Brandenburgische Technische Universität Cottbus - Senftenberg
Fakultät 1 – Mathematik, Naturwissenschaften, Informatik
Lehrstuhl Technische Informatik

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Systematische Lebensdauer-Optimierung für hochintegrierte Systeme auf der Basis von Nano-Strukturen durch Stress-Optimierung und Selbstreparatur

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

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigte Dissertation

vorgelegt von

Master of Science (M.Sc.)

Markus Ulbricht

Geboren am 08.07.1984 in Altenburg

Gutachter: Prof. Dr.-Ing. Heinrich Theodor Vierhaus

Gutachter: Prof. Dr.-Ing. Rolf Kraemer

Gutachter: Prof. Dr. Raimund Ubar

Tag der mündlichen Prüfung: 27.08.2014

Kurzfassung

Die Skalierung von Bauelementen der Mikroelektronik bis hin zu atomaren Dimensionen hat einen zunehmenden negativen Einfluss auf deren Zuverlässigkeit und mittlere Lebensdauer. Durch uneinheitliches Skalieren der räumlichen Ausdehnung im Vergleich zur Versorgungsspannung steigen die internen Energiedichten und damit die Temperatur. Diese trägt wesentlich zur Verstärkung gewisser Fehlereffekte bei. Maßnahmen wie Selbstreparatur und Entstressen können dieser Entwicklung entgegenwirken, wobei Kombinationen solcher Ansätze noch bessere Ergebnisse versprechen. Ziel der vorliegenden Dissertationsschrift ist es, eine dieser Kombinationen auf ihre Kosten und Nutzen zu untersuchen. Damit kann am Ende die Aussage getroffen werden, ob die Activity Migration eine gewinnbringende Ergänzung in einem zur Selbstreparatur genutzten M aus N System darstellt oder nicht.

Um dieses Ziel zu erreichen, werden zuerst fünf verschiedene Implementierungen des Ansatzes in einem VLIW Prozessor erstellt. Anschließend findet die Simulation der Implementierungen in verschiedenen Strukturgrößen, in Bezug auf ihr Temperaturverhalten statt. Über entsprechende Modellierungen wird daraufhin die Zuverlässigkeit und mittlere Lebensdauer der Originalsysteme, Systeme mit reinem M aus N System und den Systemen mit integrierter Activity Migration in Abhängigkeit von Temperatur und Fläche ermittelt. Die erzeugten Ergebnisse belegen, dass das Hinzufügen der Activity Migration bei den untersuchten Prozessormodellen, mit starker thermischer Kopplung zwischen Funktions- und Ersatzbaugruppen, warmer Redundanz, Rekonfiguration im laufenden Betrieb und den gegebenen Formeln zur Berechnung der Fehlerraten verschiedener Fehlermodelle, keinen nennenswerten Gewinn an Lebensdauer erbringt, da das reine M aus N System bereits zu einer deutlichen Lebensdauersteigerung führt.

Abstract

The continued scaling of microelectronic elements down to atomic dimensions has a growing negative influence on their reliability and lifetime. By scaling the spatial dimensions faster than the supply voltage, internal power densities rise and with that the temperature. This substantially accelerates certain fault effects. Countermeasures such as self repair and destressing are able to slow down this development, whereby a combination of these approaches promises even better results. The aim of this thesis is to examine one of these approaches to expose its costs and benefits. By the end of this research, it is possible to decide, whether the Activity Migration is a beneficial supplement to an existing k out of n system, used for self repair, or not.

To achieve this goal, five different versions of the approach are implemented in a VLIW processor. Afterwards, those implementations are simulated in different feature sizes to determine their properties, regarding the temperature. In the next part, the reliability and mean time to failure of the original systems, of the systems with pure k out of n system and of the systems with integrated activity migration are calculated in relation to the temperature and area, using adequate models. The generated results show that adding activity migration to an existing k out of n system with strong thermal coupling between the active and passive components, warm redundancy, online reconfiguration and the given formulas for the calculation of failure rates under certain fault effects, brings no gain in lifetime worth mentioning, since the pure k out of n system by itself already leads to a significant increase in expected lifetime.

Danksagung

Im Folgenden möchte ich mich bei allen Personen bedanken, die mir bei der Anfertigung dieser Arbeit hilfreich zur Seite standen. Besonderer Dank gilt Dr. rer. nat. Mario Schölzel und M.Sc. Tobias Koal für verschiedene Denkanstöße und fachliche Hilfe. Weiterhin möchte ich meine Dankbarkeit gegenüber der IHP GmbH in Frankfurt (Oder) und speziell Prof. Dr.-Ing. Rolf Kraemer dafür ausdrücken, dass ich ihre Ressourcen und Zellbibliotheken nutzen durfte. Dies hat die Aussagekraft der ermittelten Ergebnisse deutlich verbessert. Dank an meine Freunde, die mir vor allem dabei geholfen haben, die Arbeit mit Abstand und unter anderen Blickwinkeln zu betrachten. Insbesondere bedanke ich mich bei meinen Eltern für die kontinuierliche Unterstützung während meines gesamten Studiums und für den stetigen Ansporn, mein Bestes zu geben. Weiterhin danke ich Prof. Dr.-Ing. H. T. Vierhaus. Ihm gelang es, mein Interesse an der technischen Informatik zu wecken und mir fachliches Wissen zu vermitteln, ohne das diese Arbeit nicht hätte entstehen können. Ein spezieller Dank gilt der Deutschlehrerin Frau Beutel. Sie half mir dabei, die „Knoten“ aus so manchem Satz zu ziehen und damit nicht nur den Lesefluss, sondern auch das Verständnis deutlich zu verbessern.
Vielen Dank auch an meine Stefanie für ihre Unterstützung und manch beruhigendes Wort, wenn mal wieder nichts geklappt hat.

Danke euch allen!

Die vorgestellte Arbeit wurde vom Ministerium für Wissenschaft, Forschung und Kultur (MWKF) des Landes Brandenburg im Rahmen der Internationalen Graduiertenschule der Brandenburgischen Technischen Universität Cottbus (BTU) unterstützt.

Abkürzungsverzeichnis

ALU	arithmetisch logische Einheit
AM	Activity Migration
BISR	Built In Self Repair
CDF	Cumulative Distribution Function
CMOS	Complementary Metal Oxide Semiconductor
DR	Decode Register
DSE	Design Space Exploration
DTM	Dynamic Thermal Management
EXE	Execute
FE	Funktionseinheit
FIT	Failures In Time
FPGA	Field Programmable Gate Array
FR	Fetch Register
HCI	Hot Carrier Injection
IC	Integrierter Schaltkreis
ID	Instruction Decode
IF	Instruction Fetch
IPC	Instructions Issued Per Cycle
LB	Load Balancing
LIF	Lifetime Improvement Factor
MaNS	M aus N System
MAR	Memory Address Register
MBR	Memory Buffer Register
MEU	Multiple Event Upset
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
MTTF	Mean Time To Failure
NBTI	Negative Bias Temperature Instability
PBTI	Positive Bias Temperature Instability
PDF	Probability Density Function
SEGR	Single Event Gate Rupture
SEL	Single Event Latchup
SEU	Single Event Upset
SMT	Simultaneously Multi-Threaded
SOC	System on Chip
STM	Static Thermal Management
TDDB	Time Dependent Dielectric Breakdown

TMR	Triple Modular Redundancy
TTF	Time To Failure
V_{dd}	Versorgungsspannung
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VLIW	Very Long Instruction Word
WB	Write Back
WR	Write Back Register

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die stetig voranschreitende Verkleinerung der Strukturgrößen in CMOS Technologie ist einer der wichtigsten Gründe, weshalb die Rechenkapazitäten von Prozessoren in den letzten Jahrzehnten immer weiter zunehmen konnten. Diese Entwicklung bringt aber nicht nur Vorteile mit sich, sondern auch neue Problemstellungen, die einer Lösung bedürfen.

Dieses Kapitel soll als Einstieg kurz erläutern, was bei der Technologieskalierung genau geschieht und welche Vor- und Nachteile dieser Trend mit sich bringt. Daraufhin ist es im nächsten Teil möglich, die vorliegenden Untersuchungen zu motivieren und das Ziel dieser Arbeit festzulegen. Der letzte Abschnitt gibt einen Überblick über den Aufbau dieser Schrift.

1.1. Die Skalierung der Technologie und ihre Folgen

Laut [Che04, S. 120 f.] existieren verschiedene Ansätze der Technologieskalierung, wobei die meisten aber Derivate der Skalierung mit konstanter elektrischer Feldstärke (*constant field scaling*) darstellen. Bei dieser werden sowohl die lateralen und vertikalen Dimensionen des Metal Oxide Semiconductor Field Effect Transistor (MOSFET) als auch die Versorgungsspannung um den selben Faktor κ skaliert, wodurch das elektrische Feld im Transistor unverändert bleibt. Dadurch verringert sich die Stromstärke am Drain Gebiet (siehe Abb.: 1.1) und die Kapazität des Gate, was zu geringerem Stromverbrauch pro Transistor und geringeren Ladezeiten des Gate führt.

Andere Methoden halten beispielsweise die Spannung gleich und skalieren die Oxiddicke (*constant voltage scaling*), ändern alle drei Faktoren, aber um einen abweichenden Faktor κ' (*quasi constant voltage scaling*), oder skalieren zusätzlich noch das Substrat (*general scaling*).

Unabhängig davon, welche Skalierungsmethode letztendlich aber Einsatz findet, die Ziele sind gleich: Verringerung der Dimensionen, der Schaltzeiten und des Stromverbrauchs. Ein sehr guter Überblick über die positiven Auswirkungen der Technologieskalierung wird in [Bor99] geboten. Zwar ist diese Veröffentlichung nicht mehr ganz aktuell, aber viele Aussagen stimmen nach wie vor. Es wird davon ausgegangen, dass sich die Verzögerung in den Transistoren und die diagonalen und vertikalen Dimensionen um 30% pro Technologiegeneration verringern. Damit ergeben sich folgende positive Veränderungen:

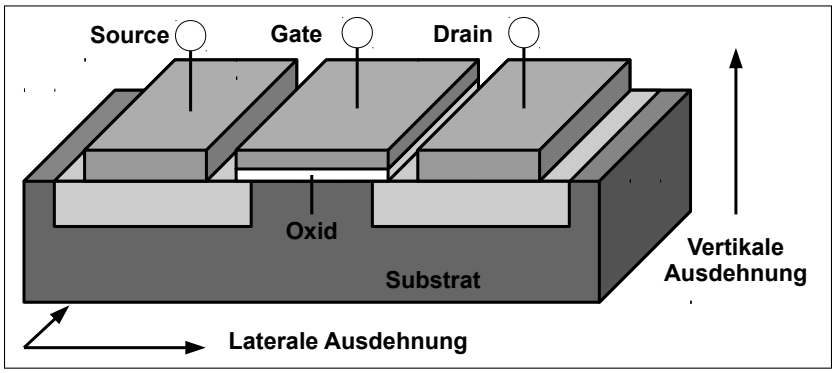


Abbildung 1.1.: Schematische Darstellung eines MOSFET

- **Steigerung der Arbeitsleistung:**
  Die Reduzierung der Verzögerung in den Gattern um 30% ermöglicht es, die Taktfrequenz um ca. 43% zu erhöhen.

- **Dichte der Transistoren:**
  Durch die Skalierung der Dimensionen können die Transistoren nun ungefähr doppelt so dicht platziert werden. Dies ermöglicht die Unterbringung von mehr Funktionen, wenn die Chipfläche konstant bleibt.

- **Reduzierter Energieverbrauch:**
  Die verbrauchte Energie pro Schaltvorgang verringert sich ungefähr um 65%, wodurch ca. 50% Leistung eingespart werden können (bei einer um 43% gestiegenen Taktfrequenz).

Der wichtigste Unterschied dieser Quelle zur heutigen Entwicklung ist, dass die Taktfrequenzen nicht wie 1999 noch erwartet, immer weiter erhöht wurden. Unter den Intel Prozessoren hatte bereits der Pentium 4 (Prescott 2M) im Jahr 2005 mit seinen maximalen 3,8 GHz die Spitze erreicht ([Sch12]). Heutige Core Architekturen beispielsweise siedeln sich eher maximal um die 3,3 GHz an. Die Steigerung der Arbeitsleistung wird zur Zeit mit höherer Integration von Speichern und parallel arbeitenden Einheiten auf dem Chip erreicht.

Die wichtigsten negativen Auswirkungen des anhaltenden Trends zur Technologieskalierung können laut [UKV12b] wie folgt zusammengefasst werden:

- **Schwankungen im Herstellungsprozess:**
  Je kleiner die Strukturen erzeugt werden, desto größer ist der Einfluss von Schwankungen im Herstellungsprozess. Dadurch weichen die elektrischen Eigenschaften der Transistoren voneinander ab, was zu unspezifiziertem Verhalten und damit letztlich Fehlern führen kann ([Bor05]).

- **Zeitabhängiger Verschleiß:**
  Wenn die Versorgungs- und Schwellspannungen nicht ideal mit der Technologie

skalieren, erhöhen sich interne Energiedichten und Temperaturen und beschleunigen Alterungseffekte in dem Grad der Skalierung überproportionalem Maße. Beispielsweise bedeutet der Umstieg von 180nm auf 65nm Technologie bei Ausführung des gleichen Programms laut [SABR05] eine Temperaturerhöhung von 69°C auf 84°C, was nach [VWWL00] ungefähr einer Halbierung der Lebensspanne des Systems gleichkommt.

- **Zufällige Fehler durch externe Effekte:**
  Kleinere Schaltkreise mit geringeren Versorgungsspannungen sind anfälliger für elektro-magnetische Interferenzen und Alpha-Strahlen. Diese können einzelne und mehrere Schaltungsknoten umladen - Single Event Upset (SEU) und Multiple Event Upset (MEU) - oder sogar die Gate-Isolation zerstören - Single Event Gate Rupture (SEGR) ([PTB+10]) und Single Event Latchup (SEL) ([Red01]).

1.2. Motivation und Zielsetzung

Obwohl die negativen Auswirkungen der Technologieskalierung immer mehr an Bedeutung gewinnen, wird die Strukturverkleinerung stetig vorangetrieben. Das liegt vor allem daran, dass der Wettbewerb unter den Herstellern integrierter Systeme beispielsweise im Bereich von PCs, Spielekonsolen und mobilen Geräten sehr stark ist. Deshalb wird die Erforschung neuer Methoden zur Abschwächung der negativen Auswirkungen dringender denn je benötigt.

Dies ist nun der Ansatzpunkt der vorliegenden Schrift. Grundlage bildet die in [KV10] vorgeschlagene Methode. Sie verbindet die Möglichkeit, zeitabhängigen Verschleiß zu verlangsamen, mit der Fähigkeit, das System trotz einmal aufgetretener statischer Fehler in gewissen Grenzen weiter lauffähig zu erhalten. Somit ist sie besonders vielversprechend, da sie an mehreren negativen Auswirkungen der Technologieskalierung gleichzeitig ansetzt. Ziel dieser Arbeit ist es nun, die Methode umzusetzen und deren Kosten und Nutzen zu bestimmen.

Die Komplexität der Aufgabe ergibt sich dabei einerseits aus der Größe des Suchraums. Die Methode ist auf eine Vielzahl von Systemen anwendbar und wird in unterschiedlichen Technologien unterschiedliche Resultate liefern. Hier muss ein Weg gefunden werden, dennoch aussagekräftige Ergebnisse zu erzeugen. Andererseits besitzt die Methode Merkmale, die sie von anderen deutlich unterscheidet. Aus diesem Grund müssen passende Metriken gefunden werden, die nicht nur Kosten und Nutzen beschreiben, sondern auch den Vergleich zu anderen Ansätzen ermöglichen.

1.3. Aufbau der Arbeit

Der weitere Aufbau der vorliegenden Schrift ist wie folgt: Kapitel 2 gibt einen Überblick über die wichtigsten Grundlagen dieser Arbeit, die für das spätere Verständnis notwendig sind. Dazu gehört zuerst der Aufbau des sogenannten VLIW Prozessors. Dieser bildet nicht nur die Voraussetzung für einige Methoden, die im weiteren Verlauf vorgestellt wer-

den, sondern soll aufgrund seiner imminenten Eigenschaft der Skalierbarkeit Grundlage für die durchgeführten Implementierungen bilden. Weiterhin werden die Begriffe Stress, Zuverlässigkeit, Fehlerrate und mittlere Lebensdauer (MTTF) definiert, da sie später als Maß für den Nutzen des Systems von Bedeutung sind. Zum Schluss folgt ein tiefgehender Einblick in die Bedeutung der Temperatur als Metrik für Stress, wie diese durch die Technologieskalierung beeinflusst wird und welche negativen Auswirkungen eine erhöhte Temperatur auf das System hat.

Kapitel 3 beschreibt den derzeitigen Stand der Technik in Bezug auf die Lebensdaueroptimierung. Dazu gehört zuerst eine Darlegung der Fehlzustände, in denen sich ein System befinden kann, damit der Ansatzpunkt der Optimierungen besser verdeutlicht werden kann. Danach folgt eine ausführliche Vorstellung zweier Teilgebiete der Lebensdaueroptimierung integrierter Systeme: Das Thermal Management und die Selbstreparatur.

In Kapitel 4 wird der zu untersuchende Ansatz noch einmal sehr detailliert vorgestellt. Dazu gehört die Verknüpfung von Selbstreparatur und Activity Migration und wichtige Eigenschaften, die sich daraus ergeben. Zum Schluss werden die Vorteile des Ansatzes unterstrichen und damit noch einmal herausgearbeitet, wie er die durchgeführten Untersuchungen motivieren konnte.

Kapitel 5 behandelt die Umsetzung im VLIW Prozessor. Dabei wird zunächst verdeutlicht, warum dieses Prozessormodell für die Implementierung so geeignet scheint und welche Eigenschaften es besitzt um trotz des großen Suchraums aussagekräftige Ergebnisse zu liefern. Darauf folgt die Beschreibung der Umsetzung in fünf verschiedenen Implementierungen und die Diskussion wichtiger Details der Funktionsweise und des Aufbaus. Hierüber werden später die durch die Realisierung entstehenden Zusatzkosten bestimmt.

Die Temperatur wurde bereits in Kapitel 2 als wichtige Metrik für die „Güte“ des Systems in Bezug auf eine Lebensdaueroptimierung erkannt. In Kapitel 6 folgt die Beschreibung, wie die Temperatur des Systems ohne und mit Einsatz der zu untersuchenden Methode ermittelt wird. Dazu gehört die Auflistung der Programme und Techniken, die Einsatz finden und die für einen reibungslosen „Toolflow“ notwendigen Formatierungen von Zwischenergebnissen.

Um von der Temperatur auf die Lebensdauer zu schließen, bietet Kapitel 7 einen Überblick über die Modellierung der Systeme in Bezug auf ihr Zuverlässigkeit und mittlere Lebensdauer und beschreibt damit die Wege, wie Ergebnisse errechnet werden.

In Kapitel 8 werden wichtige getroffene Annahmen und die damit ermittelten Ergebnisse dargelegt. Dies umfasst eine Auflistung der Kosten in Form von Hard- und Softwareaufwand und des Nutzens in Form einer Temperatursenkung. Weiterhin findet der Vergleich zu anderen Systemen und die Umrechnung der Temperatursenkung in einen Gewinn an Lebensdauer statt.

Das letzte Kapitel fasst die ermittelten Ergebnisse noch einmal übersichtlich zusammen, beschreibt den Bereich ihrer Gültigkeit, wertet sie aus und gibt Ausblicke auf mögliche weiterführende Arbeiten.

Kapitel 2

Grundlagen

Ziel dieses Kapitels ist es, das für die weitergehenden Untersuchungen eingesetzte Modell des VLIW Prozessors vorzustellen und grundlegende Begriffe einzuführen. Dazu wird zuerst kurz die Intention vermittelt, die hinter der VLIW Architektur steht. Daran schließt sich die Beschreibung zweier unterschiedlicher Versionen des Prozessors, einmal ohne und einmal mit der Bildung von Clustern. Daraufhin folgt die Erläuterung der Begriffe Stress, Fehlerrate, Zuverlässigkeit und mittlere Lebensdauer und wie diese in Verbindung stehen. Im letzten Teil wird auf die Bedeutung der Temperatur als wichtiges Anzeichen für Stress und ihre Auswirkungen auf die Beschleunigung gewisser Fehlereffekte eingegangen.

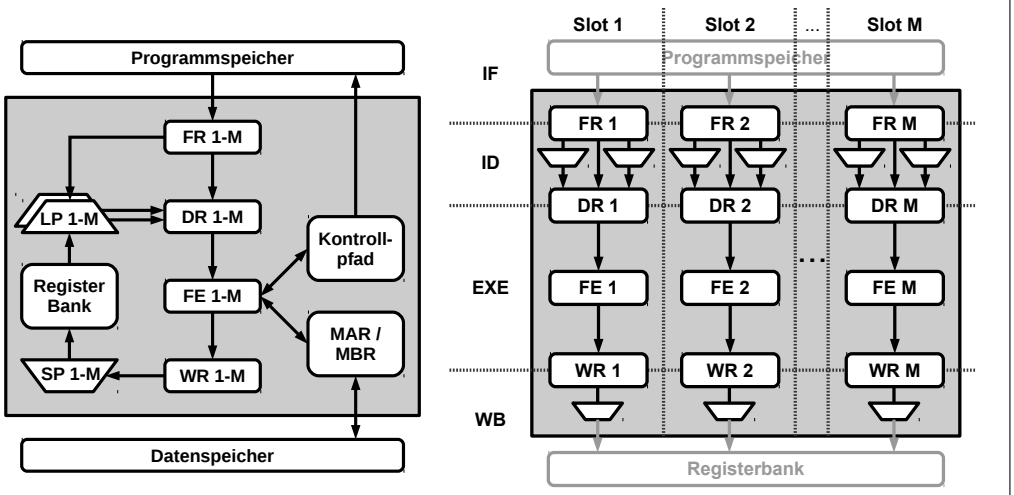
2.1. Der VLIW Prozessor

Eingeführt wurde die VLIW Architektur 1983 von Joseph A. Fisher ([Fis83]). Er stellte sie dabei sehr treffend mit folgenden Worten dar:

> „*Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.*“ [Fis83, S. 263]

Hierbei bezeichnete er als wichtigen Trend, dass „alle“ billige Hardware parallel betreiben wollen, um bestehende Rechenkapazitäten zu erhöhen. Sein vorgeschlagener Ansatz war, einfach den bevorzugten RISC-Prozessor so auszulegen, dass er 10 bis 30 Operationen parallel pro Takt ausführen kann und ihn dann mit einem einzelnen, sehr langen Instruktionswort - Very Long Instruction Word (VLIW) - anzusteuern. Um dies weiter zu formalisieren, wies er der VLIW-Architektur folgende Eigenschaften zu:

- Es existiert eine Kontrolleinheit, die pro Takt ein einzelnes, sehr langes Instruktionswort ausgibt.
- Jede lange Instruktion besteht aus vielen, eng beieinander stehenden, aber voneinander unabhängigen Operationen.
- Jede Operation benötigt eine kleine, statisch festgelegte Anzahl von Takten zur Ausführung.



(a) VLIW - Datenfluss (b) VLIW - Pipelines und Slots

Abbildung 2.1.: Schematischer Aufbau des VLIW Prozessors

> „*Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.*“[Fis83, S. 263]

Diese Eigenschaften und dass die Abarbeitung der einzelnen Operationen schrittweise in getrennten Pipelines geschehen kann, waren deutliche Unterschiede zu bereits bestehenden Multiprozessoren, auf denen die Tasks asynchron laufen und Datenfluss-gesteuerten Maschinen, die keinen zentralen Kontrollfluss besitzen. Außerdem enthalten VLIW Prozessoren nicht zwingend die Regularität, wie sie bei Vektor- oder Arrayprozessoren notwendig ist.

2.1.1. Prinzipieller Aufbau

Die soeben genannten Eigenschaften spiegeln sich auch in der vorliegenden Implementierung des VLIW Prozessors wieder. Aufgebaut ist er grundlegend aus Programm- und Datenspeicher, dem Memory Address Register (MAR) und Memory Buffer Register (MBR) für die Interaktion mit dem Datenspeicher, einem einfachen Kontrollpfad, den Pipelines zur Abarbeitung der Operationen (Slots) und einer gemeinsam genutzten Registerbank (siehe Abb. 2.1(a)). Die Slots bilden die vertikale Unterteilung des Prozessors (Abb. 2.1(b)), sind im vorliegenden Fall identisch ausgelegt und bestehen aus den Pipelineregistern Fetch Register (FR), Decode Register (DR) und Write Back Register (WR) und den Funktionseinheiten (FE). Zusätzlich dazu enthalten sie einen Schreib- und zwei Leseports in die Registerbank. Dies sind große (De-) Multiplexerstrukturen, die anhand einer angelegten Adresse ein Datum in ein spezielles Register der Registerbank schreiben oder daraus

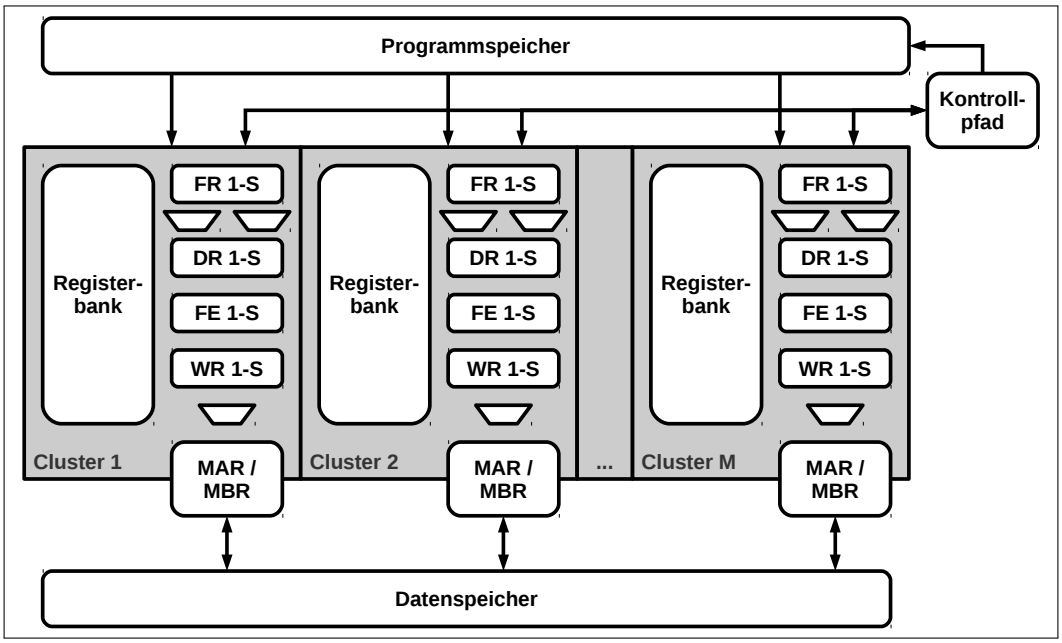


Abbildung 2.2.: Der geclusterte VLIW Prozessor

lesen. Weiterhin lassen sich die Slots horizontal in die vier Pipelinestufen Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE) und Write Back (WB) unterteilen (Abb. 2.1(b)), wobei die Abarbeitung eines Befehls innerhalb eines Slots folgendermaßen abläuft (siehe Abb. 2.1(a)):

In der ersten Abarbeitungsphase der Pipeline (IF) werden die *M* Operationen des VLIW in die Fetch Register der *M* Slots gelesen. In der zweite Phase (ID) folgt deren Dekodierung, was hier lediglich das Auslesen zweier Werte aus der Registerbank über die Leseports beinhaltet. Der Rest der Operation bleibt unverändert und steht zum Schluss der Phase zusammen mit diesen Werten in den Decode Registern. Daran schließt sich die Ausführung der Befehle mit Hilfe der Funktionseinheiten, was arithmetisch / logische Operationen, die Interaktion mit dem Datenspeicher über MAR / MBR oder das Lesen oder Schreiben der Programmadresse im Kontrollpfad beinhalten kann. Das Schreiben in MAR / MBR und den Kontrollpfad ist in diesem Fall exklusiv, kann also innerhalb eines Takts nur von einem Slot aus geschehen. Das eventuelle Ergebnis dieser Phase wird im Write Back Register abgelegt. Die letzte Phase WB übernimmt das Sichern dieser Ergebnisse in die Registerbank über die Schreibports. Da jeder Slot eigene Lese- und Schreibports besitzt, können alle in jedem Takt gleichzeitig aus der Registerbank lesen und in sie schreiben.

2.1.2. Der geclusterte VLIW Prozessor

Die hohe Parallelität beim Lesen und Schreiben der Slots in die Registerbank bedeutet kurze Zugriffszeiten. Das gilt aber nicht, wenn die Anzahl *M* der Slots hoch und die Registerbank sehr lang ist. In diesem Fall entstehen sehr große Ports, die entsprechend langsam arbeiten und damit die Performance des Prozessors senken können. Um die

Größe der Ports zu reduzieren, aber gleichzeitig M und die Anzahl der Register gleich zu halten, bietet es sich an, die Registerbank in mehrere Teile aufzutrennen und nur gewissen Gruppen von Slots Zugriff auf entsprechende Teile zu gewähren.

Die für die vorliegende Arbeit umgesetzte Implementierung dieses Ansatzes ist in Abbildung 2.2 dargestellt. Hier besteht der Prozessor aus mehreren kleineren Teilgruppen, sogenannten Clustern, die jeweils eine eigene Registerbank, eine Anzahl von S Slots und ein MAR und MBR besitzen. Die Slots haben immer nur Zugriff auf Werte in der Registerbank ihres Clusters. Der Datenaustausch zwischen den Clustern ist über den Datenspeicher möglich (andere Möglichkeiten mit zusätzlicher Hardware für die Kommunikation finden sich beispielsweise in [Ter07]). Im Gegensatz zur nicht geclusterten Version können nun alle Cluster gleichzeitig im Datenspeicher lesen und schreiben, davon aber wieder nur ein Slot. Der gemeinsam genutzte Kontrollpfad bleibt erhalten. Somit bearbeiten die Cluster ein gemeinsames VLIW (eine andere Möglichkeit, bei der sie unterschiedliche Programme bearbeiten, wurde beispielsweise in [ZFMS05] vorgestellt).

2.1.3. Weitere wichtige Eigenschaften

Eine weitere wichtige Eigenschaft der VLIW Prozessoren ist der statisch geplante Programmablauf. Hier übernimmt der Compiler die Zuweisung, wann welcher Slot in welchem Cluster eine Operation ausführt und legt dies vor der eigentlichen Abarbeitung fest. Das hat den Vorteil, dass der große Hardwareaufwand für online Scheduling-Techniken entfällt und der Prozessor somit einen sehr kleinen und einfachen Kontrollpfad besitzt. Außerdem bietet es Spielraum für Optimierungen im Programmcode, die offline vorbereitet werden können.

Zusätzlich enthalten viele VLIW Prozessoren ein sogenanntes Forwarding Netzwerk. Dieses Netzwerk dient dazu, die Wartezeit für das Speichern von Ergebnissen und ihr erneutes Laden zu verkürzen, indem die WB und ID Phasen ausgelassen und einmal errechnete Ergebnisse direkt von der EXE Phase zurück in die EXE Phase geleitet werden.

Außerdem ist zu bemerken, dass die Funktionseinheiten der Slots nicht zwingend identisch sein müssen. So kann es sich in Bezug auf ihre Größe als günstig erweisen, wenn nicht jede FE alle Operationen umfasst, sondern in einigen Fällen zum Beispiel nur Multiplikation oder Sprungoperationen.

2.2. Stress, Zuverlässigkeit, Fehlerrate und MTTF

Die Begriffe Stress, Zuverlässigkeit (Reliability), Fehlerrate und mittlere Lebensdauer (Mean Time To Failure - MTTF) sind für die vorliegende Schrift von zentraler Bedeutung. In diesem Kapitel werden sie näher erläutert und ihre Zusammenhänge verdeutlicht.

2.2.1. Stress in integrierten Schaltkreisen

Im weiteren Sinne beschreibt Stress die Belastung, der ein System zu einem bestimmten Zeitpunkt unterliegt und durch deren Einwirkung die Lebensdauer des Systems direkt oder indirekt beeinflusst wird. Diese Belastung entsteht durch externe Einflüsse wie Temperatur,

Luftfeuchtigkeit und Erschütterungen oder interne Begebenheiten, zum Beispiel Ausdehnungskoeffizienten des Metalls, Spannungen oder Feldstärken.

In der vorliegenden Dissertationsschrift steht der Begriff Stress ausschließlich für die Belastung durch die Schaltaktivität, die aus der Software oder der hardwaremäßigen Implementierung resultiert und den damit verbundenen Folgeerscheinungen wie Stromverbrauch, Verlustleistung und entstehende Wärme in genau abgegrenzten Komponenten. „Destressen“ bedeutet somit eine gezielte Verringerung der Schaltaktivität und damit auch der Folgeerscheinungen.

2.2.2. Zuverlässigkeit, Fehlerrate und mittlere Lebensdauer

Die drei Quellen [KK07], [RH03] und [EPS05] bieten einen sehr umfassenden Überblick über die Betrachtung der Zuverlässigkeit eines Systems. Zusammenfassend kann dabei Folgendes festgehalten werden:

Der Zustand eines Systems zum Zeitpunkt t kann mit der Zustandsvariable $X(t)$ folgendermaßen beschrieben werden:

$$X(t) = \begin{cases} 1 & \text{wenn das System zum Zeitpunkt t funktioniert} \\ 0 & \text{wenn das System zum Zeitpunkt t nicht funktioniert} \end{cases} \tag{2.1}$$

Die Zeit bis zum Ausfall T, auch bekannt als Time To Failure (TTF), verdeutlicht die Zeitspanne vom Zeitpunkt, an dem das System zum ersten Mal eingesetzt wird ($t = 0$) bis zu seinem Ausfall, also wenn $X(t \geq 0)$ von 1 auf 0 wechselt. Zumindest bis zu einem gewissen Grad unterliegt die Zeit zum Ausfall dabei zufälligen Einflüssen. Somit kann T als Zufallsvariable betrachtet werden.

Fehlerrate Die Fehlerrate ($\lambda(t)$) beschreibt die Rate zwischen der bedingten Wahrscheinlichkeit, dass der Zeitpunkt T des Ausfalls eines Systems in einen gegebenen Zeitintervall $(t_0, t_0 + \Delta t)$ fällt, und der Länge dieses Intervalls Δt, wenn Δt gegen Null strebt und unter der Bedingung, dass das System zum Zeitpunkt t_0 korrekt gearbeitet hat ([EPS05, S.21]).

Die Fehlerrate wird dabei in Ausfällen pro Zeiteinheit angegeben, also z.B. Ausfälle pro Jahr, pro Stunde oder pro Takt. Weiterhin findet auch die Einheit Failures In Time (FIT) Einsatz, die hier im Speziellen die Anzahl der Ausfälle pro 10^9 Arbeitsstunden einer Komponente beschreibt.

Wie die Autoren von [KK07, S. 11, ff.] erläutern, ist die Fehlerrate der wichtigste Parameter bei der Zuverlässigkeitsanalsye. Grundlegend ist sie von mehreren Faktoren abhängig, z.B. dem Alter der Komponente, physikalischen Einflüssen (z.B. Erschütterungen), Spannungsschüben, der Umgebungstemperatur und der Technologie. Der Bezug zum Alter wird oft mit Hilfe der Badewannenkurve verdeutlicht. Diese ist in Abbildung 2.3 dargestellt und aus ihr lassen sich folgende Zusammenhänge ableiten: Am Anfang der Lebenszeit von Komponenten ist ihre Fehlerrate noch ziemlich hoch. Der Grund dafür ist, dass nicht immer alle Systeme, die Herstellungsfehler enthalten, bei der Überprüfung nach der Produktion sofort aussortiert werden. Sie fallen aber anschließend bei der Benutzung sehr frühzeitig aus. Danach bleibt die Fehlerrate für lange Zeit gleichmäßig auf einem relativ geringen

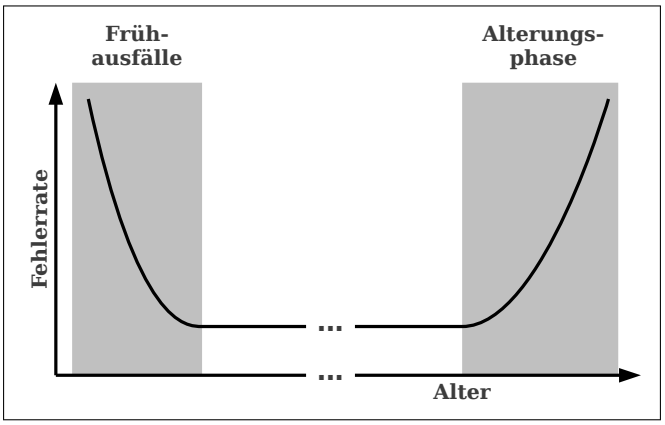


Abbildung 2.3.: Die Badewannenkurve

Niveau, bis Alterungseffekte einsetzen und nach und nach mehr verschleißbedingte Fehler entstehen.

Ausfallwahrscheinlichkeit Die Ausfallwahrscheinlichkeit beschreibt die bedingte Wahrscheinlichkeit, dass ein System im Zeitintervall (t_0, t) ausfällt, unter der Bedingung, dass es zum Zeitpunkt t_0 korrekt gearbeitet hat. Als Funktion der Zeit $(F(t))$, mit $F : \mathbb{R} \to \mathbb{R}$, kann sie folgendermaßen ausgedrückt werden, wobei $\lambda(t)$ wieder die Fehlerrate ist ([EPS05, S.20]):

$$F(t) = 1 - e^{-\int_{t_0}^{t} \lambda(x)dx} \tag{2.2}$$

Zuverlässigkeit Die Zuverlässigkeit beschreibt die bedingte Wahrscheinlichkeit, dass ein System im Zeitintervall (t_0, t) korrekt arbeitet, unter der Bedingung, dass es zum Zeitpunkt t_0 bereits korrekt gearbeitet hat. Diese Wahrscheinlichkeit kann als Funktion der Zeit $(R(t))$, mit $R : \mathbb{R} \to \mathbb{R}$ folgendermaßen ausgedrückt werden, wobei $\lambda(t)$ die Fehlerrate ist ([EPS05, S.20]):

$$R(t) = 1 - F(t) = e^{-\int_{t_0}^{t} \lambda(x)dx} \tag{2.3}$$

Mittlere Lebensdauer (MTTF) Die Mean Time To Failure (MTTF) beschreibt die zu erwartende mittlere Zeit bis zum Ausfall eines Systems und kann allgemein folgendermaßen ausgedrückt werden ([EPS05, S.23]), wobei $R(t)$ die Zuverlässigkeit ist:

$$MTTF = \int_{0}^{\infty} R(t)dt \tag{2.4}$$

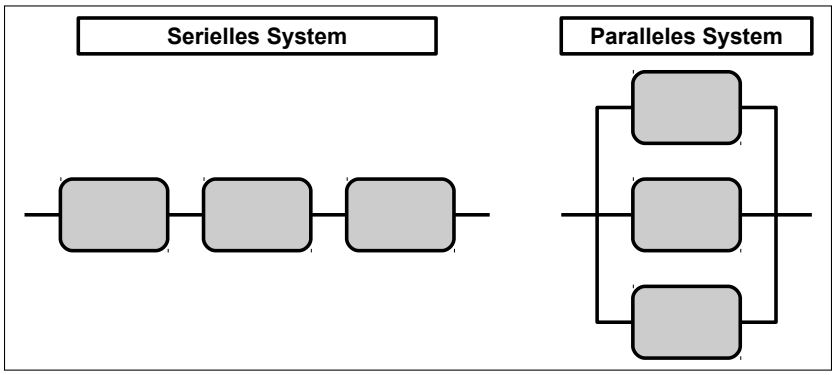


Abbildung 2.4.: Serielles / Paralleles System

Konstante Fehlerrate und Exponentialverteilung Um die Zusammenhänge dieser Variablen besser verdeutlichen zu können, bietet sich ein Beispiel an. In der Fachliteratur wird für die Untersuchungen zur Zuverlässigkeit oft eine konstante Fehlerrate angenommen ($\lambda(t) = \lambda, \lambda > 0$). Die Zeit T bis zum Ausfall ist eine exponentiell verteilte Zufallsvariable.

Bei der Ermittlung von Ausfallwahrscheinlichkeit, Zuverlässigkeit und MTTF wird dann wie folgt vorgegangen: $f(t)$ ($f : \mathbb{R} \to \mathbb{R}$, $t \in \mathbb{R}$, $\int_{-\infty}^{\infty} f(t)dt = 1$), ist die Wahrscheinlichkeitsdichtefunktion (PDF) der exponentiell verteilten Zufallsvariable T:

$$f(t) = \begin{cases} \lambda e^{-\lambda t}, & t \geq 0 \\ 0, & t < 0 \end{cases} \tag{2.5}$$

Die Ausfallwahrscheinlichkeit $F(t)$ ist die kumulative Dichtefunktion (CDF) von $f(t)$ und gibt für jedes t ein Maß für Wahrscheinlichkeit P an, dass T kleiner t ist, das System also bereits vor dem Zeitpunkt t ausfiel.

$$F(t) = P(T < t) = \int_{-\infty}^{t} f(x)dx = \begin{cases} 1 - e^{-\lambda x}, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{2.6}$$

Für die Zuverlässigkeit $R(t)$ und damit die Wahrscheinlichkeit $P(T > t)$, also, dass das System nach dem Zeitpunkt t noch funktioniert, gilt Folgendes:

$$R(t) = 1 - F(t) = 1 - (1 - e^{-\lambda x}) = e^{-\lambda x} \tag{2.7}$$

Die mittlere Lebensdauer MTTF kann dann wie folgt ermittelt werden:

$$MTTF = \int_{0}^{\infty} R(t)dt = \int_{0}^{\infty} e^{-\lambda t} = \frac{1}{\lambda} \tag{2.8}$$

Serielle und parallele Systeme Mit Hilfe der bedingten Wahrscheinlichkeiten ist nicht nur die Zuverlässigkeit einzelner Komponenten, sondern sogar die komplexer Systeme

modellierbar. Als Beispiele sollen hier das serielle und parallele System aus Abbildung 2.4 dienen. Laut [KK07, S. 16] wird ein serielles System als eine Menge von N Modulen definiert, die so miteinander verbunden sind, dass der Ausfall eines einzelnen Moduls das ganze System zum Ausfall bringt (Abb. 2.4, links). Unter der Annahme, dass die Module unabhängig voneinander ausfallen, ist die Zuverlässigkeit des Gesamtsystems das Produkt der Zuverlässigkeiten seiner N einzelnen Module. Wenn also $R_G(t)$ die Zuverlässigkeit des Gesamtsystems und $R_i(t)$ die Zuverlässigkeit des Moduls i ist, gilt:

$$R_G(t) = \prod_{i=1}^{N} R_i(t) \tag{2.9}$$

Bei einem parallelen System hingegen (Abb. 2.4 rechts) müssen laut [KK07, S. 16] alle Module ausfallen, damit das gesamte System ausfällt. Deshalb gilt hier bei gleicher Notation:

$$R_G(t) = 1 - \prod_{i=1}^{N} (1 - R_i(t)) \tag{2.10}$$

2.3. Die Bedeutung der Temperatur

Wie bereits am Anfang dieses Kapitels angedeutet, soll die Temperatur des Systems als Anzeichen für Stress und ihre Reduzierung als Nachweis des Destressens eingesetzt werden. Der folgende Abschnitt beschreibt nun, wie Wärme in ICs entsteht und welchen Einfluss die Skalierung auf sie hat. Daran schließt sich eine Aufzählung der wichtigsten, durch erhöhte Temperatur verstärkten Fehlereffekte an. Mit ihrer Hilfe kann der Zusammenhang zwischen Wärmeerzeugung und Beschleunigung der Alterungseffekte in ICs gezeigt werden. Der letzte Abschnitt fasst dies noch einmal zusammen und begründet abschließend, warum die Temperatur ein guter Indikator für Stress ist.

2.3.1. Wärmeerzeugung und ihr skalierungsbedingter Anstieg

Laut [PN06, S.1489] wird in ICs an zwei Stellen Wärme erzeugt: in den Verbindungsleitungen und in den Kanalzonen der Transistoren. In den Verbindungsleitungen entsteht allein dadurch Wärme, dass Strom durch die Leiter fließt. Dieser Strom ist zwar relativ gering, aber da die Leiter durch mehrere Metallebenen von der Chipunter- und Oberseite getrennt und somit weiter von den Wärmeableitern entfernt sind, kann die hier erzeugte Hitze durchaus von Bedeutung sein. Im vorliegenden Fall soll aber eher die Wärmeentwicklung der Komponenten, die ins Substrat eingebettet sind, betrachtet werden.

Stark vereinfacht kann die Betriebstemperatur eines IC mit der folgenden linearen Gleichung bestimmt werden:

$$T_{chip} = T_u + R_\theta \cdot \frac{P_{tot}}{A}, \tag{2.11}$$

wobei T_{chip} die durchschnittliche Temperatur des Chips, T_u die Umgebungstemperatur, P_{tot} (in W) die gesamte Leistung, A (in cm^2) die Fläche des Chips und R_θ der äquivalente

Wärmeleitwiderstand des Substrats (Si), des Gehäuses und des Kühlers (engl.: *heat sink*) ist (in $cm^2{}^\circ C/W$).

Die gesamte Leistung (P_{tot}) eines Schaltkreises in CMOS berechnet sich wiederum wie folgt:

$$P_{tot} = P_{dynamisch} + P_{kurzschluss} + P_{statisch} \tag{2.12}$$

$P_{dynamisch}$ ist dabei die dynamische Leistung, die umgesetzt wird, wenn sich der Ausgang eines Gatters ändert. $P_{kurzschluss}$ entsteht, wenn sich durch nicht-ideales Sperrverhalten von an sich abgeschalteten Transistoren zwischen V_{dd} und GND doch eine leitende Verbindung ergibt. $P_{statisch}$ ist die durch statische Leckströme entstehende Leistung.

Die Berechnung von P_{tot} bleibt dabei unabhängig von der Technologiegeneration immer gleich. Würde also die Leistung im gleichen Maße wie die Fläche sinken, bliebe laut Formel 2.12 auch die Temperatur im Chip gleich. Dass dies aber nicht der Realität entspricht, wird in [SABR04] folgendermaßen zusammengefasst:

- **Die Versorgungsspanung sinkt nicht in idealem Maße:**
  Somit kann die dynamische Leistung im Transistor ($P_{dynamisch}$) nicht ideal sinken. Ein Grund dafür, dass die Spannung nicht ideal sinkt, ist der Versuch, die Frequenz auf relativ hohem Niveau zu halten. Hierzu muss auch die Spannung groß angesetzt werden. Weiterhin wird die Empfindlichkeit gegen Rauschen immer stärker, je geringer der Abstand zwischen Versorgungs- und Schwellspannung ausfällt. Auch hier muss die Versorgungsspannung zur Erhöhung der Immunität relativ hoch bleiben.

- **Die Leckströme steigen an:**
  Die Schwellspannungen ideal zu skalieren, würde bedeuten, dass die Leckströme pro Transistor und Technologiegeneration sogar um das Fünffache ansteigen. Dieser Effekt wird durch die exponentielle Abhängigkeit von Leckströmen und Temperatur nur noch verstärkt.

Zusammenfassend kann also gesagt werden, dass P_{tot} nicht im gleichen Maße wie die Fläche sinkt. Dies liegt vor allem daran, dass $P_{dynamisch}$ durch hohe Versorgungsspannung, Frequenz und Schaltaktivität der wachsenden Anzahl von Gattern und $P_{statisch}$ durch die steigenden Leckströme ein gleichmäßiges Skalieren verhindern. Somit steigt die interne Energiedichte (P_{tot}/A) und damit laut Formel 2.12 die Temperatur im Chip.

2.3.2. Auswirkung der Temperatur auf die Fehlerrate

Der durch die Skalierung verursachte Temperaturanstieg hat zur Folge, dass bestimmte Fehlereffekte deutlich verstärkt auftreten und damit Zuverlässigkeit und mittlere Lebensdauer eines Systems sinken. Der Zusammenhang zwischen der Temperatur und vier Fehlereffekten wird in [WB08, S.179 f.] anhand von zwei Werten beschrieben: Einer konstanten, dem Effekt zugehörigen Fehlerrate λ und dem sogenannten Beschleunigungsfaktor (acceleration factor - AF). Letzterer hat seinen Ursprung dabei im Systemtest und beschreibt laut [JED11, S. 1] das Verhältnis der Zeit, die vergeht, bis ein festgelegter Teil einer Anzahl von Systemen unter bestimmtem Stress ausfällt, zur Zeit, die vergeht, bis die gleiche Anzahl von Systemen unter erhöhtem Stress (Temperatur, Spannung) ausfällt.

- **Elektromigration:**
  Elektromigration (EM) tritt dann auf, wenn Metallatome beschleunigt und infolgedessen innerhalb der Leiter transportiert werden. Die Atome wandern dabei von Stellen hoher Stromdichte zu Regionen mit niedriger Stromdichte, was zu erhöhten Widerständen (dünner werdende Leiter) und Kurzschlüssen (unerwünschte Brücken) führen kann. Die Fehlerrate kann bei der EM wie folgt modelliert werden:

$$\lambda_{EM} \propto (V_D)^n \cdot exp\left[\frac{-E_{aEM}}{kT}\right], \tag{2.13}$$

  wobei V_D die Spannung, E_{aEM} die Aktivierungsenergie der EM in eV, k die Boltzmannkonstante und T die Temperatur in Kelvin ist. n stellt eine weitere Konstante dar, die vom Material der Verdrahtung abhängt. Für Kupfer liegt sie zum Beispiel zwischen 1 und 2. Der Beschleunigungsfaktor lässt sich wie folgt bestimmen:

$$AF_{EM}^{V_O,T_O;V_A,T_A} = \left(\frac{V_A}{V_O}\right)^n \cdot exp\left[\frac{E_{aEM} \cdot (T_A - T_O)}{k \cdot T_A \cdot T_O}\right], \tag{2.14}$$

  wobei V_O und T_O die normale Spannung und Temperatur und V_A und T_A die Spannung und Temperatur unter Stress angeben.

- **HCI:**
  Ein weiterer wichtiger Alterungsfaktor ist die sogenannte Hot Carrier Injection (HCI). Diese tritt laut [Mor11, S. 13] auf, wenn sich ein Ladungsträger entlang des Kanals eines MOSFET, nahe der Drain Seite bewegt. Dort wird er durch das elektromagnetische Feld beschleunigt. Ist die durch diese Beschleunigung entstehende, kinetische Energie größer als die Energie, die das Kristallgitter in thermischem Gleichgewicht besitzt, werden sie als „heiße“ Ladungsträger (hot carrier) bezeichnet. Mit Hilfe dieser Energie kann der Ladungsträger selbst ins Oxid des Gate eindringen oder per Stoßionisation andere Ladungsträger dahin schießen. Die Fehlerrate der HCI wird wie folgt bestimmt:

$$\lambda_{HCI} \propto exp\left[\frac{-\gamma_{HCI}}{V_D}\right] \cdot exp\left[\frac{-E_{aHCI}}{kT}\right], \tag{2.15}$$

  wobei V_D die Spannung, E_{aHCI} die Aktivierungsenergie der HCI in eV, k die Boltzmannkonstante und T die Temperatur in Kelvin ist. γ_{HCI} beschreibt den Beschleunigungsfaktor der Spannnung. Der Beschleunigungsfaktor der HCI wird folgendermaßen berechnet:

$$AF_{HCI}^{V_O,T_O;V_A,T_A} = exp\left[\gamma_{HCI}\frac{(V_A - V_O)}{V_A \cdot V_O}\right] \cdot exp\left[\frac{E_{aHCI} \cdot (T_A - T_O)}{k \cdot T_A \cdot T_O}\right] \tag{2.16}$$

- **Time Dependent Dielectric Breakdown (TDDB):**
  TDDB, auch bekannt unter Gate-Oxide Breakdown, ist das langsame Altern des

Dielektrikums, wodurch es zu einem Ausfall des Transistors kommt. Hier wird die Ausfallrate wie folgt bestimmt:

$$\lambda_{TDDB} \propto exp[\gamma_{TDDB} \cdot V_G] \cdot exp\left[\frac{-E_{aTDDB}}{kT}\right], \tag{2.17}$$

wobei γ_{TDDB} der Beschleunigungsfaktor der Spannung, V_G die Spannung, E_{aTDDB} die Aktivierungsenergie des TDDB in eV, k die Boltzmannkonstante und T die Temperatur in Kelvin ist. Der Beschleunigungsfaktor für TDDB wird folgendermaßen berechnet:

$$AF_{TDDB}^{V_O,T_O;V_A,T_A} = exp[\gamma_{TDDB} \cdot (V_A - V_O)] \cdot exp\left[\frac{E_{aTDDB} \cdot (T_A - T_O)}{k \cdot T_A \cdot T_O}\right] \tag{2.18}$$

- **NBTI:**
  Diese elektrochemische Reaktion findet in PFETs statt. Negative Bias Temperature Instability (NBTI) führt dabei zu einer Erhöhung der Schwellspannung in einem Transistor, was wiederum Fehler durch Verletzung von zeitlichen Bedingungen verursachen kann. Die Formel zur Bestimmung der Fehlerrate bei NBTI lautet:

$$\lambda_{NBTI} \propto exp[\gamma_{NBTI} \cdot V_G] \cdot exp\left[\frac{-E_{aNBTI}}{kT}\right] \tag{2.19}$$

  Hier ist γ_{NBTI} der Beschleunigungsfaktor der Spannung, V_G die Spannung, E_{aNBTI} die Aktivierungsenergie der NBTI in eV, k die Boltzmannkonstante und T die Temperatur in Kelvin. Die Berechnung des Beschleunigungsfaktors geschieht wie folgt:

$$AF_{NBTI}^{V_O,T_O;V_A,T_A} = exp[\gamma_{NBTI} \cdot (V_A - V_O)] \cdot exp\left[\frac{E_{aNBTI} \cdot (T_A - T_O)}{k \cdot T_A \cdot T_O}\right] \tag{2.20}$$

Beispielwerte für den Beschleunigungsfaktor der Spannung und die jeweilige Aktivierungsenergie bei der 130nm Technologie können aus [WB08, S.184 f.] entnommen werden und sind noch einmal in Tabelle 2.1 dargestellt. Die Fehlerrate des Gesamtsystems λ_S kann, unter der Annahme, dass die Fehlermechanismen sich nicht gegenseitig beeinflussen, wie folgt bestimmt werden:

$$\lambda_S = \lambda_{EM} + \lambda_{HCI} + \lambda_{TDDB} + \lambda_{NBTI} \tag{2.21}$$

Der Beschleunigungsfaktor kann dann folgendermaßen ausgedrückt werden:

$$AF_S = \frac{\lambda_S^{V_A,T_A}}{\lambda_S^{V_O,T_O}} = \frac{\lambda_{EM}^{V_A,T_A} + \lambda_{HCI}^{V_A,T_A} + \lambda_{TDDB}^{V_A,T_A} + \lambda_{NBTI}^{V_A,T_A}}{\lambda_{EM}^{V_O,T_O} + \lambda_{HCI}^{V_O,T_O} + \lambda_{TDDB}^{V_O,T_O} + \lambda_{NBTI}^{V_O,T_O}} \tag{2.22}$$

Tabelle 2.1.: Parameter der Fehlerrate für EM, HCI, TDDB und NBTI

	Beschleunigunsparameter der Spannung n bzw. γ	Aktivierungsenergie E_a (in eV)
EM	2	1,2
HCI	16	-0,2
TDDB	12	0,7
NBTI	6	0,4

2.3.3. Temperatur als Indikator für erhöhten Stress

Die letzten beiden Abschnitte haben folgenden Zusammenhang verdeutlicht: Hohe umgesetzte Leistung verursacht bei konstanten restlichen Faktoren der Formel 2.11 höhere Wärmeerzeugung, steigende Fehlerraten und daraus ergibt sich eine Reduzierung der zu erwartenden mittleren Lebensdauer. Diese Umstände werden durch die Technologieskalierung nur noch verstärkt, da sich das Verhältnis der Leistung zur Fläche des Systems erhöht.

Soll die mittlere Lebensdauer wiederum gesteigert werden, bieten sich als Methoden die Senkung der Temperatur und der Leistung an. Doch bleibt die Frage, welche sich im vorliegenden Fall besser eignet. Für beide existieren bereits viele Möglichkeiten, deren Unterschiede laut [Jay09, S. 13, ff] wie folgt zusammengefasst werden können:

- **Global oder lokal:**
  Die Methoden zur Senkung der Leistung zielen meist auf die gesamte Leistung des Chip ab, während sich die Methoden zur Temperatursenkung oft speziell auf die heißeste Komponente konzentrieren. Diese kann um 15°C heißer als andere Komponenten sein.

- **Unterschiede in der Verteilung der Leistung und der Temperatur:**
  In Mikroprozessoren stellen die Caches die größten Verbraucher der Leistung dar, während der Ausführungskern (Registerbank + Funktionseinheiten) und die Sprungvorhersage die heißesten Komponenten sind. Dies liegt daran, dass die Caches wesentlich größer sind und somit ein deutlich besseres Verhältnis von Leistung pro Fläche besitzen.

- **Momentanleistung und Temperatur:**
  Änderungen in der Temperatur gehen relativ langsam vor sich und reflektieren kontinuierliche Veränderungen in der Leistungsaufnahme über ein längeres Zeitfenster. Deshalb betreffen Methoden zur Senkung der Momentanleistung nicht die Temperatur.

