zuordnung für das System, basierend auf den Defektzuständen und den Anforderungen der Anwendungen. Ist ein geeigneter Kandidat als Ersatz ermittelt, kann das Vertauschen der Anwendungen vorgenommen werden. In Abbildung 5.6 konnte Kern 2 die zugeordnete Anwendung $t2$ nicht an den Fehler in der dritten FU anpassen. In diesem Szenario stellt der Kern 1 noch genügend Redundanz bereit, damit $t2$ auf ihn angepasst werden kann. Dazu müssen nun die Anwendungen $t1$ und $t2$ aus den jeweiligen Programmspeichern vertauscht und an die Architektur des jeweils neu zugeordneten Kerns angepasst werden.

Zur eigentlichen Durchführung des Task-Rebinding ergeben sich die folgenden zu beachtenden Schwerpunkte:

- Die Berechnung einer neuen Zuordnung zwischen den Kernen des Systems und den Anwendungen, basierend auf Kennzahlen zur Bewertung der Leistungsanforderungen,
- Eine Zustandssicherung einer in Teilen umgeplanten Anwendung, deren Rekonfiguration fehlschlug, und
- Eine geschickte Organisation des Programmspeichervertauschens, mit der die Ressourcenbeschränkungen eingehalten werden.

Im nächsten Abschnitt erfolgt eine Diskussion dazu, wie ein fehlertolerantes System mit minimalen Spare-Komponenten gebildet werden kann. Der sich anschließende Abschnitt führt die zuvor erwähnten Kennzahlen zur Bewertung der Leistungsanforderung einer Anwendung ein, anhand derer die Berechnungen für das Task-Rebinding erfolgen.

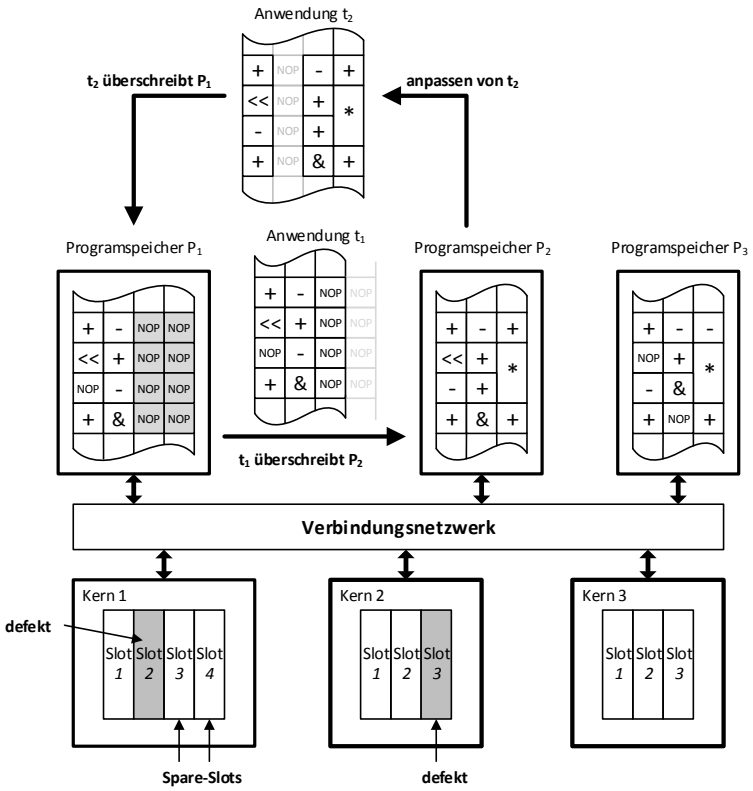


Abbildung 5.6.: Schematische Darstellung des Taks-Rebindings für ein System mit 3 Kernen, einem Defekt im zweiten Kern und zwei Spare-Slots im ersten Kern

Der letzte Abschnitt beschreibt das technische Vorgehen zum Task-Rebinding. Es werden Lösungen für die Zustandssicherung einer Anwendung und für das Vertauschen zweier Programmspeicherinhalte präsentiert.

### 5.3.1. Bestimmung einer fehlertoleranten Systemkonfiguration

In einem Mehrkernsystem mit minimalen Hardware-Ressourcen sind für eine erfolgreiche Rekonfiguration mittels Task-Rebinding im Allgemeinen nicht ausreichend redundante Baugruppen vorhanden. Typischerweise wird ein Kern mit redundanten Komponenten erweitert, damit sicher eine gewisse Anzahl an Fehlern behandelt werden kann. Wenn nur eine lokale Rekonfiguration durchgeführt werden kann, muss jeder Kern mit Reserve-Komponenten erweitert werden. Durch den Einsatz des Task-Rebindings ist es im Idealfall ausreichend, nur einen Kern zu erweitern und die systemweite Anzahl an Reservekomponenten kann reduziert werden.

Das Erweitern eines Systems mit zusätzlichen Prozessorbaugruppen erfolgt auf der Basis einer minimalen Grundkonfiguration eines Mehrkernsystems. Die Granularität,

mit der die Kerne redundant ausgelegt werden, ist dabei auf die Ausführungsslot beschränkt. Die Architektur eines Systems wird als **Systemkonfiguration** beschrieben, welche die Anzahl der verfügbaren Slots eines jeden Kerns angibt. Die Darstellung erfolgt als Vektor der Form $(c_1, ..., c_n)$, wobei eine Komponente in der $i$-ten Position die Anzahl der Slots des Kerns $i$ wiedergibt. Zum Beispiel hat das System mit der Konfiguration $(2, 3, 4)$ drei Kerne, wobei Kern 1 zwei Slots aufweist, Kern 2 drei Slots und Kern 3 vier Slots. Die minimale Konfiguration eines Systems ist typischerweise das Ergebnis einer optimierten Entwurfsexploration [82], die keine Redundanz aufweist, weil die ausgeführten Anwendungen auch auf den jeweiligen Datenpfad optimiert sind.

Ein System, dessen Konfiguration mit zusätzlicher Hardware-Redundanz erweitert wird, wird als **fehlertolerantes System** bezeichnet. Die Konfiguration wird entsprechend als **fehlertolerante Konfiguration** benannt. In Abhängigkeit davon, wie viele Fehler auf der jeweiligen Systemebene sicher und gegebenenfalls auch ohne Leistungsverlust behandelbar sein sollen, müssen entsprechend viele redundante Baugruppen integriert werden. Dabei ist zu beachten, dass ein Kern, der mit weiteren Slots ausgestattet wird, weiterhin die gleiche Anwendung ausführt. Die Instruktionen der Anwendung werden demzufolge mit Nop-Operationen erweitert. Für die zuvor genannte minimale Konfiguration $(2, 3, 4)$ ist $(4, 3, 4)$ eine fehlertolerante Konfiguration, bei welcher der erste Kern um zwei Slots erweitert wurde. Mit Hilfe des Task-Rebindings können diese beiden zusätzlichen Slots nun kernübergreifend verwendet werden, um Fehler in den Kernen 2 bzw. 3 zu behandeln.

Die grundsätzliche Idee bei der Bestimmung einer fehlertoleranten Konfiguration ist es:

- den Kern mit der geringsten Slot-Anzahl auf das Niveau des Kerns mit der höchsten Parallelität anzuheben.

Der so erweiterte Kern führt weiterhin seine ursprünglich zugewiesene Anwendung aus, verfügt aber nun über zusätzliche Slots, die nicht genutzt werden. Somit fungiert dieser Kern als **Spare-Kern** für das System und kann im Fehlerfall die Ausführung einer Anwendung mit mehr Ressourcenbedarf übernehmen. Ein weiterer Einflussfaktor auf die Festlegung einer fehlertoleranten Konfiguration ist die Anzahl an tolerierbaren Fehlern, die gleichzeitig im System vorhanden sein dürfen. Mit einer steigenden Anzahl an Fehlern kann es erforderlich sein, mehrere oder auch alle Kerne mit redundanten Baugruppen auszustatten.

Tabelle 5.2 zeigt dazu, wie sich die fehlertoleranten Konfigurationen entwickeln, wenn mehr Fehler tolerierbar sein sollen. Zusätzlich werden die Konfigurationen für zwei verschiedene Reparaturstrategien angegeben. In der ersten Strategie kann lediglich eine lokale Reparatur erfolgen, wohingegen in der zweiten, der globalen Reparaturstrategie, zusätzlich das Task-Rebinding eingesetzt wird. Soll ein Fehler tolerierbar sein, muss in dem System mit der lokalen Reparatur jeder Kern um einen Slot erweitert werden und analog dazu für zwei Fehler jeweils um 2 Slots. In der Konfiguration für die globale Strategie ist zur Behandlung eines Fehlers nur der erste Kern um 2 Slots erweitert. Zur Behandlung von zwei Fehlern müssen zusätzlich die Kerne 2 und 3 um einen Slot erweitert werden, da sich die Fehler auf zwei Kerne verteilen können. Vergleicht man die beiden Strategien, ist zu erkennen, dass die globale Strategie weniger

Reserve-Komponenten erfordert und somit ein besseres Verhältnis zwischen tolerierbaren Fehlern und Hardware-Mehraufwand bietet.

| Minimale Konfiguration | Fehler je Kern | Fehler im System | nur lokale Rekonfig. | zusäztl. glob. Rekonfig. | zusäztl. Slots lok bzw. glob |
|---|---|---|---|---|---|
| (3,3,4) | 0-1 | 0-1 | $(4,4,5)$ | $(4,3,4)$ | 3 - 1 |
| (2,3,3,4) | 0-1 | 0-1 | $(3,4,4,5)$ | $(4,3,3,4)$ | 4 - 2 |
| (2,3,3,4) | 0-2 | 0-2 | $(4,5,5,6)$ | $(4,4,4,4)$ | 8 - 4 |
| (3,3,4,4,5) | 0-2 | 0-2 | $(5,5,6,6,7)$ | $(6,5,5,5,5)$ | 10 - 7 |

Tabelle 5.2.: Fehlertolerante Konfiguration auf Basis unterschiedlicher minimaler Systemkonfiguration, tolerierbarer Fehlern und angewendeter Reparaturstrategie

### 5.3.2. Bestimmung einer neuen statischen Bindung

Um eine neue statische Bindung zwischen den Anwendungen und den Kernen zur Reparaturzeit berechnen zu können, werden zwei Kennzahlen eingeführt. Die Kennzahlen bewerten die Leistungsanforderung einer Anwendung und die verfügbare Rechenleistung eines Kerns. Es handelt sich um einfache Bewertungskriterien, weil die Kennzahl für die Rechenleistung eines Kerns zur Reparaturzeit unter Berücksichtigung des Defektzustands ermittelt wird. Anhand der Kennzahlen bestimmt der Masterkern eine passende Verteilung der Anwendung auf die verfügbaren Kerne. Der Leistungsindex eines Kerns $c$ errechnet sich folgendermaßen:

$$perf(c) = slots_c - \frac{1}{opTypes_c} * faultyOps_c$$

Die Formel ermittelt ausgehend vom fehlerfreien Zustand eines Kern den Leistungsindex, indem einmal die defekten Slots und zu einem geringeren Anteil die ausgefallenen Operatoren gewertet werden. Ist ein Kern $c$ fehlerfrei so ist der Wert $perf(c)$ gerade die Parallelität im Datenpfad bestimmt durch die Anzahl an vorhandenen Slots $slots_c$. Die drei Parameter der Formel ergeben sich wie folgt:

- Der Wert $slots_c$ berechnet sich aus der ursprüngliche Anzahl an vorhandenen Slots verringert um den den Wert der ausgefallenen Slots. Ein ausgefallener Slot ist ein Slot dem das Planungsverfahren keine Operationen (außer der Nop-Operation) zur Berechnung zuordnen kann. Da ausgefallene Slots im Wert $slots_c$ gewichtet werden, ist es wichtig die Operatorkonfiguration und den Operatorzustand ausgefallener Slots nicht weiter in den Werten von $opTypes_c$ und $faultyOps_c$ zu berücksichtigen.

- $opTypes_c$ ergibt sich aus der Anzahl an echt verschiedenen Operationstypen, die von den nicht ausgefallenen Slots in ihrer fehlerfreien Ausgangskonfiguration bereitgestellt wurden. Stellen zum Beispiel mehrere Slots einen Operator des Typs

Addition bereit, so wird dieser nur einmal in $opTypes_c$ gewertet. Sind beispielsweise alle Operatoren der Multiplikation in den nicht ausgefallenen Slots defekt, so wird die Multiplikation trotzdem einmal in $opTypes_c$ gezählt, da dieser Operationstyp in der fehlerfreien Konfiguration unterstützt wurde.

- In $faultyOps_c$ werden alle defekten Operatoren der nicht ausgefallenen Slots gezählt. Die defekten Operatoren eines ausgefallenen Slots $s$ werden nicht in $faultyOps_c$ berücksichtigt, da der Ausfall von $s$ bereits in $slots_c$ gewichtet wurde.

Der Leistungsindex aller Kerne ergibt zusammengefasst die **Systemleistung**, die in einem Vektor $v = (perf(c_1), ..., perf(c_n))$ dargestellt wird. Sind alle Slots defekt, und dadurch kein Operationstyp unterstützt, ist der Leistungsindex 0 und kann nicht über die Formel berechnet werden. Ist ein bestimmter Operationstyp für die Planung einer Task erforderlich der nicht durch die Architektur unterstützt wird, so kann dies erst durch den Planungsalgorithmus festgestellt werden.

Tabelle 5.3 zeigt Beispiele für Systeme mit unterschiedlichen Datenpfadkonfigurationen, die entsprechende Systemleistung im fehlerfreien Fall und die Systemleistung bei einem Komponentenausfall. Die Konfigurationen in der ersten Spalte sind so zu lesen, dass ein Klammerpaar und die enthaltenen Symbole die Operatorkonfiguration für einen Slot angeben. Für jeden vorhandenen Slot in einem Kern ist ein Klammerpaar angegeben. In der Konfigurationen im Fehlerfall repräsentiert ein leeres Klammerpaar einen Ausfall des entsprechenden Slots.

| fehlerfreie Datenpfad-konfiguration | resultierender Leistungsindex | Konfiguration mit permanenten Fehler | geänderter Leistungsindex |
|---|---|---|---|
| ((+,-,*),(+,-)) | 2 | ((-,*),(+)) | 1,33 |
| ((+,-),(+,-)) | 2 | ((+),(+)) | 1 |
| ((+,-),(+,-,*),(+)) | 3 | ((+),(+,-),( )) | 1,33 |
| ((+,-),(+),(+,-),(+,-)) | 4 | ((+,-),(+),( ),(+,-)) | 3 |

Tabelle 5.3.: Veänderung der Leistungsindizies für Kerne mit 2,3 und 4 Slots und ausgefallenen Operatoren bzw. Slots

Der Kern der ersten Zeile besitzt zwei Slots und unterstützt insgesamt 3 unterschiedliche Operationstypen. Der ursprüngliche Leistungsindex beträgt 2. Der beispielhafte Fehlerfall zeigt eine geänderte Konfiguration, bei der zwei Operatoren nicht nutzbar sind und für die sich der Leistungsindex auf 1,33 reduziert hat. Im zweiten Beispiel sind alle ursprünglichen Operatoren der Subtraktion defekt. Der Wert von $opTypes_c$ entspricht 2 sowie der von $faultyOps_c$. Als Leistungsindex ergibt sich für das zweite Beispiel der Wert 1. Das dritte Beispiel hat einen Leistungsindex im fehlerfreien Fall von 3. In der Defekt-Konfiguration entfällt der dritte Slot, der Subtraktions-Operator des ersten Slots und die Multiplikation des zweiten Slots. Der errechnete Leistungsindex ergibt 1,33 da $slots_c$ den Wert 2 hat und weiterhin gilt $opTypes_c = 3$ und $faultyOps_c = 2$. Die Multiplikation wird weiterhin in $opTypes_c$ gewertet, da der zweite Slot weiterhin dem Planungsalgorithmus zur Verfügung steht. Wäre der zweite Slot vollständig defekt,

dann würde die Multiplikation nicht mehr berücksichtigt werden. Das Beispielsystem der letzten Zeile besitzt vier Slots und unterstützt zwei Operationstypen. Der sich daraus ergebende Leistungsindex beträgt 4. Die fehlerhafte Konfiguration beschreibt einen Ausfall des dritten Slots mit einem resultierenden Leistungsindex von 3.

Die zweite Kennzahl, der Leistungsindex einer Anwendung, bewertet die Anforderung einer Anwendung bezüglich der Hardware, auf der sie ausgeführt wird. Der Leistungsindex einer Anwendung wird zur Übersetzungszeit des Systems bestimmt und in den Metadaten dem Reparaturverfahren zur Verfügung gestellt. Der Leistungsindex basiert auf der durchschnittlichen Anzahl parallel ausgeführter Operationen in den zeitkritischen Basisblöcken einer Anwendung. Durch Aufrunden des so ermittelten Wertes auf die nächst größere natürliche Zahl ergibt sich die minimale Anzahl an parallelen Slots, die für die Ausführung der Anwendung im vorgegebenen zeitlichen Rahmen erforderlich sind. Für das gesamte System werden die Leistungskennzahlen der einzelnen Tasks durch den Vektor $task\ vector = (perf_1, ..., perf_n)$ angegeben. Eine Komponente $perf_i$ gibt die Leistungskennzahl der Task $t_i$ an, die in Bezug auf das Originalsystem dem Kern $c_i$ zugeordnet ist.

In der Tabelle 5.4 sind verschiedene Systemkonfigurationen angegeben mit einer Gegenüberstellung der Leistungskennzahlen für die Kerne und den Tasks sowie eine fehlertolerante Systemkonfiguration. Die fehlertolerante Systemkonfiguration für die Systeme der ersten beiden Zeilen soll einen beliebigen Fehler innerhalb des Systems tolerieren können. Für das System der dritten Zeile ist die fehlertolerante Konfiguration so gewählt, dass systemweit zwei Fehler tolerierbar sind. Das System in der ersten Zeile besteht aus zwei Kernen. Der erste Kern besitzt zwei Slots und der zweite Kern drei. Der Vektor zur Systemleistung ist mit $v = ((2), (3))$ angegeben. Die zugewiesenen Tasks haben einen Leistungsindex von 1,8 bzw. 2,2. Für die fehlertolerante Konfiguration ist der erste Kern erweitert und der Vektor der Systemleistung ändert sich zu $v = ((3), (3))$. Für das System in der zweiten Zeile wird in der fehlertoleranten Konfiguration auch der erste Kern erweitert mit einer Steigerung der Leistungskennzahl von 2 auf 4, um im Fehlerfall die dritte Task ausführen zu können. Im System der dritten Zeile müssen durch die höhere Anzahl an tolerierbaren Fehlern alle Kerne um mindestens einen Slot erweitert werden. Darüber hinaus wird der erste Kern um zwei Slots erweitert, um im Fehlerfall die dritte Task ausführen zu können, falls auf dem dritten Kern zwei Slots ausfallen.

| Systemkonfiguration und Systemleistung | Task-Vektor | Systemleistung des fehlertoleranten Systems |
|---|---|---|
| ((2),(3)) | ((1,8),(2,2)) | ((3),(3)) |
| ((2),(2),(4)) | ((1,2),(1,9),(2,7)) | ((4),(3),(4)) |
| ((3),(4),(5)) | ((2,7),(3,9),(4,8)) | ((5),(5),(6)) |

Tabelle 5.4.: Gegenüberstellung verschiedener Systemkonfigurationen mit den Leistungsvektoren der Kerne, der Anwendungen und einer beispielhaften fehlertoleranten Konfiguration

Der nachfolgende Algorithmus 5.1 bestimmt anhand des Systemzustands, dem Vektor der Systemleistung und dem Vektor der Leistungsanforderungen der Anwendungen eine neue statische Bindung der Anwendungen im System. Ziel ist es, für jede Anwendung, deren lokale Reparatur fehlschlug, einen Kern mit passenden Leistungsindex zuzuordnen. Die Bestimmung einer neuen Task-Bindung startet mit der Anwendung mit dem höchsten Leistungswert, deren Reparatur nicht ausführbar war.

**Algorithm 5.1** Bestimme Task-Rebinding

1: **Eingabe:***core vector* $(perf(c_1), ..., perf(c_n))$, *task vector* $(perf_1, ..., perf_n)$
2: **Ausgabe:***task vector* $v'$
3:
4: **repeat**
5: - bestimme die nächste Anwendung $t_i$ mit dem höchsten Leistungswert $perf_i$, deren lokale Reparatur fehlschlug
6: - finde einen Kern $c_j$ mit niedrigsten Wert $perf(c_j)$ passend zu $t_i$ und dem passenden Leistungsindex $perf_j$ zu $c_i$
7: **if** solch ein Kern konnte nicht bestimmt werden **then**
8: - globale Reparatur nicht erfolgreich
9: **end if**
10: - führe Task-Rebinding bezüglich der Kerne $c_i$ und $c_j$ durch
11: **until** Alle Anwendungen mit fehlgeschlagener lokaler Reparatur sind zugeordnet

Algorithmus 5.1 versucht in der äußeren Schleife alle Anwendungen, die nicht erfolgreich umgeplant werden konnten, einem neuen Kern zuzuordnen. In jedem Durchlauf wird die Anwendung mit dem höchsten Leistungsindex behandelt (Zeile 5). Für eine Anwendung $t$ wird ein Kern $c$ mit einem Leistungsindex bestimmt, der am wenigsten von der Anforderung von $t$ abweicht (Zeile 6). Darüber hinaus muss beachtet werden, dass der Kern, an den $t$ aktuell gebunden ist, einen passenden Leistungsindex für die Anwendung bereitstellt, die $c$ zugewiesen ist. Kann eine solche Neubindung nicht bestimmt werden, ist die gesamte Reparatur fehlgeschlagen. Andernfalls kann das Vertauschen der Anwendungen im Programmspeicher mit gleichzeitiger Anpassung an die Hardware der Kerne erfolgen.

