TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Elektrotechnik und Informationstechnik - Institut für Nachrichtentechnik

Diplomarbeit

ANALYSE UND ENTWICKLUNG EINER BEFEHLSSATZERWEITERUNG FÜR DIE BESCHLEUNIGUNG VON BITMAP-INDIZES GROSSER DATENBANKSYSTEME

Erik Laux
Mat.-Nr.: 34 72 79 4

Betreut durch:
Prof. Dr.-Ing. G. Fettweis
Prof. Dr.-Ing. W. Lehner

und:
Dr.-Ing. O. Arnold
Dipl. Inf. T. Karnagel / Dr.-Ing. B. Schlegel

Eingereicht am 31. Juli 2014

TECHNISCHE UNIVERSITAT DRESDEN

FAKULTÄT ELEKTROTECHNIK UND INFORMATIONSTECHNIK

Aufgabenstellung für die Diplomarbeit

für Herrn **Erik Laux** (Mat. Nr. 3472794 / IST 2008)

Thema: **Analyse und Entwicklung einer Befehlssatzerweiterung für die Beschleunigung von Bitmap-Indizes großer Datenbanksysteme**

Zielsetzung:
Die Laufzeit von datenintensiven Anwendungen wird hauptsächlich durch die Taktfrequenz und Architektur des eingesetzten Prozessors bestimmt. Während die Erhöhung der Taktfrequenz durch physikalische Grenzen limitiert ist, kann durch die Anpassung der Architektur an die Anwendung eine hohe Beschleunigung erzielt werden. Typischerweise kann hierbei gleichzeitig eine Reduzierung der benötigten Energie für die Abarbeitung der Anwendung erzielt werden.

In dieser Diplomarbeit soll ein Prozessor durch eine Befehlssatzerweiterung beschleunigt werden. Hierbei wird sowohl das Laden der Daten als auch die eigentliche Berechnung betrachtet. Im ersten Schritt werden Bitmap-Indizes untersucht und ihre Abarbeitung analysiert. Diese sind ein wichtiger Bestandteil aktueller und zukünftiger Datenbanksysteme. Anhand der erzielten Resultate werden neue Befehle in den Prozessor integriert. Eine Beschleunigung ist z. Bsp. möglich durch eine Parallelisierung auf Befehlsebene, als auch durch das explizite Vorladen von Daten. Für eine Parallelisierung stehen mehrere Möglichkeiten zur Verfügung. Sowohl die Zusammensetzung von existierenden Befehlen zu einem neuen Befehl als auch die Anwendung eines Befehles auf mehrere Datenwörter ist denkbar. Des Weiteren ist eine Kombination der Ansätze möglich. Eine Anpassung des Registersatzes und zusätzliche interne Speicher erlauben eine weitere Beschleunigung.

Die vorgeschlagenen Implementierungen sind mit relevanten Leistungsparametern (zum Beispiel Skalierbarkeit, Latenz und Durchsatz) entsprechend zu quantifizieren. Eine Analyse von Beschränkungen des gewählten Ansatzes rundet die Diplomarbeit ab.

Betreuer: Dipl. Ing. Oliver Arnold und
Dipl. Inf. Tomas Karnagel / Dipl. Inf. Benjamin Schlegel
/ Prof. Dr.-Ing W. Lehner

Ausgehändigt am: 1. Februar 2014
Einzureichen bis: 31. Juli 2014

TU Dresden
Prüfungsamt
Fakultät Elektrotechnik und Informationstechnik

Prof. Dr.-Ing. habil. M. Wollschlaeger
Vorsitzender des Prüfungsausschusses

Prof. Dr.-Ing. Dr. h.c. G. Fettweis
Verantwortlicher Hochschullehrer

ERKLÄRUNG

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, 31. Juli 2014

ZUSAMMENFASSUNG

Prozessoren gehören zu den zentralen Elementen in einem Computersystem, wenn es um die Datenverarbeitung geht. Aus diesem Grund ist eine effiziente Konstruktion der Prozessorarchitektur genauso wichtig für die Verarbeitungsleistung, wie die effiziente Nutzung der, durch die Architektur gegebenen Möglichkeiten. In der Vergangenheit wurde die Beschleunigung vor allem durch die kontinuierliche Steigerung der Taktfrequenz erreicht. Aufgrund von physikalischen Aspekten sind diesem Wachstum jedoch zunehmend Grenzen gesetzt. Eine weitere Leistungssteigerung kann jedoch auch durch eine spezifische, an die jeweilige Aufgabe angepasste, Prozessorarchitektur erreicht werden.
Ein besonders exponiertes Gebiet für kontinuierlich notwendige Leitungssteigerungen stellen die Datenbanksysteme dar. Aufgrund der weltweit wachsenden digitalen Datenmenge steigt auch die Notwendigkeit für Datenbanken, immer mehr zu speichern und zu verarbeiten. Für den effizienten Betrieb ist hierbei ein hoher Datensatz genauso wichtig wie geringe Latenzen und eine gute Gesamtperformance, auch in Hinblick auf den Energieverbrauch. Das Potential der Anpassung von Hard- und Software aufeinander wurde in der Vergangenheit bereits in unterschiedlichen Veröffentlichungen beschrieben. Auf dieser Grundlage wird in der vorliegenden Arbeit der *Xtensa LX4* als konfigurierbarer Basisprozessor durch die *Tensilica Instruction Extension* mit aufgabenspezifischen Instruktionen erweitert. Bitmap Indizes stellen in Datenbanksystemen einen Weg zur Beschleunigung der Verarbeitung großer Datenmengen dar. Um hierbei die Menge der zusätzlich notwendigen Daten so gering wie möglich zu halten, werden unterschiedliche Kompressionsalgorithmen eingesetzt, die in der letzten Dekade weiter verbessert wurden, um die Kompressionsleistung zu erhöhen. Untersuchungen der Algorithmenbeschleunigung durch Hardwareanpassungen existieren jedoch bisher nicht. Diese Arbeit dient dem Schließen der Forschungslücke in diesem Bereich. Hierfür wurden, nach der Implementierung unterschiedlicher Algorithmen in C, Möglichkeiten der Beschleunigung identifiziert und durch neu entworfene Instruktionen realisiert. Im Ergebnis werden bei einem Bruchteil des üblichen Verbrauches von Energie und Fläche bei Standardprozessoren Beschleunigungen von bis zu Faktor 108 erreicht.

SUMMARY

Processors belong to the central elements in computer systems regarding data processing. Due to this fact, an efficient design of the processor architecture is as critical for the processing performance as the proper utilization of possibilities is. The acceleration of performance was mainly in the past mainly achieved by an increasing clock frequency. However, because of physical issues, the frequency enhancement becomes more difficult. But beside this possibility a speed optimization can be reached by designing the architecture tailored to the problems.
One of the most critical domains for continuously needed performance enhancements are database systems. Due to the increasing amount of data worldwide, the need for databases for storing and processing more data increases. To ensure an efficient database system, a high data bandwidth is as critical as low latencies and a good overall performance with respect to energy consumption. The potential for adjusting hard- and software to each other has already been proven by different publications in history. In the view of that fact, this diploma thesis will use the *Xtensa LX4* as a configurable processor basis and extend it with tailor-made instructions using the *Tensilica Instruction Extension*. Bitmap Indices are used to accelerate database systems for processing large amounts of data. To minimize the additionally needed data, different algorithms for compression are used. While there have been releases of different compression algorithms with enhanced compression rates in the last decade, there are no investigations on optimizing their performance with specialized hardware. Due to that fact, this thesis will help to close the research gap. To reach this goal, different compressions algorithms were implemented in C and after the identification of bottlenecks, they were replaced by special instructions which were created during the present thesis. This results in a custom processor which needs a fraction of the energy and space normal processors need while accelerating some of the operations by factor 108.

INHALTSVERZEICHNIS

Aufgabenstellung i

Selbständigkeitserklärung ii

Zusammenfassung iii

Summary iv

Inhaltsverzeichnis v

Abkürzungsverzeichnis viii

Abbildungsverzeichnis ix

Tabellenverzeichnis xii

1 Einleitung **1**

1.1 Motivation zur Bearbeitung der Themenstellung 1

1.2 Erläuterung der Aufgabenstellung 2

1.3 Aufbau dieser Arbeit 3

2 Theoretischer Hintergrund **4**

2.1 Rechnerarchitektur 4

2.2 Datenbanken 7

2.3 Bitmap Indizes 8

2.3.1 BBC Komprimierung 11

2.3.2 WAH Komprimierung 13

2.3.3 EWAH Komprimierung 15
2.3.4 PLWAH Komprimierung 15
2.3.5 COMPAX Komprimierung 16
2.3.6 Zusammenfassung 18

3 Tensilica Instruction Extension 20

3.1 Arbeitsablauf 20
3.2 Programmierung von Instruktionen mit TIE 22
3.3 Optimierungstechniken 26
3.3.1 Fusion 26
3.3.2 SIMD 27
3.3.3 FLIX 27
3.3.4 Datenorganisation 28
3.3.5 Hardwareaspekte 29
3.4 Prozessormodell 30

4 Verwandte Arbeiten 31

4.1 Laufzeitoptimierung unter Nutzung von SIMD Instruktionen 31
4.2 Laufzeitoptimierung unter Nutzung von GPUs 34
4.3 Laufzeitoptimierung unter Nutzung von FPGAs 35
4.4 Laufzeitoptimierung unter Nutzung von speziellen Datenbanksystemen 36

5 Implementierung 38

5.1 Allgemeine Betrachtung der TIE Implementierung 39
5.2 WAH-Operation mit TIE-Entwicklungsstufen 41
5.2.1 Beispielhafter Ablauf 41
5.2.2 Ablaufanalyse und TIE Optimierung 41
5.3 PLWAH Operation 50
5.3.1 Beispielhafter Ablauf 50
5.3.2 Ablaufanalyse 51

5.3.3 Optimierung mit TIE 52
5.4 COMPAX Operation 52
5.4.1 Beispielhafter Ablauf 53
5.4.2 Ablaufanalyse 54
5.4.3 Optimierung mit TIE 55
5.5 WAH Komprimierung 55
5.5.1 Beispielhafter Ablauf 56
5.5.2 Ablaufanalyse 57
5.5.3 Optimierung mit TIE 57
5.6 WAH Dekomprimierung 59
5.6.1 Beispielhafter Ablauf 59
5.6.2 Ablaufanalyse 59
5.6.3 Optimierung mit TIE 60
5.7 Limitierungen und Einschränkungen 62

6 Experimentelle Resultate 63
6.1 Prozessormodelle 64
6.2 Synthesedetails 65
6.3 WAH Operation 67
6.3.1 Vergleich der WAH Entwicklungsstufen 67
6.3.2 Messergebnisse der finalen Implementierung 68
6.4 PLWAH Operation 70
6.5 COMPAX Operation 71
6.6 WAH Kompression 72
6.7 WAH Dekompression 74

7 Zusammenfassung und Ausblick 76

A Anhang 78
A.1 WAH - Speedup Messungen 79
A.2 PLWAH - Speedup Messungen 80
A.3 COMPAX - Speedup Messungen 81

ABKÜRZUNGSVERZEICHNIS

SIMD Single Instruction, Multiple Data

HDL Hardware Describtion Language

VLIW Very Long Instruction Word

I/O Input/Output

FLIX Flexible Length Instruction Xtension

CPU Central Processing Unit

OLTP On-Line Transaction Processing

OLAP On-Line Analytical Processing

CPU Central Processing Unit

ISA Instruction Set Architecture

RISC Reduced Instruction Set Computer

CISC Complex Instruction Set Computer

RLE Run-Length Encoding

BBC Byte-Aligned Bitmap Code

WAH Word-Aligned Hybrid Code

PLWAH Position List Word-Aligned Hybrid Code

EWAH Extended Word-Aligned Hybrid Code

COMPAX Compressed Adaptive Index Format

FIFO First In – First Out

ABBILDUNGSVERZEICHNIS

2.1 Bitpositionen mit MSB(8) und LSB(1) in einem 8-Bit Datenwort 5

2.2 Abarbeitung von Befehlen ohne Pipelining 6

2.3 Abarbeitung von Befehlen mit Pipelining 6

2.4 Beispielrelation zur Einführung der Datenbankterminologie 8

2.5 BBC - Typ 1 Run mit Erläuterung der Bitpositionen 11

2.6 BBC - Typ 2 Run mit Erläuterung der Bitpositionen 12

2.7 BBC - Typ 3 Run mit Erläuterung der Bitpositionen 12

2.8 BBC - Multibyte Counter Beispiel mit Abgrenzung der relevanten Bitpositionen . . 12

2.9 BBC - Typ 4 Run mit Erläuterung der Bitpositionen 13

2.10 WAH - Fill mit Erläuterung der Bitpositionen 14

2.11 WAH - Fill mit Erläuterung der Bitpositionen 14

2.12 EWAH - Signalwort mit Erläuterung der Bitpositionen 15

2.13 PLWAH - Fill mit Sparse und Erläuterung der Bitpositionen 16

2.14 COMPAX - Literal mit Erläuterung der Bitpositionen 16

2.15 COMPAX - 0-Fill mit Erläuterung der Bitpositionen 17

2.16 COMPAX - Literal-Fill-Literal mit Erläuterung der Bitpositionen 17

2.17 COMPAX - Fill-Literal-Fill mit Erläuterung der Bitpositionen 18

3.1 Arbeitsablauf zur Entwicklung eines angepassten Prozessors 21

3.2 TIE Speicherzugriff - *aligned* 23

3.3 TIE Speicherzugriff - *unaligned* 24

3.4 Beispielinstruktion im FLIX Format 28

3.5 Abarbeitung von Befehlen ohne Pipelining 29

3.6 Xtensa LX4 Prozessor Blockschaltbild 30

4.1 SIMD Beispielinstruktion von Intel 32

5.1 Pipelineausschnitt während der Abarbeitung von Instruktionen der WAH Operation 48

5.2 Architekturdiagramm der WAH Operation in TIE - Teil 1 48

5.3 Architekturdiagramm der WAH Operation in TIE - Teil 2 49

5.4 Pipelineausschnitt bei der Verarbeitung von Instruktionen der WAH-Kompression . 58

5.5 Pipelineausschnitt bei der Verarbeitung von Instruktionen der WAH-Dekompression 61

6.1 Architekturdiagramm des oli_imdata_6 Prozessors 64

6.2 Architekturdiagramm des Diamond Standard 108mini 65

6.3 Vergleich der benötigten Takte für die WAH Verarbeitung - AND Verknüpfung . . . 67

6.4 Vergleich der Entwicklungsstufen der WAH Verarbeitung - OR-Verknüpfung 68

6.5 Speedup bei der WAH Operation - Intersektion 69

6.6 Vergleich der benötigten Takte für die PLWAH Verarbeitung - OR-Verknüpfung . . . 70

6.7 Speedup bei der PLWAH Operation - Vereinigung 71

6.8 Vergleich der benötigten Takte für die COMPAX Verarbeitung - OR-Verknüpfung . . 71

6.9 Speedup bei der COMPAX Operation - symmetrische Differenz 72

6.10 Vergleich der benötigten Takte für die Kompression mit WAH - OR-Verknüpfung . 73

6.11 Speedup bei der WAH Kompression - AND-Verknüpfung 73

6.12 Vergleich der benötigten Takte für die Dekompression mit WAH - OR-Verknüpfung 74

6.13 Speedup bei der WAH Dekompression - AND-Verknüpfung 75

A.1 Speedup bei der WAH Operation - Vereinigung 79

A.2 Speedup bei der WAH Operation - symmetrische Differenz 79

A.3 Speedup bei der PLWAH Operation - Intersektion 80

A.4 Speedup bei der PLWAH Operation - symmetrische Differenz 80

A.5 Speedup bei der COMPAX Operation - Intersektion 81

A.6 Speedup bei der COMPAX Operation - Vereinigung 81

TABELLENVERZEICHNIS

2.1 Klassifikation Prozessorarchitekturen nach Flynn 6

2.2 Beispiel einer beliebigen Relation aus einer Datenbank 9

2.3 Bitmap Indizes der Beispielrelation 9

2.4 Vergleich der Kompressionsalgorithmen 19

3.1 Vordefinierte Load/Store Interfacesignale 23

3.2 Vordefinierte Load/Store Interfacesignale (Auszug) 24

5.1 Allgemeiner Aufbau der C-Funktionen mit TIE Intrinsics 40

5.2 Beispielablauf des WAH-Algorithmus mit AND Verknüpfung (HEX) 41

5.3 Instruktionsbeschreibung für die WAH Operation - Version 1 43

5.4 Instruktionsbeschreibung für die WAH Operation - Version 2 44

5.5 Instruktionsbeschreibung für die WAH Operation - Version 3 46

5.6 Instruktionsbeschreibung für die WAH Operation - finale Version 47

5.7 Beschreibung der WAH Pipelinestufen 47

5.8 Beispielablauf des PLWAH-Algorithmus mit OR Verknüpfung (HEX) 50

5.9 Instruktionsbeschreibung für die PLWAH Operation 52

5.10 Beispielablauf der COMPAX Verarbeitung mit XOR Verknüpfung (HEX) 53

5.11 Instruktionsbeschreibung für die COMPAX Operation 55

5.12 Konzeptioneller Ablauf der Word-Aligned Hybrid Code (WAH)-Komprimierung . . . 56

5.13 Instruktionsbeschreibung für die WAH-Kompression 57

5.14 Beschreibung der Pipelinestufen der WAH-Kompression 58

5.15 Konzeptioneller Ablauf der WAH-Dekomprimierung 59

5.16 Instruktionsbeschreibung für die WAH-Kompression 60

5.17 Beschreibung der Pipelinestufen der WAH-Dekompression 61

6.1 Syntheseergebnisse 66

6.2 Abschätzung der Größenanteile 66

1 EINLEITUNG

Dieser Abschnitt erläutert die Motivation zur Bearbeitung des Themas der Befehlssatzerweiterung für die Beschleunigung von Bitmap Indizes großer Datenbanksysteme. Die sich anschließenden Kapitel werden dann die Aufgabenstellung im Detail betrachten und schließlich den Aufbau der vorliegenden Arbeit verdeutlichen.

1.1 MOTIVATION ZUR BEARBEITUNG DER THEMENSTELLUNG

Die zunehmende Technologisierung sowohl im kommerziellen, als auch im privaten Umfeld, führt zu einem ständigen Anwachsen der Menge digitaler Daten. Eine 2003 durchgeführte Studie der Universität Berkley kam zu dem Schluss, dass sich die weltweite Datenmenge zwischen 1999 und 2002 - also in drei Jahren - etwa verdoppelt hat [LV04]. Die International Data Corporation (IDC) prognostiziert bereits heute eine Verdoppelung der weltweit verfügbaren Daten alle zwei Jahre [TGRM14]. Diese steigende Tendenz ist unter anderem auf die zunehmende Vernetzung von Technologiekomponenten zurückzuführen. Weiterhin vergrößert sich die Bandbreite der Datennetze stetig, so dass immer mehr Daten weltweit zur Verfügung stehen.
Um diese Daten in einer strukturierten Weise verfügbar zu machen, wird ein großer Teil mit Datenbanken indiziert. Besonders populäre Vertreter hierfür sind beispielsweise Internet-Suchmaschinen und soziale Netzwerke. Für den Betrieb und die Verwendung dieser Datenbanken müssen spezifische Anforderungen erfüllt werden. Eine möglichst geringe Latenz für die Ausführung der Operationen gehört genauso dazu wie eine möglichst hohe Energieeffizienz dazu. Diese Anforderungen gelten jedoch heute genauso wie in zwei Jahren, in denen sich die Datenmenge allerdings, wie oben beschrieben, wohlmöglich verdoppelt hat.
Die Anforderungen werden weiter steigen, da die verfügbaren Daten im zunehmenden Maße ausgewertet werden, um beispielsweise neue Korrelationen und Verbindungen in den Daten zu finden. Um die Anforderungen auch in Zukunft erfüllen zu können, muss sich die Performance des gesamten Systems weiter verbessern und die Speicherbandbreite weiter steigen. Das wurde in Vergangenheit vor allem durch die Erhöhung des Systemtaktes erreicht. Dieses Vorgehen stößt jedoch zunehmend an physikalische Grenzen. So steigen bei weiterer Erhöhung der Taktfrequenz Energieverbrauch und Wärmeverlust exponentiell, was nicht nur die Leistungsbilanz verschlechtert, sondern an die Grenzen einer ökonomisch sinnvollen Kühlung führt [OH05]. Diesem Fakt wird heute dadurch begegnet, dass statt der Takterhöhung einer Recheneinheit die Parallelisierung in mehreren dieser Rechenkerne vorgenommen wird. Eine weitere Möglichkeit ist die Veränderung der Architektur dieser Rechenkerne. So verspricht die Verwendung von Prozessoren, die speziell für bestimmte Anwendungsfälle konstruiert werden, mehr Verarbeitungsgeschwindigkeit bei

weniger Energieverbrauch. Das ist darauf zurückzuführen, dass durch die größtmögliche Anpassung der Architektur an die Aufgabe zum einen die Abarbeitung beschleunigt werden kann und sich zum anderen keine überflüssigen Komponenten mit zusätzlichem Energieverbrauch in den Kernen befinden.

Die vorliegende Diplomarbeit setzt an genau dieser Stelle an. So werden in dieser Arbeit Bitmap Indizes als datenintensiver Anwendungsfall des Datenbankbereiches genutzt, um Potentiale von speziell angepassten Prozessoren zu analysieren.

1.2 ERLÄUTERUNG DER AUFGABENSTELLUNG

Eine beispielhafte Abfrage in Datenbanksystemen (*Query*) könnte lauten:
„Zeige mir alle Fahrzeuge, die seit 2010 verkauft wurden und eine schwarze Lackierung haben".
Diese Abfrage kann durch sequentielles Scannen der Datenbank realisiert werden, bei dem jeder einzelne Eintrag vollständig analysiert und auf seine Eignung für die Abfrage überprüft wird. Eine andere Möglichkeit ist, separate Listen für „alle verkauften Fahrzeuge seit 2010" und „alle schwarzen Fahrzeuge" zu haben und auszuwerten. Die Schnittmenge der beiden Listen entspricht dann der o.g. Abfrage. Hierfür muss nur eine wesentlich geringere Datenmenge ausgewertet werden, da beispielsweise keine weiteren Details zu den Verkäufen oder Fahrzeugen ausgewertet werden müssen. Auf diese Weise kann die Abfrage deutlich effizienter implementiert werden. Dieses Verfahren ist die Idee der Indizierung in Datenbanken und wird aufgrund der Effizienzvorteile vor allem bei großen Datenmengen angewandt.
Ein denkbarer Anwender für das beschriebene Szenario könnte ein regionales Autohaus sein. In diesem Fall wäre der Aufwand für diese Abfrage wohl überschaubar. Genauso könnte es sich hierbei aber auch um einen weltweit agierenden Autohersteller handeln, der diese Information aus den global verteilten Datenbanken möglichst ohne Latenz generieren möchte.
Das Beispiel macht deutlich, dass es sich hierbei um eine sehr datenintensive Anwendung handeln und daher ein interessantes Themenfeld für die Beschleunigung durch angepasste Prozessoren bieten kann.
Diese Idee wird im Rahmen der vorliegenden Diplomarbeit aufgegriffen. Bitmap Indizes, die eine mögliche Form der oben beschriebenen Indizes sind, werden hierbei zunächst auf die Komplexität der unterschiedlichen Verarbeitungsschritte überprüft. Dazu werden ausgesuchte Algorithmen in C Implementiert und im Rahmen des sogenannten *Profilings* auf besonders rechen- oder datenintensive Abschnitte (*Hot-Spots*) untersucht. Anschließend wird mit TIE (*Tensilica Instruction Extension*) eine Erweiterung des Prozessors *Xtensa LX4* mit speziellen Instruktionen vorgenommen. Optimierungsmöglichkeiten ergeben sich für die Hot-Spots zum Beispiel durch die Kombination mehrerer Operation in einer Instruktion (*Fusion*), der Nutzung einer Instruktion für viele Daten (*Single Instruction, Multiple Data (SIMD)*) oder durch das Verknüpfen mehrerer Instruktionen zu einem großen Instruktionswort (*Very Long Instruction Word (VLIW)*). Außerdem kann der Prozessor im Hinblick auf Architektur sowie interne Register und Speicher konfiguriert und verändert werden. Nach Anpassung der implementierten Algorithmen mit den Optimierungen, werden diese auf ihre Leistungscharakteristiken untersucht, um Verbesserungen und Limitierungen aufzuzeigen.

1.3 AUFBAU DIESER ARBEIT

Nach den einleitenden Ausführungen werden in Kapitel 2 relevante Themen der Rechnerarchitektur und der Datenbanken beschrieben und die betrachteten Algorithmen erläutert. Da die Ergebnisse der vorliegenden Arbeit auf Basis von TIE angefertigt wurden, werden die damit zusammenhängenden Grundlagen und Charakteristiken in Kapitel 3 gesondert dargestellt. Dabei wird nicht nur auf den gewählten Arbeitsablauf eingegangen, sondern es werden auch die Basisbestandteile der Programmiersprache sowie den Möglichkeiten der Optimierung für die spezielle Prozessorarchitektur aufgezeigt. Neben den Ergebnissen, die im Rahmen der vorliegenden Diplomarbeit generiert wurden, stellt die Einordnung der Arbeit in das Fachgebiet einen zentralen Baustein dar. Um sowohl die aktuellen Forschungshintergründe als auch die historischen Entwicklungen aufzuzeigen, wird in Kapitel 4 auf die unterschiedlichen Ansätze anderer Publikationen eingegangen. Dabei wird insbesondere unterschieden, ob die Arbeiten sich auf einen Anpassung der Algorithmen bzw. ihrer Implementierung beschränken, oder ob auch die zugrundeliegende Hardware eine Anpassung erfahren hat.
Im Anschluss wird in Kapitel 5 auf die Implementierung der einzelnen Algorithmen eingegangen, wobei die Verarbeitung durch einführende Beispiele erläutert wird, die in Ausführungen zur Realisierung in C und den zugehörigen Optimierungen münden. Nachdem die Implementierungen beschrieben wurden, werden im anschließenden Kapitel 6 die relevanten Leistungsparameter für jede Implementierung aufgezeigt und ausgewertet. Abschließend wird in Kapitel 7 eine Zusammenfassung der Arbeit gegeben, die sowohl die Analyse der Ergebnisse enthält, als auch einen Ausblick für die Forschungspotentiale über diese Arbeit hinaus.