Wie sich erkennen lässt, ist der Einfluss der Leistungsaufnahme auf die Temperatur und damit die beschleunigte Alterung des Systems nicht immer gegeben. Im Gegensatz dazu steht erhöhte Temperatur einerseits für verkürzte Lebensdauer und andererseits für ein schlechtes Verhältnis der Leistungsaufnahme zur Fläche in einem abgegrenzten Bereich

und das über ein relativ großes Zeitfenster. Somit ist die Temperatur ein gutes Maß für Stress und eine Reduzierung der Temperatur kann als Anzeichen einer Verringerung des Stresses und damit einer Erhöhung der MTTF ausgelegt werden.

Kapitel 3

Lebensdaueroptimierung - Stand der Technik

Wie sich in den ersten beiden Kapiteln gezeigt hat, besitzt die fortschreitende Miniaturisierung der Technolgie neben ihren Vorteilen auch deutliche Nachteile. Da die in größerer Anzahl auftretenden Fehler und vor allem ihre Ursachen aber sehr intensiv erforscht werden, existieren bereits viele Gegenmaßnahmen, die ihre Auswirkungen stark abschwächen oder sogar ganz beheben. Um nur auf die in Abschnitt 1.1 beschriebenen Beispiele einzugehen, können unter Anderem statische Fehler in Speicherblöcken, FPGA-Zellen und logischen Schaltkreisen gefunden und sogar repariert werden ([LYH+03, MHS+04, KV08]). Um Verschleiß zu reduzieren, finden beispielsweise Low-Power-Methoden und das sogenannte Dynamic Thermal Management (DTM) Einsatz ([WV96, BM01]). Und auch gegen zufällige Fehlereffekte, wie Single Event Upsets (SEUs) und Multiple Event Upsets (MEUs), können durch Anwendung von Codes oder die sogenannte Triple Modular Redundancy (TMR) gute Ergebnisse erzielt werden ([LTRN92, Neu56]).

Das folgende Kapitel beschreibt nun zuerst die verschiedenen Fehlzustände, in denen sich ein System befinden kann. Dies soll es erleichtern, die verschiedenen Methoden zur Lebensdaueroptimierung nach ihrem Ansatzpunkt zu klassifizieren. Anschließend wird sich ganz gezielt mit zwei sehr speziellen Maßnahmen auseinandergesetzt, die jeweils gute Ergebnisse versprechen, sich in einer Verbindung aber äußerst positiv auf Zuverlässigkeit und MTTF auswirken könnten. Hierbei handelt es sich einerseits um die Activity Migration und das Load Balancing als Methoden des Thermal Managements. Durch ihren Einsatz sollen sowohl die mittlere als auch die Spitzentemperatur des Prozessors gesenkt und somit Verschleißeffekte verlangsamt werden. Andererseits soll der Fokus dieser Arbeit auf der Selbstreparatur als Teilgebiet der Fehlertoleranz liegen. Dieses Gebiet umfasst verschiedene Methoden mit deren Hilfe ein System, selbst bei Auftreten eines oder mehrerer Fehler, lauffähig erhalten wird.

3.1. Fehlzustände eines Systems

Grundlegend lassen sich die Fehlzustände eines Systems in die fünf Kategorien Ideal, Defekt, Fehler, Störung und Ausfall aufteilen. Wie in Abbildung 3.1 erkennbar, ist der ideale Zustand der Grundzustand, bei dem das System fehlerfrei arbeitet. Altert anschließend das System, kann es zu einem Defekt (engl. Defect) kommen, der die eigentliche physikalische Ursache des Fehlers beschreibt. Defekte sind zum Beispiel Leiterbahnbrüche, Oxiddurchbrüche im Gate eines Transistors u.Ä.. Der resultierende Fehler (engl. Fault) beschreibt die

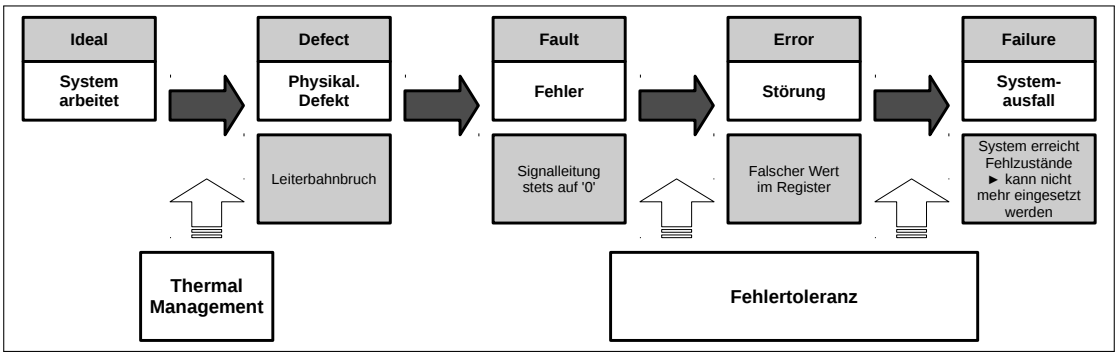


Abbildung 3.1.: Ideal, Defekt, Fehler, Störung, Ausfall

Auswirkung des Defekts auf das System und repräsentiert damit die Ursache, falls später ein falsches Bit in einem Register steht, und kann über verschiedene Fehlermodelle (Haftfehler, Verzögerungsfehler, etc.) modelliert werden. Dieser Fehler muss sich aber nicht weiter auf das System auswirken, beispielsweise wenn die betroffenen Schaltungsteile zur Zeit nicht benutzt werden oder wenn ein Gatterausgang stets auf logisch ’0’ liegt und dieser Wert zufällig das richtige Ergebnis wäre. Erst wenn er sich auswirkt, also zum Beispiel als falsches Bit in einem Register steht, spricht man von einer Störung (engl. Error). Wird dieses Bit anschließend ausgelesen und führt etwa über den Programmzähler zum Sprung an eine falsche Adresse, führt dies zum Ausfall des Systems (engl. Failure), da nicht mehr vorhersagbar ist, wohin der Sprung führt und das System somit einen Fehlzustand annimmt.

Es existieren auch andere Möglichkeiten hier zu Unterscheiden. So werden in [Nel90] nur Fault, Error und Failure beschrieben, wohingegen die Autoren von [Par97] ganze sieben Zustände auflisten.

Alle Maßnahmen zur Lebensdaueroptimierung verfolgen nun das gleiche Ziel: die in Abbildung 3.1 dargestellte Kette zu unterbrechen. Jedoch kann ihr Ansatzpunkt dabei durchaus variieren. Mit dem Thermal Management wird beispielsweise versucht, Verschleißerscheinungen zu verlangsamen und damit zu verhindern, dass ein Defekt entsteht. Ordnet man dies in die Wannenkurve aus Abbildung 2.3 ein, resultiert daraus, dass sich die Alterungsphase mit vielen Ausfällen zeitlich nach hinten verschiebt. Mit Hilfe der Fehlertoleranz wird erreicht, dass sich einmal entstandene Fehler nicht auswirken oder Störungen beispielsweise durch Maskierung nicht zum Ausfall führen. Dies „senkt“ die gesamte Badewannenkurve, da Ausfälle durch statische oder transiente Fehler in allen Lebensphasen seltener auftreten.

3.2. Thermal Management

In Bezug auf die vielfältigen Ansätze des Thermal Managements wurde in [Jay09, S. 9 ff] ein sehr guter Überblick präsentiert. Kurz zusammengefasst gibt es dabei zwei grundlegen-

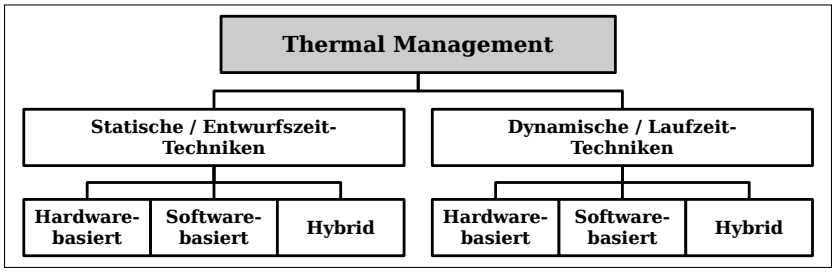


Abbildung 3.2.: Überblick - Ansätze des Thermal Managements

de Arten: Einerseits Methoden, die für eine bessere Wärmeabfuhr sorgen, und andererseits welche, die die Wärmeerzeugung an sich reduzieren. Zur ersten Gruppe gehören Maßnahmen wie die Verbesserung von Wärmeleitern und -verteilern und optimierteres Design für bessere Luftzufuhr. Auch Wasserkühlung ist ein möglicher Weg, die erzeugte Wärme besser abzuführen. Wie in [Jay09] aber erläutert wird, steigen die Kosten dieser Methoden mit ihrer Komplexität deutlich an. Problematisch hieran ist, dass die Wärme in einem Prozessor in sogenannten „Hotspots“ bis zu 15°C über dem Durchschnitt liegen kann. Damit diese Kühlmöglichkeiten aber nicht auf den schlimmsten Fall ausgelegt werden müssen, was hohe Kosten verursachen würde, sind neben der reinen Ableitung der durchschnittlichen Wärme auch Methoden notwendig, die die Erzeugung der Hitze im Prozessor so gering wie möglich halten und speziell die Hotspots vermeiden.

Die Reduzierung der Hitzeerzeugung kann dabei auf statische - Static Thermal Management (STM) - oder dynamische - Dynamic Thermal Management (DTM) - Weise geschehen und lässt sich jeweils in hardwarebasierte, softwarebasierte und gemischte (hybride) Ansätze unterteilen. Abbildung 3.2 stellt diese Unterscheidung grafisch dar. Im Folgenden werden die Gruppen genauer vorgestellt.

3.2.1. Static Thermal Management (STM)

Laut [Jay09] finden die statischen Methoden des Thermal Managments einerseits im Bereich der eingebetteten Systeme Anwendung, wo der Einsatzzweck des Systems bereits im Vorfeld bekannt ist und die Software sehr gezielt auf das Hardwaredesign angepasst werden kann. Andererseits aber auch bei der Design Space Exploration auf mikroarchitektureller Ebene, bei der die thermale Sicherheit direkt als Kernparameter in den Entwurf mit einfließt. Im Folgenden nun eine detailliertere Erläuterung des soft- und hardwarebasierten STM und deren Mischformen.

Softwarebasiertes STM Die Möglichkeiten des softwarebasierten STM umfassen im Allgemeinen die Auswahl und Steuerung gewisser Systemparameter, wie zum Beispiel die Versorgungsspannung und die Taktfrequenz, und das Scheduling der Tasks beziehungsweise die Auswahl der ausführenden Komponenten. Dabei gilt es aber oft, nicht nur gewisse Temperaturgrenzen einzuhalten, sondern auch Anforderungen an Echtzeitfähigkeit, Leistung und Energieverbrauch zu erfüllen und eine möglichst günstige Balance

zwischen all diesen Parametern zu finden. Aus den genannten Ansprüchen heraus fügt sich das softwarebasierte STM sehr gut in den Bereich der eingebetteten Systeme ein, da die jeweiligen Komponenten und die darauf ausgeführten Programme vor der eigentlichen Laufzeit bekannt sind und so die Software und die Steuerung der Systemparameter sehr gut optimiert werden können.

[Jay09, S. 15 f.] geben einen umfangreichen Überblick über die verschiedenen Möglichkeiten des softwarebasierten STM. In der vorliegenden Arbeit soll sich aber nur auf eine davon, nämlich die in [MLN+06] vorgestellte compiler-basierte Methode, konzentriert werden. Dies liegt erstens daran, dass sich ihr Vorgehen in drei große Schritte unterteilen lässt, die im Bereich Thermal Management häufig auftreten. Zweitens finden zwei Methoden Einsatz, die auch später in dieser Arbeit angewendet werden sollen, und drittens identifizieren sie die Funktionseinheiten (FE) als mit die heißesten Komponenten in einem Very Long Instruction Word (VLIW) Prozessor, was für die Methodik der vorliegenden Schrift von großer Bedeutung ist.

Wie bereits erwähnt, ist es beim Thermal Management (neben dem Abtransport der Wärme) von größter Bedeutung, einerseits die mittlere Temperatur zu senken, andererseits aber auch Hotspots zu vermeiden. In [MLN+06] wird dies, wie oben bereits angedeutet, über die folgenden drei Schritte erreicht:

1. **Schaffung von Abkühlmöglichkeiten:**
   Der erste Schritt basiert auf dem in [KVKI03] vorgeschlagenen IPC-Tuning und beinhaltet als Grundprinzip das „Zusammenschieben“ von Operationen in einem Very Long Instruction Word (VLIW) Prozessor (siehe Kapitel 2.1). Beruhend auf der Beobachtung, dass die Anzahl der pro Takt ausgegebenen Instruktionen - Instructions Issued Per Cycle (IPC) - oft weit geringer ist, als die Anzahl der zur Verfügung stehenden Funktionseinheiten (FE), kann der Code während des Kompiliervorganges so umsortiert werden („Tuning“), dass einige FE in der Größenordnung von ganzen Programmschleifen keine Operationen zugeteilt bekommen. Dies erlaubt es, die leerlaufenden FE während dieser Zeit durch Trennung von der Versorgungsspannung abzuschalten, wodurch sie abkühlen können.
   Das Ausnutzen von zeitlichem Freiraum oder die Schaffung desselben, um Prozessorkomponenten zur Kühlung zu entlasten, ist ein weit verbreitetes Mittel bei der STM.

2. **Senkung der mittleren Temperatur:**
   Der zweite Schritt beinhaltet eine Methode, die auch im späteren Verlauf dieser Arbeit von großer Bedeutung sein wird, nämlich die **Activity Migration**. Sie wurde ursprünglich in [HBA03] vorgestellt, findet bei [MLN+06] aber in leicht abgeänderter Form Anwendung. Grundlegendes Ziel ist es hier, die oben geschaffenen Abkühlmöglichkeiten auf alle FE anwenden zu können, die Aktivität also auf regelmäßiger Basis zwischen heißen und kalten FE zu migrieren, um so die mittlere Temperatur aller Einheiten zu senken.
   Zur Durchführung dieser Migration werden die auszuführenden Operationsblöcke oder Schleifen rotiert, also zwischen den Slots hin und her geschoben. Dabei gilt es

nun eine optimale Strategie zu finden, in welcher Reihenfolge die jeweiligen FE an- bzw. abzuschalten sind und welche sich damit entweder aufheizen oder abkühlen. In [MLN+06] wird vorgeschlagen, dies über die Hamming-Distanz der FE-Pattern zu berechnen. Diese Pattern oder Vektoren beschreiben, welche FE in der letzten Phase aktiv waren ('1') und welche nicht ('0'), wobei ab links mit der ersten FE angefangen wird zu zählen. Angenommen, in der letzten Programmschleife wurden von insgesamt 6 FE nur den FE2, FE3 und FE6 Operationen zugewiesen, so dass die restlichen inaktiv waren. Damit ergibt sich folgendes FE-Pattern: {011001}. In der nächsten Schleife sollen nun insgesamt 5 FE aktiv sein. Die möglichen Muster wären dann: {011111, 101111, 110111, 111011, 111101, 111110}. Der Hamming Abstand beschreibt die Anzahl der unterschiedlichen Bit zum vorhergehenden FE-Pattern, also in den genannten Fällen: 2, 4, 4, 2, 2, 4. Um nun die bestmögliche Kühlung zu ermöglichen, wird einer der Vektoren mit der größten Hamming-Distanz gewählt, also einer aus der Menge mit Abstand 4: {101111, 110111, 111110}.

Die Activity Migration ist ein wichtiges Mittel, mit dessen Hilfe sich gleichartige Komponenten gegenseitig entlasten können. Wie in [HBA03] gezeigt, können dafür aber nicht nur zeitliche Freiräume, sondern auch extra für diesen Zweck eingebaute, redundante Baugruppen verwendet werden. Dies geschieht später auch in Kapitel 4.2.

3. **Vermeidung von Hotspots:**
   Hotspots sind gewisse Komponenten im Prozessor, die durch anhaltende erhöhte Last bis zu 15°C heißer sein können als die durchschnittliche Temperatur der restlichen Einheiten [Jay09]. Da dies laut [VWWL00] ungefähr einer Halbierung der MTTF gleichkommt und eine spezielle Kühlung hier sehr teuer wäre, müssen diese Punkte weitgehend vermieden werden. Um dies zu erreichen, wird in [MLN+06] das sogenannte **Load Balancing** vorgeschlagen, welches eine weitere wichtige Methode für die vorliegende Schrift darstellt.
   Ziel des Load Balancing ist es, durch Ausgleichen der Last zwischen den Funktionseinheiten eine „thermische Balance" unter ihnen herzustellen. In [MLN+06] wird festgestellt, dass verschiedene Operationen die FE unterschiedlich aufheizen. Beispielsweise ist die Multiplikation wesentlich rechenintensiver als ein LOAD-Befehl. Beim Load Balancing werden die Operationen nun nach ihrem Energieverbrauch klassifiziert und so über die aktiven FE verteilt, dass alle in der Größenordnung von Basisblöcken im Code ungefähr den gleichen Verbrauchswert aufweisen und damit keine Hitzekonzentrationen oder Hotspots entstehen. Basisblöcke sind dabei Abschnitte im Code, die nur an der ersten Zeile betreten und an der letzten verlassen werden können, dazwischen befinden sich keine Sprunganweisungen. Der Einsatz des VLIW Prozessors vereinfacht die Umverteilung, da es hier keine Rolle spielt, auf welcher FE eine Operation ausgeführt wird, solange nur seine zeitlichen Abhängigkeiten erfüllt sind.

Der hier eingesetzte Algorithmus für das Load Balancing umfasst dabei folgende Schritte:

1. Erstellen eines performance orientierten Schedules.
2. Ermittlung der gesamten umgesetzten Leistung pro Funktionseinheit anhand der zugeteilten Operationen im Schedule.
3. Bestimmung der idealen Leistung pro FE, falls alle die gleiche Arbeit zu verrichten hätten (ausbalanciert wären).
4. Auswahl der FE mit der höchsten Leistung.
5. Auswahl von Operation der FE in ansteigender Ablaufzeit, für die jeweils gilt, dass die für die Operation aufzubringende Leistung kleiner ist als die Differenz zwischen gesamter Leistung der FE und der idealen Leistung.
6. Berechnung der Mobilität der Operation. Diese beschreibt den Zeitraum im Schedule, in dem eine Operation ausgeführt werden kann, ohne zeitliche Abhängigkeiten zu verletzen. Das heißt, nach ihrer frühest möglichen Ausführung, wenn beispielsweise alle Daten vorhanden sind, und noch vor ihrer spätest möglichen Ausführung, wenn ihre Ergebnisse benötigt werden ([WGK08, S. 234]).
7. Bestimmung eines Verschiebungsziels der Operation, also eine andere FE, zu der die Operation innerhalb ihrer Mobilität verschoben werden kann und deren gesamte Leistung mit der Operation weiterhin kleiner ist als die ideale Leistung.
8. Verschieben der Operation
9. Neuberechnung der gesamten umgesetzten Leistung der Funktionseinheiten.
10. Wiederholung der Schritte 4 bis 9, bis alle FE bearbeitet sind.

Abbildung 3.3 stellt den gesamten Vorgang noch einmal sehr vereinfacht grafisch dar. Ausgegangen wird von einem VLIW Prozessor mit fünf gleichartigen FE. Die dunkelgrauen Blöcke stehen für ihre Auslastung in der jeweiligen Schleife. In Bild 3.3 a) ist der originale Ablauf dargestellt, bei dem alle FE stets Instruktionen erhalten. Somit liegen sie unter ständiger Last und können sich nicht abkühlen. Durch ihre besonders starke Beanspruchung würden in diesem Beispiel die FE1 und FE2 die Hotspots bilden. In Abbildung 3.3 b) hat das IPC-Tuning bereits Anwendung gefunden, die Operationen wurden also auf weniger FE „zusammengeschoben“. Dadurch ist zeitlicher Freiraum entstanden, der für die Abkühlung der FE genutzt werden kann. Dieser liegt bisher aber nur bei FE4 und FE5. Abhilfe schafft das in Abbildung 3.3 dargestellte Ergebnis der **Activity Migration**. Jetzt sind die Abkühlphasen über (fast) alle FE gleichmäßig verteilt und können die allgemeine Temperatur senken. Nur FE3 liegt noch unter ständiger Last und müsste beispielsweise in einer vierten Schleife keine Operationen zugeteilt bekommen. Die Auswahl, welche FE wann abzuschalten ist, wurde dabei wieder über die oben beschriebenen FE-Pattern getroffen. Aus der Abbildung lässt sich aber erkennen, dass die Operationen immer noch

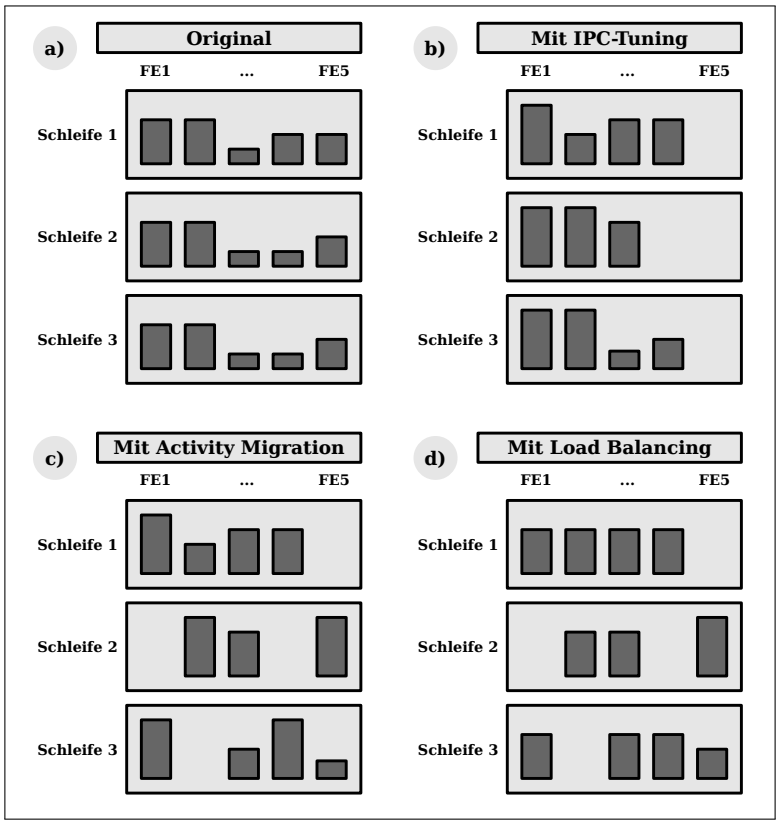


Abbildung 3.3.: Beispiel für softwarebasiertes Static Thermal Management

sehr ungleichmäßig verteilt sind und die FE weiterhin unterschiedlich intensive Arbeit verrichten. Dies wird mit dem letzten Schritt, dem **Load Balancing** - dargestellt in Abbildung 3.3 d) - vermieden. Nun sind alle Funktionseinheiten gleichmäßig ausgelastet, besitzen (fast) alle eine Abkühlphase, müssten somit einheitlich altern und dieselbe MTTF aufweisen.

Mit diesen drei Schritten erreichen [MLN+06] in den Hotspots durchschnittlich eine Temperatursenkung um 9,4%, was in einem der angeführten Beispiele immerhin ungefähr 10°C ausmacht.

Hardwarebasiertes STM Laut [Jay09] liegt das Hauptaugenmerk des hardwarebasierten STM auf der sogenannten mikroarchitekturellen Design Space Exploration (DSE). Die DSE beschreibt einen sehr komplexen Prozess, bei dem versucht wird, passende Konfigurationen für mikroarchitektonische Strukturen (Cache-Größen, Anzahl der FE, etc.) zu finden, um eine Reihe von in Konkurrenz stehenden Zielen wie Performance, Leistung, Komplexität der Schaltkreise und Kosten zu bedienen. Bisher hat sich die Forschung aber nur mit einem kleinen Teil dieses Raumes zur Temperatursenkung beschäftigt. Als Beispiel werden in [Jay09, S. 17] mehrere Arbeiten zum Thema Temperatur gegen Performance mit folgendem Ergebnis zusammengefasst: „[...] Sie zeigen, dass es für hohen Durchsatz und

nicht-speicherintensive Anwendungen [aus Sicht der Temperatur] besser ist, eine große Anzahl einfacherer Prozessoren mit kleineren Caches zu betreiben, wobei für speicherintensive Anwendungen eine kleinere Anzahl von Prozessoren mit größeren Caches die optimalere Auswahl im Design ist“.

Hybrides STM Ergänzend zu [Jay09] soll an dieser Stelle die hybride STM vorgestellt werden. Dabei handelt es sich um eine Verbindung zwischen der Design Space Exploration, also der gezielten Auswahl von Hardwarekomponenten, und der statisch geplanten Steuerung des Systems durch Software, um eine Temperatursenkung zu erreichen. Aufgrund seiner Bedeutung für die vorliegende Arbeit, da es nicht nur eine Verbindung von hard- und softwarebasierten STM darstellt, sondern gleichzeitig Methoden zur Selbstreparatur beinhaltet, soll das Beispiel für diese Form des STM in Abschnitt 4.2 ausgelagert werden. Somit kann nun die Beschreibung des Dynamic Thermal Managements folgen.

3.2.2. Dynamic Thermal Management (DTM)

Laut [UKV12a] lassen sich unter DTM all die Methoden zusammenfassen, die auf die vorherrschende Temperatur im Chip aktiv reagieren. Auslöser können beispielsweise Temperatursensoren, Aktivitätszähler oder dynamisches Profiling sein. Die Maßnahmen reichen vom Herunterskalieren der Taktfrequenz und der Versorgungsspannung bis hin zum Einfügen von Leerlaufbefehlen [BM01].

Auch das DTM lässt sich, wie in [Jay09] beschrieben, in hard-, softwarebasierte und hybride Techniken unterteilen. Diese werden nun im Einzelnen vorgestellt.

Hardwarebasiertes DTM Beim hardwarebasierten DTM spielen vor allem auf dem Chip aufgebrachte Temperatursensoren eine wichtige Rolle. Mit ihrer Hilfe misst ein Thermal Management Controller in regelmäßigen Abständen die Temperatur. Übersteigt diese einen gewissen Grenzwert, werden passende Methoden zur ihrer Senkung ausgelöst. Die Maßnahmen des hardwarebasierten DTM umfassen dabei:

- **Fetch Throttling:**
  Das zeitweise Aussetzen der Befehlsholphase in einer Pipeline, um den gesamten Prozessor abzukühlen ([SAS02]).
- **Dynamisches Skalieren der Versorgungsspannung:**
  Verringerung der Verarbeitungsgeschwindigkeit und damit der durch Schaltvorgänge umgesetzten Energie ([MM06]).
- **Clock Gating:**
  Gewisse Prozessorkomponenten werden von der Clock entkoppelt und schalten somit nicht mehr ([BM01]).
- **Activity Migration:**
  Das oben bereits beschriebene Auslagern von Last auf kühlere oder inaktive Prozessorkomponenten ([HBA03]).

- **Cluster Assignment / Power Gating:**
  Die Zuweisung von Operationen auf bestimmte Teilgruppen von Prozessoren (engl. Cluster) in einem Mehrkernsystem, während die anderen von der Versorgungsspannung (V_{dd}) getrennt sind ([CGG04]). Dieser Prozess ist auch bekannt unter „Power Gating“ ([HBS+04]).

- **Frequency Selection:**
  Eine Methode, die in [Jay09] nicht genannt wird, aber dennoch von großer Bedeutung ist. Wie beispielsweise in [MM06] beschrieben, wird dabei die Taktfrequenz des jeweiligen Prozessorkerns dynamisch so angepasst, dass er weniger Operationen pro Zeiteinheit ausführt, damit weniger Wärme produziert und somit eine gewisse Temperatur nicht überschreitet.

Softwarebasiertes DTM Wie in [Jay09] beschrieben, ist es auf der Ebene des softwarebasierten DTM ein weit verbreitetes Mittel, die verschiedenen Anwendungen dynamisch auf ihr Temperaturverhalten zu prüfen und jeweils zu einer kalten Anwendung zu wechseln, wenn der Prozessor zu heiß wird. Grundlegend wurden dabei Methoden für Architekturen vorgeschlagen, die entweder mehrere Anwendungen gleichzeitig ausführen - Simultaneously Multi-Threaded (SMT) - oder mehrere Kerne besitzen.

- **SMT Architekturen:**
  Hierfür bringen [Jay09] zwei Beispiele an, die sich mit der sogenannten „fetch policy“ auseinandersetzen. Dies betrifft die Strategie, mit der entschieden wird, wann Instruktionen aus welchem Thread zu holen sind. Ändert man diese Strategie dynamisch, kann je nach Temperaturverhalten des Prozessors ein kühlerer/ heißerer Thread gewählt und so die Wärmeentwicklung direkt beeinflusst werden.

- **Mehr-Kern Architekturen:**
  Bei Mehr-Kern Architekturen handelt es sich letztendlich wieder um eine Form der Activity Migration. Wird ein Kern zu heiß, kann ihm dynamisch ein neuer Thread zugewiesen und er damit abgekühlt werden. Die Strategien, die hier Einsatz finden, basieren dabei unter anderem auf dem Temperaturunterschied zwischen den Kernen, der Häufigkeit, mit der jeder Kern eine Obergrenze erreicht, und der Geschwindigkeit, mit der ein Kern sich erwärmt.

Hybrides DTM Die hybriden Formen des DTM sind wiederum eine Mischung der hardware- und softwarebasierten Methoden. Als Beispiel wird in [Jay09] die in [SA03] vorgeschlagene Methode für das Thermal Management in Multimedia-Anwendungen angeführt. Dort werden zusätzlich zum Regeln der Versorgungsspannung und der Prozessorkonfiguration in Software auch Teile des Programms passend über die Funktionseinheiten des Prozessors verteilt.

Auch die in [Jay09] vorgeschlagenen Methoden selbst sind dem hybriden DTM zuzuordnen, da sie über die Konfiguration in Ein- und Mehr-Kern Prozessoren und gleichzeitig mit Hilfe des Scheduling ihrer Anwendungen und Threads die Performance ihrer Prozessoren

maximieren, ohne dabei gesetzte Temperaturgrenzen zu überschreiten.

Das Thermal Management ist nicht die einzige Methode, um die Auswirkungen erhöhter Temperatur abzuschwächen und somit das Entstehen von Defekten zu verlangsamen. Ein weiterer wichtiger Ansatz ist die Auswahl der richtigen Technologie. Je nach Anforderungen an das System kann es durchaus sinnvoll sein, nicht die kleinste und neueste Generation zu wählen. Ältere sind bezüglich ihrer Fehlermechanismen besser bekannt und beherrscht, was sich positiv auf die Frühausfallrate auswirkt. Außerdem sind sie von sich aus nicht so empfindlich gegenüber zufälligen Fehlern und Alterungseffekten. Deshalb werden auch beispielsweise in der Automobilbranche robustere Technologien für sicherheitskritische Anwendungen bevorzugt.

Einen anderen wichtigen Ansatz stellen die sogenannten „Low Power Methoden“ dar. Hier wird versucht, die umgesetzte Leistung so gering wie möglich zu halten, was sich nach Formel 2.11 wieder positiv auf die Wärmeerzeugung auswirkt. Eine Zusammenfassung der wichtigsten Möglichkeiten wird in [CB95] und [DM95] geboten. Die vorliegende Arbeit soll sich jedoch mit der Reduzierung der Temperatur beschäftigen, weshalb Technologieauswahl und Low Power Methoden nur am Rande bemerkt bleiben sollen.

3.3. Maßnahmen im Fehlerfall

Die Methoden des Thermal Managements dienen dazu, Verschleißerscheinungen zu reduzieren und damit das Auftreten von Defects zu verzögern. Ganz verhindern können sie es jedoch nicht. Aus diesem Grund müssen Maßnahmen getroffen werden, um auf Fehler entsprechend zu reagieren. Laut [Sie91] sollte bereits beim Entwurf eines Systems eine koordinierte Fehlerantwort berücksichtigt werden, die einige, wenn nicht sogar alle, der nun folgenden Schritte beinhaltet. Die Reihenfolge stimmt dabei weitestgehend mit der aufgelisteten Chronologie überein, obwohl der tatsächliche Ablauf in einigen Instanzen durchaus abweichen kann ([SS98]).

1. **Fehlererkennung:**
   Das Feld der Fehlererkennung wird nach wie vor eingehend untersucht und so existieren viele verschiedenen Methoden, wie Paritäts- und Konsistenzprüfer oder Protokollüberwacher. Grundlegend lassen sich die Techniken zur Fehlererkennung in zwei Klassen unterteilen: Einerseits offline Detektion, bei der die Fehlererkennung vor und zu bestimmten Zeitpunkten während des Betriebs durchgeführt wird. Ein typischer Vertreter ist die Ausführung eines Testprogramms. Andererseits gibt es die online Erkennung, welche während des laufenden Betriebs ausgeführt wird. Somit stellt sie Möglichkeiten zur Echtzeiterkennung zur Verfügung. Beispiele hierfür sind Paritätsprüfung und die Duplizierung mit Vergleich.

2. **Fehlermaskierung:**
   Techniken zur Fehlermaskierung verhindern die Aktivierung eines Fehlers, damit er sich nicht zu einer Störung weiterentwickeln kann. Prinzipiell wird dabei einer Information so viel Redundanz hinzugefügt, dass sie mögliche fehlerhafte Informa-

tionen überlagert und somit das Original wieder hergestellt werden kann. In seiner Grundform ist mit der Maskierung keine Fehlererkennung möglich. Jedoch können viele Techniken um diese Eigenschaft erweitert werden.
Ein Beispiel für die Fehlermaskierung ist der Mehrheitsentscheid.

3. **Erneute Ausführung:**
   Ein zweiter Versuch der Ausführung ist in vielen Fällen erfolgreich. Dies gilt vor allem für beispielsweise durch Strahlung verursachte Störungen, die keinen permanenten physischen Schaden verursachen.

4. **Diagnose:**
   Falls die Methoden zur Fehlererkennung keine Informationen über die Position und/oder Eigenschaften des Fehlers bereitstellen, kann ein diagnostischer Schritt notwendig sein.

5. **Fehlerisolation:**
   Um die Ausbreitung eines Fehlers auf angrenzende Gebiete einzudämmen, wird versucht, seine Ausdehnung auf eine bestimmte Komponente zu begrenzen. Fehlereingrenzung kann mit Hilfe von fehlererkennenden Schaltkreisen, Konsistenzprüfern und mehreren Anfrage- und Bestätigungsverhandlungen vor der Ausführung einer Funktion erreicht werden. Diese Techniken finden in Hardware oder Software Einsatz.

6. **Rekonfiguration:**
   Wird ein Fault erkannt und der entsprechende Defekt lokalisiert, kann ein System mit der Möglichkeit zur Rekonfiguration entweder die fehlerhafte Komponente austauschen oder sie zumindest vom Rest des Systems isolieren. Der Austausch wird dabei über redundante Komponenten durchgeführt und das Isolieren über Ausschalten der Komponente. Letzteres degradiert jedoch das System, da es mit einer ausgeschalteten Komponente beispielsweise nicht mehr die Rechenkapazitäten wie vorher besitzt. Im Englischen bezeichnet man dies als *graceful degradation*.

7. **Wiederherstellung:**
   Nach Detektion und (falls nötig) Rekonfiguration ist es notwendig, die Auswirkungen des Fehlers zu eliminieren. Dazu wird das System in einen Zustand zurückgeführt, in dem es sich vor dem Auftreten des Fehlers befand, und die Ausführung von da an erneut gestartet. Diese Form der Wiederherstellung (engl.: *rollback*) greift häufig auf Strategien zurück, die Sicherheitskopien, das Checkpointing oder Journaling nutzen. Bei diesem Prozess spielt die Zeitspanne zwischen Auftreten des Defekts und Fehlererkennung eine wichtige Rolle. Der Wiedereinstiegspunkt muss so weit zurückliegen, dass er die Auswirkungen weiterer unentdeckter Defekte umgeht, die noch vor dem Detektierten auftraten.

8. **Neustart:**
   Die Wiederherstellung kann dann nicht möglich sein, wenn zu viel Information durch

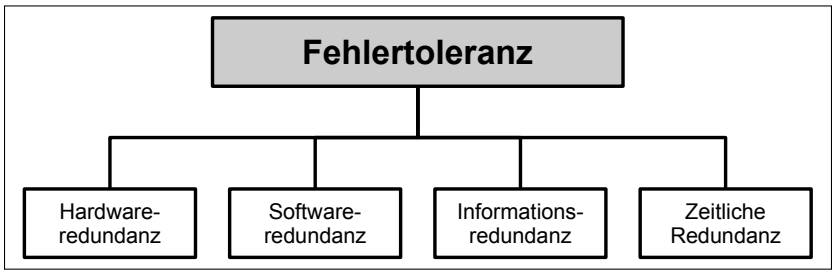


Abbildung 3.4.: Fehlertoleranz - Formen der Redundanz

den Fehler zerstört wurde oder das System einfach nicht dafür ausgelegt wurde (kein Speichern des Zustands). Dann ist ein Neustart notwendig. Dabei unterscheidet man drei verschiedene Arten:

- **„Heißer“ Neustart:** Die Fortführung aller Operationen zur Zeit der Fehlererkennung. Dies ist nur dann möglich, wenn kein Schaden verursacht wurde.
- **„Warmer“ Neustart:** Die Neuaufnahme einiger Prozesse ohne Informationsverlust.
- **„Kalter“ Neustart:** Das komplette System wird neu aufgesetzt. Keine Prozesse überleben.

Auf die letzten beiden Punkte Reparatur und Reintegration soll, da sie für die vorliegende Arbeit nicht weiter von Relevanz sind und aus Platzgründen, verzichtet werden.

3.4. Fehlertoleranz

Der vorhergehende Abschnitt hat die möglichen Maßnahmen beschrieben, die auf das Auftreten eines Fehlers folgen können. Diese sind, je nach Art des Systems, Typ des Fehlers und Ziel der Fehlerantwort verschieden kombinierbar. Die Fehlertoleranz fasst nun diejenigen Kombinationen zusammen, mit denen ein System die Möglichkeit erhält, weiterhin seinen Dienst in Übereinstimmung mit der Spezifikation zu erfüllen, obwohl Fehler aufgetreten sind oder immernoch auftreten ([Lap95]).

Die folgenden Abschnitte führen nun weiter in dieses Thema ein. Dazu wird zuerst der Redundanzbegriff erläutert, da die Fehlertoleranz laut [KK07] immer auf Redundanz basiert. Daraufhin folgt ein Überblick über die wichtigsten Beispiele fehlertoleranter Systeme, sortiert nach Art der Redundanz. Anschließend wird gezielt auf eine spezielle Untergruppe der Fehlertoleranz eingegangen: die Selbstreparatur. Ihr besonderes Merkmal ist, dass sie im Wesentlichen auf der Rekonfiguration basiert.

3.4.1. Redundanz

Die Fehlertoleranz beinhaltet laut [KK07] immer das Ausnutzen von Redundanz. Diese beschreibt die Eigenschaft, mehr Ressourcen zu besitzen als mindestens zum Abarbeiten

einer Aufgabe nötig wären und kann in vier Formen vorliegen: Hardware, Software, Information oder Zeit (siehe Abb. 3.4) ([KK07, S. 3, f.]):

- **Hardware Redundanz:**
  Hierbei handelt es sich um zusätzliche Hardware, die ins System eingebaut wird, um die Effekte einer fehlerhaften Komponente zu detektieren oder zu überlagern. Zum Beispiel könnten anstatt eines einzelnen Prozessors drei eingesetzt werden, von denen alle die gleiche Funktion ausführen. Mit Hilfe von zwei Prozessoren kann der Ausfall eines einzelnen detektiert werden; mit Hilfe von drei kann die fehlerhafte Ausgabe eines einzelnen mit der mehrheitlich richtigen Ausgabe der anderen beiden überschrieben werden. Dies ist ein Beispiel der *statischen (passiven) Hardware Redundanz*, deren Hauptaufgabe die sofortige Maskierung von Fehlern ist. Eine weitere Form ist die *dynamische (aktive) Hardware Redundanz*, bei der zusätzliche Komponenten - sozusagen Ersatzteile - eine derzeitig aktive Komponente bei Ausfall ersetzen. Auch die Kombination der statischen und dynamischen Hardware Redundanz, die sogenannte *hybride Hardware Redundanz*, ist möglich.
  Laut [JG11, S. 82] kann die aktive Hardware Redundanz wiederum in „kalte“ und „heiße“ unterteilt werden. Bei der kalten sind die zusätzlichen Komponenten von der Spannungsversorgung getrennt. Damit verbrauchen sie keinen Strom, altern nicht, benötigen aber eine gewisse Zeit zum Aktivieren. Bei der heißen aktiven Hardware Redundanz liegen die „Ersatzteile“ ununterbrochen unter Spannung und am Takt. Dadurch tritt mindestens eine Verdopplung der Verlustleistung mit gleichzeitigen Alterungseffekten auf, aber transiente Fehler können oft noch im selben Takt erkannt und (bei Verdreifachung) kompensiert werden.
  Unabhängig davon, welche Form der Hardware Redundanz jedoch eingesetzt wird, der zusätzliche Aufwand an Hardware ist beträchtlich. Deshalb wird diese Form oft nur in kritischen Systemen implementiert, bei denen dieser Aufwand auch gerechtfertigt ist.

- **Software Redundanz:**
  Es ist durchaus begründet zu behaupten, dass *jede* große Software, die bisher geschrieben wurde, Fehler (Bugs) enthält. Die Software Redundanz soll im Wesentlichen mit diesen Fehlern umgehen, was aber sehr teuer werden kann: ein Weg ist die unabhängige Produktion von zwei oder mehr Versionen einer Software (bestenfalls auch von unabhängigen Programmierern). Damit wird es sehr unwahrscheinlich, dass beide Versionen mit den identischen Eingaben denselben Fehler produzieren. Die zusätzlichen Versionen könnten auf einfacheren und weniger genauen Algorithmen basieren und nur dazu eingesetzt werden, akzeptable Resultate zu liefen, falls die Hauptversion ausfällt. Genau wie bei der Hardware Redundanz können die Versionen entweder nebenläufig (auf zusätzlicher Hardware) oder sequentiell (mit zeitlicher Redundanz) ausgeführt werden, falls ein Fehler erkannt wird.

- **Informationsredundanz:**
  Die bekannteste Form der Informationsredundanz sind Fehlererkennungs- und -

korrekturcodes. Hierbei werden zusätzliche Bits (sogenannte Prüfbits) in ein originales Datenwort integriert, um Fehler in diesem Wort erkennen oder korrigieren zu können. Die Fehlererkennungs- und -korrekturcodes finden heutzutage breite Anwendung in den verschiedensten Arten von Speichern, um einen Schutz gegenüber nicht allzu schwerwiegenden Fehlern zu bieten. Codes benötigen aber (wie jede andere Form der Informationsredundanz) zusätzliche Hardware, um die redundanten Daten (Prüfbits) zu verarbeiten.
Fehlererkennungs- und -korrekturcodes werden weiterhin eingesetzt, um Daten zu schützen, die über nicht rauschfreie Kanäle übertragen werden. Solche Kanäle sind zum Beispiel die Kommunikationsverbindung zwischen weit entfernten Rechnern (das Internet) oder Prozessoren, die ein lokales Netzwerk bilden.

- **Zeitliche Redundanz:**
  Prozessoren können zeitliche Redundanz nutzen, indem sie dasselbe Programm auf derselben Hardware erneut ausführen. Zeitliche Redundanz wird im Wesentlichen gegen kurzzeitige transiente Fehler eingesetzt, die vorwiegend durch Partikelstrahlung oder durch elektromagnetische Kopplungseffekte entstehen. Falls Fehler solcher Natur sind, ist es sehr unwahrscheinlich, dass bei zwei getrennten Ausführungen derselbe Fehler mehrfach auftritt. Im Gegensatz zu anderen Formen der Redundanz verursacht die zeitliche nur geringe Kosten an Hardware und Software, kann aber die Rechenleistung eines Systems deutlich senken.

3.4.2. Fehlertolerante Systeme

Um nun einen Überblick über die verschiedenen Methoden der Fehlertoleranz bieten zu können, sollen die in [KK07] vorgestellten Beispiele herangezogen werden. Diese sind anhand der Form der vorliegenden Redundanz (siehe Abb. 3.4) sortiert.

Fehlertoleranz mit Hardwareredundanz Laut [KK07, S. 11] ist fehlertolerante Hardware das am meisten untersuchte Teilgebiet in diesem Bereich. Die Techniken, die hier entwickelt wurden, sind dementsprechend vielfältig und fanden bereits in der Praxis in kritischen Anwendungen Einsatz, die von Telefonverbindungen bis hin zu Raumfahrtmissionen reichen. Was aber ihre stärkere Verbreitung bisher gebremst hat, waren die erhöhten Kosten, die für die zusätzliche Hardware nötig sind, um Fehlertoleranz zu erreichen. Da Hardware jedoch immer preiswerter wird, ist dies nicht länger ein Hindernis, was die Bedeutung fehlertoleranter Hardware in Zukunft wohl stark ansteigen lässt. Dennoch gibt es auch andere Faktoren, allen voran die Leistungsaufnahme, die den massiven Einsatz von Redundanz in vielen Anwendungen begrenzen.

[KK07] führen nun verschiedene Beispiele für Hardware Fehlertoleranz an:

- **M aus N Systeme:**
  Diese Architektur - welche in der englischsprachigen Literatur oft auch als als *k out of n:G system* ([KZ02, S.231]) bezeichnet wird - kann wie folgt beschrieben werden: Grundlage bildet ein System mit N identischen Komponenten. M davon

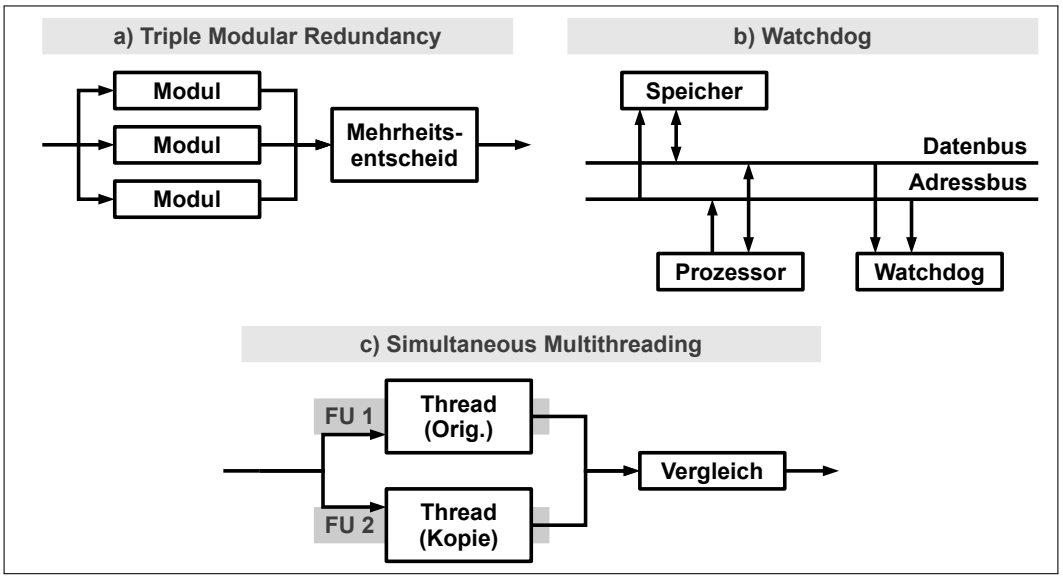


Abbildung 3.5.: Beispiele Hardware Fehlertoleranz

sind mindestens notwendig, um den Zweck des Systems zu erfüllen. Die übrigen $R = N - M$ Komponenten sind redundant und können somit zur Fehlererkennung, -maskierung und Rekonfiguration eingesetzt werden. Das *G* in der englischsprachigen Bezeichnung steht dabei für *good*, also dass mindestens *k* - hier *M* - der *N* Systeme zur Erfüllung der Aufgabe des Systems notwendig sind. Umgedreht existieren auch *k out of n:F system*, bei dem maximal *k* nicht benötigt werden.
Das wahrscheinlich am häufigsten vorkommende Beispiel für M aus N Systeme ist die sogenannte Dreifachauslegung oder Triple Modular Redundancy (TMR) (Abb. 3.5 a)). Hierbei wird ein Modul, beispielsweise eine ALU, dreifach ausgelegt und jedes dieser einzelnen Module mit identischen Eingangssignalen verbunden. Die Ausgangssignale werden über eine sogenannte „Voter“ -Logik verknüpft, die über die Ergebnisse einen Mehrheitsentscheid durchführt. Unterscheidet sich nun eines dieser Ergebnisse von den anderen beiden, kann der Voter dennoch das richtige Ergebnis weitergeben und somit den Fehler maskieren. Da es sich hier um ein zwei-aus-drei-System handelt, müssen aber mindestens zwei Ergebnisse richtig sein, ansonsten funktioniert dieser Ansatz nicht mehr.

- **Watchdogs:**
  Ein sogenannter „Watchdog“ (Abb. 3.5 b)) ist ein Coprozessor, der Fehler auf der Systemebene erkennt, indem er die Kommunikation auf den Bussen zwischen Prozessor und Speicher überwacht. Diese Überwachung zielt oft auf den Kontrollfluss ab und prüft, ob der Prozessor die richtigen Codeblöcke zum richtigen Zeitpunkt und in der korrekten Reihenfolge ausführt. Dazu können beispielsweise gewisse Signaturen eingesetzt werden, die der Watchdog bereits kennt, der Prozessor aber

durch Abarbeitung des Programms erst (in der richtigen Reihenfolge) generieren muss. Entdeckt der Watchdog eine Diskrepanz, setzt er ein Fehlersignal.

- **Simultaneous Multi Threading:**
  Heutige Prozessoren sind nicht nur in der Lage, Threads auf derselben Funktionseinheit zu überlappen, um beispielsweise Wartezeiten auf Daten zu überbrücken, sie können diese sogar echt parallel auf mehreren Funktionseinheiten ausführen. Diese Eigenschaft nennt sich „Simultaneous Multithreading“ (Abb. 3.5 c)) und kann die Laufzeit von Programmen stark verkürzen. Um nun in diesem Fall die Eigenschaft der Fehlertoleranz zu erreichen, kann beispielsweise ein Thread kopiert und auf zwei Funktionseinheiten parallel ausgeführt werden. Ein anschließender Vergleich zeigt, ob die Ergebnisse richtig sind oder gegebenenfalls neu gerechnet werden muss.

Die Fehlertoleranz in Hardware und speziell M aus N Systeme werden später auch bei der Selbstreparatur in Abschnitt 3.4.3 eine große Rolle spielen.

Informationsredundanz Es gibt viele Möglichkeiten, wobei sich Fehler auf Daten auswirken können, beispielsweise bei der Übertragung der Daten von einer Einheit zu einer anderen oder von einem System zu einem anderen oder sogar, während die Daten nur im Speicher liegen. Um solche Fehler zu tolerieren, fügt man Redundanz zu den Daten. Diese kann mehrere Formen annehmen und wird in [KK07] anhand von drei Beispielen erläutert: Codes, RAID Systeme und das Replizieren.

- **Codes:**
  Kurz zusammengefasst wird bei Codes ein d-Bit Datenwort in ein c-Bit Codewort codiert ($c, d \in \mathbb{N}$), welches aus mehr Bit besteht, als das originale Wort, also $c > d$. Durch diese Codierung wird Redundanz hinzugefügt, da mehr Bit eingesetzt werden, als absolut notwendig. Als Konsequenz dieser Informationsredundanz ergibt sich, dass nicht alle 2^c binären Kombinationen der c Bit valide Codewörter sind. Wird nun bei dem Versuch, aus dem c-Bit Codewort wieder das d-Bit Datenwort zu extrahieren, ein nicht-valides Codewort erkannt, deutet das darauf hin, dass bei der Übertragung ein Fehler aufgetreten ist. Mit manchen Codierschemata ist es sogar möglich, einige Fehler nicht nur zu erkennen, sondern zu korrigieren.

- **RAID Systeme:**
  Das Akronym RAID steht für „Redundant Array of Independent (Inexpencive) Disks“ und stellt eine Form der Informationsredundanz dar, die auf einer höheren Ebene als den Datenworten stattfindet. RAID Systeme haben die Aufgabe, Daten von einer Festplatte nach gewissen Schemata über mehrere zu kopieren, codieren und umzuverteilen, so dass die Daten gegen die Auswirkungen von Fehlern gesichert sind und teilweise sogar schneller gelesen werden können.

Fehlertoleranz in Software Laut [KK07] kann beim Schreiben von Software von zwei Fehlerquellen ausgegangen werden: den essentiellen und den unbeabsichtigten. Essentielle

entstehen durch die inhärente Herausforderung, eine komplexe Anwendung und ihre Operationsbedingungen zu verstehen und eine Struktur zu erstellen, die aus einer extrem großen Anzahl von Zuständen besteht, bei denen die Zustandsübergänge sehr komplexen Regeln folgen. Weiterhin unterliegt Software ständigen Änderungen, wenn neue Eigenschaften hinzugefügt oder die Anforderungen der Anwendung geändert werden. Zusätzlich dazu ändern sich auch Hardware und Betriebssysteme mit der Zeit, was weitere Anpassungen der Software mit sich bringt.

Unbeabsichtigte Fehlerquellen beim Schreiben von Software entstehen dadurch, dass man selbst bei einfachen Aufgaben algorithmische und logische Fehler machen kann. Den detaillierten Programmentwurf in korrekt funktionierenden Programmcode umzusetzen mag nicht so umfassendes Können voraussetzen, wie den Entwurf überhaupt erst zu erstellen. Es ist aber auch fehleranfällig. Zusammenfassend wird also zwischen Fehlern im Entwurf und Fehlern bei der Umsetzung unterschieden.

In [KK07] werden nun vier Möglichkeiten vorgestellt, wie die durch diese Quellen entstandenen Fehler toleriert werden können: in Einzelversionen, bei der Programmierung von N Versionen, über Wiederherstellung und mit der Behandlung von Ausnahmen.

- **Fehlertoleranz bei Einzelversionen:**
  Diese Möglichkeit beschäftigt sich mit der Fehlertoleranz bei Einzelversionen von Software. Es werden also keine Kopien angelegt und dann verglichen, sondern zusätzlicher Code geschrieben oder die Vielfalt der Daten verändert. Ein Beispiel hierfür wäre das Schreiben von „Wrapper-“ oder Mantelklassen, die die Ein- und Ausgaben von Programmen steuern und auf Fehler kontrollieren.

- **N-Versions-Programmierung (Software-Diversity):**
  Bei der N-Versions-Programmierung werden letztendlich N unterschiedliche Versionen ein- und desselben Programms erstellt, gleichzeitig ausgeführt und die Ergebnisse miteinander verglichen. Der Einsatz unterschiedlicher Versionen soll dabei die Möglichkeit minimieren, dass überall die gleichen Programmierfehler gemacht werden. Das richtige Ergebnis wird nach der Berechnung per Mehrheitsentscheid ermittelt, womit dieser Ansatz stark den M aus N Systemen aus der Hardware Fehlertoleranz ähnelt. Fehler im Entwurf des Programms selbst können jedoch nicht gefunden werden.

- **Wiederherstellungsansatz:**
  Ähnlich wie bei der N-Versions Programmierung existieren beim Wiederherstellungsansatz mehrere Versionen eines Programms, jedoch wird hier zu Anfang nur eine ausgeführt. Ein Akzeptanztest entscheidet anschließend darüber, ob die Ergebnisse annehmbar sind oder nicht. In letzterem Fall wird der ursprüngliche Zustand des Systems wieder hergestellt und eine zweite Version des Programms gestartet. Sind die Ergebnisse wieder nicht annehmbar, wird eine dritte Version probiert, etc.. Die Schwierigkeit liegt hier darin, einen passenden Akzeptanztest zu finden und Fehler in ihm auszuschließen.

- **Ausnahmebehandlung:**
  Ausnahmen werden dann ausgelöst, wenn bei der Ausführung eines Programms etwas vorgefallen ist, das Aufmerksamkeit erregt. Dies kann zum Beispiel geschehen, wenn irgendwelche Aussagen durch Soft- oder Hardwarefehler verletzt werden. Das Auslösen einer Ausnahme bewirkt dann das Aufrufen einer Ausnahmebehandlung, also einer speziellen Routine, die entsprechende Maßnahmen ergreift. Ein Beispiel hierfür wäre ein Überlauf bei der Berechnung von $c = a \cdot b$. In diesem Fall wäre das Ergebnis nicht korrekt, da der Ergebnisvektor die gesamte Zahl nicht mehr fassen kann. Eine Ausnahme kann nun dafür sorgen, dass das System entsprechend reagiert.

Zeitliche Redundanz Die erneute Ausführung eines Programms wurde bereits im letzten Abschnitt als Beispiel für zeitliche Redundanz aufgeführt. Doch gibt es selbst auf den heutigen Rechnern Probleme, die sehr viel Zeit zur Bearbeitung benötigen. Dann ist eine erneute Ausführung nicht nur unpraktisch, sondern kann regelrecht teuer werden. Aus diesem Grund wird die folgende Form der zeitlichen Redundanz eingesetzt:

- **Checkpointing:**
  Beim Checkpointing werden in regelmäßigen Abständen „Schnappschüsse“ von z.B. Prozessen gemacht und in den Speicher ausgelagert. Diese Momentaufnahmen enthalten den kompletten Zustand des Prozesses zu einem gewissen Zeitpunkt, also alle seine Variablen etc.. Wird nun beispielsweise ein Fehler festgestellt, kann der Prozess in einen vorhergehenden, korrekten Zustand zurückversetzt und die entsprechende Berechnung erneut durchgeführt werden, ohne das Programm noch einmal komplett durchlaufen zu müssen.