Die folgende Tabelle 5.5 zeigt beispielhaft, wie sich der Task-Vektor verändert, wenn das Task-Rebinding angewendet wird. Die Beispiele basieren auf den fehlertoleranten Systemkonfigurationen aus Tabelle 5.4.

| fehler-tolerant. System | alter Task-Vektor | def. Kerne | neuer System-Vektor | neuer Task-vektor |
|---|---|---|---|---|
| ((3),(3)) | ((1,8),(2,2)) | 2 | ((3),(2)) | ((2,2),(1,8)) |
| ((4),(3),(4)) | ((1,2),(1,9),(2,7)) | 3 | ((4),(3),(3)) | ((2,7),(1,9),(1,2)) |
| ((5),(5),(6)) | ((2,7),(3,5),(4,8)) | 1,2 | ((4),(4),(6)) | ((3,5),(2,7),(4,8)) |

Tabelle 5.5.: Veränderung des Task-Vektors, nachdem das Task-Rebinding aufgrund einer fehlgeschlagenen lokalen Reparatur angewandt wurde

In der ersten Zeile wird von einem ausgefallenen Slot im zweiten Kern ausgegangen und der Systemvektor ändert sich von $v = ((3), (3))$ zu $((3), (2))$. Es werden die beiden Tasks vertauscht und der resultierende Task-Vektor ist $((2, 2), (1, 8))$. In der dritten Zeile wird von zwei ausgefallenen Komponenten ausgegangen. Im ersten und zweiten Kern sind jeweils ein Slot ausgefallen. Deshalb werden die Tasks der betreffenden Kerne ausgetauscht. Trotz des Fehlers im ersten Kern kann dieser die zweite Task ausführen, da in der fehlertoleranten Konfiguration der erste Kern entsprechend dem dritten Kern dimensioniert ist.

### 5.3.3. Organisation des Programmspeichertauschens

Nachdem der Master eine neue Zuordnung zwischen den Tasks und den Kernen bestimmt hat, erfolgt das eigentliche Vertauschen der Anwendungen und eine Anpassung der jeweiligen Task an die Gegebenheiten des neu zugewiesenen Kerns. Es kann erforderlich sein, dass mehrere Tasks nacheinander zu vertauschen sind. Die endgültige Zuordnung aller Tasks an einen geeigneten Kern stellt sich dann erst nach mehreren Durchläufen von Algorithmus 5.1 ein.

Das folgende Szenario verdeutlicht diesen Vorgang. Für ein System mit vier Kernen wird die fehlertolerante Datenpfadkonfiguration $(4, 3, 3, 4)$ betrachtet, wobei der erste Kern als Spare-Kern erweitert ist. Der Task-Vektor ist mit $((1, 7), (2, 2), (2, 8), (3, 8))$ gegeben und die Systemleistung durch $((4), (3), (3), (4))$. Durch Defekte im dritten und vierten Kern ändert sich die Systemleistung zu $((4), (3), (2), (3))$. Algorithmus 5.1 identifiziert im ersten Durchlauf Task 4 als Kandidat zum Vertauschen. Task 4 wird dem ersten Kern zugeordnet und der Task-Vektor verändert sich zu $((3, 8), (2, 2), (2, 8), (1, 7))$. In einem weiteren Durchlauf wird Task 3 als nächster Kandidat bestimmt und an den vierten Kern gebunden. Die endgültige Systemkonfiguration ist mit dem Task-Vektor $((3, 8), (2, 2), (1, 7), (2, 8))$ gefunden. In diesem Szenario wurde Task 1 mehrmals verschoben, bis schließlich Kern 4 als endgültige Zuordnung gefunden wurde.

Um eine Rekonfiguration einer Task im optimalen Fall nur für den endgültig zugewiesenen Kern durchzuführen, erfolgt das Vertauschen der Programmspeicherinhalte und somit der Tasks in einem überlappendem Verfahren aus Rekonfiguration und Verschieben von Speicherinhalten. Zusätzlich umgeht dieses Vorgehen einmal die ohnehin vorhandenen Ressourcenbeschränkungen und darüber hinaus das Problem möglicher unterschiedlich breiter Datenpfade und dadurch verschiedener Instruktionswort- und Programmspeicherbreiten.

Ein weiterer Schwerpunkt, der noch vor dem Vertauschen der Programmspeicherinhalte zu beachten ist, ist die Zustandssicherung einer in Teilen umgeplanten Anwendung, wenn eine Rekonfiguration nicht erfolgreich zu Ende durchgeführt werden kann. Die Zustandssicherung wird im folgenden Abschnitt besprochen. Die technische Organisation zum Vertauschen der Tasks wird im darauffolgenden Abschnitt präsentiert.

#### Zustandssicherung einer fehlgeschlagenen lokalen Reparatur

Die Rekonstruktion des Ausgangszustands einer in Teilen umgeplanten Anwendung ist für den allgemeinen Fall zu aufwendig und nicht praktikabel. Statt dessen wird ein Zustand hergestellt, auf dessen Basis eine Rekonfiguration von vorn beginnen kann, ohne

Kenntnis davon zu haben, dass die Anwendung schon in Teilen einer Rekonfiguration unterzogen wurde. Die grundsätzliche Idee ist es, die Rekonfiguration bis zum Ende durchzuführen, ohne weitere Änderungen an der Ablaufplanung vorzunehmen. Am ursprünglichen Code werden nur dann noch Änderungen vorgenommen, wenn dies der bereits umgeplante Code erforderlich macht.

Die folgenden Punkte zählen zu den Auswirkungen, die durch den bereits umgeplanten Code entstehen:

- Die Zieladressen von Sprungoperationen, die in den rekonfigurierten Code verzweigen sind ungültig und müssen auf den gültigen Wert aktualisiert werden.
- Ungepatchte Sprünge aus dem bereits geplanten Code sind nachträglich anzupassen, weil die Offset-Verschiebung sich auch auf den nachfolgenden Code auswirkt.
- Die entstandenen Verschiebungen durch den bereits geplanten Code müssen auch während der Abarbeitung des noch ungeplanten Codes beachtet werden.

Damit eine Rekonfiguration zu einem späteren Zeitpunkt und durch einen anderen Kern erneut vorgenommen bzw. weitergeführt werden kann, müssen die folgenden Punkte während der Zustandssicherung durchgeführt werden:

- Die Metadaten der Anwendung bezüglich der geänderten Adressen von Funktionen sind zu aktualisieren.
- Zu jeder Funktion muss der Metadaten-Bereich die gültigen internen Sprungadressen enthalten.
- Alle Sprungoperationen des Programmcodes müssen eine gültige Adresse verwenden.

Die lokale Reparatur ist entsprechend zu erweitern, damit diese Punkte Berücksichtigung finden. Zunächst muss durch die lokale Reparatur erkannt werden, wann ein Basisblock nicht erfolgreich planbar war. Ein Merkmal zur Erkennung einer gescheiterten Rekonfiguration ist es, wenn dem Planungsalgorithmus keine weiteren Ressourcen zur Verfügung stehen, um noch ungeplante Operationen zu planen. Ein anderer Ressourcenengpass ist erreicht, wenn die Anzahl an Reserveadressen nicht ausreichend ist, um einen verlängerten Basisblock zu speichern.

Abbildung 5.7 zeigt den Zustand einer Rekonfiguration zum Zeitpunkt, wenn die Reserveadressen überschritten sind. Eine Funktion $f$ wurde in Teilen umgeplant. Das nächste Fragment zur Umplanung befindet sich decodiert im Datenspeicher und gleichzeitig weiterhin codiert im Programmspeicher im Bereich des noch ungeplanten Codes. Der Reservebereich liegt zwischen dem bereits zurückgeschriebenen Binärcode und den noch nicht rekonfigurierten Codeteilen. Mit der Umplanung des Basisblocks $b_{i+1}$ ergibt sich eine Verlängerung, die nicht mehr in den Datenspeicher zurückgeschrieben werden kann. Zur Erkennung, ob ein Basisblock nicht zurückgeschrieben werden darf, werden die Werte des aktuellen Offsets, der Anfangsadresse des Fragments und die Längen der bereits geplanten Basisblöcke des aktuellen Fragments betrachtet. Ist die aufsummierte Verlängerung aller bereits geplanten Basisblöcke des aktuellen Fragments größer oder

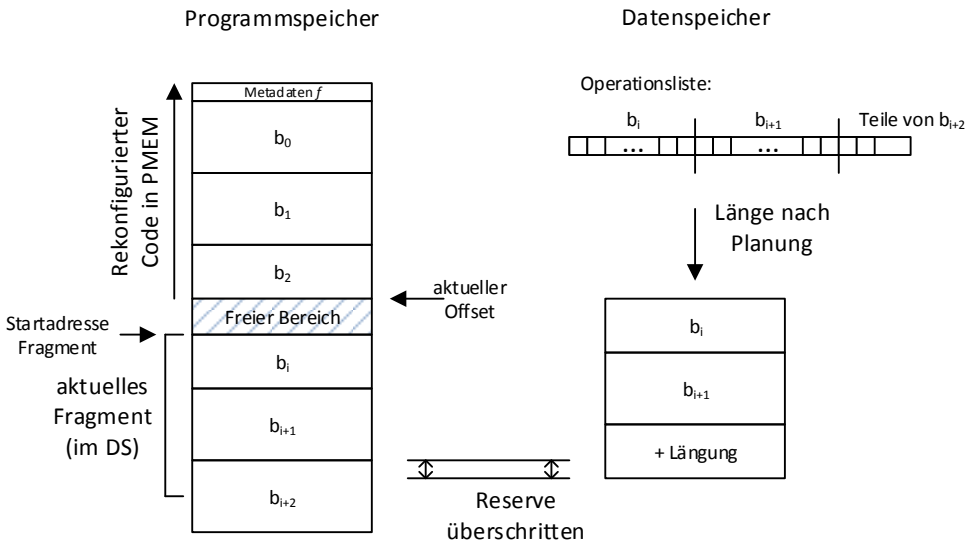


Abbildung 5.7.: Zwischenstand einer in Umplanung befindlichen Funktion

gleich der Differenz aus Anfangsadresse des Fragments und aktuellem Offset, dann kann der aktuell bearbeitete Basisblock nicht zurückgeschrieben werden.

Nachdem erkannt wurde, dass der aktuell geplante Basisblock $b$ Ressourcenbeschränkungen verletzt, werden die Änderungen bezüglich $b$ verworfen. Dazu wird der Code aus dem Programmspeicher für $b$ erneut eingelesen. Zu $b$ wird sich zunächst eine eventuell geänderte Startadresse gemerkt. Darüber hinaus muss eine mögliche Sprungoperation am Ende von $b$ behandelt werden. Ansonsten verbleibt der Basisblock unverändert.

Die Abfrage, ob die lokale Planung fehlschlug, wird in Algorithmus 4.9 zur Steuerung der lokalen Reparatur vor den Aufruf für das Sprungpatching (Zeile 17) eingefügt. Die Abfrage erfolgt vor dem Sprungpatching, weil der Code des Basisblocks erneut eingelesen und eine mögliche Sprungoperation am Ende des Basisblocks erst danach durch das Sprungpatching angepasst wird. Zur weiteren Behandlung einer fehlgeschlagenen Rekonfiguration wird ein Flag gesetzt, das die Ablaufplanung für die restlichen Basisblöcke der Rekonfiguration unterdrückt. Dazu werden die Aufrufe der Backends in Algorithmus 4.9 in Abhängigkeit vom Zustand des Flags gesetzt. Alle anderen Maßnahmen der Rekonfiguration werden regulär zu Ende durchgeführt. Dadurch werden Funktionen und Basisblöcke weiterhin identifiziert und ihre Adressen den aktuellen Gegebenheiten angepasst. Das Gleiche erfolgt auch für Sprungoperationen, die sich entweder schon gemerkt wurden oder die im weiteren Verlauf noch gefunden werden.

Nachdem eine Funktion vollständig abgearbeitet wurde, müssen die Metadaten zu den Sprungzielen für diese Funktion aktualisiert und im Programmspeicher hinterlegt werden. Dazu werden die Metadaten schrittweise ausgelesen. Da diese im Programmspeicher liegen bedeutet schrittweise, einen Datenblock in Instruktionswortbreite ein-

zulesen. In jedem Schritt werden die so ausgelesenen Adressen durch die in der Basisblockliste hinterlegten neuen Adressen ersetzt. Die neuen Adressen werden dann in der richtigen Reihenfolge in den Programmspeicher zurückgeschrieben. Das Aktualisieren der Metadaten wird generell in Ablauf zur Rekonfiguration integriert, weil eine Aktualisierung im Nachhinein durch den Verlust des Bezuges zwischen alten und neuen Adressen nicht möglich ist. Das liegt daran, weil die Basisblockliste am Ende der Rekonfiguration einer Funktion verworfen wird. Zum Zeitpunkt, an dem erkannt wird, dass eine Rekonfiguration nicht durchführbar ist, können dann bereits mehrere Listen überschrieben worden sein. Das Aktualisieren erfolgt in Bezug auf Algorithmus 4.9 nach dem Durchlaufen der Schleife zur Rekonfiguration einer Funktion (Zeile 22).

Am Ende der Rekonfiguration sind die geänderten Anfangsadressen aller Funktionen bekannt. Die Informationen dazu stehen in der Funktionsliste, in der zu jeder alten Anfangsadresse die neue Adresse abgebildet ist. Mit diesen Informationen wird abschließend der Metadatenbereich zu den Funktionen der Anwendung aktualisiert. Der rein technische Vorgang zum Aktualisieren erfolgt dabei analog zum Aktualisieren der Metadateninformationen bezüglich der Basisblöcke einer Funktion.

Auf die Behandlung eines defekten Registers mit der lokalen Registerallokation soll kurz gesondert eingegangen werden. Für den Ausfall eines globalen Registers und den Einsatz der Registervergabe aus Kapitel 4.2.4 ergeben sich keine zusätzlich zu beachtenden Probleme. Die Registervergabe kann zu einem späteren Zeitpunkt wieder aufgenommen werden. Bei der wiederholten Ausführung wird das Verfahren die gleichen Register zur Ersetzung defekter globaler Register verwenden. Eine Besonderheit ergibt sich jedoch für die bereits umbenannten globalen Register. Diese müssen in einem abschließenden Renaming-Durchlauf wieder auf den alten Wert gebracht werden. Andernfalls werden bei einer erneuten Registerallokation durch Algorithmus 4.4 falsche Deskriptoren erzeugt, da die Abbildung zwischen globalen und lokalen Registern nicht mehr gültig ist.

Damit eine wiederholte Rekonfiguration einfacher durchführbar ist, ist es sinnvoll, den Reservebereich des Programmspeichers nicht vollständig auszunutzen, sondern einen rechtzeitigen Rekonfigurationsabbruch bei verbleibenden 5 % vorzunehmen. Durch das Zurückhalten von Einträgen steht einer erneuten Rekonfiguration ein zugesichertes Minimum an Einträgen zur Verfügung. Andernfalls kann es im ungünstigsten Fall passieren, dass eine Rekonfiguration ohne Reserveeinträge durchgeführt wird.

#### Vertauschen zweier Programmspeicherinhalte

Nach der Zustandssicherung einer fehlgeschlagenen lokalen Reparatur und der anschließenden Neuzuordnung zwischen zwei Tasks und den korrespondierenden Kernen, kann das eigentliche Vertauschen der Programmspeicherinhalte erfolgen. Die wesentlichen Schwerpunkte beim Vertauschen ergeben sich aus den bereits bekannten Ressourcenbeschränkungen und den unterschiedlich breiten Programmspeichern. Das Vertauschen der Speicherinhalte erfolgt in einem überlappenden Verfahren. Das Verfahren führt schrittweise eine Rekonfiguration für eine der beiden Tasks durch, wohingegen die andere in ihrem Ausgangszustand bestehen bleibt. Kann am Ende eines Schrittes der rekonfigurierte Code zurückgeschrieben werden, erfolgt dies in den neu zugeordneten Programmspeicher. Im Anschluss wird eine gleichgroße Anzahl an Bytes von der unge-

planten Task in deren neu zugewiesene Programmspeicher kopiert, ohne dass der Code rekonfiguriert wird.

Von zwei Tasks wird diejenige direkt rekonfiguriert, deren lokale Reparatur zuvor fehlschlug. Diese Task soll im Weiteren als $t_{fail}$ bezeichnet werden. Weil eine solche Task typischerweise einem Kern $c_{spare}$ mit Spare-Komponente zugeordnet wird, wird die zweite Task, welche ursprünglich $c_{spare}$ zugeordnet war, als $t_{spare}$ bezeichnet. Der Ablauf zum Vertauschen wird durch Algorithmus 5.2 beschrieben. Zu Beginn wird der Code von $t_{spare}$ an des Ende des Programmspeichers kopiert. Dadurch wird im vorderen Bereich Platz für die Aufnahme des rekonfigurierten Codes von $t_{fail}$ geschaffen. Anschließend erfolgt die schrittweise Rekonfiguration von $t_{fail}$ mit einem lokalen Backend, wie dem Rescheduling oder der Registerallokation.

**Algorithm 5.2** Steuerung des Task-Rebinding

**Eingabe: ID** $core1$**, ID** $core2$**, Task** $t_{spare}$**, Task** $t_{fail}$
**Ausgabe: keine**

- verschiebe Code von $t_{spare}$ an das Programmspeicherende
- $globalOffset$ := Startadresse von $t_{fail}$
- einlesen der Metadaten und initialisieren von $funcs$
**for all** $f$ aus $funcs$ **do**
    - schreibe Metadaten von $f$ zurück an Adresse $globalOffset$ in $PMEM_{core1}$
    - kopiere entsprechend viele Bytes von $t_{spare}$ nach $PMEM_{core2}$
    - vermerke in $funcs$ den Wert von $globalOffset$ als neue Startadresse von $f$
    - einlesen der Metadaten von $f$ und initialisieren von $bbs$
    - initialisieren von $cWnd$ mit der ersten Codeadresse von $f$
    **repeat**
        - lese Code von $cWnd$ in $ops$ ein und führe Basisblockrekonstruktion aus
        **for all** Basisblöcke $b$ gefunden in $cWnd$ **do**
            - rekonfiguriere $b$
            - führe Sprungpatching aus, lade externe Sprünge aus $PMEM_{core1}$
        **end for**
        - schreibe rekonfigurierten Code nach $PMEM_{core1}$, und aktualisiere $globalOffset$
        - verschiebe entsprechend viele Bytes von $t_{spare}$ nach $PMEM_{core2}$
        - setze $cWnd$ weiter
    **until** Endadresse von $f$ erreicht
**end for**

Im Verlgeich zu einer lokalen Rekonfiguration die auf dem gleichem Programmspeicher arbeitet ergeben sich Veränderungen in der Organisation. Die erforderlichen Änderungen sind:

- Die zu rekonfigurierende Task $t_{fail}$ wird nicht an das Ende des Programmspeichers verschoben.
- Der rekonfigurierte Code von $t_{fail}$ wird in einen anderen Programmspeicher zurückgeschrieben.

- Nachdem der rekonfigurierte Code zurückgeschrieben wurde, wird eine entsprechende Anzahl an Bytes der Task $t_{spare}$ in den Programmspeicher von $t_{fail}$ kopiert. Das bedeutet, $t_{fail}$ wird funktions- und fragmentweise kopiert, wohingegen $t_{spare}$ nur auf Basis von Byteblöcken kopiert wird, ohne die Struktur von $t_{spare}$ zu berücksichtigen.

Algorithmus 5.2 beinhaltet die notwendigen Erweiterungen für das Vertauschen zweier Tasks im Vergleich zum lokalen Rekonfigurieren mit Algorithmus 4.9. Die wesentlichen Unterschiede sind in den Zeilen 4, 9, 17, 19 und 20 aufgetreten. Darüber hinaus erfolgt kein Rebinding bzw. Renaming, da dieser Programmcode bereits in Teilen umgeplant ist und in einen gültigen Zustand gebracht werden muss. An den Stellen, an denen ein Verschieben des Codes von $t_{spare}$ erfolgt (Zeile 9 und Zeile 20), bezieht sich die Byteanzahl auf die tatsächliche Blockgröße, die aus dem Speicher von $t_{fail}$ gelesen wurde, ohne etwaige verlängerte Basisblöcke einzubeziehen. Für das Sprung-Patching ist zusätzlich zu beachten, dass gemerkte externe Sprünge zum Patchen im anderen Programmspeicher zu aktualisieren sind (Zeile 17).