2 THEORETISCHER HINTERGRUND

Die Basis der vorliegenden Arbeit stellen Computersysteme dar, die nach spezieller Modifikation eine höhere Verarbeitungsgeschwindigkeit aufweisen sollen. Als Teilgebiet der Datenbanktechnologie werden Kompressionsalgorithmen für Bitmap Indizes analysiert und beschleunigt. Die fachlichen Hintergründe in Bezug auf Rechnersysteme und dem Anwendungsfall der Bitmap Indizes werden in den folgenden Abschnitten erläutert.

2.1 RECHNERARCHITEKTUR

Mit der am stärksten abstrahierten Sicht auf die einzelnen Komponenten einen Computer Systems beschäftigt sich das Teilgebiet der Rechnerarchitektur. Aus dieser Perspektive besteht ein Rechner typischerweise aus dem Prozessor als zentraler Recheneinheit (Central Processing Unit (CPU)), Speichereinheiten zum Lesen und Schreiben von Daten sowie Schnittstellen als Module für die Ein- und Ausgabe (Input/Output (I/O)) [Häb11]. Alle Komponenten sind durch Bus-Systeme verbunden, die den Datentransport zwischen den unterschiedlichen Stationen organisieren und realisieren. Wie einführend bereits erwähnt, soll im Rahmen dieser Diplomarbeit eine Steigerung der Rechenleistung durch Veränderung der CPU erreicht werden. Hierbei werden Aspekte der Rechnerarchitektur relevant, die bereits in der historischen Entwicklung der Prozessoren für Möglichkeiten der Beschleunigung gesorgt haben:

Taktfrequenz

Der offensichtlichste Weg zur Erhöhung der Abarbeitungsgeschwindigkeit ist die Steigerung des Systemtaktes. Hierdurch verringert sich die Zeit, die der Prozessor für jeden einzelnen Schritt einer Verarbeitung benötigt, was in einer geringeren Ausführungszeit für das gesamte Programm (Laufzeit) mündet. Die Steigerung des Prozessortaktes geht jedoch mit Nachteilen wie einem steigenden Energiebedarf, höherer Wärmeverlustleistung und neuen Anforderungen an das Gesamtsystem einher. Der Weg einer weiteren Leistungsverbesserung durch erhöhte Taktfrequenz stößt daher schnell an wirtschaftliche und technische Grenzen. Im Rahmen dieser Diplomarbeit wird ein teilweise vorgegebener Prozessor verwendet, dessen Maximalfrequenz festgelegt ist.

Wort- und Verarbeitungsbreite

Um die Effizienz des Systems zu erhöhen, kann auch die Menge der gleichzeitig verarbeiteten Daten erhöht werden. Eine wesentliche Eigenschaft der Systeme ist ihre Verarbeitungsbreite. Diese bezeichnet die maximale Anzahl an Bits, die gleichzeitig über den Systembus

übertragen und vom Prozessor verarbeitet werden können. Die Breite eines Datenwortes entspricht genau dieser Menge an Bits. Historisch haben sich die Wortbreiten von anfangs 8 Bit zu heute üblichen 32 oder 64 Bit gesteigert. Bei der im Rahmen dieser Diplomarbeit verwendeten Big-Endian Bitdarstellung befindet sich das höchstwertigste Bit (MSB bzw. Most Significant Bit) am weitesten links im Bitvektor und das niederwertigste Bit (LSB bzw. Least Significant Bit) befindet sich entsprechend am weitesten rechts. Diese Bezeichnungsweise ist in Abbildung 2.1 noch einmal schematisch dargestellt:

8	7	6	5	4	3	2	**1**

Abbildung 2.1: Bitpositionen mit MSB(8) und LSB(1) in einem 8-Bit Datenwort

Befehlssatz

Die Architektur des Befehlssatzes (Instruction Set Architecture (ISA)) bezeichnet die Gestaltung der Schnittstellen zwischen Prozessor und der Software eines Rechners. Diese vom Prozessor angebotenen Schnittstellen lassen sich in den zwei Gruppen der Reduced Instruction Set Computer (RISC) und Complex Instruction Set Computer (CISC) zuordnen. Die RISC Architektur verfolgt hierbei das Ziel, mit möglichst wenig Befehlen alle notwendigen Funktionen abzubilden. CISC hingegen bieten mehr Instruktionen mit einem größeren Funktionsumfang, die jedoch weniger universell verwendbar sind und somit nur in speziellen Fällen angewendet werden können. Programme können grundsätzlich mit einer RISC- oder einer CISC-Architektur abgearbeitet werden, allerdings werden bei der RISC Architektur üblicherweise mehr Takte benötigt, um die allgemeinen Instruktionen zu einem komplexen Programmablauf zu verbinden. Bei CISC hingegen sind im idealen Fall die benötigten Operationen bereits in wenigen komplexen und speziell angepassten Instruktionen implementiert, sodass weniger Takte für die vollständige Abarbeitung des gleichen Programms benötigt werden. Entsprechend lässt sich bei diesen Architekturen mit einem angepassten Befehlssatz eine Beschleunigung erreichen.

Prefetching und Pipelining

Wenn es wahrscheinlich ist, dass Daten zu einem späteren Abarbeitungszeitpunkt verwendet werden, besteht die Möglichkeit, diese Daten zu einem früheren Zeitpunkt vorzuladen. Das führt dazu, dass die Daten bei der späteren Verarbeitung direkt zur Verfügung stehen und es zu keinen Verzögerungen durch Ladeoperationen kommt. Diese Technik des Vorladens wird als *Prefetching* bezeichnet. Dieses Prefetching lässt sich jedoch nicht nur für Daten realisieren, sondern auch für Instruktionen. Wenn dem Prozessor die nächste Instruktion bereits vorab zur Verfügung steht, kann er sie frühestmöglich abarbeiten.
Üblicherweise werden diese Abarbeitungsschritte bei einem Prozessor in fünf Phasen unterteilt:

1. IF: Holen der Instruktion (*Instruction Fetch*)
2. ID: Lesen der Register und Dekodierung der Instruktion (*Instruction Decode*)
3. EX: Ausführen der Instruktion (*Execute*)
4. MA: Speicherzugriff (*Memory Access*)
5. WB: Schreiben der Ergebnisse (*Write Back*)

Die einfachste Form der Abarbeitung dieser Schritte ist in Abbildung 2.2 dargestellt. Hier werden die Befehle in sequentieller Reihenfolge abgearbeitet. Diese Form der Verarbeitung startet immer erst, wenn der jeweils vorherige Befehl vollständig abgearbeitet ist.

Befehl 1	IF	ID	EX	MA	WB					
Befehl 2						IF	ID	EX	MA	WB
Takt	1	2	3	4	5	6	7	8	9	10

Abbildung 2.2: Abarbeitung von Befehlen ohne Pipelining

Um die Auslastung des Prozessors zu verbessern und somit seine Abarbeitungsgeschwindigkeit zu steigern, muss er möglichst vollständig ausgelastet sein, und in jeder Verarbeitungsphase alle seine Ressourcen nutzen. Diese Form der Ressourcenauslastung wird als *Pipelining* bezeichnet. In Abbildung 2.3 ist die gleiche Verarbeitung noch mal mit Verwendung einer 5-stufigen Pipeline dargestellt.

Befehl 1	IF	ID	EX	MA	WB					
Befehl 2		IF	ID	EX	MA	WB				
Takt	1	2	3	4	5	6	7	8	9	10

Abbildung 2.3: Abarbeitung von Befehlen mit Pipelining

Obwohl sich die in Abbildung 2.3 noch nicht im eingeschwungenen Zustand befindet, wird durch die parallele Abarbeitung der Stufen bereits eine Reduzierung der benötigten Takte erreicht. Als eingeschwungen wird der Zustand bezeichnet, bei dem die Pipeline ihr Verhalten in Bezug auf die durchgeführten Operationen nicht mehr verändert und sich demzufolge regelmäßige Wiederholungen der gleichen Befehle aufweist. Damit die Pipeline wie im dargestellten Beispiel optimal arbeitet, müssen jedoch einige Ausgangsbedingungen an die Struktur, Datenorganisation und Steuerung erfüllt sein. Unter anderem müssen die Eingangsdaten genauso unabhängig voneinander sein wie die Ergebnisse, damit es zu keinen Konflikten oder Wartezeiten kommt.

Unabhängig von spezifischen Merkmalen von Rechner und Prozessor hat Michael Flynn 1966 eine allgemeinere Unterscheidung geprägt [Fly66]. Er hat die Architektur anhand der vorhanden Daten- und Instruktionsströme klassifiziert. Die folgende Tabelle 2.1 verdeutlicht die Architekturbezeichnungen auf Basis der englischen Anfangsbuchstaben:

	Single Instruction	Multiple Instruction
Single Data	**SISD**	**MISD**
Multiple Data	**SIMD**	**MIMD**

Tabelle 2.1: Klassifikation Prozessorarchitekturen nach Flynn

Ein Datenstrom und ein Instruktionsstrom (SISD)

Traditionelle Rechner mit einem Prozessor werden dieser Gruppe zugeordnet. Ohne Parallelität wird bei dieser Architekturform ein Datenstrom mit einem Instruktionsstrom verarbeitet. Zu diesen Rechnern gehören die klassischen Theoriemodelle der Von-Neumann- oder Harvard-Architektur.

Ein Datenstrom und mehrere Instruktionsströme (MISD)

Diese Gruppe ist in der Literatur umstritten, da in der Regel nicht mehrere Instruktionen parallel auf einen Datenstrom bzw. ein Datum angewendet werden. Flynn selbst ordnet dieser Gruppe beispielsweise Systeme zu, die Daten verarbeiten und weiterleiten. Redundante Systeme zur Fehlererkennung und - vermeidung lassen sich aber auch hier einordnen.

Mehrere Datenströme und ein Instruktionsstrom (SIMD)
Zu dieser Gruppe zählen Computer mit Array- oder Vektorprozessor, bei denen gleiche Instruktionen auf mehrere Eingangsdatenströme parallel angewendet werden.

Mehrere Datenströme und mehrere Instruktionsströme (MIMD)
Rechner mit dieser Architektur führen parallel unterschiedliche Operationen auf unterschiedlichen Daten durch. Zu dieser Gruppe zählen also beispielsweise verteilte Systeme, Großrechner oder auch der Verbund von mehreren SISD Prozessoren zu einem größeren System.

Zusammenfassend gibt es sowohl aus der abstrakten Architektursicht, als auch beim konkreten Design von Prozessoren eine Vielzahl von Anpassungs- und Optimierungsmöglichkeiten. Generell lässt sich sagen, dass die stärkere Anpassung des Systems an ein spezielles Problem zwar zeit- und kostenintensiv ist, am Ende aber eine verbesserte Performance des Gesamtsystems verspricht. Diese Ausrichtung der Hardware auf spezielle Teile der Software wird unter dem Begriff des *Hardware-/Software Codesigns* zusammengefasst [DMG97]. Hierbei werden durch gleichzeitiges Design von Hard- und Software die Synergien zwischen den beiden optimal genutzt, sodass am Ende der Entwicklung das bestmögliche Zusammenwirken möglich wird. Dieser Ansatz wird auch in der vorliegenden Diplomarbeit verfolgt. Einer der ersten Untersuchungsschritte befasst sich mit der Identifikation der kritischen Verarbeitungsabschnitte in einem Programm. Diese Abschnitte haben den größten Anteil an der gesamten Programmlaufzeit und werden daher als *Bottleneck* (engl. Flaschenhals) bzw. *Hot-Spot* bezeichnet. Diese Performanceengpässe lassen sich in abstrakter Form auch wieder aus der Architektursicht betrachten. In der Literatur wird hierbei zwischen zwei Schwerpunkten unterschieden, die für die Begrenzung der Leistung verantwortlich sind:

I/O-Bound
In diese Gruppe fallen alle Programme oder Prozesse die vorrangig auf das Ein- oder Ausgeben von Daten warten. Die Laufzeit ist hier also primär von der Fertigstellung der Ein- und Ausgabe abhängig.

CPU-Bound
Programme oder Prozesse dieser Art sind vorrangig durch Rechenprozesse innerhalb der CPU in der Performance begrenzt. Anders als im I/O-Bound Fall werden hier die Daten zwar schnell genug empfangen oder gesendet, aber die Verarbeitung der Daten nimmt den Großteil der Laufzeit in Anspruch.

Im Fokus dieser Diplomarbeit steht Beschleunigung von Bitmap Indizes als Teilgebiet der Datenbanktechnologien. Für viele Datenbankoperationen ist die Laufzeit der eigentlichen Berechnung gegenüber den Lade- und Speicheroperationen vernachlässigbar (I/O-Bound). Jedoch stellte sich bei Untersuchungen von komprimierten Bitmap Indizes heraus, dass diese Annahme hier nicht gilt, sondern abhängig von der Komplexität der Kompressionsalgorithmen ist [SW07]. Bevor jedoch auf die im Rahmen dieser Arbeit gemachten Untersuchungen eingegangen wird, werden in nächsten Kapiteln zunächst die theoretischen Grundlagen zu Datenbanken und den zugehörigen Bitmap Indizes erläutert.

2.2 DATENBANKEN

Die Idee der Relationalen Datenbanken wurde 1970 von Edgar Codd etabliert [Cod70]. Die zu diesem Zeitpunkt entwickelte Terminologie wird in den folgenden Kapiteln verwendet, da sie in der an anderen Stellen zitierten Referenzliteratur als Standard gilt. Der folgende Abschnitt soll eine kurze Einführung in die Begrifflichkeiten geben:

Relation R	Attribut A	Attribut B
Tupel 1	Ausprägung α	Ausprägung β
Tupel 2	Ausprägung γ	Ausprägung δ

Abbildung 2.4: Beispielrelation zur Einführung der Datenbankterminologie

Um den Zusammenhang zwischen den unterschiedlichen Komponenten einer Datenbankrelation zu verdeutlichen, ist eine Beispielrelation schematisch in Abbildung 2.4 dargestellt. Innerhalb einer Relation sind jedem Tupel Attribute zugeordnet, sodass die Tupel unterschiedliche Ausprägungen besitzen können. Wenn die Relation also beispielsweise Personeninformationen enthalten würde und ein Attribut das Geschlecht wäre, würden die zugehörigen Ausprägungen „männlich" und „weiblich" lauten.
Der Inhalt der Relationen wird mit *Queries* (engl. Abfragen) an die Datenbank angefragt und nach der Auswertung entsprechend beantwortet. Ein wesentliches Maß für die Performance von Datenbanken ist die Zeit, die vom Stellen der Anfrage bis zum Erhalt des Ergebnisses vergeht – die sog. *Query Response Time*. Der übliche Fall ist hierbei, dass unterschiedliche Relationen mit bestimmten Operatoren zusammengeführt werden und das Ergebnis entsprechend die Beantwortung einer komplexen Anfrage enthält. Neben der Aggregation und Verknüpfung von Daten über den Gruppierungs- bzw. Verbundoperator gehört der Scan zu den zentralen Operatoren in Datenbanksystemen. Die Scan-Operation realisiert den Zugriff auf alle oder einige ausgewählte Blöcke einer Relation. Insbesondere vor dem Hintergrund großer Datenbanksysteme hat das Einlesen einzelner, restriktiv ausgewählter Datenblöcke (*Index Scan*) gegenüber dem Lesen aller verfügbaren Blöcke einer Relation (*Table Scan*) offensichtliche Geschwindigkeitsvorteile. Nachdem die Daten von einem beliebigen Speichermedium digitaler Daten eingelesen wurden, werden sie an den Folgeoperator zur Datenverarbeitung weitergeleitet. Da dieser Speicher prinzipiell beliebige Größen zwischen einigen Megabyte auf einem einzelnen Speicher bis zu einigen Petabyte auf einem verteilten Speicher annehmen kann, hat die Effizienz dieser Systeme gerade für schnelle Auswertungen eine große Bedeutung.
Das Auffinden relevanter Daten im Rahmen der Scan-Operation kann im einfachsten Fall durch sequentielles Durchlaufen der Datenbasis erfolgen. Um dieses Vorgehen zu beschleunigen, können Indexstrukturen für die Datenbasis angelegt werden, die mit Hilfe von klassischen Entscheidungsbäumen wie dem *B-Baum* und ähnlichen Variationen einen effizienteren Zugriff ermöglichen. Hierbei werden die Tupel nach einer bestimmten Funktion auf Verzweigungskriterien in einem Baum abgebildet, sodass die Suche nach einem bestimmten Tupel immer nur in einem Teil des Baumes und nicht auf den gesamten Daten erfolgen muss. [Leh03]
Die meisten Datenbankprodukte haben eine Version des *B-Baums* implementiert, da es sehr effizient für On-Line Transaction Processing (OLTP) Anwendungen ist. Das ist vor allem darauf zurückzuführen, dass in OLTP Anwendungen die Operationen der Suche und der Aktualisierung etwa mit gleicher Frequenz genutzt werden und diese Operationen bei der Baum-basierten Indizierung auch etwa die gleiche Verarbeitungskomplexität aufweisen.
Bei On-Line Analytical Processing (OLAP) Anwendungen steht hingegen vor allem die Analyse bestehender Datensätze im Vordergrund, weshalb hier die Suche von Daten deutlich häufiger benötigt wird, als das Aktualisieren. Durch das allgemein steigende Datenaufkommen werden diese Datenbankanwendungen zudem mit einer steigenden Menge von Daten konfrontiert. Hierbei hat sich der Bitmap Index als das geeignetste Verfahren herausgestellt. [SW07]

2.3 BITMAP INDIZES

Die Terminologie der Bitmap Indizes wurde von O'Neil [O'N89] im Rahmen eines kommerziellen Datenbankproduktes für IBM Computer eingeführt. Allerdings war die Idee dieser Technik grundsätzlich nicht neu und wird als eine Modifikation von früheren Techniken wie dem *Bit-Transposed File* oder *Inverted Index* gesehen [SW07].

Die Grundidee des Bitmap Index ist die Indizierung aller auftretenden Ausprägungen eines Attributes in einer Relation. Hierbei wird typischerweise für jede mögliche Ausprägung eine separate Bitmap erstellt. Diese Bitmap ist eine geordnete Sequenz von Bits, bei der jedes Bit eine Aussage über den Attributwert in der Ursprungsrelation macht. Aus diesem Grund hat eine Bitmap genauso viele Einträge wie Tupel in der Ursprungsrelation vorhanden sind. Wenn nun das Bit an der Stelle *k* in der Bitmap gesetzt ist (Bitwert ist also „1"), hat das *k*. Tupel der ursprünglichen Relation den Attributwert, der mit der Bitmap assoziiert ist. Ein nicht gesetztes Bit (Bitwert „0") steht entsprechend für einen beliebigen anderen Attributwert [OOW07].

Der Aufbau von Bitmap Indizes wird im folgenden Beispiel noch einmal verdeutlicht. Tabelle 2.2 zeigt beispielhaft eine Relation in einer Datenbank. Hierzu zeigt Tabelle 2.3 die Bitmap Indizes von den zwei Attributen „Beruf" und „Geschlecht" der Ursprungsrelation. Hierbei wird deutlich, dass ein Bit in den Bitmaps jeweils genau ein Äquivalent in der Beispielrelation hat und somit auch die gleiche Anzahl an Einträgen. Die Anzahl der Bitmaps ergibt sich somit aus den Kardinalitäten der Attribute. So hat beispielsweise das Attribut „Beruf"in diesem Fall drei mögliche Ausprägungen, weshalb drei Bitmaps zu Indizierung benötigt werden.

Zeile	Person	Geschlecht	Beruf
0	Arthur	männlich	Student
1	Elizabeth	weiblich	Professor
2	Julia	weiblich	Student
3	Robert	männlich	Gärtner

Tabelle 2.2: Beispiel einer beliebigen Relation aus einer Datenbank

Zeile	BI_1: männlich	BI_2: weiblich	BI_3: Student	BI_4: Professor	BI_5: Gärtner
0	1	0	1	0	0
1	0	1	0	1	0
2	0	1	1	0	0
3	1	0	0	0	1

Tabelle 2.3: Bitmap Indizes der Beispielrelation

Da Bitoperationen in der Computerhardware sehr effizient implementiert werden können, führt diese Art der Indizierung zu einer Steigerung der Systemperformance, insbesondere bei großen Datenbanken mit aufwändigen Abfragen. Beispielsweise kann in diesem Fall die Abfrage „Zeige alle männlichen Studenten" durch eine einfache, logische UND-Verknüpfung der Indizes BI_1 und BI_3 erreicht werden.
Dieser Geschwindigkeitsvorteil geht jedoch mit dem Nachteil des zusätzlichen Speicheraufwands einher. Der Sachverhalt, dass die generierte Datenmenge proportional zur Kardinalität des Attributes ist, wird in der Literatur als *Curse of Cardinality* (engl. Fluch der Kardinalität) bezeichnet. Damit wird das Problem umschrieben, dass sich bei vielen Ausprägungen eines Attributs ergibt. Bitmap Indizes arbeiten sehr effizient, wenn die Kardinalität eines Attributes relativ klein ist – hier beispielweise das Attribut „Geschlecht". Allerdings wären auch „Wohnort" oder „Postleitzahl" Beispiele für andere Attribute, die indiziert werden können – jedoch mit wesentlich höherer Kardinalität. Ein vollständiger Bitmap Index für das Attribut der Postleitzahl wäre durch die hohe Zahl der unterschiedlichen Ausprägungen und der daraus folgenden Menge an benötigten Bitmap Indizes entsprechend groß, was die Effizienz des Index beeinträchtigt.

Angenommen, N beschreibt die Anzahl der Tupel in der Relation und $K(X)$ die Kardinalität eines Attributes X in der Ursprungsrelation, dann beträgt der zusätzlich benötigte Speicherbedarf für die Indizierung: $N \cdot K(X)$
Im schlechtesten Fall hat jedes Tupel eine andere Attributsausprägung, womit sich $K(X) = N$ ergibt und der zusätzliche Speicheraufwand mit N^2 Bits quadratisch mit der Anzahl der Tupel in der Ursprungsrelation wächst.
In diesem Extremfall bestehen die N Bitmaps jedoch aus lediglich einem gesetzten Bit und $N-1$ 0-Bits. Ein offensichtlicher Ansatz, um diese Bitfolge Speichereffizienter zu beschreiben, ist eine Aussage über die Länge zu treffen, statt die Folge selbst zu speichern. Zum Beispiel:

$$0000\ 0000\ 0000\ 0000\ 1\ 0000\ 0000 \Rightarrow 16x0; 1x1; 8x0 \tag{2.1}$$

Dieses Prinzip wird als Lauflängencodierung bzw. Run-Length Encoding (RLE) bezeichnet. Wenn das Element A in einem Datenstrom M-mal hintereinander auftaucht, dann kann mit Hilfe von RLE die M Vorkommen von A durch ein einziges AM ersetzt werden. Diese Form der Kompression ist verlustfrei und sie eignet sich insbesondere für Datenfolgen bei denen eine hohe Wahrscheinlichkeit von gleichen, aufeinander folgenden Elementen besteht. Zudem besteht die Möglichkeit, direkt aus dem analysierten Element eine Aussage über die unkomprimierten Daten zu erhalten [Sal07]. Das führt dazu, dass für eine Datenverarbeitung nicht immer komplett komprimiert und dekomprimiert werden muss. Obwohl in der Literatur bereits speichereffizientere Kompressionsalgorithmen vorhanden sind, hat RLE durch die Möglichkeit der direkten Verarbeitung einen weitgehend einzigartigen Vorteil, sodass diese Technik als primäre Kompressionsstrategie für Bitmap Indizes verwendet wird.
Die Erfinder der Algorithmen verwenden zum Teil eine unterschiedliche Termininologie zur Beschreibung ihrer Abarbeitungsvorschriften. Aus Gründen der Übersichtlichkeit und in Übereinstimmung mit einem Großteil der Referenzliteratur wird in dieser Arbeit folgende Termininologie zur Beschreibung der Algorithmen verwendet:

Codewort
Der aus der Informationstheorie stammende Begriff des Codeworts wird in diesem Zusammenhang verwendet, um ein komprimiertes Wort zu kennzeichnen.

Literal
Ein Literal ist eine Sequenz von Bits, die nicht komprimiert werden kann und daher unkomprimiert in die komprimierte Datenmenge übernommen wird.

Fill (engl. Füllung)
Fill bezeichnet ein lauflängenkodiertes Element. Dieses Codewort enthält Informationen über den Typ des komprimierten Elements und einem Zähler, der die Länge der Sequenz beinhaltet.

Sparse (engl. spärlich)
Ein Sparse eine Folge von Bits, in denen nur sehr wenige 1-Bits existieren. Von diesem Muster wird in der Literatur vor allem dann gesprochen, wenn es sich um einen Bitvektor mit einem 1-Bit umgeben von 0-Bits handelt.

Run (engl. Durchlauf)
Run umschreibt die, bei einer Iteration gleichzeitig betrachtete Sequenz von Bits. Diese kann prinzipiell zwischen beliebigen Größen variieren – die betrachten Algorithmen nehmen übliche Größen der Rechnerverarbeitungsbreite (den Wörtern) an.

Chunk (engl. Fragment)
Chunk bezeichnet ein Fragment von Ursprungsdaten, das nach einer definierten Weise aus den Originaldaten extrahiert wird.

In den nächsten Unterkapiteln werden unterschiedliche Algorithmen vorgestellt, die alle auf dem Prinzip von RLE basieren. Sie sind daher vor allem effizient, wenn sie bei langen, gleichartigen Bitfolgen eingesetzt werden. Um hierbei die Erfindungen im historischen Kontext sehen zu können, erfolgt die Vorstellung der einzelnen Algorithmen in chronologischer Reihenfolge. Die unterschiedlichen Systeme stellen eine Auswahl der bekanntesten Algorithmen dar, welche sowohl in der jeweiligen Veröffentlichung als auch in unabhängigen Studien miteinander verglichen wurden [Joh99, FVDD13a].