3.4.3. Selbstreparatur (BISR)

Die Selbstreparatur, als Teilgebiet der aktiven Hardwareredundanz, ist in der vorliegenden Schrift von besonderer Bedeutung. Aus diesem Grund wird sie im Folgenden gesondert betrachtet und ihre wichtigsten Einsatzgebiete im Einzelnen präsentiert.

Defekte in integrierten Schaltungen tatsächlich zu reparieren, gestaltet sich eher schwierig, wenn nicht sogar unmöglich. Es existieren zwar Ansätze für spezielle Formen, beispielsweise gezieltes Be- und Entladen bidirektionaler Busleitungen zur Heilung der Effekte von Elektromigration ([AVU+08]), aber ein einmal unterbrochener Leiter kann nicht ohne weiteres wieder zugelötet werden. Deshalb wird die Selbstreparatur zwischen dem Maskieren der Fehler in anderen Arten der Fehlertoleranz und der eben angedeuteten Heilung einzelner Fehlereffekte eingeordnet. Ziel ist es, durch Rekonfiguration der Hardware und / oder Software des Systems, die fehlerhafte Komponente zu isolieren und damit den Fehler am Auswirken zu hindern.

Der Name Selbstreparatur leitet sich davon ab, dass die dazu notwendigen Funktionen und Komponenten bereits während der Fertigung auf dem Chip untergebracht werden. Somit ist es dem System möglich, sich im Feld und damit während seiner Einsatzzeit weitestgehend selbstständig wiederherzustellen.

Die Isolierung der fehlerhaften Komponente kann, wie bereits angedeutet, per An-

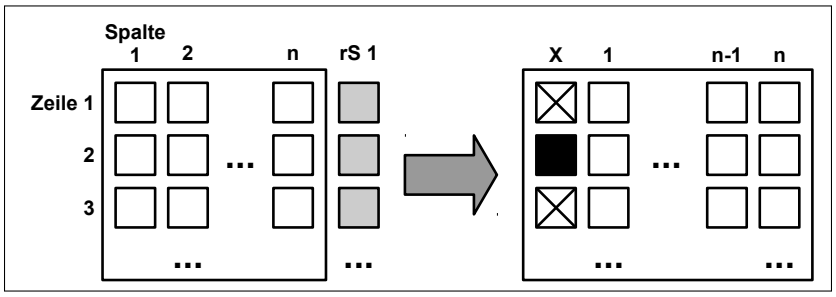


Abbildung 3.6.: Selbstreparatur in eingebetteten Speichern

passung der Software oder Hardware geschehen. Die Möglichkeiten innerhalb dieser beiden Arten unterscheiden sich aber weniger anhand ihrer Methodik, als an den zugrunde liegenden Systemen. Deshalb werden in diesem Abschnitt verschiedene Methoden der Selbstreparatur jeweils in Speichern, Field Programmable Gate Arrays (FPGAs), Verbindungsstrukturen und Prozessorkomponenten, -kernen oder ganzen Mehrkernsystemen vorgestellt.

- **Selbstreparatur in Speichern:**
  Besonders die Selbstreparatur in eingebetteten Speichern gehört zu den Feldern, die seit „einigen Jahren bekannt sind und recht gut beherrscht werden“ ([KHV06, S. 125]). Dies liegt vor allem daran, dass Speicher mittlerweile große Teile der Chipfläche einnehmen und damit die Ausbeute bei der Herstellung direkt betreffen. Weiterhin bietet sich ihr hochregulärer Aufbau speziell für hardwarebasierte Methoden der Selbstreparatur an. Ein stark verbreiteter Ansatz ist dabei die Reparatur über redundante Spalten und / oder Zeilen (siehe Abb. 3.6). Die linke Bildhälfte zeigt einen Speicher mit n „normalen“ und einer redundanten Spalte. Fällt nun beispielsweise die Zelle in Spalte 1, Zeile 2 aus (rechte Hälfte), dann kann die gesamte Spalte mit der redundanten ersetzt werden. Nach dem Verschieben der Adressen um 1 nach rechts $(n+1)$ ist der Fehler so isoliert, dass er nach außen hin nicht mehr sichtbar ist. Dieser Ansatz bildet nun die Grundlage weiträumiger Forschungsarbeit. In [LYH+03] wurde sich beispielsweise intensiv mit der Methodik zur Reparatur auseinandergesetzt, wogegen in [CPL+02] untersucht wurde, wie groß die optimale Anzahl der Zeilen und Spalten für gewisse Szenarien sein sollte.
  Auf Seiten der Software wurde zum Beispiel in [Sch11b] vorgeschlagen, einige wenige Register der Registerbank für die Selbstreparatur zu reservieren. Fällt nun ein normales Register aus, kann per Umbenennung der Register im Programmcode vom fehlerhaften Original zu einem der redundanten, der Fehler isoliert werden.

- **Selbstreparatur in FPGAs:**
  Die einfachste Methode zur Selbstreparatur bei FPGAs ist, angelehnt an die bei Speichern, die Abschaltung einer ganzen Zeile oder Spalte im FPGA Feld und deren Ersetzung durch Redundanz ([DP94]). Dieser Ansatz erweist sich aber als relativ hardwareaufwändig, da für die Isolation einer fehlerhaften Zelle gleich eine gesamte

Zeile oder Spalte geopfert werden muss. Günstiger ist der in [KHV06] beschriebene Ansatz, die Redundanz „schachbrettartig“ in die normalen Zellen einzufügen, so dass jede Zelle von vier redundanten umgeben ist. Hier muss im Fehlerfall nur eine Zelle ersetzt werden, die Rechenzeit für die Bestimmung der Ersatzzelle steigt aber enorm an, falls ein direkter Nachbar schon vergeben ist und ein neuer berechnet werden muss. Für die Rekonfiguration selbst wurde in [MHS+04] vorgeschlagen, zwei FPGAs miteinander zu koppeln. Stellt nun einer der FPGAs durch einen Selbsttest fest, dass er einen Fehler hat, teilt er es dem anderen mit. Dieser unterbricht daraufhin seine normale Arbeit, rekonfiguriert seinen Nachbarn nach bestimmten Vorgaben und nimmt anschließend seine Arbeit wieder auf.

- **Selbstreparatur in Verbindungsstrukturen:**
  Die Selbstreparatur in Verbindungsstrukturen wurde noch nicht so intensiv erforscht wie beispielsweise bei Speichern. Hardwaremäßig basieren viele Ansätze auf dem Hinzufügen redundanter Leitungen, die im Bedarfsfall aktiviert werden können. In [Sch11a] wurde dies in Kombination mit Codes untersucht, um gleichzeitig auch die Fehlertoleranz zu verbessern. In [GISP06] steht die optimale Anzahl redundanter Leiter und Kreuzverbindungen im Mittelpunkt, um die Ausbeute nach der Herstellung zu erhöhen.
  Auf Seite der Software wird in [LLP07] die Anpassung der Übertragung als eine Art Multiplexverfahren beschrieben, die aber eher der Fehlertoleranz als der Selbstreparatur ähnelt: Im Bedarfsfall wird in die sogenannte „split transmission“ umgeschaltet. Dies bedeutet, dass die zu übertragende Nachricht halbiert und beide Hälften verdoppelt übertragen werden. Somit ist zumindest eine Fehlererkennung und, da die Hälften per Hamming Code verschlüsselt sind, auch eine Fehlerkorrektur möglich.

- **Selbstreparatur in Prozessorbausteinen:**
  Das Feld der Selbstreparatur in Prozessorkomponenten auf Hardwarebasis gestaltet sich sehr vielschichtig. Allein die Frage, auf welcher Abstraktionsebene die Reparatur stattfinden soll, unterscheidet viele Ansätze voneinander: [KSV09] beschreiben hier die Gruppierung von einzelnen Gattern oder Komponenten wie ALUs, die mit redundanten Einheiten versehen und im Bedarfsfall ersetzt werden können. In [GFA+08] wird ein Multiprozessorsystem auf Basis von OpenRISC 1600 Kernen entworfen, bei dem die Pipelinestufen der einzelnen Kerne untereinander frei austauschbar sind. Auf höchster Ebene modellieren [ARJS07] ein System mit der Möglichkeit, ganze Prozessorkerne per Auswechselung zu reparieren.
  Auf Softwareseite wurde beispielsweise ein interessanter Ansatz in [SM10] beschrieben. Grundlage ist hier ein VLIW-Prozessor. Erleidet eine Funktionseinheit einen Fehler, so wird die Software so rekonfiguriert, dass dieser Slot nur noch „leere“ Operationen zugeteilt bekommt und damit nicht mehr genutzt wird. Operationen, die eigentlich auf diesem Slot verarbeitet werden sollten, werden (unter zusätzlichem Zeitaufwand) auf die anderen Slots verteilt.

Kapitel 4

Ansatz - Verbindung von Selbstreparatur und Activity Migration

Kapitel 3 hat gezeigt, dass mehrere Möglichkeiten existieren, um den durch die Technologieskalierung beschleunigten Ausfall integrierter Schaltkreise zu verzögern. Zum einen durch Verlangsamung des Entstehens von Defekten, indem die Temperatur des Systems mit Hilfe des Thermal Management kontrolliert wird. Zum anderen durch Maskierung oder Isolation der Auswirkungen von dennoch unvermeidlich auftretenden Fehlern durch die Fehlertoleranz, damit das System weiterhin funktionsfähig bleibt. Da die Skalierung der Technologie aber weiter voran schreitet, werden sich die Fehlerraten in absehbarer Zukunft eher verstärken als verringern. Dies zu Kompensieren motiviert die Untersuchung neuer oder die gewinnbringende Verbindung bereits etablierter Methoden.

Eine dieser Verbindungen, nämlich das Thermal Management in zur Selbstreparatur eingesetzten M aus N Systemen, wurde in [KV10] vorgeschlagen. Sie wird nun eingehend betrachtet.

4.1. Selbstreparatur durch M aus N Systeme

Im weiteren Verlauf dieser Arbeit wird sich die Selbstreparatur in M aus N Systemen gezielt auf Prozessorbausteine (siehe Abschnitt 3.4.3) und damit auf folgendes, bereits in [KV10] vorgestelltes, System beziehen:

Insgesamt existiert eine Anzahl von N baugleichen und fehlerfreien Bausteinen, bei denen es sich beispielsweise um arithmetisch logische Einheiten (ALUs) oder Gruppen von Blöcken wie Prozessorpipelines handeln kann. M davon müssen mindestens funktionieren, um die Aufgabe des Systems zu erfüllen. Es handelt sich also nach [KZ02, S.231] um ein *M aus N:G* System, da mindestens M Blöcke „G wie gut“ sein müssen. $R = N - M$ der N Blöcke sind redundant. Dabei kann es sich wiederum, je nach Art der Blöcke und Anforderungen an das System, um heiße oder kalte aktive Hardware Redundanz handeln.

Fällt nun einer der M originalen Blöcke durch einen permanenten Fehler aus, wird dieser durch einen bisher unbenutzten und daher sicher fehlerfreien, redundanten Baublock ersetzt und das System kann seinen Betrieb weiter ausführen - jetzt aber als *M aus (N-1)* System. Dies kann solange wiederholt werden, bis R originale Blöcke durch alle R redundanten Blöcke ersetzt wurden und somit $M = N$ und $R = 0$ ist. Erleidet eine weitere Komponente einen Defekt, kann nun kein Austausch mehr stattfinden und das komplette System ist unbrauchbar.

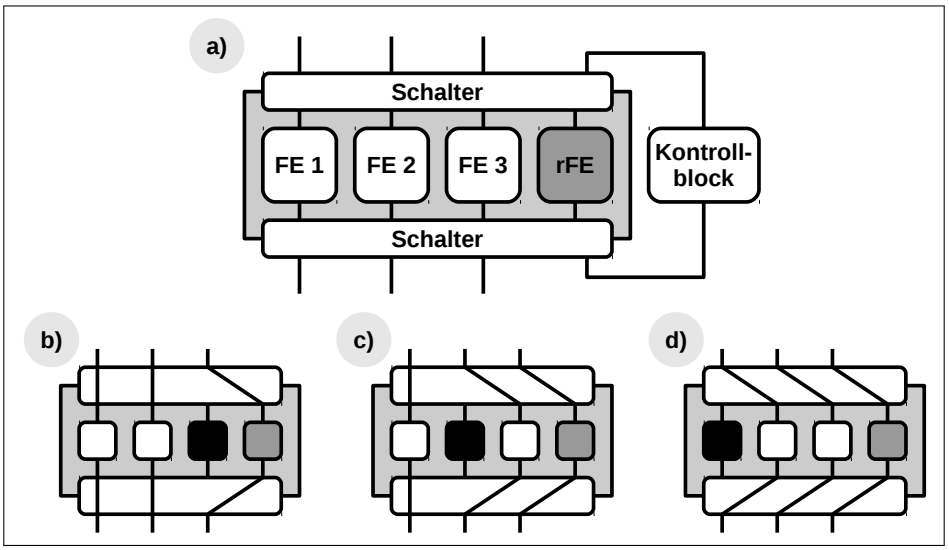


Abbildung 4.1.: M aus N System mit Ersetzung

Eine Beispielimplementierung für ein M aus N System ist in Abbildung 4.1 a) dargestellt. Zu sehen sind drei originale Funktionseinheiten (FE) 1-3, die durch eine vierte, redundante Funktionseinheit (rFE) unterstützt werden; es handelt sich also um ein 3 aus 4 System. Für den Austauschvorgang selbst werden weiterhin der Kontrollblock und die beiden Schaltersysteme vor und hinter den FE benötigt.

Kommt es zum Ausfall einer FE durch einen permanenten Fehler, werden die beiden Schaltersysteme vor und hinter den FE mittels des Kontrollblocks so umgestellt, dass die Ein- und Ausgangssignale den defekten Block „umgehen“, also isolieren und stattdessen die Redundanz nutzen. Abbildung 4.1 b) stellt dabei den Ausfall und die Ersetzung der (schwarz hinterlegten) FE 3, Abbildung 4.1 c) den der FE 2 und Abbildung 4.1 d) den der FE 1 dar. Den Verlust einer weiteren Komponente könnte das System in diesem Fall nicht kompensieren und wäre damit unbrauchbar.

4.2. Thermal Management in M aus N Systemen

Der soeben beschriebene Ansatz ist aus thermischer Sicht aber nicht optimal. Daraus entstehen zwei entscheidende Nachteile: Einerseits liegen die „originalen“ Blöcke unter ständiger Last, heizen sich also stark auf und altern somit frühzeitig. Die redundanten Blöcke andererseits werden bis zu ihrem Einsatz als „Ersatzteil“ gar nicht benutzt, verbrauchen bis dahin also prinzipiell nur Ressourcen (Platz, Transistoren etc.). Weiterhin sind sie bei dauernd anliegender Versorgungsspannung anfälliger für Effekte wie Negative Bias Temperature Instability (NBTI) oder HCI, was bei ihrer Aktivierung unter anderem Verzögerungsfehler verursachen könnte ([LGS09]).

In diesem Zusammenhang wird in [KV10] beschrieben, dass die redundanten Blöcke nicht nur offline zur Reparatur genutzt werden, sondern auch online Last übernehmen

und damit aktiv zur Senkung der Temperatur in den anderen Blöcken beitragen können. Ein großer Vorteil von dieser Art der M aus N Systeme ist, dass sie sich durch ihren Aufbau sehr gut für das Thermal Management eignen, da sie die in Kapitel 3.2.1 unter „Softwarebasiertes STM“ beschriebenen ersten beiden Punkte bereits erfüllen:

- **Schaffung von Abkühlmöglichkeiten:**
  Die redundanten Funktionseinheiten (rFE) bekommen keine Operationen zugeteilt. Damit sind sie immer kühl und stellen somit die Ausweichmöglichkeiten dar, die in [MLN+06] erst durch das IPC-Tuning geschaffen werden.

- **Senkung der mittleren Temperatur:**
  Wird die Kontrolllogik der Schalter leicht angepasst, kann sie nicht nur zur Selbstreparatur eingesetzt werden, sondern auch zur Activity Migration. Ein einfacher Timer würde beispielsweise reichen, um in regelmäßigen Abständen die Last von einer wechselnden FE auf die rFE umzuleiten. Somit wäre für jede FE eine Abkühlphase bereitgestellt und ihre mittlere Temperatur kann sinken. Die Schalter selbst benötigen hierfür keine extra Anpassung.

Nur die Vermeidung von Hotspots muss gesondert betrachtet werden. Um sie zu ermöglichen und die in Abbildung 4.1 beschriebene Verschaltung beibehalten zu können, wird in [KV10] ein offline Profiling der Anwendungen mit anschließendem statischen Umsortieren vorgeschlagen. Der Ablauf ist dabei exemplarisch in Abbildung 4.2 dargestellt. Bild 4.2 a) zeigt den Grundzustand, in dem die Anwendungen noch zufällig über die drei FE verteilt sind. Anwendung 1 erzeugt dabei hohe, Anwendung 2 mittlere und Anwendung 3 geringe Last. Die Anordnung der Anwendungen ist dabei aber eher problematisch, da es für die in Abbildung 4.1 b) - d) gezeigten Umschaltmöglichkeiten keine optimale Lösung zur Activity Migration gibt. Zwar könnten die Anwendungen gut auf andere FE und die rFE umgeschaltet werden, die Last wäre jedoch nicht gleichmäßig verteilt, sondern würde sich auf FE 2 konzentrieren. Deshalb müssen die Anwendungen passend umsortiert werden, so wie beispielsweise in Abbildung 4.2 b) in Zustand 1 geschehen. Nun sind die Anwendungen so geordnet, dass sie sich vor (Abb. 4.2 b)) und nach (Abb. 4.2 c)) dem Umschaltvorgang sehr gut ergänzen. Das heißt, dass hohe Last auf eine leerlaufende und mittlere Last auf eine FE mit geringer Last geschaltet wird. Geschieht dies auf regelmäßiger Basis, wird gleichzeitig die mittlere Temperatur gesenkt und nun auch Hotspots vermieden. Gleichzeitig behält das M aus N System seine Reparaturfunktion bei. Erst nachdem eine FE ausgefallen ist, können in diesem einfachen Beispiel weder Reparatur noch Activity Migration durchgeführt werden.

4.3. Wichtige Eigenschaften

Um die Vorstellung dieses Systems zu vervollständigen, müssen noch zwei weitere Punkte betrachtet werden. Erstens die Kosten des M aus N Systems in Bezug auf zusätzliche Hardware. Dies ist einerseits von Interesse, um einen optimalen Punkt zwischen Aufwand und Nutzen zu finden. Andererseits werden diese Zahlen benötigt, um den darüber hinaus

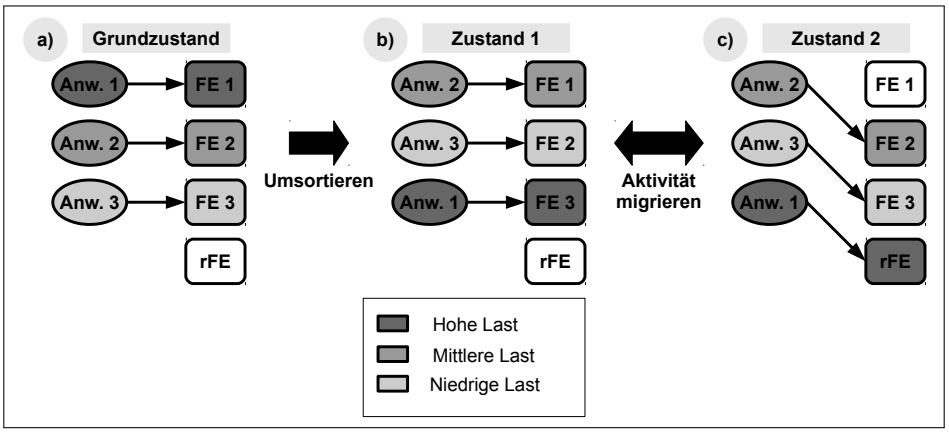


Abbildung 4.2.: Destressing mit Umsortieren der Anwendungen

entstehenden Aufwand für die Implementierung der Acitivity Migration berechnen zu können. Zweitens sollen solche Fehler Beachtung finden, die sich außerhalb der funktionalen Blöcke des M aus N Systems befinden. Dies ist für die vorliegende Arbeit zwar von untergeordnetem Interesse, aber für weiterführende Arbeiten in Richtung Test des Systems durchaus relevant. Am Ende des Abschnittes werden die wichtigsten Eigenschaften noch einmal zusammengefasst. Dies dient der Abgrenzung von anderen Ansätzen der Lebensdaueroptimierungen und gleichzeitig der Begründung, warum genau diese Methode hier untersucht wird.

4.3.1. Der Mehraufwand für M aus N Systeme

Der zusätzliche Aufwand, der für ein M aus N System benötigt wird, hängt einerseits wesentlich von der Anzahl R der redundanten Blöcke ab. Andererseits hat aber auch die Anzahl der Ein- und Ausgänge der jeweils verwendeten Blöcke deutlichen Einfluss. Je höher ihre Anzahl ist, desto mehr Schalter bedarf es für das Abkoppeln bzw. Umleiten der Signale. Setzt man Anzahl und Größe der notwendigen Schalter in Relation zu der eigentlichen Größe des funktionalen Blocks, kann sehr gut abgeleitet werden, welchen Umfang der Mehraufwand besitzt. In [KSV09] werden M aus N Systeme für die unterschiedlichsten Größen von funktionalen Blöcken sehr eingehend studiert. Diese Blöcke, die die Autoren zu einem 3-aus-4-System zusammenfassen, reichen in ihren Rechenbeispielen von einzelnen Gattern, über Halbaddierer, bis hin zu kompletten 8 bit ALUs. Die ermittelten Ergebnisse sind in Tabelle 4.1 dargestellt. Die erste Spalte gibt dabei den Namen des jeweiligen funktionalen Blocks, die zweite die Anzahl der Transistoren für die drei grundlegenden Blöcke (T_{3B}), die dritte die Anzahl der Transistoren für den vierten, redundanten Block (T_{rB}), die vierte die Anzahl der Transistoren für die Schalter (T_S) und die fünfte den tatsächlichen Mehraufwand an, also:

$$Mehraufwand = \frac{T_{3B} + T_{rB} + T_S}{T_{3B}} - 1 \tag{4.1}$$

Tabelle 4.1.: Mehraufwand für die Implementierung von M aus N Systemen

Baugruppe	T_{3B}	T_{rB}	T_S	Mehraufwand
2-NAND	12	4	18	183%
2-AND	18	6	18	133%
2-XOR	18	6	18	133%
Halbaddierer	36	12	24	100%
Volladdierer	90	30	30	66%
8-bit-ALU	4500	1500	168	37%

Die Tabelle 4.1 wurde so angepasst, dass sie den zusätzlichen Aufwand für den Test des Systems nicht enthält. Die Kontrolllogik wurde weiterhin ausgelassen, da sie in jedem Fall gleich groß ist.

Insgesamt lässt sich erkennen, dass die Implementierung eines M aus N Systems vorteilhafter erscheint, je größer die funktionalen Blöcke an sich und im Verhältnis zur Anzahl ihrer Ein- und Ausgänge sind. Dies lässt sich durch den bloßen Hardwareaufwand begründen, der im Vergleich zu den funktionalen Blöcken für die administrativen Gruppen notwendig ist. Weiterhin muss aber auch bedacht werden, dass Schalter und Kontrollblock selbst fehleranfällig sind. Diese Problematik wird im folgenden Abschnitt eingehender betrachtet.

4.3.2. Fehler in den Schaltern / Kontrollblock / Verbindungsstrukturen

Defekte funktionale Blöcke sind innerhalb eines M aus N Systems gut austauschbar. Fehler in den administrativen Baugruppen (Schalter, Kontrollblock, Verbindungsstrukturen) dürfen jedoch nicht missachtet werden, da hier keine Reparaturfunktion vorgesehen ist und die Fehlerwahrscheinlichkeit in diesen Komponenten, je nach Art des Systems, sehr hoch ausfallen kann.

Grundlegend ist dabei mit zwei Arten von Fehlern zu rechnen. Solchen, die sich wie eine defekte Komponente innerhalb des M aus N Systems auswirken, und solchen, die für die Funktionsweise des Gesamtsystems kritisch sind. Bei der ersten Gruppe können die Fehler (z.B. in den Schaltern oder Speicherzellen der Kontrolllogik) als zur FE zugehörig modelliert und durch die Reparaturfunktion behoben werden. Dies ist der günstigste Fall und tritt beispielsweise auf, wenn ein Schalter auf einer Datenleitung an seinem Ausgang immer eine logische ’1’ setzt. Dann ist nur die entsprechende FE zu deaktivieren, eine redundante zu aktivieren und das System kann weiter fehlerfrei arbeiten.

Schwieriger zu handhaben sind dagegen Fehler in den Verbindungsstrukturen oder bei anderen Defekten in den Schaltern, die nicht zu den FE hinzugerechnet werden können. Speziell bei kleineren funktionalen Blöcken mit vielen Ein- und Ausgängen ist die Anzahl der Schalter im Vergleich sehr groß, womit sich dann ein überwiegender Teil der Verdrahtung nicht in den Blöcken, sondern zwischen ihnen und den Schaltern befindet. Dafür besitzt die in Abbildung 4.1 dargestellte Schaltung aber keinerlei Reparaturmaßnahmen. An diesem Punkt setzen die Autoren von [KSV10] an und beschreiben, dass in beiden

Fällen mit sowohl Verdopplung der Verdrahtung als auch redundanten Transistoren in den Schaltern gute Ergebnisse erzielt werden können.

Zusammenfassend ist zu sagen, dass die Größe der administrativen Baugruppen hier eine zentrale Rolle spielt. Je kleiner sie im Vergleich zu den funktionalen Blöcken ausfällt, desto weniger Bedeutung erhält ihre Fehlerwahrscheinlichkeit und wird somit deutlich unkritischer ([KV11]). Außerdem unterliegt speziell die Kontrolllogik oft nur geringem Stress und ist somit weniger anfällig für alterungsbedingte Defekte. Dennoch wird schnell klar, dass diese Probleme nie ganz zu lösen sind, da es immer einen sogenannten „Single Point of Failure“, also einen nicht-reparierbaren Punkt im Prozessor, geben wird.

4.4. Zusammenfassung und Zielsetzung

Die wichtigsten Eigenschaften, die diesen Ansatz von anderen unterscheiden und damit für diese Arbeit so interessant machen, lassen sich wie folgt zusammenfassen:

- **Verbindung von Selbstreparatur und Destressen:**
  Die wichtigste Eigenschaft ist die hier eingesetzte Verbindung von Fehlertoleranz und Thermal Management. Die Kühlung über zusätzliche, redundante Baugruppen, die gleichzeitig zur Fehlerisolierung eingesetzt werden können, verspricht eine sehr hohe Lebensdauer des Systems. Dieser Ansatz wurde so in keiner anderen Methode beschrieben, die bei der im Vorfeld zu dieser Arbeit durchgeführten Literaturrecherche ermittelt wurden und ist somit Alleinstellungsmerkmal.

- **Breites Anwendungsfeld:**
  Da es sich um einen abstrakten Ansatz handelt, sind hier gewonnene Erkenntnisse gut auf andere Systeme mit gleichartigen Komponenten übertragbar. Dies reicht von einzelnen Gattern oder Gattergruppen bis hin zu Mehrkernprozessoren.

- **Skalierbarkeit:**
  Die Größen von M und N sind im Grunde genommen frei wählbar und gewonnene Erkenntnisse für alle Fälle gültig. Theoretisch kann damit auch der Grad der Abkühlung bestimmt werden. Je größer die Differenz zwischen M und N, also R gesetzt wird, desto mehr kühle Komponenten sind vorhanden und umso besser ist die Abkühlung. Jedoch muss hier bedacht werden, dass die redundanten Komponenten zusätzliche Kosten verursachen und die Komplexität der Schalter-Funktion und damit ihre Verzögerung und Fehleranfälligkeit immer größer werden.

- **Keine Systemdegradation:**
  Selbstreparatur und Destressen sind prinzipiell degradationsfrei. Die Rekonfiguration und damit Isolierung einer fehlerhaften Komponente wird ohne Verringerung der Rechenkapazitäten durchgeführt. Höchstens der Verlust einer redundanten Komponente oder die zusätzlichen Verzögerungen in den durch die Schalter länger werdenden Pfaden können als Degradation (Verringerung der Fähigkeiten des Systems) angesehen werden. Aber auch die Activity Migration verlangsamt im grundlegenden Fall

nicht das System, so wie es beispielsweise bei Frequency Scaling oder dem Einfügen von Leerlaufbefehlen der Fall wäre.

- **Geringe Zusatzkosten für die Activity Migration:**
  Die Aktivität in den Komponenten des M aus N Systems zu migrieren, verursacht nur geringen Zusatzaufwand in der Kontrolllogik und in der Software (das oben beschriebene Profiling). Die Schalter, die redundanten Blöcke etc. bleiben unverändert.

Aufgrund der Vielzahl dieser Vorteile ist es Ziel der vorliegenden Dissertationsschrift, den Ansatz zu untersuchen und seine Kosten und Nutzen zu bestimmen. Dazu gehören im wesentlichen drei Schritte:

1. Umsetzung des Ansatzes, um den nötigen Aufwand zu errechnen, der durch Anpassungen in der Soft- und Hardware verursacht wird.

2. Simulation der Umsetzung, um den Einfluss auf die im System vorherrschende Temperatur zu ermitteln.

3. Bestimmung der Auswirkungen des Ansatzes auf Zuverlässigkeit und mittlere Lebensdauer des Systems, die einerseits durch die Fähigkeit zur Selbstreparatur und andererseits durch die Beeinflussung der Temperatur entstehen.

Diese Schritte werden im Folgenden nun detailliert beschrieben, was die Klärung wichtiger Fragen der Umsetzung, Dokumentation getroffener Annahmen und Arbeitsabläufe beinhaltet. Daran schließt sich die Auflistung der ermittelten Ergebnisse und ihre Auswertung in Bezug auf eine mögliche Steigerung von Zuverlässigkeit und mittlerer Lebensdauer an.

Kapitel 5

Umsetzung im VLIW Prozessor

Dieses Kapitel beschreibt nun die Umsetzung des in [KV10] vorgeschlagenen Ansatzes zum Thermal Management in M aus N Systemen. Für die Implementierung wird der VLIW Prozessor als Grundlage gewählt. Eine Begründung dieser Entscheidung und gewisse Eigenschaften, die ihm dafür zugewiesen werden müssen, gibt der nächste Abschnitt. Daraufhin folgt eine detaillierte Beschreibung von fünf verschiedenen Implementierungen des M aus N Systems im Prozessor. Dazu gehört der grundlegende Ablauf der Ansteuerung, der Aufbau der zur Ansteuerung eingesetzten Komponenten und wie die Systeme letztendlich umgesetzt wurden. Der dritte Teil erläutert die Erweiterung der Systeme um das Thermal Management. Speziell wird hierbei zuerst auf notwendige Anpassungen in der Software und anschließend der Steuerungslogik für das M aus N System eingegangen. Am Ende folgt die Darlegung, wie sich die Anforderungen an die Komponenten in den Systemen durch das Thermal Management verändert haben und welche Erweiterungen dies zur Folge hat.

5.1. Der VLIW Prozessor als Grundsystem

In der vorliegenden Dissertationsschrift soll der VLIW Prozessor als Beispielsystem dazu dienen, die Verbindung aus Selbstreparatur und Thermal Management zu untersuchen. Um dies zu ermöglichen, müssen ihm einige Eigenschaften zugewiesen werden, die im Folgenden aufgelistet sind:

- Alle FE besitzen einen identischen Aufbau und als Funktionsumfang die arithmetischen und logischen Grundoperationen, die Multiplikation, Speicher- und Sprungoperationen (siehe Anhang B).
- Die Slots sind identisch aufgebaut.
- Auf ein Forwarding-Netzwerk wird verzichtet, da solch eine Rückkopplung zwischen den Slots bei der Implementierung eines M aus N Systems über die Funktionseinheiten oder Slots hinderlich wäre.
- Die Cluster sind identisch aufgebaut.
- Auf zusätzliche Hardware zur Kommunikation zwischen den Clustern wird verzichtet. Diese läuft ausschließlich über den Datenspeicher

Mit diesen Anpassungen ist der VLIW Prozessor gut für die hiesigen Untersuchungen geeignet und zwar aus zwei Gründen. Einerseits besitzt er nun mehrere Gruppen von gleichartigen Komponenten, die sich gegenseitig nicht beeinflussen (FE, Slots, Cluster). Dies begünstigt die Implementierung von M aus N Systemen, da sich der Austausch der Komponenten sehr einfach gestaltet und in den meisten Fällen keine Abhängigkeiten zwischen ihnen beachtet werden müssen. Weiterhin können diese Gruppen miteinander verglichen werden und so vielfältigere Ergebnisse liefern. Zweitens vereinfachen die gleichartigen Komponenten und der statisch geplante Programmablauf Optimierungen im Programmcode und die Reproduzierbarkeit der Ergebnisse, da beispielsweise keine Sprungvorhersage statt findet und alle Abläufe damit deterministisch sind.

5.2. M aus N Systeme im VLIW Prozessor

Der folgende Abschnitt beschreibt nun fünf Implementierungen von M aus N Systemen im VLIW Prozessor. Da die Rekonfiguration offline stattfindet, kann der Austausch in allen Systemen ohne Beachtung irgendwelcher Nebeneffekte des zeitlichen Verhaltens etc. durchgeführt werden. Das bedeutet, dass der grundlegende Ablauf und die Ansteuerung bei allen Versionen identisch modellierbar ist. Aus diesem Grund werden die bei allen Versionen eingesetzten Basiskomponenten und deren Funktionsweise im Folgenden zuerst vorgestellt. Daran schließt sich die detaillierte Beschreibung der fünf Implementierungen und die Begründung ihrer Auswahl an.

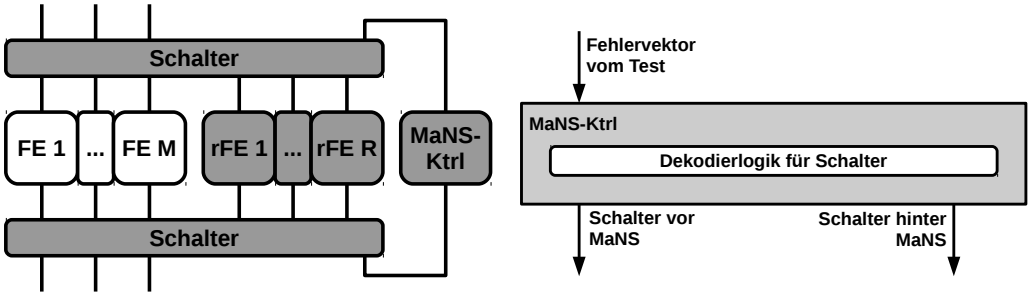
5.2.1. Grundlegender Aufbau und Ablauf

Der grundlegende Aufbau ist noch einmal in Abbildung 5.1(a) dargestellt. Um die parallel arbeitenden, gleichartigen Funktionseinheiten (FE) (in weiß) zu einem M aus N System (MaNS) zusammenzufassen, benötigt es drei weitere Arten von Komponenten (grau hinterlegt): die redundanten Funktionseinheiten (rFE), eine Kontrolllogik (MaNS-Ktrl - 5.1(b)) und die Schalter (5.2).

Für das M aus N System wird offline, zum Beispiel beim start-up, ein diagnostischer Test (siehe [KV11] oder [Ulb10]) durchgeführt. Fehlerhafte Komponenten werden über die Bits eines „Fehlervektors“ abgespeichert. Dieser bildet die Basis für die Rekonfiguration zur Selbstreparatur und liegt als Signal von außen an. Als Alternative könnte beispielsweise auch ein Datenwort an einer festen Adresse im Speicher u.Ä. gewählt werden. Der Vektor besitzt immer die Größe *N* und jedes seiner Bit repräsentiert den Zustand einer FE / rFE. Als Beispiel, wenn alle FE fehlerfrei sind, wäre er in einem 4 aus 8 System „00000000“ und wenn FE 1 und FE 2 einen Fehler besitzen „11000000“.

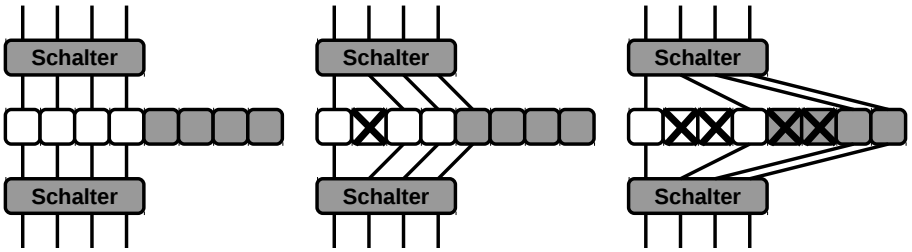
Der Vektor wird anschließend von der Kontrolleinheit (MaNS-Ktrl) gelesen, dekodiert und an die Schalter weitergegeben. Da die Reparatur offline stattfindet, also die Konfiguration während des Betriebs nicht geändert wird und der Vektor als fehlerfreie Kombination der FE angenommen wird, benötigt die Kontrolleinheit keine Zustände und kann so als reine Logik ohne sequentielle Bauteile implementiert werden.

Das dekodierte Signal wird anschließend an die Schalter weitergegeben. Diese haben die Funktion, die Eingangssignale einer FE im Fehlerfall auf einen seiner *R* rechten Nachbarn



(a) Grundlegendes Schema des MaNS

(b) Kontrollblock des MaNS



(c) Ausfall einiger FE im 8 aus 8 System

Abbildung 5.1.: M aus N Systeme - Grundlegende Komponenten

um- und wieder zurückzuleiten. Beispiele hierfür sind schematisch in Abbildung 5.1(c) dargestellt (links - fehlerfreier Zustand, Mitte - Ausfall einer FE, rechts - Ausfall von vier FE).

Eine ausführliche Darstellung der Schalter in einem 4 aus 8 System am Beispiel der Funktionseinheiten (FE) befindet sich in Abbildung 5.2. Um die Signale aller FE auf R rechte Nachbarn umleiten zu können, werden insgesamt $M \cdot R$ Demultiplexer und $(M-1) \cdot R$ ODER Gatter benötigt, die eine für den in die FE eingehenden Bus passende Bitbreite aufweisen müssen. Angesteuert durch MaNS-Ktrl, dienen pro FE jeweils R Demultiplexer dazu, die entsprechende Funktionseinheit „abzuklemmen“ und die Signale nach rechts weiter zu verteilen. Durch die R ODER Verknüpfungen werden diese Signale dann in die gewünschten benachbarten FE eingespeist und dort verarbeitet. Nur bei der letzten FE sind keine ODER Gatter vonnöten, da die rFE redundant sind und somit keine eigenen Kontroll- und Datenworte aus der Pipeline erhalten.

Um das M aus N System für das übrige System transparent zu halten, befinden sich hinter den FE wiederum jeweils R Multiplexer, also insgesamt $M \cdot R$, die die Signale wieder zurück in ihren ursprünglichen Slot leiten. Die Bitbreite der Demultiplexer vor und der Multiplexer hinter den FE ist dabei nicht gleich, da die Funktionseinheiten oft wesentlich mehr Ein- als Ausgänge besitzen.

Das Layout der Schalter ist natürlich austauschbar. So könnte diese Funktion auch beispielsweise mit Tri-State Elementen oder sogenannten Crossbar Schaltern, wie vorgestellt in [GAFM10], implementiert werden. Auch zusätzliches Clock- bzw. Power Gating (siehe

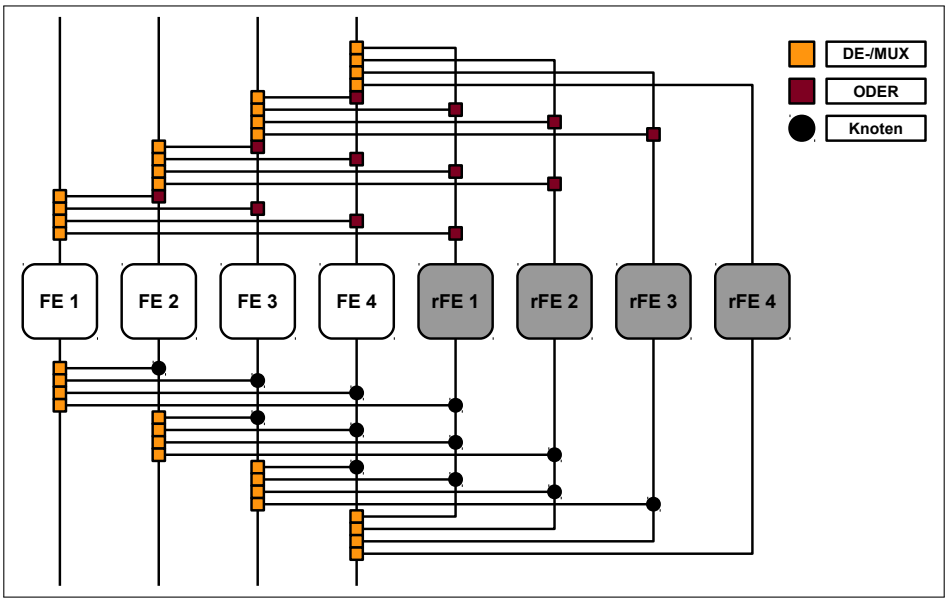


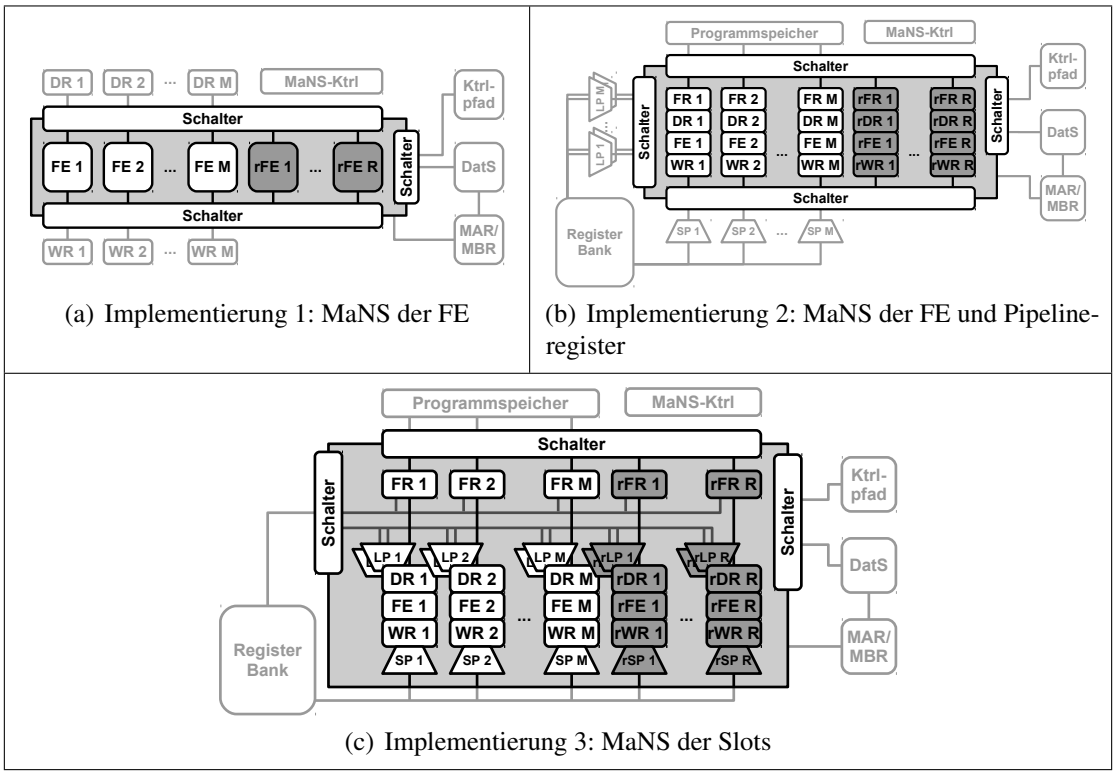
Abbildung 5.2.: Schematischer Aufbau der Schalter

Abschnitt 3.2.2) wäre denkbar. Für die vorliegende Arbeit wurde jedoch eine möglichst einfache Lösung gesucht, die folgende Eigenschaften besitzt:

- Sehr genaue Beschreibung in VHDL, um der Synthese möglichst wenig Spielraum zu lassen und dadurch ermittelte Ergebnisse der Gatterzahl nachvollziehbar und nachrechenbar zu halten.
- Eine Verbindung wie in Crossbars ist nicht notwendig, da der Austausch nur nach dem oben genannten Schema durchgeführt wird und nicht „alle mit allen“ verdrahtet werden müssen.
- Der Austausch soll möglichst in einem Takt durchführbar sein. Power und Clock Gating benötigen hingegen eine gewisse Zeit, um die Transistoren vorzuladen.
- Für einen Test, der in weiterführender Arbeit folgen könnte, ist diese Form der Schalter leicht zu modellieren.

5.2.2. Die fünf Implementierungen

Das M aus N System in fünf unterschiedlichen Versionen zu implementieren, soll zwei Zielen dienen: die Größenordnung des Hardwareaufwands zur Implementierung von M aus N System mit unterschiedlichen Komponenten zu bestimmen und den Einfluss des Verhältnisses von Fläche zu Leistung auf die Temperaturerzeugung (und -senkung) zu untersuchen. Dieser Abschnitt soll die fünf Systeme nun vorstellen und begründen, warum genau diese ausgewählt wurden. Auf die detaillierte Beschreibung der Schalter und der Kontrolllogik für das M aus N System (MaNS-Ktrl) wurde dabei verzichtet, da sie in jedem Fall gleichartig sind.



(a) Implementierung 1: MaNS der FE

(b) Implementierung 2: MaNS der FE und Pipelineregister

(c) Implementierung 3: MaNS der Slots

Abbildung 5.3.: Grafische Übersicht der M aus N Systeme 1-3

1. M aus N System der Funktionseinheiten Hierbei handelt es sich um den Basisfall, dargestellt in Abbildung 5.3(a). Zu erkennen sind in der Mitte des Bildes die *M* in weiß gehaltenen „originalen“ Funktionseinheiten. Diese werden durch *R* redundante rFE (mit grauer Füllung) unterstützt. Die Schalter begrenzen das System und kontrollieren den Wechsel zwischen den *M* äußeren und *N* inneren Signalen. Die Komponenten, deren Signale umgeschaltet werden müssen, sind hellgrau markiert. In diesem Fall handelt es sich um die Decode Register (DR 1...M) und die Register der Write Back Phase (WR).

Die Verbindungen von und zu dem Datenspeicher (über MAR / MBR) und dem Kontrollpfad müssen an dieser Stelle etwas näher betrachtet werden. Da alle FE identisch aufgebaut sind, haben auch alle die Möglichkeit, in MAR / MBR und den Kontrollpfad zu schreiben. Somit waren bereits vor der Implementierung des M aus N Systems in Kontrollpfad und MAR / MBR Strukturen vorhanden, die „Eins aus M“ schreibende Signale auswählten. Hier erschien es sinnvoller, diese Auswahl auf „Eins aus N“ zu erweitern, anstatt weitere Schalter hinzuzufügen. In Richtung der FE, wenn diese also Daten aus dem Datenspeicher oder den Programmzähler aus dem Kontrollpfad lesen, müssen jedoch auf jeden Fall Schalter vorhanden sein. Dadurch wird verhindert, dass sich hier ändernde Werte in eine inaktive FE propagieren und sie trotz Ruhephase zum Schalten bringen. Diese Festlegungen über die Schalter für die Signale von und zu den FE gelten auch für alle weiteren Implementierungen.

Die Implementierung des M aus N Systems über die Funktionseinheiten wurde aus zwei Gründen gewählt. Einerseits, da es sich hier um den „offensichtlichsten“ Fall handelt, der den vorgeschlagenen Ansatz aus [KV10] direkt umsetzt. Andererseits wurden die FE mit den ALUs und Multiplizierern bereits in [MLN+06] als heißeste Komponenten identifiziert. Damit sollte sich die Activity Migration hier am stärksten auswirken.

2. M aus N System der FE und Pipelineregister Mit Hilfe der zweiten Implementierung können nun zusätzlich zu den FE die Pipelineregister ausgetauscht werden (siehe Abb. 5.3(b)). Grau dargestellte Komponenten, wie die Lese- und Schreibports (LP und SP) der Registerbank und die Registerbank selbst, befinden sich weiterhin außerhalb des Systems.

Implementierung 2 wurde umgesetzt, da es der nächste logische Schritt zu sein scheint, statt nur der FE die Pipelineregister mit auszutauschen. Weiterhin ist es interessant zu erfahren, ob diese Register klein genug sind und ausreichend Aktivität entfalten, damit sich die AM positiv auswirken kann.

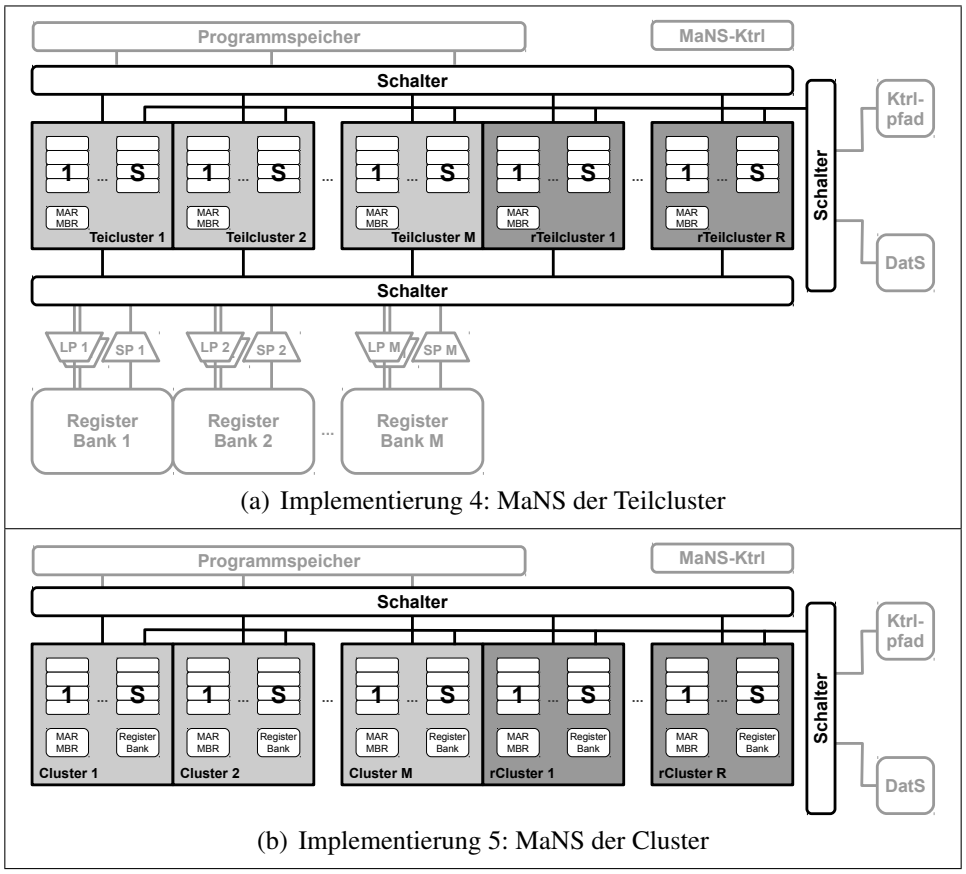
3. M aus N System der Slots Der dritte Fall enthält nun alle Komponenten, die im Ein-Cluster-Modell des VLIW Prozessors mehrfach auftreten, also die gesamten Slots, inklusive der Lese- und Schreibports (Abb. 5.3(c)). Die LPs treten immer doppelt auf, da in einem Takt gleichzeitig zwei Werte aus der Registerbank gelesen werden. Die Schalter sind dabei so integriert, dass sie zwischen Registerbank und Ports liegen, im Prinzip also nur eine weitere Ebene im Multiplexerbaum der Ports darstellen.

Zusätzlich besitzt dieser Fall noch eine weitere interessante Eigenschaft, denn hier kann komplett darauf verzichtet werden, die *N* aus dem M aus N System führenden Signale wieder auf *M* zu reduzieren. Dies liegt daran, dass Kontrollpfad und MAR / MBR, wie bereits angedeutet, selbst „Eins aus N“ auswählen. Auch sind die Schreibports an sich schon Schalter, die keine Daten durchlassen, wenn der Slot deaktiviert ist. Damit muss hier nicht im Nachhinein noch ausgewählt werden.

Diese Variante ist für die Untersuchungen interessant, da sie einerseits alle mehrfach auftretenden Komponenten in einem Cluster enthält. Andererseits müssten die Ports das Verhältnis von Fläche zu Leistung deutlich verschieben, da zu erwarten ist, dass sie trotz enormer Größe nur relativ geringe Aktivität entfalten.

4. M aus N System der Cluster, ohne Ports und Registerbänke In Implementierung 4 können Teile der Cluster ausgetauscht werden. Diese wurden in Abb. 5.4(a) Teilcluster genannt und enthalten die FE, Pipelineregister und MAR / MBR Register eines Clusters, aber nicht deren Ports und Registerbank. Das heißt, fällt in Teilcluster 1 eine FE aus, wird dieser gesamte Teilcuster durch einen anderen ersetzt. Für die Leseports beispielsweise gibt es jedoch keine Redundanz.

Die Schalter sind dabei ähnlich wie in Implementierung 2 aufgebaut, nur dass sie in diesem Fall wesentlich breiter ausfallen, da deutlich mehr Slots ausgetauscht werden müssen. Weiterhin findet nun eine Umschaltung zwischen *N* Ausgängen der MAR / MBR Register und den *M* Eingängen des Datenspeichers (DatS) statt.



(a) Implementierung 4: MaNS der Teilcluster

(b) Implementierung 5: MaNS der Cluster

Abbildung 5.4.: Grafische Übersicht der M aus N Systeme 4-5

Diese Implementierung ist interessant, da sie, ähnlich wie die zweite Version, kleine Komponenten mit hoher Aktivität enthält, diese aber auf höherer Ebene austauscht.

5. M aus N System der Cluster Die fünfte Version enthält alle mehrfach auftretenden Komponenten eines geclusterten VLIW Prozessors und somit auch die Ports und Registerbänke. Fällt also eine Komponente eines Cluster aus, wird der komplette Cluster ersetzt.

Version 5 wird im Wesentlichen durch das Hinzukommen der Registerbänke interessant. Diese sind sehr groß, mit vergleichsweise geringer Aktivität, was deutliche Auswirkungen auf die Ergebnisse der Activity Migration haben sollte. Außerdem müssen später, bei der online AM, während des laufenden Betriebes, die Inhalte der Registerbänke kopiert werden. Hierzu sind spezielle Lösungen notwendig.

Die Schalter stehen nun nur noch zwischen Programmspeicher und Fetch Registern, MAR / MBR Registern und Datenspeicher und FE und Datenspeicher / Kontrollpfad.

5.3. Implementierung der Activity Migration

Beim Hinzufügen der Activity Migration ist nun dafür zu sorgen, dass die Last während des laufenden Betriebs zwischen originalen und redundanten Komponenten migriert werden kann. Dafür sind entsprechende Anpassungen in der Soft- und Hardware notwendig. Der statisch geplante Programmablauf des VLIW Prozessors hat dabei den Vorteil, dass die Software bereits vor der Ausführung vorbereitet werden kann. Eigenschaften, die sie dadurch erhält, können anschließend bei der hardwaremäßigen Erweiterung als Voraussetzung betrachtet und das System dementsprechend ausgelegt werden.

Der folgende Abschnitt beschreibt nun diese Eigenschaften der Software und ihre Umsetzung. Darauf aufbauend werden anschließend der Ablauf der Activity Migration und die notwendigen Anpassungen der Hardware erläutert.

5.3.1. Anpassungen in der Software

Das auszuführende Programm beeinflusst die Hitzeerzeugung und damit den Nutzen der Activity Migration auf zwei Arten:

- **Allgemeine Temperaturerzeugung:**
  Je nachdem, wie stark die Komponenten beansprucht werden, desto mehr oder weniger Hitze erzeugen sie. Aus diesem Grund hängt es stark von der Software ab, ob Methoden zur Activity Migration überhaupt notwendig und wie erfolgreich sie sind. Dies kann über die Auswahl der passenden Programme gut gesteuert werden.

- **Entstehung von Hotspots:**
  Werden die Komponenten unterschiedlich stark belastet, entstehen Hotspots. Dies kann dazu führen, dass einzelne Komponenten trotz Methoden zur Activity Migration deutlich heißer werden als andere und damit wesentlich eher ausfallen. Dies muss durch Anpassungen der Software vermieden werden.

Die folgenden beiden Abschnitte beschreiben nun die Wahl geeigneter Software und deren Anpassung zur Vermeidung von Hotspots.

Die Wahl der passenden Software Prinzipiell kann auf zwei Wegen Software für die späteren Untersuchungen zum Nutzen der Activity Migration gewonnen werden: durch die Auswahl repräsentativer Benchmarkalgorithmen, die auf das Zielsystem kompiliert werden müssen, oder durch Generierung von (Quasi-) Zufallscode. Für die vorliegende Arbeit wird die letztere Methode gewählt und zwar aus folgenden Gründen:

- Zum Zeitpunkt der Niederlegung dieser Schrift steht kein funktionsfähiger Compiler zur Verfügung.