Abbildung 5.8 zeigt einen schematischen Ausschnitt aus einer beispielhaften Konfiguration für das Vertauschen zweier Tasks. Die Abbildung zeigt den Stand zu Beginn der Planung des zweiten Fragments der Task 2. Der erste Kern ist der Spare-Kern und stellt den dritten Slot als Reserve bereit. Dazu ist angedeutet, dass Task 1 nicht den vollen Programmspeicher in der Breite belegt, sondern für den dritten Slot Nop-Operationen enthält. Task 1 wurde zu Beginn an das Ende des Programmspeichers verschoben, wohingegen dies für Task 2 nicht erfolgt ist. Nachdem das erste Fragment von Task 2 umgeplant wurde, wurde es in den Programmspeicher 1 geschrieben. Anschließend konnte die entsprechende Anzahl an Bytes des zweiten Programmspeichers mit dem Code der Task 1 überschrieben werden.

Abschließend seien noch einige Einschränkungen für das Verfahren der globalen Rekonfiguration erwähnt, damit das Verfahren der globalen Systemrekonfiguration in der beschriebenen Form umsetzbar ist. Bei einer Rekonfiguration werden immer die gesamten Anwendungen zwischen den Programmspeichern getauscht. Das bedeutet, dass ein Kern immer alle Aufgaben eines anderen Kerns übernimmt. Durch diese Herangehensweise wird vermieden, dass die Kommunikation zwischen den Aufgaben vom Rekonfigurationsprozess berücksichtigt werden muss. Die Kommunikation im vorgestellten System kann einmal über Ports und über den gemeinsamen Speicher erfolgen. Der gemeinsame Speicher kann von jedem Kern geschrieben und gelesen werden, weshalb der Kommunikationsvorgang zwischen getauschten Kernen unberührt bleibt. Der Datenaustausch über Ports ist im vorliegenden Modell in der Form implementiert, dass jeder Kern beliebig auf alle Ports zugreifen kann. Somit ist auch hierfür die Kommunikation zwischen den Kernen abgesichert, solange Lese- und Schreibrechte nicht exklusiv zwischen einem Kern $C_x$ und einem Port $P_x$ vereinbart werden.

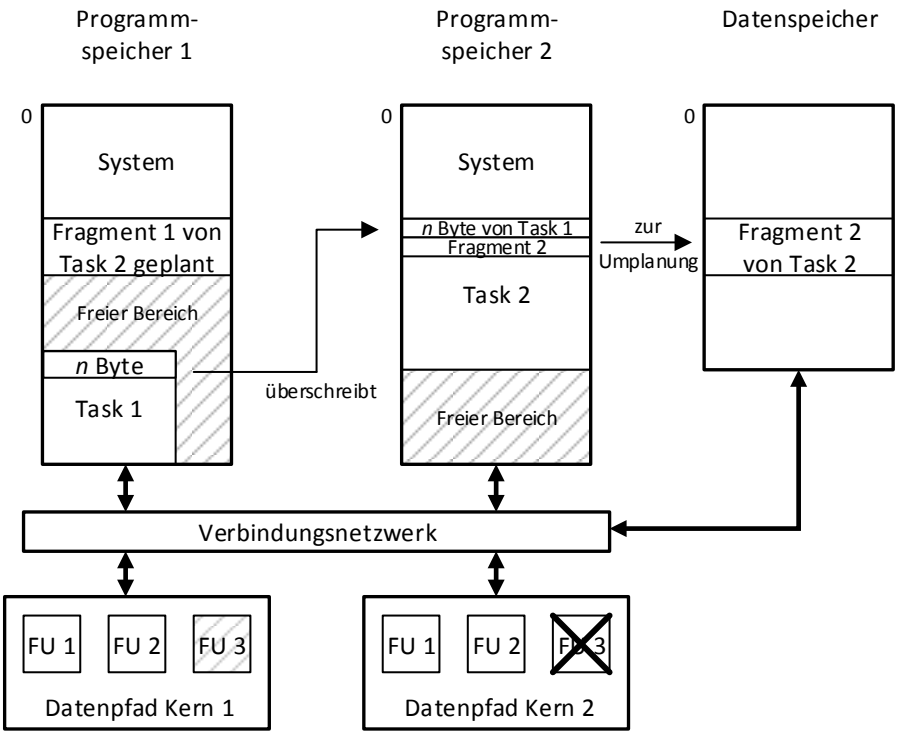


Abbildung 5.8.: Schematisches Vorgehen zum Vertauschen zweier Tasks

# 6. Qualität der hierarchischen Rekonfiguration

Das folgende Kapitel bewertet die entwickelte Rekonfigurationsstrategie bezüglich der Qualitätsmerkmale Zuverlässigkeit, zeitlicher Aufwand der Rekonfiguration und erhöhter Hardwarekosten. Zunächst wird auf die prototypische Systemimplementierung eingegangen. Dazu zählt einmal die Hardware-Umsetzung des Systemmodells und weiterhin die Software-Implementierung der Reparaturalgorithmen. Die Reparaturalgorithmen wurden bezüglich einer degradierten Systemleistung für unterschiedliche Komponentenausfälle untersucht. Basierend auf diesen Ergebnissen wird der erhöhte Speicher- und Hardwareaufwand mehrerer Systemkonfigurationen und Defektkonstellationen bestimmt. Im Anschluss werden die Reparaturzeiten für unterschiedliche Reparaturszenarien angegeben, die sich für eine lokale beziehungsweise globale Reparatur ergeben. Im abschließenden Unterkapitel wird mit Hilfe von Fehlersimulationen eine Steigerung der mittleren Lebensdauer bei Verwendung der vorgestellte Reparaturstrategie in einem fehlertoleranten System nachgewiesen.

## 6.1. Systemimplementierung

Zur Untersuchung der hierarchischen Reparaturstrategie dieser Dissertation erfolgten verschiedene prototypische Implementierungen. Das angegebene Prozessormodell wurde in VHDL implementiert und in eine Implementierung des Systemmodells integriert. Im folgenden Abschnitt wird dieser Teil als Hardware-Implementierung präsentiert. Die entwickelten Algorithmen der Reparatur wurden in Assembler programmiert und werden im Abschnitt zur Software-Implementierung vorgestellt. Weil die Untersuchung der Algorithmen in Hinblick auf ihre Eigenschaften durch eine Simulation des VHDL-Modells zu zeitaufwendig ist, wurde zusätzlich eine Simulationsumgebung mit grafischer Oberfläche in C++ entwickelt. Die Simulationsumgebung implementiert für eine konkrete Systemkonfiguration für jeden Kern einen taktgenauen Befehlssatz-Simulator. Darüber hinaus ist auch das zeitliche Verhalten des Verbindungsnetzwerkes integriert mit den Zugriffen auf die Programmspeicher. Die Simulationsumgebung wird im weiteren Verlauf nicht weiter besprochen, da sie lediglich eine komfortable Möglichkeit zur Analyse der Algorithmen bietet. Die VHDL-Implementierung des Systems liefert die wesentlichen Daten für die durchgeführte Fehlersimulation.

### 6.1.1. Hardware-Implementierung

Das beschriebene Systemmodell und die VLIW-Kerne wurden in zwei Versionen implementiert. In der ersten Variante entfallen alle Mechanismen, die für die software-basierte Rekonfiguration erforderlich sind. Diese Mechanismen sind in der zweiten Variante, dem

eigentlichen fehlertoleranten System, implementiert. Mit einem Vergleich der beiden Implementierungen kann der zusätzliche Platzaufwand in Hardware für ein System, das die software-basierte Rekonfiguration einsetzen kann, angegeben werden. Die notwendigen Erweiterungen wurden in den zurückliegenden Kapiteln besprochen und beziehen sich auf die folgenden Teile:

- Implementierung eines Programmtransfercontrollers im Verbindungsnetzwerk, um Daten zwischen Programm- und Datenspeicher auszutauschen.
- Erweiterung des Speichercontrollers im Verbindungsnetzwerk mit einem Zugriffsschutz für Systemressourcen.
- Erweiterung der I/O-Anbindungen der Kerne mit einer Möglichkeit zur Ansteuerung der neuen Funktionalitäten im Verbindungsnetzwerk.

Die Implementierung erfolgte auf Registertransferebene mit der Sprache VHDL. Zur Synthese wurde dem rtl-Compiler die 45 nm Bibliothek OpenCellLibrary[1] zur Verfügung gestellt. Es wurden zwei Synthesedurchläufe mit verschiedenen Optimierungen durchgeführt. Das erste Optimierungsziel teilt dem Compiler mit, dass eine Minimierung des Platzbedarfes der Schaltung höchste Priorität hat. Im zweiten Durchlauf wurde als Optimierungsziel das Erreichen einer höchst möglichen Taktfrequenz angegeben.

Tabelle 6.1 zeigt die Ergebnisse für eine platzoptimierte Synthese eines Mehrkernsystems in der Konfiguration $(4, 4, 4, 4)$. Das System setzt sich aus vier Kernen zusammen, die jeweils vier Verarbeitungsslots besitzen und jeweils auf 64 allgemeine Register zugreifen können. Es sind weiterhin 64 allgemeine Portregister zur internen sowie externen Kommunikation verfügbar. Die Verarbeitungsbreite im Datenpfad beträgt 16 Bit. Zur Adressierung des Daten- und Programmspeichers werden 16 Bit verwendet. Im Datenspeicher werden je Eintrag 16 Bit angesprochen und im Programmspeicher 104 Bit. Die Ergebnisse der Synthese sind in Tabelle 6.1 auf Nand2-Flächenäquivalente umgerechnet. Demzufolge entspricht der Wert 1 der Fläche von einem Nand2-Gatter, wenn es mit der genannten Bibliothek erzeugt wurde.

Das nicht fehlertolerante System besteht aus 115.589 Flächeneinheiten. Das Verbindungsnetzwerk besitzt einen Anteil von 1,13 % an der gesamten Schaltung. Die Aufgaben des Verbindungsnetzwerkes umfassen die interne und externe Kommunikation und die Zugriffskoordination für den gemeinsamen Datenspeicher. Ein einzelner Kern setzt sich aus 28.569 Flächeneinheiten (24,71 % des Gesamtsystems) zusammen. Den Großteil der Fläche eines Kerns belegt die Registerbank mit den Lese- und Schreibports (64,87 %). Ein einzelner Verarbeitungsslot entspricht 7,56 % eines Kerns. Eine 16 Bit FU hat in etwa den gleichen Flächenaufwand wie ein Leseport der Registerbank. Die genannten Prozessorbaugruppen zählen alle zu den Komponenten, deren Nutzung sich durch die software-basierte Rekonfiguration anpassen lässt. Die Verwendung des Programmzählers (1,18 % eines Kerns) und der I/O-Anbindung (3,69 % eines Kerns) können nicht durch die Rekonfiguration angepasst werden. Daraus ergibt sich, dass in etwa 95 % der Prozessorstrukturen durch die software-basierte Rekonfiguration bei ausreichender Hardwareredundanz reparierbar sind. Die Komponenten, die nicht durch

[1] Die Bibliothek steht unter http://www.nangate.com/ zur freien Verfügung.

| Komponente | nicht-tolerantes System | %-Anteil am Gesamtsystem | fehlertolerantes System | Overhead |
|---|---|---|---|---|
| System (4,4,4,4) | 115.589 | 100 | 119.850 | 3,68 |
| Kerne | 114.276 | 98,87 | | |
| Verbindungsnetz | 1.313 | 1,13 | 3.534 | 169,15 |
| Kern (4 Slots) | 28.569 | 24,71 | 29.079 | 1,78 |
| Registerbank | 18.591 | 64,87 | | |
| 1 Lese-Port | 998 | 3,49 | | |
| 8 Lese-Ports | 11.976 | 41,19 | | |
| I/O-Anbindung | 1.132 | 3,69 | 1.549 | 36,83 |
| Programmzähler | 339 | 1,18 | | |
| 1 Slot | 2.166 | 7,56 | | |
| Fetch | 180 | 0,63 | | |
| Decode | 420 | 1,47 | | |
| FU (16 Bit) | 938 | 3,27 | | |
| Bypass | 456 | 1,59 | | |
| WB | 172 | 0,60 | | |

Tabelle 6.1.: Größenangaben von Systemkomponenten in Nand2-Flächenäquivalenten

die Rekonfiguration abgedeckt werden, werden im weiteren Verlauf auch als **kritische Komponenten** bezeichnet.

Der Anteil an kritischen Komponenten erhöht sich für das fehlertolerante System. Die kritischen Komponenten stellen gleichzeitig auch den Hardware-Mehraufwand dar, den der Einsatz einer software-basierten Rekonfiguration erforderlich macht. Wie schon zu Beginn erwähnt, ergibt sich der Mehraufwand durch die Erweiterungen am Verbindungsnetzwerk und die entsprechende Anbindung der Kerne. Der höhere Platzaufwand fällt moderat aus mit einer Steigerung von 3,68 % für das Gesamtsystem und einer Steigerung von 1,78 % je Kern.

Das Syntheseprogramm gab für den längsten kritischen Pfad bei einer platzoptimierten Synthese eine Verzögerung von 2,9 ns an. Daraus ergibt sich ein Systemtakt von rund 340 MHz. Für das fehlertolerante System ergaben sich keine zusätzlichen Verzögerungen. In einem zeitoptimierten Syntheselauf wurde ein Systemtakt von rund 700 MHz für beide Systemkonstellationen ermittelt.

### 6.1.2. Software-Implementierung

Im folgenden Abschnitt werden die Kennzahlen für die Software-Implementierung der lokalen Rekonfiguration präsentiert. Die Darstellung des Overheads erfolgt getrennt für die verschiedenen Aufgabenteile der lokalen Reparatur. Die Kennzahlen des Overheads zeigt Tabelle 6.2 einmal für den Programm- und einmal für den Datenspeicher.

Der gesamte Assemblercode der lokalen Rekonfiguration umfasst 3.011 Instruktionen. Für eine VLIW-Architektur mit 4 Slots und einer Instruktionskodierung nach Abbildung 3.2 sind dafür im Programmspeicher 39.611 Byte bzw. 38,68 kByte erforderlich. Der Overhead für den Datenspeicher beträgt 9.884 Bytes.

| Aufgabenteile | lokale Reparatur | |
|---|---|---|
| | Programmspeicher-einträge | Datenspeicher-einträge |
| Code umkopieren | 22 | 4 Byte |
| Metadaten einlesen | 45 | 210 Byte |
| Fragment einlesen (Code kopieren/decodieren) | 174 | 7.080 Byte |
| BBlock-Rekonstruktion | 323 | 202 Byte |
| Code zur Steuerung | 263 | 16 Byte |
| Backends | | |
| Rebinding | 305 | 148 Byte |
| Rescheduling | 717 | 304 Byte |
| Renaming | 104 | 272 Byte |
| Register-Allokation | 554 | 1.198 Byte |
| Sprungpatching | 353 | 450 Byte |
| Write-Back (Codieren und Kopieren) | 151 | |
| Insgesamt lok. Rep. | 3.011 | 9.884 Byte |

Tabelle 6.2.: Belegung an Programmspeicher- und Datenspeichereinträgen für die einzelnen Aufgaben der lokalen Reparatur

Die beiden komplexen Backends Rescheduling und Registerallokation besitzen den größten Anteil am Programmspeicheroverhead mit 717 Instruktionen bzw. 554 Instruktionen. Das Einlesen und Dekodieren der Fragmente macht den größten Anteil am Datenspeicheroverhead aus. Die 7.080 Byte bestehen dabei im Wesentlichen aus dem Puffer für den eingelesenen Binärcode und aus der Operationsliste, in der die decodierten Operationen gespeichert werden. Die Registerallokation besitzt mit 1.198 Byte den zweitgrößten Overhead am Datenspeicher. Dieser Overhead entsteht maßgeblich durch den Platzaufwand der temporär geführten Listen *lookup*, *regs* und *vars*.

Der Overhead für den Defektspeicher ist nicht in Tabelle 6.2 zu angegeben und wird gesondert ausgewiesen. Der Defektspeicher für einen Kern mit 64 Registern und 8 Slots, belegt 47 Bytes bzw. 367 Bit. Für jede Komponente beziehungsweise Funktionalität, deren Ausfall durch eine Rekonfiguration behandelt werden soll, ist ein Bit reserviert, um den entsprechenden Zustand zu repräsentieren. Für die Register werden 64 Bit, für die Slots 8 und für die Ports 16 (2 je Port) reserviert. Es werden 8 Operationstypen je FU unterschieden. Dazu zählen die Grundrechenoperationen $(+, -, *, /)$, die logischen und

Bitoperationen, Speicheroperationen, Verzweigungsoperationen und I/O-Operationen. Der Bypass setzt sich für jeden Slot aus 56 Bit zusammen. Je Slot werden der linke und der rechte Operand sowie zwei vorhergehende Takte unterschieden. Dazu kommen jeweils 8 Bit, die anzeigen, ob aus dem jeweiligen Slot ein Forwarding erfolgen kann.

Tabelle 6.3 zeigt die Zahlen für den Overhead, der durch den Einsatz der globalen Reparaturstrategie entsteht. Der gesamte Overhead für den Programm- und Datenspeicher wird unterteilt in den Code zur Administration der globalen Reparatur und in die Erweiterungen der lokalen Reparatur.

| Aufgabenteile | globale Reparatur | |
|---|---|---|
| | Programmspeicher-einträge | Datenspeicher-einträge |
| Guard (4 Slots) | 216 | 4 Byte |
| Synchronisation | 37 | 10 Byte |
| Fremreparatur | 237 | 4 Byte |
| Task-Rebinding | 259 | 42 Byte |
| Gesamt | 749 | 60 Byte |
| Erweiterung lokal für Zustandsicherung | 233 | 4 Byte |
| Erweiterung lokal für globale Reparatur | 154 | 6 Byte |

Tabelle 6.3.: Overhead für Programm- und Datenspeicher für die globale Reparaturstrategie

Die Erweiterungen der lokalen Reparaturstrategie ergeben sich einmal für die Sicherung einer abgebrochenen lokalen Rekonfiguration und durch die Modifikation der lokalen Rekonfiguration, um diese im globalen Kontext einzusetzen. Für die Sicherung sind 233 zusätzliche Instruktionen und 4 Byte Datenspeicher notwendig. Die Implementierung zur Sicherung umfasst, das Aktualisieren der Metadaten, das Erkennen eines Abbruchs und das erneute Einlesen des letzten Basisblocks sowie die Abfrage, ob überhaupt ein Backend auszuführen ist. Die Modifikation des Codes der lokalen Reparatur umfasst 154 Instruktionen und 6 zusätzliche Bytes im Datenspeicher. Die Modifikationen integrieren das Zurückschreiben des Codes der Task $t_{spare}$ an den Stellen im Ablauf der lokalen Reparatur.

Der Overhead für die Administration der globalen Reparatur setzt sich aus den Teilen Guard, Synchronisation, Fremdreparatur und Task-Rebinding zusammen. Insgesamt bestehen diese Teile aus 749 Instruktionen und benötigen 60 Byte zusätzlichen Datenspeicher. Dieser Code ist auf jedem Kern des Systems vorhanden, weil jeder Kern die Aufgaben des Masters übernehmen kann.

## 6.2. Ergebnisse der software-basierten Rekonfiguration

Im Folgenden werden die Untersuchungsergebnisse für die software-basierte Rekonfiguration hinsichtlich der Ausführungszeiten, der Systemdegradation und dem Mehraufwand diskutiert. Zunächst wird der statische Laufzeit-Overhead der lokalen und der globalen Reparaturstrategie präsentiert. Der **statische Laufzeit-Overhead** bezeichnet die erforderliche Ausführungszeit der Reparaturalgorithmen auf dem System während des Systemstarts. Die Ausführung der Rekonfiguration verzögert den Start des Systems und wird deshalb als statisch bezeichnet. Der Aufwand zur Berechnung ist nicht erneut erforderlich und hat keinen Einfluss auf die Ausführungszeiten der regulären Anwendung. Im Gegensatz dazu beeinflusst eine rekonfigurierte Anwendung mit einer verlängerten Ablaufplanung die Laufzeiten der regulären Ausführung. Der sich so ergebende Overhead wird als **dynamischer Laufzeit-Overhead** bezeichnet. Der dynamische Laufzeit-Overhead wird im Abschnitt zur Degradation der Systemleistung besprochen. Abschließend werden für verschiedene Systemkonfigurationen die Mehrkosten an Hardware diskutiert, die sich für bestimmte Fehlerkonstellationen ergeben.

### 6.2.1. Laufzeiten der Rekonfiguration

In diesem Abschnitt werden die Ausführungszeiten der Reparaturalgorithmen dargestellt. Um repräsentative Werte zu ermitteln, wurden fiktive Benchmarkprogramme erzeugt, die verschiedene Werte für die Anzahl an Funktionen, Basisblöcken und Sprüngen aufweisen.

Tabelle 6.4 zeigt eine Auswahl der erzeugten Pseudo-Benchmarks mit Angaben zu deren relevanten Eigenschaften. Die Bezeichnung der Benchmark-Programme gibt einmal die Anzahl der enthaltenen Funktionen und die durchschnittliche Anzahl an Basisblöcken je Funktion wieder. Die Programme sind aufsteigend nach der Anzahl an Funktionen angegeben. Zum Beispiel besitzt das Benchmarkprogramm *app_f4_b8* 4 Funktionen, die durchschnittlich 8 Basisblöcke enthalten. Insgesamt sind 30 Basisblöcke und 322 Instruktionen vorhanden. Das Programm belegt, inklusive der Metadaten, im Programmspeicher 4.446 Bytes und ist für eine Architektur mit 4 Slots übersetzt. Das Benchmarkprogramm in der letzten Spalte besitzt 28 Funktionen und besteht insgesamt aus 295 Basisblöcken. Die 3.417 Instruktionen belegen 46.241 Byte im Programmspeicher.