2.3.1 BBC Komprimierung

Die ursprüngliche Version des Byte-Aligned Bitmap Code (BBC) wurde 1994 von Gennady Antoshenkov patentiert [Ant94]. Wie bereits aus der Bezeichnung ersichtlich ist, arbeitet der Algorithmus in jedem Run auf Bytelänge. Entsprechend genau können gleichförmige Bitsequenzen von alternierenden abgegrenzt werden, was zu einer sehr guten Komprimierungsleistung führt. BBC arbeitet hierbei mit Signalwörtern innerhalb des komprimierten Datenstroms (sog. *Control Bytes*).
Diese Signalwörter dienen, ähnlich den in der Nachrichtentechnik verwendeten Pilotsequenzen, zur Einordnung des darauf folgenden Datenstromes. Jedes Signalwort gibt hierbei Aufschluss über eine bestimmte Länge von darauf folgenden Bits. Sobald der zu einem Signalwort gehörende Datenstrom beendet ist, schließt sich entweder das nächste Signalwort mit entsprechendem Bitstrom an, oder die komprimierte Bitsequenz ist zu Ende. BBC verwendet in der patentierten Version insgesamt acht verschiedene Typen dieser Signalwörter. Durch die genaue Anpassung des Signalwortes an die jeweilige Sequenz der zu komprimierenden Bits kann, zusätzlich zu der bereits erwähnten genauen Trennung der Wörter, eine gute Komprimierungsleistung erreicht werden. In einer unabhängigen Studie wurde bei Zufallsdaten eine Komprimierung auf dem Niveau von gzip nachgewiesen [Joh99]. Als eines der Standard-Kompressionverfahren hat gzip vor allem das Ziel einer besonders starken Komprimierung der Daten. Da BBC neben der Komprimierungsleistung auch eine möglichst kurze Laufzeit als Ziel hat, ist diese Kompressionsleistung – verglichen mit anderen RLE Verfahren für Bitmap Indizes – außergewöhnlich gut.
Die Aufteilung der komprimierten Bits in acht verschiedene Gruppen ist jedoch vor allem bei der Verknüpfung mehrerer komprimierter Bitmaps relativ aufwändig. Das liegt daran, dass die Signalwörter in der Ergebnisbitmap auf Basis des Verknüpfungsergebnisses immer wieder verändert werden müssen. In der Literatur haben sich daher unterschiedliche Subformen des ursprünglichen Algorithmus gebildet. Beispielsweise wird in der o.g. Studie bereits unterschieden, ob der Algorithmus lediglich Folgen von 0-Bits als Fill beschreiben (*one-sided BBC*) oder zusätzlich auch 1-Bitfolgen abbilden kann (*two-sided BBC*). Um die Komplexität von BBC zu reduzieren und trotzdem die grundlegenden Kompressionseigenschaften beizubehalten, wurde eine weitere BBC Abwandlung mit einer reduzierten Anzahl an Codewörtern etabliert [WOSN01]. Da nur für diesen Algorithmus Laufzeituntersuchungen dokumentiert sind, die als Vergleich zu dieser Arbeit dienen können, wird im Folgenden auch nur dieser Algorithmus weiter betrachtet.

Grundsätzlich werden in diesem Algorithmus komprimierte Bitgruppen anhand von vier unterschiedlichen Signalwörtern unterschieden. Diese werden, der Literatur folgend, in vier verschiedene Run Typen eingeteilt und im Folgenden näher erläutert.

Typ 1 Run:

Codewort Bsp.:	**1**	0	01	1001
Bitpositionen:	8	7	6-5	4-1

Bitposition	Erläuterung
8	Identifiziert das Codewort (fester Wert)
7	Typ des Fills
6-5	Anzahl der im Fill repräsentierten Bytes
4-1	Anzahl der auf das Signalwort folgenden Literale

Abbildung 2.5: BBC - Typ 1 Run mit Erläuterung der Bitpositionen

Ein Signalwort des ersten Typs beschreibt eine Sequenz von bis zu drei aufeinander folgenden Fill-Bytes, gefolgt von bis zu 15 Literalen. Die Struktur des Signalworts ist in Abbildung 2.5 dargestellt. Neben dem ersten Bit als Identifikation des Typ 1 Signalwortes, beschreibt der Beispielvektor ein 0-Fill über die Länge von einem Byte und neun darauf folgende Literale, die im komprimierten Bitstrom auf dieses Signalwort folgen würden.

Typ 2 Run:

Codewort Bsp.:	**01**	1	10	010
Bitpositionen:	8-7	6	5-4	3-1

Bitposition	Erläuterung
8-7	Identifiziert das Codewort (fester Wert)
6	Typ des Fills
5-4	Anzahl der im Fill repräsentierten Bytes
3-1	Position des gesetzten Bits in einem Sparse Bitvektor

Abbildung 2.6: BBC - Typ 2 Run mit Erläuterung der Bitpositionen

Ein Typ 2 Run beschreibt bis zu 3 Fill Bytes gefolgt von einem Sparse, welches über die Position des gesetzten Bits abgebildet wird. Im Abbildung 2.6 dargestellten Beispiel handelt es sich also um ein 1-Fill über 2 Byte und einem Sparse Byte, das – bis auf das gesetzte Bit an der zweiten Bitposition – nur Null-Bits enthält.

Typ 3 Run:

Codewort Bsp.:	**001**	1	1000
Bitpositionen:	8-6	5	4-1

Bitposition	Erläuterung
8-6	Identifiziert das Codewort (fester Wert)
5	Typ des Fills
4-1	Anzahl der auf das Signalwort folgenden Literale

Abbildung 2.7: BBC - Typ 3 Run mit Erläuterung der Bitpositionen

Das dritte mögliche Signalwort bildet eine besonders lange Folge von Fills ab. Hierfür wird in dem in Abbildung 2.7 dargestellten Signalwort lediglich der Run Typ, gefolgt vom Fill Typ und die sich anschließenden Literale abgebildet. Der Lauflängenzähler wird aufgrund der Größe in den, auf das Signalwort folgenden, *Multibyte Counter* integriert. Ein Beispiel für diesen Lauflängenzähler ist in Abbildung 2.8 dargestellt:

1	111 1111		**0**	000 0011

Abbildung 2.8: BBC - Multibyte Counter Beispiel mit Abgrenzung der relevanten Bitpositionen

Neben dem eigentlichen Zähler für die Lauflängenkodierten Fills in den sieben niederwertigsten Bits dient das höchstwertigste dazu, eventuell folgende Multibyte Counter über das Setzen des MSB anzukündigen. Das passiert immer dann, wenn ein Byte nicht ausreicht, um den Fill-Zähler abzubilden, weshalb dann entsprechend die Summe aus mehreren Multibyte Zählern gebildet wird. Das in Abbildung 2.8 aufgeführte Beispiel entspricht also insgesamt 130 Fill Bytes ($111\ 1111_{bin} = 127_{dec}$ aus dem ersten und $000\ 0011_{bin} = 3_{dec}$ aus dem zweiten Multibyte Counter). Die zusammen in einem komprimierten Bitstrom auftreten Beispielvektoren aus Abbildung 2.7 und Abbildung 2.8 würden in unkomprimierter Form also einem Bitstrom von 130x8 1-Bits, gefolgt von 8 Literalen entsprechen.

Typ 4 Run:

Codewort Bsp.:	**0001**	0	101
Bitpositionen:	8-5	4	3-1

Bitposition	Erläuterung
8-5	Identifiziert das Codewort (fester Wert)
4	Typ des Fills
3-1	Position des gesetzten Bits in einem Sparse Bitvektor

Abbildung 2.9: BBC - Typ 4 Run mit Erläuterung der Bitpositionen

Das vierte und letzte Signalwort wird verwendet, wenn im unkomprimierten Bitstrom eine lange Folge von Fills auftritt und im Anschluss ein Sparse folgt. Entsprechend folgen dem in Abbildung 2.9 dargestellten Codewort wie im vorherigen Typ 3 Run ein oder mehr Multibyte Counter im komprimierten Bitstrom.

2.3.2 WAH Komprimierung

Kensheng Wu etablierte die, im vorherigen Kapitel bereits erläuterte Abwandlung von BBC. Während seiner Forschungsarbeiten untersuchte er unterschiedliche Algorithmen auf ihre Vor- und Nachteile. Bei der Laufzeitanalyse von BBC stellte sich heraus, dass der Algorithmus einen hohen Verarbeitungsaufwand verursachte. Der Großteil der Laufzeit wurde während der Verarbeitung in der CPU benötigt und nur ein geringerer Teil durch das Lesen der Bitmaps von der Festplatte [WOS04]. Mit einem verringerten Rechenaufwand in der CPU und einem erhöhten Datendurchsatz ließe sich also eine Laufzeitverbesserung erzielen.
Erforderlich war daher ein verhältnismäßig leicht zu realisierender Kompressionsalgorithmus mit einer schnelleren Abarbeitung der Daten. Aus dieser Idee resultierte der 2004 von Kensheng Wu patentierte Word-Aligned Hybrid Code (WAH) [WSO04]. Die Grundidee von RLE bzw. BBC aufgreifend, wird auch bei WAH zwischen Fills und Literalen unterschieden - jedoch ohne die Notwendigkeit von zusätzlichen Signalwörtern. Da moderne Computerhardware für 32 oder 64 Bit Wortgrößen ausgelegt ist, wird bei WAH auf Basis der namensgebenden Wörtern gearbeitet. Das Prinzip unterscheidet sich für die beiden Wortlängen nicht, weshalb der Algorithmus in den nachfolgenden Ausführungen aus Gründen der Lesbarkeit lediglich in der 32 Bit Variante betrachtet wird. Grundsätzlich sieht der Algorithmus zudem vor, dass im letzten Wort die Anzahl der verwendeten Bits aus dem Wort davor gespeichert werden. Da im Rahmen dieser Arbeit davon ausgegangen wird, dass die Menge der Eingangsbits bekannt ist, wird in den Ausführungen nicht näher auf das letzte Wort eingegangen.
Dem Algorithmus folgend, wird der Eingangsdatenstrom zunächst in 31 Bit Gruppen - die sog. *Chunks* – aufgeteilt. Diese Gruppen können dann den im Folgenden erläuterten Codewörtern zugeordnet werden:

Fill:

Codewort Bsp.:	**1**	1	00 0000 0000 0000 0000 0000 0000 0100
Bitpositionen:	32	31	30-1

Bitposition	Erläuterung
32	Identifiziert das Codewort (fester Wert)
31	Typ des Fills
30-1	Anzahl der im Fill repräsentierten Wörter

Abbildung 2.10: WAH - Fill mit Erläuterung der Bitpositionen

Sofern es im unkomprimierten Eingangsdatenstrom mindestens eine 31 Bit Gruppe gibt, die ausschließlich 0- oder 1-Bits enthält, wird diese durch ein Fill repräsentiert. Wie in Abbildung 2.10 ersichtlich, wird in diesem Codewort zur Identifizierung das MSB auf 1 gesetzt und durch die Bitposition 31 gekennzeichnet, ob es sich um ein 0- oder 1-Fill handelt. Die restlichen 30 Bit des Vektors dienen als Zähler für die aufeinander folgenden Chunks mit der Größe von 31 Bit. Im Beispiel handelt es sich entsprechend um $100_{bin} = 4_{dec}$x31 Bits, die alle auf 1 gesetzt sind.

Literal:

Codewort Bsp.:	**0**	000 0010 0000 0010 0000 1100 0000 0100
Bitpositionen:	32	31-1

Bitposition	Erläuterung
32	Identifiziert das Codewort (fester Wert)
31-1	Unkomprimierter 31 Bit Chunk

Abbildung 2.11: WAH - Fill mit Erläuterung der Bitpositionen

Wenn sich in einem Chunk mindestens ein Bit befindet, dass sich vom Rest in der gleichen 31 Bit Gruppe unterscheidet, wird dieses Wort in WAH nicht komprimiert, sondern in unkomprimierter Form in den komprimierten Datenstrom übernommen. Hierbei wird, wie aus dem Beispiel in Abbildung 2.11 ersichtlich, lediglich das MSB des komprimierten Wortes auf 0 gesetzt und der 31 Bit Chunk unkomprimiert durch die restlichen Bits dieses Codeworts dargestellt.

Bei WAH gibt es keine weiteren Codewörter Typen, weshalb sich die Verarbeitung gegenüber BBC vereinfacht. Jedes der zwei möglichen Codewörter kann ohne die Notwendigkeit der Interpretation von Signalwörtern verwendet werden. Deshalb kann die Verarbeitung von komprimierten Bit Vektoren ohne das partielle Dekomprimieren des Datenstromes durchgeführt werden. Aufgrund der Verarbeitung von Wörtern statt Bytes hatten die Erfinder des WAH Algorithmus mindestens mit einer Vervierfachung der Performance gegenüber BBC gerechnet [WOS01]. Tatsächlich wurden logische Operationen auf den Bitmaps bei WAH 12x schneller durchgeführt als auf BBC Bitmaps. Das ist vor allem auf die Abhängigkeiten der einzelnen BBC Wörter untereinander zurückzuführen. Da die Signalwörter zunächst ausgewertet und nach jeder Operation angepasst werden müssen, entstehen während der Abarbeitung Wartezeiten beim Pipelining, sodass sich die Laufzeit des Algorithmus schnell vervielfacht. Neben der verbesserten Laufzeit wurde jedoch auch eine weniger starke Komprimierungsleistung gegenüber BBC beobachtet. Den Untersuchungen von Wu folgend, verbraucht WAH aufgrund der Optimierung auf Datendurchsatz und Laufzeiteffizenz etwa 50% mehr Speicherplatz als BBC. Alle Messungen sowohl mit synthetischen als auch mit realen Daten durchgeführt.

2.3.3 EWAH Komprimierung

Das größte Potential zur Verbesserung von WAH findet sich also offensichtlich in der Größe der komprimierten Daten. Um 31 Bit Literale zu speichern benötigt WAH 32 Bit in dem komprimierten Datenstrom. Entsprechend wird bei einem Ausgangsdatenstrom, der nur Literale enthält, ein zusätzliches Bit pro 31 Bit Chunk benötigt. Im schlechtesten Fall vergrößert sich die komprimierte Datenmenge gegenüber der unkomprimierten also um etwa 3%. Um dieses Problem zu lösen, hat das Team von Owen Kaser 2008 einen leicht angepassten WAH Algorithmus unter dem Namen Extended Word-Aligned Hybrid Code (EWAH) veröffentlicht [KLA08]. Um die Vergrößerung der Bitmap bei vielen Literalen zu verhindern, darf also offensichtlich kein weiteres Bit in die Literale eingefügt werden. Um trotzdem die benötigte Unterscheidung zwischen den Codewörtern vornehmen zu können, werden bei EWAH wieder Signalwörter eingeführt. Um das System so einfach wie möglich zu halten, gibt es jedoch nur ein mögliches Signalwort:

Codewort Bsp.:	1	0000 0000 0000 0100	000 0000 0000 1000
Bitpositionen:	32	31-16	15-1

Bitposition	Erläuterung
32	Typ des Fills
31-16	Anzahl der im Fill repräsentierten Wörter
15-1	Anzahl der auf das Signalwort folgenden Literale

Abbildung 2.12: EWAH - Signalwort mit Erläuterung der Bitpositionen

Entsprechend des in Abbildung 2.12 dargestellten Schemas verwendet EWAH immer ein 32 Bit Wort zur Codierung, um welche Art von Fill Bits es sich handelt, wie lang das Fill ist und wie viele Literale sich anschließen. Im Beispiel handelt es sich um eine Folge von 4x32 Bit Wörtern, die nur aus 1-Bits bestehen. Im Anschluss an dieses Signalwort würde im komprimierten Datenstrom eine Folge von acht unkomprimierten Literalen folgen. Danach kann der Datenstrom entweder enden, oder es schließt sich das nächste Signalwort an. Aus diesem Grund startet ein EWAH Datenstrom auch immer mit einem Signalwort, aus dessen Inhalt sich die Position des nächsten Signalwortes im Datenstrom ablesen lässt. Im Ergebnis verwendet EWAH für einen Datenstrom, der nur aus Literalen besteht, ein zusätzliches 32 Bit Wort pro 32767 Literalen, die in den 15 Bit des Signalwortes codierbar sind. Da die Kompression von Bitmap Indizes grundsätzlich nur für sehr große Datenbanken attraktiv ist, wird bei EWAH – verglichen mit dem unkomprimierten Datenstrom – im schlechtesten Fall nur sehr wenig zusätzlicher Speicherplatz benötigt. Dieser Vorteil der Kompressionsrate geht jedoch mit dem Nachteil einher, dass wie bei BBC die partielle Interpretation des Datenstromes zur Verarbeitung notwendig wird.

2.3.4 PLWAH Komprimierung

Die Autoren von Position List Word-Aligned Hybrid Code (PLWAH) beobachteten, dass es bei der ursprünglichen WAH Implementierung relativ häufig vorkam, dass die höchstwertigsten Bits in den Fill Zählern nicht verwendet werden [DP10]. Aus diesem Grund schlugen die Autoren vor, diese Bits anders zu verwenden, um die Codewörter effizienter zu nutzen. Neben dem Literal, das genauso wie in WAH implementiert wird, wird also das Codewort zur Abbildung des Fills verändert. Abhängig von der Bitverteilung der Eingangsdaten ist es nach einer langen Gruppe von gleichen Bits relativ wahrscheinlich, dass sich das erste daran anschließende Wort nur relativ gering von der Gruppe unterscheidet. Beispielsweise also ein 0-Fill, auf das ein Wort mit einem gesetzten 1-Bit folgt und sich dann weitere 0-Fills anschließen. Aufgrund dieser Beobachtung lag

die Idee nahe, die nur geringfügig veränderten Wörter, im Anschluss an ein Fill, mit in diesem zu codieren. Abbildung 2.13 erläutert den Aufbau:

Codewort Bsp.:	**1**	0	0 0011	0 0000 0000 0000 0000 0000 0001
Bitpositionen:	32	31	30-26	25-1

Bitposition	Erläuterung
32	Identifiziert das Codewort (fester Wert)
31	Typ des Fills
30-26	Bitposition des abweichenden Bits
25-1	Anzahl der im Fill repräsentierten Wörter

Abbildung 2.13: PLWAH - Fill mit Sparse und Erläuterung der Bitpositionen

Verglichen mit WAH wird also der Bereich für den Lauflängenzähler um fünf Bits verkleinert und diese Bits stattdessen zur Codierung des von der Gruppe abweichenden Bits verwendet. Das Beispiel aus Abbildung 2.13 codiert entsprechend ein 0-Fill und einen Sparse Bitvektor bei dem nur das dritte Bit gesetzt ist, während der Rest des Chunks aus 0-Bits besteht. Die Idee nur genau ein abweichendes Bit zu codieren wurde gleichzeitig unter dem Namen Concise (Compressed 'n' Composable Integer Set) veröffentlicht [CP10]. Tatsächlich schlagen die Autoren von PLWAH vor, abhängig von den Eingangsdaten mehrere 5-Bit Gruppen in den Fills zu verwenden, um dann auch einen Vektor mit mehreren abweichenden Bits codieren zu können. Da Laufzeituntersuchungen nur für die grundlegende PLWAH Variante bzw. Concise vorliegen, wird in dieser Diplomarbeit ausschließlich die Codierung eines abweichenden Bits untersucht. Mit dieser Modifikation erreichen die Erfinder von PLWAH nach eigenen Messungen häufig eine komprimierte Bitmap, die nur die Hälfte des Speicherplatzes einer WAH Bitmap benötigt.

2.3.5 COMPAX Komprimierung

Die Erfinder des Compressed Adaptive Index Format (COMPAX) waren auf der Suche nach einem Kompressionsalgorithmus für Bitmap Indizes, die über Daten von Netzwerktraffic erzeugt wurden [FSV10]. In ihrem Fall wurden die Ausgangsdaten intensiv auf wiederkehrende Muster untersucht und sie fanden heraus, dass 1-Fills praktisch nie auftraten – jedoch Kombinationen von Fills und Literalen im Wechsel relativ häufig auftraten. Aus diesem Grund entwickelten sie COMPAX als einen Kompressionsalgorithmus, der ihren Typ von Datenmustern besonders gut komprimieren konnte. Im Folgenden werden die Codewörter von COMPAX einzeln erklärt:

Literal:

Codewort Bsp.:	**1**	000 0100 0000 0100 0000 1111 0000 0110
Bitpositionen:	32	31-1

Bitposition	Erläuterung
32	Identifiziert das Codewort (fester Wert)
31-1	Unkomprimierter 31 Bit Chunk

Abbildung 2.14: COMPAX - Literal mit Erläuterung der Bitpositionen

Wie in Abbildung 2.14 dargestellt, wird auch bei COMPAX das MSB verwendet, um anzukündigen, dass die restlichen 31 Bit des Vektors ein unkomprimiertes Literal sind.

0-Fill:

Codewort Bsp.:	**000**	0 0000 0000 0000 0000 0000 0000 0100
Bitpositionen:	32-30	29-1

Bitposition	Erläuterung
32-30	Identifiziert das Codewort (fester Wert)
29-1	Anzahl der im Fill repräsentierten Wörter

Abbildung 2.15: COMPAX - 0-Fill mit Erläuterung der Bitpositionen

Im Gegensatz zu den anderen Kompressionsverfahren, gibt es bei COMPAX keine Möglichkeit, ein 1-Fill abzubilden. Die Möglichkeit der Kompression von 0-Fills bleibt jedoch erhalten. Um ein 0-Fill Codewort zu kennzeichnen, werden – wie aus Abbildung 2.15 ersichtlich – die ersten drei MSB benötigt. Die restlichen 29 Bit dienen als Zähler für die Lauflänge des Fills.

Fill-Literal-Fill:

Codewort Bsp.:	**010**	01	00	0	00001111	00001111	00001111
Bitpositionen:	32-30	29-28	27-26	25	24-17	16-9	8-1

Bitposition	Erläuterung
32-30	Identifiziert das Codewort (fester Wert)
29-28	Byteposition des Literals
27-26	Nicht genutzt
25	Nicht genutzt
24-17	Letztes Byte des ersten Fill Wortes
16-9	Nicht-Null Byte des Literals
8-1	Letztes Byte des zweiten Fill Wortes

Abbildung 2.16: COMPAX - Literal-Fill-Literal mit Erläuterung der Bitpositionen

Das in Abbildung 2.16 beschriebene Spezialwort von COMPAX dient als Zusammenfassung von zwei Literalen und einem 0-Fill. Diese Zusammenfassung kann allerdings nur vorgenommen werden, wenn es in den beiden Literalen nicht mehr als eine Bytegruppe mit 1-Bits gibt. Diese Bytegruppe wird dann, zusammen mit ihrer Position, in dem Spezialwort gespeichert. Zudem darf hierbei das 0-Fill lediglich das niederwertigste Byte zum Zählen der Fills verwenden.

Literal-Fill-Literal:

Codewort Bsp.:	**001**	01	00	0	00001111	00001111	00001111
Bitpositionen:	32-30	29-28	27-26	25	24-17	16-9	8-1

Bitposition	Erläuterung
32-30	Identifiziert das Codewort (fester Wert)
29-28	Byteposition des ersten Literals
27-26	Byteposition des zweiten Literals
25	Nicht genutzt
24-17	Nicht-Null Byte des ersten Literals
16-9	Letztes Byte des Fill Wortes
8-1	Nicht-Null Byte des zweiten Literals

Abbildung 2.17: COMPAX - Fill-Literal-Fill mit Erläuterung der Bitpositionen

Analog zum vorherigen Spezialwort werden in dem Spezialwort aus Abbildung 2.17 zwei 0-Fills und ein Literal gespeichert. Allerdings weiterhin mit den Anforderungen, dass die beiden Zähler der Fills nicht über das niederwertigste Byte hinaus gehen und das Literal nur ein Byte besitzen darf, in dem sich 1-Bits befinden. Sofern das der Fall ist, werden die drei Codewörter in einem Spezialwort zusammengefasst.

COMPAX ist also als einziges der genannten Kompressionsverfahren auf ein spezielles Datenmuster ausgerichtet. Hierbei erreichten die Erfinder eine bis zu 60 % stärkere Kompressionsrate, als es bei WAH der Fall war. Das liegt vor allem an der Fähigkeit, drei Codewörter in einem zu vereinigen, sofern sie gewisse Anforderungen erfüllen.

2.3.6 Zusammenfassung

Insgesamt zeigen die in den vorherigen Unterkapiteln erläuterten Algorithmen, dass die Zweckmäßigkeit der Verwendung des einen oder des anderen Algorithmus, vor allem von den Randbedingungen abhängt. Die Bedeutung von Kompressionsrate und Häufigkeit bestimmter Datenmuster für die Laufzeitoptimierung wurde dargestellt. Im Rahmen dieser Diplomarbeit sollen die Algorithmen vor allem durch effizientes Vorladen der Daten und Parallelität auf Befehlsebene beschleunigt werden. Um diese Parallelität zu erzeugen, ist das gleichmäßige Abarbeiten von Daten notwendig. Unter diesem Aspekt sind in der folgenden Tabelle 2.4 alle Algorithmen noch einmal zusammengefasst.