- Diesen zu erstellen oder die Algorithmen per Hand in Maschinencode für alle Zielimplementierungen umzuwandeln, wäre sehr aufwändig und würde den Rahmen dieser Arbeit sprengen.

- Mit generiertem Code kann die Auslastung der Komponenten gezielt bestimmt werden.
- Diese Auslastung kann für alle Zielimplementierungen gleich gehalten werden, was die Vergleichbarkeit erhöht.
- Hotspots können direkt vermieden werden, indem ausbalancierter Code generiert wird (Dass dies auch für „echte Programme“ eine realistische Annahme ist, wird im nächsten Abschnitt begründet).

Die Generierung des Programmcodes findet dabei über ein Perl-Script statt, welches mit Hilfe von Konstanten konfiguriert wird. Diese sind in Quelltext 5.1 dargestellt: die Länge des Programms in Anzahl von *very long instruction words* (CODE_LENGTH), die Anzahl der Bit einer einzelnen Operation im VLIW (WORD_LENGTH), die Anzahl der Funktionseinheiten oder Slots (NR_FUS), die Anzahl der Cluster (NR_CLUSTER) und die Anzahl der Register in den dazugehörigen Registerbänken (REGFILE_SIZE).

Der Aufbau des Programms selbst wird über vier weitere Konstanten festgelegt, die beinhalten, wie hoch der Prozentsatz einer bestimmten Art von Operationen ausfällt. Diese wären leere Operationen (NOPS), arithmetisch / logische Befehle (ARITH), Sprungbefehle (JUMP) und Befehle zur Interaktion mit dem Datenspeicher (LO_ST). Die letzten beiden wurden jedoch nicht implementiert, wodurch der Code nur aus NOP- und arithmetisch / logischen Befehlen besteht.

Bei der Generierung werden zuerst über ldc Befehle (siehe Anhang B) sämtliche Register aller Cluster mit zufälligen Werten gefüllt. Erst danach wählt das Script solange Befehle aus, bis CODE_LENGTH Zeilen erreicht wurden. Dies ist nicht ganz dem Zufall überlassen, da, wie bereits erwähnt, die Häufigkeit einer Gruppe von Operationen bestimmt werden kann. Welche Operation aus dieser Gruppe aber gewählt wird und auf welchen Quell- und Zielregistern sie arbeitet (siehe Anhang B), ist wieder zufällig.

```
1  # Eigenschaften des Modells
2  use constant CODE_LENGTH          => 300;
3  use constant WORD_LENGTH          => 26;
4  use constant NR_FUS               => 2;
5  use constant NR_CLUSTER           => 2;
6  use constant REGFILE_SIZE         => 64;
7
8  # Prozentangaben, wie haeufig jede Befehlsart auftauchen soll
9  # Muss 100 ergeben
10 use constant PERCENT_NOPS         => 0;
11 use constant PERCENT_ARITH        => 100;
12 use constant PERCENT_JUMP         => 0;
13 use constant PERCENT_LO_ST        => 0;
```

Quelltext 5.1: Konstanten zur Codegenerierung

Vermeidung von Hotspots Im vorliegenden Fall ist es nicht notwendig, die Software anzupassen, um Hotspots zu vermeiden. Der generierte Programmcode ist bereits ausbalanciert und so sind weitere Schritte nicht erforderlich. Für echte Algorithmen gilt das jedoch nicht. In diesem Fall könnten auf zwei Ebenen Optimierungen stattfinden:

- **Innerhalb der Cluster:**
  Die Möglichkeit mit der feineren Granularität ist das bereits in Kapitel 3.2.1 beschriebene **Load Balancing** mit dem Umsortieren der Operationen innerhalb eines Clusters. Der Aufwand für die Durchführung wäre eher gering, vor allem, da bei der vorliegenden Version des VLIW Prozessors alle Befehle innerhalb eines Taktes ausgeführt werden und bei den zwei Takte dauernden Speicheropertionen die ausführende FE keine Rolle spielt. Dadurch reduziert sich der in Abschnitt 3.2.1 vorgestellte Algorithmus auf zwei Arbeitsschritte: Den ersten, um den Programmcode einmal durchzulesen und die ideale und die tatsächliche Schaltaktivität der FE zu berechnen. Den zweiten, um nacheinander alle VLIWs zu betrachten und Operationen von FEs mit hoher zu FEs mit niedriger Aktivität zu verschieben.
  Die Auswirkungen des Load Balancing auf die Activity Migration wären dann sehr positiv, da so ein sehr feingranulares Gleichgewicht zwischen den Komponenten entsteht. Dies verhindert Hotspots, sorgt aber auch dafür, dass alle Komponenten bei der AM gleich behandelt werden können, was Arbeits- und Abkühlphasen angeht. Das gilt sogar sogar bei Ausfall von einer bis $R = N - M$ Komponenten. Somit wird eine starke Automatisierung der AM ermöglicht.

- **Zwischen den Clustern:**
  Muss jedoch ein Gleichgewicht zwischen den Clustern hergestellt werden, reicht der in Abschnitt 3.2.1 beschriebene Algorithmus zum Load Balancing nicht mehr aus. Hier könnte es zu Verletzungen von Datenabhängigkeiten kommen, wenn eine Operation ohne weiteres in einen anderen Cluster verschoben wird. In diesem Fall würde sich ein ähnlicher Vorgang, wie das in [KV10] beschriebene Umsortieren der Anwendungen (siehe Abschnitt 4.2), anbieten, indem nicht nur eine, sondern alle Operationen eines Clusters zu einem anderen verschoben werden, bis wieder eine Balance vor und nach dem Umschaltvorgang hergestellt ist. Andere Möglichkeiten wären das zusätzliche Einfügen von Leerlaufbefehlen, bis alle Cluster die gleiche Aktivität entfalten oder die betroffenen Registerinhalte beim Verschieben einer Operation zu einem anderen Cluster „mitzunehmen“, also zu kopieren. Dies würde aber längere Laufzeiten des Programms bedeuten.

Weiterhin existieren auch dynamische Methoden zum Load Balancing, die beispielsweise sehr hardwarenah ([PSV05]) oder unter Zuhilfenahme des Betriebssystems ([CCF+07]) durchgeführt werden.

5.3.2. Erweiterungen der Hardware

Mit der vorangegangenen Implementierung des M aus N Systems sind bereits alle grundlegenden Voraussetzungen für die Activity Migration (AM) erfüllt, nämlich, dass durch

die Schalter die Aktivität von „heißen“ Komponenten zu „kalten“ migriert werden kann. Weiterhin erlaubt der ausbalancierte Code durch die Generierung des Programms (sonst das Load Balancing (LB)), die Migration sehr stark automatisiert durchzuführen. In diesem Abschnitt wird nun beschrieben, wie dies geschieht, wodurch die AM angeregt wird und welche Anpassungen notwendig sind.

Aus Abschnitt 3.2 lässt sich erkennen, dass es vielfältige Möglichkeiten gibt, das Thermal Management und speziell für den vorliegenden Fall, die AM auszulösen. So könnte dies beispielsweise dynamisch über Temperatursensoren oder statisch bei Erreichen eines gewissen Zählerstandes geschehen. Im vorliegenden Fall erscheint aber das Setzen eines zusätzlichen Kontrollbits im Programmcode als die beste Lösung. Umgesetzt wird dies über die Verbreiterung des VLIW um ein zusätzliches Bit (AM-Bit), welches die AM kontrolliert. Die Vorteile lassen sich wie folgt zusammen fassen:

- Es sind weniger Anpassungen im Prozessor notwendig als beispielsweise beim Einsatz von Sensoren.
- Die AM muss nicht in statischen Zeitabständen durchgeführt werden, sondern kann anhand von Profiling Daten bestimmt werden, die die Häufigkeit des Umschaltens in Abhängigkeit der Temperatur festlegen.
- Falls das gleichzeitige Ausführen von AM und beispielsweise mehrere Takte laufenden Operationen zu Problmen führt, kann der Compiler entweder die Operation oder das AM-Bit verschieben. Bei einem von außen anliegenden Signal wäre das nicht möglich.
- Die Kontrolle über die AM zu behalten, kann sich bei der Generierung eines softwarebasierten Tests als günstig erweisen.

Zusammenfassend kann also Folgendes gesagt werden: Die Generierung der Software (sonst das LB) erlaubt es, jede Komponente des M aus N Systems bei der AM gleich zu behandeln, da alle von Anfang an die gleiche Last zugeteilt bekommen. Dies spricht natürlich für eine starke Automatisierung der AM, wobei der Großteil der Steuerung vom Kontrollblock der Schalter übernommen wird. Das AM-Bit im Programm löst diesen Vorgang aus und bestimmt seine Häufigkeit, was von nun an die Bezeichnung AM-Takt erhält. Dementsprechend kann nun weiter in die Tiefe gegangen und der Ablauf in den Komponenten und speziell der Kontrollblock eingehender beschrieben werden.

Grundlegender Ablauf der AM Die folgende Beschreibung des grundlegenden Ablaufs der AM in M aus N Systemen, unter der Annahme von ausbalanciertem Code und am Beispiel von Implementierung Eins, als 4 aus 6 System, wurde bereits in [UKV12b] veröffentlicht und ist in Abbildung 5.5 a) dargestellt. Die Blöcke stehen jeweils für die 4 FE und 2 rFE, wobei eine graue Markierung eine Deaktivierung zur Abkühlung darstellen soll. Jede Zeile gibt dabei einen AM-Takt an, wobei dieser mehrere Prozessortakte umfassen kann, je nachdem, wie häufig das AM-Bit im Programmcode gesetzt ist. Genau $R = 2$ FE / rFE sind ab Anfang deaktiviert. Welche das sind, bestimmt ein entsprechender Initialvektor,

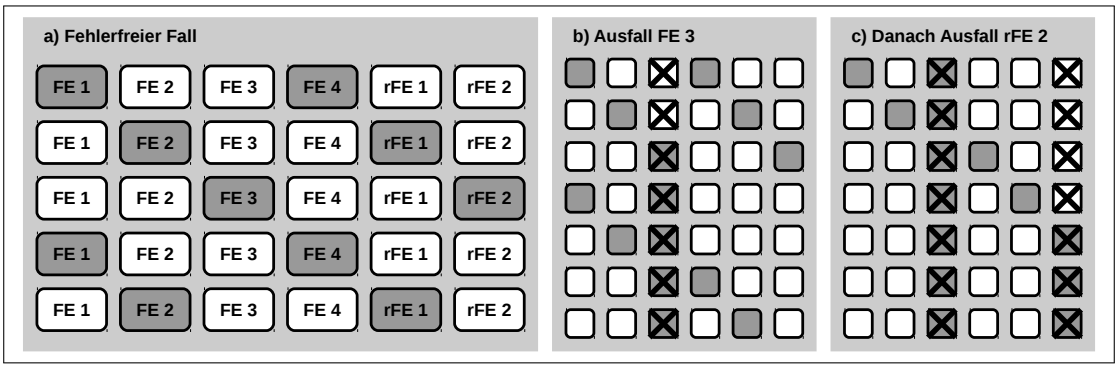


Abbildung 5.5.: Umschalten der FE / rFE im fehlerfreien / fehlerhaften Fall

der beispielsweise an einer festen Adresse im Datenspeicher steht und bei dem die zu deaktivierenden FE / rFE auf '1' gesetzt sind. Die Beschreibung mit Vektoren lehnt sich stark an die FE Pattern aus [MLN+06] an, im vorliegenden Beispiel, in Zeile 1, wäre dieser also {100100} für FE 1 und 4.

Bei gesetztem AM-Bit werden dann die Deaktivierungsphasen jeweils um eine Stelle nach rechts rotiert. Das bedeutet, dass in der zweiten Zeile FE 1 und 4 wieder aktiv sind und sich FE 2 und rFE 1 abkühlen können. Hier zeigt sich auch, dass die Einsen im Initialvektor einen möglichst großen Abstand untereinander haben sollten, da sonst mehrere Abkühlungsphasen direkt aufeinanderfolgen. Der Vektor {110000} wäre beispielsweise eine schlechte Wahl, da die FE zwei Takte lang ruhen und den Rest aktiv sind. Im nächsten durch den Nutzer ausgelösten AM-Takt wandert die Inaktivität in Zeile 3 wiederum um eine Stelle nach rechts und deaktiviert FE 3 und rFE 2. Damit sich ein Kreislauf ergibt, werden bei erneutem Setzen des Kontrollbits wieder FE 1 und 4 inaktiv (Zeile 4).

Interessanter ist jedoch der Fehlerfall, da hier nun die Redundanz für die Selbstreparatur benötigt wird. Angenommen, ein diagnostischer Test stellt fest, dass FE 3 nicht mehr funktioniert (Abb. 5.5 b)). Der Testapparat setzt das entsprechende Bit in einem Fehlervektor, der den gleichen Aufbau wie der Initialvektor besitzt und beispielsweise auch im Datenspeicher abgelegt wird. Sobald die Kontrolllogik für das M aus N System diesen ausliest, sorgt sie dafür, dass die Inaktivitätsphasen die fehlerhafte FE nicht mehr verlassen können, sobald sie diese einmal erreicht haben (Abb. 5.5 b), Zeile 3). Diese FE ist ab dann dauerhaft gesperrt. Erreicht eine weitere Inaktivitätsphase die entsprechende FE, wird diese übersprungen und der nächste rechte Nachbar ohne Sperre für einen AM-Takt deaktiviert (im Beispiel FE 4, Zeile 6). Ansonsten rotiert diese Phase normal weiter.

Fällt nun eine zweite FE aus (Abb. 5.5 c)), wird diese durch die letzte „freie" Inaktivitätsphase deaktiviert. Ab jetzt kann keine AM mehr erfolgen. Der Grund dafür liegt darin, dass nun alle Reserven des M aus N Systems erschöpft sind, keine Selbstreparatur mehr durchgeführt werden kann und nur noch die letzten M FE arbeiten.

Implementierungsdetails der Kontrolllogik Der soeben beschriebene Ablauf ist für alle fünf Implementierungen (mit kleinen Anpassungen) identisch und spiegelt sich im

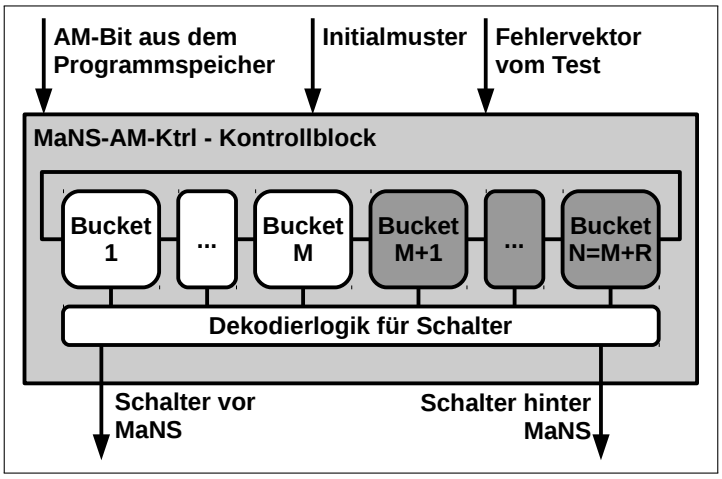


Abbildung 5.6.: Der Kontrollblock für das M aus N System

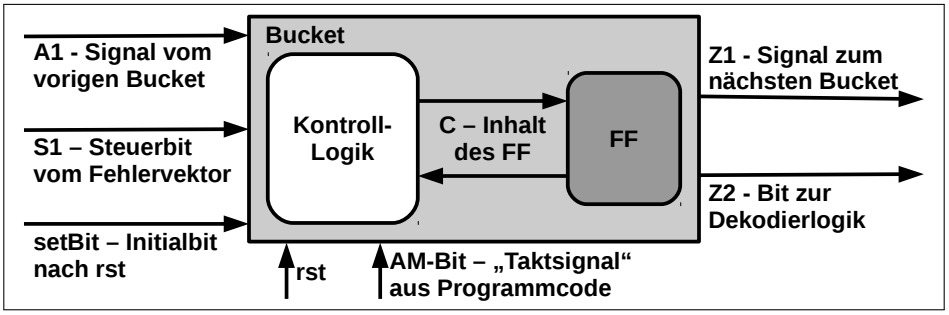


Abbildung 5.7.: Aufbau und Steuersignale des Buckets

Aufbau des Kontrollblocks und der darin enthaltenen, Buckets genannten, Bausteine wieder. Diese sind die eigentliche Steuerlogik für die AM und in den Abbildungen 5.6 und 5.7 dargestellt. Grundlegend besteht der Kontrollblock dabei immer aus genau N Buckets, wobei M Buckets die Aktivierung / Deaktivierung der originalen und R Buckets die der redundanten Komponenten kontrollieren. Weiterhin besitzt er eine Dekodierlogik für die Schalter vor und hinter dem M aus N System und erhält als Eingangssignale das oben beschriebene Kontrollbit aus dem Programmspeicher und das Initialmuster, bzw. den Fehlervektor aus dem Datenspeicher. Die Buckets sind alle identisch aufgebaut, zu einer Ringstruktur verbunden und der Inhalt ihres Flip Flops (FF) zeigt an, ob die entsprechende (redundante) Komponente aktiv bzw. inaktiv ist. Ihre Funktionsweise wurde in [UKV12b] wie folgt zusammengefasst, wobei die Signalnamen mit denen aus Abbildung 5.7 übereinstimmen: Im trivialen Nicht-Fehlerfall ist das Bit vom Fehlervektor S1 (das Ergebnis des Angenommenen Tests) immer ’0’. Damit wird in jedem durch das AM-Bit verursachten Takt der Inhalt des FF auf das Signal zum nächsten Bucket, Z1, und auf das Steuerbit für die entsprechende Komponente gelegt und das FF mit dem Wert des

Tabelle 5.1.: Wahrheitstabelle für Z1 und Z2 der Buckets

S1	C	A1	Z1	Z2
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

vorhergehenden, also A1 befüllt. Im Fehlerfall, also wenn S1 = '1' müssen mehrere Fälle unterschieden werden:

1. **Der Inhalt C des FF und A1 sind '0':**
   (Bucket wartet auf Inaktivitätsphase zur Sperrung der fehlerhaften FE)
   In diesem Fall wird wie oben vorgegangen werden, da kein Bit aus dem Kreislauf entfernt werden kann.

2. **C ist '0' und A1 ist '1':**
   (Inaktivitätsphase erreicht Bucket, dieser sperrt die FE)
   Nun verbleibt das Bit im Bucket. Das heißt, dass der Inhalt des FF und das Steuersignal für die deaktivierte Komponente Z2 auf '1' und Z1 auf '0' gesetzt werden.

3. **C ist '1':**
   (Sperre der FE bleibt erhalten)
   Das Signal A1 wird direkt an Z1 weitergeleitet, ansonsten verbleibt der Bucket im auf '1' gesetzten Zustand.

Eine Zusammenfassung der Signalkombinationen für die beiden Ausgänge Z1 und Z2 im Fehler- und Nicht-Fehlerfall ist in Tabelle 5.1 gegeben und der VHDL-Code für die Implementierung befindet sich in Anhang C.

Die wichtigsten Eigenschaften der Kontrolllogik Die im vorigen Abschnitt beschriebene Implementierung der Kontrolllogik mit den darin enthaltenen Buckets besitzt einige wichtige Eigenschaften, die im Folgenden noch einmal zusammengefasst sind:

- **Automatismus:**
  Ein großer Vorteil der Buckets ist, dass durch ihre Hilfe die AM stark vereinfacht und automatisiert wird. Von außen liegen der Initialvektor, der Fehlervektor vom Test und das AM-Bit an, ansonsten organisiert sich die Logik selbst.

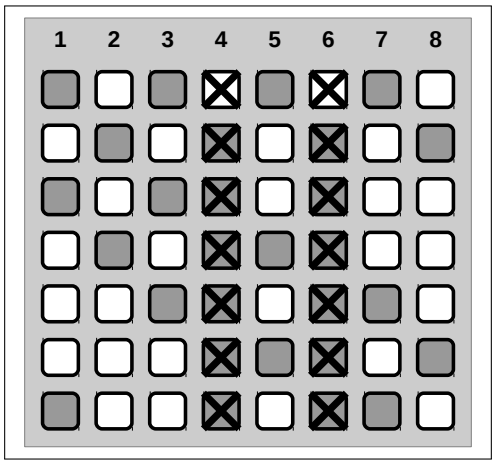


Abbildung 5.8.: Umschaltschema mit ungleich langen Aufwärm- und Abkühlphasen

- **Faire Lastverteilung:**
  Auch bei Ausfall von einer bis zu R Komponenten bleibt die Lastverteilung für alle aktiven Einheiten gleich. Nur die Dauer der Aufwärmzyklen kann ab dann variieren (ab $R > 2$). Um dies zu verdeutlichen, dient das Umschaltschema in Abbildung 5.8. Angenommen, es liegt ein 4 aus 8 System vor, das Initialmuster ist somit {10101010} und die Komponenten 4 und 6 wurden von einem diagnostischen Test als fehlerhaft markiert (auf die Darstellung der redundanten Komponenten wird hier verzichtet). Bereits ab der zweiten Zeile sind zwei der vier Inaktivitätsphasen blockiert und können nicht mehr weiter rotieren. An Komponente 8 lässt sich nun ablesen, dass sie drei aufeinanderfolgende AM-Takte aktiv ist, eine Phase aussetzt, wieder eine arbeitet, dann erneut eine aussetzt und anschließend wieder 3 Takte arbeitet. Der Grund dafür ist, dass die Inaktivitätsphasen vorher gleichmäßig verteilt waren (bsp.: {10101010}). Nun fehlen sie an einigen Stellen, was die Phasen zwischen ihnen ungleichmäßig lang macht - hier einmal drei und einmal einen Takt durch das Entfernen von zwei Phasen. Dies gilt natürlich nur, wenn noch mindestens zwei Inaktivitätsphasen vorhanden sind, da bei einer einzigen auch nur noch eine Arbeitsphase existiert.
  Mit Fair ist in diesem Zusammenhang gemeint, dass dieses Schema aber für alle Komponenten gilt und somit immer noch (unter der Annahme des LB) eine gleichmäßige Erwärmung / Abkühlung für alle gewährleistet ist, nur eben nicht mehr so optimal. Ein besserer Umgang mit den Inaktivitätsphasen in diesem Fall könnte natürlich implementiert werden, würde die Kontrolllogik aber deutlich vergrößern.

- **Skalierbarkeit:**
  Ein weiterer Vorteil der Buckets ist, dass ihre Anzahl, unabhängig von M und R, immer genau N beträgt und sie somit linear mit dem System mitwächst. Hierbei muss aber beachtet werden, dass dies für die Dekodierlogik im Kontrollblock (Abb. 5.6) nicht gilt. Ihr Wachstum fällt größer aus. Außerdem steigt mit wachsender Redun-

danz auch die Anzahl der Schalter, was deutliche Auswirkungen auf Zuverlässigkeit und MTTF mit sich bringt [KV11].

- **Online AM, offline Test:**
  Ein anderer wichtiger Punkt ist, dass die AM online durchgeführt wird. Der Test und das Anlegen der Fehlervektoren sollte jedoch offline geschehen. Dies hat den Grund, dass die Inaktivitätsphasen erst einmal solange rotieren müssen, bis alle fehlerhaften FE auch deaktiviert sind. Dies kann im Maximalfall bis zu $N-1$ AM-Takte dauern. Da ihre Häufigkeit aber im Programmcode festgelegt und sie somit in jedem Instruktionswort gesetzt werden können, ist dies in $N-1$ Prozessortakten erreichbar.

- **Activity Migration zur Lebensdaueroptimierung:**
  Um noch einmal explizit darauf hinzuweisen: Der wesentlichste Nutzen der hier vorgeschlagenen Lösung liegt in der Erhöhung der MTTF durch gleichzeitiges Anwenden der AM und der Selbstreparatur. Wie groß diese Erhöhung aber genau ist, soll in Kapitel 8 gezeigt werden.

5.3.3. Die fünf Implementierungen mit AM

Die hardwaremäßigen Erweiterungen, die an den fünf Implementierungen vorgenommen wurden, sind im Folgenden zusammengefasst und in Abbildung 5.12 grafisch dargestellt. Alle gleich gebliebenen Komponenten sind entweder weiß oder, im Falle des M aus N Systems, hellgrau. Neu hinzugekommene oder veränderte Komponenten sind dunkelgrau.

1. AM im M aus N System der Funktionseinheiten In der ersten Implementierung musste das AM-Bit in den Programmspeicher eingefügt und die Kontrolllogik (MaNS-Ktrl) erweitert werden (siehe Abb. 5.12(a)). Letzteres beinhaltete einerseits das Hinzufügen der Buckets, um die AM zu ermöglichen. Andererseits waren Anpassungen in der Dekodierlogik für die Schalter notwendig, da sich Probleme mit dem Timing ergaben. Diese sind stark vereinfacht in Abbildung 5.9 dargestellt worden. Die obere Hälfte zeigt dabei die Signalwerte gewisser Komponenten im nicht angepassten Fall. Der Ablauf ging dabei folgendermaßen vonstatten:

1. Gleichzeitig mit dem Clock-Signal wird das AM-Bit aus dem Programmspeicher gelesen (Zeile 1 & 2, Zeitpunkt t1).

2. Anschließend berechnen die Buckets die neue Kombination der (nicht) aktiven FE und werden zum Zeitpunkt t2 stabil (Zeile 3).

3. Daraufhin kann die Dekodierlogik die neue Ansteuerung für die Schalter ermitteln und ist ab t3 stabil.

4. Da die Schalter relativ komplex sind, benötigen sie bis t4, um stabil zu werden. Das Gleiche gilt für die Opcodes der FE, weil sie von den Schaltern gesteuert werden (Zeile 6)

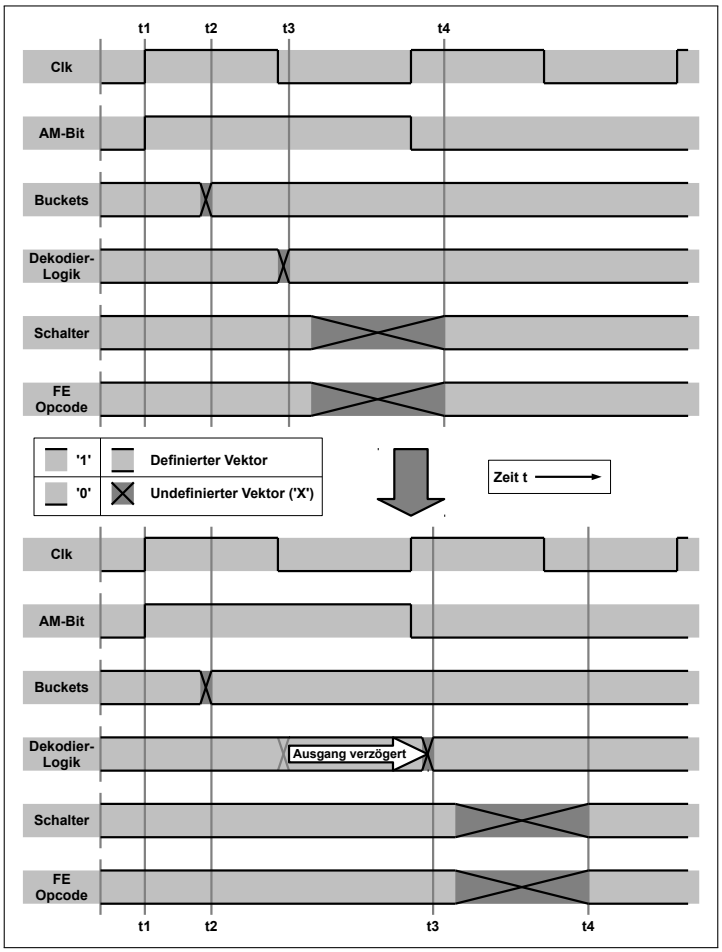


Abbildung 5.9.: Timingprobleme durch zu lange Pfade

Problematisch hieran ist, dass all diese Vorgänge nicht innerhalb von einem Takt ausführbar waren. Das hatte zur Folge, dass die FE bei der nächsten steigenden Taktflanke einen unbestimmten Opcode zugewiesen bekamen und damit eine quasi-zufällige Operation ausführten, was unter anderem auch Sprünge waren. Um dies zu beheben, wurden die Ausgänge der Dekodierlogik soweit verzögert, dass sie ihr Ergebnis erst zu Anfang des nächsten Taktes an die Schalter weitergeben (unteres Bild, t3). Damit erhalten die Schalter genügend Zeit, innerhalb eines Taktes stabil zu werden (t4) und somit ist auch der Zustand der FE wieder definiert.

2. AM im M aus N System der FE und Pipelineregister Bei der zweiten Implementierung des M aus N Systems kommt nun durch die online durchgeführte AM noch mehr zeitliches Verhalten hinzu. Das Problem hierbei ist, dass die Pipeline vier Takte benötigt, um einen Befehl zu bearbeiten. Wird sie dabei unterbrochen, kann es zu Fehlern kommen. Aus diesem Grund werden die Stufen der zu deaktivierenden Pipeline je nachdem, in welcher Stufe der abzuarbeitende Befehl gerade ist, sukzessive abgeschaltet, während die des

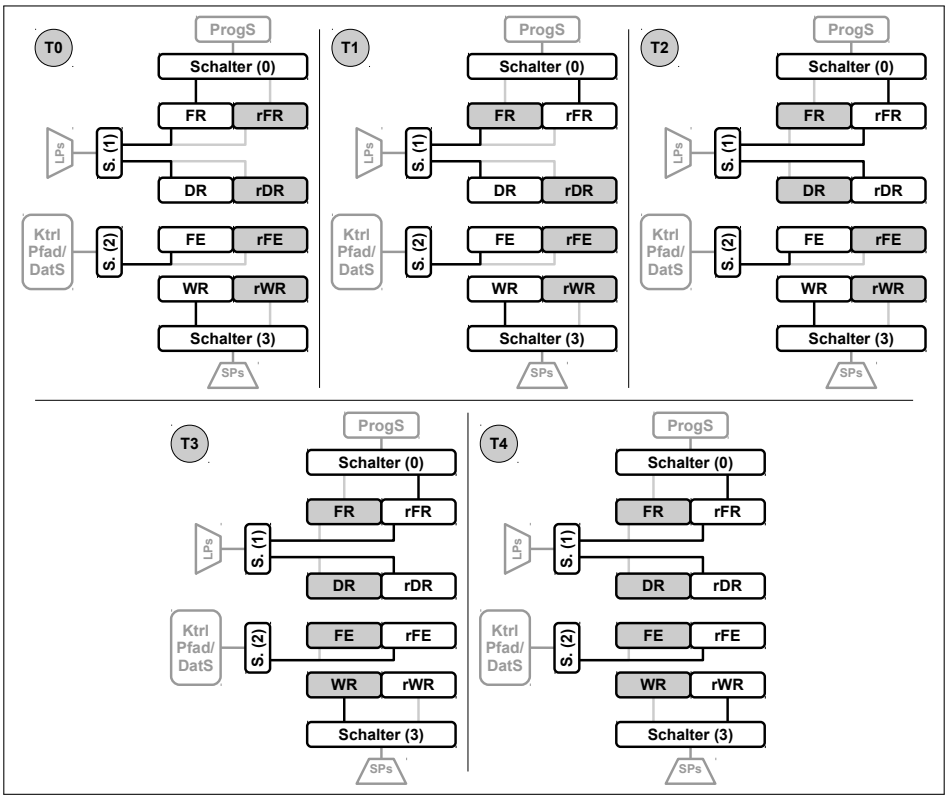


Abbildung 5.10.: Timing der Pipeline durch die Schalter

zu aktivierenden Slots nacheinander angeschaltet werden. Der so entstehende Ablauf ist schematisch in Abbildung 5.10 dargestellt. Aus Platzgründen gibt es nur einen originalen (links) und einen redundanten Slot (rechts). Auf die Darstellung von Registerbank und Datenspeicher wurde verzichtet. T0 repräsentiert dabei den Grundzustand, in dem der originale Slot aktiv ist, Buckets und Dekodierlogik aber bereits den neuen Zustand bestimmt haben (t3 in Abb. 5.9 unten). Im nächsten Takt können also die Schalter angesteuert werden. Weiße Blöcke sind hier wieder in der jeweiligen Phase aktiv.

In T1, also der Fetch Phase, werden die Schalter vor den Slots umgeschaltet. Damit erhält der redundante Slot eine Operation und der originale nicht. Die bereits im originalen Slot befindliche Operation läuft aber weiter durch. In der Fetch Phase, T2, werden die Schalter von und zu den Leseports aktiv. Damit kann der redundante Slot einen neuen Wert lesen und der originale wird entkoppelt. Die neu gelesenen Werte stehen nun im rDR. In der Ausführungsphase EXE (T3) findet nur die Entkopplung der in das System hinein führenden Signale von Kontrollpfad und Datenspeicher statt, da die nach außen führenden von Kontrollpfad und MAR / MBR selbst auf 1 aus N reduziert werden. Damit kann der redundante Slot jetzt beispielsweise die Programmadresse lesen, der originale nicht mehr. Der letzte Umschaltvorgang folgt in T4, der WB Phase. Der originale Slot hat nun bereits seinen Wert zurückgeschrieben und damit die Operation beendet. Somit kann er nun vom Schreibport entkoppelt werden und der redundante Slot sein Ergebnis schreiben.

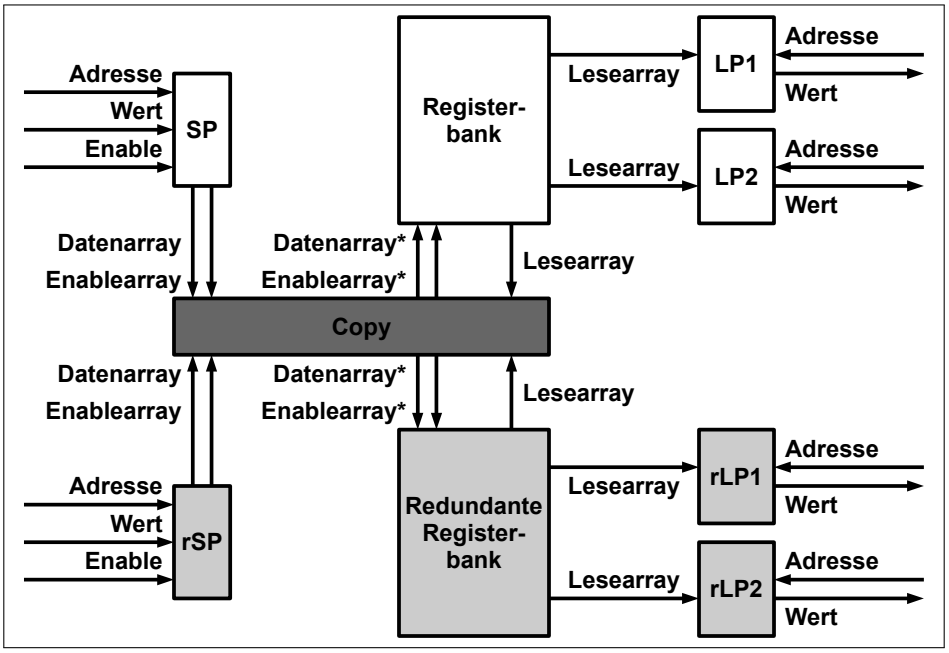


Abbildung 5.11.: Der Kopieroperator Copy

Auch hier war die einzige Komponente, die Anpassungen erfahren hat, die Kontrolllogik durch das Hinzufügen der Buckets und des zeitlichen Verhaltens in die Dekodierlogik.

3. AM im M aus N System der Slots, inklusive der Lese- und Schreibports Hier fanden die gleichen Erweiterungen wie in Implementierung 2 statt: Hinzufügen der Buckets und Erweitern der Dekodierlogik um das zeitliche Verhalten (Abb. 5.12(c)).

4. AM im M aus N System der Cluster, ohne Ports und Registerbänke Da im Prinzip nur weitere Slots (ohne SPs und LPs) hinzugekommen sind, benötigt auch Implementierung 4 die gleichen Anpassungen wie Implementierung 2 und 3.

Den einzigen Unterschied macht die Anwesenheit von MAR und MBR im M aus N System. Sie erhalten ihre Werte von den FE in der EXE Phase, geben diese aber erst in der WB Phase an den Datenspeicher weiter. Damit müssen die an den Datenspeicher gehenden Signale am mit (3) bezeichneten Zeitpunkt (siehe Abb. 5.12(d)) umgeschaltet werden, während die vom Datenspeicher an die FE zurückgehenden Signale bereits zum Zeitpunkt (2) umzuschalten sind. So kann die AM auch ausgeführt werden, während Lese- und Schreibvorgängen vom oder in den Datenspeicher statt finden.

5. AM im M aus N System der Cluster In Implementierung 5 werden nun online komplette Cluster ausgetauscht. Das beinhaltet neben beispielsweise den Pipelineregistern auch die Registerbank, was den Vorgang deutlich verkompliziert. Der Grund dafür ist, dass die Inhalte der Registerbank beim Umschalten von einem zum anderen Cluster gerettet werden

müssen, damit sie nicht verloren gehen. Hier musste nun als wichtigste Frage geklärt werden, wie dies geschehen kann. Mögliche Optionen sind der Austausch per Software, beispielsweise über zusätzliche Transferbefehle, Zwischenspeichern im Datenspeicher oder zusätzliche Kopierhardware.

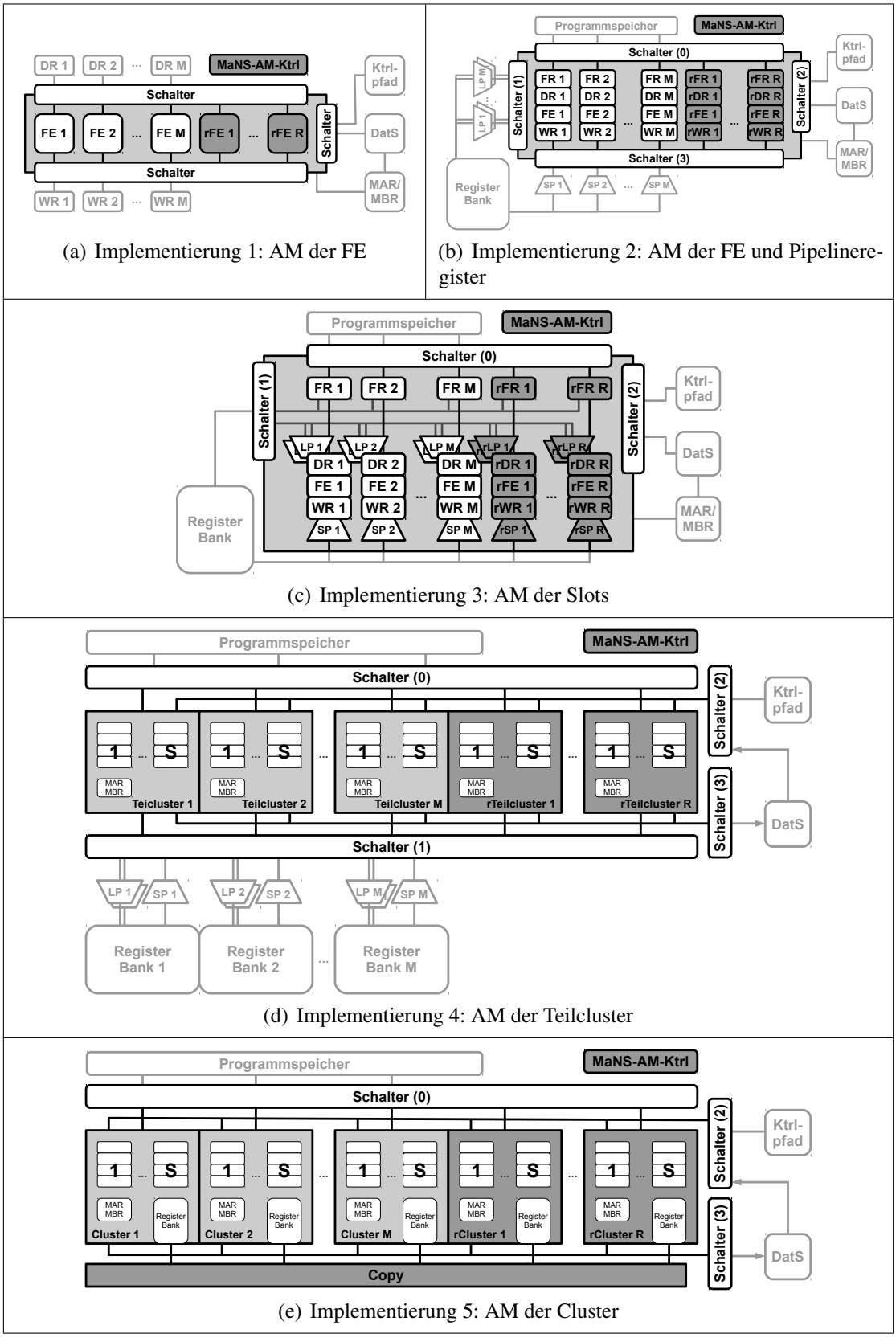
Für die vorliegende Arbeit ist der Einsatz von zusätzlicher Kopierhardware (Copy - siehe Abb. 5.12(e)) die beste Lösung. Diese ist imstande, eine vorher festgelegte Anzahl von Registern in den zu aktivierenden Cluster zu kopieren. Durch ihren Einsatz steigt der Hardwareaufwand für die AM zwar deutlich an, sie benötigt aber keine zusätzliche Zeit für Kopieroperationen, was das Temperaturverhalten des Prozessors beeinflussen könnte.

Der Aufbau der Kopierhardware ist noch einmal detailliert in Abbildung 5.11 dargestellt. Copy hat lesend auf die Registerbank Zugriff über den Lesearray, so wie auch die Leseports (LP1 und LP2). Zum Schreiben erhält er den Daten- und Enablearray (Die Signale für die Schreibfreigabe in die Register) der Schreibports (SP) und kann diese in die abgewandelten Signale Datenarray* und Enablearray* anpassen. Durchgeführt wird der Kopiervorgang in der IF Phase, also in Abbildung 5.12(e) zum Zeitpunkt (0). Die Anzahl der zu kopierenden Register kann dabei zwischen einem und allen beliebig in der VHDL-Beschreibung festgelegt werden.
Der Vorgang beginnt durch Setzen des AM-Bit. Im darauffolgenden Takt (0) liest Copy die Registerinhalte der originalen Registerbank über den Lesearray. Gleichzeitig prüft sie dessen Daten- und Enablearray, ob gerade noch in die zu kopierenden Register geschrieben wird und nimmt diese neuen Werte mit auf. Anschließend kopiert sie innerhalb des selben Taktes alle Werte an die gleichen Adressen in der redundanten Registerbank über dessen Datenarray* und Enablearray* Signale. Operationen, die sich jetzt in der IF, ID und EXE Phase im abzuschaltenden Cluster befinden, werden bis zum Ende ausgeführt. Daten, die dabei in die Registerbank geschrieben werden, gehen aber verloren. Dafür hat der neue Cluster direkt in dem auf die Kopieroperation folgenden Takt Zugriff zu den geretteten Werten.

Aufgrund dieses Verhaltens der Copy Komponente kann bei Implementierung 5 das Programm nicht mehr, wie ursprünglich geplant, ausgeführt werden, sonst kommt es zu Datenverlusten. Mögliche Lösungen wären, vor dem Umschalten drei Leerlaufbefehle einzufügen (NOP) oder diese Zeit zum Datentransfer in den Datenspeicher zu nutzen. Dann könnte auch der Funktionsumfang der Copy Komponente unter Umständen geringer ausgelegt werden, da weniger Register zu retten sind. Prinzipiell wäre es auch möglich, die Registerbank des redundanten Clusters als Verlängerung des originalen zu nutzen. Die Werte werden beim Umschalten nicht gelöscht. Damit wäre nur ein Umschaltvorgang notwendig, um weitere Register zur Verfügung zu haben. An dieser Stelle muss jedoch sichergestellt werden, dass die Daten beim Umschalten ausschließlich zwischen diesen beiden Clustern wechseln und keine anderen mit darauf migriert werden. Dies wird aber äußerst kompliziert, sobald Cluster aufgrund eines Fehlers ausfallen, da sich dann das Umschaltschema ändert. Im vorliegenden Fall müssen Datenabhängigkeiten aber nicht betrachtet werden, da die Generierung der Software zufällig erfolgt. Wichtig ist nur, die originalen und redundanten Registerbänke vor Beginn der AM mit Werten zu füllen, falls nicht alle Werte kopiert werden. Dann greifen die zufällig generierten Operationen nicht

auf die leeren Teile der Registerbänke zu und verfälschen nicht die späteren Simulationen. Im Normalfall wäre es Aufgabe des Assemblers, nur die Register zu nutzen, die auch befüllt wurden.

Aufbau und Ansteuerung der Schalter ist ansonsten ähnlich wie in Implementierung 4. Nur kann jetzt auf Schalter zwischen den Pipelineregistern und den Lese- und Schreibports verzichtet werden.



(a) Implementierung 1: AM der FE

(b) Implementierung 2: AM der FE und Pipelineregister

(c) Implementierung 3: AM der Slots

(d) Implementierung 4: AM der Teilcluster

(e) Implementierung 5: AM der Cluster

Abbildung 5.12.: Grafische Übersicht der fünf Implementierungen mit AM

Kapitel 6

Simulation der Temperatur in den Prozessoren

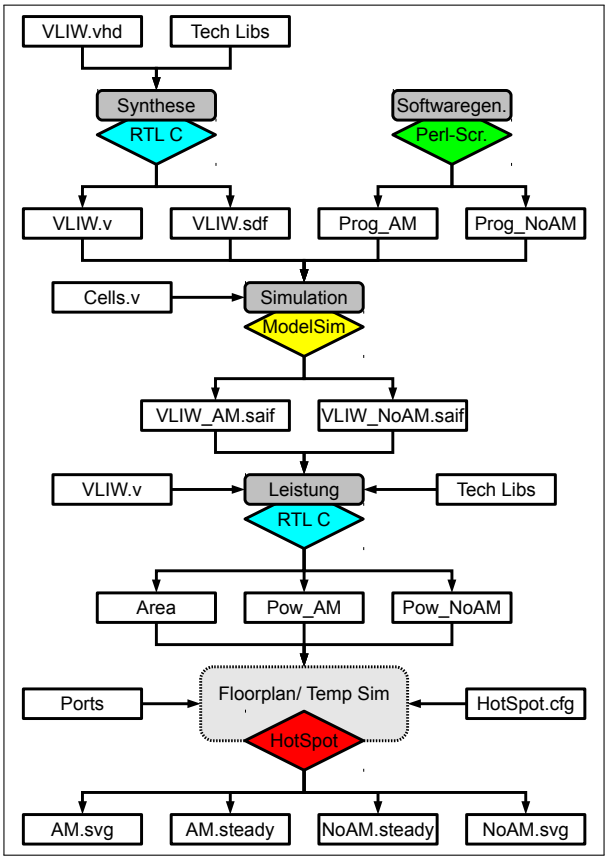


Abbildung 6.1.: Toolflow für die Hitzesimulationen

Im letzten Kapitel wurde die Implementierung von M aus N Systemen mit Thermal Management im VLIW Prozessor beschrieben und die dadurch entstehenden Kosten begründet. Zweiter Teil der Untersuchungen ist es, den Nutzen dieser Methode zu bestimmen. Dies geschieht, wie bereits angedeutet, über verringerte Temperatur als Anzeichen für reduzierten Stress und weniger stark hervortretende Fehlereffekte. Die wohl genaueste Möglichkeit, Ergebnisse zu erzielen, wäre das Herstellen der Systeme und das anschließende Messen der Betriebstemperatur. Da aber fünf verschiedene Implementierungen

untersucht werden, wäre der Aufwand hierfür enorm. Aus diesem Grund wird auf Simulationen zurückgegriffen. Diese sind zwar weniger exakt, ermöglichen aber umfassende Untersuchungen und liefern zumindest realitätsnahe Ergebnisse.

Das vorliegende Kapitel beschreibt nun die Durchführung der Simulationen mit ihren teils sehr aufwändigen Zwischenschritten. Abbildung 6.1 dient dabei als Überblick über die jeweiligen Schritte und generierte Ein- und Ausgaben. Getroffene Annahmen, eingesetzte Parameter und die ermittelten Ergebnisse werden anschließend in Kapitel 8 dargelegt.

6.1. Synthese

Der sogenannte „Toolflow“, also die Kette von Programmen zur Generierung der Ergebnisse, beginnt mit der Synthese. Diese übersetzt die Beschreibungen der Implementierungen von Register-Transfer-Ebene auf Logikebene, wozu der Cadence Encounter RTL Compiler (RTL C, siehe Abb. 6.1 und Anhang A) eingesetzt wird. Als Eingabe dienen jeweils die fünf VHDL Beschreibungen der Implementierungen (VLIW.vhd) und die Technologiebibliotheken (Tech Libs).

```
1  # Technologie Bibliotheken einlesen
2    set_attribute library {Tech.lib}
3    set_attribute lef_library {Tech.lef}
4  # VHDL Beschreibung einlesen
5    read_hdl -vhdl VLIW.vhd
6  # VHDL elaborieren
7    elaborate [Top]
8  # Die Clock setzen
9    define_clock -design vliwCore -name clk1 -period 1000 /
     designs/vliwCore/ports_in/clock
10 # Technologieunabhängige Synthese
11   synthesize -to_generic
12 # Technologieabhängige Synthese
13   synthesize -to_mapped -effort high
14 # Ergebnis ausgeben
15   write_hdl > VLIW.v
16 # Datei mit Verzögerungen der Gatter ausgeben
17   write_sdf -celltiming nochecks > file.sdf
```

Quelltext 6.1: RTL Compiler: Synthese

Die Synthese selbst wird im vorliegenden Fall wie in Quelltext 6.1 konfiguriert. Erster Schritt ist dabei das Einlesen der Technologie Bibliotheken und der VHDL Beschreibungen. Anschließend wird die Taktrate des Systems festgelegt. Diese variiert je nach Implementierung (hier 1000ps - 1 GHz). Danach folgt die Synthese und die Ausgabe der Designs im *Verilog* Format. Als zweite Ausgabe entsteht eine standard delay format (sdf) Datei, die die Verzögerungen in den Gattern angibt. Diese wird später für zeitlich genaue Simulationen mit ModelSim benötigt. Da ModelSim immer eine Fehlermeldung

hervorbrachte, sobald Blöcke mit dem Label „TIMINGCHECK“ auftraten, mussten diese bei der Erstellung ausgeschlossen werden (-celltiming nochecks).

6.2. Generierung des Programmcodes

Die Generierung der Software geschieht, wie bereits in Abschnitt 5.3.1 angedeutet, über ein Perl Script. Das entstehende, zehntausend zeilige Programm enthält am Anfang die Initialisierung der Registerbänke mit Zufallswerten und anschließend in jeder Zeile zufällig gewählte, arithmetische und logische Operationen sowie Transferbefehle mit Konstanten in die Registerbank.

Von dieser Routine werden dabei immer zwei Versionen erstellt: eine ohne gesetztes Bit für den AM-Takt (Prog_NoAM) und eine, bei der dieses aller zehn Takte gesetzt ist (Prog_AM). Letzteres initiiert dabei somit die Activity Migration aller zehn Takte. Die Routinen sind (ob mit oder ohne AM) jeweils für die Implementierungen 1 bis 3 und 4 und 5 identisch.

Die AM Zyklen zehn Takte lang zu gestalten, ist eher willkürlich geschehen, orientiert sich aber grundlegend an der Länge von „Basisblöcken“, wie in Abschnitt 3.2.1 beschrieben. Für die Implementierungen 1 bis 4 ist dies durchaus realistisch. Bei Implementierung 5 hingegen wird somit das Kopieren der Registerinhalte sehr oft angesteuert, was bei Ausführung eines echten Programms aufgrund von Datenabhängigkeiten etc. problematisch sein kann. Im vorliegenden Fall führt es jedoch lediglich dazu, dass die Kopiereinheit COPY sehr viel Leistung umsetzt.

6.3. Simulation

Der nächste Schritt ist die Simulation der Implementierungen während der Ausführung des soeben generierten Programmcodes, um die Schaltaktivität innerhalb der Prozessoren zu ermitteln. Als Eingabe dienen die Verilog-Beschreibungen der Implementierungen und der Zellen in den Bibliotheken (VLIW.v und Cells.v), die .sdf Datei mit den Verzögerungen (VLIW.sdf) und die beiden Versionen des Programms (Prog_AM/NoAM).

Als ausführendes Programm für die Simulationen dient ModelSim (siehe Anhang A), welches mit folgenden Befehlen gestartet wird:

```
1  vsim -sdfmax /vliw_tsb/VLIW/core=vliwCore.sdf -t ps work.
   vliw_tsb -novopt
2  power add -r *
3  run 10000 ns
4  power report -all -bsaif vliwCore.saif
```

Quelltext 6.2: ModelSim: Erzeugung der SAIF-Datei

Die erste Zeile startet dabei die Simulation mit den maximalen Verzögerungen (typ und min waren nicht angegeben) im Zeitbereich Pikosekunden und ohne interne Optimierungen.

Zeile Zwei stellt sicher, dass alle Signale des Prozessors auf Schaltaktivität untersucht werden, und die dritte legt die Simulationsdauer fest. Diese variiert je nach Taktrate, da mehr oder weniger Zeit zur Abarbeitung der Routine fester Länge benötigt wird. Die letzte Zeile legt die Ausgabe auf das switching activity interchange format (SAIF) fest und gibt die entsprechenden Werte aus. Das SAIF ist dabei so aufgebaut, dass für jeden Ein- und Ausgang jeder Zelle des Systems festgehalten wird, wie viel Zeit der Simulationsdauer er auf logisch '0' oder '1' war und wie oft dazwischen umgeschaltet wurde. Diese Datei entsteht einmal mit (VLIW_AM.saif) und einmal ohne Activity Migration (VLIW_NoAM.saif).

6.4. Ermittlung der Leistung

Die Ermittlung der durchschnittlichen Leistung der Komponenten geschieht nun erneut mit Hilfe des RTL Compilers (RTL C, siehe Abb. 6.1 und Anhang A). Dabei dienen die Verilog Beschreibung der Implementierungen, die Technologie Bibliothek und die soeben generierten .saif Dateien als Eingabe. Ausgeführt wird das Programm dabei mit folgenden Befehlen:

```
1  # Technologie Bibliotheken einlesen
2   set_attribute library {Tech.lib}
3   set_attribute lef_library {Tech.lef}
4  # Verilog Beschreibung einlesen
5    read_hdl vliwCore.v
6  # Verilog Beschreibung elaborieren
7    elaborate [Top]
8  # Die Clock setzen
9    define_clock -design vliwCore -name clk1 -period 1000 /
     designs/vliwCore/ports_in/clock
10 # SAIF Datei einlesen
11   read_saif -instance core vliw.saif
12 # Die Leistungsaufnahme des Systems ausgeben
13   report power -hier -full_instance_names > Pow
14 # Die Fläche des Systems und seiner Komponenten ausgeben
15   report area -physical > reports/Area
```

Quelltext 6.3: RTL Compiler: Erzeugung der Area- und Power-Reports

Als Ausgaben entstehen der Bericht zur Fläche des Systems und seiner Komponenten (Area) und die Angabe deren aufgenommener Leistung (Pow_AM/NoAM).

6.5. Floorplanning und Temperatursimulation mit HotSpot

Die nächsten beiden Schritte sind die Generierung der Floorplans, also die Festlegung der Anordnung der Komponenten auf dem Chip, und anschließend die Simulation des Systems, um die entstehende Temperatur zu erhalten. Hierzu wird in beiden Fällen das Programm

HotSpot eingesetzt. Da es im Gegensatz zu den anderen, kommerziellen Programmen wie ModelSim oder dem RTL Compiler noch keinen so großen Bekanntheitsgrad besitzt, wird es im Folgenden noch einmal genauer vorgestellt. Daran schließt sich die Beschreibung der Erzeugung der Floorplans und der Durchführung der Simulationen an.

6.5.1. Das HotSpot Programm

HotSpot wurde an der Universität von Virginia in den USA entwickelt und wird auf ihrer Seite (siehe Anhang A) wie folgt beschrieben:

> *„HotSpot is an accurate and fast thermal model suitable for use in architectural studies. It is based on an equivalent circuit of thermal resistances and capacitances that correspond to microarchitecture blocks and essential aspects of the thermal package. The model has been validated using finite element simulation.“* [Hot13]

Stark vereinfacht arbeitet HotSpot wie folgt: Die Komponenten des Systems werden als Blöcke mit X und Y Position und einer horizontalen und vertikalen Ausdehnung im Floorplan beschrieben. Jedem dieser Blöcke wird eine gewisse Leistungsaufnahme an gewissen Zeitpunkten zugewiesen. Anschließend folgt die Modellierung dieser Blöcke als Schaltkreisäquivalent mit thermalen Widerständen und Kapazitäten. Diese Äquivalente werden daraufhin mit der durch die umgesetzte Leistung verursachten Wärmeerzeugung über die Zeit und der Wärmeleitung zwischen den Blöcken simuliert. Als Resultat erzeugt HotSpot eine mittlere statische Temperatur des Systems, den Temperaturverlauf über die Simulationsdauer und eine grafische Version der statischen Temperatur.

Für die hier durchgeführten Untersuchungen wurde HotSpot als Simulationsumgebung gewählt, da es:

- kostenfrei und open source zur Verfügung gestellt wird,
- in Fachkreisen immer breitere Anwendung findet,
- den Floorplanner mitbringt, also dafür kein zusätzliches Programm vonnöten ist und
- den Floorplan sehr einfach und in abstraktem Format beschreibt, was Freiheiten beispielsweise für die Modellierung des Systems als Teil eines System on Chip (SOC) lässt (mehr dazu in Kap. 8).