Weil der dominierende Faktor für die Laufzeiten der Reparatur die Neuberechnung der Ablaufplanung ist, werden die Backends zusätzlich mit typischen Mediabenchmarks untersucht. Untersucht wird das Verhalten einmal in Hinblick auf die Rekonfigurationszeit und die Qualität der Rekonfiguration selbst. Zur Untersuchung werden die Basisblöcke mit den zentralen Berechnungen der inneren Schleife eines jeden Benchmarks betrachtet. Tabelle 6.5 zeigt eine Übersicht der Eigenschaften der untersuchten Basisblöcke. Die verwendeten Operationstypen in allen Benchmarks beschränken sich auf die Addition, Subtraktion und Multiplikation. Zum Beispiel enthält der Basisblock des *Auto Regression Filter* 32 Operationen, die, für eine Architektur mit 4 Slots, in 8 Instruktionen geplant wurden. Der Basisblock der Diskreten Cosinustransformation enthält in der Implementierung *DCT-LEE* 56 Operationen und besitzt eine Länge von 14 Instruktionen, wenn der Code für 4 Slots geplant wurde.

| | app_f4_b8 | app_f8_b7 | app_f9_b12 | app_f16_b13 | app_f28_b10 |
|---|---|---|---|---|---|
| Funktionen | 4 | 8 | 9 | 16 | 28 |
| Basisblöcke | 30 | 52 | 109 | 215 | 295 |
| ∅ Instr. je Block | 11 | 12 | 11 | 10 | 12 |
| Operationen (ohne Nop) | 1.171 | 2.321 | 4.359 | 8.182 | 12.305 |
| Instruktionen | 322 | 648 | 1.202 | 2.259 | 3.417 |
| Größe in Byte (mit Metadaten) | 4.446 | 8.944 | 16.211 | 30.407 | 46.241 |

Tabelle 6.4.: Ausgewählte Eigenschaften der Pseudo-Benchmarkprogramme

| Benchmark | Eigenschaft | |
|---|---|---|
| | Operationen | Länge bei 4 Slots |
| Auto Regression Filter (ARF) | 32 | 8 |
| Eliptic Wave Filter (EWF) | 56 | 14 |
| Fast Fourier Transform (FFT) | 40 | 10 |
| Discrete Cosine Transform (DCT-DIF) | 44 | 11 |
| Discrete Cosine Transform (DCT-DIT) | 56 | 14 |
| Discrete Cosine Transform (DCT-LEE) | 56 | 14 |

Tabelle 6.5.: Ausgewählte Eigenschaften der untersuchten Mediabenchmarks

#### Statischer Laufzeit-Overhead der lokalen Rekonfiguration

Die Ausführungszeit der lokalen Rekonfiguration setzt sich aus den Laufzeiten der regelmäßig wiederkehrenden Aufgaben zur Administration und den Laufzeiten der Planungsverfahren zusammen. Zunächst werden die Laufzeiten für die gesamte lokale Rekonfiguration betrachtet, wobei hier das Hauptaugenmerk auf den Laufzeiten für die Organisation der Reparatur liegt. Die Laufzeiten für die eigentliche Rekonfiguration werden im Anschluss detailliert dargestellt.

Tabelle 6.6 zeigt den statischen Laufzeitoverhead für die fünf genannten Pseudo-Anwendungen. Die Anwendungen sind aufsteigend nach der Funktionsanzahl und der Instruktionsanzahl sortiert. Der Code der Anwendungen ist für eine VLIW-Architektur mit 4 Slots im Datenpfad übersetzt. Zunächst lässt sich festhalten, dass mit steigender Instruktionsanzahl auch die Gesamtzeit der Rekonfiguration steigt. Für die Anwendung mit 4 Funktionen (322 Instruktionen) beträgt die lokale Rekonfiguration 1.899.682 Takte und für die Anwendung mit 28 Funktionen (3.417 Instruktionen) 18.546.548 Takte.

Die Laufzeit der Rekonfiguration setzt sich aus dem Anteil zur Organisation und der Laufzeit des Backends zusammen. Für die Untersuchung der fiktiven Anwendungen wurde als Backend generell das Rescheduling eingesetzt mit der Vorgabe, dass ein Defekt im Slot 4 zu behandeln ist. Das Verhältnis zwischen Rekonfiguration und Or-

ganisation an der Gesamtlaufzeit beträgt durchschnittlich 80% für die Rekonfiguration und 20% für die administrativen Aufgaben der lokalen Reparatur.

Das Einlesen der Metadaten als administrative Aufgabe hat einen geringen Anteil an der Laufzeit und steigt mit der Anzahl an Funktionen, weil dadurch mehr Startadressen einzulesen sind. Das Verschieben der Anwendung fällt mit einem durchschnittlichen Aufwand von 0,27% der Gesamtlaufzeit wenig ins Gewicht. Das Präprocessing umfasst das Einlesen der Metadaten zu jeder Funktion und das Vorinitialisieren der Basisblockliste. Mit einer höheren Anzahl an Funktionen und Basisblöcken steigt dieser Aufwand, fällt aber im Verhältnis zur Gesamtlaufzeit gering aus. Das Code-Einlesen, also das Kopieren des Programmcodes in den Datenspeicher, hat einen ähnlichen Aufwand wie das Verschieben der Anwendung. Das anschließende Dekodieren des Binärcodes und das Initialisieren der Operationsliste stellen einen nicht unerheblichen zeitlichen Aufwand dar. Die Laufzeit dafür beträgt durchschnittlich 10% der Gesamtlaufzeit und steigt mit der Anzahl an Operationen in der Anwendung. Mit durchschnittlich 3,6% der Gesamtlaufzeit fällt die Rekonstruktion der Basisblöcke verhältnismäßig günstig aus. Im Vergleich dazu ergibt sich für das Anpassen der Sprünge ein noch geringerer Aufwand mit 0,31% für $app\_f4\_b8$ und bis zu 1,23% für $app\_f28\_b10$. Das Codieren der Operationen in Maschinencode und das Zurückschreiben in den Programmspeicher hat, im Vergleich zu den anderen Teilen der Rekonfiguration, einen hohen Anteil an der Gesamtlaufzeit.

Zusammenfassend kann festgehalten werden, dass der Mehraufwand zur Organisation der lokalen Reparaturstrategie moderat im Verhältnis zum Gesamtaufwand ausfällt. Die Rekonfiguration als solches stellt den dominierenden Faktor dar. Insgesamt bietet die Implementierung noch wesentliches Potential für Verbesserungen im Hinblick auf die Laufzeitanforderungen. Das Dekodieren und Kodieren der Instruktionen bietet Spielraum für Einsparungen. Es ist weiterhin zu beachten, dass der Code der Reparaturstrategie im Hinblick auf die globale Reparaturstrategie fehlertolerant implementiert ist und dass das Potential der parallelen Operationsausführung des VLIWs nicht genutzt wird.

In Abhängigkeit von dem verwendeten Backend ergeben sich unterschiedliche Gesamtlaufzeiten für die lokale Rekonfiguration. Wird lediglich das Rebinding eingesetzt, ist die statische Reparaturzeit wesentlich geringer als der Einsatz des Reschedulings oder der Registerallokation. Diese beiden Backends dominieren die Ausführungszeit der lokalen Rekonfiguration im Vergleich zum Anteil der Aufgaben der Administration. Tabelle 6.7 zeigt nochmals gesondert die Laufzeiten der vier Backends, angewendet auf die Basisblöcke der Mediabenchmarks. Der Code der Basisblöcke ist für einen VLIW-Prozessor mit 4 Slots erzeugt. Die Backends Rebinding und Rescheduling rekonfigurieren die Blöcke mit der Annahme, dass die vierte FU defekt ist. Das Renaming und die Registerallokation rekonfigurieren die Blöcke mit der Annahme, dass das Register $r_1$ defekt ist.

Die kürzeste Ausführungszeit hat das Renaming mit einem Wert von 1.661 Takten für die Rekonfiguration des ARF-Benchmarks und bis zu 2.886 Takten für die Rekonfiguration des Benchmarks *DCT-DIT*. Das Rebinding hat leicht höhere Ausführungszeiten als das Renaming. Die Laufzeiten reichen von 2.504 Takten für den Benchmark *ARF* und bis zu 5.106 Takte für die Rekonfiguration des Benchmarks $DCT - LEE$.

| Aufgabenteil | app_f4_b8 | app_f8_b7 | app_f9_b12 | app_f16_b13 | app_f28_b10 |
|---|---|---|---|---|---|
| Metadaten lesen | 126 | 241 | 356 | 471 | 816 |
| Umkopieren | 5.142 | 10.332 | 18.717 | 35.097 | 53.367 |
| Preprocessing | 680 | 1.360 | 1.602 | 2.848 | 4.872 |
| Code lesen | 7.445 | 14.365 | 27.449 | 51.638 | 71.926 |
| Decodieren | 157.242 | 300.404 | 581.580 | 1.088.459 | 1.508.950 |
| Basisblöcke | 60.835 | 114.161 | 252.696 | 473.100 | 622.330 |
| Rekonfiguration | 1.548.529 | 2.960.453 | 5.763.379 | 10.674.793 | 14.963.686 |
| Sprünge | 5.933 | 16.149 | 63.940 | 169.688 | 229.008 |
| Codieren | 111.220 | 212.252 | 411.372 | 771.512 | 1.069.264 |
| Gesamttakte | 1.899.682 | 3.634.461 | 7.129.396 | 13.283.882 | 18.546.548 |
| Gesamtzeit in s (für 350MHz) | 0,0054 | 0,0104 | 0,0204 | 0,0380 | 0,0530 |

Tabelle 6.6.: Statische Ausführungszeiten der lokalen Rekonfiguration für die Back-Ends Rescheduling und Register-Allokation

Die beiden komplexen Backends, das Rescheduling und die Registerallokation, haben erwartungsgemäß deutlich höhere Ausführungszeiten als die einfachen Backends. Die komplexen Backends haben eine um den Faktor 10 höhere Laufzeit. Das Rescheduling benötigt für die Umplanung des *ARF* Benchmarks 30.276 Takte und für den Code der *DCT-DIF* 74.536 Takte. Die Registerallokation benötigt zur Rekonfiguration des Benchmarks *ARF* 25.131 Takte. Eine deutlich höhere Rekonfigurationszeit ist für den Benchmark *DCT-LEE* erforderlich mit 57.812 Takten.

| | ARF | EWF | FFT | DCT-DIT | DCT-DIF | DCT-LEE |
|---|---|---|---|---|---|---|
| Rebinding 1 FU-Ausfall | 2504 | 4391 | 3112 | 3468 | 4397 | 5.109 |
| Renaming 1 Reg-Ausfall | 1.661 | 2.871 | 2.051 | 2.886 | 2.276 | 2.881 |
| Rescheduling 1 FU-Ausfall | 30.276 | 59.036 | 46.559 | 53.698 | 74.536 | 72.468 |
| Registerallokation 1 Reg-Ausfall | 25.131 | 50.179 | 38.356 | 44.289 | 57.216 | 57.812 |

Tabelle 6.7.: Statischer Laufzeitoverhead der Backends für die Rekonfiguration der Mediabenchmarks

Mit der Kenntnis über das Laufzeitverhalten der einzelnen Backends ist die Motivation gegeben, die eingesetzten Backends während der Rekonfiguration zu variieren, so wie es in Algorithmus 4.9 angegeben ist. Denn lediglich die Coderegionen einer Anwen-

dung mit einer hohen Parallelität auf Instruktionsebene erfordern die Berechnung einer neuen Ablaufplanung. Für Codebereiche mit einer geringen Parallelität ist die Ausführung eines Rebindings ausreichend. Wenn davon ausgegangen wird, dass lediglich 10 bis 20% des Anwendungscodes eine sehr hohe Parallelität aufweist, dann ergeben sich durch das Wechseln des Backends deutliche Laufzeiteinsparungen.

In Tabelle 6.8 sind nochmals die Rekonfigurationszeiten der fiktiven Anwendungen aus Tabelle 6.6 betrachtet, allerdings mit der Annahme, dass nur 10%, 20% und 30% des Codes durch das Rescheduling angepasst werden und der Rest durch das Rebinding. Diese Codebereiche sind in der Tabelle als laufzeitkritische Coderegionen bezeichnet. Die letzten beiden Zeilen der Tabelle zeigen als Referenz die Laufzeiten für die Rekonfiguration und die Gesamtlaufzeit für die jeweilige Anwendung, wenn der Code vollständig durch das Rescheduling angepasst wird. Werden lediglich 10% des Codes durch das Rescheduling angepasst, dann verringert sich die Zeit für die Rekonfiguration auf die Laufzeit der Administration. Für die Anwendung *app_f4_b8* erfordert die Rekonfiguration, wenn 10% laufzeitkritischer Code vorhanden sind, 343.921 Takte und entspricht in etwa dem Aufwand für die Administration mit 351.153 Takten. Der ursprüngliche Aufwand von 1.899.682 Takten kann dadurch auf 695.074 Takte gesenkt werden.

Mit der Annahme, dass der laufzeitkritische Code 30% der Anwendung *app_f16_b13* beträgt, ergibt die Zeit für die Rekonfiguration in etwa das doppelte der Zeit für die Administration. Die Gesamtlaufzeit für die lokale Reparatur ist nur die Hälfte im Vergleich zu einer vollständigen Anpassung durch das Rebinding.

| krit. Code | Aufgabenteil | app_f4_b8 | app_f8_b7 | app_f9_b12 | app_f16_b13 |
|---|---|---|---|---|---|
| 10% | Rekonfiguration | 343.921 | 572.840 | 1.279.276 | 2.257.970 |
| | Organisation | 351.153 | 674.008 | 1.366.017 | 2.609.089 |
| | Gesamt | 695.074 | 1.246.848 | 2.645.293 | 4.867.059 |
| 20% | Rekonfiguration | 477.767 | 838.131 | 1.777.509 | 3.193.172 |
| | Organisation | 351.153 | 674.008 | 1.366.017 | 2.609.089 |
| | Gesamt | 828.920 | 1.512.139 | 3.143.526 | 5.802.261 |
| 30% | Rekonfiguration | 611.612 | 1.103.421 | 2.275.743 | 4.128.375 |
| | Organisation | 351.153 | 674.008 | 1.366.017 | 2.609.089 |
| | Gesamt | 962.765 | 1.777.429 | 3.641.760 | 6.737.464 |
| 100% | Rekonfiguration | 1.548.529 | 2.960.453 | 5.763.379 | 10.674.793 |
| | Gesamttakte | 1.899.682 | 3.634.461 | 7.129.396 | 13.283.882 |

Tabelle 6.8.: Entwicklung der Zeiten für die Rekonfiguration bei einem kombinierten Einsatz der Backends

#### Globale Rekonfiguration

Der statische Overhead für Szenarien, bei denen die globale Reparaturstrategie zum Einsatz kommt, beinhaltet einmal die Laufzeiten der lokalen Rekonfiguration und zusätzlich die Laufzeiten für die Organisation und Rekonfiguration bei Einsatz einer Frem-

dreparatur oder dem Task-Rebinding. Für die Untersuchungen werden unterschiedliche Szenarien betrachtet. Das Ausgangssystem setzt sich aus 4 Kernen mit den entsprechenden Programmspeichern und dem Datenspeicher zusammen. Der Performance-Vektor für die Kerne des fehlertoleranten Systems ist mit $((4),(3),(3),(4))$ gegeben. Die in der folgenden Reihenfolge angegeben Tasks *app_f8_b7*,*app_f9_b12*, *app_f16_b13* und *app_f28_b10* sind den Kernen $c_1$ bis $c_4$ zugeordnet. Der Task-Performance-Vektor ist mit $((1,7),(2,2),(2,8),(3,8))$ gegeben. Kern $c_1$ ist als Spare-Kern um 2 Slots erweitert.

In Tabelle 6.9 sind die Ergebnisse für die Untersuchungen der Fremdreparatur angegeben. Eine Fremdreparatur erfolgt, wenn ein Kern den Guard aufgrund eines Fehlers nicht durchläuft. Es wurden für 1 bis 3 defekte Kerne die entsprechenden Laufzeiten der Rekonfiguration ermittelt. Die Rekonfiguration erfordert für einen defekten Kern 140.954 Takte und für drei defekte Kerne 422.686 Takte. Die Gesamtzeit der Rekonfiguration beträgt im Minimum 3.268.481 Takte und maximal 31.064.150 Takte. Der längste Rekonfigurationsvorgang ergab sich für 2 defekte Kerne. Es zeigt sich, dass die Gesamtrekonfigurationszeit im Wesentlichen von der Rekonfiguration der Anwendung selbst abhängt. Weil im zweiten Fall der Kern $c_4$ mit der Task *app_f28_b10* defekt ist, dauert die gesamte Rekonfiguration am längsten, weil die Task die meisten Instruktionen enthält.

| | Anzahl defekter Kerne (Slot 1 jeweils defekt) | | |
|---|---|---|---|
| | 1 Kern ($C_1$) | 2 Kerne ($C_2$ und $C_4$) | 3 Kerne ($C_1$,$C_2$ und $C_3$) |
| Guard (4 Slots) | 204 | 204 | 204 |
| Synchronisation | 103 | 103 | 103 |
| Fremdreparatur | 140.954 | 281.820 | 422.686 |
| Rekonfiguration | 3.127.527 | 30.782.330 | 13.010.002 |
| Gesamt | 3.268.481 | 31.064.150 | 13.432.688 |
| Zeit in s (für 350MHz) | 0,01 | 0,09 | 0,04 |

Tabelle 6.9.: Statischer Overhead für globale Reparaturszenarien, bei denen 1 bis 3 Kerne Defekte im ersten Slot aufweisen

Sollte eine Rekonfiguration einer Anwendung nicht erfolgreich sein, wird das Task-Rebinding als zweite globale Reparaturstrategie eingesetzt. Tabelle 6.10 zeigt die Laufzeiten für das Task-Rebinding, wenn ein oder zwei Kerne ihre zugeordnete Anwendung nicht an die Fehlersituation anpassen konnten. Im ersten Fall wird von einem Defekt im zweiten Slot des zweiten Kerns ausgegangen. Das Task-Rebinding bestimmt eine neue Bindung und vertauscht die Programmspeicher der Kerne $c_1$ und $c_2$. Im zweiten Szenario wird von jeweils einem ausgefallenen Slot in den Kernen $c_3$ und $c_4$ ausgegangen. Das Task-Rebinding führt zur Reparatur zwei Tauschvorgänge durch. Im ersten Durchlauf werden die Programmspeicherinhalte von $c_1$ und $c_4$ vertauscht, wobei die Task *app_f28_b10* rekonfiguriert in den Speicher von $c_1$ geschrieben wird. Im zweiten Durchlauf erfolgt ein Vertauschen von $c_3$ und $c_4$, bei dem die Task *app_f16_b13* di-

rekt rekonfiguriert wird. Abschließend erfolgt eine Rekonfiguration der Task *app_f8_b7* bezüglich des Kerns $c_3$.

Die Zahlen aus Tabelle 6.10 zeigen deutlich, dass die Administration der globalen Reparaturstrategie keinen wesentlichen Einfluss auf die Gesamtlaufzeit hat. Die Gesamtlaufzeit mit 33.547.964 Takten bzw. 56.211.249 Takten ergibt sich maßgeblich aus den Zeiten der Rekonfiguration der jeweiligen Tasks und zusätzlich durch die Laufzeit der abgebrochenen lokalen Rekonfiguration mit der anschließenden Zustandssicherung. Das Task-Rebinding ist so konzeptioniert, dass die Rekonfiguration durch den Master erfolgt. Dadurch werden mehrere Rekonfigurationen nacheinander ausgeführt. Hier besteht Potential für Verbesserungen, falls eine parallele Rekonfiguration durch die Kerne selbst erfolgt.

| | Task-Rebinding für Anzahl an Tasks: | |
|---|---|---|
| | 2 Tasks | 3 Tasks |
| Guard (4 Slots) | 204 | 204 |
| abgebrochene Rekonfig. (Rekonfig. + Sicherung) | 6.923.209 | 19.681.275 |
| Berechnung Task-Rebinding | 561 | 993 |
| Rekonfiguration | 26.623.981 | 36.528.768 |
| Gesamtzeit in Takten | 33.547.964 | 56.211.249 |
| Zeit in s (für 350MHz) | 0,10 | 0,16 |

Tabelle 6.10.: Statischer Overhead für globale Reparaturszenarien, bei denen 1 bis 2 Kerne Defekte aufweisen und 2 bzw. 3 Tasks miteinander vertauscht werden

Im Folgenden werden die Laufzeiten für kombinierte Szenarien betrachtet. Das erste Szenario aus Tabelle 6.11 beschreibt den Fall, dass zunächst eine Fremdreparatur durch den Masterkern erfolgt und sich ein Task-Rebinding anschließt. Der Kern $c_1$ deklariert sich als Master und führt eine Fremdreparatur für $c_2$ durch. Anschließend erfolgt ein Task-Rebinding zwischen den Tasks von $c_1$ und $c_2$. Im zweiten Szenario führen alle Kerne eine lokale Rekonfiguration durch, an die sich ein Task-Rebinding anschließt. Der Kern $c_1$ hat die Anwendung mit den wenigsten Instruktionen und beendet dadurch als Erster die lokale Rekonfiguration und deklariert sich als Master für die globale Rekonfiguration. Anschließend erfolgt ein Task-Rebinding bezüglich der Kerne $c_1, c_2$ und $c_4$. Das dritte und aufwendigste Szenario beginnt mit einer lokalen Reparatur der Kerne $c_1$ und $c_3$. Die Kerne $c_2$ und $c_4$ müssen durch eine Fremdreparatur angepasst werden, nachdem $c_1$ und $c_3$ erfolgreich ihre lokale Reparatur abschließen konnten. Der Kern $c_1$ besitzt die kürzeste Anwendung und deklariert sich als Erster erfolgreich zum Master. Anschließend werden die Fremdreparaturen durchgeführt, die von einem Task-Rebinding der Kerne $c_1, c_2$ und $c_4$ gefolgt werden.