Eigenschaft	BBC	WAH	PLWAH	EWAH	COMPAX
Wortbreite [Bit]	8	32/64	32/64	32	32
Signalwörter	☒	☐	☐	☒	☐
Zustandslose Verarbeitung	☐	☒	☒	☐	☒
Sequentieller Datenzugriff	☐	☒	☒	☐	☒

Tabelle 2.4: Vergleich der Kompressionsalgorithmen

BBC und EWAH als die Algorithmen mit Signalwörtern weisen Abhängigkeiten der Daten untereinander auf, da jedes Datum erst mit Kenntnis des zugehörigen Signalwortes interpretiert werden kann. Es handelt sich hierbei also um zustandsbehaftete Algorithmen, welche die Daten nicht ohne Wissen über den aktuellen Zustand verarbeiten können. Wie aus der Tabelle 2.4 hervorgeht, folgt daraus auch die Notwendigkeit von zufälligen Speicherzugriffen, da beispielsweise beim Verknüpfen mehrerer Indizes die Zustände abhängig von den neu generierten Daten verändert werden müssen. Zusammenfassend kostet also die Verwendung von Signalwörtern zusätzliche Verarbeitungszeit für die Interpretation und Generierung der Zustände. Dieser Umstand verlangsamt die Kompression, Dekompression und Verknüpfung der Bitmap Indizes mit diesen Algorithmen zu Gunsten einer verbesserten Kompression. In dieser Arbeit sollen jedoch vor allem Geschwindigkeitspotentiale einer angepassten Prozessorarchitektur aufgezeigt werden. Dadurch liegt der Schwerpunkt dieser Arbeit vor allem auf der Laufzeitoptimierung der Algorithmen. Um die *runtime* so gering wie möglich zu halten, werden im Folgenden nur Algorithmen ohne Signalwörter untersucht.

3 TENSILICA INSTRUCTION EXTENSION

Die technische Grundlage dieser Arbeit stellt die von der Firma *Tensilica, Inc.* entwickelte Programmiersprache TIE (*Tensilica Instruction Extension*) dar. Das mittlerweile von *Cadence Design Systems, Inc.* aufgekaufte Unternehmen stellt damit eine Sprache ähnlich den klassischen Hardwarebeschreibungssprachen (Hardware Describtion Language (HDL)). zur Verfügung [Gon00]. Hiermit ist es möglich, den Instruktionssatz von Prozessoren aus der *Xtensa* Familie zu erweitern. Diese Prozessoren stellen bereits Basisfunktionalitäten zur Verfügung, sodass die Anwender lediglich ihre spezifischen Instruktionen entwerfen und in den Prozessor einbringen müssen. Die generelle Idee der Entwicklung mit TIE startet mit Messungen, welche Abschnitte im Ursprungsquellcode besonders daten- oder rechenintensiv sind (*Hot-Spots*). Diese Abschnitte können dann ganz oder teilweise durch Instruktionen im Prozessor ersetzt werden. Nach Beschreibung der Instruktionen können die Entwickler ihre Instruktion als normalen Funktionsaufruf in Ihrem Quellcode verwenden, sodass der Ansatz des Hardware/ Software Codesigns direkt umgesetzt werden kann. Neben der Möglichkeit des manuellen Entwurfs der Instruktionen, bieten die TIE Softwarepordukte auch die Möglichkeit der automatischen Optimierung von Quellcodes. Diese wurden im Rahmen der vorliegenden Arbeit jedoch nicht verwendet.
Die nachfolgenden Unterkapitel erläutern die Grundlagen des Entwurfs mit TIE. Dazu wird auf spezielle Konstrukte der Sprache genauso eingegangen, wie auf Möglichkeiten der Optimierung und das verwendete Prozessormodell. Die Ausführungen basieren maßgeblich auf der, vom Hersteller zur Verfügung gestellten, Dokumentation zum Produkt [Ten11, Lei06].

3.1 ARBEITSABLAUF

Um die Optimierung eines Programmcodes mit TIE zu erreichen, gibt es eine Entwicklungsstrategie, die sich vorrangig in die Gruppen Simulation und Synthese einordnen lässt [AHF+14]. In der Simulationsphase werden die Analysen der Ursprungsquellcodes durchgeführt, die in der optimierten Version des Quellcodes mit TIE Instruktionen münden. Im Anschluss wird während der Synthesephase der Prozessor zunächst auf Parameter wie Fläche, Frequenz und Energieverbrauch optimiert um am Ende der Phase den Prozessor mit allen gewünschten Charakteristiken des Designs generieren zu können. Die Abbildung 3.1 stellt den Ablauf noch einmal schematisch dar:

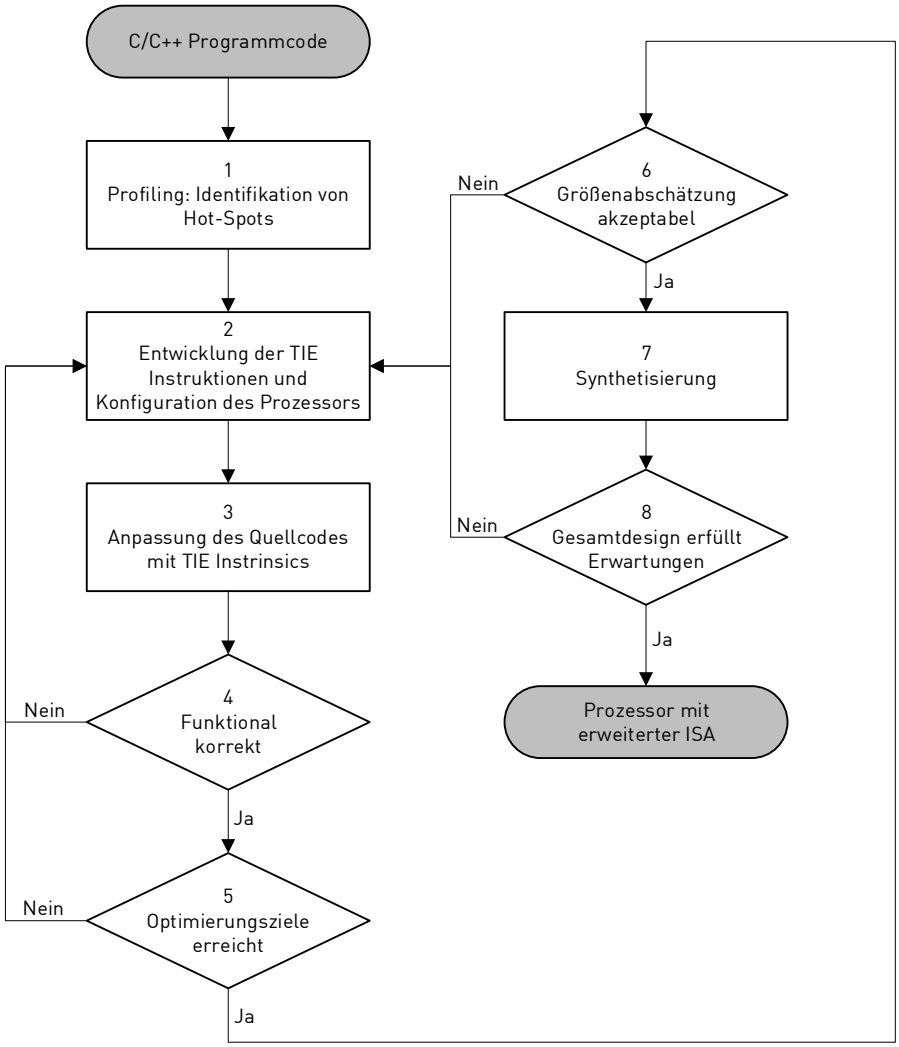

Abbildung 3.1: Arbeitsablauf zur Entwicklung eines angepassten Prozessors

Der Abschnitt der Simulation beginnt mit der zyklenakkuraten Untersuchung (*Profiling*) des zu optimierenden Quellcodes (1). Diese Analyse zeigt das Laufzeitverhalten der unterschiedlichen Abschnitte im Code. Die Teile, die durch häufige Aufrufe oder rechenintensive Operationen einen besonders großen Anteil an der Laufzeit haben, werden hierbei visualisiert und als mögliche Ansatzpunkte für neue Intruktionen identifiziert. Im Anschluss wird die Funktionalität dieser Abschnitte in eigenen Instruktionen im Prozessor abgebildet (2). Außerdem kann der Prozessor hierbei für den jeweils benötigten Anwendungsfall, wie beispielsweise einem zusätzlichen Speicherinterface in Form einer *Load/Store-Unit*, konfiguriert werden. Die hinzugefügten Instruktionen können nun in Form von speziellen Funktionsaufrufen (*Intrinsics*) in den Quellcode integriert werden (3). Im Anschluss muss zunächst die funktionale Korrektheit des Codes überprüft werden (4). Sofern der überarbeitete Quellcode die richtigen Ergebnisse liefert, folgt der nächste Schritt (5). Durch die eingefügten Intrinsics verändert sich das Laufzeitverhalten der optimierten Abschnitte. Die Laufzeit der Abschnitte sollte sich reduziert haben und eventuell findet eine Verschiebung der Bottlenecks statt. Entsprechend wird bei einem erneuten Profiling festgestellt, ob die Optimierungsergebnisse im Rahmen der Erwartungen liegen (5). Sofern es weitere Möglichkeiten der Optimierung gibt,

können weitere Instruktionen eingebracht werden, die erneut die genannten Schritte durchlaufen (3). Diese Iteration wird durchgeführt, bis der Code das gewünschte Laufzeitverhalten aufweist und in die nächste Phase übergehen kann.

In der Synthesephase geht es vor allem um die Erfüllung der Designvorgaben. Dazu zählen die Größe des Cores, sowie Leistungsparameter wie Taktfrequenz und Energieverbrauch. Aus diesem Grund wird zunächst überprüft, ob die Größenabschätzung des Cores die Erwartungen erfüllt (7). Größenoptimierungen können beispielsweise durch Wiederverwendung von Hardwarebestandteilen oder die exakte Bemessung der verwendeten Einheiten erreicht werden. Sollten die angestrebten Ziele erfüllt sein, wird im nächsten Schritt der Prozessor synthetisiert (8). Zum Schluss wird überprüft, ob das erstellte Design die gewünschten Anforderungen erfüllt. Sofern es weiteren Optimierungsbedarf gibt, müssen die TIE Instruktionen überarbeitet werden (3). Andernfalls ist das Prozessormodell mit anwendungsspezifischem Registersatz (*Application-Specific ISA*) bereit zur Produktion.

3.2 PROGRAMMIERUNG VON INSTRUKTIONEN MIT TIE

Die von Tensilica entworfene Programmiersprache TIE dient dem Entwurf neuer Prozessorinstruktionen. Es handelt sich hierbei high-level Programmiersprache, mit der sich die Instruktionserweiterungen der Xtensa Prozessorfamilie realisieren lassen. Im Folgenden werden einige grundlegende Bestandteile der Sprache erläutert.
Zunächst werden alle entworfenen Instruktionen über eine TIE spezifische Datei geschrieben und als header Datei zum C/C++ Quelltext hinzugefügt. Der folgende Befehl ermöglicht den Aufruf der entworfenen Instruktionen aus der Datei *extended-isa.tie* über Intrinsics:

```
#include <xtensa/tie/extended-isa.h>
```

Innerhalb dieser eingebundenen Datei, können alle Sprachkonstrukte von TIE verwendet werden. Einige dieser Konstrukte werden nachfolgend genauer erläutert, da sie die Grundlage der weitergehenden Konzepte darstellen.

Registerbänke

Als Gruppe von Registern, können Registerbänke (*register files*) dem Speichern von Instruktionsspezifischen Inhalten dienen. Die Breite der einzelnen Register sowie die Tiefe der Files kann dabei definitiert werden. Die notwendigen Lade- und Speicheroperationen werden automatisch generiert, sofern die Registerbreite kleiner oder gleich der maximalen Verarbeitungsbreite des Speicherinterfaces ist. Hierbei wird innerhalb von C/C++ ein neuer Datentyp generiert, der äquivalent zu den Standardtypen (`short`, `int`, `char`, usw.) verwendet werden kann. Die neu definitierten Datentypen können aus diesem Grund auch beliebig oft verwendet werden, obwohl es nur eine beschränkte Anzahl physischer Register zur Verfügung steht. Für die bereits in C/C++ integrierten Datentypen stellt TIE automatisch *Adress Register* (`AR`) zur Verfügung, sodass diese ohne separate Konfiguration verwendet werden können.

Beispiel:

```
regfile exampleReg 128 8 er
```

Dieser Befehl erstellt ein Register `exampleReg` der Tiefe acht mit jeweils 128 Bit Breite. Neben dem neuen Datentyp `exampleReg` wird bei der Erstellung die Kurzbezeichnung `er` vergeben, welche sich später in den Assembleranweisungen wiederfindet.

Statusvariablen

Neben den bereits erwähnten Registern, die eher einen generellen Charakter haben, lassen sich mit Hilfe von *States* exklusive Speicher für spezielle Verwendungen generieren. Der Programmierer muss hier zu jeder Verwendung manuell das Lesen und Schreiben verwalten, wobei beides innerhalb eines Taktes auf dem gleichen State erfolgen kann.

Beispiel:

```
state exampleState 32 32'd0 add_read_write
```

Dieser Befehl erstellt einen State `exampleState` mit 32 Bit Breite, initialisiert diesen mit Null und generiert direkte Zugriffsmöglichkeiten für den Quellcode. Die Initialisierung erfolgt mit `32'd0`, was einem Vektor mit 32 Bit entspricht der den Dezimalwert Null hat. Äquivalent dazu kann dies auch in binärer- (`32'b0`) oder hexadezimaler (`32'h0`) Form geschehen. Dieser Initialisierungswert ist bei der Definition des States genauso optional wie `add_read_write`, wodurch der direkte Statezugriff aus dem C/C++ Quelltext ermöglicht wird.

Lade- und Speicheroperationen

Um Transfers zwischen Speicher und Prozessor vorzunehmen, existieren von TIE vordefinitierten *Interface* Signale für die Lade- und Speichereinheiten (*Load/Store Units*). Neben der Definition von Zugriffsbreite und Direktionalität besteht hierbei auch die Möglichkeit von bedingten Lade- und Speicheroprationen in Abhängigkeit von Eingangswerten, wodurch sich Verzweigungen im Ursprungscode reduzieren lassen. Exemplarisch werden in der Tabelle 3.1 die für diese Arbeit besonders relevanten Signale zusammengefasst.

Signal	Beschreibung
`VAddr`	32 Bit breite Speicheradresse für die Load/Store (L/S) Signale
`MemDataIn128`	128 Bit der von L/S Einheit aus dem Speicher gelesenen Daten
`MemDataOut128`	128 Bit der durch die L/S Einheit zu speichernden Daten
`LoadByteDisable`	Bitmaske zum Deaktivieren des Ladens einzelner Bytes
`StoreByteDisable`	Bitmaske zum Deaktivieren des Speicherns einzelner Bytes

Tabelle 3.1: Vordefinierte Load/Store Interfacesignale

Zu beachten ist hierbei zum einen, dass TIE aus Effizienzgründen Speicherzugriffe immer ausgerichtet (*aligned*) durchführt und zum anderen die Ordnung der Wörter anpasst. In Abbildung 3.2 ist der Speicherzugriff auf vier 32 Bit Wörter mit 128 Bit Alignment schematisch dargestellt. Die durchnummerierten Wörter (Abbildung 3.2a) werden bei der 128 Bit Ladeoperation in entgegengesetzter Reihenfolge im Prozessor angeordnet (Abbildung 3.2b). Bei Speicheroperationen erfolgt die Neuordnung entgegensetzt.

(a) Speicherabbild (b) Geladener Speicherinhalt

Abbildung 3.2: TIE Speicherzugriff - *aligned*

In Abbildung 3.3 ist der Speicherzugriff ohne Ausrichtung dargestellt. Wenn der Programmierer beispielsweise das zweite Wort (1) als Adresse des Speicherzugriffs festlegt, erfolgt der 128 Bit Zugriff TIE intern weiterhin auf die niedrigere Adresse mit 128 Bit Ausrichtung (Abbildung 3.3a). Das so geholte Datum wird im Anschluss solange nach rechts rotiert, bis das eigentlich adressierte Wort am weitesten rechts steht (Abbildung 3.3b).

(a) Speicherabbild (b) Geladener Speicherinhalt

Abbildung 3.3: TIE Speicherzugriff - *unaligned*

Variablen

Innerhalb von Instruktionen bedarf es aus Effizienz- und Übersichtlichkeitsgründen der Nutzung von Variablen. Ähnlich der Verwendung von temporären Variablen in C/C++, stellen die `wire` Operationen diese Funktionalität innerhalb von TIE bereit. Sofern nicht anders initialisiert, sind sie 1 Bit breit.

Beispiel:

```
wire [31:0] exampleVar = 32'hFFFFFFFF;
```

Dieser Befehl initialisiert ein `wire` der Breite 32 Bit mit der Bezeichnung `exampleVar` und setzt alle Bits auf 1.

Zuweisungen

Neben der erwähnten lokalen Zuweisung, stellt das `assign` Kommando die Möglichkeit der Operations- und Zyklenübergreifenden Definition zur Verfügung. Die Zuweisungen (*Assignments*) können in jedem Takt nur einmal gelesen und geschrieben werden. Darüber hinaus werden die Assignments nicht in der Reihenfolge des Aufrufs ausgeführt, sondern unabhängig von der Reihenfolge erfolgt zunächst das Lesen aller Eingangswerte. Im Anschluss werden die Zuweisungen in Reihenfolge der Abhängigkeiten verarbeitet und erst dann geschrieben.

Beispiel:

```
assign increment = value+1;
```

Dieser Befehl weist `increment` den um eins erhöhten Wert von `value` zu. Hierbei kann `value` entweder eine andere gelesene Zuweisung oder ein `wire` sein.

Operatoren

Die Verarbeitung und Verknüpfung von Werten geschieht durch unterschiedliche Operatoren. Diese sind bereits vordefiniert und umfassen neben allgemeinen arithmetischen und logische Operatoren auch speziellere Operationen für die Werteverarbeitung. Diese werden in der folgenden Tabelle 3.2 exemplarisch vorgestellt, da sie als Grundlage späterer Ausführungen dienen.

Befehl	Beschreibung
`x[m:n]`	Extraktion der Bitgruppe mit den Postionen n bis m
`{a,b}`	Konkatenation der Variablen a und b
`x?y:z`	Wenn-Dann Abfrage: wenn x wahr ist, dann y, sonst z

Tabelle 3.2: Vordefinierte Load/Store Interfacesignale (Auszug)

Operationen

Die zu entwickelnden Instruktionen werden zu diesem Zeitpunkt des Entwurfes im TIE Kontext als Operationen bezeichnet. Nachdem mögliche Bestandteile dieser Operationen erläutert wurden, folgt nun die Beschreibung der `operation` selbst. Diese spezifiziert einen Namen, die notwendigen Werte und das Verhalten selbst.

Beispiel:

```
operation exchangeMSBLSB {out AR result, in AR variable}{}
{
    assign result =  {variable[0],variable[30:1],variable[31]};
}
```

Dieser Quellcode definiert die Instruktion bzw. Operation `exchangeMSBLSB`. Hierbei werden zunächst der Name definiert und im Anschluss jeweils in geschweiften Klammen die Argumente, die States und schließlich die Funktionalität selbst. Als Argumente werden hier die AR Register verwendet, um einen Eingabe- und einen Rückgabewert zu haben. Die Direktionalität der Operanden kann neben `in` und `out` auch `inout` sein, sodass sowohl Lesen als auch Schreiben möglich sind. Da innerhalb dieser Operation keine States verwendet werden, ist der folgende Block geschweifter Klammern leer. Der eingelesene Wert `variable` mit der Breite 32 Bit wird hierbei aufgeteilt und wieder zusammen gesetzt. Der Zugriff auf einzelne Bits oder Bitbereiche wird über die Bereichsdefinition in den eckigen Klammern realisiert während das Zusammensetzen durch Nutzung der geschweiften Klammern ausgeführt wird. Das zurückgegebene Resultat `result` dieser Operation liefert also den Eingangswert mit vertauschtem ersten und letzten Bit zurück. Über die Intrinsics kann diese Operation nun folgendermaßen im C/C++ Quelltext verwendet werden:

```
newVal=exchangeMSBLSB(0x80000000);
```

Dieser Abschnitt des C-Quelltextes setzt den Wert der Variable `newVal` auf 1.

Module

Mit den TIE Operationen können komplexe Abläufe realisiert werden. Neben diese vollständig neu zu entwickelnden Funktionen, bietet TIE bereits vordefinierte *Module* zum Realisieren komplexerer Funktionen, als es allein mit den Operatoren möglich wäre. Beispielhaft zu nennen ist hier die *multiply-accumulate* Funktionalität, die sich im FPGA Kontext etabliert hat. Während die vollständige Liste der Module in [Ten11] eingesehen werden kann, soll an dieser Stelle beispielhaft auf das *Multiplex* (MUX) Modul eingegangen werden, da es zu den relevantesten Modulen innerhalb der vorliegenden Arbeit gehört.
Bei der Operation `TIEmux` handelt es sich um einen n-Wege Multiplexer, der folgendermaßen aufgebaut ist:

```
TIEmux(x, y_0, y_1, ..., y_n-1);
```

Auf Basis der Bitbreite b von x, wird die Anzahl der möglichen Wege n festgelegt, wobei $n = 2^b$ gilt. In Abhängigkeit des Wertes von x, wird dann der entsprechende Weg gewählt.

Beispiel:

```
wire [1:0] shiftSel = 2'b10;
assign x = TIEmux(shiftSel, x<<32, x<<64, x<<128, x<<256);
```

Dieser Quellcode erzeugt eine Variable `shiftSel` mit der Breite 2 Bit und dem Dezimalwert 2. Im Anschluss wird x durch die Nutzung des MUX sein vorheriger Wert zugewiesen, jedoch mit einem Bitshift nach links um 128 Bit.

3.3 OPTIMIERUNGSTECHNIKEN

Nachdem die Grundkonstrukte im vorherigen Abschnitt 3.2 erläutert wurden, folgt nun die nähere Betrachtung unterschiedlicher Strategien zur Optimierung des Ursprungsprogramms. Den Beginn des in Abschnitt 3.1 beschriebenen Ablaufs, stellt das Profiling zum Finden der Hot-Spots im Quellcode dar. Im Anschluss an deren Identifizierung, folgt die Optimierung durch TIE Operationen. In Abhängigkeit davon, ob der Hot-Spot beispielsweise durch rechen- oder datenintensive Abschnitte entstanden ist, gibt es unterschiedliche Formen der Optimierung. Diese bestehen in den meisten Fällen aus Abwägungen. Beispielsweise wirkt sich der Umfang der implementierten Operationen auf die Größe des Chips oder dessen Taktfrequenz aus, sodass für jede Implementierung eine Abwägung zwischen den unterschiedlichen Parametern notwendig ist. Weiterhin können die zu optimierenden Algorithmen normalerweise in unterschiedlicher Form implementiert werden, wodurch sich auch verschiedene Hot-Spots und Optimierungspotentiale ergeben. Bei der Entwicklung stellt sich auch die Frage, wie spezialisiert die Instruktionen für das jeweilige Problem sein sollen. So können Instruktionen von generellerer Natur möglicherweise für unterschiedliche Zwecke wiederverwendet werden, während sehr spezielle praktisch nur eine gleichbleibende Aufgabe erfüllen können. Da höher spezialisierte Instruktionen aber normalerweise für mehr Geschwindigkeitsverbesserung sorgen, als ihre generalisierten Pendants, werden im Rahmen dieser Arbeit Instruktionen speziell für den jeweiligen Problemfall erstellt. Die einzelnen Techniken und Strategien zur Optimierung lassen sich in unterschiedliche Gruppen einordnen, die im Folgenden näher erläutert werden.

3.3.1 Fusion

Das Zusammenführen mehrerer Operationen in einer Instruktion wird als *Fusion* bezeichnet. Ein Optimierungsbeispiel zeigt der folgende Ausschnitt eines Quellcodes in C:

```
int a,b,c_quadrat;
...
c_quadrat=(a*a)+(b*b);
```

Zur Berechnung von `c_quadrat` werden die Variablen `a` und `b` zunächst quadriert und die Zwischenergebnisse dann addiert. Für die Operation werden mit den Standardinstruktionen also mindestens zwei Takte zur Berechnung benötigt. Wenn diese Kalkulation jetzt unter Nutzung von Fusion in TIE implementiert wird, könnte das folgendermaßen geschehen:

```
 operation PYTHAGORAS {out AR ergebnis, in AR a, in AR b} {} {
        wire [31:0] a_quadrat = a * a;
        wire [31:0] b_quadrat = b * b;
        assign ergebnis = a_quadrat+b_quadrat;
}
```

Die Operation `PYTHAGORAS` verwendet hierbei die bereits im Prozessor integrierten Register um die Eingangsvariablen zu empfangen und das Ergebnis zu übertragen. Ohne die Betrachtung von Initialisierung und ähnlichen notwendigen Aspekten, wird dann innerhalb der neuen Instruktion die gleiche Operation wie zuvor in C umgesetzt, aber in einem Takt abgearbeitet. Die Nutzung dieser neuen Instruktion kann nun, wie bereits geschildert, über die Intrinsics in C verwendet werden, nachdem die TIE Datei eingebunden wurde. Der C Quellcode würde nach der Optimierung folgende Form haben:

```
#include <xtensa/tie/pythagoras.h>
int a,b,c_quadrat;
...
c_quadrat=PYTHAGORAS(a,b);
```

3.3.2 SIMD

Eine weitere Möglichkeit der Optimierung stellt die Nutzung von mehreren Daten durch eine Instruktion dar. Die Verwendung dieser Technik bietet sich vor allem bei der wiederholten Verarbeitung von Daten mit der gleichen Instruktion an. Ein Beispiel zeigt der folgenden C-Code:

```
unsigned int i,a[4], b[4], gleichheit;
...
for (i=0; i<4; i++)
        if(a[i]==b[i])
         gleichheit++;
```

Die vier 32 Bit Wörter in den Arrays `a` und `b` werden jeweils zum Vergleich geladen und bei Übereinstimmung wird `gleichheit` inkrementiert. Alternativ könnte dieser auch in der folgenden SIMD Instruktion verarbeitet werden:

```
state vektor1 128
state vektor2 128

operation VERGLEICH {out AR ergebnis} {in vektor1, in vektor2}
{
     wire vgl1 = ( vektor1[31:0]   ==  vektor2[31:0]  );
     wire vgl2 = ( vektor1[63:32]  ==  vektor2[63:32] );
     wire vgl3 = ( vektor1[95:64]  ==  vektor2[95:64] );
     wire vgl4 = ( vektor1[127:96] ==  vektor2[127:96]);

     assign ergebnis = vgl1+vgl2+vgl3+vgl4;
}
```

Abgesehen von der Initialisierung, die für die beiden Vektoren benötigt wird, können mit Hilfe der neuen Instruktion 4 Iterationen der Schleife in einer kombiniert werden. Innerhalb der Operation wird jeder temporären Variable das Ergebnis der Überprüfung auf Gleichheit zugewiesen. Der gesamte Bitbereich der Vektoren zwischen den Bits 0 und 127 wird dabei entsprechend auf die vier Vergleiche aufgeteilt und die Summe der Vergleiche dem `ergebnis` zugewiesen, dass über das AR Register dann wie im vorhergehenden Beispiel direkt im C Code verfügbar ist. Denkbar wäre auch, das vorher geschilderte Beispiel der Fusion Optimierung in einer Schleife zu haben und zusammen mit SIMD zu optimieren. Die vorgestellten Optimierungsansätze stellen immer nur eine Grundlage dar und können unterschiedlich miteinander kombiniert werden, um noch größere Beschleunigungen zu erreichen.