6.5.2. Vorbereiten des Floorplannings

Um Floorplans für die Implementierungen erstellen zu können, werden drei Dateien benötigt (siehe Abb. 6.2): Eine .desc Datei, die Angaben zu den Flächen und Verbindungsleitungen zwischen den Komponenten enthält, eine .p Datei, die deren Leistungsaufnahme beschreibt und die Konfigurationsdatei .config, die die Gewichte für den Planungsalgorithmus festlegt. Diese müssen im Vorfeld aus den bisher erlangten Ergebnissen erstellt werden, was nun beschrieben wird.

Zuerst zur .desc Datei. Die Angaben über die Fläche der Komponenten stehen in der Area Datei und werden vom RTL Compiler wie folgt ausgegeben:

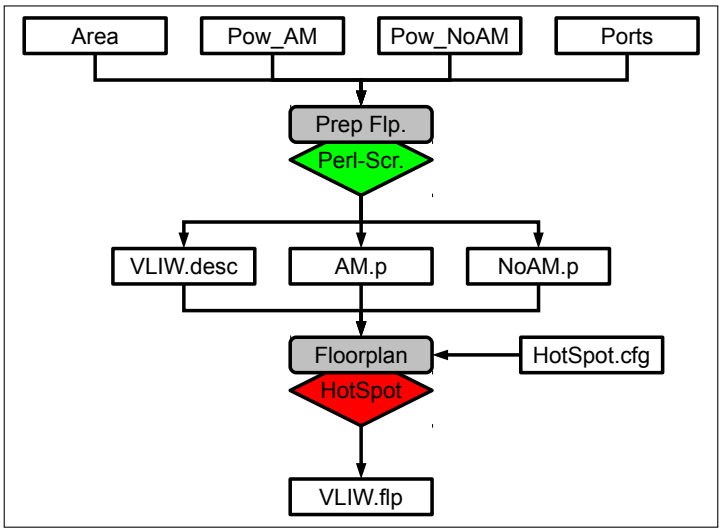


Abbildung 6.2.: Erstellen der Floorplans mit HotSpot

```
1   Instance                          Cells   Cell Area   Net Area
2  ------------------------------------------------------------------
3  vliwCore                          49587       75117     223269
4    readPorts                       11771       15331      46510
5    registerFile                     2049       14164      30273
6    writePorts                       5067        6817      20521
7    FUs[2]_FU                        3336        3820      13327
8      mul_198_39                     1228        1502       4378
9      csa_tree_sub_108_33_groupi      282         301        825
10     addinc_add_104_33_1             259         260        786
11     ...
```

Quelltext 6.4: Area Datei

Es lässt sich erkennen, dass die Komponenten des Systems immer weiter eingerückt werden, je tiefer sie in der Designhierarchie platziert sind. Im vorliegenden Fall interessieren dabei nur die einmal eingerückten Komponenten, da das den bisherigen Beschreibungen der Systeme (Abb. 5.3 und 5.4) entspricht. Von diesen Werten wird weiterhin nur die Cell Area, also die Summe der Größe der eingeschossenen Zellen, für die Modellierung des Floorplans benötigt. Die Flächenangabe erfolgt jeweils in μm^2. Sie muss für die .desc Datei in m^2 umgerechnet und in folgendes Format gebracht werden:

```
1 #Format: <unit-name> <area-in-m2> <min-aspect-ratio> <max-aspect
  -ratio> <rotable>
2
3 readPorts                15331e-12      1      4      1
4 registerFile             14164e-12      1      4      1
5 writePorts               6817e-12       1      4      1
6 FUs[2]_FU                3820e-12       1      4      1
7 ...
```

Quelltext 6.5: .desc Datei: Flächeninformation

Die letzten drei Werte geben die minimale und maximale Streckung (das Verhältnis von Länge zu Breite) und die Rotierbarkeit an (0 = nicht rotierbar, 1 = rotierbar) und sind an die mit HotSpot mitgelieferten Beispieldateien angelehnt.

Weiterhin müssen die Informationen über die Konnektivität der Komponenten in folgendem Format in die .desc Datei geschrieben werden:

```
1 # Format <unit1-name> <unit2-name> <wire_density>
2
3 FETCHs[0]_Fetch          DECs[0]_decodeReg          26
4 FETCHs[0]_Fetch          readPorts                  12
5 readPorts                DECs[0]_decodeReg          64
6 DECs[0]_decodeReg        BUFs[0]_aluBuf             6
7 ...
```

Quelltext 6.6: .desc Datei: Konnektivität

Die Angaben wurden den Busbreiten in den VHDL Beschreibungen der Implementierungen entnommen (zusammengefasst als Datei Ports).

Anschließend folgt die Generierung der .p Dateien. Der RTL Compiler gibt die Informationen zur Leistungsaufnahme der Komponenten wie folgt aus (aus Platzgründen wird hier auf die Darstellung der Spalten *Cells, Leakage Power* und *Dynamic Power* verzichtet):

```
1                                                  Total
2              Instance                           Power(nW)
3 ---------------------------------...-------------------
4
5 vliwCore                                 ...  90179727.856
6   readPorts                              ...   6336273.363
7   registerFile                           ...  11360291.390
8   writePorts                             ...   4475531.882
9   RFUs[5]_FU                             ...   6025536.336
10    mul_198_39                           ...   2694746.843
11    csa_tree_sub_108_33_groupi           ...    591467.790
12    final_adder_sub_106_26               ...    354709.494
```

```
13      addinc_add_104_33_1          ...          377440.065
14      ...
```

Quelltext 6.7: Power Datei

Die Informationen müssen wiederum in Watt umgerechnet und in folgendem Format in der .p Datei ausgegeben werden, wobei auch hier nur die einmal eingerückten Komponenten interessieren:

```
1 readPorts                 6336273.363e-9
2 registerFile             11360291.390e-9
3 writePorts                4475531.882e-9
4 RFUs[5]_FU                6025536.336e-9
5 ...
```

Quelltext 6.8: .p Datei: Durchschnittliche Leistung

6.5.3. Erstellen der Floorplans

Sind die .desc und .p Dateien erzeugt, kann der HotSpot eigene Floorplanner gestartet werden. Wie auch später das Programm zur Temperatursimulation, benötigt dieser eine Konfigurationsdatei namens HotSpot.cfg. Sie enthält unter anderem Angaben zu den thermischen Eigenschaften des Systems und weitere Faktoren, wie seine Taktrate etc. Außerdem werden darin die Parameter des Floorplanners gesetzt, wobei im vorliegenden Fall nur die letzten drei von Interesse sind:

```
1 # weights for the metric: lambdaA * A + lambdaT * T + lambdaW *
  W
2    # weight for the area term
3      -lambdaA                  1.0e+09
4    # weight for the temperature term
5      -lambdaT                        0
6    # weight for the wire length term
7      -lambdaW                        1
```

Quelltext 6.9: Konfiguration des Floorplanners

Hier wird die Kostenfunktion des Floorplanners beeinflusst, dessen Algorithmus im Wesentlichen auf der *Simulierten Abkühlung* beruht. Das heißt, dass die Größe dieser Zahlen bestimmt, worauf bei der Planung am meisten Wert gelegt wird: die Fläche, die Temperatur oder die Verdrahtungslänge. Natürlich können über die Parameter auch Mischformen bestimmt werden. Eine weitere Aufgabe dieser Zahlen ist es, die Unterschiede in den Dimensionen, beispielsweise zwischen Fläche (m^2e-12) und Verdrahtungslänge (in mm),

aufzuheben. Als Beispiel wurde hier das Gewicht für die Fläche auf $\lambda A = 1.0e+09$ gesetzt. Der zweite Parameter wurde stets auf $\lambda T = 0$ festgelegt, um Temperatureinflüsse vorerst nicht zu beachten. Das Gewicht für die Verdrahtungslänge wurde durch Probieren herausgefunden und anschließend auf $\lambda W = 1$ bestimmt. Die resultierenden Floorplans sprechen für diesen Wert, da stark verbundene Komponenten immer sehr nah beieinander liegen.

Der Floorplanner selbst wird anschließend mit folgenden Befehlen gestartet:

```
1 ./hotfloorplan -c HotSpot.cfg -f vliw.desc -p vliw.p -o vliw.flp
2 perl tofig.pl vliw.flp  | fig2dev -L ps | ps2pdf - vliw_flp.pdf
```

Quelltext 6.10: Ausführen des Floorplanners

Die erste Zeile führt dabei das hotfloorplan Programm mit der Konfigurationsdatei HotSpot.cfg und der passenden .desc und .p Datei aus. Anschließend entsteht der Floorplan des Prozessors im HotSpot eigenen .flp Format:

```
1 readPorts 0.00006365561 0.00024084285 0.00000000000
  0.00013189642
2 registerFile  0.00006006066 0.00023621950 0.00006365561
  0.00013651978
3 writePorts  0.00004635602 0.00014705750 0.00012371628
  0.00018206552
4 ...
5 _0  0.00001132475 0.00001382855 0.00000000000 0.00011806787
6 _1  0.00000343213 0.00000530138 0.00019884739 0.00009418311
7 _2  0.00000038206 0.00000604152 0.00020189746 0.00004048654
8 _3  0.00000333045 0.00000604152 0.00014788237 0.00033737434
```

Quelltext 6.11: .flp Datei: Floorplan

Die erste Zeile gibt dabei den Namen der Komponente, die zweite und dritte deren Breite und Höhe und die vierte und fünfte deren X und Y Position ab der linken unteren Ecke an. Da diese Zahlen schwer zu lesen sind, wandelt sie der zweite Befehl aus Code 6.10 in eine Grafik um und gibt sie als PDF aus. Eine Besonderheit des Floorplans sind die mit einem Unterstrich beginnenden Komponenten. Diese tauchen nicht in der originalen VHDL Beschreibung auf, sondern sind vom Floorplanner eingefügte „Lückenfüller“ oder „Platzhalter“. Sie ergeben sich daraus, dass die rechteckigen Komponenten in Summe nicht wieder ganz genau eine rechteckige Fläche ergeben. Der Rest wird mit solchen Platzhaltern befüllt.

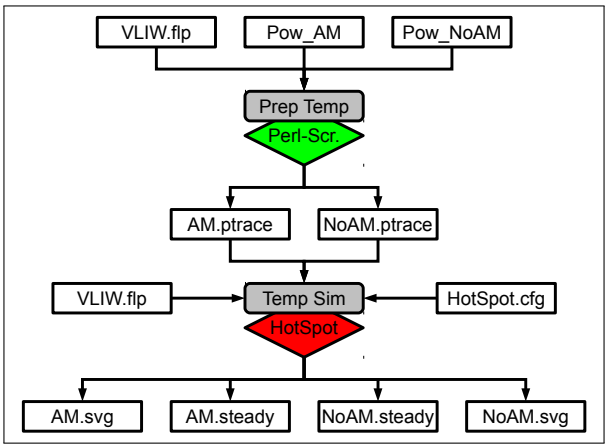


Abbildung 6.3.: Temperatursimulation mit HotSpot

6.5.4. Temperatursimulation

Wie bereits der Floorplanner, benötigt auch das Programm zur Temperatursimulation von HotSpot einige Dateien als Eingaben, die teilweise noch erstellt werden müssen.

Von zentraler Bedeutung ist hier erneut die Konfigurationsdatei .cfg, die Parameter wie Dicke des Chips, Wärmeleitfähigkeit des Substrats etc. enthält. Darin müssen nun zwei weitere Parameter angepasst werden: das Aufrufintervall und die Frequenz des Prozessors.

```
1 # hotspot calling interval - every 100 cycles
2   -sampling_intvl    1e-06
3 # base processor frequency in Hz - here: 1 GHz
4   -base_proc_freq    1e+09
```

Quelltext 6.12: Konfiguration der Temperatursimulation

Die Prozessorfrequenz gibt die Taktrate der jeweiligen Implementierung an (hier: $1e^9$ HZ - 1 GHz). Der zweite Wert hängt mit einer weiteren Eingabedatei zusammen, der .ptrace. Darin finden sich Momentanwerte der Leistung pro Komponente in Zeilen untereinander. Das heißt, die erste Zeile gibt den Momentanwert der Leistung der Komponenten zu Zeitpunkt A, die zweite zu Zeitpunkt B, die dritte zu Zeitpunkt C usw. an. Das Aufrufintervall bestimmt nun die tatsächliche Häufigkeit dieser Zeitpunkte, wann also HotSpot die Momentanleistung „misst“. Im obigen Beispiel wäre dies aller $1e-06$ Sekunden oder aller 1000 Takte.

Die .ptrace Datei gibt demnach die Messwerte der Momentanleistung an, die Hotspot vom System erhält. Um hier möglichst realistische Ergebnisse zu erhalten, müssen bei der Erstellung zwei etwas konträre Sachverhalte beachtet werden. Einerseits ist es wichtig, diese Datei mit vielen Zeilen zu füllen und das Aufrufintervall relativ hoch zu setzen, um lange Laufzeit des Programms zu simulieren und damit Messwerte nach der Aufwärmpha-

se des Systems zu erhalten, wenn es von der Temperatur her also stabil ist. Andererseits soll aber die Activity Migration abgebildet werden, was heißt, dass beispielsweise die Funktionseinheiten aller zehn Takte zwischen aktiver Phase (hohe Leistung) und passiver Phase (Leistung aufgrund des Ruhestroms) wechseln sollten. Dies würde aber die Simulationszeit deutlich verlängern, wenn lange Laufzeiten das Ziel sind. Da im vorliegenden Fall aber nur die mittlere Temperatur von Bedeutung ist und nicht, wie beispielsweise die Funktionseinheiten sich während ihrer aktiven und passiven Phase kurzzeitig aufwärmen und wieder abkühlen, werden die Dateien anhand der mittleren Leistung einmal mit und einmal ohne Activity Migration erstellt.

Demnach sehen die .ptrace Dateien wie folgt aus: Die erste Zeile enthält immer die Namen aller Komponenten (inklusive Platzhalter). Ab Zeile Zwei wird anschließend nur noch die mittlere Leistung der Komponenten wiederholt ($P_{Platzhalter} = 0W$), bis über die Anzahl von Zeilen die gewünschte Simulationsdauer erreicht wurde. Der folgende Block gibt zu Demonstrationszwecken eine stark gekürzte Version der .ptrace von Implementierung 1 an. Die Bezeichner der Komponenten wurden dafür gegen die in dieser Arbeit eingesetzten Bezeichner ausgetauscht und die Zahlen von beispielsweise $10083387.917e-9$ auf zwei Dezimalstellen vor und eine nach dem Komma gekürzt. Auch auf die Zehnerpotenz wurde verzichtet. Alle aufgeführten Werte sind also $* e^{-}3$ zu nehmen und in Watt angegeben.

```
1 # Ohne Activity Migration (NoAM.ptrace:)
2 ... FR3 FE0 FE1 FE2 FE3 rFE0 rFE1 rFE2 rFE3 _0 ...
3 ... 0.3 10  0.2 11  0.2 11  0.2  10  0.2  0 ...
4 ...
5
6 # Mit Activity Migration (AM.ptrace:)
7 ... FR3 FE0 FE1 FE2 FE3 rFE0 rFE1 rFE2 rFE3 _0 ...
8 ... 0.3 5.7 5.8 6.2 6.2 6.1  6.0  5.8  5.6  0 ...
9 ...
```

Quelltext 6.13: .ptrace Dateien: Verlauf der Leistungsaufnahme

Bereits hier lässt sich erkennen, wie die Activity Migration die mittlere Leistungsaufnahme der Komponenten beeinflusst.

Nun kann die Temperatursimulation gestartet werden. Dies geschieht mit folgenden Befehlen:

```
1 ./hotspot -c 045.cfg -f vliw.flp -p vliw.ptrace -model_type grid
  -grid_steady_file vliw.grid.steady
2 perl grid_thermal_map.pl vliw.flp vliw.grid.steady > vliw.svg
```

Quelltext 6.14: Ausführen der Temperatursimulationen

Die erste Zeile bewirkt, dass HotSpot in den „Grid Modus“ wechselt. Das heißt, dass es die Komponenten nicht an sich simuliert, sondern in noch kleinere Blöcke zerlegt und erst anschließend simuliert. Dies hat den Vorteil einer erhöhten Genauigkeit, verlangsamt aber die Berechnungen. Als Ausgabe entsteht eine Datei, die die statische Temperatur dieser Blöcke enthält. Soll der Verlauf der Erwärmung zusätzlich angezeigt werden, muss die Option *-o vliw.ttrace* hinzugefügt werden. Anschließend kann über die zweite Zeile eine SVG Datei erzeugt werden, die die statische Temperatur der kleineren Blöcke im Floorplan grafisch ausgibt.

Kapitel 7

Modellierung der Zuverlässigkeit und mittleren Lebensdauer

Notation

N	Anzahl aller Komponenten im M aus N System
M	Anzahl der aktiven Komponenten im M aus N System
R	Anzahl der passiven Komponenten im M aus N System ($R = N - M$)
B	Anzahl der Baugruppen innerhalb einer austauschbaren Komponente des M aus N Systems
S	Anzahl der Komponenten außerhalb des M aus N Systems
C	Anzahl der Technologiezellen einer Komponente
t, u	Variablen der Zeit
i	Zählvariable
K, L	Bezeichner für Komponenten
A	Fläche einer Komponente / Baugruppe
λ	Zeitlich konstante Fehlerrate
$f(t)$	Wahrscheinlichkeitsdichtefunktion (PDF) eines Ausfallzeitpunktes in Abhängigkeit der Zeit t
$F(t)$	Ausfallwahrscheinlichkeit in Abhängigkeit der Zeit t
$R(t)$	Zuverlässigkeit in Abhängigkeit der Zeit t
$MTTF$	Mittlere Lebensdauer
$P(.)$	Wahrscheinlichkeit P des Eintritts eines Ereignisses
T	Ausfallzeitpunkt des Gesamtsystems, exponentiell verteilte Zufallsvariable
X, Y	Ausfallzeitpunkte von Baugruppen / Komponenten / Subsystemen, exponentiell verteilte Zufallsvariablen
$1a1$	Index als Zusammenfassung für alle Komponenten außerhalb des M aus N Systems
$NoAM$	Index als Zusammenfassung für alle Komponenten innerhalb des M aus N Systems, wenn keine Activity Migration durchgeführt wird
AM	Index als Zusammenfassung für alle Komponenten innerhalb des M aus N Systems, wenn Activity Migration durchgeführt wird
$mean(.)$	Das arithmetische Mittel der Werte in Klammern

Eine Senkung der Temperatur lässt zwar Rückschlüsse auf eine Verringerung des Stresses zu, reicht aber für sich betrachtet noch nicht aus, um den Nutzen des Systems in Bezug auf eine Lebensdaueroptimierung zu bestimmen. Hierzu müssen die veränderten Fehlerraten errechnet und darüber die Zuverlässigkeit und mittlere Lebensdauer der Systeme ermittelt werden. Dies wird im nun folgenden Kapitel beschrieben.

Grundlegend lassen sich die Komponenten der Implementierungen danach unterteilen, ob sie ersetzt werden können oder nicht. Zu der ersten Kategorie gehören die Komponenten im M aus N System, in Implementierung 1 beispielsweise die Funktionseinheiten. Zur zweiten Kategorie gehören die Komponenten außerhalb. in Implementierung 1 wären dies Kontrollpfad, Registerbank, die Schalter des M aus N Systems und so weiter. Im Folgenden werden nun zunächst die Komponenten ohne Selbstreparatur betrachtet. Daran schließt sich die Beschreibung der Komponenten innerhalb des M aus M Systems an und wie diese sich durch die Activity Migration ändert. Der letzte Teil fasst beide Formeln zusammen und legt so fest, wie im nächsten Kapitel die Ergebnisse bestimmt werden.

7.1. Komponenten ohne Möglichkeit zur Selbstreparatur

HotSpot liefert als Ergebnis einen Temperaturwert für jede Komponente des Systems. Damit lassen sich über die Formeln 2.13 bis 2.19 die Fehlerraten für die einzelnen Fehlereffekte und mit Formel 2.21 ihre Summe errechnen. Unter der Annahme, dass für Komponente K die Fehlerrate (λ_K) konstant ist, es sich bei dem Ausfallzeitpunkt T um eine exponentiell verteilte Zufallsvariable handelt und dass die Zeit t nicht negativ sein kann ($t \geq 0$), lautet die PDF wie folgt:

$$f_K(t) = \lambda_K e^{-\lambda_K t} \tag{7.1}$$

Die Zuverlässigkeit der Komponente K berechnet sich dann folgendermaßen:

$$R_K(t) = 1 - F_K(t) = e^{-\lambda_K t} \tag{7.2}$$

Fällt eine der S Komponenten außerhalb des M aus N Systems aus, ist der gesamte Prozessor nicht mehr funktionstüchtig. Somit bilden diese Komponenten in Bezug auf ihre Zuverlässigkeit ein seriell geschaltetes System (siehe Abschnitt 2.2). Wenn λ_i die Fehlerrate der Komponente i ($i = 1 \ldots S$), $R_i(t)$ ihre Zuverlässigkeit und $R_{1a1}(t)$ die des gesamten Systems ist, gilt:

$$R_{1a1}(t) = \prod_{i=1}^{S} R_i(t) = \prod_{i=1}^{S} e^{-\lambda_i t} \tag{7.3}$$

Diese Formel betrachtet die Anzahl der Komponenten und ihre jeweilige Fehlerrate. Zwischen der Implementierung mit und ohne AM ist es jedoch möglich, dass die Temperatur einer Komponente gleich bleibt, sich ihre Fläche aber deutlich ändert. Ein Beispiel wäre die Kontrolllogik MaNS-Ktrl / MaNS-AM-Ktrl. Dieser Unterschied ist in der Formel

bisher nicht modellierbar. Eine mögliche Lösung ist die in [KV11] eingesetzte Beschreibung über sogenannte Flächenäquivalente. Annahme ist eine diskrete Fläche A mit der entsprechenden Fehlerrate λ_A. Ihre Zuverlässigkeit berechnet sich wie folgt:

$$R_A(t) = e^{-\lambda_A t} \tag{7.4}$$

Jede Komponente des Systems besteht aus einer Anzahl C dieser Flächenäquivalente. Angenommen, alle Äquivalente besitzen innerhalb der Komponente die gleiche Fehlerrate λ_A, dann ist die Zuverlässigkeit $R_K(t)$ der Komponente K:

$$R_K(t) = R_A(t)^C = e^{-\lambda_A t C} \tag{7.5}$$

Da keine Informationen über die Größe der Flächenäquivalente, aber über die Anzahl der bei der Synthese verwendeten Technologiezellen vorliegt, sollen Flächenäquivalente und Technologiezellen gleichgesetzt werden. Das heißt, jede Zelle einer Komponente wird als kleinstmögliche Einheit angesehen, die von den über die Fehlerraten modellierten Fehlereffekten getroffen werden kann und hat damit die gleiche Fehlerrate, wie die anderen Zellen einer Komponente. Somit gilt: Jede der S in Serie geschalteten Komponenten i ($i = 1 \ldots S$) außerhalb des M aus N Systems besitzt eine Anzahl von C_i Zellen, die pro Komponente die gleiche Fehlerrate λ_i aufweisen. Damit lässt sich die gesamte Zuverlässigkeit dieser Komponenten $R_{1a1}(t)$ wie folgt ausdrücken:

$$R_{1a1}(t) = \prod_{i=1}^{S} R_i(t)^{C_i} = \prod_{i=1}^{S} e^{-\lambda_i t \cdot C_i} \tag{7.6}$$

oder, aufgrund der Additivität der Fehlerraten (siehe Abschnitt 2.2):

$$R_{1a1}(t) = e^{-\lambda_{1a1} t}, \qquad \lambda_{1a1} = \sum_{i=1}^{S} \lambda_i \cdot C_i \tag{7.7}$$

Die mittlere Lebensdauer ($MTTF_{1a1}$) dieser Komponenten kann laut Formel 2.8 wie folgt bestimmt werden:

$$MTTF_{1a1} = \frac{1}{\lambda_{1a1}} \tag{7.8}$$

7.2. Komponenten im M aus N System

Das M aus N System bildet im Prinzip die Obergruppe zu dem soeben angesprochenen seriellen und dem in Abschnitt 2.2 beschriebenen parallelen System, da es beide Fälle und alle Möglichkeiten dazwischen abbilden kann, je nachdem, wie groß M und N gesetzt werden. Das spiegelt sich auch in den Berechnungen zur Zuverlässigkeit und mittleren Lebensdauer wieder. Eine häufig eingesetzte Methode, um diese bestimmen zu können, verfährt über die Summe der exponentiell verteilten Ausfallzeitpunkte der Komponenten des Systems. Im Folgenden wird diese Grundlage zuerst erläutert. Daran schließt sich die Anwendung auf M aus N Systeme ohne und mit Activity Migration.

7.2.1. Summe exponentiell verteilter Zufallsvariablen

Angenommen, es existiert eine Komponente K mit exponentiell verteiltem Ausfallzeitpunkt X und Fehlerrate λ, dann gilt zum Zeitpunkt t ($t \geq 0$) für dessen PDF $f_K(t)$, Ausfallwahrscheinlichkeit $F_K(t)$ und Zuverlässigkeit $R_K(t)$ (siehe Abschnitt 2.2):

$$\begin{aligned} f_K(t) &= \lambda e^{-\lambda t}, \\ F_K(t) &= P(X < t) = 1 - e^{-\lambda t}, \\ R_K(t) &= P(X > t) = e^{-\lambda t} \end{aligned} \tag{7.9}$$

Um die Lebensdauer des Systems zu erhöhen, wird ihm eine identische Kopie L eingebaut. Diese wird aktiviert, sobald K ausfällt, und besitzt ab dann die gleiche Fehlerrate λ. Das System bleibt damit so lange funktionstüchtig, wie mindestens eine der beiden Komponenten funktioniert. Da beide unabhängig voneinander ausfallen, errechnet sich der Ausfallzeitpunkt T des Systems dann über die Summe der Ausfallzeitpunkte X und Y der Komponenten K und L:

$$T = X + Y \tag{7.10}$$

$f_X(u)$ ist die PDF der Komponente K in Abhängigkeit der Zeit u und $f_Y(t-u)$ ist die PDF der Komponente L in Abhängigkeit der darüber hinausgehenden Zeit $t-u$, wobei $0 \leq u \leq t$. Damit ergibt sich folgende PDF in Abhängigkeit der Zeit t:

$$\begin{aligned} f_{X+Y}(t) &= \int_0^t f_X(u) \cdot f_Y(t-u) du = \int_0^t \lambda e^{-\lambda u} \lambda e^{-\lambda (t-u)} du \\ &= \lambda^2 t e^{-\lambda t}, \end{aligned} \tag{7.11}$$

Allgemein lässt sich die Summe der Ausfallzeitpunkte von N Komponenten folgendermaßen ausdrücken:

$$f_N(t) = \frac{\lambda^N t^{N-1}}{(N-1)!} e^{-\lambda t} \tag{7.12}$$

Die Zuverlässigkeit $R_N(t)$ des Systems mit N Komponenten bestimmt sich dann wie folgt ([KZ02, S. 134]):

$$\begin{aligned} R_N(t) &= 1 - F(t) = 1 - \int_0^t \frac{\lambda^N t^{N-1}}{(N-1)!} e^{-\lambda t} \\ &= e^{-\lambda t} \sum_{i=0}^{N} \frac{(\lambda t)^i}{i!} \end{aligned} \tag{7.13}$$

und die MTTF:

$$MTTF_N = \int_0^\infty R_N(t) dt = \frac{N}{\lambda} \tag{7.14}$$

7.2.2. Zuverlässigkeit und MTTF des M aus N Systems ohne AM

Die Modellierung der Zuverlässigkeit basiert auf der soeben beschriebenen Summenformel, nur dass jetzt immer M Komponenten aktiv sind, welche alle die gleiche Fehlerrate λ_a besitzen. Bei der Zuweisung einer Fehlerrate für die Redundanz muss nun unterschieden werden. Bereits in Abschnitt 3.4.1 wurde angedeutet, dass es heiße und kalte Redundanz gibt. Laut [SP92] existieren in diesem Zusammenhang aber drei Fälle: Kalte Redundanz, bei der die Fehlerraten der inaktiven (*dormant*) Komponenten gleich Null sind ($\lambda_d = 0$), heiße Redundanz, bei der die Fehlerraten der inaktiven Komponenten identisch mit den Fehlerraten der aktiven Komponenten sind ($\lambda_d = \lambda_a$), und warme Redundanz, bei der die Fehlerraten zwischen Null und den Raten der aktiven Komponenten liegen ($0 \leq \lambda_d \leq \lambda_a$). Heiße und kalte Redundanz bilden also Sonderfälle der warmen. Im Folgenden nun die wichtigsten Eigenschaften dieser drei Formen und die Formeln zur Berechnung von Zuverlässigkeit und MTTF.

Kalte Redundanz Grundsätzlich sind in einem funktionierenden M aus N System immer genau M Komponenten aktiv. Fällt eine zum Zeitpunkt X_1 aus, wird sie sofort durch eine redundante Komponente ersetzt. Fällt eine weitere zum Zeitpunkt X_2 aus, wird auch diese ersetzt, solange, bis $N - M + 1$ Ausfälle stattgefunden haben, keine Redundanz mehr vorhanden ist und damit das gesamte System ausfällt. Somit ergibt sich die Lebensdauer T des Gesamtsystems aus der Summe der exponentiell verteilten Zufallsvariablen:

$$T = X_1 + X_2 + ... + X_{N-M+1} \tag{7.15}$$

Immer nach dem Ausfall einer Komponente des M aus N Systems wird ein Zeitpunkt X_i erreicht. Zwischen den Zeitpunkten X_{i-1} und X_i bilden die funktionstüchtigen Komponenten des M aus N Systems ein serielles Subsystem ([KZ02, S. 264]). Dadurch ist es möglich, die Fehlerraten λ_a der aktiven Komponenten wie in Abschnitt 2.2 zu $M \cdot \lambda_a$ zu addieren. Somit kann die PDF dieses Systems, unter der Bedingung, dass $\lambda_d = 0$ ist ($f_{NoAM|\lambda_d=0}(t)$), folgendermaßen beschrieben werden:

$$f_{NoAM|\lambda_d=0}(t) = \frac{M\lambda_a^N t^{N-1}}{(N-1)!} e^{-M\lambda_a t} \tag{7.16}$$

Daraus ergeben sich nach [KZ02, S. 265] folgende Zuverlässigkeit ($R_{NoAM|\lambda_d=0}(t)$) und mittlere Lebensdauer ($MTTF_{NoAM|\lambda_d=0}$)

$$\begin{aligned} R_{NoAM|\lambda_d=0}(t) &= \sum_{i=0}^{N-M} \frac{(M\lambda_a t)^i e^{-M\lambda_a t}}{i!} \\ MTTF_{NoAM|\lambda_d=0} &= \frac{N-M+1}{M\lambda} \end{aligned} \tag{7.17}$$

Heiße Redundanz In diesem Fall besitzen die redundanten Komponenten ab Anfang ($t = 0$) die gleiche Fehlerrate wie die aktiven Komponenten ($\lambda_d = \lambda_a = \lambda$) und können

damit bereits vor ihrem eigentlichen Einsatz ausfallen. Das spiegelt sich auch in der Formel zur Berechnung der Zuverlässigkeit des Systems $R_{NoAM|\lambda_d=\lambda_a}(t)$ wieder, da sie aktive und inaktive Komponenten gleichbehandelt. Kurz zusammengefasst werden dabei die bedingten Wahrscheinlichkeiten für alle Zustände des Systems summiert, bei denen noch mindestens M Komponenten funktionieren ([KZ02, S. 257]):

$$\begin{aligned} R_{NoAM|\lambda_d=\lambda_a}(t) &= \sum_{i=M}^{N} \binom{N}{i} R(t)^i F(t)^{N-i} \\ &= \sum_{i=M}^{N} \binom{N}{i} (e^{-\lambda t})^i (1-e^{-\lambda t})^{N-i} \end{aligned} \tag{7.18}$$

Für die mittlere Lebensdauer des M aus N Systems mit heißer Redundanz ($MTTF_S$) gilt dann ([KZ02, S. 257]):

$$MTTF_{NoAM|\lambda_d=\lambda_a} = \sum_{i=M}^{N} \frac{1}{i\lambda} \tag{7.19}$$

Warme Redundanz Bei der warmen Redundanz ist die Fehlerrate der inaktiven Komponenten (λ_d) kleiner als die der aktiven ($0 \leq \lambda_d \leq \lambda_a$), wodurch die Bestimmung der Zuverlässigkeit des Systems ($R_{NoAM|\lambda_d \leq \lambda_a}(t)$) deutlich an Komplexität gewinnt. In [SP92] wird dazu folgender geschlossener Ausdruck eingeführt:

$$\begin{aligned} R_{NoAM|\lambda_d \leq \lambda_a}(t) = &\frac{1}{(N-M)!\lambda_d^{N-M}} \sum_{i=0}^{N-M} (-1)^i \binom{N-M}{i} \\ &\cdot \left[\sum_{\substack{j=0 \\ j \neq i}}^{N-M} (M\lambda_a + j\lambda_d) \right] e^{-(M\lambda_a + i\lambda_d)t} \end{aligned} \tag{7.20}$$

Die MTTF wird in diesem Fall in [KZ02] nicht angegeben. Deswegen folgt nun die Herleitung.

Die MTTF des Systems ergibt sich aus der Summe der Erwartungswerte $E(X_i)$ zum Zeitpunkt i ([KZ02, S. 130]). Diese Erwartungswerte bestimmen sich wie folgt, wenn $f_i(t)$ die PDF des Systems zum Zeitpunkt i ist und X_i eine exponentiell verteilte Zufallsvariable mit Rate λ_i ([KZ02, S. 15]):

$$E(X_i) = \int_0^\infty t f_i(t) dt = \int_0^\infty t \lambda_i e^{-\lambda_i t} dt = \frac{1}{\lambda_i} \tag{7.21}$$

Da die Komponenten des Systems zwischen den Zeitpunkten X_{i-1} und X_i als serielles Subsystem aufgefasst werden können, ergibt sich die gesamte Fehlerrate aus der Summe der einzelnen Fehlerraten der Komponenten. Zwischen dem ersten Einsatz des Systems und dem ersten Ausfall einer Komponente, also $0 \leq t \leq X_1$, kann genau eine der N

Komponenten ausfallen, also eine der M aktiven mit Rate λ_a oder eine der $N-M$ inaktiven mit Rate λ_d. Zwischen dem Ausfall der ersten und der zweiten Komponente ($X_1 \leq t \leq X_2$) können noch $N-1$ Komponenten ausfallen und da immer M aktiv sind, haben dann wieder M die Rate λ_a und $N-M-1$ die Rate λ_d. Dies geht so weiter, bis nur noch M Komponenten aktiv und alle restlichen ausgefallen sind. Daraus ergibt sich für die mittlere Lebensdauer des Systems ($MTTF_{NoAM|\lambda_d \leq \lambda_a}$):

$$\begin{aligned} MTTF_{NoAM|\lambda_d \leq \lambda_a} &= E(X_1) + E(X_2) + ... + E(X_{N-M+1}) \\ &= \frac{1}{M\lambda_a + (N-M)\lambda_d} + \frac{1}{M\lambda_a + (N-M-1)\lambda_d} + ... + \frac{1}{M\lambda_a + 0\lambda_d} \\ &= \sum_{i=0}^{N-M} \frac{1}{M\lambda_a + i\lambda_d} \end{aligned} \tag{7.22}$$

Diese Summe stimmt bei angepassten Fehlerraten der inaktiven Komponenten (λ_d) mit der MTTF der heißen und kalten Redundanz überein.

7.2.3. Zuverlässigkeit und MTTF des M aus N Systems mit AM

Durch die AM werden die redundanten Komponenten nun nicht erst eingesetzt, sobald eine originale ausfällt, sondern tragen von Anfang an die Last mit. Dadurch ändert sich die Beschreibung der Zuverlässigkeit vom reinen M aus N System hin zum „Shared Load“ M aus N System, so wie es in [Sch88] eingeführt wird. Grundlage bildet wieder die Summe der exponentiell verteilten Ausfallzeitpunkte X_i.

Laut [AMP06] kann die Zuverlässigkeit des Systems wie folgt hergeleitet werden: Zum Zeitpunkt $t_0 = 0$, an dem das System zum ersten Mal eingesetzt wird, funktionieren alle Komponenten und teilen sich die für das System vorgesehene Last gleichmäßig. In diesem Fall hat jede Komponente die Fehlerrate λ_0 und da N Komponenten arbeiten, hat das System insgesamt eine Fehlerrate von $\alpha_1 = N \cdot \lambda_0$. Fällt eine Komponente zum Zeitpunkt X_1 aus, müssen die anderen ihre Last mit übernehmen (haben im vorliegenden Fall weniger Abkühlphasen) und damit eine Fehlerrate von λ_1. Das System erhält damit eine Fehlerrate von $\alpha_2 = (N-1) \cdot \lambda_1$. Fällt die Komponente i an X_i aus, besitzt jede der restlichen $N-i$ Komponenten eine Fehlerrate von λ_i ($0 \leq i \leq N-M$) und das gesamte System $\alpha_i = (N-i+1) \cdot \lambda_{i-1}$. Das System ist nicht mehr einsatzfähig, sobald $N-M+1$ Komponenten ausgefallen sind. Die Lebensdauer des Systems ergibt sich nun wieder aus der Summe der exponentiell verteilten Ausfallzeitpunkte mit den Parametern α_i:

$$T = X_1 + X_2 + ... + X_{N-M+1} \tag{7.23}$$

Nun müssen anhand der Größe der α_i laut [Sch88] folgende Fälle unterschieden werden:

Alle α_i sind gleich (also α) Für die Zuverlässigkeit des Systems $R_{AM|\alpha_i=\alpha_j}(t)$ gilt dann:

$$R_{AM|\alpha_i=\alpha_j}(t) = \sum_{i=0}^{N-M} \frac{(\alpha t)^i e^{-\alpha t}}{i!} \tag{7.24}$$

Dieser Fall tritt ein, wenn die Fehlerrate der überlebenden Komponenten direkt proportional zur Last ist, die sie tragen müssen.

Alle α_i sind ungleich Dies ist der allgemeine Fall, wenn die Fehlerrate nicht proportional zur Last ist. Die Zuverlässigkeit des Gesamtsystems $R_{AM|\alpha_i \neq \alpha_j}(t)$ wird dann wie folgt bestimmt:

$$\begin{aligned} R_{AM|\alpha_i \neq \alpha_j}(t) &= \sum_{i=1}^{N-M+1} W_i \cdot e^{-\alpha_i t} \\ W_i &\equiv \prod_{\substack{j=1 \\ j \neq i}}^{N-M+1} \frac{\alpha_j}{\alpha_j - \alpha_i}; \quad i = 1, ..., n-k+1 \end{aligned} \tag{7.25}$$

Alle α_i sind stufenweise gleich Genauer nehmen die α_i exakt a $(1 < a < N-M+1)$ verschiedene Werte $\beta_1, \beta_2, ..., \beta_a$ an. Durch Umbenennen der α_i gilt dabei, wenn r_a die Anzahl der α mit gleichem Wert ist:

$$\begin{aligned} \alpha_1 = ... = \alpha_{r_1} &= \beta_1 \\ \alpha_{r_1+1} = ... = \alpha_{r_1+r_2} &= \beta_2 \\ \vdots & \\ \alpha_{r_1+r_2+...+r_{a-1}+1} = ... = \alpha_{r_1+r_2+...+r_a} &= \beta_a, \end{aligned} \tag{7.26}$$

wenn folgende Annahmen getroffen werden:

1. $a \geq 1$ und eine ganze Zahl,

2. alle β_i sind ungleich,

3. die Summe der r_i ist gleich $N-M+1$ und

4. jedes $r_i \geq 1$ und eine ganze Zahl.

Damit ergibt sich die Zuverlässigkeit des Systems $R_{AM|\beta_i \neq \beta_j}(t)$ hier wie folgt, wobei $\text{poif}(k, \lambda)$ die CDF der Poisson Verteilung ist:

$$
\begin{aligned}
R_{AM|\beta_i \neq \beta_j}(t) &= B \sum_{j=1}^{a} \sum_{l=1}^{r_j} \frac{\Phi_{jl}(-\beta_j)}{(l-1)!\beta_j^{r_j-l+1}} \cdot \text{poif}(r_j - l; \beta_j \cdot t) \\
B &\equiv \prod_{j=1}^{a} (\beta_j)^{r_j} \\
\text{poif}(r_j - l; \beta_j \cdot t) &\equiv \sum_{i=0}^{r_j-l} \frac{e^{-\beta_j t}(\beta_j t)^i}{i!} \\
\Phi(t) &\equiv (-1)^{(l-1)}(l-1)! \sum_{\substack{m_j=0; m_1,\ldots,m_a \geq 0 \\ m_1+\ldots+m_a=l-1}} \prod_{\substack{i=1 \\ i \neq j}}^{a} v(i, m_i, t) \\
v(i, m_i, t) &\equiv \binom{r_i + m_i - 1}{m_i} (\beta_i + t)^{-(r_i+m_i)}
\end{aligned}
\tag{7.27}
$$

Dieser Fall kann beispielsweise eingesetzt werden, um zusätzliche redundante Komponenten zu beschreiben, die nicht von Anfang an die Last mit tragen, sondern erst bei Ausfall einer anderen aktiv werden.

Um die Zunahme der Fehlerraten bei Ausfall der Komponenten zu formalisieren, wird in [BS89] eingeführt und in [Mad93] generalisiert, Folgendes beschrieben: Am Anfang teilen sich alle Komponenten die Last und haben damit eine Fehlerrate von $\alpha_1 = N \cdot \lambda_0$. Fällt eine Komponente aus, teilen sich $N-1$ Komponenten die Last und das System besitzt eine Fehlerrate von $\alpha_2 = (N-1) \cdot \lambda_1$. Dabei ist λ_1 typischerweise größer als λ_0 und wird über einen Exponenten δ ermittelt:

$$
\lambda_1 = \lambda_0 \cdot \left(\frac{N}{N-1} \right)^{\delta} \tag{7.28}
$$

Damit gilt allgemein für α_i $(1 \leq i \leq N - M + 1)$:

$$
\alpha_i = (N - i + 1) \cdot \lambda_0 \cdot \left(\frac{N}{N-i+1} \right)^{\delta} \tag{7.29}
$$

Mit $\delta = 1$ kann der Fall erzeugt werden, dass alle α_i gleich sind, und mit $\delta \neq 1$ der Fall, dass sich alle unterscheiden. Die mittlere Lebensdauer des Systems ($MTTF_{AM}$) wird wieder über die Summe der Erwartungswerte ([KZ02, S. 259]) bestimmt:

$$
MTTF_{AM} = \sum_{i=1}^{N-M+1} \frac{1}{\alpha_i} \tag{7.30}
$$

7.3. Zusammenführung der beiden Modellierungen

Bisher wurden in diesem Kapitel die Formeln zur Berechnung der Komponenten außerhalb das M aus N Systems (R_{1a1}) und innerhalb des M aus N Systems, jeweils ohne Activity

Migration (R_{NoAM}) und mit Activity Migration (R_{AM}), vorgestellt. Der letzte Abschnitt beschreibt nun, wie mit ihrer Hilfe Ergebnisse erzeugt werden, um damit den Nutzen in Bezug auf eine Lebensdaueroptimierung der Activity Migration in M aus N Systemen bestimmen zu können.

7.3.1. Zuverlässigkeit und MTTF der Implementierungen ohne AM

Nach Simulation und Synthese liegen die für die Betrachtungen notwendigen Flächen und Temperaturen und mit Hilfe der Formel 2.21 auch die Fehlerraten jeder Komponente der Implementierungen vor. Diese werden zuerst danach gruppiert, ob sie zu den aktiven oder passiven Komponenten des M aus N Systems gehören oder außerhalb davon liegen. Anschließend werden die Fehlerraten für die jeweiligen Subsysteme erstellt. Das heißt, ein λ_{1a1} für die Komponenten außerhalb und jeweils ein λ_a für die aktiven und ein λ_d für die passiven Komponenten des M aus N Systems. λ_{1a1} kann nach Formel 7.7 erstellt werden. Für die Bestimmung von λ_a und λ_d wird, wie im Folgenden beschrieben, vorgegangen.

Aus den fünf Implementierungen ist bekannt, dass die austauschbaren Komponenten aus mehreren Baugruppen bestehen können (z.B. Pipelineregister und FE in Implementierung 2). Fällt eine dieser Baugruppen aus, muss die gesamte Komponente ausgewechselt werden. Damit bilden diese Baugruppen wieder ein serielles System und können wie die Komponenten außerhalb des M aus N Systems in Abschnitt 7.1 beschrieben werden. Durch diese Formulierung ist es auch erneut möglich, den Einfluss der Fläche der Baugruppen über die Anzahl der Technologiezellen mit in die Fehlerrate einfließen zu lassen. Das heißt, es existiert eine Menge von M Gruppen von aktiven Komponenten, die jeweils aus B Baugruppen bestehen, von denen jede eine Fehlerrate $\lambda_{j,i}$ und eine Anzahl von $C_{j,i}$ Technologiezellen besitzt ($i = 1 \ldots B$, $j = 1 \ldots M$). Um λ_a zu erhalten, wird das arithmetische Mittel der nach Formel 7.7 bestimmten Summen dieser Gruppen von Fehlerraten gebildet.

$$\lambda_a = mean(\sum_{i=1}^{B} \lambda_{1,i} \cdot C_{1,i}, \sum_{i=1}^{B} \lambda_{2,i} \cdot C_{2,i}, \ \ldots\ , \sum_{i=1}^{B} \lambda_{M,i} \cdot C_{M,i}) \tag{7.31}$$

Das Gleiche gilt für die $N - M = R$ Gruppen von passiven Komponenten:

$$\lambda_d = mean(\sum_{i=1}^{B} \lambda_{1,i} \cdot C_{1,i}, \sum_{i=1}^{B} \lambda_{2,i} \cdot C_{2,i}, \ \ldots\ , \sum_{i=1}^{B} \lambda_{R,i} \cdot C_{R,i}) \tag{7.32}$$

Da diese nicht von der Versorgungsspannung getrennt sind und einen gewissen Temperaturwert besitzen, der wahrscheinlich unter dem der aktiven Komponenten liegt, kann davon ausgegangen werden, dass es sich hier um warme Redundanz handelt. Somit wird über Formel 7.20 die Zuverlässigkeit $R_{NoAM|\lambda_d \leq \lambda_a}(t)$ des M aus N Systems bestimmt.

Die Zuverlässigkeit der gesamten Implementierung $R_{Sys|NoAM}$ ergibt sich dann aus der seriellen Schaltung der Komponenten außerhalb und innerhalb des M aus N Systems:

$$R_{Sys|NoAM}(t) = R_{1a1}(t) \cdot R_{NoAM|\lambda_d \leq \lambda_a}(t) \tag{7.33}$$

Laut Formel 2.8, errechnet sich die MTTF zweier unabhängiger Komponenten K und L bei konstanten Fehlerrate λ_K und λ_L und exponentieller Verteilung des Ausfallzeitpunkts wie folgt:

$$MTTF_K = \frac{1}{\lambda_K}, \qquad MTTF_L = \frac{1}{\lambda_L}, \tag{7.34}$$

Werden beide Komponenten seriell geschaltet, ist die mittlere Lebensdauer $MTTF_G$ des Gesamtsystems über die Summe der Fehlerraten folgendermaßen zu ermitteln:

$$MTTF_G = \frac{1}{\lambda_K + \lambda_L} \tag{7.35}$$

Das Umstellen und Einsetzen der Formeln 7.34 in Formel 7.35 ergibt dann:

$$MTTF_G = \frac{1}{\frac{1}{MTTF_K} + \frac{1}{MTTF_L}} \quad \text{oder} \quad MTTF_G = \frac{MTTF_K \cdot MTTF_L}{MTTF_K + MTTF_L} \tag{7.36}$$

Die vorliegenden Implementierungen sind nicht mehr funktionsfähig, sobald eine der Komponenten außerhalb des M aus N Systems ausgefallen ist oder $N - M + 1$ Komponenten innerhalb. Das bedeutet, dass sie als serielles System modelliert werden können, woraus sich für die mittlere Lebensdauer $MTTF_{Sys|NoAM}$ der Implementierungen ohne AM folgendes ergibt:

$$MTTF_{Sys|NoAM} = \frac{MTTF_{1a1} \cdot MTTF_{NoAM|\lambda_d \leq \lambda_a}}{MTTF_{1a1} + MTTF_{NoAM|\lambda_d \leq \lambda_a}} \tag{7.37}$$

7.3.2. Zuverlässigkeit und MTTF der Implementierungen mit AM

Für die Bestimmung der Zuverlässigkeit der Implementierungen mit AM wird die gesamte Fehlerrate der Komponenten außerhalb des M aus N Systems, λ_{1a1} und die in Abschnitt 7.2.3 beschriebenen α_i benötigt. λ_{1a1} ist dabei analog zu den Implementierungen ohne AM. Weiterhin werden jedoch die α_i benötigt. Diese können entweder über die verschiedenen Ausfallraten λ_0 bis λ_{N-M} bestimmt werden, die aufgrund der Lastzunahme bei ausfallenden Komponenten auftreten, oder über Formel 7.29, mit λ_0 und δ als Parameter. Da die λ_i der ersten Lösung aber mit vielen aufwändigen Simulationen verbunden wären, soll der zweite Weg verfolgt werden.

Hierzu werden die Anzahl der Technologiezellen und die Fehlerraten aller Komponenten innerhalb des M aus N Systems benötigt, die sich, da noch keine Komponente ausgefallen ist, alle die gleiche Last teilen. Diese werden in N Mengen von B Baugruppen innerhalb der austauschbaren Komponenten, mit der jeweiligen Anzahl von Technologiezellen $C_{j,i}$, Temperatur und damit Fehlerrate $\lambda_{j,i}$ ($i = 1 \ldots B$, $j = 1 \ldots N$), unterteilt. Wie bereits in Abschnitt 7.2.3 beschrieben, ergibt sich die erste Fehlerrate α_1 des Gesamtsystems aus $\alpha_1 = N \cdot \lambda_0$. Das λ_0 ist die Fehlerrate der einzelnen Komponenten und kann mit dem arithmetischen Mittel der Summen der Fehlerraten der Gruppen, multipliziert mit der jeweiligen Anzahl der Zellen, bestimmt werden:

$$\lambda_0 = mean(\sum_{i=1}^{B} \lambda_{1,i} \cdot C_{1,i}, \sum_{i=1}^{B} \lambda_{2,i} \cdot C_{2,i}, \ \ldots\ , \sum_{i=1}^{B} \lambda_{N,i} \cdot C_{N,i}) \tag{7.38}$$

Der fehlende Parameter δ liegt nicht vor und muss aus diesem Grund ermittelt werden. Hierzu findet das λ_a aus der Simulation ohne AM Einsatz. Dieses bildet im Prinzip das zweite Extrem. Während λ_0 die Fehlerrate der Komponenten darstellt, wenn alle N funktionieren, steht λ_a für die Last, wenn nur noch M arbeiten, $N-M$ bereits ersetzt wurden und das System damit beim nächsten Fehler ausfallen würde. Mit einem 4 aus 8 System als Beispiel kann δ wie folgt errechnet werden:

$$\begin{aligned} &\alpha_1 = 8 \cdot \lambda_0, \quad \alpha_5 = 4 \cdot \lambda_d \\ &\alpha_5 = (8-4) \cdot \lambda_0 \cdot \left(\frac{8}{8-4}\right)^{\delta} = 4 \cdot \lambda_0 \cdot 2^{\delta} \\ &\quad \Longrightarrow \delta = log_2\left(\frac{\lambda_d}{\lambda_0}\right) \end{aligned} \tag{7.39}$$

Mit Hilfe dieses δ werden anschließend nach Formel 7.29 die α_i bestimmt. Spätere Versuche haben gezeigt, dass diese sich nicht gleichen. Das heißt, dass für die Bestimmung der Zuverlässigkeit der Fall $\alpha_i = \alpha_j = \alpha$ entfällt. Weiterhin sind keine zusätzlichen Ersatzelemente vorgesehen, wonach der Fall $\beta_i \neq \beta_j$ entfällt. Somit wird hier die Zuverlässigkeit über Formel 7.25 ($R_{AM|\alpha_i \neq \alpha_j}(t)$) bestimmt.

Da die Komponenten außerhalb zum M aus N System wieder seriell geschaltet sind, gilt für die gesamte Zuverlässigkeit $R_{Sys|AM}$ der Implementierung:

$$R_{Sys|AM} = R_{1a1} \cdot R_{AM|\alpha_i \neq \alpha_j}(t) \tag{7.40}$$

und für die MTTF:

$$MTTF_{Sys|AM} = \frac{MTTF_{1a1} \cdot MTTF_{AM}}{MTTF_{1a1} + MTTF_{AM}} \tag{7.41}$$

Kapitel 8

Ergebnisse

Die vorgestellte Verbindung von Selbstreparatur und Thermal Management verspricht den Vorteil, die Zuverlässigkeit und mittlere Lebensdauer eines Systems mit relativ geringem Aufwand zu steigern. Dies basiert einerseits auf dem Hinauszögern der Entstehung von Defekten durch Senkung der Temperatur im System und andererseits durch Tolerieren von einmal aufgetretenen Fehlern mittels Rekonfiguration. Das nun folgende Kapitel prüft diese Behauptung anhand ermittelter Ergebnisse. Dazu gehört als Erstes die Auflistung der Kosten des Systems, was den zusätzlichen Soft- und Hardwareaufwand umfasst. Daran schließt sich die Beschreibung der positiven Auswirkungen des Systems an, die über eine Temperatursenkung angegeben werden. Im dritten Teil folgt schließlich die Auswertung dieser Resultate und die Bestimmung ihrer Auswirkung auf Zuverlässigkeit und mittlere Lebensdauer.

8.1. Kosten von M aus N System und Activity Migration

Dieser Abschnitt beschreibt den zusätzlichen Aufwand, der betrieben werden muss, um die Activity Migration in ein M aus N System zu integrieren. Dies kann in software- und hardwareseitigen Aufwand unterteilt werden.

8.1.1. Softwareaufwand

Die notwendigen Anpassungen seitens der Software sind, wie bereits in Kapitel 5.3.1 beschrieben, das Load Balancing auf Ebene der Slots (Operationen) oder Cluster. Da für die vorliegende Arbeit aber Programmcode generiert wird, der bereits ideal ausbalanciert ist, kann der somit entstehende Aufwand nicht ermittelt werden.

8.1.2. Hardwareaufwand

Der zusätzliche Aufwand in Bezug auf die Hardware entsteht in Implementierung 1 ausschließlich durch Erweiterung der Kontrolllogik (siehe Abb. 5.1(b)) um die Buckets und die Anpassungen in der Dekodierlogik (siehe Abb. 5.6). Da in den Implementierungen 2 - 5 die Pipelineregister ausgetauscht werden, ist es hier notwendig, die Schalter zeitlich versetzt anzusteuern. Somit sind einerseits wieder der Einsatz der Buckets und andererseits weitere Anpassungen in der Dekodierlogik für die Schalter vorzunehmen (Abb. 5.12). In Implementierung 5 findet weiterhin das Kopiernetzwerk Copy Einsatz, das die Registerinhalte zur jeweils im nächsten AM-Takt zu aktivierenden Registerbank verschiebt (Abb.

5.12).

Um genaue Zahlen über den zusätzlichen Aufwand in Hardware nennen zu können, wurden alle fünf Implementierungen mit dem Cadence RTL Compiler und der 45nm Nangate Open Cell Library [Nan13] mit einer Taktfrequenz von 500 MHz und einer Datenbreite von 32 Bit synthetisiert. Ausgelegt sind sie als 4 aus 8 System, das heißt:

- Implementierung 1 mit vier originalen und vier redundanten Funktionseinheiten,
- Implementierung 2 mit vier originalen und vier redundanten FR, DR, FE und WR,
- Implementierung 3 mit vier originalen und vier redundanten Slots,
- Implementierung 4 mit vier originalen und vier redundanten „Teilclustern“, die aus jeweils vier FR, DR, FE und WR und einmal MAR und MBR bestehen und
- Implementierung 5 mit vier originalen und vier redundanten Clustern.

Die Ergebnisse sind in Anhang D gegeben. Jede Tabelle enthält dabei Aussagen über die Größe der drei Versionen in Anzahl von Zellen: des originalen Systems (BS - 1. Spalte), des Systems mit reinem M aus N System (MaNS - 2. Spalte) und der um die AM erweiterten Version (MaNSAM - 3. Spalte). Kleinere Abweichungen in gleichartigen Komponenten entstehen dabei in der Synthese, da sie beispielsweise durch Optimierungen nicht immer exakt gleiche Ergebnisse liefert. Auf Nennung von Komponenten, die auch im originalen System auftreten, wurde aus Platzgründen verzichtet.

Wie sich erkennen lässt, entsteht der größte Teil des Aufwandes meist durch den Einsatz des M aus N Systems selbst, also durch die redundanten Komponenten und Schalter. Der Aufwand für das Hinzufügen der AM ist hingegen vergleichsweise gering. Dieser beträgt in den Implementierungen 1 - 4 maximal $0,43\%$ im Vergleich zum MaNS. Erst Implementierung 5 wird durch die Copy Komponenten deutlich aufwändiger ($+45,45\%$).