In Tabelle 6.11 sind für manche Szenarien nicht alle Aufgabenteile mit Zahlen hinterlegt. Die fehlenden Zahlen ergeben sich für den Fall, dass entweder der Aufgabenteil

nicht ausgeführt wurde oder durch Zeiten anderer Aufgabenteile maskiert wird, da diese parallel ausgeführt werden. Zum Beispiel sind in den letzten beiden Szenarien die Zeiten für den Guard und die Synchronisation nicht enthalten, weil ein fehlerfreier Kern sich als Master deklariert parallel zur Ausführung der Rekonfiguration. Weil die Rekonfiguration länger dauert, maskiert diese die anderen Zeiten.

| | Szenarien | | |
|---|---|---|---|
| | Fremdrep. und Task-Rebinding | lok. Rekonf. und Task-Rebinding | lok. Rekonf. und Fremdrep. und Task-Rb. |
| lok. Rekonf. | / | / | 7.129.396 |
| Guard | 204 | / | / |
| Synchronisation | 103 | / | / |
| abgebroche Rekonf. | 6.923.209 | 19.681.275 | 19.681.275 |
| Fremdreparatur | 140.954 | / | 281.820 |
| Task-Rebinding | 473 | 885 | 885 |
| Rekonfiguration | 26.623.981 | 30.421.433 | 30.421.433 |
| Gesamtzeit | 33.688.978 | 50.103.778 | 57.515.120 |
| Zeit in s (für 350MHz) | 0,10 | 0,14 | 0,16 |

Tabelle 6.11.: Statischer Overhead für kombinierte Szenarien der lokalen und globalen Rekonfiguration

Die Gesamtreparaturzeiten der kombinierten Szenarien reichen von 0,1 Sekunden Rekonfigurationszeit bis zu 0,16 Sekunden für ein System, das mit 350 MHz getaktet ist. Für Anwendungen mit mehr Instruktionen als die Task *app_f28_b10* können diese Zeiten sich deutlich erhöhen. Die kombinierten Szenarien bestätigen, dass die Zeit für die Systemrekonfiguration im Wesentlichen durch die Zeit für die Rekonfiguration des Anwendungscodes bestimmt wird. Der Fehlschlag einer lokalen Rekonfiguration und die Anzahl der durch ein Task-Rebinding zu rekonfigurierenden Tasks ergeben zusammen die maßgebliche Reparaturzeit für ein System. Ein nächster einflussreicher Faktor ist die Anzahl der Instruktionen aller Tasks, die umgeplant werden müssen.

### 6.2.2. Degradation der Systemleistung

Ein wichtiges Merkmal zur Bewertung der software-basierten Rekonfiguration ist die Degradation der Systemleistung für eine bestimmte Anzahl an Fehlern. Eine degradierte Systemleistung ergibt sich durch eine verlängerte Ablaufplanung für einen Basisblock. Aufgrund der zusätzlichen Instruktionen dauert die Ausführung eines solchen Basisblocks länger, was bereits als dynamischer Overhead eingeführt wurde. Die Backends Rebinding und Renaming erlauben keine Verlängerung der Ablaufplanung, weshalb sie in diesem Abschnitt nicht betrachtet werden. Die Untersuchungen erfolgen für das Rescheduling und die Registerallokation und wurden für die Basisblöcke der

Mediabenchmarks durchgeführt, weil diese eine geringe bis gar keine Anzahl an Nop-Operationen aufweisen. Dadurch können die Backends weniger Hardware-Redundanz nutzen und müssen die Operationen in spätere Operationen planen. Bei den angegebenen Untersuchungsergebnissen handelt es sich jeweils um die worst-case Verlängerung in der Ablaufplanung für einen konkreten Benchmark. Für allgemeinen Programmcode mit einer geringeren Parallelität auf Instruktionswortebenen kann sich eine deutlich geringere Degradation ergeben.

Tabelle 6.12 zeigt die Ergebnisse des Reschedulings. Die Basisblöcke der Mediabenchmarks wurden für 1 und 2 defekte FUs durch das Rescheduling angepasst. Die ursprünglichen Basisblöcke sind für eine Architektur mit 4 Slots erzeugt. Bei einer defekten FU wurde im besten Fall eine Verlängerung der Ablaufplanung von 12,5% für den Benchmark *EWF* bestimmt. Die worst-case Verlängerung ergab sich mit 30% für den Benchmark *DCT-LEE*. Für die Behandlung von zwei defekten FUs wurde eine worst-case Verlängerung von 56,5% für den Benchmark *FFT* berechnet und im besten Fall eine Verlängerung von 15,8% für den Benchmark *EWF*.

| Benchmark | Länge original* | Länge je defekter FU* | | Overhead je defekter FU in % | |
|---|---|---|---|---|---|
| | | 1 | 2 | 1 | 2 |
| ARF | 8 | 10 | 14 | 20,0 | 28,6 |
| EWF | 14 | 16 | 19 | 12,5 | 15,8 |
| FFT | 10 | 14 | 23 | 28,6 | 56,5 |
| DCT-DIF | 11 | 15 | 24 | 26,7 | 54,2 |
| DCT-DIT | 14 | 17 | 24 | 17,6 | 41,7 |
| DCT-LEE | 14 | 20 | 26 | 30,0 | 46,2 |

Tabelle 6.12.: Verlängerung der Ablaufpläne für die Mediabenchmarks bei einer Rekonfiguration durch das Rescheduling (*Angaben entsprechen der Anzahl an Instruktionen)

Die Ablaufplanverlängerungen für 1 bis 3 Registerausfälle bei einer Rekonfiguration durch das Backend der Registerallokation zeigt Tabelle 6.13. Für jeden Benchmark kann ein ausgefallenes Register toleriert werden mit einer Degradation von 6% für den Benchmark *DCT-LEE* und 17,6% für den Benchmark *EWF*. Weitere Registerausfälle können nicht für jeden Benchmark, trotz weiterer Reduktion der Parallelität, toleriert werden. Jedoch können für die Benchmarks *FFT*, *DCT-DIF* und *DCT-LEE* zwei ausgefallene Register toleriert werden, wobei die worst-case Degradation 23% für die *FFT* beträgt. Bis zu drei Registerausfälle können für die Benchmarks *DCT-DIF* und *DCT-LEE* behandelt werden bei einer Degradation von 21,4% bzw. 44%.

Damit die gezeigten Werte für die degradierte Systemleistung in der Praxis eingesetzt werden können, muss die entsprechende Anzahl an zusätzlichen Instruktionsworten durch ein fehlertolerantes System, welches auf der software-basierten Rekonfiguration basiert, bereitgestellt werden. Dazu muss für jeden Basisblock einer Anwendung die angegebene prozentuale Anzahl an zusätzlichen Instruktionen vorhanden sein. Aufgrund der Organisation der präsentierten Rekonfigurationsstrategie können sich die

| Benchmark | Länge original* | Länge je defekt. Register* | | | Overhead je defekt. Register | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 1 | 2 | 3 |
| ARF | 8 | 9 | - | - | 11,1% | - | - |
| EWF | 14 | 17 | - | - | 17,7% | - | - |
| FFT | 10 | 11 | 13 | - | 9,0% | 23,08% | - |
| DCT-DIF | 11 | 12 | 13 | 14 | 8,3% | 15,3% | 21,4% |
| DCT-DIT | 14 | 17 | - | - | 17,7% | - | - |
| DCT-LEE | 14 | 15 | 17 | 25 | 6,7% | 17,7% | 44,00% |

Tabelle 6.13.: Verlängerung der Ablaufpläne für verschiedene Benchmarks bei entsprechend hohen Registerausfällen (*Angaben entsprechen der Anzahl an Instruktionen)

Reserve-Einträge zusammenhängend am Ende des Programmspeichers befinden. Die ursprüngliche Programmspeichergröße muss dann lediglich um den prozentualen Wert erhöht werden.

Ein Overhead von 44% für den gesamten Programmspeicher, wie im Fall der *DCT-LEE* zur Behandlung von drei ausgefallenen Registern, ist verhältnismäßig hoch. Dieser worst-case Aufwand kann verringert werden, wie in Tabelle 6.14 dargestellt. Zu Beginn des Abschnitts wurde bereits erwähnt, dass sich die worst-case Degradation nicht generell für den allgemeinen Programmcode ergeben wird, weil die Instruktionsworte die volle Parallelität des Datenpfades nicht ausnutzen. Tabelle 6.14 zeigt dazu Abschätzungen, wie hoch der Programmspeicher-Overhead ausfallen muss, wenn angenommen wird, dass nur ein bestimmter Bereich des Programmcodes eine worst-case Degradation erzeugt. Die Abschätzungen erfolgen für Anwendungen, die 10%, 20% und 30% laufzeitkritische Coderegionen aufweisen. Für diese Regionen wird eine höhere Degradation angenommen, da die Rekonfiguration durch ein komplexes Backend erfolgt. Für den restlichen Programmcode, also 90%, 80% und 70%, wird eine notwendige Degradation von 10% bzw. 20% angenommen (vgl. Spalte 2 in Tabelle 6.14). Die Ergebnisse in den Spalten 3 bis 5 sind für steigende Fehlerzahlen (1 bis 3) und der entsprechenden worst-case Degradation (18% bis 44%) für den laufzeitkritischen Code ermittelt.

Für die Konstellation, dass ein Register defekt ist und für 10% laufzeitkritischen Programmcode 18% Reserveeinträge bereitzustellen sind, ergibt sich ein gesamter Overhead von 10,8% für den Programmspeicher. Für die Abschätzung, wenn drei Fehler zu behandeln sind, ergeben sich 24,8% Overhead für den Programmspeicher. Dazu zählt, dass für 30% laufzeitkritischen Code 44% Reserveeinträge bereitgestellt werden und für die restlichen 70% des Codes weitere 20% Reserveeinträge. In der Praxis bedeutet dies, dass in einem fehlertoleranten System auf Basis der software-basierten Rekonfiguration, der Programmspeicher um 27,2% vergrößert werden muss, um drei Fehler tolerieren zu können. Die Leistungsdegradation beträgt dann für den laufzeitkritischen Code bis zu 44% und für den regulären Code 20%.

| laufzeit kritischer Code | Overhead regulärer Code | gesamter worst-case Overhead für bis zu | | |
|---|---|---|---|---|
| | | 1 Fehler (18% Overhead) | 2 Fehler (23% Overhead) | 3 Fehler (44% Overhead) |
| 10% | 10% | 10,8 | 11,3 | 13,4 |
| | 20% | 19,8 | 20,3 | 22,4 |
| 20% | 10% | 11,6 | 12,6 | 16,8 |
| | 20% | 19,6 | 20,6 | 24,8 |
| 30% | 10% | 12,4 | 13,9 | 20,2 |
| | 20% | 19,4 | 20,9 | 27,2 |

Tabelle 6.14.: Reduzierter Overhead für verschiedene Kombinationen von laufzeitkritischen und unkritischen Coderegionen

### 6.2.3. Hardware-Overhead für verschiedene Konfigurationen

Der Hardware-Overhead für ein fehlertolerantes System ergibt sich aus dem Overhead für den Programmspeicher, dem Datenspeicher und der Fläche für die redundanten Hardware-Baugruppen. Der Speicheroverhead ermittelt sich aus den Mehrkosten für die Implementierung der Reparaturstrategie, durch die zusätzlichen Programmspeichereinträge zur Behandlung der Degradation und durch eine Vergrößerung des Programmspeichers durch die redundanten Baugruppen. Die Anzahl an Reserve-Slots ergibt sich aus der Anzahl an Defekten, die systemweit tolerierbar sein sollen. Die im Folgendem betrachteten fehlertoleranten Systeme setzen alle die in dieser Arbeit beschriebene software-basierte Reparaturstrategie ein. Dabei ist zu berücksichtigen, dass die Administration der Spare-Slots durch die globale Rekonfiguration erfolgt.

Zur Untersuchung des Overheads werden die in Tabelle 6.15 dargestellten Systeme genauer betrachtet. Tabelle 6.15 gibt gleichzeitig den Hardware-Overhead in Nand2-Flächenäquivalenten für die verschiedenen fehlertoleranten Systemkonfigurationen an. Das erste System hat die Grundkonfiguration $(2, 3, 4)$ mit einer Fläche von 78.285 Nand2-Äquivalenten. Damit das System einen Fehler behandeln kann wird es zur fehlertoleranten Konfiguration $(3, 3, 4)$ erweitert. Der zusäztliche Slot kostet 5,05 % mehr Hardware-fläche. Die Aussage, dass ein Fehler tolerierbar ist, basiert auf der Annahme, dass die globale Reparaturstrategie eingesetzt wird und der Fehler in einer Komponente auftritt, die nicht kritisch ist. Zu den kritischen Komponenten zählen jene, die nicht im Zuge einer software-basierten Rekonfiguration durch eine redundante Komponente ersetzbar sind. Das System mit der Grundkonfiguration $(2, 2, 3, 3, 5)$ kann in der fehlertoleranten Konfiguration $(5, 3, 4, 4, 5)$ bis zu 4 Fehler im System durch den Einsatz der globalen Reparaturstrategie tolerieren. Der erforderliche Overhead an Flächeneinheiten beläuft sich dabei auf 16,31 %. Es sei darauf hingewiesen, dass durch eine lokale Rekonfiguration mit Degradation weitere Fehler behandelt werden können. Die Angaben in Tabelle 6.15 geben eine minimale Abschätzung der tolerierbaren Fehler an.

Die folgenden Tabellen stellen den notwendigen Speicher-Overhead der fehlertoleranten Systemkonfiguration dem Speicherbedarf der Grundkonfiguration gegenüber. Die

| Grund-konfiguration | f.-tolerante Konfig. | behandelbare Fehler | Fläche Grundkonfig. | Fläche f.-tolerant | Overhead |
|---|---|---|---|---|---|
| (2,3,4) | (3,3,4) | 1 Fehler | 78.285 | 82.447 | 5,05 |
| (1,2,3,4) | (4,2,3,4) | 1 Fehler | 94.878 | 107.364 | 11,63 |
| (2,2,3,4) | (4,3,3,4) | 2 Fehler | 99.040 | 111.526 | 11,20 |
| (2,2,3,3,5) | (5,4,4,3,5) | bis 4 Fehler | 128.119 | 153.091 | 16,31 |

Tabelle 6.15.: Angaben der Flächen in Nand2-Äquivalenten und des prozentualen Overheads für Systeme in ihrer Grundkonfiguration und einer fehlertoleranten Konfiguration

Berechnungen des Mehraufwands basieren auf unterschiedlich großen Anwendungen, deren Anzahl an Instruktionen hypothetisch angenommen wird. Für die Betrachtungen wird vereinbart, dass für eine festgelegte Instruktionszahl (zum Beispiel 1.000 Instruktionen) alle Tasks in einem Mehrkernsystem die gleiche Instruktionszahl aufweisen. Für die Untersuchungen werden hypothetische Tasklängen von 1.000 bis 25.000 Instruktionen betrachtet.

Tabelle 6.16 zeigt die Ergebnisse für die fehlertoleranten Konfigurationen $(3, 3, 4)$ und $(4, 2, 3, 4)$ und Tabelle 6.17 für die fehlertoleranten Systemkonfigurationen $(4, 3, 3, 4)$ und $(5, 4, 4, 3, 5)$. In der ersten Spalte ist die jeweilige Instruktionsanzahl angegeben. Die nächsten beiden Spalten geben die minimale Programmspeichergröße für die Systeme der Grundkonfiguration an. Bei dem Wert handelt es sich um die Zusammenfassung aller Programmspeicher des jeweiligen Mehrkernsystems mit einer Operationskodierung von 26 Bit, wie in Kapitel 3.1 in Abbildung 3.2 gezeigt.

Die Berechnung des Programmspeicher-Overheads $mem_{Overhead}$ für eine gegebene Taskgröße (Anzahl an Instruktionen) und eine fehlertolerante Konfiguration setzt sich aus drei Teilen zusammen:

$$mem_{Overhead} = mem_{static} + mem_{degredation} + mem_{spare}$$

Der Wert für $mem_{static}$ ergibt sich aus der erforderlichen Anzahl an Instruktionen für die software-basierte Reparaturstrategie, die sich in jedem Programmspeicher des fehlertoleranten Systems befindet (vgl. Tabelle 6.2 und 6.3). Der Wert für $mem_{degredation}$ errechnet sich aus der prozentualen Verlängerung der Programmspeicher zur Behandlung der Degradation durch die software-basierte Reparatur. Die prozentuale Verlängerung wird für den worst-case angenommen (vgl. Tabelle 6.14 mit dem Wert 27,2%) und auf 30% gesetzt. Die zusätzlichen Spare-Slots der fehlertoleranten Konfiguration erfordern mehr Platz für die Instruktionskodierung im Programmspeicher. Dieser Overhead fließt durch den Wert $mem_{spare}$ in die Berechnung ein.

Tabelle 6.16 zeigt, dass sich ein relativer Overhead von 413,1% bzw. 628,1% für die beiden fehlertoleranten Systeme ergibt, wenn die Länge aller Tasks 1.000 Instruktionen beträgt. Der hohe Overhead ergibt sich, weil die Anzahl an Instruktionen für die software-basierte Rekonfiguration 4.147 Instruktionen beträgt. Der Code der Reparatur muss auf jedem Kern vorhanden sein und dominiert dadurch den Programmspeicher,

der für jede Tasks nur 1.000 Instruktionen beträgt. Mit größeren Anwendungen verschiebt sich dieses Verhältnis. Für Anwendungen mit 5.000 Instruktionen ergibt sich ein Overhead von 118,2% bzw. 196,8% und für Anwendungen mit 25.000 Instruktionen wird im besten Fall ein Overhead von 59,2 % erreicht.

| Instruktionen | Programmspeicher d. Grundkonfg. in kByte | | Overhead der fehlertoleranten Konfig. | | | |
|---|---|---|---|---|---|---|
| | | | (4,3,4) | | (4,2,3,4) | |
| | (2,3,4) | (1,2,3,4) | in kB | relativ | in kB | relativ |
| 1.000 | 28,6 | 31,7 | 118,0 | 413,1 % | 199,4 | 628,1 % |
| 2.000 | 57,1 | 63,5 | 130,7 | 228,8 % | 227,6 | 358,6 % |
| 5.000 | 142,8 | 158,7 | 168,8 | 118,2 % | 312,3 | 196,8 % |
| 10.000 | 285,6 | 317,4 | 232,2 | 81,3 % | 453,6 | 142,9 % |
| 25.000 | 714,1 | 793 | 422,7 | 59,2 % | 877,3 | 110,6 % |

Tabelle 6.16.: Overhead für den Programmspeicher der fehlertoleranten Konfigurationen $(3, 3, 4)$ und $(4, 2, 3, 4)$ in kByte und relativ zur Grundkonfiguration

Tabelle 6.17 bestätigt die vorherigen Angaben, dass mit steigender Instruktionsanzahl in den eigentlichen Anwendungen der Overhead für die software-basierte Rekonfiguration ein günstigeres Verhältnis erreicht. Für Anwendungen mit 10.000 bzw. 25.000 Instruktionen bewegt sich der Programmspeicher-Overhead für die fehlertoleranten Systeme in einem Bereich von 86,6 und 140,1 % Mehraufwand.

Beide Tabellen reflektieren nicht den Mehraufwand für den Datenspeicher. Für die vorgenommene Implementierung wurde angegeben, dass 9,72 kByte an Datenspeicher erforderlich sind. Wie bereits für den Programmspeicher gezeigt wurde, kann sich für kleine Speichergrößen ein erheblicher Mehraufwand ergeben. Für größere Speicheranforderungen bezüglich der Grundkonfiguration sind bessere Verhältnisse möglich. Darüber hinaus handelt es sich bei den 9,72 kByte im Wesentlichen um dynamische Daten, die im Vergleich zum Programmspeicher nicht exklusiv der Reparatur zur Verfügung stehen müssen. Dadurch können die 9,72 kByte Datenspeicher sich mit der Nutzung durch die eigentliche Anwendung überschneiden. Erfordern die regulären Anwendungen sogar mehr als die 9,72 kByte Datenspeicher, entfällt der Overhead für den Datenspeicher.

## 6.3. Simulationsergebnisse

Bisher wurden die Kosten diskutiert, die der Einsatz einer software-basierten Reparaturstrategie verursacht. Im Folgenden soll der Nutzen der entwickelten Rekonfigurationsstrategie in Form einer gesteigerten Lebensdauer und der durchschnittlich tolerierbaren Anzahl an Fehlern gezeigt werden. Aufgrund der Komplexität der software-basierten Rekonfiguration erfolgt der Nachweis durch eine modellbasierte Fehlersimulation anstelle einer exakten analytischen Berechnung.