3.3.3 FLIX

Neben der Möglichkeit, die notwendigen Operationen innerhalb einer Instruktion durchzuführen, gibt es auch den Weg der gleichzeitigen Ausführung mehrerer Instruktionen. Es handelt sich hierbei um die Flexible Length Instruction Xtension (FLIX) Funktion, die ähnlich zum Very Long Instruction Word (VLIW) ist. Mit Hilf dieser Funktionalität ist es möglich, ein Instruktionswort größerer Länge zu erzeugen und mehrere Instruktionen in den sog. *Slots* der großen Instruktion zu kombinieren. Die Namen der Instruktionen (sog. *Opcodes*) können hierbei sowohl aus den vordefinierten Instruktionen des Prozessors bestehen, als auch den selbst entworfenen Operationen. Ein Beispiel für ein FLIX-Wort ist in Abbildung 3.4 dargestellt. Das insgesamt 64 Bit Länge (0 bis 63) Wort enthält hierbei zwei Opcode Slots mit 35 und 24 Bit Länge. Dieses, als `xt_format2`

bezeichnete Schema, wird auch im Rahmen der vorliegenden Arbeit zur Beschleunigung verwendet.

63	62 28	27 4	3 0
0	**xt_flix64_slot3**	**xt_flix64_slot0**	**1111**
1	35	24	4

Abbildung 3.4: Beispielinstruktion im FLIX Format

Neben der Möglichkeit der automatischen Suche für Instruktionen, die durch FLIX beschleunigt werden können, kann die Zuweisung mit der Hilfe von TIE auch manuell erfolgen. Der Notwendige Befehl `slot_opcodes` kann hierbei die Slots zu den jeweiligen Opcodes zuweisen. Das folgende Beispiel verdeutlicht die Nutzung von FLIX anhand der Instruktionen aus den vorherigen Unterkapiteln:

```
slot_opcodes xt_flix64_slot3  {VERGLEICH}
slot_opcodes xt_flix64_slot0  {PHYTHAGORAS}
```

Ohne die Notwendigkeit einer Veränderung im C-Code, werden damit die beiden neuen Instruktionen zu einer FLIX Instruktionen nach dem Schema in Abbildung 3.4 zusammengefasst. Dadurch können sie nun parallel ausgeführt werden. Die nebenläufige Ausführung ist jedoch an die Unabhängigkeit der Ausgangsdaten gebunden. Prinzipiell können beliebig viele Instruktionen den einzelnen Slots zugewiesen werden, jedoch teilen sich Opcodes keinerlei Hardware, wenn sie in unterschiedlichen Slots stehen. Je nachdem welches Optimierungsproblem vorliegt, kann also sowohl der Einsatz als auch der Verzicht auf FLIX sinnvoll sein.

Einen wesentlichen Grund des Einsatzes von FLIX kann die Nutzung von zwei Load/Store Einheiten darstellen. Da diese Funktionalität innerhalb der vorliegenden Arbeit verwendet wird, soll genauer auf diesen Anwendungsfall eingegangen werden. Der folgende Ausschnitt aus dem TIE Quellcode dient hierbei der Verdeutlichung:

```
slot_opcodes xt_flix64_slot3  {load128_LS1}
slot_opcodes xt_flix64_slot0  {load128_LS2}
```

In dem zugehörigen TIE Code existieren die Operationen `load128_LS1` und `load128_LS2` als eigenständige Instruktionen, die jeweils einen Speicherzugriff von 128 Bit auslösen. Bei der Definition entsprechend des o.g. Schemas, wird jedem Slot der FLIX-Instruktion eine Load/Store Einheit zugewiesen, sodass die beiden Ladeoperationen parallel ausgeführt werden können. Prinzipiell kann diese Funktionalität auch von Speicheroperationen verwendet werden. Jedoch muss in jedem Fall auf die Anordnung der Daten im Speicher geachtet werden.

3.3.4 Datenorganisation

Die parallele Nutzung von Lade- und Speicherbefehlen ist nur möglich, wenn die zwei Load/Store Einheiten auf unterschiedliche Speicherbereiche zugreifen. Um das zu erreichen, muss sichergestellt werden, dass die Pointer auf die zugehörigen Daten in unterschiedliche Speicherbereiche zeigen. Das kann zum Beispiel dadurch erreicht werden, dass die Variablen bereits während der Definition manuell in unterschiedliche Speicherbereiche gelegt werden. Der folgende Ausschnitt des C-Quellcodes verdeutlicht, wie die Arrays `array1` und `array2` mit Hilfe des `__attribute__` Kommandos in die unterschiedlichen Speicher `dram0` und `dram1` gelegt werden:

```
int array1[100] __attribute__((section(".dram0.data")));
int array2[200] __attribute__((section(".dram1.data")));
```

Wenn nun die Load/Store Einheiten auf die beiden Arrays zugreifen, kann der Zugriff parallel und somit auch innerhalb einer FLIX Instruktion erfolgen.

Neben der Definition der Speicherbereiche für die Instruktionen, ist auch die Abhängigkeit der Daten untereinander ein wesentlicher Aspekt. Der von TIE verwendete Prozessor, arbeitet mit einer 5-stufigen Pipeline, die bereits in Abschnitt 2.1 beschrieben wurde. Das optimale Pipelining in Form der gleichzeitigen Ausführung von fünf Instruktionen kann jedoch nur umgesetzt werden, wenn die Daten untereinander keine Abhängigkeiten besitzen. Neben der Abhängigkeiten die eventuell bereits extern durch die Algorithmen vorgegeben werden, existiert auch innerhalb des Prozessors die Abhängigkeit der Daten untereinander. Das ist zum Beispiel der Fall, wenn eine Ladeoperation ausgeführt wird.A Abbildung 3.5 verdeutlicht die Problematik

load128_LS1	IF	ID	EX	MA	WB			
load128_LS2		IF	ID	EX	MA	WB		
processData			IF	ID		EX	MA	WB
Takt	1	2	3	4	5	6	7	8

Abbildung 3.5: Abarbeitung von Befehlen ohne Pipelining

Wenn beispielsweise die vorher beschriebenen Ladeoperationen ausgeführt werden, stehen die geladenen Daten erst am Ende der Memory Access (MA) Phase zur Verfügung, werden aber von der Beispieloperation `processData` bereits in der Execute (EX) Phase benötigt. In diesem Fall wird die Pipeline um einen Takt verzögert (graues Feld), um die Daten im nächsten Takt zu verarbeiten. Dieser Umstand wird auch als *Pipeline Stall* bezeichnet und gehört in die Gruppe der Pipline Konflikten bzw. *Pipeline Hazards*. Da sich die Laufzeit dadurch insgesamt verlängert, ist Vermeidung dieser Wartezeiten anzustreben. Bei den Ladevorgängen lässt sich dieser Umstand durch den Einsatz von Hilfsregistern bzw. *Shadowregistern* verhindern. Dazu schreiben die Ladeoperationen ihre Daten zunächst in einen Zwischenstate und Kopieren den Inhalt bei ihrem nächsten Aufruf in den eigentlichen State zur Verarbeitung. Da dieses Verschieben innerhalb eines Taktes möglich ist und nicht wie beim Laden zwei Takte in Anspruch nimmt, kann der Pipeline Stall verhindert werden.

3.3.5 Hardwareaspekte

Wie bereits zu Beginn des Kapitels erwähnt, stellt die Optimierung immer eine Abwägung zwischen unterschiedlichen Charakteristiken dar. So stehen sich zum Beispiel die letztendliche Größe des Chips und die Parallelität der Hardware gegenüber. Wenn mehrere Einheiten zur parallelen Verarbeitung in Hardware vorgehalten werden sollen, führt das zu einem größeren Chip als wenn jegliche Hardware geteilt wird. Einen weiteren, fundamentalen Aspekt des letztendlichen Designs stellt die Taktfrequenz dar. Diese wird maßgeblich durch die Länge des *kritischen Pfades* vorgegeben. Sofern der Pfad zu lang wird, kann der Chip nicht mehr mit einer bestimmten Frequenz getaktet werden, da dieser Pfad noch nicht abgearbeitet ist, bevor der nächste Takt kommt. Die Länge wird dabei vor allem durch sequentielle Verarbeitung im Rahmen der Fusion Optimierung verursacht. So stellen parallele Verarbeitungen zwar keine Problem dar, aber wenn eine Verarbeitung vom Ergebnis der vorherigen Verarbeitung abhängt und diese weitere Abhängigkeiten aufweist, verlängert sich der kritische Pfad. Eine Möglichkeit dem entgegen zu wirken ist der Einsatz von Pipelining innerhalb der Instruktionen, was auch im Rahmen der vorliegenden Arbeit angewandt wurde.

3.4 PROZESSORMODELL

Die Basis für die Instruktionsentwicklung stellt, neben der bereits erläuterten Sprache für die Software, die Hardware zur Implementierung dar. Tensilica hat zu diesem Zweck den konfigurier- und erweiterbaren Prozessor der *Xtensa* Familie entwickelt. Diese Prozessoren verfügen lediglich über Basisfunktionalitäten, die jedoch über die TIE Sprache erweitert werden können. Zudem lassen sich die bestehenden Komponenten entsprechend des jeweiligen Anwendungsfalls konfigurieren. Neben den vollständig konfigurierten *Xtensa Diamond Cores*, gibt es den *Xtensa LX4 Mikroprozessor*, der Grundlage dieser Arbeit ist und daher nun näher beschrieben wird. Der Xtensa LX4 ist, in Anlehnung an [Ten12], in Abbildung 3.6 als Blockschaltbild dargestellt, um den Aufbau und einige wesentliche Komponenten näher zu erläutern.

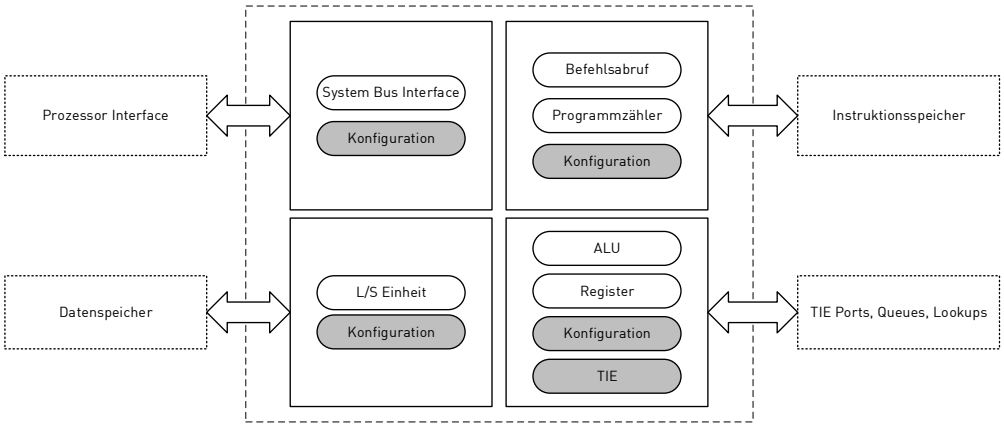

Abbildung 3.6: Xtensa LX4 Prozessor Blockschaltbild

Abgesehen von den Grundkomponenten des Prozessors sind alle weiteren Bestandteile optional und können abhängig vom Anwendungsfall eingesetzt werden. Beispielsweise kann das System Bus Interface, das der Kommunikation des Prozessors mit dem System dient, optional um ein *Prozessor Interface* (PIF) erweitert werden, um weitere Kommunikationskanäle zwischen Prozessor und Systembus zu realisieren. Weiterhin können der Befehlsabruf (*Instruction Fetch*) oder Programmzähler (*Programm Counter*) an unterschiedliche Formen von Instruktionsspeichern angeschlossen sein. Dazu zählen *Caches* genauso wie *RAMs* (*Random Access Memories*) für wahlfreien Zugriff oder *ROMs* (*Read-Only Memories*) als Festwertspeicher. Diese Teile des Prozessors sorgen für den Programmablauf und können Instruktionen bis zu einer Breite von 64 Bit (bei Nutzung von FLIX) verarbeiten. Diese unterschiedlichen Speichertypen können auch an die *Load/Store Einheit* angeschlossen werden, wobei ihre Anzahl auf zwei limitiert ist. Neben der dargestellten L/S Einheit lässt sich optional auch eine zweite Einheit einbringen, um die Speicherbandbreite zu verdoppeln. Der letzte Funktionsblock im Prozessorschema enthält neben der ALU (*Arithmetical-Logical Unit*) zur Verarbeitung von arithmetisch- logischen Befehlen und den vordefinierten Registern auch den von TIE eingebrachten Teil. Hier werden die zusätzlich etablierten Register und States genauso realisiert wie die programmierten Operationen zur Erweiterung des Instruktionssatzes. Sofern benötigt, stehen auch hier zusätzliche Kommunikationsschnittstellen über TIE Queues, TIE Ports und TIE Lookups zur Verfügung.
Neben den variablen, prozessorexternen Komponenten, lassen sich alle der genannten Funktionsblöcke im Xtensa LX4 problemspezifisch konfigurieren.

4 VERWANDTE ARBEITEN

Aufgrund der immer weiter steigenden Menge an Daten, die verarbeitet werden müssen, hat sich die Suche nach Beschleunigungsmöglichkeiten in unterschiedliche Richtungen entwickelt. Aus der in dieser Diplomarbeit vorrangig behandelten Hardwaresicht unterscheiden sich die Konzepte der Optimierung vor allem durch den Spezialisierungsgrad. Das Spektrum der Untersuchungen reicht hierbei von gezielten Ausnutzung bestimmter Funktionen in Standardhardware bis zu der Konzeption eines speziellen Rechners nur für Datenbanken. Um die vorliegende Arbeit zu den bereits durchgeführten Forschungen in Relation zu setzen, wird im Folgenden näher auf unterschiedlichen Ansätze eingegangen. Die Aufteilung des Abschnittes erfolgt hierbei nach den oben genannten Formen der Hardwarespezialisierung.

4.1 LAUFZEITOPTIMIERUNG UNTER NUTZUNG VON SIMD INSTRUKTIONEN

Große Prozessorhersteller wie AMD und Intel waren kontinuierlich auf der Suche nach neuen Möglichkeiten zur Beschleunigung ihrer Prozessoren Im Rahmen dieser Bemühungen analysierte Intel unterschiedliche Problemstellungen der Software aus den Bereichen der Multimediaverarbeitung. Hierzu gehörten insbesondere Grafik- und Videoverabeitung, Synthese und Kompression von Musik und Sprache sowie die Herausforderungen von Computerspielen an die CPU. Die Untersuchungen führten zu der Erkenntnis, dass alle rechenintensiven Teile der Software ähnliche Charakteristiken aufwiesen. So handelte es sich meistens um eine Vielzahl von Daten, die nacheinander von der gleichen Operation verarbeitet werden mussten. Diese Gemeinsamkeit ergab sich aus der Etablierung von speziellen Instruktionen für diese Art der Verarbeitung – Intels MMX Technologie (*Multimedia Extension*) [PW96]. Hierbei wurden einfache, immer wieder verwendete Algorithmen in einer separaten Instruktion zusammengefasst. Die neue Instruktion konnte nun nach dem SIMD Prinzip parallelisiert werden und durch die erreichte Verringerung der benötigten Takte einen Geschwindigkeitszuwachs erzielen [PWW97]. Dieses Prinzip wurde in den Folgejahren weiterentwickelt und ausgebaut. Durch den Erfolg dieser Herangehensweise folgten auch andere Hersteller mit der Erweiterung ihrer CPUs durch spezielle Instruktionen wie beispielsweise AMD mit *3DNow!* [OFW99]. Um noch mehr rechenintensive Anwendungen zu beschleunigen folgten eine Reihe von Erweiterungen wie beispielsweise die von Intel patentierten *SSE* (*Streaming SIMD Extensions*) mit den Ergänzungen *SSE2* und *SSE3*.
Insgesamt versuchten die Hersteller die Generalität ihrer RISC Prozessoren und damit die Kompatibilität zu Standardsystemen beizubehalten, indem sie auf die bereits etablierten Normen aufsetzen. Die stärkere Spezialisierung einiger Instruktionen für häufig vorkommende Problemstellungen stellt hierbei eine Ergänzung zur Verfügung, die die Entwickler für die Optimierungen ihrer

Algorithmen verwenden können.
Ein Beispiel für die hinzugefügten SIMD Operation ist die Verarbeitung von vier 32 Bit Gleitkommazahlen in einer Instruktion der Intel SSE Familie. Diese, in Abbildung 4.1 schematisch dargestellte Instruktion, verwendet zwei 128 Bit Operanden die jeweils aus vier Wörtern bestehen, welche mit unterschiedlichen Operationen verknüpft werden können. Um diese Operation äquivalent auf einem Prozessor ohne Instruktionssatzerweiterung durchzuführen, würde entsprechend die vierfache Laufzeit benötigt.

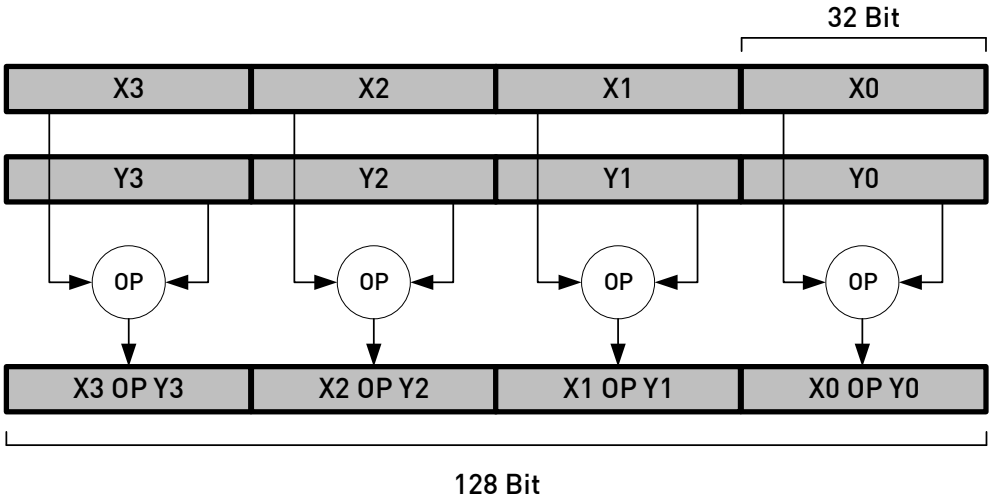

Abbildung 4.1: SIMD Beispielinstruktion von Intel

Dieser gesteigerte Datendurchsatz ist natürlich auch im Umfeld der Datenbanken interessant, da hier genauso wie bei den Multimedia-Anwendungen viele Daten mit der gleichen Operation verarbeitet werden müssen. Aus diesem Grund untersuchten Zhou und Ross 2002 in ihrer Veröffentlichung „*Implementing Database Operations Using SIMD Instructions*" Möglichkeiten der Nutzung von SIMD Instruktionen im Datenbankumfeld [ZR02]. Im Rahmen der häufig verwendeten Operationen in einer Datenbank wie beispielsweise sequentiellen Scans, der Aggregation oder bei Index Operationen wurden Optimierungsmöglichkeiten untersucht. Mit Hilfe der in Abbildung 4.1 dargestellten Instruktion vermuteten die Autoren für einfache Befehle im besten Fall eine Vervierfachung der Geschwindigkeit, da nun die Verarbeitung von zwei 128 Bit Wörtern, statt der bisher möglichen 32 Bit Wörter realisiert werden konnte. Allerdings konnte mit der größeren Verarbeitungsbreite auch auf *Wenn-Dann* Verzweigungen innerhalb des Quellcodes verzichtet werden. Um die Pipeline optimal auszulasten, versuchen moderne Prozessoren den wahrscheinlichsten Weg der Verzweigung zu ermitteln (*Branch Prediction*). Wenn diese Vorhersage jedoch falsch ist, kostet das durch die große Anzahl an Pipelinestufen in modernen Prozessoren zusätzliche Takte. Durch den vierfachen Datendurchsatz und die Eliminierung von Verzweigungen konnten die Autoren Beschleunigungen von 10% bis über 400% erreichen.

Mit der kontinuierlich steigenden Datenmenge wächst auch der Bedarf nach Kompression. Das sich auch hier neue Verarbeitungsgeschwindigkeiten erreichen lassen, haben Schlegel et al. in ihrer Veröffentlichung „*Fast Integer Compression using SIMD Instructions*" gezeigt [SGL10]. Der dort verfolgte Weg war das Ausnutzen der Pipelines moderner Prozessoren. Durch die hierbei erreichte Parallelität werden mit jeder Iteration des Algorithmus mehr Instruktionen ausgeführt. Das schlägt sich positiv in der Kenngröße *Instructions per Cycle* nieder und sorgt auf diese Weise für eine reduzierte Laufzeit. Neben diesem Weg des Nutzens moderner Prozessoren, gibt es auch Einsatzmöglichkeiten für spezielle SIMD Instruktionen. Da die Algorithmen jedoch ursprünglich

nicht für die Abarbeitung mit Befehlsparallelität gedacht waren, mussten die verwandten Kompressionstechniken zunächst konzeptionell für die Verwendung mit SIMD Instruktionen optimiert werden. Durch die hiermit realisierten Speicheranpassungen konnten Geschwindigkeitszuwächse zwischen 50% und 670% ohne Reduktion der Kompressionsleistung erreicht werden. Die Autoren machten zudem deutlich, dass diese Steigerungen durch Ausdehnung der Verarbeitungsbreite von 128 Bit auf 256 Bit weiter verbessert werden können.

Durch die immer weiter gesteigerte Verarbeitungsleistung der Prozessoren, wird die Bandbreite zum Hauptspeicher immer mehr der Flaschenhals einer datenintensiven Operation. Diese Limitierung kann jedoch durch den Einsatz Kompressionstechniken reduziert werden [LB12]. Die durch SIMD unterstützte Steigerung der Kompressionsleistung ist in jüngster Vergangenheit Gegenstand mehrerer weiterer Forschungsarbeiten, die die Bedeutsamkeit dieses Forschunggebietes unterstreichen [ZZSY13, LBK14]. Insgesamt haben die verschiedenen Untersuchungen gezeigt, dass die Beschleunigung unterschiedlicher Algorithmen durch Verwendung der erweiterten Instruktionssätze möglich ist. Abseits der bereits erwähnten Anwendungsfälle hat Hassaballah [HOM08] in seiner Veröffentlichung *„A Review of SIMD Multimedia Extensions and Their Usage in Scientific and Engineering Applications"* eine Vielzahl weiterer Nutzungsmöglichkeiten, beispielsweise aus den Bereichen der Verschlüsselung oder der Simulation, zusammengefasst.

Im speziellen Teilgebiet der Bitmap Indizes wurde die Verarbeitung der Daten mit speziellen Instruktionen bisher nur erwähnt und nie wirklich untersucht. In ihrer Publikation *„Notes on Design and Implementation of Compressed Bit Vectors"* erwähnten Wu et al. bereits 2001 das erste Mal die Instruktionserweiterungen in diesem Zusammenhang [WOSN01]. Sie machten hier darauf aufmerksam, dass die ersten Rechner mit MMX Technologie ausgestattet sind und das sie die Verarbeitung beschleunigen könnte. Einschränkend erkannten sie jedoch, dass hier nur einfache Kompressionsalgorithmen ohne Abhängigkeit der Daten untereinander beschleunigt werden könnten. Weitere Untersuchungen des Themas folgten nicht bis zur expliziten Erwähnung von Instruktionserweiterungen bei den Autoren von PLWAH [DP10]. Zum Beschleunigen der Identifikation von Sparse-Vektoren und der benötigten Bitposition nutzten sie die Funktionen *popcount* und *bitscan* aus AMD Prozessoren der 10. Generation bzw. Intels 2007 veröffentlichtem SSE4 Paket. Die Instruktionen können die Anzahl der auf 1 gesetzten Bits (popcount) bzw. die Position des ersten gesetzten LSB (bitscan) jeweils in einem Takt durchführen. Durch diese Beschleunigung werden die sonst benötigten Iterationen über den jeweiligen Wörtern verhindert und der Algorithmus daher beschleunigt. Diese Optimierung wirkt sich besonders dann aus, wenn viele Sparse-Vektoren in den unkomprimierten Daten vorkommen. Generell wird die Verarbeitungsgeschwindigkeit dadurch jedoch nur zum Teil beeinflusst, da die Autoren keinen Gebrauch von SIMD Funktionalitäten in ihrem Algorithmus machten.

Resümierend zeigen die Untersuchungen, dass Instruktionserweiterungen ein geeignetes Mittel zur Beschleunigung von Algorithmen darstellen. Jedoch basieren die Arbeiten auf einem Standardprozessor, dessen Instruktionserweiterungen fest sind und nur durch Anpassung von Algorithmen und Software effizienter genutzt werden konnten. Im Gegensatz dazu werden in der vorliegenden Arbeit sowohl Instruktionserweiterungen entworfen als auch die Software für bestehende Algorithmen aufeinander angepasst. Vor dem Hintergrund der Bitmap Indizes werden hier zudem die noch ungenutzten Potentiale der SIMD Instruktionen untersucht. Der hierfür verwende Basisprozessor verfügt nur über einen Bruchteil des Instruktionssatzes der angesprochenen CPUs, wodurch sich maßgebliche Vorteile in Bezug auf Energieverbrauch und Flächenbedarf ergeben. Die Autoren der erwähnten Publikationen konnten bereits mit vordefinierten Instruktionserweiterungen hohe Geschwindigkeitszuwächse erreichen. Da davon auszugehen ist, dass bei weiterer Spezialisierung der Instruktionen noch weniger Takte für die Verarbeitung der Algorithmen benötigt werden, stellt dies einen guten Ausgangspunkt für die Untersuchungen innerhalb dieser Diplomarbeit dar.