Interessant ist aber nicht nur, wie groß der Aufwand für eine spezielle Version des M aus N Systems ist, sondern auch, wie dieser mit den Variablen M und N wächst. Hierzu wurde Implementierung 2 ohne und mit AM in unterschiedlichen Versionen, aber ansonsten mit den gleichen Parametern wie oben, synthetisiert und die Tabellen D.6 und D.7 erstellt. Die erste Spalte betitelt dabei die entsprechende Komponente, die zweite die Anzahl der originalen Komponenten und die restlichen unterschiedliche Anzahlen der redundanten Komponenten. Die Größenangaben sind wieder in Anzahl von Zellen.

Um das Wachstum in Abhängigkeit von der Anzahl der M aktiven und $R = N - M$ passiven Komponenten des M aus N Systems besser ablesen zu können, wurden die Diagramme in Abbildung 8.1 erstellt. Die Diagramme oben links und oben in der Mitte geben dabei die Größe der Schalter in das MaNS (links) und aus dem MaNS (mitte) in Abhängigkeit von $M = 1...4$ und $R = 1...4$ mit und ohne AM an. Hier lässt sich gut erkennen, dass die Schalter in fast allen Fällen gleich groß sind und ihre Größe in Abhängigkeit von M und R gleich ansteigt. Kleinere Abweichungen entstehen wieder aufgrund der Synthese. Die Dekodierlogik (unten links) ist mit der AM in allen Fällen etwas größer. Dies liegt daran, dass der zeitliche Versatz der Schalter mit implementiert werden muss, um keine Abhängigkeiten in der Pipeline zu verletzten. Ansonsten scheint das Wachstum in beiden Fällen

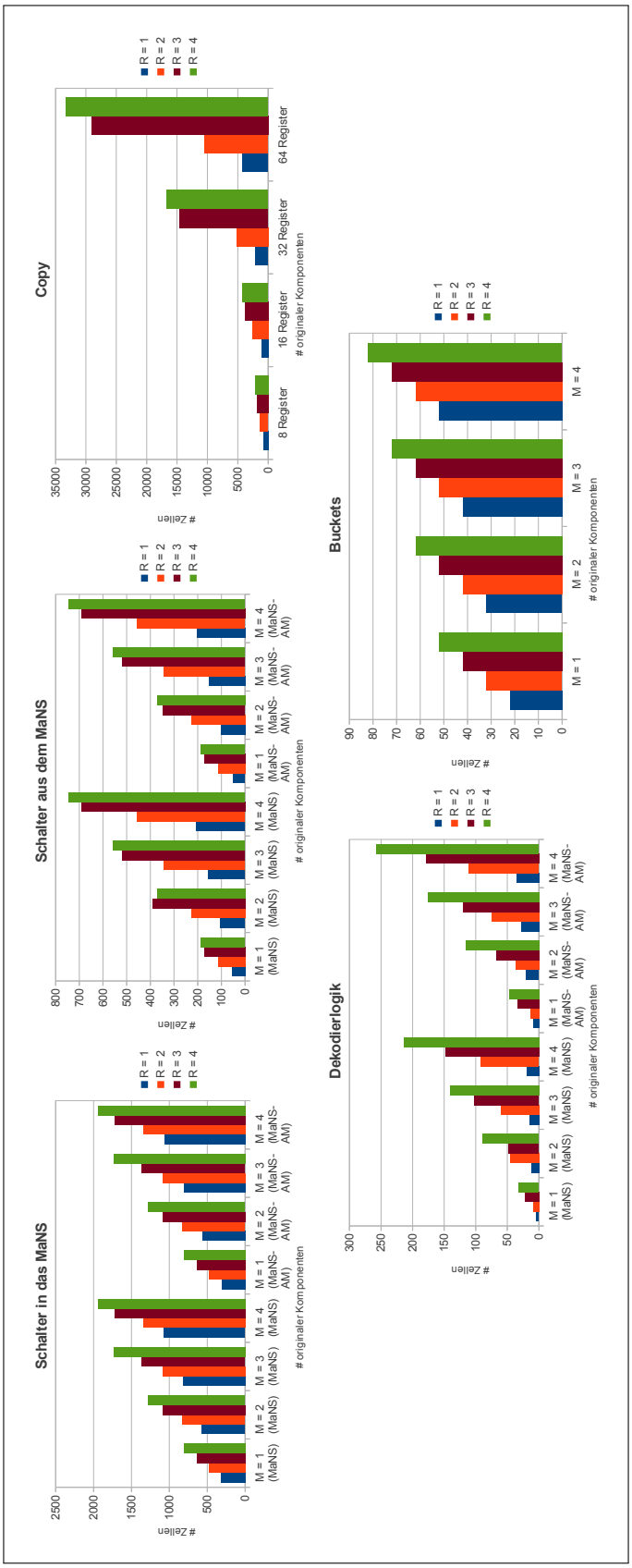


Abbildung 8.1.: Wachstum der administrativen Baugruppen

und Richtungen (M und R) in etwa linear. Lediglich die Größe und dessen Wachstum unterschiedlicher Versionen der Buckets (unten rechts) können direkt vorhergesagt werden, da sie immer genau $10 \cdot (M+R)+2$ Zellen beträgt.

Die Hardware für die Kopierfunktion der Registerbänke Copy trägt, wie bereits angedeutet, deutlich zum Aufwand in Implementierung 5 bei. Aus diesem Grund wurde sie noch einmal als alleinstehende Komponente synthetisiert und die Ergebnisse in Tabelle D.8 dargestellt. Da diese Funktion im Wesentlichen nur von der Anzahl der redundanten Komponenten und der zu kopierenden Register abhängt, wurde auf weitere Untersuchungen zur Größe von M verzichtet. Die erste Spalte der Tabelle gibt die Größe des M aus N Systems an und die restlichen beinhalten die Anzahl der Zellen in Abhängigkeit der zu kopierenden Register. In Abbildung 8.1 oben rechts ist dies auch noch einmal grafisch dargestellt. Das Wachstum von Copy scheint sich dabei mit steigender Anzahl der Register exponentiell zu beschleunigen, während eine zunehmende Anzahl von Redundanz das Wachstum langsamer ansteigen lässt. Durch die kleine Anzahl von Versuchsreihen sind diese Trends aber nur grobe Schätzungen.

8.2. Auswirkungen auf die Temperatur im System

Wie Kapitel 6 erahnen lässt, sind die Temperatursimulationen ein komplexer Vorgang mit vielen beeinflussenden Parametern. Im ersten Teil dieses Abschnittes werden deshalb die eingesetzten Parameter beschrieben und ein Problem und dessen Lösungsweg erläutert, das sich aufgrund der Fläche der Systeme und den Modellen in HotSpot ergeben hat. Danach folgt die Auflistung ermittelter Ergebnisse und deren Auswertung.

8.2.1. Gesetzte Parameter und Modellierung als SOC

Während der Vorbereitungen zu den Temperatursimulationen wurde festgestellt, dass die Implementierungen mit einer Datenbreite von 32 Bit und synthetisiert in der 45nm Technologie flächen- und leistungsmäßig zu klein sind, um ein aussagekräftiges Maß an Wärme zu erzeugen ($< 2^{\circ}C$). Aus diesem Grund mussten verschiedene Maßnahmen getroffen werden, um realistische Ergebnisse zu liefern. Diese lassen sich wie folgt zusammenfassen:

- **Erhöhung der Datenbreite:**
  Um die Fläche des Prozessors an sich zu erhöhen, wurde dieser für die Temperatursimulationen von 32 auf 128 Bit Datenbreite vergrößert. Dies verbessert die Simulierbarkeit, ändert aber nichts am relativen Verhältnis beim Zusatzaufwand zwischen Originalsystem, M aus N System und M aus N System mit Activity Migration. Auch das Verhältnis von Leistung zu Fläche bleibt gleich.

- **Einsatz unterschiedlicher Technologien:**
  Strukturen in der 45nm Technologie sind sehr klein und ihre umgesetzte Leistung gering. Erst in Summe, da viel mehr Komponenten auf gleicher Fläche untergebracht werden können, ergibt sich eine Erhöhung der Leistung pro Fläche und damit die verstärkte Wärmeerzeugung im Vergleich zu anderen Technologien. Um die

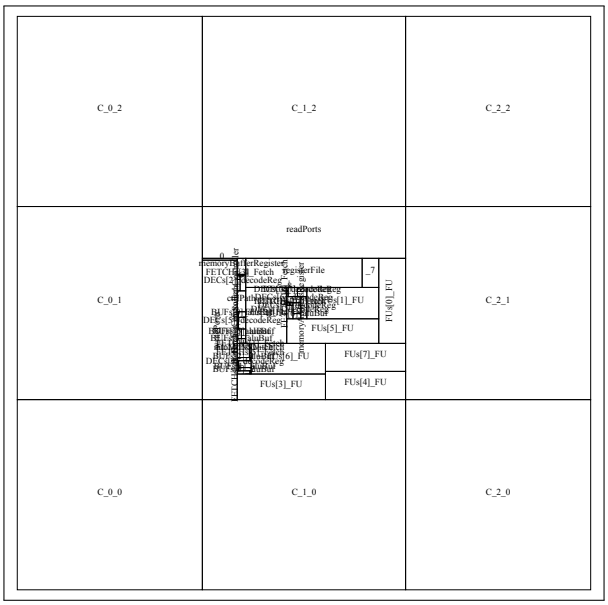


Abbildung 8.2.: Der Prozessor als zentrale Komponente eines SOC

Fläche steigern zu können und realistischere Ergebnisse zu erzeugen, wurde die Entscheidung getroffen, mehrere Technologien zu untersuchen:

- **45nm** als „high-end“ Technologie,
- **130nm** als „zuverlässige“ Generation und
- **250nm** als etwas ältere, aber flächenmäßig größte Generation.

So kann gleichzeitig die Fläche erhöht und untersucht werden, ob sich die Auswirkungen der AM in unterschiedlichen Generationen und damit unterschiedlichem Verhältnis von Leistung zu Fläche verändern.
Die Bibliotheken der 130 und 250nm Technologie wurden freundlicherweise von der IHP GmbH in Frankfurt (Oder) zur Verfügung gestellt.

- **Modellierung als SOC:**
  Während des schriftlichen Austausches mit den Machern von HotSpot wurde festgestellt, dass die Modelle des Programms auf eine Kantenlänge des zu untersuchenden Systems von 1,3cm ausgelegt sind. Zu diesem Zeitpunkt hatte aber trotz 128 Bit und verschiedenen Technologien kein System diese Größe. Daher wurde in Absprache mit dem HotSpot Team die Entscheidung getroffen, die Implementierungen als Teil eines System on Chip (SOC) zu modellieren. Wie dies geschah, ist in Abbildung 8.2 dargestellt. Das eigentliche System befindet sich dabei immer in der Mitte. Umgeben wird es zur Vergrößerung der Fläche von acht virtuellen Komponenten. Diese sind immer so modelliert, dass das Gesamtsystem quadratisch ist und insgesamt eine Kantenlänge von 1,3cm erhält. Das Vorhandensein dieser Komponenten ist dabei nicht unrealistisch. So könnten sie zum Beispiel den Daten- oder Programmspeicher,

Caches oder andere Kerne repräsentieren. Um einen Mittelweg zwischen angemessener Erwärmung von außen und dem System als heißeste Komponente in der Mitte einzuschlagen, wird den umgebenden Komponenten das 0,75-fache Leistung zu Fläche Verhältnis des Prozessorkerns zugewiesen. Das heißt, wenn A_V und P_V die gesamte Fläche und Leistung des VLIW Prozessors in der Mitte sind und A_C die Fläche einer der umgebenden Komponenten ist, ergibt sich folgende Formel für ihre Leistung:

$$P_C = 3/4 \cdot \frac{A_C \cdot P_V}{A_V} \tag{8.1}$$

- **Angepasster Wärmeleitwiderstand:**
  Weiterhin wurde in Absprache mit dem HotSpot Team der Wärmeleitwiderstand des heat sink (r_convec) von 0,1 auf 0,8 angepasst. Dies ist damit immernoch ein durchaus realistischer Parameter, sorgt aber dafür, dass sich das System ein wenig stärker erwärmt, .

Trotz dieser Anpassungen hat sich der in Kapitel 6 beschriebene Ablauf nur wenig verändert. Die Synthese, Simulation und Ermittlung der Leistung sind in allen Fällen gleich, außer, dass Systeme mit größeren Technologien langsamere Taktraten erhalten. Der generierte Programmcode bleibt in allen Fällen identisch. Bei der Arbeit mit HotSpot musste jedoch für praktisch jedes System die .cfg Datei angepasst werden. Dies umfasste folgende Werte:

- Die Gewichte der Fläche beim Floorplanning. Diese wurden, je nach Technologie, auf folgende Werte gesetzt:
  - **45nm:** $\lambda A = 1.0e+09$
  - **130nm:** $\lambda A = 1.0e+08$
  - **250nm:** $\lambda A = 1.0e+07$

- Das Sampling Intervall (sampling_intvl) bleibt immer bei $1e^{-6}$, damit sich bei einer Programmlänge von 10000 Zeilen unabhängig von der Taktfrequenz des Systems eine Simulationsdauer von 10ms ergibt. Dies hat sich als ausreichende Zeitspanne erwiesen, damit die Systeme eine stabile Temperatur erreichen.

- Die Anzahl der Zeilen (grid_rows) und Spalten (grid_cols) des Gitters im „Grid Modus“ wurde von 64 auf jeweils 256 Zeilen und Spalten erhöht, um eine genauere Auflösung bei den kleinen Komponenten zu erhalten. Damit dies auch bei der abschließenden Erzeugung der .svg Datei beachtet wird, war der Befehl wie in Quelltext 8.1 anzupassen.

- Die Taktfrequenz (base_proc_freq) musste in jedem Fall extra eingetragen werden, so dass sie für alle Implementierung korrekt ist. Entsprechende Werte sind im nächsten Abschnitt einzeln aufgelistet.

```
1       perl grid_thermal_map.pl vliw.flp vliw.grid.steady 256 256
        > vliw.svg
```

Quelltext 8.1: Thermal Map mit mehr Zeilen und Spalten

8.2.2. Resultate der Simulationen

Eine durch die AM im M aus N System verursachte Temperatursenkung gegenüber dem Prozessor ohne redundante Komponenten entsteht durch das Zusammenspiel von drei Faktoren. Erstens werden durch das Hinzufügen der Redundanz und die hierfür notwendigen Schalter die Pfade im System länger. Dies macht es notwendig, die Taktrate zu senken. Da aber das Drosseln der Taktrate, wie in Abschnitt 3.2.2 unter *Frequency Selection* beschrieben, eine häufig angewandte Methode des Thermal Management ist, muss dieser Effekt einzeln betrachtet werden. Zweitens kommen durch das M aus N System redundante Komponenten hinzu. Dabei kann es sich, je nach Art, um sehr große Flächen mit geringer Leistung handeln. Dies verändert das Verhältnis von Leistung zu Fläche des Prozessors und wird zu einer Senkung der Temperatur führen. Drittens durch die AM selbst, die zusätzlich die mittlere und Spitzentemperatur des Prozessors senken kann. Um den Einfluss dieser Punkte unterscheiden zu können, müssen neben den Prozessoren mit M aus N System und mit oder ohne AM auch das ursprüngliche Prozessormodell auf höchstmöglicher Taktrate und das ursprüngliche Modell mit gesenkter Taktrate, entsprechend dem M aus N System, simuliert werden.

Bei der Durchführung der Temperatursimulationen wurden die fünf Implementierungen und die zwei originalen Versionen zuerst mit den drei Technologiebibliotheken synthetisiert. Anschließend folgte die Simulation des generierten Programmcodes mit ModelSim, wobei die AM in einer Version des Programms aller zehn Takte aktiviert wurde und in der anderen nicht. Danach wurde die Leistung bestimmt, die, gesteuert durch die Programme, in den Prozessoren umgesetzt wird. Zum Schluss fand die Generierung der Floorplans und die Temperatursimualtion mit HotSpot statt. Hier wurden alle Parameter, bis auf die Gewichte beim Floorplanning und die Taktfrequenzen, immer gleich gehalten. Die umgebenden Komponenten bei der Modellierung als SOC erhielten dabei das 0,75-fache Verhältnis von Leistung zu Fläche, wie das originale System auf höchster Taktrate. Dies sorgt dafür, dass alle Systeme gleich erwärmt werden und Unterschiede nur durch die oben genannten Faktoren entstehen können.

Um Simulationszeit zu sparen, wurde in den Fällen ohne und mit AM jeweils die gleiche Hardware, mit eingebauter AM, verwendet. Dies ist möglich, da sich die Fälle im Hardwareaufwand um maximal 0,5% unterscheiden und die Differenzen in Taktrate, Hitzeerzeugung usw. damit vernachlässigbar klein sind. Lediglich in Implementierung 5 musste für die Simulation ohne AM die Kopierhardware entfernt und ein neuer Floorplan erstellt werden, da diese im Vergleich zum Rest des Systems sehr groß ist und damit das Temperaturverhalten beeinflusst. Die Taktrate konnte jedoch beibehalten werden, da Copy nicht im kritischen Pfad liegt.

Tabelle 8.1.: Die Periodendauer eines Taktes der Implementierungen in ps

	45nm	130nm	250nm
Impl 1	1200	2650	6250
Impl 2	1100	2300	6400
Impl 3	1100	2300	6400
Impl 4	1150	2450	6000
Impl 5	1150	2450	6000
B	1050	2250	5500
B-Slow	1200	2650	6400
C	1100	2350	5800
C-Slow	1150	2450	6000

Die Resultate der Simulationen sind in den Graphen in den Abbildung 8.3 und 8.4 dargestellt. Diese sind zuerst zeilenweise in die drei Technologien 45nm, 130nm und 250nm unterteilt. In jeder Zeile befinden sich wiederum 4 bzw. 3 Graphen, von denen jeder, wie an der Beschriftung ersichtlich, für ein unterschiedliches System steht. Die Graphen geben die jeweilige Systemtemperatur auf der Ordinate in °C an und sind auf der Abszisse in vier Teilsysteme untergliedert. Für die Darstellung der Werte wurden hier sogenannte „Fehlerbalken“ eingesetzt, die oft die fehlerhafte Abweichung von einem bestimmten Mittelwert beschreiben. Sie sind wie folgt zu interpretieren: Das Quadrat in der Mitte gibt das arithmetische Mittel aller Komponenten an. Die jeweilige obere und untere Grenze werden durch die heißeste bzw. kälteste Komponente im System bestimmt. Auf die Angabe der Werte der Komponenten, die innerhalb dieses Bereichs liegen, wurde aus Gründen der Übersichtlichkeit verzichtet. Bei den Kategorien auf der Abszisse gibt das erste Zeichen das jeweilige System an: 1-5 für die Implementierung 1-5, B für das Basissystem mit einem Cluster und C für das Basissystem mit vier Clustern. Weiterhin sind sie danach unterteilt, ob kein M aus N System vorliegt (NoMoN), ob die Taktrate gesenkt wurde (Slow) oder ob die AM ausgeführt wird (*AM*) oder nicht (*NoAM*). Der Ausdruck nach dem letzten Unterstrich beschreibt, welche Komponenten im jeweiligen Fall betrachtet wurden. Dies sind einerseits alle Komponenten im System (*All*) und andererseits ausschließlich die austauschbaren Komponenten im M aus N System (*MaN*). Bei den Systemen ohne M aus N System sind dies entweder die Funktionseinheiten bei B oder die Cluster bei C. Tabelle 8.1 gibt die Periodendauer eines Taktes in *ps* an, mit denen die jeweiligen Systeme synthetisiert wurden.

Die wichtigsten Zusammenhänge, die sich aus den Abbildungen 8.3 und 8.4 ablesen lassen, sind im Folgenden zusammengefasst:

- **Senkung der Taktrate:**
  Die durch die Senkung der Taktrate verursachte Reduzierung der Temperatur beträgt in allen Fällen ca. 0,5 bis 1°C.

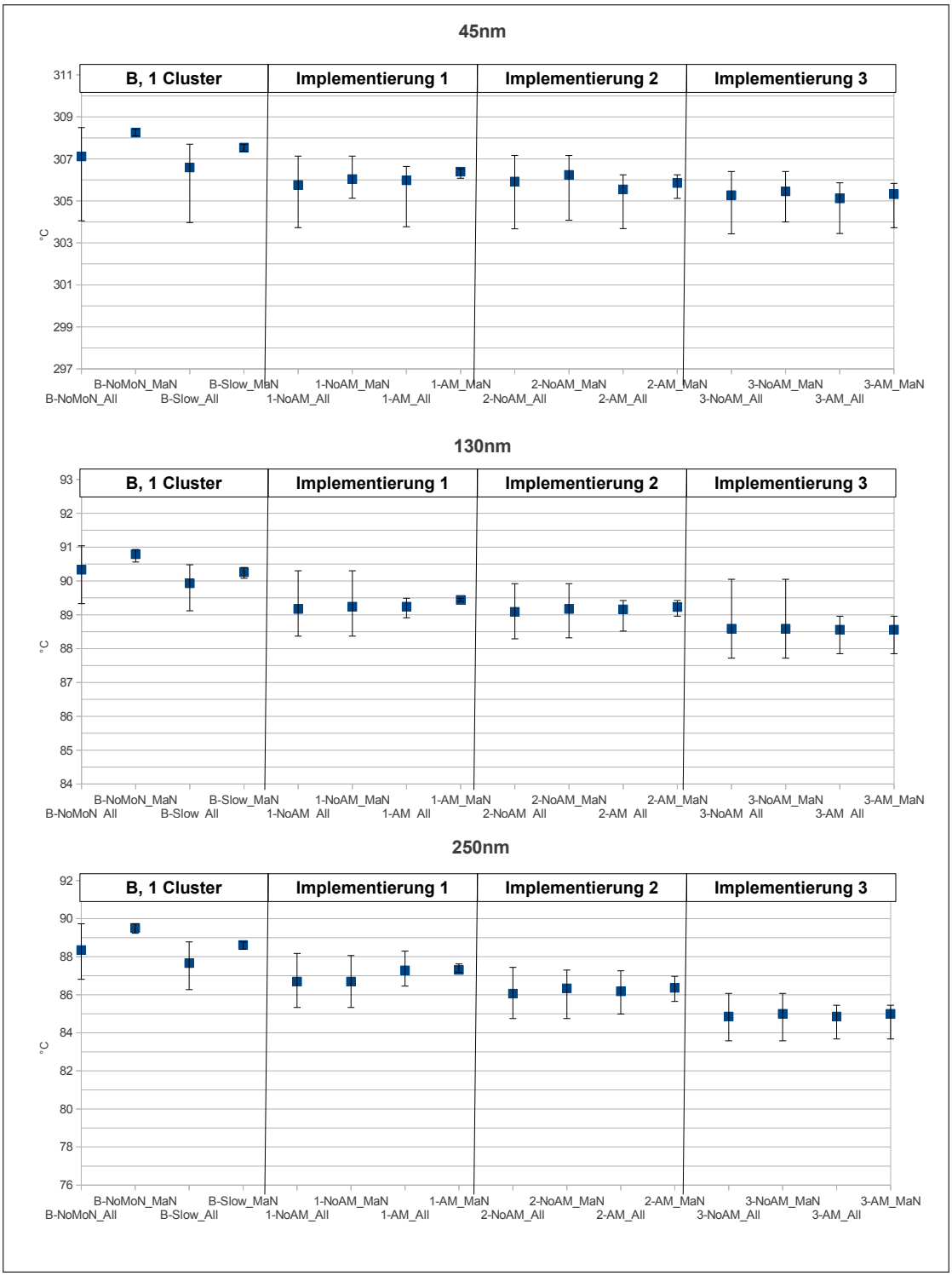


Abbildung 8.3.: Vergleich der Temperaturen des Originalsystems B, des verlangsamten Originalsystems und der Implementierungen 1-3 des M aus N Systems ohne und mit AM

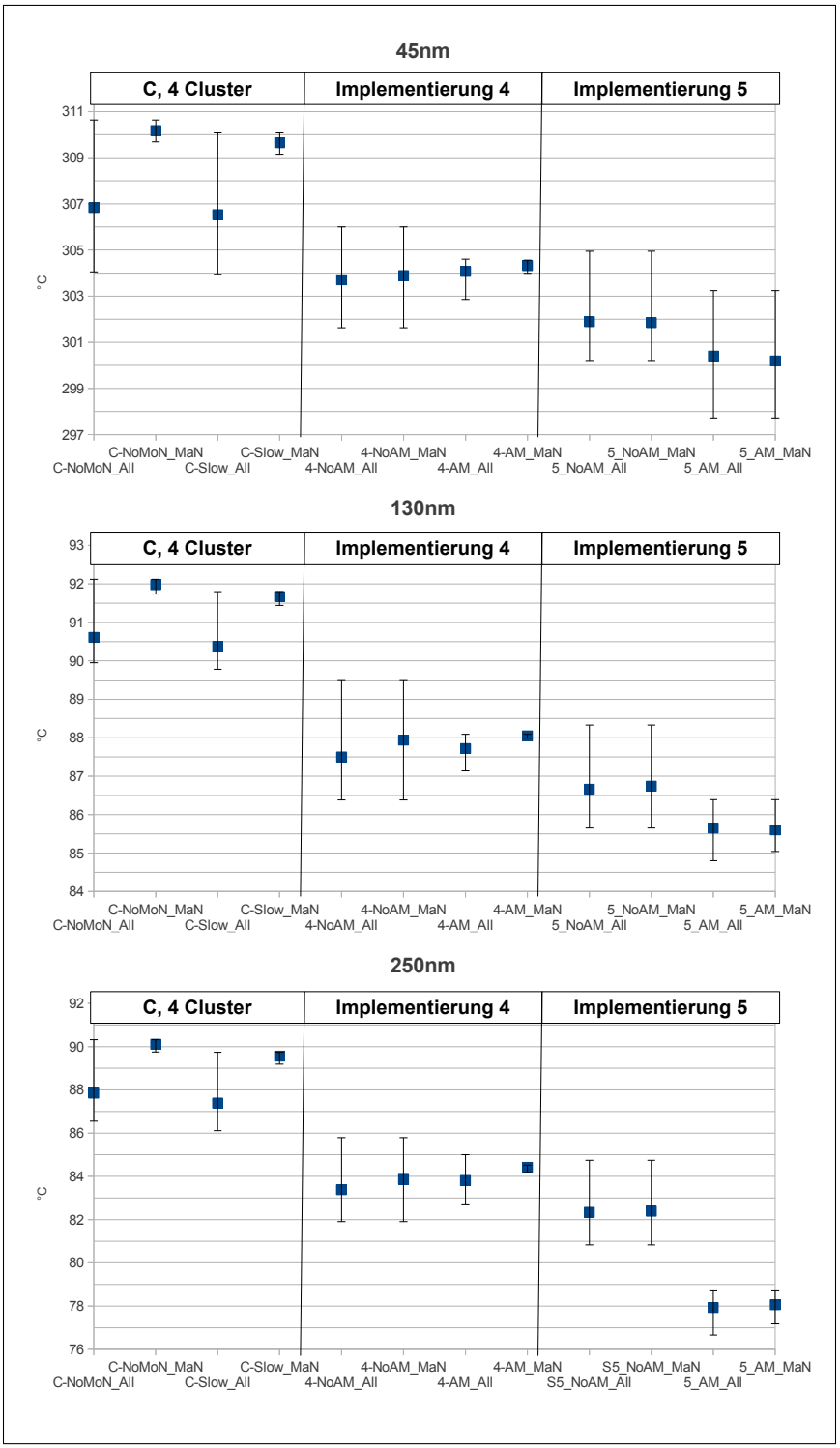


Abbildung 8.4.: Vergleich der Temperaturen des Originalsystems C, des verlangsamten Originalsystems und der Implementierungen 4-5 des M aus N Systems ohne und mit AM

- **Die Größe der Redundanz:**
  Das Hinzufügen bzw. Vergrößern der Redundanz hat zusätzlichen positiven Einfluss auf die Temperatur. Dies ist vor allem bei den Mehr-Cluster Versionen erkennbar.

- **Einfluss der Activity Migration:**
  Dadurch, dass die AM die kalten, inaktiven Komponenten belastet und die heißen, aktiven entlastet, nähern sich die Tiefst- und Höchstwerte der Temperaturverteilung dem Mittelwert. Dieser bleibt aber in etwa gleich, wenn er nicht durch die zusätzliche Schaltaktivität etwas ansteigt. Die „Mittelung“ dieser Werte kann damit begründet werden, dass es sich um ein 4 aus 8 System handelt, die Last also genau halbiert wird. Bei einem 4 aus 10 System würde sich dieser Wert wohl weiter nach unten verschieben.
  Zusammenfassend heißt das: HotSpots können im vorliegenden Fall (meist) vermieden werden, eine Senkung der allgemeinen Temperatur ist nicht erkennbar.

- **Einfluss der Technologie:**
  Die Technologien unterscheiden sich untereinander im Mittelwert der Temperatur und in der Spanne ihres Höchst- und Tiefstwertes. Die Auswirkungen der AM bleiben innerhalb dieser Werte gleich. Was aber zu bemerken bleibt, ist die ungewöhnlich hohe Temperatur der Implementierungen in der 45nm Technologie. Diese fällt mit über 290°C unrealistisch hoch aus. Der Grund hierfür liegt in der identischen Modellierung der Parameter von HotSpot für alle Technologien. Mit stimmigen Werten für Kühlung, Wärmeleitwiderstand und -kapazität und passenden Angaben für die Leistungsaufnahme der umliegenden Komponenten wären hier sicherlich realitätsnähere Ergebnisse möglich gewesen. Diese Werte liegen aber nicht vor, deshalb wurden sie für alle Technologien als identisch angenommen.

- **Wahl der Implementierungen:**
  Deutlich erkennbar ist: Je mehr Baugruppen zu den austauschbaren Komponenten des M aus N Systems gehören, desto größer ist die Spanne zwischen höchster und tiefster Temperatur nach der AM. Bei den geclusterten Versionen wäre dies sicher auch der Fall. Da aber jeder Cluster als eine Komponente modelliert wurde, besitzen diese auch nur jeweils eine gemittelte Temperatur und so fallen die Höchst- und Tiefstwerte zwischen ihnen merklich enger aus.
  Zusammenfassend kann gesagt werden, dass bei Implementierung 1 mit den Funktionseinheiten genau die heißesten Komponenten gekühlt werden. Bei den anderen kommen, an der größer werdenden Spanne erkennbar, Komponenten hinzu, die von Anfang an nicht so heiß waren und damit nicht unbedingt gekühlt werden müssten. Die Auswirkungen der AM bleiben abgesehen davon aber über alle Implementierungen gleich. Bei Implementierung 5 treten sie sogar am stärksten hervor, da bei der Version ohne AM die relativ große, aber dafür größtenteils inaktive Kopierhardware fehlt und damit das Verhältnis von Leistung zu Fläche des Systems deutlich ansteigt.

- **Unterschied zwischen ungeclusterten und geclusterten Versionen der Implementierungen:**
  Die „Temperatursprünge“ zwischen den geclusterten Versionen fallen deutlich größer aus als bei den ungeclusterten. Dies liegt einerseits daran, dass wesentlich mehr Redundanz hinzu kommt. Andererseits sind die ungeclusterten Versionen deutlich kompakter. Da die Komponenten stärker thermisch koppeln, sich also mehr gegenseitig aufwärmen, sind die Spannen kleiner. Dem könnte entgegengewirkt werden, indem die Komponenten thermisch entkoppelt, also über die gesamte Fläche des Chip verteilt, würden. Damit müssten eventuell aber auch längere Signallaufzeiten und damit eine geringere Taktrate in Kauf genommen werden.

- **Die allgemeine Temperaturspanne:**
  Als letzter Punkt muss bemerkt werden, dass die allgemeine Temperaturspanne eher gering ausfällt. In [Jay09] wird angedeutet, dass die Temperatur der heißesten Komponente 15°C über dem Durchschnitt liegt. In diesem Fall würde sich auch die AM in Bezug auf die Vermeidung von Hotspots stärker auswirken. Solche hohen Werte treten bei den vorliegenden Untersuchungen aber nicht auf. Dies kann einerseits an den Modellierungen der Hitzesimulationen liegen, andererseits am Prozessor selbst, der insgesamt weniger oder eine gleichmäßiger über die Komponenten verteilte Leistung umsetzt.

Um die Betrachtungen über die Auswirkungen der AM abzuschließen, soll noch ein weiterer, bereits angedeuteter Punkt untersucht werden: Die thermische Kopplung zwischen den Komponenten. Es kann davon ausgegangen werden, dass die bisherige Modellierung der SOC den schlechtesten Fall in Bezug auf die Temperatur darstellt. Alle Komponenten liegen eng beieinander und heizen sich gegenseitig auf. Weiterhin befinden sie sich in der Mitte des SOC und sind umgeben von Komponenten, die das 0,75-fache Leistung zu Fläche Verhältnis des Prozessors besitzen und so stark zur Erwärmung beitragen. Damit der Einfluss der thermischen Kopplung auf die Abkühlung der Komponenten durch die AM gezeigt werden kann, sind zwei weitere Versuche notwendig:

1. die thermische Entkopplung der Komponenten des M aus N Systems unter Beibehaltung aller anderen Parameter, um zu prüfen, wie sie sich gegenseitig beeinflussen und

2. das „Ausschalten“ der umgebenden, virtuellen Komponenten, um ihre Einfluss auszuklammern.

Als Untersuchungsobjekte wurden hier Implementierung 1 und 4 in 130nm gewählt, da sie zwei Extreme darstellen, die aber keine „störenden“ Faktoren besitzen, wie zum Beispiel die Kopierhardware in Implementierung 5 und 130nm als robuste Technologie gilt. Für die Versuche wurden die Floorplans, wie in Abbildung 8.5 dargestellt, verändert. Auf der linken Seite sind dabei die Floorplans der originalen SOC von Implementierung 1 (oben) und 4 (unten) zu sehen. Der VLIW Prozessor in der Mitte wurde zur besseren Unterscheidung grau hinterlegt. Auf der rechten Seite befinden sich die thermisch entkoppelten

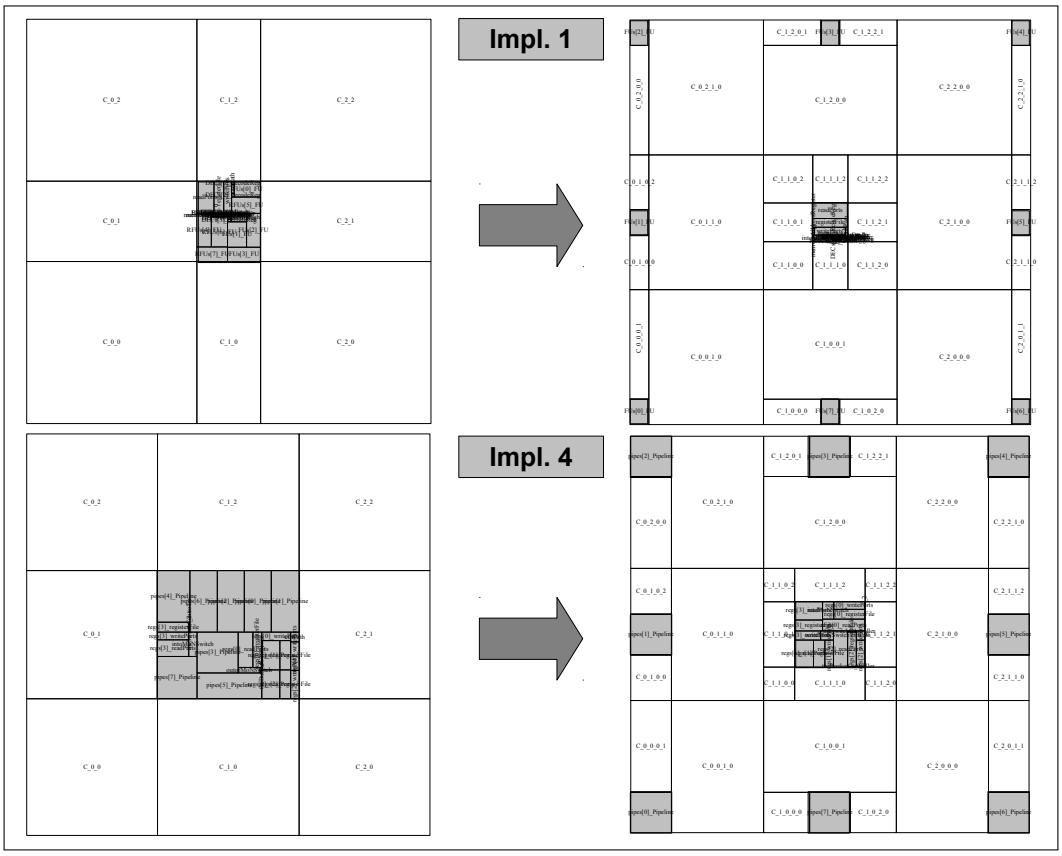


Abbildung 8.5.: Die Floorplans der SOC von Impl. 1 und 4 ohne und mit thermischer Entkopplung

Versionen, bei denen die Komponenten des M aus N Systems jeweils in den Ecken und Mitten der Außenkanten verteilt sind. Die Komponenten des VLIW Prozessors, die nicht zum M aus N System gehören, befinden sich weiterhin in der Mitte. Um den Einfluss der virtuellen Komponenten zu ermitteln, wurden diese mit ihrer originalen Leistung und mit einer Leistung von $0W$ simuliert.

Die Ergebnisse dieser Simulationen können Abbildung 8.6 entnommen werden. Auf der linken Seite sind dabei noch einmal die Temperaturwerte der originalen Versionen, in der Mitte die der entkoppelten Versionen und rechts die der entkoppelten Versionen mit deaktivierten umgebenden Komponenten dargestellt. Der Einfluss der Kopplung wird deutlich. Links sind alle Komponenten in etwa gleich heiß. In der Mitte sind die Komponenten des M aus N Systems um ca. 1°C kühler, was daran liegt, dass sie sich nun am Rand befinden und besser ihre Wärme abführen können. In den rechts abgebildeten Versionen hat eine deutliche Abkühlung um ca. 40°C stattgefunden. Der Grund hierfür liegt darin, dass durch die abgeschalteten virtuellen Komponenten das Verhältnis von Leistung zu Fläche des SOC deutlich sinkt. Die nicht zum M aus N System gehörenden Komponenten des VLIW sind dabei merklich kühler als die des M aus N Systems, da sie nur noch von kühlen virtuellen Komponenten umgeben sind und selbst nicht so stark Wärme produzieren.

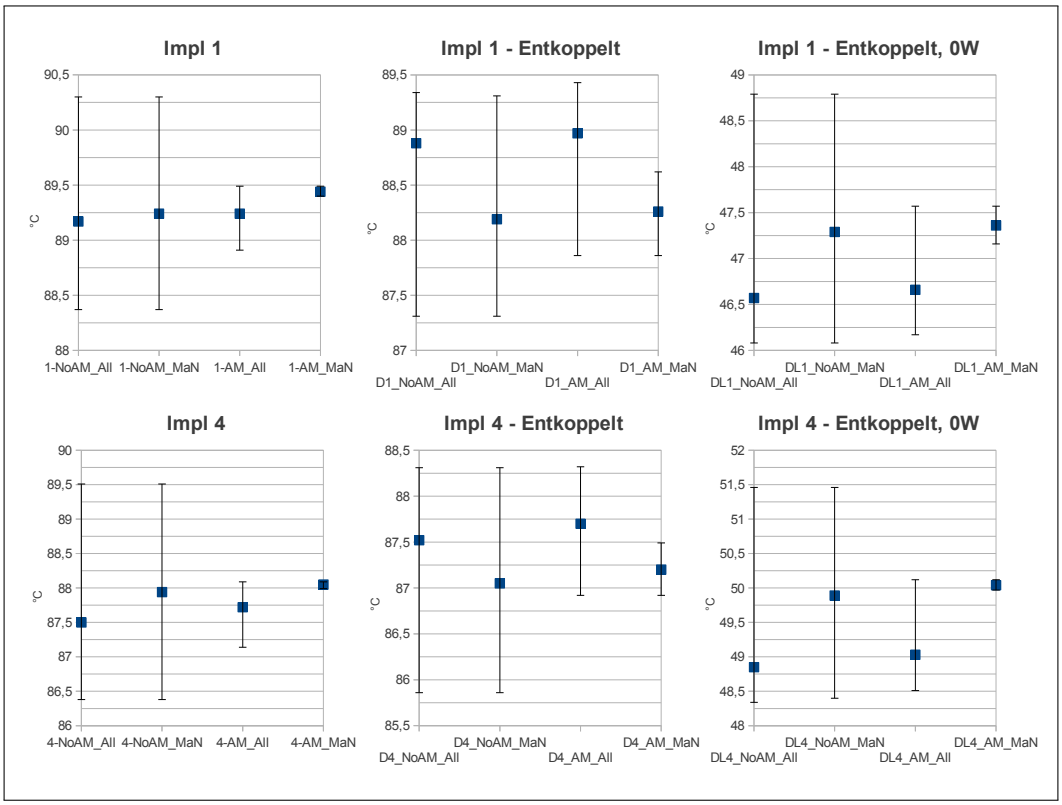


Abbildung 8.6.: Die Temperaturwerte der SOC von Impl. 1 und 4 in 130nm ohne und mit thermischer Entkopplung

Bezüglich des Einflusses der AM auf die Temperaturentwicklung kann keine größere Veränderung festgestellt werden. Die mittlere Temperatur steigt erneut leicht im Vergleich zu den Ausführungen ohne AM und die Temperatur von Hotspots wird erkennbar reduziert.

8.3. Auswirkungen auf Zuverlässigkeit und MTTF

Zuverlässigkeit und MTTF sind geeignete Metriken, um den Nutzen der Activity Migration in M aus N Systemen zu beschreiben, da über die Formeln die Fläche, Temperatur und das Ersetzungsschema gleichzeitig mit in die Berechnungen einfließen. Dieser Abschnitt fasst noch einmal kurz den Ablauf zusammen, listet die ermittelten Ergebnisse auf und setzt sie zueinander in Relation.

Für die Bestimmung der Zuverlässigkeit nach den in Abschnitt 7 beschriebenen Formeln sind mehrere Parameter notwendig. Die folgende Übersicht fasst diese noch einmal zusammen und erläutert, wie sie für die vorliegenden Untersuchungen erzeugt wurden oder aus welchen Quellen sie stammen.

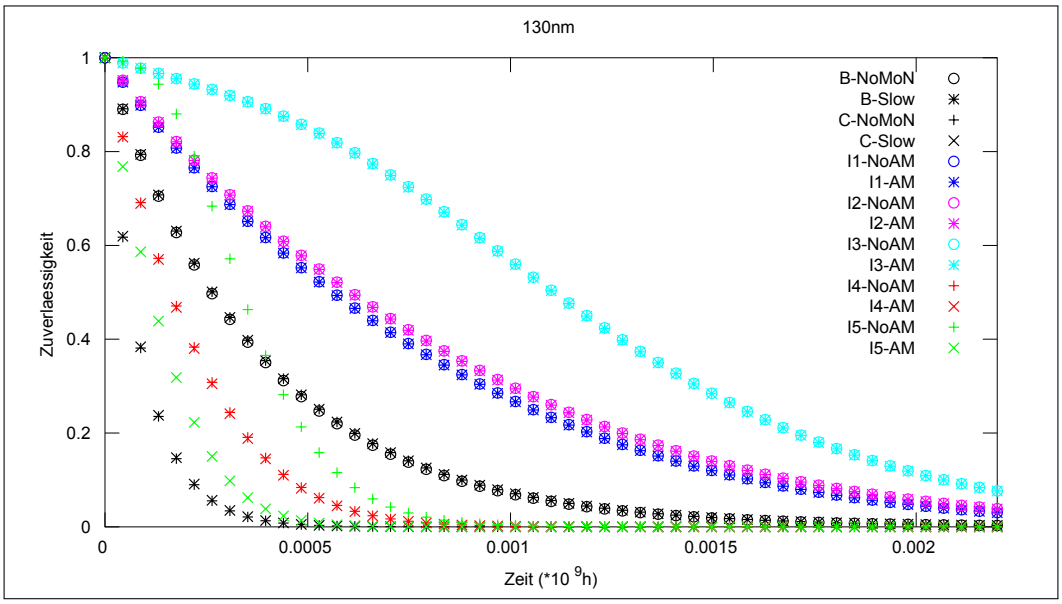


Abbildung 8.7.: Zuverlässigkeit der Systeme in 130nm

- **Temperaturen der Komponenten**
  Die Temperaturen stammen aus den Simulationen (siehe letzter Abschnitt) und müssen in Kelvin vorliegen.

- **Beschleunigungsparameter und Aktivierungsenergie**
  Diese beiden Faktoren werden bei den Berechnungen der zu den jeweiligen Fehlereffekten gehörigen Fehlerraten benötigt. Für die vorliegenden Untersuchungen konnten jedoch nur die bereits in Tabelle 2.1 dargestellten Werte ermittelt werden. Da sie nur für die 130nm Technologie gelten, soll im weiteren Verlauf auch nur diese betrachtet werden.

- **Boltzmann Konstante:**
  Die Boltzmann Konstante beträgt $k = 8,6173324 \cdot 10^{-5}\ eV/K$.

- **Fläche der Komponenten:**
  Um die Flächen der Komponenten bei den Berechnung mit in Betracht zu ziehen, wird jeweils die Anzahl der Technologiezellen nach Formel 7.7 zur Fehlerrate der Komponente multipliziert.

Für die Durchführung wurden die Flächenverhältnisse und Temperaturen aus den vorhergehenden Untersuchungen zusammengefasst und die Ergebnisse mit Octave (siehe Anhang A) ermittelt. Die beiden Basissysteme in schneller und langsamer Version konnten dabei als reine serielle Systeme (Formel 7.7) beschrieben und entsprechende Resultate bestimmt werden. Die fünf Implementierungen wurden ohne AM nach Formel 7.33 und mit AM nach Formel 7.40 modelliert.

Die Ergebnisse sind in Abbildung 8.7 dargestellt. Die Zuverlässigkeit wird dabei auf der Ordinate und die Zeit auf der Abszisse angegeben. Zu den Zeitangaben muss aber gesagt werden, dass diese mit hoher Wahrscheinlichkeit nicht der Realität entsprechen. Eine Lebensdauer eines Systems von über 10^7 Stunden (über 1140 Jahre) ist als eher unwahrscheinlich zu betrachten. Die Zeitangabe FIT war aber die einzige, die der Quelle [WB08], aus der auch die Formeln zur Berechnung der Fehlerraten stammen, entnommen werden konnte, obwohl sie nie mit den Formeln direkt in Verbindung gebracht wird. Die mangelnden Angaben sind jedoch eher ein allgemeines Problem. Um dieses zu umgehen, benutzen beispielsweise die Entwickler von RAMP (einem Programm für Zuverlässigkeitsberechnung) die Angabe 30 Jahre als mittlere Lebensdauer eines Systems in der 180nm Technologie und ziehen von diesem Wert Rückschlüsse auf die Fehlerraten ([SABR04]). Weiterhin werden in [WB08] keine Angaben zum Bezug zur Fläche gemacht. Wo im vorliegenden Fall eine Technologiezelle als kleinste, von einem Fehler beeinflussbare Einheit angenommen wird, könnten sich die Werte und Formeln aus [WB08] auf $1\mu m^2$ oder die Größe eines Transistors beziehen. Dies würde die großen Werte erklären. Da aber die Verhältnisse zwischen den Flächen über die Zellen in den hier eingesetzten Modellen integriert wurden, würde dies die Zuverlässigkeitskurve nur stauchen oder strecken. Die Verhältnisse zwischen den Kurven und damit zwischen den Implementierungen bleiben aber gleich und sind damit aussagekräftig.

Was den Graphen aus 8.7 entnommen werden kann, lässt sich wie folgt zusammenfassen:

- Die Implementierungen übertreffen in Bezug auf die Zuverlässigkeit immer das jeweilige Basissystem (in schwarz).
- In den Implementierungen 1-4 bedeutet das Hinzufügen von Redundanz und das Integrieren von mehr Hardware in das M aus N System einen Gewinn an Zuverlässigkeit. Das ist damit zu erklären, dass mehr Komponenten repariert werden können, weniger außerhalb des M aus N Systems liegen und die Temperatur sinkt.
- Implementierung 5 mit AM hat im Bezug zum Basissystem den geringsten Zuwachs an Zuverlässigkeit. Dies liegt daran, dass die große Kopierhardware nicht repariert werden kann. Bei der Version ohne AM entfällt diese Hardware und damit wird sie zuverlässiger als Implementierung 4.
- Bis auf Implementierung 5 sind in allen Fällen die Zuverlässigkeiten ohne und mit AM nahezu identisch.

Interessant ist aber nicht nur das Verhältnis der Systeme zueinander, sondern auch das Verhältnis der Zuverlässigkeit der Komponenten außerhalb des M aus N Systems zur Zuverlässigkeit der Komponenten darin. Diese sind in den Abbildungen 8.8 und 8.9 dargestellt. Zu erkennen ist, dass einerseits wieder in den meisten Fällen die Zuverlässigkeit der Systeme ohne AM der Zuverlässigkeit der Systeme mit AM entspricht. Andererseits kann nun aber auch abgelesen werden, dass bei Implementierung 4 und Implementierung 5 ohne AM (und damit ohne Kopierhardware) die Komponenten außerhalb des M aus N Systems flächenmäßig so klein sind, dass die im Vergleich riesigen M aus N Systeme

Tabelle 8.2.: MTTF der Implementierungen in $10^9 h$

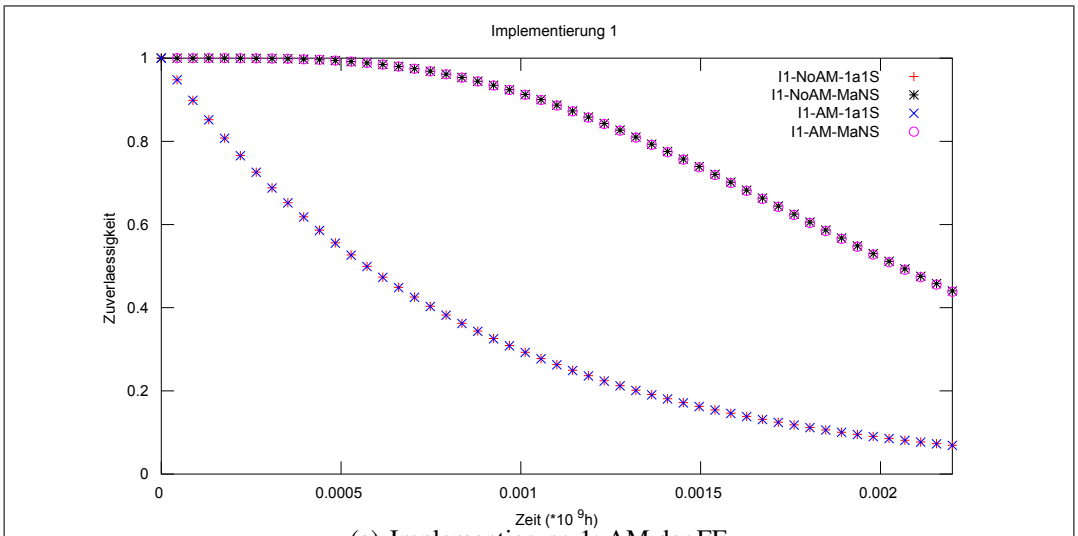
	B	C			
NoMoN	3,78e-4	9,13e-5			
Slow	3,82e-4	9,19e-5			
Slow LIF	1,011	1,007			
	I1	I2	I3	I4	I5
NoAM-1a1S	8,24e-4	8,94e-4	3,92e-3	2,39e-4	6,19e-3
NoAM-MaNS	2,21e-3	2,24e-3	1,46e-3	6,16e-4	3,75e-4
AM-1a1S	8,23e-4	8,91e-4	3,88e-3	2,37e-4	1,67e-4
AM-MaNS	2,21e-3	2,24e-3	1,46e-3	6,15e-4	3,85e-4
LIF_{NoAM}	1,59	1,69	2,82	1,89	3,87
LIF_{AM}	1,59	1,69	2,81	1,87	1,28

wesentlich unzuverlässiger werden. Dennoch sind sie insgesamt weiterhin zuverlässiger als die Basisversionen.

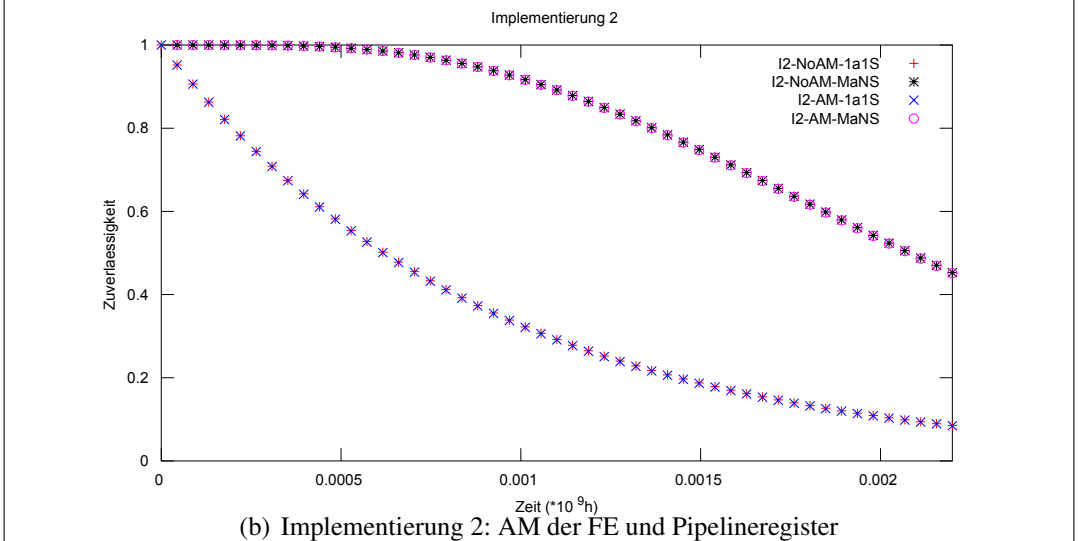
Als letzte zu untersuchende Metrik ist nun noch die MTTF anzugeben. Die über die Formeln 7.37 und 7.41 ermittelten Ergebnisse sind dabei in Tabelle 8.2 dargestellt. B und C stehen wieder für die Basissysteme und I1-I5 für die Implementierungen. Die Angaben sind in FIT, also 10^9 Stunden, deren Realitätsnähe bereits diskutiert wurde. Viel aussagekräftiger ist die Relation zwischen den jeweiligen Ausführungen ohne und mit AM. Werden die entsprechenden Werte für die mittlere Lebensdauer ins Verhältnis gesetzt, kann sehr gut abgelesen werden, um welchen Faktor sie sich ändern. Dieser Faktor ist auch bekannt als Lifetime Improvement Factor (LIF). Im vorliegenden Fall ist der LIF immer durch das Verhältnis zur jeweiligen Basisversion ermittelt worden. Das heißt folgendermaßen, wenn $MTTF_B$ die mittlere Lebensdauer des Basissystems, $MTTF_{1a1}$ die der Komponenten außerhalb und $MTTF_{Sys|NoAM}$ die der Komponenten innerhalb des M aus N Systems ohne und $MTTF_{Sys|AM}$ der Komponenten mit AM, ist:

$$LIF_{NoAM} = \frac{\frac{MTTF_{1a1} \cdot MTTF_{Sys|NoAM}}{MTTF_{1a1} + MTTF_{Sys|NoAM}}}{MTTF_B} \quad \text{oder} \quad LIF_{AM} = \frac{\frac{MTTF_{1a1} \cdot MTTF_{Sys|AM}}{MTTF_{1a1} + MTTF_{Sys|AM}}}{MTTF_B} \tag{8.2}$$

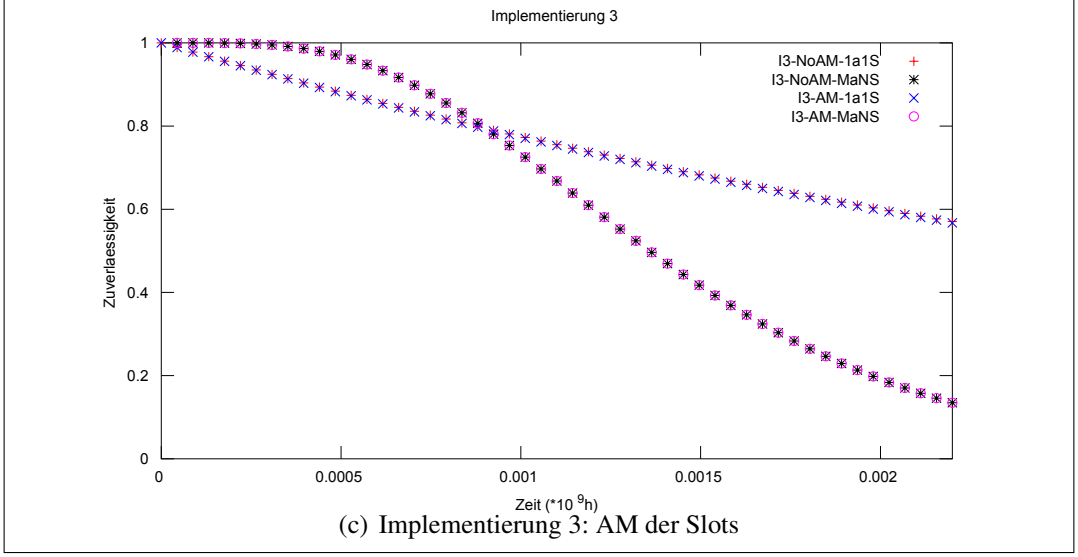
Aus der vierten Zeile kann abgelesen werden, dass die Verringerung der Taktrate die Lebensdauer trotz leichter Temperatursenkung kaum beeinflusst. Bei den M aus N Systemen ohne und mit AM sind die Auswirkungen wesentlich deutlicher. Implementierung 3 mit einem LIF von 2,82 hat dabei den höchsten Wert. Die anderen unterscheiden sich dagegen kaum. Ausnahme bildet natürlich Implementierung 5 ohne AM. Hier ist der Einfluss der fehlenden Kopierhardware sehr stark, die ohne AM die Komponenten außerhalb des M aus N Systems sehr klein und mit AM sehr groß macht, dadurch deren Fehleranfälligkeit stark beeinflusst und den LIF zusätzlich hebt oder senkt.