Das für die Fehlersimulation eingesetzte Systemmodell ist anhand der Komponenten der VHDL-Implementierung beschrieben. Den einzelnen Komponenten des Modells wird eine Anzahl an Flächenäquivalenten entsprechend den Ergebnissen aus der Syn-

| Instruktionen | Programmspeicher d. Grundkonfg. in kByte | | Overhead der fehlertoleranten Konfig. | | | |
|---|---|---|---|---|---|---|
| | | | (4,3,3,4) | | (5,3,4,4,5) | |
| | (2,2,3,4) | (2,2,3,3,5) | in kB | relativ | in kB | relativ |
| 1.000 | 34,9 | 47,6 | 207,1 | 593,3 % | 315,4 | 662,6 % |
| 2.000 | 69,8 | 95,2 | 230,0 | 329,4 % | 354,5 | 372,3 % |
| 5.000 | 174,6 | 238,0 | 298,5 | 171,0 % | 471,6 | 198,1 % |
| 10.000 | 349,1 | 476,1 | 412,8 | 118,2 % | 666,8 | 140,1 % |
| 25.000 | 872,8 | 1.190,2 | 755,6 | 86,6 % | 1.252,4 | 105,2 % |

Tabelle 6.17.: Overhead für den Programmspeicher der fehlertoleranten Konfigurationen $(4, 3, 3, 4)$ und $(5, 3, 4, 4, 5)$ in kByte und relativ zur jeweiligen Grundkonfiguration

these (vgl. Tabelle 6.1) zugeordnet. Zwischen den Flächenäquivalenten herrscht eine Gleichverteilung bezüglich der Wahrscheinlichkeit, dass sie durch einen Fehler betroffen sind. Somit hat eine Komponente mit einer Fläche von 3000 im Vergleich zu einer Komponente mit 300 Flächeneinheiten die 10-fache Wahrscheinlichkeit, von einem Fehler betroffen zu sein. In jedem Simulationsschritt wird zufällig bestimmt, ob ein Fehler vorliegt. Der dazu zufällig ermittelte Wert stammt aus dem Bereich $\{0...10^6\}$, was einer allgemeinen Fehlerrate von einem Fehler in $10^6$ Betriebsstunden entspricht. Anschließend wird geprüft, ob dieser Wert in der Fläche einer Komponente liegt. Am Ende eines Simulationsschrittes wird geprüft, ob das System weiterhin intakt ist. Handelt es sich um ein nicht-fehlertolerantes System, führt ein Fehler zum Ausfall des Systems. Für die fehlertoleranten Systeme werden zwei Reparaturstrategien betrachtet. Der erste Systemtyp setzt lediglich die lokalen software-basierten Reparaturverfahren ein. Der zweite Typ kann zusätzlich die globale Rekonfiguration anwenden. Anhand der Systemkonfiguration und der verwendeten Reparaturstrategie wird ermittelt, ob das System den im aktuellen Schritt injizierten Fehler tolerieren kann. Die Simulation erfolgt für eine Million Systeme und es wird jedes System bis zum Ausfall simuliert.

Die Untersuchungen erfolgen für die bereits zuvor präsentierten Systemkonfigurationen (vlg. Tabelle 6.15) und sind in Tabelle 6.18 mit der Konfiguration für die lokale Reparaturstrategie angegeben. Die Berechnung zur Tolerierbarkeit eines oder mehrerer Fehler basiert auf den Angaben zur Systemdegradation für bestimmte Fehlerkombinationen nach Tabelle 6.14. Für alle Systeme wird angenommen, dass 30% kritischer Code vorhanden ist und eine degradierte Systemleistung von bis zu 30% akzeptiert wird. Basierend auf den Ergebnissen von Tabelle 6.13 und Tabelle 6.12 können damit 2 ausgefallene Register bzw. ein ausgefallener Slot behandelt werden. Für die globale Reparaturstrategie werden die Task-Performancevektoren $((1, 7), (2, 2), (3, 8))$, $((1, 7), (2, 2), (2, 8), (3, 8))$ und $((1, 7), (2, 2), (2, 8), (3, 8), (4, 6))$ verwendet. Ist eine kritische Komponente von einem Fehler betroffen, wird das entsprechende System als defekt betrachtet.

Die Fehlersimulation erfolgt weiterhin unter den folgenden Annahmen:

- Alle Komponenten, die durch einen injizierten Fehler betroffen sind, können vom Test gefunden werden.
- Die Fehler sind gleich verteilt bezüglich der Gatter und deren Flächenäquivalenten.
- Sind mehrere Flächenäquivalente einer Komponente durch einen Fehler betroffen, wird dies als ein gemeinsamer Fehler gewertet, der durch die Rekonfiguration tolerierbar ist.
- Der Programmspeicher, der Datenspeicher, der Overhead der Speicher und Fehler im Speicher sind nicht von der Simulation berücksichtigt in Ermangelung an repräsentativen Daten bezüglich der Fehlerwahrscheinlichkeiten und der Flächenverhältnisse des Speichers im Vergleich zu den Strukturen innerhalb eines Prozessors.

| | nicht-fehlertolernt. Systemkonfig. | fehlertolerante Konfig. (lokale Stratg.) | fehlertolerante Konfig. (globale Stratg.) |
|---|---|---|---|
| Szenario 1 | (2,3,4) | (3,4,5) | (4,3,4) |
| Szenario 2 | (1,2,3,4) | (2,3,4,5) | (4,2,3,4) |
| Szenario 3 | (2,2,3,4) | (3,3,4,5) | (4,3,3,4) |
| Szenario 4 | (2,2,3,3,5) | (3,3,4,4,6) | (5,4,4,3,5) |

Tabelle 6.18.: Eigenschaften der untersuchten Systeme für die Fehlersimulation

#### Entwicklung der Systemausfälle über die Zeit

Im Folgenden soll das Grundsystem $(2, 2, 3, 4)$ ausführlich anhand der Ergebnisse der Fehlersimulation diskutiert werden, um den Nutzen der entwickelten Reparaturstrategie darzulegen. Die Untersuchungen der restlichen Konfiguration bestätigen die Verbesserungen durch die entwickelte Reparaturstrategie. Da sich keine signifikanten Unterschiede zwischen den Ergebnissen der Systeme ergeben, befindet sich die Auswertung der Fehlersimulationen für die restlichen Konfigurationen im Anhang A.

Abbildung 6.1 stellt die Entwicklung der Systemausfälle über die Zeit dar. Die Kurven sind für drei Systeme angegeben. Das erste System *non-tolerant System* entspricht der Grundkonfiguration $(2, 2, 3, 4)$. Die Grundkonfiguration wird für das erste fehlertolerante System *ftSystemLokal* zu $(3, 3, 4, 5)$ erweitert. Das System setzt die lokale software-basierte Rekonfiguration zur Fehlerbehandlung ein. Das zweite fehlertolerante System *ftSystemGlobal* wird zur Konfiguration $(4, 3, 3, 4)$ erweitert und setzt zusätzlich die globale Reparaturstrategie ein. Auf der y-Achse wird die prozentuale Anzahl an lebendigen Systemen zu einem Zeitpunkt $x$ aufgetragen. Die Skalierung der x-Achse entspricht $10^6$ Stunden.

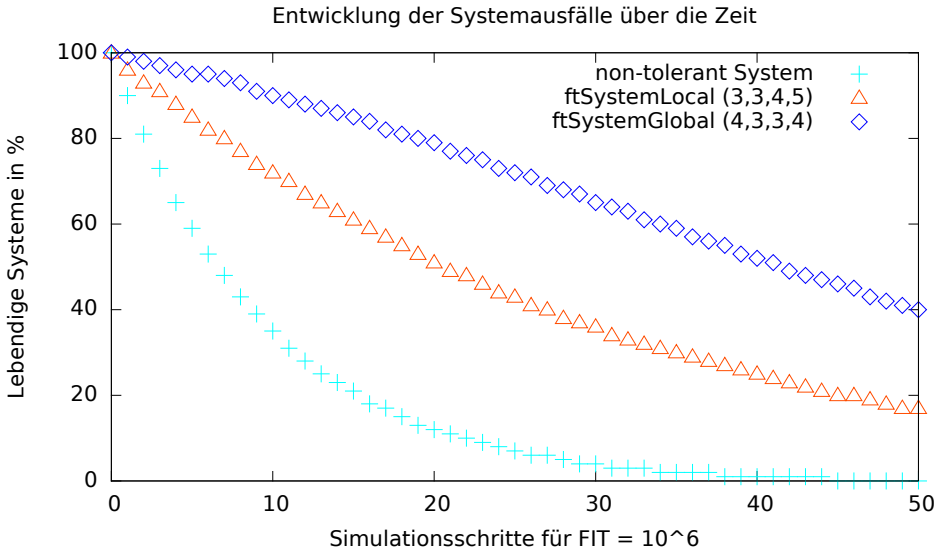


Abbildung 6.1.: Entwicklung der mittleren Lebenszeit der fehlertoleranten Systeme *ftSystemLokal* und *ftSystemGlobal* sowie der nicht-fehlertoleranten Grundkonfiguration $(2, 2, 3, 4)$

Die Fehlersimulation ergab, dass die mittlere Lebensdauer für das System *ftSystemLokal* um den Faktor $2,6$ höher liegt im Vergleich zum nicht-fehlertoleranten System. Das System mit der globalen Reparaturstrategie hat im Vergleich zum System mit der lokalen Reparaturstrategie eine $1,6$-fach höhere mittlere Lebensdauer. In Abbildung 6.1 ist zu sehen, dass die prozentuale Anzahl an lebendigen Systemen für das nicht-fehlertolerante System stark abfällt im Vergleich zu den Kurven der fehlertoleranten Systeme. Im Vergleich der Kurven der Systeme *ftSystemLokal* und *ftSystemGlobal* zeigt das System mit der globalen Reparaturstrategie im Anfangsbereich, in dem wenige Fehler auftreten, einen flacheren Abfall als das System mit der lokalen Reparatur. Das begründet sich damit, dass der Ausfall einer Komponente, die für die lokale Reparatur eines Kerns erforderlich ist, das gesamte System ausfallen lässt.

Tabelle 6.19 stellt die Ergebnisse für die Verbesserung in der Lebenszeit dar, wenn die prozentuale Anzahl an lebendigen Systemen fixiert ist. Der jeweilige prozentuale Wert an lebendigen Systemen ist in der ersten Spalte angegeben. Die zweite Spalte zeigt die jeweilige Verbesserung der Lebenszeit des fehlertoleranten Systems $ftSystemLokal$ gegenüber dem nicht-fehlertoleranten Grundsystem. Die letzte Spalte setzt die Lebenszeiten der beiden fehlertoleranten Systeme ins Verhältnis und stellt die Verbesserung für das System $ftSystemGlobal$ dar.

Die erste Zeile der Tabelle zeigt, dass durch den Einsatz der globalen Strategie eine Verdreifachung der Lebenszeit erreicht wird für den Fall, dass 98% der Systeme lebendig sind. Die lokale Strategie bringt im Vergleich zum nicht-fehlertoleranten System keine Verbesserung. Erst wenn 95% der Systeme lebendig sein sollen, erreicht die lokale Strategie eine Verbesserung um den Faktor 2 im Vergleich zum nicht-fehlertoleranten System. Das System *ftSystemGlobal* erreicht nochmals eine Steigerung um den Faktor 3

gegenüber dem System *ftSystemLokal*. Die Werte in der Spalte zur lokalen Strategie zeigen, dass sich der Einsatz der lokalen Strategie allgemein lohnt, um eine Verbesserung in der Lebenszeit zu erreichen. Die Werte in der Spalte für die globale Rekonfiguration fallen in etwa gleichmäßig ab. Während für 98% an lebendigen Systemen ein Faktor von 3 erreicht wird, verringert sich dieser für 50% lebendige System auf den Wert 2. Unter Berücksichtigung der Ergebnisse aus Abbildung 6.1 kann geschlussfolgert werden, dass die globale Reparaturstrategie die lokale dahingehend verbessert, dass weniger Einzelfehler zum sofortigen Ausfall führen.

| lebendige | Verbesserung um Faktor | |
|---|---|---|
| Systeme in % | lokale Strategie | globale Strategie |
| 98 | 1 | 3,0 |
| 95 | 2,0 | 3,5 |
| 92 | 3,0 | 3,0 |
| 90 | 2,0 | 2,8 |
| 85 | 3,0 | 2,7 |
| 80 | 2,7 | 2,5 |
| 70 | 3,0 | 2,3 |
| 60 | 3,2 | 2,2 |
| 50 | 3,0 | 2,0 |

Tabelle 6.19.: Verbesserung der Lebenszeit für die lokale und globale Reparaturstrategie, wenn die prozentuale Anzahl an lebendigen Systemen fixiert ist

#### Tolerierbare Fehler

Die Fehlersimulation des Systems *ftSystemLokal* ergab, dass durchschnittlich 3 Fehler tolerierbar sind. Das fehlertolerante System mit der Konfiguration $(4, 3, 3, 4)$ kann durch den Einsatz der globalen Reparaturstrategie im Durchschnitt 5 Fehler im gesamten System tolerieren. Tabelle 6.20 zeigt ausführlicher den prozentualen Anteil an Systemen, die eine bestimmte Anzahl an Fehlern tolerieren. Die Werte sind für die beiden Reparaturstrategien lokal und global gegenüber gestellt. Einen Fehler konnten 73,6 % der Systeme mit lokaler Strategie tolerieren, wohingegen dies für 93,4 % der Systeme mit globaler Strategie möglich ist. Drei Fehler können von 36,1 % der Systeme *ftSystemLokal* toleriert werden und durch 70,9 % der Systeme *ftSystemGlobal*. Für fünf Fehler fällt der Wert auf 16,5 % bzw. 43,1 % für die Systeme mit globaler Reparaturstrategie.

Die teilweise recht hohen Werte an tolerierbaren Fehlern werden nur in wenigen Fällen erreicht und erfordern eine bestimmte Fehlerkonstellation. Wenn ein System den Ausfall eines oder auch mehrerer Slots kompensieren kann, werden automatisch auch alle weiteren Komponentenausfälle des gleichen Slots kompensiert.

| Fehler | überlebende Systeme in % | |
|---|---|---|
| | lokale Strategie | globale Strategie |
| 1 | 73,6 | 93,4 |
| 2 | 51,8 | 82,9 |
| 3 | 36,1 | 70,9 |
| 4 | 24,7 | 57,1 |
| 5 | 16,5 | 43,1 |
| 6 | 10,7 | 30,5 |
| 7 | 6,8 | 20,2 |
| 8 | 4,1 | 12,5 |
| 9 | 2,4 | 7,3 |
| 10 | 1,4 | 4,0 |

Tabelle 6.20.: Prozentuale Angabe der überlebenden Systeme bei einer entsprechenden Anzahl an Fehlern, für die Systeme mit einer lokalen bzw. globalen Reparaturstrategie

# 7. Zusammenfassung und Ausblick

In dieser Arbeit wurde eine software-basierte Reparaturstrategie zur Behandlung permanenter Fehler in statisch geplanten Mehrkernsystemen entwickelt. Die Prozessorkerne des Systems basieren auf einer VLIW-Architektur mit der Eigenschaft, dass die parallele Ausführung und Ressourcennutzung explizit im Instruktionswort ausgedrückt ist. Die Kompensation einer defekten Prozessorbaugruppe erfolgt durch eine Rekonfiguration der Instruktionen im Programmspeicher, so dass diese die Verwendung der fehlerhaften Komponente vermeiden. Dazu werden die Operationen innerhalb einer Instruktion von der defekten Baugruppe auf eine redundante Baugruppe verschoben oder zu einem späteren Ausführungszeitpunkt geplant. Durch die Planung einer Operation in einen späteren Zeitpunkt kann sich eine degradierte Systemleistung ergeben. Die Berechnungen zur Rekonfiguration erfolgen durch ein Programm, das durch den defekten Kern selbst oder in der erweiterten Reparaturstrategie durch einen anderen Kern des Systems ausgeführt wird.

Die software-basierte Rekonfiguration wurde in eine lokale und globale Rekonfiguration unterschieden. Die lokale Rekonfiguration kompensiert die defekten Baugruppen für einen Prozessorkern. Für die erforderlichen Anpassungen wurden vier Backends entwickelt, mit denen verschiedene Anpassungen erfolgen können. Das Rebinding kann die Bindung von Operationen innerhalb einer Instruktion anpassen und dadurch defekte Baugruppen der Prozessorslots und des Bypasses kompensieren. Das Rescheduling erzeugt eine neue Ablaufplanung für die Operationen auf der Granularität von Basisblöcken und kann Operationen spätere Ausführungszeitpunkte zuweisen. Zur Behandlung von defekten Registern kann das Renaming oder die Registerallokation eingesetzt werden. Das Renaming ersetzt ein defektes Register durch ein Reserve-Register und bildet die Grundlage der Registerallokation. Die Registerallokation erzeugt für einen Basisblock eine neue Registervergabe. Die verfügbaren Register werden dazu statisch in zwei Gruppen unterteilt. Die Register der einen Gruppe werden ausschließlich für globale Werte verwendet. Die zweite Gruppe wird analog dazu nur für lokale Werte genutzt. Ein defektes globales Register wird durch ein lokales Register ersetzt mit Hilfe des Renaming. Das verwendete lokale Register wird durch eine Neuvergabe der Registerallokation kompensiert.

Durch die Planung von Operationen zu einem späteren als den ursprünglichen Ausführungszeitpunkt können Verlängerungen in einem neuen Ablaufplan entstehen. Durch die Verschiebungen der Instruktionen im Programmspeicher mussten die folgenden Probleme behandelt werden:

- ein Überschreiben des ungeplanten Codes durch rekonfigurierte Basisblöcke und
- ungültige Sprungoperationen aufgrund geänderter Adressbezüge.

Zur Vermeidung von Überschreibungen wird der Programmcode einer Anwendung an das Ende des Programmspeichers verschoben. Die Rekonfiguration des Programmcodes erfolgt auf der Granularität von Basisblöcken. Zur Rekonstruktion der Basisblockinformationen wird die Anwendung zunächst in Funktionen zerlegt, die dann bezüglich der vorhandenen Basisblöcke untersucht werden. Nachdem ein Basisblock rekonfiguriert wurde, erfolgt die Behandlung ungültiger Sprungoperationen. Sprungoperationen, die nicht direkt mit einer gültigen Adresse aktualisiert werden können, werden sich vom Verfahren gemerkt und dann angepasst, wenn die neue Adresse bekannt ist.

Die globale Rekonfiguration erweitert die lokale Reparaturstrategie und behandelt Situationen, in denen diese fehlschlägt. Es gibt zwei Fälle, in denen eine lokale Rekonfiguration nicht erfolgreich durchführbar ist:

1. falls der Programmcode der Reparaturalgorithmen die Verwendung defekter Komponenten vorsieht und

2. die neu berechnete Ablaufplanung Ressourcenbeschränkungen verletzt oder der Reservespeicher nicht ausreicht für die Aufnahme des rekonfigurierten Codes.

Im ersten Fall werden die Reparaturalgorithmen des defekten Kerns angepasst und die lokale Reparatur wiederholt ausgeführt. Die Anpassungen werden durch einen anderen Kern des Systems vorgenommen. Diese Form der globalen Reparatur wurde als Fremdreparatur bezeichnet. Die Behandlung des zweiten Falls erfolgt auch durch einen anderen Kern des Systems. Kann ein defekter Kern seine Anwendung nicht erfolgreich rekonfigurieren, wird diese Anwendung mit einer anderen Anwendung des Systems getauscht. Das Ziel ist es, die zuvor nicht rekonfigurierbare Anwendung an die Architekturparameter des neu zugewiesenen Kerns anpassen zu können. Diese Methode der globalen Reparatur wurde als Task-Rebinding bezeichnet. Die Administration der globalen Reparatur erfolgt zentral gesteuert durch einen beliebigen Kern des Systems, der zur Reparaturzeit bestimmt wird.

Durch den Einsatz des Task-Rebindings in einem fehlertoleranten Mehrkernsystem entsteht die Möglichkeit, den Overhead von Spare-Komponenten zu reduzieren im Vergleich zu einem System, basierend auf einer lokalen Reparaturstrategie. Für den Einsatz einer lokalen Reparatur müssen in jeden Kern redundante Baugruppen integriert werden. Durch das Task-Rebinding können redundante Baugruppen kernübergreifend zur Rekonfiguration einer Anwendung genutzt werden. Zum Beispiel kann eine Anwendung, die nicht erfolgreich rekonfiguriert werden konnte, auf einen Kern mit zusätzlichen redundanten Baugruppen verschoben werden. Durch ein geschicktes Verteilen der redundanten Komponenten kann sich der systemweite Overhead verringern, wenn sich die Kerne des Systems in ihrer Konfiguration unterscheiden. Für Systeme mit homogenen Kernen können keine Verbesserungen erzielt werden.