4.2 LAUFZEITOPTIMIERUNG UNTER NUTZUNG VON GPUS

Neben der Möglichkeit die Anwendungen für die CPU zu optimieren, hatte die Industrie der Computerspiele schon immer das Ziel der effizienten Nutzung des Grafikprozessors bzw. *Graphics Processing Unit* (*GPU*). Da auch die Anforderungen an die Leistung des Grafikprozessors immer weiter stiegen, entwickelten sie zunehmen Möglichkeiten, rechenintensive Prozesse von der CPU auf die GPU auszulagern. Hiermit begann 2000 die Zeit der GPGPUs (*General Purpose Computation on GPUs*) [SM13]. Die Industrie der Grafikkarten etablierte in dieser Zeit zunächst OpenGL und Direct3D als Grafikprogrammiersprachen. Um die Nutzung der schneller werdenden Grafikprozessoren einem größeren Publikum möglich zu machen, veröffentlichte NVIDA 2006 ihre an C angelehnte Programmiersprache *CUDA* die zum ersten Mal den Zweck hatte, generelle Verarbeitungsaufgaben auf der GPU zu berechen. 2007 folgte dann die herstellerunabhängige Version *OpenCL* (Open Computing Language). Eine GPU besteht aus vielen Recheneinheiten innerhalb von SISD Kernen, die in Gruppen zusammengeschaltet werden und *Streaming Multiprocessors* (SM) bilden. Grafikkarten haben mehrere dieser SM, die jeweils über kleine Caches verfügen, die alle mit dem großen Speicher der Grafikkarte verbunden sind. Die Datenverarbeitung mit Grafikkarten ist daher vor allem von hoher Parallelität gekennzeichnet. Sofern es sich nicht um integrierte Grafikkarten handelt, müssen jedoch alle zu verarbeitenden Daten zunächst von aus dem RAM des Systems (*Host*) auf in den Grafikspeicher transferiert werden, um diese zu verarbeiten.

Bereits vor Veröffentlichung der an C angelehnten Sprachen, untersuchten 2004 die Forscher Govindaraju et al. in ihrer Arbeit *„Fast Computation of Database Operations using Graphics Processors"* die Möglichkeiten der Nutzung von GPU für Datenbankanwendungen [GLW+04]. Ihr Fokus zur Laufzeitoptimierung lag dabei sowohl auf allgemeinen Datenbankqueries als auch auf den Befehlen der Selektion und Aggregation mit hochparallelen GPUs. Sie ermittelten hierbei 20- bis 40-fache Geschwindigkeitsverbesserungen bei der Verarbeitung von realen Datensätzen mit 1 Mio. Elementen. Insgesamt sprachen sich die Forscher für die weitere Untersuchung von GPUs als Co-Prozessoren aus, da Leistungsverbesserungen bereits mit Hardware erreicht werden können, die nicht im High-End Bereich liegt. Grafikkarten werden zukünftig noch mehr Leistung und Parallelität ermöglichen und so weitere Optimierungsmöglichkeiten mit GPUs erschließen.

Nach der Einführung der neuen Programmiersprachen und mit steigender Leistung der Grafikkarten, wurden diese immer mehr zum Forschungsschwerpunkt für Anwendungsbeschleunigungen. Das spezielle Teilgebiet der Kompression von Bitmap Indizes untersuchten Andrzejewski und Wrembel erstmals 2010 in ihrer Veröffentlichung *„GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid"* [AW10]. Hierfür wählten sie WAH als Kompressionsalgorithmus, dessen Teilschritte sie für die Parallelisierung auf Grafikkarten optimierten. Mit der auf CUDA basierenden Implementierung erreichten sie Faktoren der Leistungsverbesserung im Bereich von 3,8 bis 5,6 im Vergleich zu ihren Algorithmen, die nur auf der CPU verarbeitet wurden. Als größtes Problem stellte sich hierbei der Datentransfer zwischen Host-System und Grafikkarte heraus, da dieser verglichen mit der Operation relativ viel Laufzeit beanspruchte.

Zu diesem Ergebnis kamen 2013 auch Fusco et al. in ihrer Publikation *„Indexing Million of Packets per Second using GPUs"* [FVDD13b]. Sie suchten vor dem Hintergrund von Sicherheits- oder Forensikbetrachtungen nach Möglichkeiten, die steigende Datenlast auf Netzwerken weiterhin in Echtzeit auszuwerten. Dieses Ziel der Indexerstellung auf Netzwerkdaten wollten sie mit kostengünstiger, handelsüblicher Hardware (*commodity Hardware*) bei einer Datenrate von 10 GBit/s erreichen. Wie die bereits oben genannte Untersuchung setzten sie hierbei auf den WAH Kompressionsalgorithmus, führten die Untersuchungen aber parallel mit PLWAH durch. Ihre Ausgangsdaten waren 320 Mio. Bits, die gleichverteilt waren und mit den Kardinalitäten zwischen 256 und 65536 synthetisch erzeugt wurden. Die Verarbeitung auf der Grafikkarte lief bei der Implementierung um dem Faktor 20 schneller, als auf der CPU, jedoch wurde die Leistung durch notwendige Datentransfers zwischen Host und GPU auf Faktor 10 reduziert. Im Ergebnis hat auch diese Untersuchung deutliche Geschwindigkeitsverbesserungen erreicht. Zur Überwindung des Flaschenhalses der Datenübertragung empfehlen die Autoren die Verwendung von besseren

Kompressionsalgorithmen wie PLWAH in ihrem Fall, da sich die Leistungsbegrenzung dadurch reduzieren lässt.

Zusammenfassend kann festgestellt werden, dass die zitierten Untersuchungen gezeigt haben, dass durch die hohe Parallelität der GPUs deutliche Geschwindigkeitssteigerungen im Datenbankumfeld möglich sind. Mit der Nutzung von Parallelität zu Optimierungszwecken befasst sich auch die vorliegende Arbeit mit Beschränkung auf SIMD mit einer Verarbeitungseinheit. Mit diesem Ansatz wird nicht nur wesentlich weniger Energie bei der Verarbeitung in Anspruch genommen, sondern durch die Nutzung des Xtensa Prozessors entfällt auch der Flaschenhals des Datentransfers zwischen System und Grafikkarte.

4.3 LAUFZEITOPTIMIERUNG UNTER NUTZUNG VON FPGAS

Neben den im vorherigen Abschnitt erwähnten Untersuchungen zur Nutzung von Standardhardware, gibt es auch Forschungsarbeiten, die Leistungssteigerungen durch Anpassung von Hard- und Software untersuchen. Ein Weg dieser Hardwareanpassung stellt das Programmieren und Konfigurieren von FPGAs (*Field-Programmable Gate Arrays*) dar. Die 1985 erstmals von Freeman und Vonderschmitt hergestellte Hardware, machte die mehrfache Programmierung von Chipverhalten möglich [SM13]. Sie bestehen aus einer Kette von konfigurierbaren Logikblöcken, die untereinander verbunden sind. FPGAs beinhalten weiterhin Interfaces für den externen Datentransfer und Möglichkeiten zum Zwischenspeichern von Daten. Aufgrund der Konfigurierbarkeit haben FPGAs den Nachteil, einer relativ geringen Taktfrequenz im Vergleich zu normalen CPUs. Damit einher geht jedoch auch ein geringerer Energieverbrauch beim Vergleich dieser Systeme.

Der verringerte Energieverbrauch und die steigende Anzahl der Mehrprozessorsysteme machten FPGAs 2009 zum Forschungsgebiet von Müller et al. [MTA09]. Sie untersuchten in ihrer Veröffentlichung *„Data Processing on FPGAs"* zunächst, wie die Datenverarbeitung generell von der Nutzung der FPGAs profitieren kann. Insbesondere unter dem Aspekt der geringeren Taktrate im Vergleich zu CPUs musste ein maßgeblicher Geschwindigkeitsvorteil durch die effiziente Anpassung der Logik erreicht werden, um zunächst die Laufzeit von normalen CPUs zu erreichen. Durch die erreichte Parallelität bei der Verarbeitung konnten die Autoren äquivalente Laufzeiten zur Verarbeitung auf Standardprozessoren erreichen. Noch im selben Jahr legten die Autoren ihre Veröffentlichung *„FPGA: What's in it for a Database?"* mit der Untersuchung von Datenbankanwendungen in diesem Kontext vor [MT09]. Die Forscher gehen davon aus, dass FPGAs bald als zusätzliche Verarbeitungseinheit in handelsübliche Computer integriert werden. In Ihrer Publikation wollen Müller und Teubner deswegen vor allem über das Potential von FPGAs im Kontext der Datenbanken aufklären. Da hier besonders datenintensive Anwendungen zu vermuten sind, empfehlen die Forscher vor allem das Nutzen der Parallelität in FPGAs um einen hohen Datendurchsatz für die Verarbeitung zu erzeugen und damit in Konkurrenz zur traditionellen Verarbeitung auf der CPU zu treten.

Den Einsatz von FPGAs als Co-Prozessor schlägt auch Langi in seiner Publikation *„An FPGA Implementation of a Simple Lossless Data Compression Coprocessor"* vor [Lan11]. Mit dem Ziel der Entlastung des Hauptprozessors, schlug er den Einsatz von FPGAs zur Kodierung und Dekodierung nach dem Rice Verfahren vor. Die erfolgreiche Implementierung des Verfahrens beruht hierbei auch auf der erzeugten Verarbeitungsparallelität des Algorithmus, wie bereits in anderen Untersuchungen. Zum Beispiel wird dieses Ergebnis wird auch von Shi et al. in ihrer Veröffentlichung *„FPGA Acceleration for Intersection Computation in Frequent Itemset Mining"* gestützt [SQW13]. Diese aktuellere Arbeit aus dem Datenbankumfeld beschreibt die Laufzeitoptimierung durch einen parallelisierbaren Algorithmus zum Auffinden wiederkehrender Muster (Eclat). Der hierbei erreichte Faktor der Geschwindigkeitssteigerung zwischen 6 und 26,7 im Vergleich zur Referenzimplementierung verdeutlicht die Möglichkeiten parallelisierter Verarbeitung im Datenbankumfeld.

Zusammenfassend zeigen die Untersuchungen von FPGA Implementierungen, genauso wie die zitierten Arbeiten aus dem vorherigen Kapitel, dass sich mit Hilfe erhöhter Parallelität deutliche Optimierungspotentiale für die Algorithmen ergeben. Verglichen mit den Hardwareelementen aus dem vorherigen Kapitel brauchen die FPGAs durch die geringe Taktfrequenz jedoch nur einen Bruchteil der Energie. Um trotzdem zu ähnlichen oder verbesserten Laufzeiten zu kommen, ist sowohl eine Anpassung der Software auf die Hardware notwendig, als auch in entgegengesetzter Richtung. Durch die immer mögliche Rekonfigurierung der FPGAs konnten die Hard- und Software optimal aufeinander abgestimmt werden. Dieses Prinzip des Hardware/Software Codesigns wird auch in der vorliegenden Arbeit aufgegriffen. Allerdings führt die Entwicklung in einem erweiterten Instruktionssatz des Prozessors, der nicht mehr veränderlich ist. Dieses Prinzip ist ähnlich zu den anwendungsspezifischen integrierten Schaltungen bzw. *ASICs* (*application-specific integrated circuit*), da eine vorhandene Prozessorbasis um neue, anwendungsspezifische Instruktionen erweitert wird. Die Flexibilität der FPGAs führt, verglichen mit ASICs, zu einigen Nachteilen. So stellten Kuon und Rose 2007 in ihrer Veröffentlichung *„Measuring the Gap Between FPGAs and ASICs"* fest, dass FPGAs im Vergleich mit ASICs bis zu 35-mal größer und 4,6-mal langsamer sind und darüber hinaus 14-mal mehr Energie verbrauchen [KR07]. Da die Flexibilität für die zu untersuchenden Algorithmen der Bitmap Indizes nicht nötig ist, kann die TIE Implementierung hier kleiner, schneller und energieeffizienter sein, als dies mit FPGAs möglich wäre.

4.4 LAUFZEITOPTIMIERUNG UNTER NUTZUNG VON SPEZIELLEN DATENBANKSYSTEMEN

Neben der Möglichkeit zur Anpassung einzelner Systemkomponenten zur Erhöhung der Abarbeitungsgeschwindigkeit im Datenbankumfeld, wurde auch die Idee eines spezialisierten Komplettsystems untersucht. Hsiao veröffentlichte diese Idee bereits 1979 in seiner Publikation *„Data Base Machines are Coming, Data Base Machines are Coming!"* [Hsi79]. Er definierte Datenbankmaschinen als ein System, dass viele der grundlegenden Aufgaben einer Datenbank in Hardware statt wie bisher in Software abbildet. Das Hauptargument bildete dabei für ihn die Performance, da die klassische Architektur nicht für das Verarbeiten großer Datenmengen ausgelegt war. Beispielsweise erforderten die kleinen Hauptspeicher, dass häufige Kopiervorgänge zwischen Festplatte und RAM notwendig waren, um einen großen Datensatz komplett zu verarbeiten. Er schlug ein System mit mehreren Prozessoren vor, die zentral gesteuert werden und jeweils über eigene Speicher verfügen. Die dadurch ermöglichte Parallelität sollte maßgeblich zur Beschleunigung beitragen.
Diese Idee wurde allerdings auch von Boral und DeWitt untersucht und vier Jahre später bereits kritisch hinterfragt [BD83]. In ihrer Publikation *„Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines"* gehen sie auf mögliche Schwachstellen des Ansatzes ein. Die angedeutete Parallelarchitektur von CPUs und Festplatten halten sie beispielsweise für fragwürdig, da die Festplattentechnologie die dafür benötigte Bandbreite nicht zur Verfügung stellen kann. Weiterhin sehen sie die Entwicklung von spezialisierten Prozesssoren kritisch, da die Implementierung aller benötigten Funktionen zeitaufwändig ist und die Technologie der Standardprozessoren in dieser Zeit auch schneller wird, sodass sich der Aufwand am Ende nicht lohnt. Aus diesen beiden Gründen sind sie davon überzeugt, dass der Einsatz von bis zu 3 Standardprozessoren bereits die Grenze der I/O Bandbreite erreicht und die Entwicklung einer noch stärker parallelisierten Maschine an der gleichen Grenze scheitert und daher nicht zweckmäßig ist.

Tatsächlich sorgten in den Folgejahren die konstant wachsende Verarbeitungsgeschwindigkeit der Rechnerkomponenten dafür, dass das Thema der Datenbankmaschinen weniger relevant wurde.

Da aber mittlerweile die Prozessortechnologie an ihre Grenzen stößt, gibt es nun Untersuchungen zur Herstellung eines speziellen Prozessors für Datenbankinstruktionen. So publizierten Wu et al. in ihrer Veröffentlichung *„Q100: The Architecture and Design of a Database Processing Unit"* einen ASIC speziell für Datenbankinstruktionen [WLP+14]. Ihr vorgeschlagener Prozessor implementiert grundlegende Datenbankfunktionalitäten wie *select*, *join* oder *aggregate* und achtet hierbei auf ein besonders gutes Verhältnis zwischen Energieverbrauch und Abarbeitungsgeschwindigkeit. Durch die effiziente Nutzung von Parallelität und Pipelining erreichen sie hierbei durchschnittlich Geschwindigkeitssteigerungen um Faktor den 70, verglichen mit der üblichen Softwareimplementierung und Nutzung der Standardprozessoren. Weiterhin verbraucht ihr mit Synopsys synthetisierter Chip mit 32nm Design dafür etwa 300-mal weniger Energie und 10 Prozent der Fläche verglichen mit einem Intel Xeon Prozessor. Der entwickelte Chip kann hierbei jedoch nur mit 315 Mhz getaktet werden, da der längste kritische Pfad einer Instruktion dies begrenzt.

Darüber hinaus publizierten Arnold et al. in ihrer Arbeit *„An application-specific instruction set for accelerating set-oriented database primitives"* die Beschleunigung von *sorted-set* Operationen [AHF+14]. Diese Arbeit wurde mit den gleichen Ausgangsbedingungen angefertigt, wie die vorliegende Diplomarbeit. Die angefertigte Implementation erreicht die gleiche Verarbeitungsgeschwindigkeit wie moderne Prozessoren, jedoch mit 960-mal weniger Energieverbrauch.

Insgesamt zeigen die Untersuchungen, dass durch Parallelisierung und Pipelining die Optimierung von Datenbankoperationen möglich ist. Dies betrifft sowohl die Verarbeitungsgeschwindigkeit, als auch Flächen- und Energieverbrauch. Die vorliegende Arbeit reiht sich damit in die Publikationen zur Anpassung von Prozessorinstruktionen ein und untersucht vor diesem Hintergrund im Speziellen die Kompression von Bitmap Indizes.

5 IMPLEMENTIERUNG

Die Untersuchung der einzelnen Algorithmen und ihrer Optimierungspotentiale wird gemäß dem in Abschnitt 3.1 erläuterten TIE Arbeitsablauf schrittweise realisiert. Hierzu bedarf es zunächst der Entwicklung einer C-Implementierung der vorgestellten Algorithmen, die dann auf Bottlenecks untersucht werden können, um diese dann durch neue Instruktionen zu beschleunigen. In diesem Kapitel werden zunächst die allgemeinen Charakteristiken der Implementierung erläutert und im Anschluss für jeden Algorithmus im Detail betrachtet. Die Algorithmen werden dafür an einem einführenden Beispiel demonstriert und in eine Ablaufanalyse überführt. Danach werden die Optimierungspotentiale des Codes untersucht und in neue Instruktionen überführt. Anschließend werden die Instruktionen für jeden Algorithmus separat beschrieben. Das generelle Vorgehen entspricht hierbei den folgenden Schritten:

Entwicklung des C-Quellcodes

Die Basis der TIE Entwicklung für die unterschiedlichen Algorithmen besteht zunächst jeweils aus deren Implementierung in C. Der Quellcode wurde hierbei vollständig selbst entwickelt und die funktionale Korrektheit anhand jeder Algorithmusdefinition überprüft. Hierfür wurden zunächst Kompression und Dekompression der drei Algorithmen Implementiert. Nachdem die Tests der Implementierung korrekte Ergebnisse lieferten, wurden die Funktionen zur Verarbeitung von 2 komprimierten Indizes für jeden Algorithmus entwickelt. Die Verarbeitung umfasst hierbei die Verknüpfung der zwei Indizes mit den logischen Operatoren *AND*, *OR* und *XOR* um jeweils die Schnittmenge, Vereinigungsmenge oder Symmetrische Differenz zwischen den beiden Indizes zu erzeugen. Am Ende dieser Phase existierten also C-Implementierungen aller drei Algorithmen für Kompression, Dekompression und Verarbeitung des jeweiligen Algorithmus. Zur Laufzeitoptimierung wurden grundlegende Optimierungen der C-Implementierung vorgenommen. Laufzeitoptimierungen des C-Codes abseits der TIE Optimierungen beispielsweise in Hinblick auf parallelisierte Verarbeitung wurde nicht betrachtet.

Test der Implementierung

Die syntaktische und semantische Korrektheit wurde jeweils über automatisierte Systemtests sichergestellt. Diese Tests beinhalten die Überprüfung aller Funktionen für den jeweiligen Algorithmus. Hierfür wurden die unkomprimierten Bitmaps X und Y zunächst direkt in unkomprimierter Form mit der jeweiligen Operation verknüpft. Im Anschluss daran wurden die unkomprimierten Bitmaps mit Hilfe der Implementierungen komprimiert (*com*() Funktion), verarbeitet (*op*() Funktion) und dekomprimiert (*decom*() Funktion). Der letzte Schritt der Dekompression ist notwendig, weil das Ergebnis von *op*() ein komprimierter Bitmap Index

ist. Für einen erfolgreichen Test muss demzufolge folgende Gleichung (5.1) erfüllt sein:

$$decom(\ op(com(X), com(Y))\ ) \ = \ X \circ Y, \text{ wobei } \circ \in \{AND, OR, XOR\} \tag{5.1}$$

Mit der Kontrolle ob jedes Bit der linken Seite der Gleichung (5.1) dem Bit auf der rechten Seite gleicht, ist ein Test gegeben, der die korrekte Implementierung aller Teilfunktionen mit einem Durchlauf überprüft. Dabei wurde darauf geachtet, dass die Testdaten eine Vielzahl der möglichen Ein- und Ausgangsdaten darstellen, um das System vollständig zu testen.

TIE Arbeitsablauf

Der funktional korrekte Quellcode konnte nun schrittweise in Instruktionen überführt werden, wobei zunächst die beim Profiling identifizierten Hot-Spots ersetzt wurden. Nach mehreren Durchläufen des TIE Entwicklungs- und Optimierungszyklus existiert für jeden Algorithmus eine Instruktion, als Ersatz für die komplette Verarbeitungsfunktion. Diese Instruktionen wurden erneut durch die beschrieben Systemtests auf Korrektheit überprüft und zusätzlich durch Instruktionsspezifische Tests verifiziert. Hierbei wurde jede der Instruktionen mit vorab konstruierten Bitmap Indizes überprüft und das Ergebnis mit dem zu erwarteten Ergebnis verglichen. Weiterhin wurden die TIE Implementierung auf Designaspekte wie Taktfrequenz und Fläche untersucht und optimiert um schließlich in der finalen Version implementiert zu werden. An dieser Stelle schließt sich das nächste Kapitel 6 an, in dem die durchgeführten Optimierungen miteinander verglichen und in Referenz gesetzt werden.

5.1 ALLGEMEINE BETRACHTUNG DER TIE IMPLEMENTIERUNG

Das Ziel der vorliegenden Diplomarbeit ist Beschleunigung der Verarbeitung von Kompressionsalgorithmen für Bitmap Indizes. Diese Beschleunigung wird durch eine Verringerung der benötigten Takte für jede Verarbeitung erreicht. Daraus ergibt sich das Ziel, dass die hierfür notwendigen Instruktionen einen möglichst großen Teil des ursprünglichen C-Quellcodes ersetzen. Die einzelnen Operationen lassen sich in die Metagruppen Lade-, Verarbeitungs- und Speicheroperationen einteilen. Bei der Entwicklung der Instruktionen wird das Ziel verfolgt, diese Metagruppen mit möglichst wenigen Instruktionen abzubilden, um die Takte der gesamten Verarbeitung zu reduzieren. Die resultierenden Funktionen in C bilden, im Anschluss an die Instruktionsentwicklung, die Funktionalität mit Hilfe der Intrinsics ab. Der Aufbau dieser Funktionen folgt immer dem gleichen Schema, das in Tabelle 5.1 näher erläutert wird.

Initialisierung: Vor dem Start der eigentlichen Verarbeitung bedarf es der Intialisierung der verwendeten States. Die mit `add_read_write` erzeugten States werden von TIE automatisch mit der Möglichkeit initialisiert, aus dem C-Code geschrieben oder gelesen zu werden. Diese 32 Bit breiten Lese- (`RUR`) und Schreibefunktionen (`WUR`) werden in der Initialisierungsphase genutzt um beispielsweise die Größe der zu verarbeitenden Datenströme oder deren Position in den Speicher des Prozessor zu laden. Dieser Vorgang muss nur einmal zu Beginn der Verarbeitung erfolgen, da die Daten im Folgenden innerhalb des Prozessors gehalten und verändert werden. Bei der Verwendung von Ladeoperationen mit Shadow Registern gehört zudem der Aufruf der Instruktionen zum Vorladen in diesen Bereich.
Hauptschleife Da alle entworfenen Instruktionen ein oder zwei Eingangsdatenströme verarbeiten, besteht der Hauptteil der Algorithmus zunächst aus der Iteration über alle Eingangselemente. Die Operanden werden im Rahmen dieser Schleife geladen, verarbeitet und gespeichert.
Nachbereitung Im Anschluss an die Hauptschleife sind eventuell weitere Schritte nötig, um die Abarbeitung gemäß den Algorithmen zu komplettieren. Das tritt zum Beispiel auf, wenn durch das eingesetzte Pipelining der Verarbeitung in der Hauptschleife noch nicht alle Elemente jede Stufe durchlaufen haben. Ein weiterer Aspekt ist das Speichern der letzten Elemente. Da alle optimierten TIE Instruktionen ihre Speicherzugriffe 128 Bit aligned durchführen, kann es bei weniger als vier zu schreibenden Wörtern auftreten, dass die letzten Elemente noch nicht von der Hauptschleife geschrieben wurden. In diesem Fall werden die Elemente mit einer Nachfolgenden Schleife aus den States geladen. Die Umsetzung des Speichervorgangs durch separate TIE Instruktionen ist für maximal drei Elemente zu aufwändig, da separate Chipfläche in Anspruch genommen werden müsste. Aus diesem Grund erfolgt das Laden innerhalb von C durch Nutzung der Leseoperationen, die bereits bei der Initialisierung erwähnt wurden.