(a) Implementierung 1: AM der FE

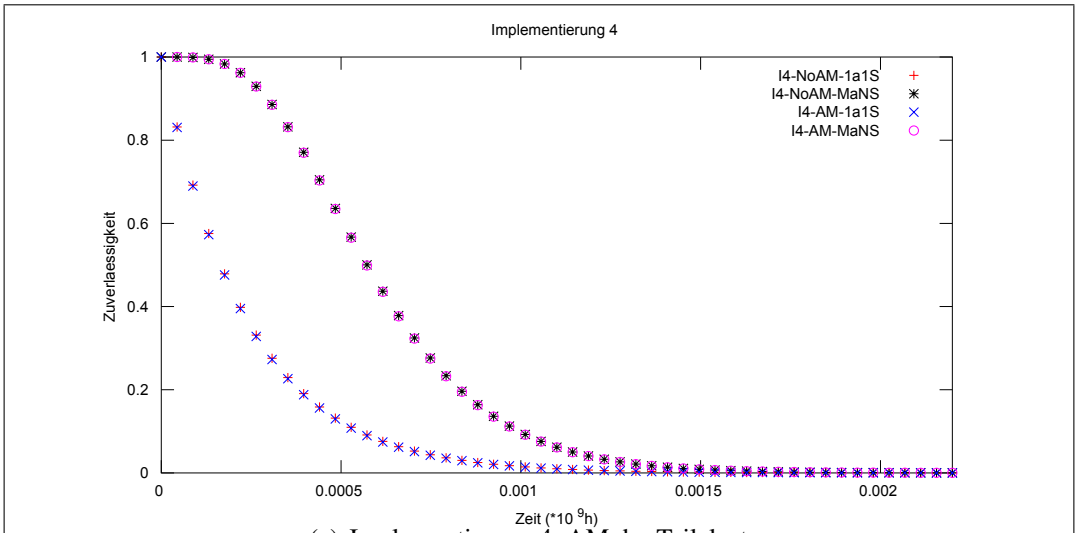


(b) Implementierung 2: AM der FE und Pipelineregister

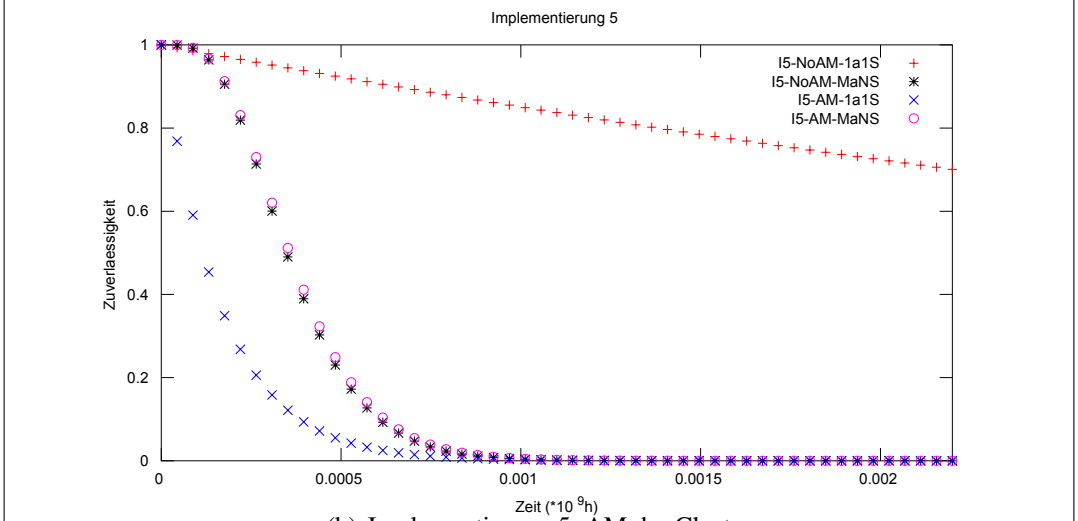


(c) Implementierung 3: AM der Slots

Abbildung 8.8.: Graphische Übersicht der fünf Implementierungen mit AM



(a) Implementierung 4: AM der Teilcluster



(b) Implementierung 5: AM der Cluster

Abbildung 8.9.: Graphische Übersicht der fünf Implementierungen mit AM

Kapitel 9

Zusammenfassung und Ausblick

Ziel dieser Schrift war es, die Kosten und den Nutzen der Activity Migration in M aus N Systemen zu ermitteln, um darüber Aussagen treffen zu können, ob dieser vielversprechende Ansatz wirklich so vorteilhaft ist, wie ursprünglich vermutet. Dazu wurde im ersten Teil das grundlegende Prozessormodell vorgestellt und die wichtigsten Begriffe, wie Zuverlässigkeit, Stress und MTTF, eingeführt. Danach fand eine Auseinandersetzung mit dem derzeitigen Stand der Technik statt. Verschiedene Methoden, die das Entstehen von Defekten verzögern und einmal aufgetretene Fehler in gewissen Grenzen tolerieren können, wurden vorgestellt. Der vierte Teil präsentierte den hier zu untersuchenden Ansatz und nannte verschiedene Punkte, weshalb er so vorteilhaft erschien:

- Die Integration von Activity Migration in ein zur Selbstreparatur eingesetztes M aus N System als Alleinstellungsmerkmal,
- das breite Anwendungsfeld des Ansatzes und seine Skalierbarkeit,
- die Degradationsfreiheit der Selbstreparatur und des Destressens und
- die geringen Zusatzkosten für die Implementierung der Activity Migration.

Danach wurde die Implementierung des Ansatzes in einem VLIW Prozessor in fünf verschiedenen Versionen beschrieben und die dabei entstehenden Problemstellungen diskutiert. Spezielles Augenmerk wurde hier auf eine höchstmögliche Automatisierung der AM gelegt, und dass diese ohne zusätzlichen zeitlichen Aufwand während des Betriebs durchgeführt werden kann. Daraufhin folgte die Beschreibung der Simulationen. Hier wurde auf eine möglichst detaillierte Darstellung geachtet, um die Nachvollziehbarkeit jedes Schrittes zu gewährleisten. Der nächste Abschnitt legte die Formeln zur Berechnung der Zuverlässigkeit und mittleren Lebensdauer in den unterschiedlichen Systemen dar und beschrieb die Herleitung, wenn diese selbst entwickelt wurde.

Die ermittelten Ergebnisse der Implementierungen in 130nm sind noch einmal in Tabelle 9.1 dargestellt. Die Spalten geben dabei die Nummer der jeweiligen Implementierung an und die Zeilen die entsprechende Metrik. Zeile 1 fasst den zusätzlichen Hardwareaufwand des M aus N Systems mit AM im Vergleich zum Basissystem zusammen. Es lässt sich erkennen, dass der Zusatz an Hardware relativ hoch ausfällt. Das liegt in den meisten Fällen aber am M aus N System selbst, wie sich in der zweiten Spalte ablesen lässt. Darin ist der Zusatzaufwand im Vergleich zu den Implementierungen mit M aus N System ohne

Tabelle 9.1.: Zusammenfassung der Ergebnisse

	I1	I2	I3	I4	I5
HW Overhead - BS	43,42%	43,58%	108,87%	44,47%	196,36%
HW Overhead - NoAM	0,2%	0,00%	0,36%	0,43%	45,45%
Temperaturdiff - BS	-1,09	-1,17	-1,77	-2,89	-4,96
Temperaturdiff - NoAM	0,07	0,07	-0,03	0,22	-1,01
LIF - BS	1,59	1,69	2,81	1,87	1,28
LIF - NoAM	0,998	0,997	0,997	0,994	0,329

AM dargestellt. Lediglich Implementierung 5 fällt mit AM deutlich größer aus, da die Kopierhardware sehr groß ist.

Im Vergleich zum Basissystem (Zeile 3) konnte in jedem Fall eine Senkung der Temperatur festgestellt werden. Aber auch diese liegt oft an der zusätzlichen Redundanz und nicht an der AM, was Zeile 4 entnommen werden kann. Nur in zwei von fünf Fällen ist es möglich, die mittlere Temperatur mit der AM weiter zu senken. Bei Implementierung 5 spielt aber erneut die Kopierhardware eine wichtige Rolle. Sie ist sehr groß, entfaltet aber vergleichsweise geringe Aktivität. Damit sinkt das gesamte Verhältnis der Leistung zur Fläche des Systems mit AM. Die letzten beiden Zeilen geben den Faktor für den Gewinn an Lebensdauer (LIF) an. Aus ihnen lässt sich erkennen, dass die Lebensdauer im Vergleich zum Basissystem deutlich gesteigert werden kann. Im Vergleich zum reinen M aus N System (letzte Zeile) ist dieser Faktor bei den Versionen mit AM aber immer etwas kleiner.

Zusammenfassend kann gesagt werden, dass sich das Einfügen der AM in Bezug auf eine Steigerung der Lebensdauer, verglichen mit dem reinen M aus N System, nicht lohnt. Dies kann damit begründet werden, dass sich die durchschnittliche Temperatur des Systems durch Migration der Aktivität kaum verändert, wenn nicht sogar etwas erhöht. Selbst die thermische Entkopplung der Komponenten ändert daran nichts. Eine positive Folge der AM ist die Senkung der Temperaturen der heißesten Komponenten im M aus N System. Falls diese einen kritischen Wert überschreiten, kann das Einfügen der AM durchaus eine lohnenswerte Methode darstellen, da die zusätzlichen Kosten an Hardware sehr gering ausfallen. An dieser Stelle müsste in weiteren Untersuchungen überprüft werden, ob die Auswirkungen der AM bei größeren Temperaturspannen zwischen Mittel- und Höchstwert, nicht doch zu einer Steigerung der Lebensdauer des gesamten Systems führen könnten.

Zum Abschluss soll noch einmal speziell auf mögliche Ungenauigkeiten hingewiesen werden. Alle Ergebnisse basieren auf Simulationen und damit Näherungsmodellen und auf den in den entsprechenden Teilen beschriebenen Annahmen. Somit kann hier keine Anspruch auf absolute Richtigkeit erhoben werden. Es wurden aber sämtliche Lösungswege detailliert geschildert und versucht, Ungenauigkeiten so weit wie möglich zu eliminieren (z.B. das Verhältnis der Fehlerrate zur Fläche). Damit können in weiteren Untersuchungen auch Teile in den Lösungswegen ausgetauscht und, falls exaktere Werte vorliegen, neue Ergebnisse ermittelt werden. Dies betrifft unter anderem die Formeln zur Bestimmung

der Fehlerraten unter den Fehlereffekten aus Kapitel 2. NBTI und PBTI gelten beispielsweise als nicht stark temperaturabhängig. Weiterhin treten NBTI und HCI verstärkt bei den passiven Komponenten auf, die im vorliegenden Fall unter konstanter Vorspannung stehen, aber nicht aktiv arbeiten. Hier kommt es sogar zu Heilungseffekten, sobald die Komponenten wieder unter Last stehen ([KV10]). Dies wurde im vorliegenden Fall nicht berücksichtigt, da die eingesetzten Modelle solche Unterschiede nicht hergeben. Außerdem soll die differenzierte Beschreibung von fehlerbeschleunigenden und -heilenden Effekten als einfließende Parameter bei der AM für zukünftige Arbeiten offen gelassen werden.

Literaturverzeichnis

[AMP06] AMARI, Suprasad V. ; MISRA, Krishna B. ; PHAM, Hoang: Reliability analysis of tampered failure rate load-sharing k-out-of-n: G systems. In: *Proceedings of tThe 12th ISSAT International Conference on Reliability and Quality in Design* (2006), S. 30–35

[ARJS07] AGGARWAL, Nidhi ; RANGANATHAN, Parthasarathy ; JOUPPI, Norman P. ; SMITH, James E.: Configurable Isolation: building high availability systems with commodity multi-core processors. In: *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)* (2007), S. 470–481

[AVU+08] ABELLA, Jaume ; VERA, Xavier ; UNSAL, Osman S. ; ERGIN, Oguz ; GONZÁLEZ, Antonio ; TSCHANZ, James W.: Refueling: Preventing Wire Degradation due to Electromigration. In: *IEEE Micro* 28 (2008), Nr. 6, S. 37–46

[BM01] BROOKS, David ; MARTONOSI, Margaret: Dynamic Thermal Management for High-Performance Microprocessors. In: *Seventh International Symposium on High-Performance Computer Architecture (HPCA'01)* (2001), S. 171–182

[Bor99] BORKAR, S.: Design challenges of technology scaling. In: *IEEE Micro* 19 (1999), Nr. 4, S. 23–29

[Bor05] BORKAR, Shekhar: Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. In: *IEEE Micro* 25 (2005), Nov./Dec., Nr. 6, S. 10–16

[BS89] BHATTACHARYYA, G. K. ; SOEJOETI, Zanzawi: A tampered failure rate model for step-stress accelerated life test. In: *Communications in Statistics - Theory and Methods* 18 (1989), Nr. 5, S. 1627–1643

[CB95] CHANDRAKASAN, Anantha P. ; BRODERSEN, Robert W.: Minimizing power consumption in digital CMOS circuits. In: *Proceedings of the IEEE* 83 (1995), Nr. 4, S. 498–523

[CCF+07] CHOI, Jeonghwan ; CHER, Chen-Yong ; FRANKE, Hubertus ; HAMANN, Henrdrik ; WEGER, Alan ; BOSE, Pradip: Thermal-aware task scheduling at the system software level. In: *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)* (2007), S. 213–218

[CGG04] CHAPARRO, P. ; GONZALEZ, J. ; GONZALEZ, A.: Thermal-aware clustered microarchitectures. In: *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'04)* (2004), S. 48–53

[Che04] CHEN, Wai K.: *The Electrical Engineering Handbook*. Elsevier Science, 2004 http://books.google.de/books?id=qhHsSlazGrQC. – ISBN 9780080477480

[CPL+02] CHOI, M. ; PARK, Naphill ; LOMBARDI, Fabrizio ; KIM, Y. B. ; PIURI, Vincenzo: Balanced Redundancy Utilization in Embedded Memory Cores for Dependable Systems. In: *Proceedings of the 17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '02)* (2002), S. 419–427

[DM95] DEVADAS, Srinivas ; MALIK, Sharad: A survey of optimization techniques targeting low power VLSI circuits. In: *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference* (1995), S. 242–247

[DP94] DURAND, S. ; PIGUET, C.: FPGA with Self-Repair Capabilities. In: *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays (FPGA '94)* 2 (1994), S. 1–10

[EPS05] EPSMA: Guidelines to Understanding Reliability Prediction / European Power Supply Manufacturers Association. Version: June 2005. http://www.epsma.org/pdf/MTBF%20Report_24%20June%202005.pdf. 2005. – Forschungsbericht

[Fis83] FISHER, Joseph A.: Very Long Instruction Word architectures and the ELI-512. In: *Proceedings of the 10th annual international symposium on Computer Architecture (ISCA '83)* (1983), S. 263–273

[GAFM10] GUPTA, Shantanu ; ANSARI, Amin ; FENG, Shuguang ; MAHLKE, Scott: StageWeb: Interweaving Pipeline Stages into a Wearout and Variation Tolerant CMP Fabric. In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)* (2010), S. 101–110

[GFA+08] GUPTA, Shantanu ; FENG, Shuguang ; ANSARI, Amin ; BLOME, Jason ; MAHLKE, Scott: The StageNet fabric for constructing resilient multicore systems. In: *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)* (2008), S. 141–151

[GISP06] GRECU, Cristian ; IVANOV, Andre ; SALEH, Res ; PANDE, Partha P.: NoC Interconnect Yield Improvement Using Crosspoint Redundancy. In: *Proceedings of the 21st IEEE International Symposium on on Defect and Fault-Tolerance in VLSI Systems (DFT '06)* (2006), S. 457–465

[HBA03] HEO, Seongmoo ; BARR, Kenneth ; ASANOVIĆ, Krste: Reducing power density through activity migration. In: *Proceedings of the 2003 international symposium on Low power electronics and design (ISLPED '03)* (2003), S. 217–222

[HBS+04] HU, Zhigang ; BUYUKTOSUNOGLU, Alper ; SRINIVASAN, Viji ; ZYUBAN, Victor ; JACOBSON, Hans ; BOSE, Pradip: Microarchitectural techniques for power gating of execution units. In: *Proceedings of the 2004 international symposium on Low power electronics and design (ISLPED '04)* (2004), S. 32–37

[Hot13] HOTSPOT: *HotSpot*. Online. http://lava.cs.virginia.edu/HotSpot/. Version: Oktober 2013

[Jay09] JAYASEELAN, Ramkumar: *Application-specific Thermal Management of Computer Systems*, National University of Singapore, Diss., 2009

[JED11] JEDEC: Failure Mechanisms and Models for Semiconductor Devices / JEDEC Solid State Technology Association. Version: 2011. http://www.jedec.org/standards-documents/docs/jep-122e. 2011 (JEP122G). – Publication

[JG11] JAIN, Madhu ; GUPTA, Ritu: Redundancy Issues in Software and Hardware Systems: an Overview. In: *International Journal of Reliability, Quality and Safety Engineering* 18 (2011), Nr. 1, S. 61–98

[KHV06] KOTHE, Rene ; HABERMANN, Sven ; VIERHAUS, Heinrich T.: Selbstreparatur von Logik-Baugruppen in hochintegrierten Schaltungen - Möglichkeiten und Grenzen. In: *Forum der Forschung - Wissenschaftsmagazin der Brandenburgischen Technischen Universität Cottbus* Bd. 19 Brandenburg University of Technology (BTU) Cottbus, 2006, S. 125–130

[KK07] KOREN, Israel ; KRISHNA, C. M.: *Fault Tolerant Systems*. Elsevier / Morgan Kaufmann Publishers, 2007

[KSV09] KOAL, Tobias ; SCHEIT, Daniel ; VIERHAUS, Heinrich T.: Selbstreparatur durch Regularisierung von Logik-Strukturen. In: *3. GMM/GI/ITG-Fachtagung: Zuverlässigkeit und Entwurf* (2009), S. 7

[KSV10] KOAL, Tobias ; SCHEIT, Daniel ; VIERHAUS, Heinrich T.: Schwachstellen und Engpässe bei Verfahren der Selbstreparatur für hochintegrierte Schaltungen und Systeme. In: *4. GMM/GI/ITG-Fachtagung für Zuverlässigkeit und Entwurf* (2010), S. 6

[KV08] KOAL, Tobias ; VIERHAUS, Heinrich T.: Basic Architecture for Logic Self Repair. In: *14th IEEE International On-Line Testing Symposium* 14 (2008), S. 177–178

[KV10] KOAL, Tobias ; VIERHAUS, Heinrich T.: Combining de-stressing and self repair for long-term dependable systems. In: *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)* 13 (2010), S. 99–104

[KV11] KOAL, Tobias ; VIERHAUS, Heinrich T.: Optimal Spare Utilization for Reliability and Mean Lifetime Improvement of Logic Built-In Self-Repair. In: *14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems* (2011), S. 219–224

[KVKI03] KIM, H. S. ; VIJAYKRISHNAN, N. ; KANDEMIR, M. ; IRWIN, M. J.: Adapting instruction level parallelism for optimizing leakage in VLIW architectures. In: *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (LCTES '03)* (2003), S. 275–283

[KZ02] KUO, Way ; ZUO, Ming J.: *Optimal Reliability Modeling: Principles and Applications*. Bd. 1. Wiley, 2002. – 231–280 S. http://www.ewp.rpi.edu/hartford/~ernesto/S2008/SMRE/Papers/Kuo-Zuo-koon.pdf

[Lap95] LAPRIE, Jean-Claude: Dpendable Computing and Fault Tolerance: Concepts and Terminology. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing* (1995), S. 2–11

[LGS09] LORENZ, Dominik ; GEORGAKOS, Georg ; SCHLICHTMANN, Ulf: Aging Analysis of Circuit Timing Considering NBTI and HCI. In: *15th IEEE International On-Line Testing Symposium* (2009), S. 3–8

[LLP07] LEHTONEN, Teijo ; LILJEBERG, Pasi ; ; PLOSILA, Juha: Online Reconfigurable Self-Timed Links for Fault Tolerant NoC. In: *VLSI Design* 2007 (2007), S. 1–13

[LTRN92] LO, J.-C. ; THANAWASTIEN, S. ; RAO, T. R. N. ; NICOLAIDIS, M.: An SFS Berger check prediction ALU and its application to self-checking processor designs. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11 (1992), Nr. 4, S. 525–540

[LYH+03] LI, Jin-Fu ; YEH, Jen-Chieh ; HUANG, Rei-Fu ; WU, Cheng-Wen ; TSAI, Peir-Yuan ; HSU, Archer ; CHOW, Eugene: A Built-In Self-Repair Scheme for Semiconductor Memories with 2-D Redundancy. In: *Proceedings of theInternational Test Conference(ITC'03)* (2003), S. 393–402

[Mad93] MADI, Mohamed T.: Multiple step-stress accelerated life test: the tampered failure rate model. In: *Communications in Statistics - Theory and Methods* 22 (1993), Nr. 9, S. 295–306

[MHS+04] MITRA, Subhasish ; HUANG, Wei-Je ; SAXENA, Nirmal R. ; YU, Shu-Yi ; MCCLUSKEY, Edward J.: Reconfigurable Architecture for Autonomous Self-Repair. In: *IEEE Design and Test of Computers* 21 (2004), Nr. 3, S. 228–240

[MLN+06] MUTYAM, Madhu ; LI, Feihui ; NARAYANAN, Vijaykrishnan ; KANDEMIR, Mahmut ; IRWIN, Mary J.: Compiler-directed thermal management for VLIW functional units. In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems (LCTES '06)* (2006), S. 163–172

[MM06] MUKHERJEE, Rajarshi ; MEMIK, Seda O.: Physical aware frequency selection for dynamic thermal management in multi-core systems. In: *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design (ICCAD '06)* (2006), S. 547–552

[Mor11] MORE, Shailesh: *Aging Degradation and Countermeasures in Deep-submicrometer Analog and Mixed Signal Integrated Circuits*, Lehrstuhl für Technische Elektronik der Technischen Universität München, Diss., 2011

[Nan13] NANGATE: *NanGate 45nm Open Cell Library*. Online. http://www.nangate.com/?page_id=22. Version: 11 2013

[Nel90] NELSON, Victor P.: Fault-Tolerant Computing: Fundamental Concepts. In: *Computer* 23 (1990), July, Nr. 7, S. 19–25

[Neu56] NEUMANN, John von: Probabilistic logics and the synthesis of reliable organisms from unreliable components. In: *Princeton University Press* 34 (1956), S. 43–98

[Par97] PARHAMI, Behrooz: Defect, Fault, Error, ... , or Failure? In: *IEEE Transactions on Reliability* 46 (1997), December, Nr. 4, S. 450–451

[PN06] PEDRAM, Massoud ; NAZARIAN, Shahin: Thermal Modeling, Analysis, and Management in VLSI Circuits: Principles and Methods. In: *Proceedings of the IEEE* 94 (2006), August, Nr. 8, S. 1487–1501

[PSV05] POWELL, Michael D. ; SCHUCHMAN, Ethan ; VIJAYKUMAR, T. N.: Balancing Resource Utilization to Mitigate Power Density in Processor Pipelines. In: *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)* (2005), S. 294–304

[PTB+10] PANTELIDES, Sokrates T. ; TSETSERIS, L. ; BECK, M.J. ; RASHKEEV, S.N. ; HADJISAVVAS, G. ; BATYREV, I.G. ; TUTTLE, B.R. ; MARINOPOULOS, A.G. ; ZHOU, X.J. ; FLEETWOOD, D.M. ; SCHRIMPF, R.D.: Performance, reliability, radiation effects, and aging issues in microelectronics – From atomic-scale

physics to engineering-level modeling. In: *Solid-State Electronics* 54 (2010), September, Nr. 9, S. 841–848

[Red01] REDMOND, Catherine: Winning the Battle Against Latch-up in CMOS Analog Switches. In: *Analog Dialogue* 35 (2001), Nr. 5, 21-23. http://www.analog.com/library/analogDialogue/archives/35-05/latchup/latchup.pdf

[RH03] RAUSAND, Marvin ; HØYLAND, Arnljot: *System Reliability Theory: Models, Statistical Methods, and Applications*. 2. John Wiley & Sons, 2003 (Wiley series in probability and statistics: Applied probability and statistics). – 664 S. http://books.google.de/books?id=gkUWz9AA-QEC

[SA03] SRINIVASAN, Jayanth ; ADVE, Sarita V.: Predictive dynamic thermal management for multimedia applications. In: *Proceedings of the 17th annual international conference on Supercomputing (ICS '03)* (2003), S. 109–120

[SABR04] SRINIVASAN, Jayanth ; ADVE, Sarita V. ; BOSE, Pradip ; RIVERS, Jude A.: The Impact of Technology Scaling on Lifetime Reliability. In: *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN '04)* (2004), S. 177–186

[SABR05] SRINIVASAN, Jayanth ; ADVE, Sarita V. ; BOSE, Pradip ; RIVERS, Jude A.: Lifetime Reliability: Toward an Architectural Solution. In: *IEEE Micro* 25 (2005), Nr. 3, S. 70–80

[SAS02] SKADRON, K. ; ABDELZAHER, T. ; STAN, M. R.: Control-theoretic techniques and thermal-RC modeling for accurate and localized dynamic thermal management. In: *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture* (2002), S. 17–28

[Sch88] SCHEUER, Ernest M.: Reliability of an m-out of-n system when component failure induces higher failure rates in survivors. In: *IEEE Transactions on Reliability* 37 (1988), Nr. 1, S. 73–74

[Sch11a] SCHEIT, Daniel: *Fault-tolerant integrated interconnections based on built-in self-repair and codes*, Brandenburg University of Technology, Diss., 2011

[Sch11b] SCHÖLZEL, Mario: Fine-Grained Software-Based Self-Repair of VLIW Processors. In: *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT '11)* (2011), S. 41–49

[Sch12] SCHNABEL, Patrick: *Intel Prozessoren*. Online. http://www.elektronik-kompendium.de/sites/com/0311051.htm. Version: Dezember 2012

[Sie91] SIEWIOREK, Daniel P.: Architecture of fault-tolerant computers: an historical perspective. In: *Proceedings of the IEEE* 79 (1991), Nr. 12, S. 1710–1734

[SM10] SCHÖLZEL, Mario ; MÜLLER, Sebastian: Combining Hardware- and Software-Based Self-Repair Methods for Statically Scheduled Data Paths. In: *Proceedings of the 2010 IEEE 25th International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '10)* (2010), S. 90–98

[SP92] SHE, J. ; PECHT, M. G.: Reliability of a k-out-of-n warm-standby system. In: *IEEE Transactions on Reliability* 41 (1992), Nr. 1, S. 72–75

[SS98] SIEWIOREK, Daniel P. ; SWARZ, Robert S.: *Reliable Computer Systems: Design and Evaluation*. Ak Peters Series, 1998 https://archive.org/download/reliablecomputer00siew/reliablecomputer00siew.pdf

[SSS+04] SKADRON, Kevin ; STAN, Mircea R. ; SANKARANARAYANAN, Karthik ; HUANG, Wei ; VELUSAMY, Sivakumar ; TARJAN, David: Temperature-aware microarchitecture: Modeling and implementation. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 1 (2004), Nr. 1, S. 94–125

[Ter07] TERECHKO, Andrei S.: *Clustered VLIW Architectures: a Quantitative Approach*, Technische Universiteit Eindhoven, Diss., 2007

[UKV12a] ULBRICHT, Markus ; KOAL, Tobias ; VIERHAUS, Heinrich T.: Activity Migration in M-aus-N-Systemen mit Hilfe des Load Balancing. In: *24. GI/GMM/ITG-Workshop: Testmethoden und Zuverlässigkeit von Schaltungen und Systemen* (2012)

[UKV12b] ULBRICHT, Markus ; KOAL, Tobias ; VIERHAUS, Heinrich T.: Activity Migration in M-of-N-Systems by Means of Load Balancing. In: *15th Euromicro Conference on Digital System Design* (2012)

[Ulb10] ULBRICHT, Markus: *Ein diagnostisches hybrides Testverfahren für einen VLIW Prozessor*, Brandenburg University of Technology, Diplomarbeit, 2010

[VWWL00] VISWANATH, Ram ; WAKHARKAR, Vijay ; WATWE, Abhay ; LEBONHEUR, Vassou: Thermal Performance Challenges from Silicon to Systems. In: *Intel Technology Journal* 4 (2000), Nr. 3, S. 1–16

[WB08] WHITE, Mark ; BERNSTEIN, Joseph B.: Microelectronics Reliability: Physics-of-Failure Based Modeling and Lifetime Evaluation / National Aeronautics and Space Administration. Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109, 2008. – Forschungsbericht

[WGK08] WANG, Gang ; GONG, Wenrui ; KASTNER, Ryan ; COUSSY, Philippe (Hrsg.) ; MORAWIEC, Adam (Hrsg.): *High Level Synthesis - From Algorithm to Digital Circuit*. Springer Netherlands, 2008. http://dx.doi.org/10.1007/978-1-4020-8588-8_13. http://dx.doi.org/10.1007/978-1-4020-8588-8_13

[WV96] WANG, Qi ; VRUDHULA, Sarma B. K.: Multi-level logic optimization for low power using local logic transformations. In: *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design* (1996), S. 270–277

[ZFMS05] ZHONG, Hongtao ; FAN, Kevin ; MAHLKE, Scott ; SCHLANSKER, Michael: A Distributed Control Path Architecture for VLIW Processors. In: *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)* (2005), S. 197–206

Publikationen mit eigener Beteiligung

[KUEV12a] KOAL, T. ; ULBRICHT, M. ; ENGELKE, P. ; VIERHAUS, H. T.: Logic Self Repair Architecture with Self Test Capabilities. In: *ITG/GI/GMM Arbeitstagung Zuverlässigkeit und Entwurf (ZuE)* (2012)

[KUEV12b] KOAL, T. ; ULBRICHT, M. ; ENGELKE, P. ; VIERHAUS, H. T.: Selbstreparatur für Logik-Baugruppen mit erweiterten Fähigkeiten für die Kompensation von Fertigungsfehlern und Frühausfällen. In: *Dresdner Arbeitstagung für Schaltungs- und Systementwurf (DASS)* (2012)

[KUEV13a] KOAL, T. ; ULBRICHT, M. ; ENGELKE, P. ; VIERHAUS, H. T.: Kombinierte On-Line-Fehlerkompensation und Selbstreparatur für Logik-Baugruppen. In: *ITG-GI-GMM Arbeitstagung Test und Zuverlässigkeit von Schaltungen und Systemen (TuZ)* (2013)

[KUEV13b] KOAL, T. ; ULBRICHT, M. ; ENGELKE, P. ; VIERHAUS, H. T.: On-Line-Test, Fehlerkorrektur und Selbstreparatur mit Time-Shared TMR. In: *Dresdner Arbeitstagung für Schaltungs- und Systementwurf (DASS)* (2013)

[KUEV13c] KOAL, T. ; ULBRICHT, M. ; ENGELKE, P. ; VIERHAUS, H. T.: On the Feasibility of Combining On-Line-Test and Self Repair for Logic Circuits. In: *16th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)* (2013)

[KUV12] KOAL, T. ; ULBRICHT, M. ; VIERHAUS, H. T.: Combining On-Line Fault Detection and Logic Self Repair. In: *15. IEEE Int. Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)* (2012)

[KUV13a] KOAL, T. ; ULBRICHT, M. ; VIERHAUS, H. T.: Fault Tolerance and Self Repair Using a Virtual TMR Scheme. In: *VDI/VDE/GMM Fachtagung Zuverlässigkeit und Entwurf (ZuE)* (2013)

[KUV13b] KOAL, T. ; ULBRICHT, M. ; VIERHAUS, H. T.: Virtual TMR Schemes Combining Fault Tolerance and Self Repair. In: *Euromicro Conference on Digital Systems Design (DSD)* (2013)

[UKV12a] ULBRICHT, Markus ; KOAL, Tobias ; VIERHAUS, Heinrich T.: Activity Migration in M-aus-N-Systemen mit Hilfe des Load Balancing. In: *24. GI/GMM/ITG-Workshop: Testmethoden und Zuverlässigkeit von Schaltungen und Systemen* (2012)

[UKV12b] ULBRICHT, Markus ; KOAL, Tobias ; VIERHAUS, Heinrich T.: Activity Migration in M-of-N-Systems by Means of Load Balancing. In: *15th Euromicro Conference on Digital System Design* (2012)

[USKV11a] ULBRICHT, Markus ; SCHÖLZEL, Mario ; KOAL, Tobias ; VIERHAUS, Heinrich T.: A New Hierarchical Built-In Self-Test with On-Chip Diagnosis for VLIW Processor. In: *23. ITG/ GI/GMM Workshop: Testmethoden und Zuverlässigkeit von Schaltungen und Systemen (TuZ)* (2011)

[USKV11b] ULBRICHT, Markus ; SCHÖLZEL, Mario ; KOAL, Tobias ; VIERHAUS, Heinrich T.: A new hierarchical built-in self-test with on-chip diagnosis for VLIW processors. In: *14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)* (2011), S. 143–146

[UV13] ULBRICHT, Markus ; VIERHAUS, Heinrich T.: Different Implementations for Destressing k out of n:G Systems. In: *Proceedings of the Work in Progress Session, Euromicro Conference on Digital Systems Design (DSD 2013)* (2013)

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Anhang **A**

Verwendete Hilfsmittel

- **Synthese, Simulation:**
  - **Hardwarebeschreibungssprache:**
    VHDL 2002
  - **Synthese, Ermittlung der Leistung:**
    Encounter(R) RTL Compiler
    Cadence
    Version: v09.10-s242_1
  - **Simulation, Debugging:**
    ModelSim
    Mentor Graphics
    Version: SE-64 10.0c
    Revision: 2011.07
    Datum: 21.07.2011
  - **Floorplan, Temperatursimulation:**
    HotSpot
    Department of Computer Science, University of Virginia, USA
    Version: 5.0
    http://lava.cs.virginia.edu/HotSpot/
    Siehe: [SSS+04]
- **Scripting, Berechnungen, Plotten:**
  - **Berechnung der Zuverlässigkeit, Plotten**
    GNU Octave
    John W. Eaton and others
    Version: 3.2.4
    http://www.octave.org
  - **Scripting**
    perl
    Larry Wall

Version: v5.12.4
http://www.perl.org/

- **Erstellen der Arbeit:**

  - **Textsatz**
    pdfTeX
    Han The Thanh
    Version: 3.1415926-1.40.10-2.2 (TeX Live 2009/Debian)

  - **LaTeX Editor**
    Kile
    the Kile Team (2003 - 2011)
    Version: 2.1.0
    http://kile.sourceforge.net

  - **Erzeugen der Graphiken**
    LibreOffice Draw
    LibreOffice und/oder deren Tochtergesellschaften, Oracle
    Version 3.4.4, OOO340m1 (Build:402)
    http://de.libreoffice.org/

Opcodes des VLIW Prozessors

```
 1 nop
 2     0000  0000  000000  000000  000000  NOP
 3
 4 arithmetisch
 5     0000  0001  a X c mov Rc <- Ra
 6     0000  0010  a b c inc Rc <- Ra + b
 7     0000  0011  a b c dec Rc <- Ra - b
 8     0000  0100  a b c add Rc <- Ra + Rb
 9     0000  0101  a b c adc Rc <- Ra + Rb + 1
10     0000  0110  a b c sub Rc <- Ra - Rb
11     0000  0111  a b c sbb Rc <- Ra - Rb - 1
12     0000  1000  a b c and Rc <- Ra and Rb
13     0000  1001  a b c or Rc <- Ra or Rb
14     0000  1010  a b c xor Rc <- Ra xor Rb
15     0000  1011  a b c cmp Rc <- Ra =? Rc
16     0000  1100  a X c shr Rc <- Ra>>
17     0000  1101  a X c shl Rc <- <<Ra
18     0000  1110  a X c rrc Rc <- >Ra>
19     0000  1110  a X c rlc Rc <- <Ra<
20     0001  0000  a X c cml Rc <- not Ra
21     0001  0011  X X c mpc Rc <- PC
22     0001  1110  a b c mul Rc <- Ra(16-0) * Rb(16-0)
23
24 sprung
25     0001  0001  a b X jz PC <-(Rb(0) == 1)?-- Ra
26     0001  0010  a b X jNz PC <-(Rb(0) == 0)?-- Ra
27     0001  0100  a X X jmp PC <- Ra
28     0001  0101  a b X jmpt PC <-(Rb != 0)?-- Ra
29     0001  0110  a b X jmpf PC <-(Rb == 0)?-- Ra
30     0001  0111  a b X js PC <-(Rb(1) == 1)?-- Ra
31     0001  1000  a b X jNs PC <-(Rb(1) == 0)?-- Ra
32     0001  1001  a b X jc PC <-(Rb(2) == 1)?-- Ra
33     0001  1011  a b X jNc PC <-(Rb(2) == 0)?-- Ra
34     0001  1100  a b X jo PC <-(Rb(3) == 1)?-- Ra
35     0001  1101  a b X jNo PC <-(Rb(3) == 0)?-- Ra
36     1110  k k k X PC <- k
37
38 load / store DatS
39     1100  1000  a X c ld0 Rc <- M[Ra]
```

```
40      1100   0000   a X c ld1 Rc <- M[Ra]
41      1100   1001   a b X st0 M[Ra] <- Rb
42      1100   0001   a b X st1 M[Ra] <- Rb
43
44 load const
45      1111   00k_7 - k_0 b c ldc Rc <- Rb(31...8) & k
46      1111   01k_7 - k_0 b c ldc Rc <- Rb(31...16) & k & Rb(7...0)
47      1111   10k_7 - k_0 b c ldc Rc <- Rb(31...24) & k & Rb(15...0)
48      1111   11k_7 - k_0 b c ldc Rc <- k & Rb(23...0)
```

Quelltext B.1: Opcodes des VLIW Prozessors

VHDL-Implementierung der Buckets

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  ENTITY Bucket IS
5  PORT(
6        rst         : IN STD_LOGIC;
7        switchBit   : IN STD_LOGIC;
8        setBit      : IN STD_LOGIC;
9        A1          : IN STD_LOGIC;
10       S1          : IN STD_LOGIC;
11       Z1          : OUT STD_LOGIC;
12       Z2          : OUT STD_LOGIC
13 );
14 END Bucket;
15
16 ARCHITECTURE behave OF Bucket IS
17
18       SIGNAL content : STD_LOGIC;
19
20 BEGIN
21
22       PROCESS (rst, switchBit)
23       BEGIN
24
25       IF rst = '1' THEN
26            content <= setBit;
27       ELSE
28            IF switchBit'event AND switchBit = '1' THEN
29                 IF S1 = '1' AND content = '0' AND A1 = '1' THEN
30                      content <= A1;
31                 ELSIF S1 = '1' AND content = '1' THEN
32                      content <= '1';
33                 ELSE
34                      content <= A1;
35                 END IF;
36            END IF;
37       END IF;
38
39 END PROCESS;
```