Das vorgestellte Systemmodell wurde in zwei Versionen in VHDL implementiert. Die beiden Implementierungen unterscheiden sich im Hinblick auf die Unterstützung der software-basierten Rekonfiguration. Durch den Vergleich der Synthese-Ergebnisse beider Systeme kann der Hardware-Overhead angegeben werden, der für den Einsatz der in dieser Arbeit entwickelten Techniken zur Fehlertoleranz erforderlich ist. Die Untersuchungen für ein Mehrkernsystem mit vier Kernen, 64 Registern je Kern und vier

Ausführungsslots ergaben, dass ein Hardware-Overhead von 3,6% entsteht. Die Softwareimplementierung der Reparaturalgorithmen erfolgte in Assembler und erfordert 52,66 kByte Programm- und 9,72 kByte Datenspeicher.

Die entwickelten Reparaturalgorithmen wurden hinsichtlich des statischen und des dynamischen Laufzeitoverheads untersucht. Zur Bestimmung des statischen Laufzeitoverheads wurden verschieden große Pseudoanwendungen untersucht. Die Ausführungszeiten der lokalen Rekonfiguration reichen von 1.899.682 Takten (0,0054s bei 350MHz) bis zu 18.546.548 Takten (0,0530s). Es wurde gezeigt, dass der wesentliche Zeitaufwand durch ein Backend entsteht, welches eine Ablaufplanung für den Programmcode der Anwendung vornimmt. Die globale Reparaturstrategie wurde für mehrere Reparaturszenarien untersucht. Die Szenarien reichten von der Durchführung einer einfachen Fremdreparatur bis hin zu Szenarien, welche die Fremdreparatur und ein mehrfaches Task-Rebinding kombiniert vornehmen. Im Szenario mit Fremdreparatur und anschließender Rekonfiguration ergab sich eine Reparaturzeit von 3.268.481 Takten bzw. 0,01s für ein System, das mit 350MHz getaktet ist. Das kombinierte Szenario aus Fremdreparatur und mehrfachem Task-Rebinding erforderte eine Reparaturzeit von 57.515.120 Takten bzw. 0,16s für die Reparatur während des Systemstarts.

Die Ergebnisse des dynamischen Laufzeitoverheads wurden für typische Media-Benchmarks ermittelt. Dazu wurden die Verlängerungen in der Ablaufplanung bei unterschiedlichen Komponentenausfällen betrachtet. Die Untersuchungen erfolgten auf der Basis eines Kerns mit vier Slots. Je nach Benchmark ergab sich für das Rescheduling eine degradierte Systemleistung von 12,5% bis zu 30% bei einer defekten FU und 15,8% bis zu 56% bei zwei ausgefallenen FUs. Mit dem Einsatz des Verfahrens zur Register-Allokation konnte der Ausfall eines Registers mit 6,7% bis zu 17,7% degradierter Systemleistung behandelt werden. Weitere Registerausfälle konnten nur für wenige Benchmarks toleriert werden mit einer worst-case Degradation von 23% für zwei Registerausfälle und 44% für drei ausgefallene Register.

Zur Behandlung einer degradierten Systemleistung muss der Programmspeicher vergrößert werden. Die redundante Auslegung eines Kerns mit Reserve-Slots wirkt sich zusätzlich auf die Größe das Programmspeichers aus. Der erforderliche Overhead für den Programmspeicher wurde für verschiedene Systemkonfigurationen ermittelt. Für kleine Anwendungen mit 1.000 Instruktionen ergeben sich für den Overhead hohe Werte, die in der Praxis nicht anwendbar sind. Erst für Systeme mit 10.000 bzw. 25.000 Instruktionen verringert sich der Overhead für den Programmspeicher auf Werte von 86% bis 142%.

Für den Nachweis einer gesteigerten mittleren Lebenszeit wurden Fehlersimulationen für verschiedene Systemkonfigurationen mit unterschiedlichen Reparaturstrategien durchgeführt. Ein fehlertolerantes System mit lokaler Reparaturstrategie erreicht gegenüber einem nicht-fehlertoleranten System eine Verbesserung in der mittleren Lebenszeit um den Faktor 2,6. Eine weitere Verbesserung wird durch ein fehlertolerantes System mit globaler Reparaturstrategie erreicht. Ein solches System kann gegenüber einem System mit lokaler Reparaturstrategie eine Steigerung um den Faktor 1,6 erzielen. Eine deutliche Verbesserung in der tolerierbaren Anzahl an Fehlern konnte von den Systemen mit globaler Reparaturstrategie gegenüber den Systemen mit lokaler Strategie

erreicht werden. Einen Fehler können 73% der Systeme mit lokaler Strategie tolerieren, wohingegen dies 93,4% der Systeme mit globaler Strategie erreichen. Drei Fehler können lediglich in 36% der Fälle durch die lokale Strategie repariert werden. Mit dem Einsatz der globalen Reparaturstrategie gelingt dies in 70,9% der Fälle.

Basierend auf den Ergebnissen kann geschlussfolgert werden, dass sich der Einsatz der entwickelten Reparaturstrategie zur Steigerung der mittleren Lebenszeit und der Anzahl an tolerierbaren Fehlern in statisch geplanten Mehrkernsystemen eignet. Im Fehlerfall muss eine degradierte Systemleistung von bis zu 44% bei drei ausgefallenen Registern bzw. 56% bei zwei ausgefallenen Slots akzeptiert werden. Der Einsatz der in dieser Dissertation entwickelten Techniken zur Fehlertoleranz ist bezüglich des Programmspeicheroverheads erst ab Anwendungsgrößen mit 10.000 Instruktionen praktikabel. Die Verzögerungen beim Systemstart fallen mit 0,0054s bis 0,053s für die lokale Rekonfiguration moderat aus. Für umfangreiche globale Rekonfigurationen kann die Verzögerung des Systemstarts bis zu 0,16s betragen.

In den Untersuchungen der mittleren Lebenszeit ist der Programmspeicher-Overhead nicht berücksichtigt in Ermangelung einer Synthesemöglichkeit für Speicherstrukturen. Weiterhin sind die Ergebnisse für das Backend Register-Allokation unzufriedenstellend, das lediglich für wenige Benchmarks mehr als einen Registerausfall behandeln konnte. Eine Möglichkeit für weiterführende Untersuchungen ist, den Algorithmus Abwurfcode erzeugen zu lassen, wenn keine freien Register zur Allokation verfügbar sind. Eine interessante Fragestellung ist dann, wie hoch eine degradierte Systemleistung ausfällt im Vergleich zur Reduktion der parallel geplanten Operation. Weiterhin muss spezifiziert werden, an welche Stellen im Speicher ausgelagerte Registerinhalte zwischengespeichert werden. Andere weiterreichende Untersuchungen betreffen das Systemmodell mit einer integrierten Interruptverarbeitung und die Konsequenzen für die software-basierte Rekonfiguration. Ein nächster Punkt, der Ansatz für Erweiterungen bietet ist, dass mehrere Tasks je Kern zugelassen werden, die untereinander kommunizieren. Dies hat Konsequenzen für das Task-Rebinding und erschwert unter Umständen das Vertauschen von Programmspeicherinhalten.

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Literaturverzeichnis


[1] Abella, J., X. Vera, O.S. Unsal, O. Ergin, A. Gonzalez und J.W. Tschanz: *Refueling: Preventing Wire Degradation due to Electromigration.* IEEE Micro, 28(6):37–46, 2008.

[2] Aggarwal, Nidhi, Parthasarathy Ranganathan, Norman P. Jouppi und James E. Smith: *Configurable Isolation: Building High Availability Systems with Commodity Multi-Core Processors.* In: *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*, Seiten 470–481, 2007.

[3] Aho, Alfred V., Ravi Sethi und Jeffrey D. Ullman: *Compilers principles, techniques, and tools.* Addison-Wesley, Upper Saddle, Prentice Hall, 2003.

[4] Alam, M. und S. Mahapatra: *A Comprehensive Model of PMOS NBTI Degradation.* Microelectronics Reliability, 45(1):71–81, 2005.

[5] Bahar, R. I., M. B. Tahoori, S. K. Shukla und F. Lombardi: *Challenges for Reliable Design at the Nanoscale.* IEEE Design and Test of Computers, 22(4):295 – 297, 2005.

[6] Bashir, M. und L. Milor: *Modeling Low-k Dielectric Breakdown to Determine Lifetime Requirements.* IEEE Design and Test of Computers, 26(6):18 –27, 2009.

[7] Benso, A., S. Di Carlo, G. Di Natale und P. Prinetto: *Online self-repair of FIR filters.* IEEE Design and Test of Computers, 20(3):50 – 57, 2003.

[8] Benso, Alfredo, Silvia Chiusano und Paolo Prinetto: *A Self-Repairing Execution Unit for Microprogrammed Processors.* IEEE Micro, 21(5):16–22, 2001.

[9] Benso, Alfredo, Silvia Chiusano, Paolo Prinetto, P. Simonotti und G. Ugo: *Self-Repairing in a Micro-Programmed Processor for Dependable Applications.* In: *Proceedings of the 15th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '00)*, Seiten 231–239, 2000.

[10] Bolchini, C. und F. Salice: *A Software Methodology for Detecting Hardware Faults in VLIW Data Paths.* In: *Proceedings of the 16th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '01)*, Seiten 170–175, 2001.

[11] Borkar, S.: *Designing Reliable Systems From Unreliable Components: The Challenges of Transistor Variability and Degradation.* IEEE Micro, 25(6):10–16, 2005.

[12] Borkar, Shekhar und Andrew A. Chien: *The Future of Microprocessors.* Communications of the ACM, 54(5):67–77, 2011.

[13] Bower, F.A., D.J. Sorin und S. Ozev: *A Mechanism for Online Diagnosis of Hard Faults in Microprocessors.* In: *Proceedings of the 38th Annual IEEE/ ACM International Symposium on Microarchitecture (MICRO-38)*, Seiten 210–223, 2005.

[14] Bower, Fred A., Paul G. Shealy, Sule Ozev und Daniel J. Sorin: *Tolerating Hard Faults in Microprocessor Array Structures.* In: *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN '04)*, Seiten 51–61, 2004.

[15] Breuer, Melvin A., Sandeep K. Gupta und T.M. Mak: *Defect and Error Tolerance in the Presence of Massive Numbers of Defects.* IEEE Design and Test of Computers, 21(3):216–227, 2004.

[16] Chan, Wah und A. Orailoglu: *High-level synthesis of gracefully degradable ASICs.* In: *Proceedings of the 1996 European conference on Design and Test (EDTC '96)*, Seiten 50–54, 1996.

[17] Chandrasekar, Kavitha, Revathi Ananthachari, Sangeetha Seshadri und Ranjani Parthasarathi: *Fault Tolerance in OpenSPARC Multicore Architecture Using Core Virtualization.* In: *HiPC 2008 Student Research Symposium*, 2008.

[18] Chen, F., K. Chanda, I. Gill, M. AngyaI, J. Demarest, T. Sullivan, R. Kontra, M. Shinosky, J. Li, L. Economikos, M. Hoinkis, S. Lane, D. McHerron, M. Inohara, S. Boettcher, D. Dunn, M. Fukasawa, B.C. Zhang, K. Ida, T. Ema, G. Lembach, K. Kumar, Y. Lin, H. Maynard, K. Urata, T. Bolom, K. Inoue, J. Smith, Y. Ishikawa, M. Naujok, P. Ong, A. Sakamoto, D. Hunt und J. Aitken: *Investigation of CVD SiCOH low-k time-dependent dielectric breakdown at 65nm node technology.* In: *Proceedings of the 43rd IEEE International Symposium on Reliability Physics*, Seiten 501 – 507, 2005.

[19] Chen, Yung-Yuan, Shi-Jinn Horng und Hung-Chuan Lai: *An Integrated Fault-Tolerant Design Framework for VLIW Processors.* In: *Proceedings of the 18th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '03)*, Seiten 555–562, 2003.

[20] Corno, F., G. Gumani, M. Sonza Reorda und G. Squillero: *Fully Automatic Test Program Generation for Microprocessor Cores.* In: *IEEE Conference on Design, Automation and Test (DATE '03)*, Seiten 1006 – 1011, 2003.

[21] Dodd, P.E. und L.W. Massengill: *Basic Mechanisms and Modeling of Single-Event Upset in Digital Microelectronics.* IEEE Transactions on Nuclear Science, 50(3):583–602, 2003.

[22] Eichelberger, E. B. und T. W. Williams: *A Logic Design Structure for LSI Testability.* In: *Proceedings of the 14th Design Automation Conference (DAC '77)*, Seiten 462–468, 1977.

[23] El-Maleh, A.H., B.M. Al-Hashimi und A. Melouki: *Transistor-Level Based Defect Tolerance for Reliable Nanoelectronics.* In: *Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2008)*, Seiten 53–60, 2008.

[24] Entner, Robert: *Modeling and Simulation of Negativ Bias Temperatur Instability.* Doktorarbeit, Technische Universität Wien, 2007.

[25] Fang, Peng, Jiang Tao, J.F. Chen und Chenming Hu: *Design in hot-carrier reliability for high performance logic applications.* In: *Proceedings of the IEEE Conference on Custom Integrated Circuits*, Seiten 525 –531, 1998.

[26] Flynn, Michael J.: *Some Computer Organizations and Their Effectiveness.* IEEE Transactions on Computers, 21(9):948–960, 1972.

[27] Franklin, M.: *A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors.* In: *Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems (DFT '95)*, Seiten 207–215, 1995.

[28] Funatsu, S., N. Wakatsuki und A. Yamada: *Designing Digital Circuits with Easily Testable Consideration.* In: *Proceedings of the International Test Conference (ITC'78)*, 1978.

[29] Galke, C., T. Koal und H. T. Vierhaus: *Möglichkeiten und Grenzen der automatischen SBST Generierung für einfache Prozessoren - Fallstudie des Testprozessors T5016tp.* In: *Tagungsband Dresdner Arbeitstagung für Schaltungs- und Systementwurf*, Seiten 39–44, 2007.

[30] Gomaa, M., C. Scarbrough, T.N. Vijaykumar und I. Pomeranz: *Transient-Fault Recovery for Chip Multiprocessors.* In: *Proceedings of the 30th Annual International Symposium on Computer Architecture*, Seiten 98 – 109, 2003.

[31] Habermann, S., R. Kothe und H. T. Vierhaus: *Built-in Self Repair by Reconfiguration of FPGAs.* In: *Proceedings of the IEEE International Symposium on On-Line Testing*, Seiten 187–188, 2006.

[32] Hennessy, John L. und David A. Patterson: *Computer Architecture; A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[33] Hu, C.-K., R. Rosenberg, H.S. Rathore, D.B. Nguyen und B. Agarwala: *Scaling Effect on Electromigration in On-Chip Cu Wiring.* In: *Proceedings of the IEEE International Conference on Interconnect Technology*, Seiten 267 –269, 1999.

[34] Hu, Jie S., Feihui Li, Vijay Degalahal, Mahmut Kandemir, N. Vijaykrishnan und Mary J. Irwin: *Compiler-Directed Instruction Duplication for Soft Error Detection.* In: *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe (DATE '05)*, Seiten 1056–1057, 2005.

[35] Huang, Wei-Je und Edward J. McCluskey: *Column-Based Precompiled Configuration Techniques for FPGA*. In: *Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, Seiten 137–146, 2001.

[36] Hwu, W. W. und Y. N. Patt: *Checkpoint Repair for Out-of-Order Execution Machines*. In: *Proceedings of the 14th Annual International Symposium on Computer architecture (ISCA '87)*, Seiten 18–26, 1987.

[37] Imhof, Michael E. und Hans-Joachim Wunderlich: *Korrektur transienter Fehler in eingebetteten Speicherelementen*. In: *Zuverlässigkeit und Entwurf - 5. GI/GMM/ITG-Fachtagung*, 2011.

[38] Joseph, Russ: *Exploring Salvage Techniques for Multi-Core Architectures*. In: *Proceedings of the 2nd Workshop on High Performance Computing Reliability Issues*, 2006.

[39] Karri, R. und A. Orailoglu: *Scheduling with Rollback Constraints in High-Level Synthesis of Self-Recovering ASICs*. In: *Digest of Papers of the 22nd International Symposium on Fault-Tolerant Computing*, Seiten 519–526, 1992.

[40] Karri, Ramesh, Kyosun Kim und Miodrag Potkonjak: *Computer Aided Design of Fault-Tolerant Application Specific Programmable Processors*. IEEE Transactions on Computers, 49(11):1272–1284, 2000.

[41] Kerns, S.E., B.D. Shafer, Jr. Rockett, L.R., J.S. Pridmore, D.F. Berndt, N. van Vonno und F.E. Barber: *The design of radiation-hardened ICs for space: a compendium of approaches*. In: *Proceedings of the IEEE*, Band 76, Seiten 1470 –1509, 1988.

[42] Khan, O. und S. Kundu: *Thread Relocation: A Runtime Architecture for Tolerating Hard Errors in Chip Multiprocessors*. IEEE Transactions on Computers, 59(5):651–665, May 2010.

[43] Koal, T., M. Ulbricht und H.T. Vierhaus: *Combining On-Line Fault Detection and Logic Self Repair*. In: *Proceedings of the 15th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, Seiten 288 – 293, 2012.

[44] Koal, T., M. Ulbricht und H.T. Vierhaus: *Virtual TMR Schemes Combining Fault Tolerance and Self Repair*. In: *Proceedings of the Euromicro Conference on Digital Systems Design (DSD 2013)*, Seiten 235 – 242, 2013.

[45] Kothe, R., H. T. Vierhaus, T. Coym, W. Vermeiren und B. Straube: *Embedded Self Repair by Transistor and Gate Level Reconfiguration*. In: *Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS '06)*, Seiten 208–213, 2006.

[46] Lach, J., W.H. Mangione-Smith und M. Potkonjak: *Algorithms for Efficient Runtime Fault Recovery on Diverse FPGA Architectures.* In: *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '99)*, Seiten 386–394, 1999.

[47] Lala, P. K.: *A Single Error Correcting and Double Error Detecting Coding Scheme for Computer Memory Systems.* In: *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '03)*, Seiten 235–241, 2003.

[48] Lala, Parag K.: *Self-Checking and Fault-Tolerant Digital Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[49] Li, Yanjing, Young Moon Kim, E. Mintarno, D.S. Gardner und S. Mitra: *Overcoming Early-Life Failure and Aging for Robust Systems.* IEEE Design and Test of Computers, 26(6):28–39, 2009.

[50] Li, Yanjing, S. Makar und S. Mitra: *CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns.* In: *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe (DATE '08)*, Seiten 885 –890, 2008.

[51] Lienig, J. und G. Jerke: *Electromigration-Aware Physical Design of Integrated Circuits.* In: *Proceedings of the 18th International Conference on VLSI Design*, Seiten 77 – 82, 2005.

[52] Long, J., W.K. Fuchs und J. Abraham: *Compiler-Assisted Static Checkpoint Insertion.* In: *Digest of Papers of the 22nd International Symposium on Fault-Tolerant Computing*, Seiten 58–65, 1992.

[53] Lu, Zhijian, Wei Huang, J. Lach, M. Stan und K. Skadron: *Interconnect Lifetime Prediction Under Dynamic Stress for Reliability-Aware Design.* In: *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD-2004)*, Seiten 327 – 334, 2004.

[54] Meixner, Albert und Daniel J. Sorin: *Detouring: Translating Software to Circumvent Hard Faults in Simple Cores.* In: *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Seiten 80 – 89, 2008.

[55] Mishra, M. und S.C. Goldstein: *Defect Tolerance at the End of the Roadmap.* In: *Proceedings of the International Test Conference (ITC 2003)*, Seiten 1201 – 1210, 2003.

[56] Mitra, Subhasish, Wei-Je Huang, Nirmal R. Saxena, Shu-Yi Yu und Edward J. McCluskey: *Reconfigurable Architecture for Autonomous Self-Repair.* IEEE Design and Test of Computers, 21(3):228–240, 2004.

[57] Miura, Yoshio und Yasuo Matukura: *Investigation of Silicon-Silicon Dioxide Interface Using MOS Structure.* JJAP Japanese Journal of Applied Physics, 5(2):180–181, 1965.

[58] Mukherjee, S.S., M. Kontz und S.K. Reinhardt: *Detailed Design and Evaluation of Redundant Multi-Threading Alternatives.* In: *Proceedings of the 29th Annual International Symposium on Computer Architecture*, Seiten 99 –110, 2002.

[59] Nakamura, Y. und K. Hiraki: *Highly Fault-Tolerant FPGA Processor by Degrading Strategy.* In: *Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing*, Seiten 75–78, 2002.

[60] Natarajan, S., M.A. Breuer und S.K. Gupta: *Process Variations and Their Impact on Circuit Operation.* In: *Proceedings of the 1998 IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '98)*, Seiten 73 –81, 1998.

[61] Noji, Ryoji, Satoshi Fujie, Yuki Yoshikawa, Hideyuki Ichihara und Tomoo Inoue: *Reliability and Performance Analysis of FPGA-Based Fault Tolerant System.* In: *Proceedings of the 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT '12)*, Seiten 245–253, 2009.

[62] Ogawa, E.T., Jinyoung Kim, G.S. Haase, H.C. Mogul und J.W. McPherson: *Leakage, Breakdown, and TDDB Characteristics of Porous Low-K Silica-Based Interconnect Dielectrics.* In: *Proceedings of the 41st Annual IEEE International Reliability Physics Symposium*, Seiten 166 – 172, 2003.

[63] Oh, N., S. Mitra und E.J. McCluskey: *ED4I: Error Detection by Diverse Data and Duplicated Instructions.* IEEE Transactions on Computers, 51(2):180–199, 2002.