Tabelle 5.1: Allgemeiner Aufbau der C-Funktionen mit TIE Intrinsics

Bei der späteren Nutzung der TIE Instruktionen wird davon ausgegangen, dass es sich um eine sehr große Datenmenge handelt, die verarbeitet werden soll. Aus diesem Grund stellt die Laufzeit der Hauptschleife den wichtigsten Aspekt bei den TIE Optimierungen dar. Während Vor- und Nachbereitung nur einige Takte in Anspruch nehmen, iteriert die Hauptschleife durch die großen Datenströme sehr häufig und ist entsprechend für den Großteil der Laufzeit verantwortlich. Da sowohl die Initialisierung als auch die Nachbereitung vernachlässigbar kleine Verarbeitungsanteile haben, wird auf ihre Erwähnung bei Detailbetrachtungen der TIE Implementierung verzichtet.
Alle Implementierungen werden zudem mit dem Compiler Optimierungslevel drei kompiliert, sodass der Code auf Laufzeit optimiert wird, auch wenn er im Gegenzug etwas größer wird. Einen wesentlichen Aspekt dieser Optimierung stellt das *Loop Unrolling* dar. Dabei wird die eigentliche Schleife durch die Aneinanderreihung der enthaltenen Befehle ersetzt. Diese Optimierung führt dazu, dass kein zusätzlicher Takt zum Überprüfen der Schleifenvariable benötigt wird, sondern dies bereits vorher durch die Anzahl der Iterationen festgelegt wird. Allerdings steht diese Möglichkeit der Optimierung nur bei *for* Schleifen zur Verfügung, da bereits vor der Ausführung des Programms die Anzahl der notwendigen Schleifendurchläufe gegeben ist. Bei den vorgestellten Algorithmen enspricht die Mindestanzahl der Schleifendurchläufe der maximalen Elementanzahl in den Eingangsdatenströmen. Durch die Kompression kann es jedoch vorkommen, dass mehr Schleifendurchläufe benötigt werden. Für diesen Fall wird der Hauptschleife im Rahmen der Nachbereitung eine `do-while` Schleife nachgeordnet, um eventuell verbliebene Elemente abzuarbeiten.

5.2 WAH-OPERATION MIT TIE-ENTWICKLUNGSSTUFEN

Der WAH Kompressionsalgorithmus bildet die Grundlage der anderen, untersuchten Algorithmen. Er wurde im Rahmen der vorliegenden Arbeit als erstes implementiert. Hierbei wurde die Entwicklungs- und Optimierungsschleife des Arbeitsablaufes von TIE mehrmals durchlaufen, da sich die Implementierung immer weiter entwickelte und die Anpassung an den Algorithmus immer effizienter wurde. Um den Verlauf dieser Untersuchung nachzubilden, wird die Entwicklung der WAH Instruktionen mit den unterschiedlichen Revisionen betrachtet. Die Algorithmen der Folgekapitel basieren dann auf der am stärksten optimierten Version, sodass die Entwicklungsstufen nur im Fall von WAH aufgezeigt werden.

5.2.1 Beispielhafter Ablauf

Um zunächst die Vorgehensweise auf Basis des Algorithmus zu verdeutlichen, wird in Tabelle 5.2 ein Beispiel beschrieben:

	Element 1	Element 2	Element 3	Element 4	Element 5	Element 6
Index 1	C0000001	C0000001	C0000001	80000001	80000001	79683F41
Index 2	C0000001	80000001	00003780	80000001	00000210	090015F1
Iteration 1	C0000001					
Iteration 2	C0000001	80000001				
Iteration 3	C0000001	80000001	00003780			
Iteration 4	C0000001	80000001	00003780	80000001		
Iteration 5	C0000001	80000001	00003780	80000002		
Iteration 6	C0000001	80000001	00003780	80000002	09001541	
Schnittmenge	C0000001	80000001	00003780	80000002	09001541	

Tabelle 5.2: Beispielablauf des WAH-Algorithmus mit AND Verknüpfung (HEX)

Beispielhaft wird in Tabelle 5.2 die UND-Verknüpfung von zwei WAH-komprimierten Bitmap Indizes in hexidezimaler Schreibweise demonstriert. Das Beispiel startet mit den zwei Bitmap Indizes *Index 1* und *Index 2* als Eingangswerten. Mit der ersten Iteration wird das erste Element der beiden Indizes ausgewertet wobei die UND-Verknüpfung von zwei 1-Fills in einen 1-Fill ergibt. In der zweiten Iteration wird ein 1-Fill mit einem 0-Fill verknüpft, was ein 0-Fill als Ergebnis liefert. Entsprechend werden die dritten Elemente beider Indizes zu einem Literal und die vierten zu einem 0-Fill kombiniert. Bei der Verarbeitung der fünften Elemente resultiert jedoch aus der Verknüpfung eines 0-Fills mit einem Literal wieder ein 0-Fill, weshalb der Lauflängenzähler im Ergebnis inkrementiert wird, statt ein neues Wort zu schreiben. Abschließend werden in der sechsten Iteration zwei Literale zu einem neuen verarbeitet und, da beide Indizes vollständig abgearbeitet sind, endet der Algorithmus nach der sechsten Iteration mit dem Ergebnis. Die Länge der Eingangsindizes von $6 \cdot 31 = 186$ Bit wurde hierbei auch im Ergebnis beibehalten - durch die Zusammenführung von zwei 0-Fills wurde jedoch der Kompressionsgrad erhöht, sodass nur noch fünf statt sechs Wörter zum Speichern benötigt werden.

5.2.2 Ablaufanalyse und TIE Optimierung

Nachdem der schrittweise Ablauf des Algorithmus im vorherigen Unterkapitel verdeutlicht wurde, wird nun die Umsetzung des Ablaufes in einen Programmcode dargestellt. Das Programm besteht aus einer Hauptschleife, der über alle Elemente der Eingangsvektoren iteriert und sie

jeweils miteinander verknüpft. Durch die Fill Elemente kann es jedoch vorkommen, dass sich die Pointer auf den Ausgangsdaten in unterschiedlicher Geschwindigkeit bewegen. Weiterhin ist es notwendig, dass Zwischenergebnis jeder Iteration beim Speichern mit dem Element zu vergleichen, dass nach der vorherigen Iteration gespeichert wurde, weil beim Speichern der gleichen Typen von Fill Wörtern, diese häufig zu einem Wort kombiniert werden können. Nach der Iteration über alle Elemente bricht die Iteration ab und der Ergebnisvektor ist komplett. Die folgenden Punkte verdeutlichen den Ablauf:

Ablauf des WAH Algorithmus
Für alle komprimierten Wörter `x` aus Bitmap Index `A` und `y` aus Bitmap Index `B`:

1. Laden
   - Wenn `x` keine weiteren Fills enthält, dann lade nächstes Element aus `A`
   - Wenn `y` keine weiteren Fills enthält, dann lade nächstes Element aus `B`
2. Verarbeiten
   - Wenn `x` und `y` Fills sind, dann
     (a) Ermittle das Minimum beider Lauflängenzähler `MIN`
     (b) Verknüpfe die Typen der Fills zu `ERG_TYP`
     (c) Setze `TEMP=ERG_TYP+MIN`
     (d) Verringere Fills um `MIN`
   - Wenn `x` und `y` ein Fill und ein Literal sind, dann
     (a) Verknüpfe Fill mit Literal zu `ERG`
     (b) Setze `TEMP=ERG`
     (c) Dekrementiere Fill
   - Wenn `x` und `y` zwei Literale sind, dann
     (a) Verknüpfe Literale zu `ERG`
     (b) Setze `TEMP=ERG`
3. Speichern
   - Wenn temporäres Ergebnis `TEMP` ein Fill ist und das Ergebnis der vorherigen Iteration ein Fill des gleichen Typs war und beide Lauflängen bei der Addition keinen Überlauf erzeugen, dann speichere die addierten Lauflängenzähler im alten Element
   - Sonst hänge das neue Ergebnis an die Ergebnisindex `C` an

Nach dem Ablauf dieser Schleife wurden also die komprimierten Bitmap Indizes `A` und `B` mit der jeweiligen Operation verknüpft und in den Bitmap Index `C` als Ergebnis geschrieben. Diese Ablaufanalyse war die Vorlage für die Implementierung in C und wurde, um die folgenden Untersuchungen zu ermöglichen, mit dieser Funktionalität implementiert. Das Ergebnis dieser Phase war ein funktionsfähiger Quellcode, der mit den üblichen Compilern auf den handelsüblichen Prozessoren lauffähig ist. In den folgenden Unterkapiteln wird auf die stufenweise Optimierung des Ursprungscodes eingegangen.

Entwicklungsstufe 1

Den offensichtlichsten Ansatz zur Beschleunigung des Quellcodes stellt das gemeinsame Ausführen mehrerer Bestandteile in einer Instruktion dar. Während des ersten Entwicklungszyklus entstanden hierfür sechs neue Instruktionen, die zusammen den vorher beschriebenen Ablauf funktional korrekt umsetzen konnten. In der folgenden Tabelle 5.3 werden die unterschiedlichen Instruktionen kurz vorgestellt:

Instruktion	Beschreibung
`checkEnd`	Überprüft auf Basis der Bitmap Index Längen, ob Iteration fortgesetzt wird.
`CondLoadX`	Lädt ein neues 32 Bit Wort zum Verarbeiten, sofern das aktuell enthaltene Wort bereits vollständig verwendet wurde. Das ist immer dann der Fall, wenn das Wort entweder ein Literal oder ein Fill mit dem Lauflängenzähler auf Null ist.
`CondLoadY`	Ladeoperation äquivalent zu `CondLoadX` aber für das zweite, zu verarbeitende Wort und somit in einen anderen Zielstate.
`TIEwahAND`	Verarbeitet die zuvor geladenen Wörter nach WAH Algorithmus mit der UND-Verknüpfung.
`calc`	Überschreibt auf Basis des Verknüpfungsergebnisses die States der Eingangswörter und schreibt das Ergebnis in einen State.
`storeZ`	Schreibt den State des Ergebnisses zurück in den Speicher.

Tabelle 5.3: Instruktionsbeschreibung für die WAH Operation - Version 1

Auf Basis dieser Instruktionen kann der C-Quellcode zur Umsetzung des Algorithmus folgendermaßen angepasst werden:

```
while(checkEnd())
{
      CondLoadX();
      CondLoadY();
      TIEwahAND();
      calc();
      storeZ();
}
```

Hierbei werden die Intrinsics in üblicher Weise verwendet und damit die Verarbeitung eines Wortes umgesetzt. Es wird bei jeder Iteration zunächst überprüft, ob es weitere Elemente für die Verarbeitung gibt. Sofern das der Fall ist, werden zwei 32 Bit Wörter geladen, verarbeitet und wieder geschrieben. Diese Implementierung nimmt also die Optimierung gemäß Fusion vor, da mehrere Funktionalitäten in einer Instruktion abgebildet werden. In der Folge wird mit Iteration genau ein Wort verarbeitet wird, wofür insgesamt sechs Instruktionen notwendig sind.

Entwicklungsstufe 2

Um eine weitere Beschleunigung gegenüber der ersten Entwicklungsstufe zu erreichen, kann die größere Speicherbandbreite des Xtensa LX4 verwendet werden. Die 128 Bit breite Schnittstelle der Load/Store Einheit kann sowohl bei Lade- als auch bei Speichervorgängen die vierfache Menge der bisher übertragenen Daten transportieren. Weiterhin stellt die weitere Kombination der

Befehle einen offensichtlichen Weg dar, um die notwendigen Instruktionen für die Verarbeitung eines Wortes zu reduzieren. Dafür werden die States für Ein- und Ausgangswerte der Verarbeitung vergrößert und gemäß der folgenden Tabelle 5.4 angepasst:

Instruktion	Beschreibung
`checkEnd`	Überprüft auf Basis der Bitmap Index Längen, ob Iteration fortgesetzt wird.
`ldXstream`	Lädt vier neue Wörter für die Verarbeitung in Abhängigkeit davon, ob im Zielstate 128 Bit frei sind.
`ldYstream`	Ladeoperation äquivalent zu `ldXstream` aber mit dem Zielstate für den zweiten Operanden.
`TIEwahAND`	Nimmt für jede Verarbeitung jeweils genau ein 32 Bit Wort aus den Eingangsstates und schreibt das Ergebnis in den State für die Resultate. Die *calc* Operation wurde also mit `TIEwahAND` zusammengeführt.
`storeZ`	Schreibt die Ergebnisse aus dem State der Resultate zurück in den Speicher, sofern vier Wörter bereit zum Speichern sind.

Tabelle 5.4: Instruktionsbeschreibung für die WAH Operation - Version 2

Der folgende C-Code beschreibt, wie die angepassten Instruktionen eingesetzt werden müssen, um den Algorithmus umzusetzen:

```
while(checkEnd())
{
   ldXstream();
   ldYstream();
   TIEwahAND();
   TIEwahAND();
   TIEwahAND();
   TIEwahAND();
   storeZ();
};
```

Mit Hilfe des neuen Quellcodes werden bei jeder Iteration der Schleife zunächst acht Wörter für die Verarbeitung geladen. Die beiden Ladeinstruktionen füllen die States der Operanden hierbei mit jeweils vier Wörtern für die Verarbeitung. Im Anschluss wird die Verarbeitung jeweils auf einem 32 Bit Wort realisiert und schließlich das Ergebnis gespeichert. Die Speicherzugriffe bei diesem Vorgehen sind sowohl beim Laden als auch beim Speichern 128 Bit aligned. Bei der Verarbeitung von vier Eingangswörtern aus zwei Bitmap Indizes wird grundsätzlich einer der beiden Eingangsdatenströme vollständig aufgebraucht. Das Nachladen der neuen 128 Bit Eingangsdaten stellt in diesem Fall kein Problem dar. Jedoch wird der Datenstrom des zweiten Operanden nicht auch in jedem Fall komplett verarbeitet. Das tritt beispielsweise dann auf, wenn ein Index nur Literale enthält und der andere Index Fills mit einer Lauflänge von mindestens zwei Wörtern enthält. Die nicht verarbeiteten Wörter verbleiben in dem State der Operanden und können daher bei der nächsten Iteration noch nicht überschrieben werden. Aus diesem Grund müssen Vorkehrungen getroffen werden, damit jede Iteration vollständig durchlaufen kann, ohne das einer der beiden Eingangsstates leer läuft. Um zu Beginn jeder Iteration garantieren zu können, dass sich mindestens 4 Wörter in den Eingangsstates befinden, müssen die States der Operanden mindestens sieben Wörter aufnehmen können. Die Sieben ergibt sich aus maximal drei Wörtern, die aus der vorherigen Iteration stammen und den Vier neu geladenen. Sofern noch vier Wörter aus der vorherigen Iteration verfügbar sind, muss nicht nachgeladen werden, da für jede Verarbeitung noch mindestens ein Wort verfügbar ist und der Speicher somit nicht leerlaufen kann.
Eine weitere Besonderheit tritt bei den 128 Bit Speichervorgängen auf. Um den Algorithmus

korrekt umzusetzen bedarf es nach jedem neuen Zwischenergebnis der Überprüfung, ob es in das vorherige Ergebnis integriert werden kann. Um jedoch keine zusätzlichen Ladevorgänge zum Einlesen des letzten Ergebnisses zu erzeugen, muss das vorherige Resultat im Ergebnisstate gehalten werden. Zum Auslasten der Load/Store Unit werden auch beim Speichern 128 Bit transferiert. Diese vier 32 Bit Wörter können jedoch erst geschrieben werden, wenn das fünfte Wort im Ergebnisstate begonnen wurde. Zu diesem Zeitpunkt ist sicher, dass das letzte der vier Ergebnisse nicht mehr verändert wird weshalb die Vierergruppe in den Speicher geschrieben werden kann. Das Speichern ist nach dem o.g. Vorgehen jedoch nur am Ende jeder Iteration vorgesehen. Um also sowohl das fünfte Wort zum Speichern abwarten zu können als auch die Möglichkeit zur Aufnahme von vier neuen Wörtern aus der Iteration zu ermöglichen, bedarf es eines Ausgangsstates mit einer Breite von mindestens 8 Wörtern.

Die Iteration von Wort zu Wort wurde bei C durch das Verschieben von Pointern realisiert. Um innerhalb der Instruktionen die Konstruktion einer ähnlichen Funktionalität zu vermeiden, wird auf Basis des Prinzips von First In – First Out (FIFO) gearbeitet. Damit wird eine Warteschlange von Wörtern innerhalb der States für Ein- und Ausgangswerte erzeugt, bei denen das erste Wort als erstes verarbeitet wird. Nachdem das erste Element verarbeitet ist, wird die Liste um ein Wort verschoben, weshalb nun das zweite Wort auf der Position des Ersten ist und weiterverarbeitet wird. Das Weiterschieben innerhalb der States hat den Vorteil, dass zum einen am Ende der Liste ein Platz für neue Elemente frei wird und zum anderen die Verarbeitung immer nur auf dem ersten Element erfolgen muss, ohne eine Konstruktion ähnlich den Pointern zu bedingen.

Zusammenfassend ist festzustellen, dass die Logik innerhalb der Instruktionen komplexer geworden ist. Dieser Aspekt sorgt genauso dafür, dass sich die vom Chip benötigte Fläche vergrößert, wie die Tatsache, dass größere States zum Vorhalten der Daten benötigt werden. Neben der Fläche stellt jedoch die Verringerung der notwendigen Instruktionen ein weiteres Optimierungsziel dar. In der zweiten Entwicklungsstufe besteht eine Iteration aus acht Instruktionen die insgesamt vier Wörter verarbeiten. Es werden somit zwei Instruktionen für die Verarbeitung eines 32 Bit Worts benötigt. In der ersten Entwicklungsvariante wurden hingegen sechs Instruktionen für die gleiche Verarbeitungsleistung benötigt. Die neue Entwicklungsstufe führt somit zu einer optimierten Abarbeitungsgeschwindigkeit.

Entwicklungsstufe 3

Obwohl die zweite Entwicklungsstufe bereits einige der Optimierungsstrategien in sich vereint, wurde die Anwendung der FLIX Instruktionen bis jetzt nicht näher betrachtet. Der Prozessor verfügt über zwei Load/Store Einheiten, sodass diese auch parallel verwendet werden können. Weiterhin ist zu beachten, dass die Daten untereinander keine Abhängigkeiten aufweisen sollten, weshalb sich im Rahmen der Verarbeitung von WAH nur die Ladeoperationen zur Parallelisierung eignen. Darüber hinaus lässt sich die Funktionalität von `checkEnd` in einer anderen Instruktion integrieren, sodass nicht nur ein Takt bei jeder Iteration gespart wird, sondern auch eine Iteration weniger notwendig ist. Tabelle 5.5 fasst die angepassten Instruktionen noch einmal zusammen, wobei aus Gründen der Vollständigkeit alle verwendeten Instruktionen beschrieben werden:

Instruktion	Beschreibung
`ldXstream / ldYstream`	Laden im Rahmen einer FLIX Instruktion parallel 128 Bit Operanden für die Verarbeitung.
`TIEwahAND`	Nimmt aus den zwei FIFO States der Operanden jeweils ein 32 Bit Wort zur Verarbeitung und schreibt das Ergebnis in den FIFO Ergebnisstate.
`storeZ`	Schreibt die Ergebnisse aus dem FIFO State der Resultate, wenn sobald 128 Bit unveränderliche Ergebnisdaten zur Verfügung stehen. Weiterhin überprüft die Instruktion, ob die Iteration fortgesetzt wird, da `checkEnd` integriert wurde.

Tabelle 5.5: Instruktionsbeschreibung für die WAH Operation - Version 3

Wie bereits in Abschnitt 3.3.4 erläutert wurde, bedarf es zur effizienten Umsetzung der Parallelität bei Ladevorgängen nicht nur unabhängiger Speicherbereiche, sondern auch der Verfügbarkeit von Shadow Registern. Um diese in erwähnter Weise einzusetzen, werden bereits vor der ersten Iteration die States gefüllt, sodass bei der nächsten Ladeoperation neben dem Nachladen von Daten nur noch das Verschieben der vorherigen Operanden notwendig ist. Der folgende Quellcode stellt die Schleife der dritten Entwicklungsstufe dar:

```
while(storeZ())
{
   ldXstream(); ldYstream();
   TIEwahAND();
   TIEwahAND();
   TIEwahAND();
   TIEwahAND();
};
```

Bis zur letzten Entwicklungsstufe war die Anzahl der Instruktionen proportional zur Anzahl der notwendigen Takte für Abarbeitung. Durch die Einführung von Instruktionen die parallel verarbeitet werden können, ändert sich das. Das für die Laufzeit relevantere Maß der Takte wurde im Rahmen dieser Entwicklungsversion noch einmal um zwei Takte reduziert. Das ist zum einen auf die FLIX Instruktion für die Ladeoperationen und zum anderen auf die vollständig eingesparte Instruktion `checkEnd` zurückzuführen, deren Funktionalität nun von `storeZ` übernommen wird.

Zusammenfassung und finale Entwicklungsstufe

Die vierte und finale Entwicklungsstufe der WAH Operation konzentriert sich neben weiteren Beschleunigung auch auf das Hinzufügen von zusätzlichen Möglichkeiten der Verarbeitung. Darüber hinaus werden Optimierungen des kritischen Pfades vorgenommen, um die letztendlich mögliche Taktfrequenz für den Chip zu steigern. Die folgende Tabelle 5.6 erläutert kurz die eigesetzten Instruktionen der Abarbeitung:

Instruktion	Beschreibung
`ldXstream / ldYstream`	Laden im Rahmen einer FLIX Instruktion parallel 128 Bit Operanden für die Verarbeitung.
`TIEwah`	Verarbeitet die jeweils die Wörter der Eingangsstates mit den drei logischen Operationen, speichert das Ergebnis sobald notwendig und prüft die Fortsetzung der Iteration, da `storeZ` integriert wurde.

Tabelle 5.6: Instruktionsbeschreibung für die WAH Operation - finale Version

Der folgende Quellcode ist ein Beispiel für den funktional korrekten Einsatz der Instruktionen:

```
for(i=0; i<MAX_LENGTH; i++)
{
    ldXstream(); ldYstream();
    TIEwah();
    TIEwah();
    TIEwah();
    TIEwah();
};
```

Diese finale Version der Implementierung verringert die benötigten Takte um zwei. Zum einen entfällt die zusätzliche Instruktion für das Speichern und zum anderen wird Loop Unrolling eingesetzt, um einen zusätzlichen Vergleichstakt pro Iteration einzusparen. Die Variable `MAX_LENGTH` entspricht hierbei der Anzahl von 128 Bit Wörtern im längsten Bitmap Index, da nach Abarbeitung des längsten Eingangsdatenstromes keine weiteren Elemente mehr verarbeitet werden müssen. Lediglich durch das Pipelining können zusätzliche Instruktionen der Verarbeitung notwendig werden.

Die erste Entwicklungsversion benötigte sieben Takte zum Verarbeiten eines Wortes, während diese Version sechs Takte zur Verarbeitung von vier Wörtern benötigt. Im Laufe der Zyklen zur Verbesserung der Instruktionen wurde die Geschwindigkeit also bereits innerhalb von TIE mehr als vervierfacht. Um trotz der immer weiter gesteigerten Funktionalitäten der `TIEwah` Instruktion den kritischen Pfad möglichst kurz zu halten, wurde Pipelining eingesetzt. Die notwendigen Operationen der Verarbeitung wurden durch die Verwendung von Zuweisungsoperatoren in ausgewählte Stufen geteilt, die jeweils in einem Takt abgearbeitet werden können. Dafür wurde die Verarbeitungsinstruktion in vier Teilschritte aufgeteilt. Um den Ablauf zu verdeutlichen werden die Pipelinestufen von `TIEwah` nachfolgend als `OP_x` bezeichnet, wobei $x \in \{1,2,3,4\}$ den jeweiligen Teilschritt angibt. Die Teile der Instruktion werden in der folgenden Tabelle 5.7 erklärt:

Teilschritt	Beschreibung
`OP_1`	Lädt die Operanden als 32 Bit Wort vor und aktualisiert die Eingangsdatenströme und aktualisiert zugehörige Indizes.
`OP_2`	Verarbeitet zwei übergebene Operanden mit der jeweiligen Verknüpfung und gibt das Ergebnis weiter.
`OP_3`	Schreibt das Resultat nach Überprüfung der korrekten Position auf den Ausgangsdatenstrom und aktualisiert zugehörige Indizes.
`OP_4`	Schreibt 128 Bit aligned in den Speicher zurück, sobald genügend Ergebnisse vorhanden sind und aktualisiert zugehörige Indizes.

Tabelle 5.7: Beschreibung der WAH Pipelinestufen

Die einzelnen Teile der Verarbeitung werden nun in unterschiedlichen Takten realisiert. Das hat zur Folge, dass die Verarbeitung eines Elements auf 4 Takte aufgeteilt wird. Da die Pipeline jedoch für die notwendige Parallelität sorgt, kommt es zu keiner Verringerung der Geschwindkeit sobald die Pipeline eingeschwungen ist. Dieser stabile Zustand der Pipeline ist in Abbildung 5.1 noch einmal schematisch als Ausschnitt dargestellt. Um die Übersichtlichkeit zu erhöhen, wurde neben `TIEwah` auch `ldXstream` und `ldYstream` in `LD_X` respektive `LD_Y` umbenannt.

	OP_4						
	OP_3	OP_4					
	OP_2	OP_3	OP_4				
	LD_X						
	LD_Y						
. . .		OP_1	OP_2	OP_3	OP_4		. . .
			OP_1	OP_2	OP_3	OP_4	
				OP_1	OP_2	OP_3	
					OP_1	OP_2	
						LD_X	
						LD_Y	
Takt	n	n+1	n+2	n+3	n+4	n+5	

Abbildung 5.1: Pipelineausschnitt während der Abarbeitung von Instruktionen der WAH Operation

Die Abbildung 5.1 macht nicht nur deutlich, dass sich die Abläufe alle fünf Takte wiederholen, sondern auch, dass alle fünf Takte 2x128 Bit geladen werden. Die Teilung der WAH Operation kann jedoch nicht beliebig häufig erfolgen, da die Abhängigkeiten zwischen den einzelnen Phasen zu jedem Takt vollständig abgearbeitet werden müssen. Um die Struktur der Operation zu verdeutlichen stellt die folgende Abbildung 5.2 die Ladeoperationen sowie die ersten beiden Pipelineschritte der Verarbeitung.