```
40
41 Z1 <= (A1 AND content) OR (content AND NOT S1);
42 Z2 <= S1 OR content;
43
44 END behave;
```

Quelltext C.1: VHDL-Code der Buckets

Hardwareaufwand der 5 Implementierungen

Tabelle D.1.: Hardwareaufwand Implementierung 1

BS	MaNS (4o8)		MaNSAM (4o8)	
# Zellen	# Zellen		# Zellen	
28992		41500		41582
	# Zellen redundante HW		# Zellen redundante HW	
	rFE 1	2388	rFE 1	2388
	rFE 2	2375	rFE 2	2375
	rFE 3	2349	rFE 3	2349
	rFE 4	2369	rFE 4	2369
	# Zellen Schalter und		# Zellen Schalter, DL,	
	Dekodierlogik		Buckets	
	SchalterMaNSRein	1821	SchalterMaNSRein	1821
	SchalterMaNSRaus	445	SchalterMaNSRaus	445
	Dekodierlogik	215	Dekodierlogik	215
			Buckets	82
	HW overhead zum BS in %		HW overhead zum BS in %	
		43,14		43,42
			HW overhead zum KoNs in %	
				0,2

Tabelle D.2.: Hardwareaufwand Implementierung 2

BS	MaNS (4o8)		MaNSAM (4o8)	
# Zellen	# Zellen		# Zellen	
28992		41628		41628
	# Zellen redundante HW		# Zellen redundante HW	
	rFR 1	27	rFR 1	27
	rFR 2	27	rFR 2	27
	rFR 3	27	rFR 3	27
	rFR 4	27	rFR 4	27
	rDR 1	103	rDR 1	100
	rDR 2	92	rDR 2	99
	rDR 3	99	rDR 1	100
	rDR 4	105	rDR 2	101
	rFE 1	2237	rFE 1	2114
	rFE 2	2350	rFE 2	2256
	rFE 1	2238	rFE 1	2255
	rFE 2	2259	rFE 2	2248
	rWR 1	40	rWR 1	40
	rWR 2	40	rWR 2	40
	rWR 3	40	rWR 3	40
	rWR 4	40	rWR 4	40
	# Zellen Schalter und		# Zellen Schalter, DL,	
	Dekodierlogik		Buckets	
	SchalterMaNSRein	1942	SchalterMaNSRein	1942
	SchalterMaNSRaus	744	SchalterMaNSRaus	744
	Dekodierlogik	214	Dekodierlogik	257
			Buckets	82
	HW overhead zum BS in %		HW overhead zum BS in %	
		43,58		43,58
			HW overhead zum KoNs in %	
				0,00

Tabelle D.3.: Hardwareaufwand Implementierung 3

BS		MaNS (4o8)		MaNSAM (4o8)	
# Zellen		# Zellen		# Zellen	
28992		60337		60557	
		# Zellen redundante HW		# Zellen redundante HW	
		rFR 1	27	rFR 1	27
		rFR 2	27	rFR 2	27
		rFR 3	27	rFR 3	27
		rFR 4	27	rFR 4	27
		rDR 1	100	rDR 1	102
		rDR 2	101	rDR 2	92
		rDR 3	99	rDR 3	103
		rDR 4	100	rDR 4	103
		rFE 1	2114	rFE 1	2231
		rFE 2	2129	rFE 2	2353
		rFE 3	2251	rFE 3	2245
		rFE 4	2139	rFE 4	2248
		rWR 1	72	rWR 1	72
		rWR 2	72	rWR 2	72
		rWR 3	72	rWR 3	72
		rWR 4	72	rWR 4	72
LPs	11744	rLPs	24924	rLPs	24924
SPs	5007	rSPs	12235	rSPs	12235
		# Zellen Schalter und Dekodierlogik		# Zellen Schalter, DL, Buckets	
		SchalterMaNSRein	978	SchalterMaNSRein	978
		Dekodierlogik	205	Dekodierlogik	249
				Buckets	82
		HW overhead zum BS in %		HW overhead zum BS in %	
		108,12		108,87	
				HW overhead zum KoNs in %	
				0,36	

Tabelle D.4.: Hardwareaufwand Implementierung 4

BS		MaNS (4o8)		MaNSAM (4o8)	
# Zellen		# Zellen		# Zellen	
116313		167625		168346	
		# Zellen redundante HW		# Zellen redundante HW	
		rPipeline 1	10400	rPipeline 1	10266
		rPipeline 2	10087	rPipeline 2	10375
		rPipeline 3	10257	rPipeline 3	10258
		rPipeline 4	10154	rPipeline 4	10144
		# Zellen Schalter und Dekodierlogik		# Zellen Schalter, DL, Buckets	
		SchalterMaNSRein	6082	SchalterMaNSRein	6082
		SchalterMaNSRaus	3481	SchalterMaNSRaus	3487
		Dekodierlogik	214	Dekodierlogik	269
				Buckets	82
		HW overhead zum BS in %		HW overhead zum BS in %	
		44,12		44,74	
				HW overhead zum KoNs in %	
				0,43	

Tabelle D.5.: Hardwareaufwand Implementierung 5

BS		MaNS (4o8)		MaNSAM (4o8)	
# Zellen		# Zellen		# Zellen	
116313		236992		344701	
		# Zellen redundante HW		# Zellen redundante HW	
		rPipeline 1	10410	rPipeline 1	10336
		rPipeline 2	10180	rPipeline 2	10442
		rPipeline 3	10448	rPipeline 3	10322
		rPipeline 4	10327	rPipeline 4	10214
		rLP 1	11744	rLP 1	11744
		rLP 2	11744	rLP 2	11744
		rLP 3	11744	rLP 3	11744
		rLP 4	11744	rLP 4	11744
		rRB 1	2049	rRB 1	2049
		rRB 2	2049	rRB 2	2049
		rRB 1	2049	rRB 1	2049
		rRB 2	2049	rRB 2	2049
		rSP 1	5007	rSP 1	5004

	rSP 2	5007	rSP 2	5004
	rSP 3	5007	rSP 3	5004
	rSP 4	5007	rSP 4	5004
	# Zellen Schalter und Dekodierlogik		# Zellen Schalter, DL, Buckets, Copy	
	SchalterMaNSRein	2289	SchalterMaNSRein	2288
	SchalterMaNSRaus	848	SchalterMaNSRaus	848
	Dekodierlogik	214	Dekodierlogik	263
			Buckets	82
			Copy (32 Register)	107109
	HW overhead zum BS in %		HW overhead zum BS in %	
		103,75		196,36
			HW overhead zum KoNs in %	
				45,45

Tabelle D.6.: Wachstum der administrativen Komponenten des MaNS in # Zellen

		#Red 1	2	3	4
SchalterMaNSRein	# Orig 1	318	477	641	810
	2	569	828	1090	1276
	3	818	1087	1363	1734
	4	1069	1346	1720	1941
SchalterMaNSRaus	# Orig 1	55	114	173	186
	2	104	228	346	372
	3	155	342	519	558
	4	206	456	692	744
Dekodierlogik	# Orig 1	4	9	22	32
	2	12	46	48	89
	3	15	60	102	141
	4	19	92	148	214

Tabelle D.7.: Wachstum der administrativen Komponenten des MaNS-AM in # Zellen

		#Red 1	2	3	4
SchalterMaNSRein	# Orig 1	313	477	641	810
	2	562	828	1090	1276
	3	813	1087	1363	1734
	4	1064	1346	1720	19416
SchalterMaNSRaus	# Orig 1	51	114	173	186
	2	102	228	346	372
	3	153	342	519	558
	4	204	456	692	744
Dekodierlogik	# Orig 1	9	14	34	47
	2	20	37	68	116
	3	28	75	120	176
	4	35	111	179	257
Buckets	# Orig 1	22	32	42	52
	2	32	42	52	62
	3	42	52	62	72
	4	52	62	72	82

Tabelle D.8.: Wachstum der Hardware für die Kopieroperation in # Zellen

	# Regs 8	16	32	64
MaN: 1a2	790	1065	2121	4241
1a3	1323	2657	5243	10473
1a4	1860	3868	14539	29112
1a5	2156	4231	16650	33275

Anhang E

Detaillierte Ergebnisse der Simulationen (Fläche, Leistung)

Tabelle E.1.: Fläche (um^2) und Leistung (nW) der Komponenten der Implementierungen in 45nm

	Fläche $me-9$	NoAM $We-9$	AM $We-9$		Fläche $me-12$	NoAM $We-9$	AM $We-9$
Implementierung 1				Implementierung 4			
1a1S				1a1S			
BUFs[0]_aluBuf	755	1468882	1465548,763	ctrlPath	6360	2384994,096	2209720,599
BUFs[1]_aluBuf	857	3999150,43	3974140,964	intoMoNSwitch	22296	44076720,241	46951561,186
BUFs[2]_aluBuf	857	3776049,62	3782541,083	outofMoNSwitch	18152	39649107,892	42904130,625
BUFs[3]_aluBuf	857	3899750,455	3897566,055	regs[0]_readPorts	96073	170313409,859	171173316,289
DECs[0]_decodeReg	1615	3529470,498	3529032,682	regs[0]_registerFile	56656	43810301,879	43842497,417
DECs[1]_decodeReg	1611	3631196,969	3630717,065	regs[0]_writePorts	25117	10678914,167	10837639,811
DECs[2]_decodeReg	1604	3645725,698	3645597,831	regs[1]_readPorts	96019	173814455,699	175607851,927
DECs[3]_decodeReg	1609	3607181,934	3607276,389	regs[1]_registerFile	56656	43764315,856	43791958,851
FETCHs[0]_Fetch	146	221299,906	221355,673	regs[1]_writePorts	25117	10771806,645	10813008,502
FETCHs[1]_Fetch	146	221100,1	221114,871	regs[2]_readPorts	97782	172530804,91	173281549,401
FETCHs[2]_Fetch	146	220993,845	221188,259	regs[2]_registerFile	56656	43798631,897	43836382,182
FETCHs[3]_Fetch	146	222052,771	222385,725	regs[2]_writePorts	25117	10900096,38	10961516,545
ctrlPath	2206	1308834,33	1298651,092	regs[3]_readPorts	97433	174583603,943	174523318,264
intoMoNSwitch	9408	8834189,41	11050560,16	regs[3]_registerFile	56656	43819653,878	43855473,312
memoryAddressRegister	1869	1285875,386	1221428,468	regs[3]_writePorts	25117	11083438,384	11130199,919
memoryBufferRegister	1938	1160929,009	1185757,55	theBucketController	112	4069,48	31062,946
outofMoNSwitch	2500	3827174,064	4036145,859	theSwitchController	569	511327,423	447630,103
readPorts	68011	137977376,744	137977376,744	MaNS			
registerFile	56656	41537590,122	41547201,197	pipes[0]_Pipeline	163937	398914945,168	217501159,82
theBucketController	112	4087,599	29959,844	pipes[1]_Pipeline	166598	23471320,485	223710870,116
theSwitchController	293	128294,753	131266,995	pipes[2]_Pipeline	164334	397353420,024	216494870,615
writePorts	25124	10843991,563	10843991,563	pipes[3]_Pipeline	162672	23200402,503	215383097,76
MaNS				pipes[4]_Pipeline	164153	398574247,465	217172138,028
FUs[0]_FU	41516	127511385,779	69960636,104	pipes[5]_Pipeline	165299	23393092,148	221633990,88
FUs[1]_FU	41442	2117626,551	68807150,227	pipes[6]_Pipeline	164123	398509201,771	217218707,148
FUs[2]_FU	40918	132399782,945	71798303,083	pipes[7]_Pipeline	165904	23400410,407	222760528,08
FUs[3]_FU	42159	2222449,771	72263103,003	Implementierung 5			
RFUs[4]_FU	42198	131298959,068	71486464,008	1a1S			
RFUs[5]_FU	41993	2170694,64	74669922,749	copyCat	586746		136882903,164
RFUs[6]_FU	41295	131289326,937	71314182,354	ctrlPath	6266	2206518,248	2204745,806
RFUs[7]_FU	42416	2195935,521	74566944,301	intoMoNSwitch	5001	2159958,645	3719535,531
Implementierung 2				outofMoNSwitch	4211	73635,109	233397,126
1a1S				theBucketController	112	4033,142	31148,675
ctrlPath	2385	1347731,646	1317748,492	theSwitchController	568	509766,14	442277,791
intoMoNSwitch	7028	8530800,774	9810139,715	MaNS			
memoryAddressRegister	1730	1095143,789	1093515,547	pipes[0]_Pipeline	162905	438672651,093	251328872,726
memoryBufferRegister	1866	1381622,704	1305039,26	pipes[1]_Pipeline	160734	23106037,399	246799130,117
outofMoNSwitch	2850	9265909,086	9916640,658	pipes[2]_Pipeline	162653	433156382,692	248293344,959
readPorts	59250	20782251,894	20820914,643	pipes[3]_Pipeline	161119	23105020,349	246687783,736
registerFile	56656	45767629,458	45771092,398	pipes[4]_Pipeline	162118	438664069,422	251343266,361
theBucketController	112	4051,37	32394,008	pipes[5]_Pipeline	161964	23202698,033	249107064,182
theSwitchController	569	533938,335	467546,515	pipes[6]_Pipeline	161708	429880143,597	246326604,566
writePorts	25114	11683076,676	11722719,367	pipes[7]_Pipeline	161543	23141156,462	246829239,446
MaNS				regs[0]_readPorts	64262	68589041,033	39932595,064
BUFs[0]_aluBuf	755	1664643,286	1317214,248	regs[0]_registerFile	56656	44440870,795	43100019,653
BUFs[1]_aluBuf	755	1096466,447	1328204,907	regs[0]_writePorts	25505	17902860,919	9583976,659
BUFs[2]_aluBuf	755	1732173,037	1356979,279	regs[1]_readPorts	64262	3018025,339	39616699,395
BUFs[3]_aluBuf	755	1103290,25	1409424,725	regs[1]_registerFile	56656	72285006,911	47223877,453
BUFs[4]_aluBuf	755	1847027,137	1407226,141	regs[1]_writePorts	25495	897535,823	9580900,507
BUFs[5]_aluBuf	755	1097082,488	1367348,956	regs[2]_readPorts	64262	68624609,859	39969932,041
BUFs[6]_aluBuf	755	1686726,672	1324373,749	regs[2]_registerFile	56656	44659606,789	42979888,716
BUFs[7]_aluBuf	755	1096213,481	1300253,817	regs[2]_writePorts	25500	18889546,532	10064229,949
DECs[0]_decodeReg	1636	4048260,368	3118709,299	regs[3]_readPorts	64262	3018025,339	39639743,77

DECs[1]_decodeReg	1637	2321194,087	3162560,235
DECs[2]_decodeReg	1651	4036917,485	3112912,299
DECs[3]_decodeReg	1638	2321232,437	3154680,814
DECs[4]_decodeReg	1637	4076388,869	3137970,391
DECs[5]_decodeReg	1639	2321456,338	3159382,298
DECs[6]_decodeReg	1637	4038809,874	3136812,477
DECs[7]_decodeReg	1636	2320755,633	3138720,152
FETCHs[0]_Fetch	146	227667,414	200414,331
FETCHs[1]_Fetch	146	211717,255	202236,844
FETCHs[2]_Fetch	146	236820,492	204857,212
FETCHs[3]_Fetch	146	211706,748	207672,768
FETCHs[4]_Fetch	146	239132,016	206099,644
FETCHs[5]_Fetch	146	211706,561	204543,543
FETCHs[6]_Fetch	146	229127,037	199219,276
FETCHs[7]_Fetch	146	211706,561	199675,397
FUs[0]_FU	39099	121583377,079	63694041,354
FUs[1]_FU	38856	1943721,04	63958063,305
FUs[2]_FU	39180	122045989,969	63865512,625
FUs[3]_FU	38669	1926850,69	61892339,631
FUs[4]_FU	39819	125698516,42	66001644,211
FUs[5]_FU	39751	1999373,87	66020826,341
FUs[6]_FU	38987	120332358,861	63471826,507
FUs[7]_FU	39569	1999165,608	65019382,179
Implementierung 3			
1a1S			
ctrlPath	2463	1562355,841	1494879,418
intoMoNSwitch	2785	653390,371	1328901,654
memoryAddressRegister	1797	1268845,513	1194334,336
memoryBufferRegister	1846	1172920,087	1170330,177
registerFile	56656	47518676,338	47520407,984
theBucketController	112	4051,37	32667,797
theSwitchController	562	530110,04	400654,463
MaNS			
BUFs[0]_aluBuf	755	2756751,413	1900259,11
BUFs[1]_aluBuf	755	1108144,766	1991811,27
BUFs[2]_aluBuf	903	5215545,108	3213384,992
BUFs[3]_aluBuf	911	1135474,5	3347431,562
BUFs[4]_aluBuf	891	4295031,995	2712195,331
BUFs[5]_aluBuf	755	1108131,286	2081057,279
BUFs[6]_aluBuf	903	5231759,741	3232881,036
BUFs[7]_aluBuf	755	1108131,286	2299590,202
DECs[0]_decodeReg	1637	4035787,506	3070036,771
DECs[1]_decodeReg	1639	2320872,763	3082260,144
DECs[2]_decodeReg	1639	4086007,966	3094804,08
DECs[3]_decodeReg	1651	2322568,56	3067214,725
DECs[4]_decodeReg	1637	4054528,708	3080833,031
DECs[5]_decodeReg	1652	2322974,508	3114422,145
DECs[6]_decodeReg	1651	4052319,887	3100229
DECs[7]_decodeReg	1653	2323032,906	3065455,451
FETCHs[0]_Fetch	146	238601,333	206638,8
FETCHs[1]_Fetch	146	211721,48	205387,805
FETCHs[2]_Fetch	146	238533,267	205575,102
FETCHs[3]_Fetch	146	211706,748	205594,397
FETCHs[4]_Fetch	146	236624,523	204560,276
FETCHs[5]_Fetch	146	211706,561	206739,03
FETCHs[6]_Fetch	146	236151,167	203492,409
FETCHs[7]_Fetch	146	211706,561	205233,615
FUs[0]_FU	39262	122535933,779	64285509,928
FUs[1]_FU	39420	1964428,916	64039013,86
FUs[2]_FU	39239	121824972,507	63615006,918
FUs[3]_FU	39778	1991852,745	65772107,106
FUs[4]_FU	39165	121373575,891	63778056,51
FUs[5]_FU	39509	1983356,264	64729975,352
FUs[6]_FU	39191	119021226,451	62921890,821
FUs[7]_FU	39314	1967834,26	64068527,016
readPorts[0]	14937,375	21464965,146	21489220,835
readPorts[1]	14937,375	21464965,146	21489220,835
readPorts[2]	14937,375	21464965,146	21489220,835
readPorts[3]	14937,375	21464965,146	21489220,835
readPorts[4]	14937,375	21464965,146	21489220,835
readPorts[5]	14937,375	21464965,146	21489220,835
readPorts[6]	14937,375	21464965,146	21489220,835
readPorts[7]	14937,375	21464965,146	21489220,835
writePorts[0]	6637,25	17409191,767	18233356,624
writePorts[1]	6637,25	17409191,767	18233356,624
writePorts[2]	6637,25	17409191,767	18233356,624
writePorts[3]	6637,25	17409191,767	18233356,624
writePorts[4]	6637,25	17409191,767	18233356,624
writePorts[5]	6637,25	17409191,767	18233356,624
writePorts[6]	6637,25	17409191,767	18233356,624
writePorts[7]	6637,25	17409191,767	18233356,624
regs[3]_registerFile	56656	72282011,662	43605865,49
regs[3]_writePorts	25494	897116,746	10045545,508
regs[4]_readPorts	64262	68633457,564	39979707,654
regs[4]_registerFile	56656	44713787,226	43058368,831
regs[4]_writePorts	25494	18687027,043	9965433,958
regs[5]_readPorts	64261	3017908,547	39639045,078
regs[5]_registerFile	56656	72282011,662	43632779,1
regs[5]_writePorts	25485	898264,723	10042170,832
regs[6]_readPorts	64261	68641517,915	39988908,212
regs[6]_registerFile	56656	45743899,775	43487397,841
regs[6]_writePorts	25485	18830382,027	10042316,344
regs[7]_readPorts	64261	3017908,547	39644483,716
regs[7]_registerFile	56656	72282011,662	47523875,41
regs[7]_writePorts	25488	897478,017	10023747,127
Basissystem 1, Cluster (B)			
BUFs[0]_aluBuf	755	1648818,707	1648818,707
BUFs[1]_aluBuf	857	4527347,007	4527347,007
BUFs[2]_aluBuf	857	4276496,343	4276496,343
BUFs[3]_aluBuf	857	4462580,198	4462580,198
DECs[0]_decodeReg	1656	4756544,597	4756544,597
DECs[1]_decodeReg	1655	4777728,476	4777728,476
DECs[2]_decodeReg	1660	4809693,041	4809693,041
DECs[3]_decodeReg	1658	4865591,083	4865591,083
FETCHs[0]_Fetch	146	247940,086	247940,086
FETCHs[1]_Fetch	146	248474,347	248474,347
FETCHs[2]_Fetch	146	250741,542	250741,542
FETCHs[3]_Fetch	146	247406,359	247406,359
ctrlPath	1881	1295698,486	1295698,486
memoryAddressRegister	1245	1140068,632	1140068,632
memoryBufferRegister	1260	1166201,05	1166201,05
readPorts	68439	154557568,746	154557568,746
registerFile	56656	47090404,589	47090404,589
writePorts	25124	12151878,822	12151878,822
FUs[0]_FU	41559	119142798,531	119142798,531
FUs[1]_FU	42411	121124497,004	121124497,004
FUs[2]_FU	42089	124104987,139	124104987,139
FUs[3]_FU	41552	121034787,263	121034787,263
Basissystem 4, Cluster (C)			
ctrlPath	3975	1595379,36	1595379,36
regs[0]_readPorts	68055	145422998,212	145422998,212
regs[0]_registerFile	56656	45669060,857	45669060,857
regs[0]_writePorts	25124	11529068,158	11529068,158
regs[1]_readPorts	68055	145422998,212	145422998,212
regs[1]_registerFile	56656	45641000,413	45641000,413
regs[1]_writePorts	25124	11529065,886	11529065,886
regs[2]_readPorts	68055	145422998,212	145422998,212
regs[2]_registerFile	56656	45685602,076	45685602,076
regs[2]_writePorts	25124	11529068,158	11529068,158
regs[3]_readPorts	68055	145422998,212	145422998,212
regs[3]_registerFile	56656	45668325,057	45668325,057
regs[3]_writePorts	25124	11529068,158	11529068,158
pipes[0]_Pipeline	170209	459961574,739	459961574,739
pipes[1]_Pipeline	170103	458081782,607	458081782,607
pipes[2]_Pipeline	170903	460674829,332	460674829,332
pipes[3]_Pipeline	168524	450854321,604	450854321,604

Tabelle E.2.: Fläche (um^2) und Leistung (nW) der Komponenten der Implementierungen in 130nm

	Fläche $me-9$	NoAM $We-9$	AM $We-9$		Fläche $me-12$	NoAM $We-9$	AM $We-9$
Implementierung 1				Implementierung 4			
1a1S				1a1S			
BUFs[0]_aluBuf	3999	2720570,553	2710119,644	ctrlPath	86875	2486532,306	2202151,883
BUFs[1]_aluBuf	3999	2743282,602	2744053,448	intoMoNSwitch	210783	23681390,071	29613889,021
BUFs[2]_aluBuf	3999	2654863,198	2656265,303	outofMoNSwitch	124371	24620260,308	29010578,777
BUFs[3]_aluBuf	3999	2685404,861	2683409,376	regs[0]_readPorts	541929	25139089,662	27385039,922
DECs[0]_decodeReg	9575	6319187,262	6316535,106	regs[0]_registerFile	325684	78275977,254	78349305,555
DECs[1]_decodeReg	9473	6057040,055	6058910,209	regs[0]_writePorts	252057	36987463,005	38480642,141
DECs[2]_decodeReg	9548	6061261,603	6061222,169	regs[1]_readPorts	542110	25199203,85	26401313,291
DECs[3]_decodeReg	9521	6234842,672	6235886,599	regs[1]_registerFile	325684	78269409,699	78386069,804
FETCHs[0]_Fetch	775	390561,421	390561,421	regs[1]_writePorts	252053	37017631,733	38555651,186
FETCHs[1]_Fetch	775	390348,22	390348,22	regs[2]_readPorts	542397	24902708,024	26155873,302
FETCHs[2]_Fetch	775	390050,804	390050,804	regs[2]_registerFile	325684	78292199,65	78368041,087
FETCHs[3]_Fetch	775	390557,002	390557,002	regs[2]_writePorts	251946	37003861,913	38448295,008
ctrlPath	24287	1362136,737	1455886,635	regs[3]_readPorts	542262	25067450,631	26487178,429
intoMoNSwitch	135982	23851414,799	29192100,356	regs[3]_registerFile	325684	78259971,378	78377704,21
memoryAddressRegister	11963	1097714,974	1099003,319	regs[3]_writePorts	252050	37013438,244	38522379,651
memoryBufferRegister	13455	1097913,361	1100590,845	theBucketController	706	3,056	39165,167
outofMoNSwitch	25659	5850559,284	6111412,974	theSwitchController	3593	610786,123	689984,624
readPorts	567734	88927400,01	88927400,01	MaNS			
registerFile	325684	72216492,086	72216492,086	pipes[0]_Pipeline	1729628	814415089,397	432492832,485
theBucketController	706	2,898	36294,524	pipes[1]_Pipeline	1788252	19507085,947	449011543,912
theSwitchController	2162	155296,461	209702,517	pipes[2]_Pipeline	1726218	814430389,333	432384655,741
writePorts	252638	34258280,906	34258280,906	pipes[3]_Pipeline	1749480	19513279,33	436812656,96
MaNS				pipes[4]_Pipeline	1741761	817889202,664	434265558,039
FUs[0]_FU	435610	229266046,361	124853198,827	pipes[5]_Pipeline	1741917	19560805,436	433526341,25
FUs[1]_FU	499975	1072,53	151260348,843	pipes[6]_Pipeline	1763376	841672637,398	446495653,161
FUs[2]_FU	491463	253792833,042	137332953,045	pipes[7]_Pipeline	1743981	19562284,222	438160185,053
FUs[3]_FU	493441	1044,934	132321577,101	Implementierung 5			
RFUs[4]_FU	498348	287564213,63	150211186,228	1a1S			
RFUs[5]_FU	488091	1024,054	142967371,056	copyCat	5657114		183218921,548
RFUs[6]_FU	498656	258439875,241	139544235,748	ctrlPath	84913	1386795,513	1442343,549
RFUs[7]_FU	464104	960,404	131313840,832	intoMoNSwitch	44889	2072487,997	4594709,831
Implementierung 2				outofMoNSwitch	37593	745,797	497256,338
1a1S				theBucketController	706	3,056	38965,812
ctrlPath	26772	1855623,007	1742194,783	theSwitchController	3453	593790,179	659988,245
intoMoNSwitch	64171	7017278,923	9140963,45	MaNS			
memoryAddressRegister	12035	1266969,118	1267230,476	pipes[0]_Pipeline	1838796	885669702,769	498057586,066
memoryBufferRegister	13597	1266983,096	1266998,185	pipes[1]_Pipeline	1833082	19160891,35	497431398,548
outofMoNSwitch	22184	6704120,15	7768502,94	pipes[2]_Pipeline	1802788	862634665,041	485239414,114
readPorts	541231	27249492,216	27922901,808	pipes[3]_Pipeline	1815809	19580540,613	496209038,854
registerFile	325684	83245061,03	83251411,126	pipes[4]_Pipeline	1823894	881098184,158	495477393,708
theBucketController	706	3,202	41704,297	pipes[5]_Pipeline	1824081	19651371,07	495843045,853
theSwitchController	3639	649215,196	729771,017	pipes[6]_Pipeline	1874784	909950531,752	511349663,117
writePorts	252375	39950482,983	41517148,199	pipes[7]_Pipeline	1901768	19359272,884	518106888,141
MaNS				regs[0]_readPorts	561040	67758702,767	36827351,648
BUFs[0]_aluBuf	3999	2293953,202	1796504,173	regs[0]_registerFile	325684	77722812,53	77323125,783
BUFs[1]_aluBuf	3999	1306559,609	1831832,507	regs[0]_writePorts	254587	39789702,291	20480636,675
BUFs[2]_aluBuf	3999	2490809,56	1896817,574	regs[1]_readPorts	560673	562,13	36410727,301
BUFs[3]_aluBuf	3999	1306553,756	1955612,787	regs[1]_registerFile	325684	85878805,672	77581427,81
BUFs[4]_aluBuf	3999	2617697,005	1962901,753	regs[1]_writePorts	252006	182,41	22283348,551
BUFs[5]_aluBuf	3999	1306550,376	1901010,444	regs[2]_readPorts	560553	67649668,171	36766372,739
BUFs[6]_aluBuf	3999	2418090,338	1860892,766	regs[2]_registerFile	325684	77706708,059	75437931,697
BUFs[7]_aluBuf	4017	1308813,063	1797585,644	regs[2]_writePorts	272666	43411170,94	22282398,743
DECs[0]_decodeReg	10009	7252491,476	5194533,885	regs[3]_readPorts	560553	561,662	36396980,611
DECs[1]_decodeReg	10058	3015676,369	5297719,553	regs[3]_registerFile	325716	85875624,947	75795927,697
DECs[2]_decodeReg	9950	7234397,191	5137222,204	regs[3]_writePorts	276044	279,439	23630027,759
DECs[3]_decodeReg	10001	3016742,258	5234026,057	regs[4]_readPorts	560553	67638641,596	36757071,537
DECs[4]_decodeReg	10078	7276539,984	5217827,535	regs[4]_registerFile	325684	77726311,448	77354068,025
DECs[5]_decodeReg	10056	3014983,67	5276312,436	regs[4]_writePorts	273635	44741284,289	22951951,875
DECs[6]_decodeReg	10063	7345761,51	5283735,049	regs[5]_readPorts	560553	561,662	36393375,637
DECs[7]_decodeReg	10043	3016526,994	5268216,038	regs[5]_registerFile	325684	85875624,671	75800686,197
FETCHs[0]_Fetch	775	420242,695	336413,233	regs[5]_writePorts	272775	268,018	22541763,289
FETCHs[1]_Fetch	775	253082,774	341585,322	regs[6]_readPorts	560553	67636200,215	36756463,848
FETCHs[2]_Fetch	775	442132,503	347744,188	regs[6]_registerFile	325693	77574030,693	75353754,255
FETCHs[3]_Fetch	775	253072,209	354084,856	regs[6]_writePorts	272931	43898820,219	22532366,067
FETCHs[4]_Fetch	775	453326,368	354029,911	regs[7]_readPorts	560553	561,662	36391574,535
FETCHs[5]_Fetch	775	253062,349	348136,775	regs[7]_registerFile	325716	85875624,619	75502942,846
FETCHs[6]_Fetch	775	431311,473	342033,834	regs[7]_writePorts	263592	232,276	22203228,086
FETCHs[7]_Fetch	775	253062,349	336130,16	Basissystem 1, Cluster (B)			
FUs[0]_FU	462268	237497457,167	123113998,314	BUFs[0]_aluBuf	3999	3206407,406	3206407,406
FUs[1]_FU	445925	1456,763	120817107,747	BUFs[1]_aluBuf	3999	3256632,098	3256632,098
FUs[2]_FU	483532	248716613,474	128122339,04	BUFs[2]_aluBuf	3999	3130237,128	3130237,128
FUs[3]_FU	474382	1065,137	126937334,686	BUFs[3]_aluBuf	4003	3166490,289	3166490,289
FUs[4]_FU	476958	250825255,222	130382226,039	DECs[0]_decodeReg	10016	7373826,604	7373826,604
FUs[5]_FU	456985	955,672	123518171,395	DECs[1]_decodeReg	9978	7403474,556	7403474,556
FUs[6]_FU	449809	233254163,766	121699516,613	DECs[2]_decodeReg	9980	7332123,686	7332123,686
FUs[7]_FU	450095	936,308	120418626,241	DECs[3]_decodeReg	10106	7542878,233	7542878,233

Implementierung 3			
1a1S			
ctrlPath	28297	1904354,371	1744583,257
intoMoNSwitch	23127	649462,07	1719234,282
memoryAddressRegister	12043	1262696,084	1265584,915
memoryBufferRegister	13597	1266988,96	1266998,915
registerFile	384704	59649452,006	59654736,34
theBucketController	706	3,202	42930,455
theSwitchController	3588	646795,678	708289,255
MaNS			
BUFs[0]_aluBuf	4003	2330028,773	1816860,721
BUFs[1]_aluBuf	3999	1313990,505	1875993,907
BUFs[2]_aluBuf	4003	2319469,914	1810830,314
BUFs[3]_aluBuf	3999	1313981,557	2470859,909
BUFs[4]_aluBuf	4003	3186461,853	2266736,801
BUFs[5]_aluBuf	3999	1313977,265	2115988,419
BUFs[6]_aluBuf	3999	2706863,983	2019460,342
BUFs[7]_aluBuf	3999	1313977,265	2597822,577
DECs[0]_decodeReg	9908	7304597,996	5194118,635
DECs[1]_decodeReg	10081	3016215,777	5279209,141
DECs[2]_decodeReg	10114	7523956,74	5306144,855
DECs[3]_decodeReg	9919	2913133,743	5173881,447
DECs[4]_decodeReg	9988	7336981,208	5177054,774
DECs[5]_decodeReg	9985	2907735,406	5200948,282
DECs[6]_decodeReg	10078	7401453,584	5272386,13
DECs[7]_decodeReg	9982	2911954,622	5128252,197
FETCHs[0]_Fetch	775	451585,488	353664,075
FETCHs[1]_Fetch	775	253084,244	352774,135
FETCHs[2]_Fetch	775	453718,753	354112,26
FETCHs[3]_Fetch	775	253072,209	351111,858
FETCHs[4]_Fetch	775	449751,773	352021,055
FETCHs[5]_Fetch	775	253062,349	354294,878
FETCHs[6]_Fetch	775	451109,971	353064,567
FETCHs[7]_Fetch	775	253062,349	352833,612
FUs[0]_FU	440462	220835916,568	114713903,481
FUs[1]_FU	442693	1375,03	118064024,177
FUs[2]_FU	459723	233376550,912	120481819,324
FUs[3]_FU	452814	1007,741	121402281,391
FUs[4]_FU	449799	234819751,808	122254083,014
FUs[5]_FU	442981	921,51	119673237,996
FUs[6]_FU	435148	222703911,472	116173004,035
FUs[7]_FU	456393	948,4	121225870,169
readPorts[0]	143431,125	26517341,054	26539333,352
readPorts[1]	143431,125	26517341,054	26539333,352
readPorts[2]	143431,125	26517341,054	26539333,352
readPorts[3]	143431,125	26517341,054	26539333,352
readPorts[4]	143431,125	26517341,054	26539333,352
readPorts[5]	143431,125	26517341,054	26539333,352
readPorts[6]	143431,125	26517341,054	26539333,352
readPorts[7]	143431,125	26517341,054	26539333,352
writePorts[0]	63491,75	43856635,764	45017061,567
writePorts[1]	63491,75	43856635,764	45017061,567
writePorts[2]	63491,75	43856635,764	45017061,567
writePorts[3]	63491,75	43856635,764	45017061,567
writePorts[4]	63491,75	43856635,764	45017061,567
writePorts[5]	63491,75	43856635,764	45017061,567
writePorts[6]	63491,75	43856635,764	45017061,567
writePorts[7]	63491,75	43856635,764	45017061,567
FETCHs[0]_Fetch	775	467569,233	467569,233
FETCHs[1]_Fetch	775	463914,862	463914,862
FETCHs[2]_Fetch	775	466663,244	466663,244
FETCHs[3]_Fetch	775	464193,593	464193,593
ctrlPath	16450	1522866,925	1522866,925
memoryAddressRegister	8401	1294078,222	1294078,222
memoryBufferRegister	8417	1294846,977	1294846,977
readPorts	525111	26039106,097	26039106,097
registerFile	325684	84809433,839	84809433,839
writePorts	252235	40481862,177	40481862,177
FUs[0]_FU	462453	237705434,781	237705434,781
FUs[1]_FU	461031	239551832,26	239551832,26
FUs[2]_FU	470605	246345070,362	246345070,362
FUs[3]_FU	460932	241212848,896	241212848,896
Basissystem 4, Cluster (C)			
ctrlPath	46931	1902374,295	1902374,295
regs[0]_readPorts	621320	161839676,892	161839676,892
regs[0]_registerFile	325684	81230600,449	81230600,449
regs[0]_writePorts	252627	38220645,95	38220645,95
regs[1]_readPorts	621269	161817207,82	161817207,82
regs[1]_registerFile	325684	81230560,748	81230560,748
regs[1]_writePorts	252627	38219744,354	38219744,354
regs[2]_readPorts	621224	161786324,25	161786324,25
regs[2]_registerFile	325684	81230596,223	81230596,223
regs[2]_writePorts	252627	38221297,909	38221297,909
regs[3]_readPorts	621146	161754523,586	161754523,586
regs[3]_registerFile	325684	81230600,918	81230600,918
regs[3]_writePorts	252627	38223545,758	38223545,758
pipes[0]_Pipeline	1833250	909449100,651	909449100,651
pipes[1]_Pipeline	1885042	949032905,932	949032905,932
pipes[2]_Pipeline	1883922	939988202,453	939988202,453
pipes[3]_Pipeline	1804594	895598861,642	895598861,642

Tabelle E.3.: Fläche (um^2) und Leistung (nW) der Komponenten der Implementierungen in 250nm

	Fläche $me-9$	NoAM $We-9$	AM $We-9$		Fläche $me-12$	NoAM $We-9$	AM $We-9$
Implementierung 1				Implementierung 4			
1a1S				1a1S			
BUFs[0]_aluBuf	20970	9146079,795	9042780,784	ctrlPath	234944	14581574,254	12370929,487
BUFs[1]_aluBuf	20970	26859604,59	26848597,787	intoMoNSwitch	980692	249253040,856	321520641,334
BUFs[2]_aluBuf	20970	47298733,485	47244958,346	outofMoNSwitch	732215	371018194,287	411377757,935
BUFs[3]_aluBuf	20970	30910575,684	30890100,66	regs[0]_readPorts	2790923	167444521,944	168365414,08
DECs[0]_decodeReg	43790	22661026,826	22661026,826	regs[0]_registerFile	1739290	329261251,915	328923543,637
DECs[1]_decodeReg	43790	23202802,166	23202802,166	regs[0]_writePorts	1175642	226577146,628	233972060,301
DECs[2]_decodeReg	43790	22258109,479	22258109,479	regs[1]_readPorts	2796504	158577233,948	161899850,633
DECs[3]_decodeReg	43790	23330212,912	23330212,912	regs[1]_registerFile	1739706	329210823,017	329859283,62
FETCHs[0]_Fetch	4050	1693595,301	1693595,301	regs[1]_writePorts	1180822	243967600,335	250179500,257
FETCHs[1]_Fetch	4050	1659507,487	1659507,487	regs[2]_readPorts	2776861	179383973,577	180276031,602
FETCHs[2]_Fetch	4050	1701862,849	1701862,849	regs[2]_registerFile	1739600	330397656,424	330549220,91
FETCHs[3]_Fetch	4050	1703235,659	1703235,659	regs[2]_writePorts	1173053	229482855,31	238016036,462
ctrlPath	93654	14127203,279	12081264,346	regs[3]_readPorts	2787649	169687785,63	172046101,895
intoMoNSwitch	352998	103172210,106	131961023,061	regs[3]_registerFile	1739720	331279803,105	330344770,406
memoryAddressRegister	69600	4081516,049	4247151,214	regs[3]_writePorts	1174471	231930988,683	239831000,681
memoryBufferRegister	70687	4843331,527	4246418,127	theBucketController	3154	5,111	216789,2
outofMoNSwitch	120135	49821697,45	50233315,917	theSwitchController	17584	1732266,048	2591184,855
readPorts	2873253	687790263,452	687790263,452	MaNS			
registerFile	1738669	322647877,888	322647877,888	pipes[0]_Pipeline	4800564	2568261816,13	1371381807,7
theBucketController	3098	5,032	203347,411	pipes[1]_Pipeline	4820638	50777696,663	1390758940,77
theSwitchController	9377	415044,272	914688,261	pipes[2]_Pipeline	4782331	2566198664,43	1369539833,94
writePorts	1154510	182364049,867	182364049,867	pipes[3]_Pipeline	4810710	50726107,001	1383069598,65
MaNS				pipes[4]_Pipeline	4890697	2605092094,13	1388599184,69
FUs[0]_FU	1116182	598753632,26	334069019,278	pipes[5]_Pipeline	4812552	50947524,78	1380397590,46
FUs[1]_FU	1141915	1218,216	362199475,222	pipes[6]_Pipeline	4856560	2587969786,26	1254012111,99
FUs[2]_FU	1134816	625782521,735	347502355,852	pipes[7]_Pipeline	4817096	51337753,206	1373155927,75
FUs[3]_FU	1188682	1268,943	425201027,355	Implementierung 5			
RFUs[4]_FU	1195604	635082030,175	352343419,824	1a1S			
RFUs[5]_FU	1173349	1229,463	357004929,5	copyCat	28078936		1431150863,78
RFUs[6]_FU	1159477	592712365,098	301494170,887	ctrlPath	231846	13017506,842	12478606,928
RFUs[7]_FU	1138754	1198,532	326059026,176	intoMoNSwitch	215801	19790328,436	45548285,136
Implementierung 2				outofMoNSwitch	200052	4932,445	2682785,147
1a1S				theBucketController	3154	5,111	215762,737
ctrlPath	91227	6751912,602	6045848,792	theSwitchController	17252	1664597,074	2490340,926
intoMoNSwitch	286029	63395401,773	82340481,067	MaNS			
memoryAddressRegister	69177	3676322,737	3891773,564	pipes[0]_Pipeline	4693940	2641609770,84	1478538501,45
memoryBufferRegister	70927	4580636,703	4196469,503	pipes[1]_Pipeline	4714170	51032375,3	1468192677,05
outofMoNSwitch	134544	104839722,706	114461752,932	pipes[2]_Pipeline	4777935	2611827128,61	1463565589,52
readPorts	2790168	167002664,591	168865634,364	pipes[3]_Pipeline	4746522	50928842,313	1473563905,18
registerFile	1738528	309144086,718	309223033,135	pipes[4]_Pipeline	4798546	2645165251,81	1479614286,16
theBucketController	3098	4,993	198046,5	pipes[5]_Pipeline	4799661	50798767,236	1470180899,68
theSwitchController	17273	1624002,084	2471011,475	pipes[6]_Pipeline	4726447	2626050508,14	1331903491,35
writePorts	1170139	188579611,62	194773360,994	pipes[7]_Pipeline	4725608	50980472,532	1456228617,56
MaNS				regs[0]_readPorts	3153432	1500233909,5	843396401,345
BUFs[0]_aluBuf	20970	8013108,482	5634546,408	regs[0]_registerFile	1737258	334830903,925	323529363,803
BUFs[1]_aluBuf	20970	2995260,514	5861219,108	regs[0]_writePorts	1160035	209531880,89	105942310,456
BUFs[2]_aluBuf	20970	9198800,4	6223743,666	regs[1]_readPorts	3298673	3243,04	841543444,58
BUFs[3]_aluBuf	20970	2885981,871	6651094,476	regs[1]_registerFile	1734640	210181337,141	319590149,294
BUFs[4]_aluBuf	20970	9784497,843	6477268,479	regs[1]_writePorts	1163880	1135,229	100933635,775
BUFs[5]_aluBuf	20970	2970934,028	6232453,32	regs[2]_readPorts	3352482	1501959831,28	846075440,07
BUFs[6]_aluBuf	20970	8608383,771	5613026,632	regs[2]_registerFile	1734873	326411421,557	315954553,456
BUFs[7]_aluBuf	20970	2983072,921	5526374,481	regs[2]_writePorts	1162010	216986840,926	110376225,857
DECs[0]_decodeReg	43790	27489388,603	17751374,004	regs[3]_readPorts	3217564	3140,659	846365235,166
DECs[1]_decodeReg	43790	6783185,019	17798533,486	regs[3]_registerFile	1734873	210181337,462	344976344,618
DECs[2]_decodeReg	43790	28111099,823	18077364,225	regs[3]_writePorts	1162497	1133,984	98997819,263
DECs[3]_decodeReg	43790	6783131,561	18875786,245	regs[4]_readPorts	3186885	1501463421,95	831430333,942
DECs[4]_decodeReg	43790	27236150,079	17556773,634	regs[4]_registerFile	1734718	331001936,126	317767757,682
DECs[5]_decodeReg	43790	6783055,549	17878684,576	regs[4]_writePorts	1164042	210864880,186	105994391,204
DECs[6]_decodeReg	43790	27593573,858	16817878,423	regs[5]_readPorts	3155521	3066,8	821503333,673
DECs[7]_decodeReg	43790	6783055,549	17469539,048	regs[5]_registerFile	1734097	210181336,387	310581196,773
FETCHs[0]_Fetch	4050	1525673,756	1127149,908	regs[5]_writePorts	1163026	1134,138	104642739,731
FETCHs[1]_Fetch	4050	625480,672	1145048,028	regs[6]_readPorts	3179321	1520571067,21	769576186,402
FETCHs[2]_Fetch	4050	1661810,68	1200037,867	regs[6]_registerFile	1734097	329923850,341	316099669,377
FETCHs[3]_Fetch	4050	625436,964	1220355,066	regs[6]_writePorts	1164550	206749996,518	93944035,241
FETCHs[4]_Fetch	4050	1710295,776	1226385,621	regs[7]_readPorts	3162189	3075,152	827142488,319
FETCHs[5]_Fetch	4050	625388,257	1184238,453	regs[7]_registerFile	1735106	210181337,786	316035689,289
FETCHs[6]_Fetch	4050	1566785,963	1104026,02	regs[7]_writePorts	1162977	1134,632	104371976,836
FETCHs[7]_Fetch	4050	625388,257	1132095,066	Basissystem 1, Cluster (B)			
FUs[0]_FU	1130463	571019111,813	299654494,437	BUFs[0]_aluBuf	20970	20724142,598	20724142,598
FUs[1]_FU	1139953	3569,493	301202456,998	BUFs[1]_aluBuf	20970	31443508,733	31443508,733
FUs[2]_FU	1122038	566195651,591	295957204,567	BUFs[2]_aluBuf	20970	13346365,336	13346365,336
FUs[3]_FU	1148844	1630,128	325494988,35	BUFs[3]_aluBuf	20970	14753829,467	14753829,467
FUs[4]_FU	1152273	570739088,365	300342399,942	DECs[0]_decodeReg	43790	29993082,974	29993082,974
FUs[5]_FU	1146297	1210,89	303125035,027	DECs[1]_decodeReg	43790	31025300,423	31025300,423
FUs[6]_FU	1159654	567182821,876	270999231,647	DECs[2]_decodeReg	43790	30447564,273	30447564,273
FUs[7]_FU	1117064	1188,714	296633677,216	DECs[3]_decodeReg	43790	30514808,035	30514808,035

Implementierung 3			
1a1S			
ctrlPath	91545	5672064,206	6039846,254
intoMoNSwitch	88510	4915939,794	12564527,173
memoryAddressRegister	68232	3191474,055	4536648,91
memoryBufferRegister	70094	4435976,408	4750797,113
registerFile	1742056	322334888,183	320015794,095
theBucketController	3098	4,993	206572,147
theSwitchController	16941	1659123,153	2523719,068
MaNS			
BUFs[0]_aluBuf	20970	27140991,387	15704154,792
BUFs[1]_aluBuf	20970	3089461,7	15018772,925
BUFs[2]_aluBuf	20970	23247330,034	13594644,993
BUFs[3]_aluBuf	20970	3186538,524	20016755,144
BUFs[4]_aluBuf	20970	21759294,292	12728722,597
BUFs[5]_aluBuf	20970	3016568,563	7954646,865
BUFs[6]_aluBuf	20970	39327050,986	20290260,014
BUFs[7]_aluBuf	20970	3247207,534	24338140,856
DECs[0]_decodeReg	43790	26972788,435	17247175,865
DECs[1]_decodeReg	43790	6783181,313	17576789,723
DECs[2]_decodeReg	43790	27667016,235	17594822,64
DECs[3]_decodeReg	43790	6783129,779	17324244,571
DECs[4]_decodeReg	43790	27336224,336	17488663,516
DECs[5]_decodeReg	43790	6783055,549	17499584,662
DECs[6]_decodeReg	43790	26785017,424	17263134,489
DECs[7]_decodeReg	43790	6783055,549	17303078,026
FETCHs[0]_Fetch	4050	1722924,776	1235056,636
FETCHs[1]_Fetch	4050	625490,217	1222640,172
FETCHs[2]_Fetch	4050	1703870,631	1222942,31
FETCHs[3]_Fetch	4050	625436,964	1294587,4
FETCHs[4]_Fetch	4050	1716302,301	1229630,291
FETCHs[5]_Fetch	4050	625388,257	1220046,474
FETCHs[6]_Fetch	4050	1723196,607	1182033,982
FETCHs[7]_Fetch	4050	625388,257	1231624,247
FUs[0]_FU	1171091	580665212,611	298550846,719
FUs[1]_FU	1161785	3564,347	304793802,732
FUs[2]_FU	1178719	579711906,218	297244397,86
FUs[3]_FU	1161093	1626,264	297994796,626
FUs[4]_FU	1162737	592153209,797	305601195,906
FUs[5]_FU	1160931	1226,625	306125192,33
FUs[6]_FU	1170174	588486175,327	301110668,696
FUs[7]_FU	1147912	1211,042	293821338,14
readPorts[0]	676149,125	175239235,78	174439669,582
readPorts[1]	676149,125	175239235,78	174439669,582
readPorts[2]	676149,125	175239235,78	174439669,582
readPorts[3]	676149,125	175239235,78	174439669,582
readPorts[4]	676149,125	175239235,78	174439669,582
readPorts[5]	676149,125	175239235,78	174439669,582
readPorts[6]	676149,125	175239235,78	174439669,582
readPorts[7]	676149,125	175239235,78	174439669,582
writePorts[0]	312439,625	179616430,175	178633286,721
writePorts[1]	312439,625	179616430,175	178633286,721
writePorts[2]	312439,625	179616430,175	178633286,721
writePorts[3]	312439,625	179616430,175	178633286,721
writePorts[4]	312439,625	179616430,175	178633286,721
writePorts[5]	312439,625	179616430,175	178633286,721
writePorts[6]	312439,625	179616430,175	178633286,721
writePorts[7]	312439,625	179616430,175	178633286,721
FETCHs[0]_Fetch	4050	2359685,366	2359685,366
FETCHs[1]_Fetch	4050	2373226,241	2373226,241
FETCHs[2]_Fetch	4050	2388679,191	2388679,191
FETCHs[3]_Fetch	4050	2387083,79	2387083,79
ctrlPath	61176	8098343,388	8098343,388
memoryAddressRegister	47127	6248483,959	6248483,959
memoryBufferRegister	48044	6778606,407	6778606,407
readPorts	2699089	164707711,342	164707711,342
registerFile	1739177	399455740,126	399455740,126
writePorts	1156161	260046620,684	260046620,684
FUs[0]_FU	1160648	667606882,256	667606882,256
FUs[1]_FU	1160289	671720559,802	671720559,802
FUs[2]_FU	1169447	680022577,739	680022577,739
FUs[3]_FU	1157565	655100840,323	655100840,323
Basissystem 4, Cluster (C)			
ctrlPath	137712	12667291,898	12667291,898
regs[0]_readPorts	2700867	170924196,678	170924196,678
regs[0]_registerFile	1738429	344018047,473	344018047,473
regs[0]_writePorts	1156845	232284452,474	232284452,474
regs[1]_readPorts	2700867	170924196,678	170924196,678
regs[1]_registerFile	1738358	344279000,831	344279000,831
regs[1]_writePorts	1157918	232344166,899	232344166,899
regs[2]_readPorts	2700867	170924196,678	170924196,678
regs[2]_registerFile	1738302	344397424,062	344397424,062
regs[2]_writePorts	1156189	232683635,207	232683635,207
regs[3]_readPorts	2700867	170924196,678	170924196,678
regs[3]_registerFile	1738288	344159187,823	344159187,823
regs[3]_writePorts	1157720	230249707,812	230249707,812
pipes[0]_Pipeline	4926422	2735708281,48	2735708281,48
pipes[1]_Pipeline	4950059	2743461538,41	2743461538,41
pipes[2]_Pipeline	4960650	2729608838,58	2729608838,58
pipes[3]_Pipeline	4957306	2752501663,98	2752501663,98

Anhang **F**

Detaillierte Ergebnisse der Simulationen (Anzahl Zellen, Temperatur)

Tabelle F.1.: Anzahl Technologiezellen und Temperaturen (K) der Komponenten aller Implementierungen in 45, 130 und 250nm

	45nm			130nm			250nm		
	# Zellen	NoAM (K)	AM (K)	# Zellen	NoAM (K)	AM (K)	# Zellen	NoAM (K)	AM (K)
Implementierung 1									
1a1S									
BUFs[0]_aluBuf	136	579,90	579,13	136	362,68	362,32	136	360,69	361,29
BUFs[1]_aluBuf	264	579,05	579,78	136	362,54	362,35	136	359,30	360,84
BUFs[2]_aluBuf	264	579,13	579,75	136	362,25	362,51	136	360,65	361,44
BUFs[3]_aluBuf	264	579,09	579,79	136	362,33	362,57	136	361,32	361,20
DECs[0]_decodeReg	356	580,10	579,31	283	362,59	362,45	283	359,66	360,46
DECs[1]_decodeReg	350	580,01	579,28	283	362,37	362,61	283	359,78	360,33
DECs[2]_decodeReg	336	579,09	579,75	283	362,25	362,37	283	360,90	361,09
DECs[3]_decodeReg	346	579,14	579,71	283	362,38	362,63	283	359,86	360,40
FETCHs[0]_Fetch	27	579,17	579,72	27	362,32	362,35	27	359,86	360,35
FETCHs[1]_Fetch	27	579,17	579,72	27	362,32	362,35	27	360,12	361,18
FETCHs[2]_Fetch	27	579,17	579,72	27	362,31	362,49	27	360,09	360,74
FETCHs[3]_Fetch	27	579,17	579,72	27	362,62	362,33	27	360,55	361,14
ctrlPath	1179	578,77	579,60	1660	362,41	362,29	2461	359,06	360,54
intoMoNSwitch	8954	579,75	579,47	8806	361,94	362,49	11085	359,84	360,51
memoryAddressRegister	1145	579,03	579,71	918	362,68	362,38	1704	359,54	360,27
memoryBufferRegister	1222	578,71	579,60	931	362,82	362,37	1734	359,69	360,77
outofMoNSwitch	2221	579,02	579,75	2653	362,46	362,42	3754	359,90	360,80
readPorts	51008	579,52	579,64	48482	362,09	362,09	90158	359,73	360,10
registerFile	8193	579,30	578,92	9217	362,10	362,15	8321	359,42	359,80
theBucketController	82	579,17	579,72	65	362,26	362,45	82	360,00	360,62
theSwitchController	224	580,03	579,18	167	362,14	362,43	251	359,81	360,27
writePorts	17736	579,29	578,86	25341	362,05	362,18	34676	359,38	359,76
MaNS									
FUs[0]_FU	40461	579,17	579,65	29709	362,96	362,55	24535	361,07	360,39
FUs[1]_FU	40664	578,63	579,68	33723	361,68	362,64	24473	358,75	360,47
FUs[2]_FU	40272	580,28	579,23	32873	362,81	362,59	24312	361,20	360,44
FUs[3]_FU	41883	578,28	579,65	33196	361,52	362,56	25080	358,60	360,77
RFUs[4]_FU	42063	579,91	579,44	33492	363,45	362,61	25007	361,21	360,33
RFUs[5]_FU	41780	578,69	579,58	32803	361,81	362,56	24672	358,48	360,52
RFUs[6]_FU	39899	580,24	579,44	33436	363,10	362,59	24463	360,87	360,38
RFUs[7]_FU	41407	578,35	579,68	31286	361,81	362,59	24254	358,55	360,34
Implementierung 2									
1a1S									
ctrlPath	1362	579,81	579,26	1778	361,94	362,16	2375	358,36	358,86
intoMoNSwitch	6248	579,52	578,60	6457	362,07	362,05	9414	359,62	359,56
memoryAddressRegister	928	580,13	579,03	927	362,37	362,21	1707	358,57	359,03
memoryBufferRegister	1136	578,27	578,69	925	362,27	362,52	1749	358,53	358,89
outofMoNSwitch	2126	580,07	578,91	2181	362,32	362,24	3839	360,59	360,41
readPorts	45481	577,35	577,79	45786	361,44	361,67	88999	357,96	358,14
registerFile	8193	578,07	577,83	9217	361,95	361,99	8313	358,47	358,63
theBucketController	82	580,23	579,06	65	362,20	362,57	82	358,14	358,84
theSwitchController	279	580,00	578,94	222	362,21	362,24	311	358,47	359,03
writePorts	17722	579,06	578,18	25423	362,06	362,10	34659	358,84	358,86
MaNS									
BUFs[0]_aluBuf	136	580,18	579,03	136	362,06	362,53	136	360,22	359,94
BUFs[1]_aluBuf	136	580,05	579,33	136	362,26	362,25	136	360,11	359,74
BUFs[2]_aluBuf	136	579,78	578,76	136	362,61	362,31	136	359,28	359,44
BUFs[3]_aluBuf	136	579,89	579,39	136	362,48	362,28	136	359,80	359,61
BUFs[4]_aluBuf	136	580,13	579,33	136	362,57	362,30	136	360,42	360,12
BUFs[5]_aluBuf	136	579,85	579,25	136	362,23	362,57	136	360,11	359,87
BUFs[6]_aluBuf	136	580,14	579,00	136	362,57	362,26	136	359,31	359,43

BUFs[7]_aluBuf	136	579,94	579,23	137	362,57	362,30	136	360,04	359,62
DECs[0]_decodeReg	392	579,88	579,37	298	362,83	362,39	283	359,86	359,46
DECs[1]_decodeReg	391	578,11	578,62	299	362,29	362,55	283	358,77	359,19
DECs[2]_decodeReg	417	579,98	579,28	298	362,00	362,11	283	360,28	359,95
DECs[3]_decodeReg	392	579,57	578,64	298	362,19	362,54	283	358,46	359,14
DECs[4]_decodeReg	392	579,72	579,27	298	362,74	362,36	283	359,67	359,52
DECs[5]_decodeReg	393	580,21	579,08	300	362,13	362,56	283	360,02	359,73
DECs[6]_decodeReg	391	579,87	579,35	298	362,14	362,56	283	360,35	359,90
DECs[7]_decodeReg	392	580,15	579,01	298	362,62	362,33	283	358,46	359,20
FETCHs[0]_Fetch	27	578,04	578,53	27	362,20	362,57	27	358,25	358,80
FETCHs[1]_Fetch	27	579,81	578,83	27	362,11	362,19	27	358,55	359,12
FETCHs[2]_Fetch	27	578,04	578,53	27	361,92	362,26	27	359,90	359,57
FETCHs[3]_Fetch	27	580,14	579,31	27	362,31	362,53	27	360,36	359,93
FETCHs[4]_Fetch	27	578,19	578,53	27	362,39	362,23	27	360,41	359,98
FETCHs[5]_Fetch	27	579,86	578,88	27	362,31	362,27	27	360,40	360,06
FETCHs[6]_Fetch	27	579,70	579,30	27	362,27	362,56	27	358,54	358,93
FETCHs[7]_Fetch	27	579,88	579,30	27	362,17	362,57	27	358,55	359,09
FUs[0]_FU	37516	580,31	579,16	31661	363,07	362,39	25157	360,27	359,41
FUs[1]_FU	37507	577,26	579,09	30585	361,47	362,28	25598	357,90	359,37
FUs[2]_FU	37619	579,73	578,63	32435	362,87	362,24	24979	359,73	359,41
FUs[3]_FU	37152	577,23	578,27	32014	361,84	362,44	25222	358,81	359,47
FUs[4]_FU	38480	579,82	578,60	32337	363,00	362,41	25053	360,24	359,39
FUs[5]_FU	38492	577,34	579,02	31634	361,59	362,32	25597	358,08	359,38
FUs[6]_FU	37410	579,54	579,05	30748	362,63	362,42	24775	360,45	359,31
FUs[7]_FU	38105	578,15	579,01	30791	361,78	362,37	25026	358,06	359,33
Implementierung 3									
1a1S									
ctrlPath	1425	579,37	578,49	1882	361,29	361,70	2426	357,99	357,56
intoMoNSwitch	2627	578,28	577,70	2688	361,13	361,47	3121	358,14	358,02
memoryAddressRegister	1054	577,57	578,56	927	361,20	361,59	1703	357,85	358,29
memoryBufferRegister	1110	579,37	578,59	925	362,15	361,57	1742	357,21	357,16
registerFile	8193	577,77	577,39	14862	360,93	361,42	8317	357,52	357,67
theBucketController	82	579,00	579,01	65	362,44	361,68	82	357,99	357,97
theSwitchController	264	579,19	578,89	212	361,71	361,36	302	357,73	357,46
MaNS									
BUFs[0]_aluBuf	136	579,28	578,82	136	361,57	361,88	136	358,47	357,95
BUFs[1]_aluBuf	136	579,07	578,98	136	361,47	361,94	136	357,77	358,42
BUFs[2]_aluBuf	264	579,28	578,69	136	361,17	361,50	136	358,17	358,20
BUFs[3]_aluBuf	264	577,72	577,94	136	361,30	361,66	136	357,59	358,38
BUFs[4]_aluBuf	264	579,29	578,87	136	362,19	361,59	136	358,37	358,19
BUFs[5]_aluBuf	136	579,15	578,92	136	362,18	361,59	136	358,39	358,26
BUFs[6]_aluBuf	264	579,19	578,89	136	362,26	361,63	136	358,59	358,60
BUFs[7]_aluBuf	136	579,19	578,88	136	361,40	361,77	136	358,16	358,28
DECs[0]_decodeReg	393	577,90	577,61	298	362,18	361,60	283	358,47	358,24
DECs[1]_decodeReg	393	579,24	578,74	298	361,10	361,40	283	358,70	358,50
DECs[2]_decodeReg	392	579,39	578,74	298	362,00	361,52	283	357,88	357,55
DECs[3]_decodeReg	417	577,61	578,66	298	361,23	361,59	283	358,03	358,31
DECs[4]_decodeReg	392	578,58	577,99	298	361,97	361,49	283	358,48	358,44
DECs[5]_decodeReg	417	579,24	578,91	298	361,58	361,96	283	358,26	358,06
DECs[6]_decodeReg	417	577,68	577,75	299	362,19	361,60	283	358,59	358,37
DECs[7]_decodeReg	418	579,32	578,73	298	361,46	361,89	283	357,80	358,20
FETCHs[0]_Fetch	27	577,66	577,82	27	361,40	361,86	27	358,69	358,40
FETCHs[1]_Fetch	27	579,27	578,77	27	361,94	361,48	27	358,65	358,17
FETCHs[2]_Fetch	27	577,66	577,82	27	361,33	361,71	27	358,55	358,16
FETCHs[3]_Fetch	27	579,19	578,93	27	361,97	361,49	27	357,47	357,33
FETCHs[4]_Fetch	27	579,22	578,81	27	361,42	361,76	27	357,60	358,07
FETCHs[5]_Fetch	27	579,09	578,96	27	361,49	361,94	27	358,50	358,22
FETCHs[6]_Fetch	27	577,80	577,64	27	362,41	361,68	27	358,66	358,19
FETCHs[7]_Fetch	27	579,22	578,81	27	361,46	361,79	27	358,78	358,36
FUs[0]_FU	37565	579,55	578,67	30247	363,01	361,99	25665	358,80	358,18
FUs[1]_FU	38109	577,16	578,55	30817	360,90	361,91	25319	356,73	358,19
FUs[2]_FU	37933	578,69	578,32	31246	362,79	361,96	25820	359,22	358,18
FUs[3]_FU	38500	577,79	578,41	31258	360,87	361,92	26035	357,46	358,31
FUs[4]_FU	37729	579,18	578,38	31036	363,20	362,11	26244	358,99	358,33
FUs[5]_FU	38283	578,85	578,97	30385	360,95	361,94	26019	356,85	358,32
FUs[6]_FU	37635	579,37	578,16	30283	363,02	361,88	25880	359,00	358,31
FUs[7]_FU	37595	577,15	578,69	30858	361,40	361,99	25182	357,21	358,30
readPorts[0]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
readPorts[1]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
readPorts[2]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
readPorts[3]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
readPorts[4]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
readPorts[5]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
readPorts[6]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
readPorts[7]	11376375	577,21	576,87	12597625	360,94	361,00	20991375	356,75	356,83
writePorts[0]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40
writePorts[1]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40
writePorts[2]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40
writePorts[3]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40
writePorts[4]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40
writePorts[5]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40
writePorts[6]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40
writePorts[7]	4558625	577,37	578,12	6157375	361,42	361,28	9198375	357,38	357,40

Implementierung 4									
1a1S									
ctrlPath	4759	577,76	577,30	5976	360,34	360,79	6979	355,84	356,87
intoMoNSwitch	19552	575,46	577,11	19568	360,12	360,76	31480	356,87	357,78
outofMoNSwitch	12515	576,88	577,44	12470	359,86	360,76	21307	358,03	358,15
regs[0]_readPorts	60654	577,59	577,75	45839	359,99	360,46	87627	355,70	356,11
regs[0]_registerFile	8193	576,08	577,11	9217	360,30	360,56	8321	356,02	356,45
regs[0]_writePorts	17727	575,56	577,08	25445	360,35	360,70	34992	356,73	356,84
regs[1]_readPorts	59895	576,98	577,54	45861	360,03	360,29	88866	355,53	355,94
regs[1]_registerFile	8193	577,04	577,38	9217	360,46	360,71	8320	356,35	356,67
regs[1]_writePorts	17727	577,42	577,42	25445	360,61	360,84	34923	356,92	356,94
regs[2]_readPorts	59459	576,41	577,48	45927	359,94	360,39	88696	356,36	356,32
regs[2]_registerFile	8193	575,80	576,90	9217	360,50	360,76	8321	356,65	356,80
regs[2]_writePorts	17727	575,84	576,85	25445	360,66	360,88	34891	357,24	357,10
regs[3]_readPorts	59507	577,59	577,73	45894	360,00	360,47	88748	355,42	355,83
regs[3]_registerFile	8193	577,46	577,27	9217	361,22	360,98	8322	356,07	356,49
regs[3]_writePorts	17727	577,61	577,40	25445	360,61	360,70	34952	356,25	356,69
theBucketController	82	576,57	577,14	65	359,89	360,43	82	357,37	357,13
theSwitchController	277	577,92	577,52	222	361,13	360,93	317	356,09	356,54
MaNS									
pipes[0]_Pipeline	142103	579,14	577,71	118820	362,19	361,21	102776	358,73	357,33
pipes[1]_Pipeline	145452	575,99	577,37	121727	360,06	361,21	103724	355,20	357,58
pipes[2]_Pipeline	143701	579,15	577,59	117899	362,62	361,24	103994	358,94	357,62
pipes[3]_Pipeline	141103	575,70	577,32	120163	359,61	361,14	103343	355,07	357,54
pipes[4]_Pipeline	143570	578,50	577,70	118496	362,48	361,22	104157	358,58	357,53
pipes[5]_Pipeline	145034	574,78	577,14	118451	359,57	361,18	103718	355,56	357,73
pipes[6]_Pipeline	142825	578,08	577,36	119755	362,66	361,24	104483	358,90	357,58
pipes[7]_Pipeline	144159	574,95	577,58	120148	359,53	361,17	103253	355,06	357,68
Implementierung 5									
1a1S									
copyCat	440252		570,94	527615		357,95	858429		350,06
ctrlPath	4844	573,58	571,79	5781	358,95	358,73	6848	355,53	351,18
intoMoNSwitch	4572	574,25	571,23	4433	359,19	358,90	7151	355,71	351,67
outofMoNSwitch	3224	574,68	572,83	3778	359,15	358,82	6366	355,08	350,90
theBucketController	82	574,83	574,42	65	359,06	358,48	82	355,04	351,39
theSwitchController	275	573,57	572,79	222	359,15	358,48	310	356,00	350,73
MaNS									
pipes[0]_Pipeline	142782	577,80	575,52	122238	361,28	359,37	102914	357,49	351,83
pipes[1]_Pipeline	139944	574,87	574,05	121518	358,80	359,26	101391	354,45	351,63
pipes[2]_Pipeline	143576	578,10	575,66	121505	361,42	359,50	101842	357,89	351,77
pipes[3]_Pipeline	140917	574,35	576,39	121221	359,21	359,28	102948	354,15	351,67
pipes[4]_Pipeline	142545	577,38	576,29	121667	361,48	359,54	102561	357,64	351,84
pipes[5]_Pipeline	142914	574,50	575,19	121841	359,58	359,30	103026	354,29	351,85
pipes[6]_Pipeline	141466	578,06	575,86	123457	361,33	359,47	102154	357,76	351,78
pipes[7]_Pipeline	140796	573,92	574,61	125228	358,91	359,54	102728	354,11	351,74
regs[0]_readPorts	50488	573,82	573,76	46475	359,91	358,42	97508	356,97	351,66
regs[0]_registerFile	8193	574,16	572,94	9217	359,95	358,67	8257	355,48	351,24
regs[0]_writePorts	18541	573,45	571,73	23822	359,68	358,52	34641	355,20	350,66
regs[1]_readPorts	50488	575,70	573,83	46475	359,56	358,36	101316	354,15	351,58
regs[1]_registerFile	8193	574,90	571,34	9217	360,28	358,48	8200	354,83	351,26
regs[1]_writePorts	18515	573,69	571,21	25131	360,21	358,41	34685	355,03	350,60
regs[2]_readPorts	50488	575,21	573,72	46475	359,31	358,45	102790	356,97	351,44
regs[2]_registerFile	8193	575,37	573,35	9217	359,67	358,67	8203	356,00	351,20
regs[2]_writePorts	18538	577,76	572,58	24334	359,65	358,82	34685	356,16	350,54
regs[3]_readPorts	50488	574,55	572,64	46475	358,91	358,39	99353	354,07	351,57
regs[3]_registerFile	8193	574,34	572,11	9217	359,43	358,64	8203	355,23	351,05
regs[3]_writePorts	18525	573,91	570,87	24707	359,46	358,19	34648	354,83	350,51
regs[4]_readPorts	50488	575,52	572,70	46475	359,96	358,31	98479	357,26	351,44
regs[4]_registerFile	8193	575,06	572,41	9217	360,30	358,68	8201	356,00	350,78
regs[4]_writePorts	18525	575,73	572,91	24584	360,45	358,61	34671	355,80	350,35
regs[5]_readPorts	50488	573,96	573,73	46475	359,44	358,51	97544	354,01	351,64
regs[5]_registerFile	8193	574,91	572,97	9217	359,88	358,65	8193	354,50	351,16
regs[5]_writePorts	18529	574,68	571,18	24419	359,73	358,40	34685	354,32	350,33
regs[6]_readPorts	50488	575,61	572,29	46475	359,63	358,47	98187	357,48	351,16
regs[6]_registerFile	8193	574,02	571,73	9217	359,78	358,77	8193	355,79	350,50
regs[6]_writePorts	18529	573,52	573,53	24477	359,72	358,51	34680	355,49	350,60
regs[7]_readPorts	50488	574,03	573,15	46475	359,58	358,34	97699	353,98	351,66
regs[7]_registerFile	8193	573,68	572,61	9217	359,85	358,71	8206	354,88	351,05
regs[7]_writePorts	18528	573,36	573,94	24325	360,01	358,67	34662	354,91	350,52
Basissystem 1, Cluster (B)									
BUFs[0]_aluBuf	136	580,56	579,97	136	363,98	363,51	136	362,22	361,41
BUFs[1]_aluBuf	264	581,49	580,70	136	364,19	363,63	136	362,42	361,55
BUFs[2]_aluBuf	264	581,64	580,85	136	363,92	363,47	136	362,24	361,38
BUFs[3]_aluBuf	264	581,06	580,49	136	364,19	363,62	136	362,33	361,39
DECs[0]_decodeReg	411	581,29	580,51	298	364,13	363,60	283	361,41	360,71
DECs[1]_decodeReg	411	581,63	580,85	299	364,16	363,59	283	361,75	361,03
DECs[2]_decodeReg	411	581,04	580,51	299	364,08	363,57	283	362,01	361,21
DECs[3]_decodeReg	411	581,19	580,42	299	363,33	362,92	283	362,16	361,43
FETCHs[0]_Fetch	27	581,58	580,80	27	364,11	363,54	27	361,98	361,23
FETCHs[1]_Fetch	27	581,58	580,79	27	364,16	363,61	27	361,63	360,91
FETCHs[2]_Fetch	27	580,43	579,92	27	364,11	363,54	27	362,08	361,30
FETCHs[3]_Fetch	27	580,43	579,89	27	363,92	363,46	27	361,10	360,50
ctrlPath	1120	581,51	580,73	1069	362,77	362,53	1453	361,66	360,97

memoryAddressRegister	536	580,65	580,09	591	363,22	362,86	1047	361,43	360,80
memoryBufferRegister	548	580,54	579,99	591	364,13	363,56	1056	361,53	360,75
readPorts	51721	580,67	580,03	45420	362,64	362,27	84995	359,96	359,42
registerFile	8193	580,41	579,72	9217	362,71	362,36	8313	360,76	359,99
writePorts	17736	580,76	580,03	25411	362,73	362,42	34907	361,37	360,55
FUs[0]_FU	40647	581,25	580,57	31550	363,99	363,42	25240	362,58	361,68
FUs[1]_FU	41829	581,54	580,84	31811	363,71	363,23	25672	362,88	361,93
FUs[2]_FU	41305	581,26	580,51	32162	363,96	363,44	25703	362,38	361,54
FUs[3]_FU	40968	581,57	580,84	31582	364,08	363,55	25066	362,79	361,91
Basissystem, 4 CLuster (C)									
ctrlPath	3027	582,20	581,67	3102	363,61	363,36	3927	361,70	361,21
regs[0]_readPorts	51085	581,11	580,69	52057	363,85	363,60	84934	360,64	360,17
regs[0]_registerFile	8193	579,46	579,14	9217	363,55	363,31	8321	360,88	360,40
regs[0]_writePorts	17736	579,12	578,88	25344	363,43	363,22	34939	360,93	360,45
regs[1]_readPorts	51085	580,79	580,34	52057	363,92	363,66	84934	360,40	359,94
regs[1]_registerFile	8193	579,21	578,87	9217	363,77	363,52	8321	360,61	360,14
regs[1]_writePorts	17736	578,91	578,65	25344	363,89	363,64	34952	360,77	360,31
regs[2]_readPorts	51085	580,51	580,11	52057	364,09	363,82	84934	360,66	360,19
regs[2]_registerFile	8193	579,25	578,94	9217	363,70	363,44	8319	360,88	360,40
regs[2]_writePorts	17736	579,03	578,79	25344	363,45	363,21	34924	360,79	360,30
regs[3]_readPorts	51085	581,29	580,87	52057	363,73	363,48	84934	360,10	359,64
regs[3]_registerFile	8193	581,07	580,61	9217	363,60	363,36	8321	360,61	360,15
regs[3]_writePorts	17736	581,27	580,77	25344	363,43	363,19	34939	360,82	360,37
pipes[0]_Pipeline	155244	583,48	582,97	123732	364,89	364,59	103183	363,26	362,74
pipes[1]_Pipeline	155152	583,79	583,23	126403	365,24	364,92	104715	363,47	362,89
pipes[2]_Pipeline	155980	582,84	582,30	126434	365,27	364,95	105288	363,41	362,86
pipes[3]_Pipeline	153303	583,19	582,69	123565	365,14	364,84	105328	362,90	362,34