[64] Oh, N., P.P, Shirvani und E.J. McCluskey: *Error Detection by Duplicated Instructions in Super-Scalar Processors.* IEEE Transactions on Reliability, 51(1):65–75, 2002.

[65] Ohm, Seong Y., Douglas M. Blough und Fadi J. Kurdahi: *High-Level Synthesis of Recoverable Microarchitectures.* In: *Proceedings of the 1996 European conference on Design and Test (EDTC '96)*, Seiten 55–62, 1996.

[66] Orailoglu, A. und R. Karri: *Coactive Scheduling and Checkpoint Determination During High Level Synthesis of Self-Recovering Microarchitectures.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2(3):304 –311, 1994.

[67] Orailoglu, Alex: *Microarchitectural Synthesis of Gracefully Degradable, Dynamically Reconfigurable ASICs.* In: *Proceedings of the 1996 International Conference on Computer Design: VLSI in Computers and Processors (ICCD '96)*, Seiten 112–117, 1996.

[68] Orailoglu, Alex und Ramesh Karri: *Automatic Synthesis of Self-Recovering VLSI Systems.* IEEE Transactions on Computers, 45(2):131–142, 1996.

[69] Porrmann, M., M. Purnaprajna und C. Puttmann: *Self-optimization of MPSoCs Targeting Resource Efficiency and Fault Tolerance.* In: *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2009)*, Seiten 467 –473, 2009.

[70] Powell, Michael D., Arijit Biswas, Shantanu Gupta und Shubhendu S. Mukherjee: *Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance.* In: *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, Seiten 93–104, 2009.

[71] Pradhan, Dhiraj K.: *Fault-Tolerant Computer System Design.* Prentice Hall Inc., Upper Sattle River, NJ, USA, 1996.

[72] Rashid, Layali, Karthik Pattabiraman und Sathish Gopalakrishnan: *Intermittent Hardware Errors Recovery: Modeling and Evaluation.* In: *Proceedings of the IEEE International Conference on Quantitative Evaluation of Systems*, Seiten 220–229, 2012.

[73] Rashid, M.W., E.J. Tan, M.C. Huang und D.H. Albonesi: *Power-Efficient Error Tolerance in Chip Multiprocessors.* IEEE Micro, 25:60 – 70, 2005.

[74] Reddy, V., A.T. Krishnan, A. Marshall, John Rodriguez, S. Natarajan, T. Rost und Srikanth Krishnan: *Impact of Negative Bias Temperature Instability on Digital Circuit Reliability.* In: *Proceedings of the 40th Annual Reliability Physics Symposium*, Seiten 248–254, 2002.

[75] Reinhardt, S.K. und S.S. Mukherjee: *Transient Fault Detection Via Simultaneous Multithreading.* In: *Proceedings of the 27th International Symposium on Computer Architecture*, Seiten 25 –36, 2000.

[76] Reis, G.A., J. Chang, N. Vachharajani, R. Rangan und D.I. August: *SWIFT: Software Implemented Fault Tolerance.* In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2005)*, Seiten 243 – 254, 2005.

[77] Romanescu, Bogdan F. und Daniel J. Sorin: *Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults.* In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, Seiten 43–51, 2008.

[78] Rotenberg, E.: *AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors.* In: *Digest of Papers of the 29th Annual International Symposium on Fault-Tolerant Computing*, Seiten 84 –91, 1999.

[79] Sato, Toshinori: *Exploiting Instruction Redundancy for Transient Fault Tolerance.* In: *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '03)*, Seiten 547–554, 2003.

[80] Scheit, Daniel: *Fault-tolerant integrated interconnections based on built-in self-repair and codes.* Doktorarbeit, Brandenburgische Technische Universität Cottbus, Institut für Informatik - Lehrstuhl Technische Informatik, 2011.

[81] Schölzel, M., T. Koal, S. Röder und H.T. Vierhaus: *Towards an Automatic Generation of Diagnostic In-Field SBST for Processor Components.* In: *Proceedings of the 14th IEEE Latin American Test Workshop*, Seiten 1 – 6, 2013.

[82] Schölzel, Mario: *Automatisierter Entwurf anwendungsspezifischer VLIW-Prozessoren.* Dissertation, Brandenburgische Technische Universität Cottbus, Institut für Informatik - Lehrstuhl Technische Informatik, 2006.

[83] Schölzel, Mario: *Reduced Triple Modular Redundancy for Built-in Self Repair in VLIW Processors.* In: *Proceedings of the IEEE Signal Processing Workshop 2007 (SPA07)*, Seiten 21–26, 2007.

[84] Schölzel, Mario: *A Delay Estimation of Rescheduling Schemes for Static Scheduled Processor Architectures.* In: *Proceedings of the 22th International Conference on Architecture of Computing Systems 2009 (ARCS '09)*, Seiten 1–8, 2009.

[85] Schölzel, Mario: *Fine-Grained Software-Based Self-Repair of VLIW Processors.* In: *Proceedings of the IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '11)*, Seiten 41 – 49, 2011.

[86] Schölzel, Mario und Sebastian Müller: *Combining Hardware- and Software-Based Self-Repair Methods for Statically Scheduled Data Paths.* In: *Proceedings of the IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '10)*, Seiten 90–98, 2010.

[87] Scholzel, M., T. Koal und H.T. Vierhaus: *An Adaptive Self-Test Routine for In-Field Diagnosis of Permanent Faults in Simple RISC Cores.* In: *Proceedings of the 15th IEEE International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, Seiten 312–317, 2012.

[88] Schuchman, E. und T.N. Vijaykumar: *BlackJack: Hard Error Detection with Redundant Threads on SMT.* In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, Seiten 327 –337, 2007.

[89] Schuchman, Ethan und T. N. Vijaykumar: *Rescue: A Microarchitecture for Testability and Defect Tolerance.* In: *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, Seiten 160–171, 2005.

[90] Shivakumar, Premkishore, S.W. Keckler, C.R. Moore und D. Burger: *Exploiting Microarchitectural Redundancy for Defect Tolerance.* In: *Proceedings of the 21st International Conference on Computer Design*, Seiten 481 – 488, 2003.

[91] Shye, A., J. Blomstedt, T. Moseley, V.J. Reddi und D.A. Connors: *PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures.* IEEE Transactions on Dependable and Secure Computing, 6(2):135 –148, 2009.

[92] Silberberg, Rein, Chen H. Tsao und John R. Letaw: *Neutron Generated Single-Event Upsets in the Atmosphere.* IEEE Transactions on Nuclear Science, 31(6):1183 –1185, 1984.

[93] Sridhara, S.R. und N.R. Shanbhag: *Coding for System-on-Chip Networks: A Unified Framework.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 13(6):655 – 667, 2004.

[94] Srinivasan, J., S.V. Adve, P. Bose und J.A. Rivers: *The Impact of Technology Scaling on Lifetime Reliability.* In: *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, Seiten 177 – 186, 2004.

[95] Srinivasan, Jayanth: *Lifetime Reliability Aware Microprocessors.* Doktorarbeit, University of Illinois, 2006.

[96] Suzumura, N., S. Yamamoto, D. Kodama, K. Makabe, J. Komori, E. Murakami, S. Maegawa und K. Kubota: *A New TDDB Degradation Model Based on Cu Ion Drift in Cu Interconnect Dielectrics.* In: *Proceedings of the 44th Annual IEEE International Reliability Physics Symposium*, Seiten 484 –489, 2006.

[97] Taber, Allen H. und Eugene Normand: *Investigation and Characterization of SEU Effects and Hardening Strategies in Avionics.* Technischer Bericht, IBM Report 92-L75-020-2, August 1992.

[98] Teich, Jürgen und Christian Haubelt: *Digitale Hardware/Software-Systeme: Synthese und Optimierung.* Springer-Verlag, Berlin, DE, 2007.

[99] Thatte, S.M. und J.A. Abraham: *Test Generation for Microprocessors.* IEEE Transactions on Computers, 29(6):429–441, 1980.

[100] Vadlamani, Ramakrishna, Jia Zhao, Wayne Burleson und Russell Tessier: *Multicore Soft Error Rate Stabilization Using Adaptive Dual Modular Redundancy.* In: *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe (DATE '10)*, Seiten 27–32, 2010.

[101] Vijaykumar, T.N., I. Pomeranz und K. Cheng: *Transient-Fault Recovery Using Simultaneous Multithreading.* In: *Proceedings of the 29th Annual International Symposium on Computer Architecture*, Seiten 87–98, 2002.

[102] Wells, Philip M., Koushik Chakraborty und Gurindar S. Sohi: *Mixed-Mode Multicore Reliability.* In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seiten 169–180, 2009.

[103] Xu, Weifeng, R. Ramanarayanan und R. Tessier: *Adaptive Fault Recovery for Networked Reconfigurable Systems.* In: *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, Seiten 143 – 152, 2003.

[104] Yu, Shu-Yi und E.J. McCluskey: *Permanent Fault Repair for FPGAs With Limited Redundant Area.* In: *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '01)*, Seiten 125–133, 2001.

[105] Zafar, S., B.H. Lee, J. Stathis, A. Callegari und Tak Ning: *A Model for Negative Bias Temperature Instability (NBTI) in Oxide and High Kappa pFETs 13x-C6D8C7F5F2.* In: *Digest of Technical Papers of the 2004 Symposium on VLSI Technology*, Seiten 208 – 209, 2004.

[106] Zajac, P. und A. Napieralski: *Estimating Performance Penalty for Various Fault-Tolerant Techniques in Multicore Processors.* In: *Proceedings of the 18th International Conference on Mixed Design of Integrated Circuits and Systems (MIXDES '11)*, Seiten 490 –495, 2011.

[107] Ziegler, J. F. und Lanford W. A.: *Effect of Cosmic Rays on Computer Memories.* Science Magazine, 206(4420):776–788, 1979.

# A. Ergebnisse der Fehlersimulation

Im Folgendem befindet sich eine Sammlung zu den Ergebnissen der Fehlersimulation für die Systeme der Grundkonfigurationen $(2,3,4)$, $(1,2,3,4)$ und $(2,2,3,4,5)$. Für jede Untersuchungsreihe ist eine Diagramm zur Entwicklung der Systemausfälle, eine Tabelle zu den gesteigerten Lebenszeiten und eine Tabelle für die tolerierbaren Fehler angegeben.

## Untersuchungsreihe für die Grundkonfiguration $(2,3,4)$

Die Grundkonfiguration ist für die fehlertolerante Konfiguration zum System $(4,3,4)$ erweitert. Abbildung A.1 stellt die Entwicklung der Systemausfälle über die Zeit dar. In Tabelle A.1 ist die Verbesserung der durchschnittlichen Lebenszeit gegeben und Tabelle A.2 zeigt die durchschnittlich tolerierbaren Fehler.

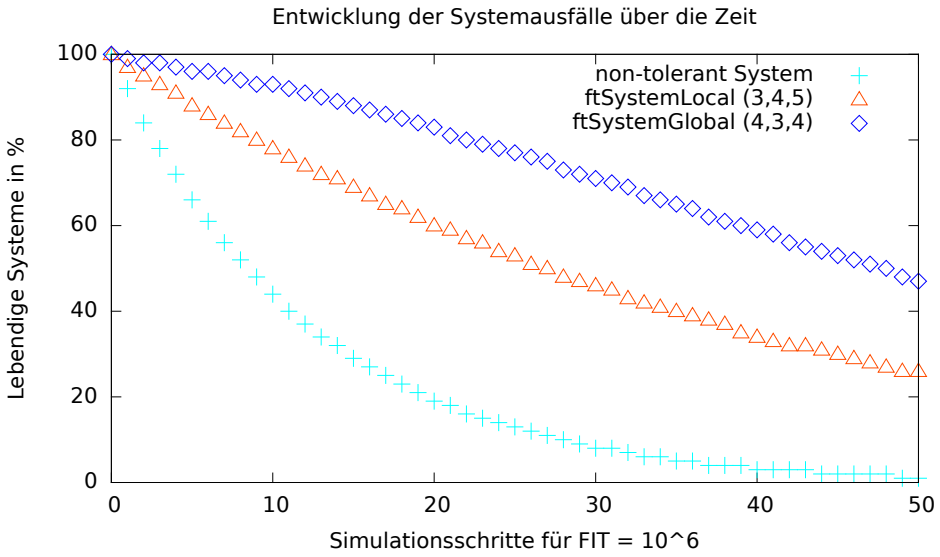


Abbildung A.1.: Entwicklung der mittleren Lebenszeit der Systeme *ftSystemLokal* und *ftSystemGlobal* sowie des nicht-fehlertoleranten Systems $(2,3,4)$

| lebendige | Verbesserung um Faktor | |
|---|---|---|
| Systeme in % | lokale Strategie | globale Strategie |
| 98 | 1 | 4,0 |
| 95 | 3,0 | 2,7 |
| 92 | 2,0 | 3,0 |
| 90 | 2,5 | 2,8 |
| 85 | 3,5 | 2,7 |
| 80 | 3,3 | 2,3 |
| 70 | 3,0 | 2,1 |
| 60 | 3,0 | 1,9 |
| 50 | 3,1 | 1,8 |

Tabelle A.1.: Verbesserung der Lebenszeit für die lokale und globale Reparaturstrategie, bei entsprechender Anzahl an lebendigen Systemen

| Fehler | überlebende Systeme in % | |
|---|---|---|
| | lokale Strategie | globale Strategie |
| 1 | 74,8 | 93,7 |
| 2 | 52,9 | 82,5 |
| 3 | 36,7 | 66,4 |
| 4 | 24,7 | 48,9 |
| 5 | 16,0 | 33,1 |
| 6 | 10,0 | 20,5 |
| 7 | 5,9 | 11,5 |
| 8 | 3,3 | 5,8 |
| 9 | 1,7 | 2,7 |

Tabelle A.2.: Prozentuale Angabe der überlebenden Systeme für die beiden Reparaturstrategien und die entsprechende Anzahl an Fehlern

## Untersuchungsreihe für die Grundkonfiguration $(1, 2, 3, 4)$

Die Grundkonfiguration ist für die fehlertolerante Konfiguration zum System $(4, 2, 3, 4)$ erweitert. Abbildung A.2 stellt die Entwicklung der Systemausfälle über die Zeit dar. In Tabelle A.3 ist die Verbesserung der durchschnittlichen Lebenszeit gegeben und Tabelle A.4 zeigt die durchschnittlich tolerierbaren Fehler.

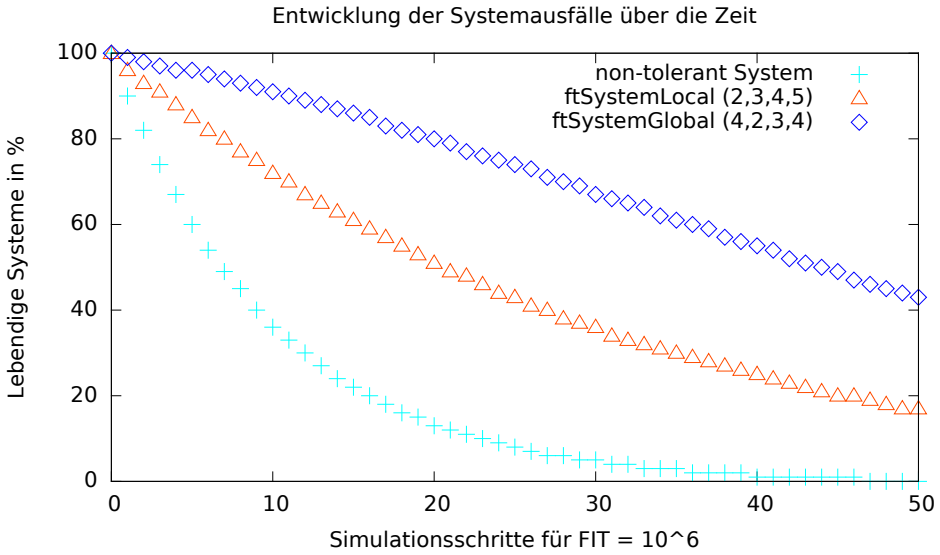


Abbildung A.2.: Entwicklung der mittleren Lebenszeit der Systeme *ftSystemLokal* und *ftSystemGlobal* sowie des nicht-fehlertoleranten Systems $(1, 2, 3, 4)$

| lebendige | Verbesserung um Faktor | |
|---|---|---|
| Systeme in % | lokale Strategie | globale Strategie |
| 98 | 1 | 3,0 |
| 95 | 2,0 | 3,5 |
| 92 | 3,0 | 3,3 |
| 90 | 2,0 | 3,0 |
| 85 | 3,0 | 2,8 |
| 80 | 2,7 | 2,6 |
| 70 | 3,0 | 2,4 |
| 60 | 2,7 | 2,3 |
| 50 | 3,0 | 2,1 |

Tabelle A.3.: Verbesserung der Lebenszeit für die lokale und globale Reparaturstrategie, bei entsprechender Anzahl an lebendigen Systemen

| Fehler | überlebende Systeme in % | |
|---|---|---|
| | lokale Strategie | globale Strategie |
| 1 | 72,6 | 93,2 |
| 2 | 50,3 | 83,5 |
| 3 | 34,5 | 71,7 |
| 4 | 23,1 | 58,1 |
| 5 | 15,0 | 44,1 |
| 6 | 9,5 | 31,3 |
| 7 | 5,8 | 20,9 |
| 8 | 3,4 | 13,1 |
| 9 | 1,9 | 7,7 |
| 10 | 1,0 | 4,2 |

Tabelle A.4.: Prozentuale Angabe der überlebenden Systeme für die beiden Reparaturstrategien und die entsprechende Anzahl an Fehlern

## Untersuchungsreihe für die Grundkonfiguration $(2, 2, 3, 4, 5)$

Die Grundkonfiguration ist für die fehlertolerante Konfiguration zum System $(5, 3, 4, 4, 5)$ erweitert. Abbildung A.3 stellt die Entwicklung der Systemausfälle über die Zeit dar. In Tabelle A.5 ist die Verbesserung der durchschnittlichen Lebenszeit gegeben und Tabelle A.6 zeigt die durchschnittlich tolerierbaren Fehler.

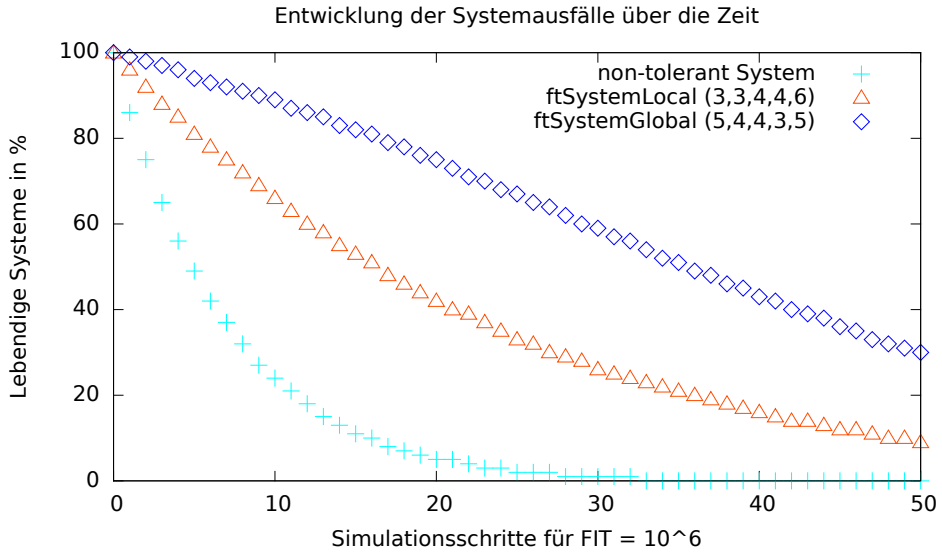


Abbildung A.3.: Entwicklung der mittleren Lebenszeit der Systeme *ftSystemLokal* und *ftSystemGlobal* sowie des nicht-fehlertoleranten Systems $(2, 2, 3, 4, 5)$

| lebendige | Verbesserung um Faktor | |
|---|---|---|
| Systeme in % | lokale Strategie | globale Strategie |
| 98 | 1 | 3,0 |
| 95 | 2,0 | 2,5 |
| 92 | 3,0 | 2,7 |
| 90 | 3,0 | 3,3 |
| 85 | 5,0 | 2,8 |
| 80 | 3,0 | 2,8 |
| 70 | 3,0 | 2,7 |
| 60 | 3,3 | 2,3 |
| 50 | 3,4 | 2,1 |

Tabelle A.5.: Verbesserung der Lebenszeit für die lokale und globale Reparaturstrategie, bei entsprechender Anzahl an lebendigen Systemen

| Fehler | überlebende Systeme in % | |
|---|---|---|
| | lokale Strategie | globale Strategie |
| 1 | 74,3 | 94,1 |
| 2 | 52,9 | 86,0 |
| 3 | 37,0 | 77,2 |
| 4 | 25,4 | 66,6 |
| 5 | 17,0 | 54,5 |
| 6 | 11,1 | 42,1 |
| 7 | 7,0 | 30,6 |
| 8 | 4,3 | 21,0 |
| 9 | 2,6 | 13,7 |
| 10 | 1,5 | 8,5 |

Tabelle A.6.: Prozentuale Angabe der überlebenden Systeme für die beiden Reparaturstrategien und die entsprechende Anzahl an Fehlern