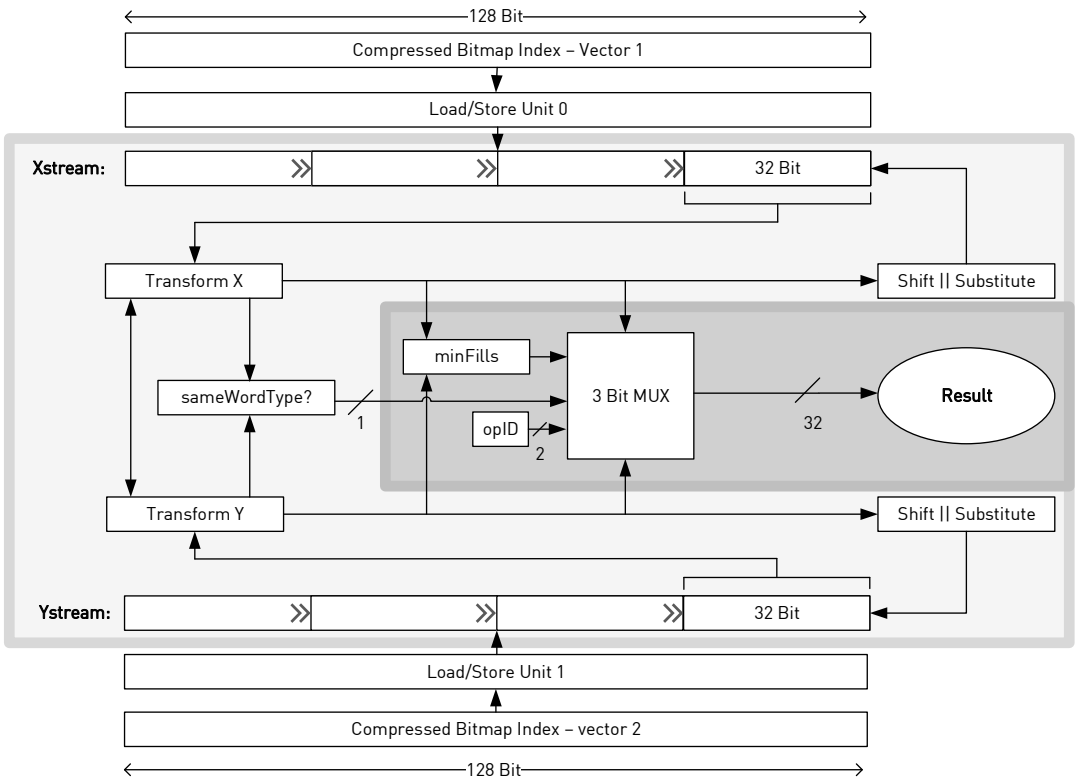

Abbildung 5.2: Architekturdiagramm der WAH Operation in TIE - Teil 1

Innerhalb der Abbildung gehören die grau-hinterlegten Teile zur WAH Verarbeitung. Die äußeren Teile an den Speichern bilden die Ladeoperationen ab. Die Pfeile kennzeichnen die Direktionalität des Datenflusses. Die Boxen stellen Funktionseinheiten dar. Auf die Darstellung der notwendigen Indizes wurde in der Abbildung allerdings ebenso verzichtet wie auf die Übergabevariablen zwischen den Pipelinephasen und die realen Breiten der States für die Operanden und Ergebnisse. Im Folgenden wird der Verlauf der gesamten WAH Operation näher erläutert. Da dieser bei jedem der Kompressionsalgorithmen ähnlich ist, wird auf die erneute Erläuterung bei den nachfolgend beschriebenen Algorithmen verzichtet.

1. Die States für die Eingangsdatenströme werden durch die beiden Ladeinstruktionen jeweils mit 128 Bit gefüllt (weiß hinterlegter Teil oben und unten).
2. Aus den Eingangsdatenströmen `Xstream` und `Ystream` werden im ersten Takt der Verarbeitungsinstruktion (`OP_1`) jeweils ein 32 Bit Operand geladen. Weiterhin wird das erste Bit der beiden Operanden verglichen um festzustellen, ob es sich um eine Operation mit zwei Fills handelt. Auf Basis dieser Information werden die Operanden für die Verknüpfung festgelegt (`Transform X/Y`) und die Eingangsdatenströme entweder mit einem veränderten Fill überschrieben, oder um 32 Bit weiter geschoben (symbolisiert durch die Doppelpfeile an den Wortgrenzen).
3. Im Rahmen von `OP_2` werden die übergebenen Daten zusammen mit der Information über die Art der Verknüpfung (`opID`) und der maximalen Lauflänge des Ergebnisses (entspricht minimaler Lauflänge der Eingangsworte (minFills)) an einen 3 Bit Multiplexer (MUX) angelegt. Der MUX berechnet das Ergebnis der Verknüpfung und gibt es in die nächste Verarbeitungsphase, die in Abbildung 5.3 dargestellt ist.
4. Der dritte Teil der Verarbeitung (`OP_3`) ist im oberen Teil von Abbildung 5.3 dargestellt. Das übergebene Resultat aus der vorherigen Phase wird innerhalb dieses Schrittes auf den Ausgangsdatenstrom `Zstream` geschrieben. Hierbei muss wieder unterschieden werden, ob die bisherigen Elemente verschoben werden um das neue Wort anzufügen oder ob das letzte Ergebnis überschrieben wird.
5. Im vierten und letzten Abschnitt der WAH Operation (`OP_4`) wird ausgewertet, ob 128 Bit des Ausgangsdatenstromes bereit zum Speichern sind und werden in diesem Fall zurück in den Speicher geschrieben (unterer Teil der Abbildung 5.3).

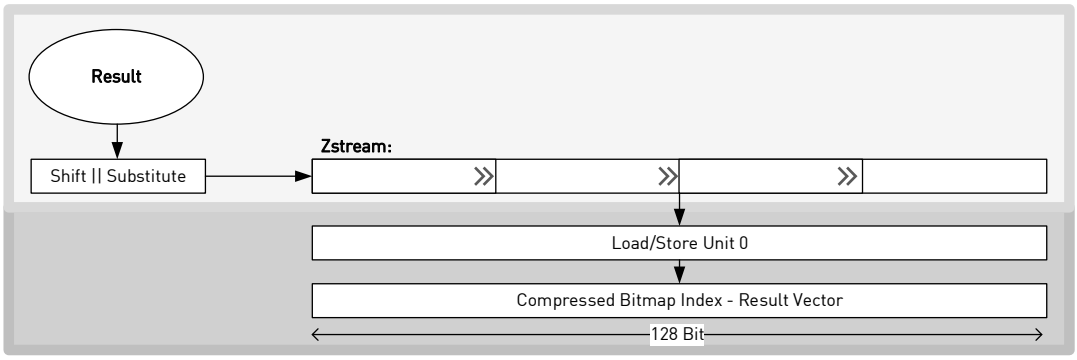

Abbildung 5.3: Architekturdiagramm der WAH Operation in TIE - Teil 2

Die Erstellung von neuen Instruktionen für die WAH Verarbeitung hat insgesamt zu einer erhöhten Parallelität geführt. Weiterhin ersetzen die TIE Instruktionen viele der ursprünglichen Funktionalitäten durch unterschiedliche Optimierungstechniken in wenigen Instruktionen. Die Erkenntnisse in Bezug auf die Lade-, Verarbeitungs- und Speicheroperationen eines RLE Algorithmus können nun auch bei den folgenden Algorithmen Anwendung finden.

5.3 PLWAH OPERATION

Nachdem der WAH Algorithmus in der finalen Ausbaustufe entwickelt war, startete die TIE Realisierung von PLWAH. Die wesentlichen Erkenntnisse des Entwurfes konnten dabei übernommen werden, sodass die Entwicklung der Instruktionen direkt auf Basis der letzten WAH Entwicklungsstufe erfolgen konnte. Zum Aufzeigen der Entwicklung wird im folgenden Abschnitt zunächst eine Verarbeitung am Beispiel aufgezeigt und darauf basierend eine Ablaufanalyse entwickelt. Nachdem dieser in C implementiert wurde, folgte die TIE Realisierung.

5.3.1 Beispielhafter Ablauf

Im Unterschied zu WAH gibt es bei PLWAH die Möglichkeit 32 Bit Wörter ineinander zu verschachteln. Voraussetzung dafür ist, dass auf ein Wort mit nur einem gesetzten Bit (Sparse) auf ein Fill folgt. Innerhalb des Fills, das durch die zwei MSBs klassifiziert wird, folgen fünf Bits die zur Positionsbeschreibung des Sparse Bits dienen. Da der Ablauf der Verarbeitung ansonsten der von WAH gleicht, soll am Beispiel in Tabelle 5.8 insbesondere dieser Spezialfall demonstriert werden:

	Element 1	Element 2	Element 3	Element 4
Index 1	82000001	C0000001	00000080	00001110
Index 2	80000002	80000001	80000001	00000247
Iteration 1	80000001			
Iteration 2	82000001			
Iteration 3	82000001	C0000001		
Iteration 4	82000001	E0000001		
Iteration 5	82000001	E0000001	00001357	
Vereinigungsmenge	82000001	E0000001	00001357	

Tabelle 5.8: Beispielablauf des PLWAH-Algorithmus mit OR Verknüpfung (HEX)

Die zwei komprimierten Bitmap Indizes *Index 1* und *Index 2* stellen im Beispiel die zu verarbeitenden Operanden zur Verfügung. Das erste Element des ersten Index ist ein Spezialwort von PLWAH. Die folgende Gleichung (5.2) stellt dies noch einmal im Detail dar:

$$\begin{aligned} & \texttt{82000001}_{16} \equiv \texttt{80000001 00000001}_{16}, \\ \text{da } & \texttt{82000001}_{16} = \underbrace{\texttt{1}}_{1}\underbrace{\texttt{0}}_{2}\underbrace{\texttt{00001 0 0000 0000 0000 0000 0000 0001}}_{3}{}_{2} \end{aligned} \tag{5.2}$$

Das Spezialwort ist also semantisch äquivalent zu einem 0-Fill der Länge eins mit einem integrierten Sparse. Das 0-Fill wird durch den mit 1 gekennzeichneten Bitbereich repräsentiert, dessen Lauflänge an den Bitpositionen von 3 festgelegt wird. Das Sparse wird durch die Bits im Bitbereich 2 gekennzeichnet. Das gesetzte Bit im Sparse befindet sich an der Position 1, da $000001_2 = 1_{10}$ gilt. Entsprechend werden in der ersten Iteration das die beiden 0-Fills der Eingangsdaten zu einem 0-Fill verknüpft. In der zweiten Iteration werden wieder die beiden ersten Elemente der Indizes verwendet, da es sich um ein Spezialwort und ein 0-Fill der Länge 2 handelte. Sparse und 0-Fill werden somit zu einem neuen Sparse verknüpft, das im Ergebnisdatenstrom wieder zu einem Spezialwort verknüpft wird. Im Anschluss werden 1-Fill und 0-Fill zu einem 1-Fill zusammengeführt in das bei der vierten Iteration wieder ein Sparse integriert wird, das sich aus der Verknüpfung der dritten Elemente des Eingangsdatenstroms ergibt. Abschließend werden in der fünften Iteration die beiden letzten Elemente zu einem Literal verknüpft. Danach endet der Algorithmus, da die Eingangsdaten vollständig verarbeitet wurden.
Das Beispiel macht zum einen deutlich, dass die Komprimierung für bestimmte Datensätze höher ist, als bei WAH. Die äquivalente Ausgangsdatenmenge hätte bei WAH zwei Wörter mehr, da keine Integration der Wörter ineinander möglich ist.

5.3.2 Ablaufanalyse

Durch die stärkere Kompressionsrate mit verschachtelten Wörtern, ergibt sich gegenüber WAH die Notwendigkeit von zusätzlichen Funktionalitäten, die dieses Vorgehen abbilden. Hierfür werden dem WAH Ablauf zwei weitere Schritte hinzugefügt, um beim Laden und Speichern den Spezialwörtern gerecht zu werden. Der folgende Ablauf stellt das Vorgehen noch einmal im Detail dar, wobei die von WAH übernommenen Punkte aus Gründen der Vollständigkeit mit aufgeführt werden.

Ablauf des PLWAH Algorithmus
Für alle komprimierten Wörter `x` aus Bitmap Index `A` und `y` aus Bitmap Index `B`:

1. Laden
   - Wenn `x` keine weiteren Fills **oder Spezialwortanteile** enthält, dann lade nächstes Element aus `A`
   - Wenn `y` keine weiteren Fills **oder Spezialwortanteile** enthält, dann lade nächstes Element aus `B`
2. Transformieren der Eingangsdaten
   - Wenn `x` ein Spezialwort ist, lade den nächsten Anteil in `x` und entferne ihn aus der Eingangsdatenmenge `A`
   - Wenn `y` ein Spezialwort ist, lade den nächsten Anteil in `y` und entferne ihn aus der Eingangsdatenmenge `B`
3. Verarbeiten
   - Wenn `x` und `y` Fills sind, dann
     (a) Ermittle das Minimum beider Lauflängenzähler `MIN`
     (b) Verknüpfe die Typen der Fills zu `ERG_TYP`
     (c) Setze `TEMP=ERG_TYP+MIN`
     (d) Verringere Fills um `MIN`
   - Wenn `x` und `y` ein Fill und ein Literal sind, dann
     (a) Verknüpfe Fill mit Literal zu `ERG`
     (b) Setze `TEMP=ERG`
     (c) Dekrementiere Fill
   - Wenn `x` und `y` ein zwei Literale sind, dann
     (a) Verknüpfe Literale zu `ERG`
     (b) Setze `TEMP=ERG`
4. Speichern
   - Wenn temporäres Ergebnis `TEMP` ein Fill ist und das Ergebnis der vorherigen Iteration ein Fill des gleichen Typs **ohne Sparse** war und beide Lauflängen bei der Addition keinen Überlauf erzeugen, dann speichere die addierten Lauflängenzähler im alten Element
   - Sonst hänge das neue Ergebnis an die Ergebnisindex `C` an
5. Transformieren der Ausgangsdaten
   - Wenn die letzten beiden gespeicherten Worte ein Fill und ein Sparse sind, dann füge sie zu einem neuen Spezialwort zusammen

Wie aus der gegebenen Ablaufanalyse deutlich wird, dient der WAH Ablauf als Basis des neuen Quellcodes, da sich beide Algorithmen, abgesehen von den Spezialwörtern, gleichen. Um die gesamte Funktionalität von PLWAH abzubilden, erweitern sich sowohl die WAH Bestandteile (fett hervorgehoben) als auch die Anzahl der notwendigen Schritte. Die Unterscheidung erfolgt insbesondere bei Lade- und Speichervorgängen, da die Spezialwörter hier für die funktionale Korrektheit umgewandelt werden müssen. Auf Basis dieser Ablaufanalyse wurde ein funktional korrekter C-Quellcode implementiert, um im Folgenden die Optimierung im Rahmen von TIE vorzunehmen.

5.3.3 Optimierung mit TIE

Genauso wie der C-Code für PLWAH auf Basis der Ergebnisse von WAH angefertigt wurde, orientiert sich auch das Design der TIE Instruktionen an der Realisierung von WAH. Tabelle 5.9 beschreibt kurz die Instruktionen, die zur Umsetzung von PLWAH benötigt werden.

Instruktion	Beschreibung
`ldXstream / ldYstream`	Laden parallel 128 Bit Operanden für die Verarbeitung im Rahmen einer FLIX Instruktion.
`TIEplwah`	Verarbeitet jeweils die Wörter der Eingangsstates mit den drei logischen Operationen, speichert das Ergebnis sobald notwendig und prüft die Fortsetzung der Iteration.

Tabelle 5.9: Instruktionsbeschreibung für die PLWAH Operation

Aus der Instruktionsbeschreibung wird deutlich, dass neben den Ladeinstruktionen für die zwei Eingangsdatenströme nur eine zusätzliche Instruktion für die Abarbeitung von PLWAH notwendig ist. Der folgende Quellcode stellt den funktional korrekten Einsatz der Instruktionen für PLWAH dar:

```
for(i=0; i<MAX_LENGTH; i++)
{
    ldXstream(); ldYstream();
    TIEplwah();
    TIEplwah();
    TIEplwah();
    TIEplwah();
};
```

Neben dem erneut eingesetzten Loop Unrolling entspricht auch die Variable `MAX_LENGTH` der Elementanzahl im längsten Eingangsdatenstrom. Weiterhin wurden die Instruktionen in Hinblick auf eine möglichst geringe Fläche konstruiert, sodass die Ladeinstruktionen von WAH übernommen wurden und somit auf einem Prozessor mit WAH und PLWAH Instruktionen nur einmal die Hardware der Ladeinstruktionen notwendig ist. Die `TIEplwah` Instruktion selbst, gleicht dem Aufbau der WAH Instruktion sowohl bei der Unterteilung in Pipelineteile als auch bei der Architektur und Verarbeitung. Die Architektur nach Abbildung 5.2 bzw. der Folgeabbildung wird lediglich in den Funktionsblöcken `Transform X` und `Transform Y` sowie
`Shift || Substitute` angepasst. Diese Komponenten werden jetzt so konfiguriert, dass über die Interpretation von Literalen und Fills hinaus, die Verarbeitung von Spezialwörtern möglich ist.

5.4 COMPAX OPERATION

COMPAX stellt sowohl den Jüngsten, als auch den komplexesten der drer implementierten Algorithmen dar. Die Kompression ist für bestimmte Ausgangsdaten stärker als bei WAH und PLWAH. Auch dieser Algorithmus folgt dem RLE Prinzip von Ein- und Ausgabestrom, aber die Verarbeitung ist aufgrund von Spezialwörtern sowohl durch eine stärkere Kompression als auch durch eine höhere Komplexität gekennzeichnet. Die Abarbeitung wird im folgenden Abschnitt am Beispiel zur Erzeugung einer Ablaufanalyse gezeigt, die schließlich in den Implementationen von C und TIE mündet.

5.4.1 Beispielhafter Ablauf

Das Vorgehen des Algorithmus wird zunächst in Tabelle 5.10 an einem Beispiel demonstriert:

	Element 1	Element 2	Element 3	Element 4	Element 5
Index 1	48010F01	00000001	800000C0		
Index 2	00000001	8000000F	8000000A	8000000C	80000080
Iteration 1	00000001				
Iteration 2	00000002				
Iteration 3	00000002	8000000A			
Iteration 4	00000002	8000000A	8000000C		
Iteration 5	00000002	8000000A	8000000C	80000040	
Sym. Differenz	00000002	8000000A	8000000C	80000004	

Tabelle 5.10: Beispielablauf der COMPAX Verarbeitung mit XOR Verknüpfung (HEX)

Neben der Verwendung von Spezialwörtern für Wortkombinationen der Form Fill-Literal-Fill (FLF) oder Literal-Fill-Literal (LFL) unterscheidet sich COMPAX von den vorherigen Algorithmen insbesondere dadurch, dass es keine 1-Fills gibt und das die Definition des MSB in den Wörtern entgegengesetzt zu der vorherigen Auslegung ist. Ein gesetztes MSB entspricht also einem Literal und die 0 als MSB kennzeichnet ein Wort mit Lauflängenzähler. Der Ablauf von COMPAX gleicht den Schilderungen der bereits diskutierten Algorithmen, sodass im Folgenden insbesondere auf die speziellen Merkmale dieser Kompressionstechnik eingegangen wird. Im Beispiel aus Tabelle 5.10 liefern die zwei Indizes wieder die zu verarbeitenden Operanden. Prinzipiell kann die unterschiedliche Größe der Indizes auch bei den anderen Kompressionsverfahren auftreten, aber bei COMPAX wird dieser Sachverhalt durch die Kombination mehrerer 32 Bit Wörter in einem Wort noch signifikanter, weshalb es hier das erste Mal demonstriert wird. Bei der Verarbeitung beider Indizes muss vor jeder Verknüpfung geprüft werden, ob die Operanden Spezialwörter enthalten. Diese lassen sich praktisch nicht mehr direkt lesen, weshalb das Spezialwort aus dem ersten Index in der folgenden Gleichung (5.3) umgewandelt wird:

$$\begin{aligned} \texttt{48010F01}_{16} &\equiv \texttt{00000001 80000000F 00000001}_{16}, \\ \text{da } \texttt{48010F01}_{16} &= \underbrace{010}_{1}\underbrace{01}_{2}\underbrace{00}_{3}\underbrace{0}_{4}\underbrace{0000\ 0001}_{5}\underbrace{0000\ 1111}_{6}\underbrace{0000\ 0001}_{7}{}_{2} \end{aligned} \tag{5.3}$$

Danach ist das erste Wort vom Typ FLF (1) mit zwei Fills der Länge eins (5,7) die durch ein Literal in der Mitte getrennt sind (6). Das Literal hat das gesetzte Byte im zweiten Viertel des Wortes (2). Die restlichen Bestandteile des Codewortes werden nicht verwendet (3,4). Die ersten drei Iteration im Beispiel aus Tabelle 5.10 verarbeiten also jeweils das erste Wort des *Index 1*. Die erste Iteration liefert bei der Verknüpfung der zwei Fills mit der Länge eins wieder in einem Fill der Länge eins. In der zweiten Iteration wird durch die XOR Verknüpfung der beiden Literale ein weiteres 0-Fill erzeugt, das dem vorherigen Ergebnis hinzugefügt wird. Bei Verknüpfung der Fills mit den Literalen in den Iterationen drei und vier ergeben sich hingegen neue Literale im Ergebnisdatenstrom. Während der fünften Iteration werden zwei Literale als Operanden erneut zu einem Literal verknüpft und der Algorithmus terminiert, da keine weiteren Eingangsdaten vorhanden sind.

5.4.2 Ablaufanalyse

Bei COMPAX ist die Ähnlichkeit zu WAH oder PLWAH signifikant. Aus diesem Grund orientieren sich sowohl der C-Code als auch das TIE Pendant an den bisher gemachten Entwicklungen. Vor der Implementierung des Algorithmus ist jedoch ein definierter Ablauf notwendig, der im Folgenden dargestellt ist:

Ablauf des COMPAX Algorithmus
Für alle komprimierten Wörter `x` aus Bitmap Index `A` und `y` aus Bitmap Index `B`:

1. Laden
   - Wenn `x` keine weiteren Fills oder Spezialwortanteile enthält, dann lade nächstes Element aus `A`
   - Wenn `y` keine weiteren Fills oder Spezialwortanteile enthält, dann lade nächstes Element aus `B`
2. Transformieren der Eingangsdaten
   - Wenn `x` ein Spezialwort ist, lade den nächsten Anteil in `x` und entferne ihn aus der Eingangsdatenmenge `A`
   - Wenn `y` ein Spezialwort ist, lade den nächsten Anteil in `y` und entferne ihn aus der Eingangsdatenmenge `B`
3. Verarbeiten
   - Wenn `x` und `y` Fills sind, dann
     (a) Ermittle das Minimum beider Lauflängenzähler `MIN`
     (b) Verknüpfe die Typen der Fills zu `ERG_TYP`
     (c) Setze `TEMP=ERG_TYP+MIN`
     (d) Verringere Fills um `MIN`
   - Wenn `x` und `y` ein Fill und ein Literal sind, dann
     (a) Verknüpfe Fill mit Literal zu `ERG`
     (b) Setze `TEMP=ERG`
     (c) Dekrementiere Fill
   - Wenn `x` und `y` ein zwei Literale sind, dann
     (a) Verknüpfe Literale zu `ERG`
     (b) Setze `TEMP=ERG`
4. Speichern
   - Wenn temporäres Ergebnis `TEMP` ein Fill ist und das Ergebnis der vorherigen Iteration ein Fill des gleichen Typs war und beide Lauflängen bei der Addition keinen Überlauf erzeugen, dann speichere die addierten Lauflängenzähler im alten Element
   - Sonst hänge das neue Ergebnis an die Ergebnisindex `C` an
5. Transformieren der Ausgangsdaten
   - Wenn die letzten drei gespeicherten Worte die Anforderungen der Worttypen FLF oder LFL erfüllen, dann füge sie zu einem neuen Spezialwort zusammen.

Die Ablaufanalyse entspricht grundsätzlich der Vorlage von WAH, nur die Abschnitte der Transformation wurden auf die Spezialworte von COMPAX angepasst. Nicht in der Ablaufanalyse ersichtlich ist allerdings, dass diese Transformationen wesentlich komplexer als bei PLWAH sind. Das ist darauf zurückzuführen, dass die Wörter nur in speziellen Fällen zusammengeführt werden können. Somit ist für die Zusammenfügung der Wörter neben einer Prüfung jedes Wortes auch die Zusammenführung mit vergleichsweise vielen Bitoperationen notwendig.
Nachdem diese Ablaufanalyse in C umgesetzt wurde, folgt die Realisierung und Optimierung in TIE, die im nächsten Abschnitt beschrieben wird.

5.4.3 Optimierung mit TIE

Durch die angesprochenen Übereinstimmungen zwischen den Algorithmen, wird auch die TIE Implementierung auf Basis der vorher gemachten Erkenntnisse umgesetzt. Tabelle 5.11 erläutert die notwendigen Instruktionen für COMPAX.

Instruktion	Beschreibung
`ldXstream / ldYstream`	Paralleles Laden von 128 Bit Operanden für die Verarbeitung im Rahmen einer FLIX Instruktion.
`TIEcompax`	Verarbeitet die jeweils die Wörter der Eingangsstates mit den drei logischen Operationen, speichert das Ergebnis sobald notwendig und prüft die Fortsetzung der Iteration.

Tabelle 5.11: Instruktionsbeschreibung für die COMPAX Operation

Da die unterschiedlichen Algorithmen jeweils nur eine andere Vor- und Nachbereitung der Daten erfordern, ist der Aufbau der COMPAX Instruktion vergleichbar zu den bisher besprochenen. Der Einsatz wird beispielhaft in dem folgenden Quellcode dargestellt:

```
for(i=0; i<MAX_LENGTH; i++)
{
    ldXstream(); ldYstream();
    TIEcompax();
    TIEcompax();
    TIEcompax();
    TIEcompax();
};
```

Der Schleifenrumpf ist äquivalent zu den bisher gemachten Ausführungen mit Loop Unrolling und der Variable `MAX_LENGTH`. Auch bei COMPAX wurden die bisher verwendeten Ladeinstruktionen wieder eingesetzt, sodass bei Realisierung aller drei Algorithmen in einem Prozessor lediglich ein Satz der Ladeinstruktionen notwendig ist.
Passend zu den bisher diskutierten Gemeinsamkeiten hat sich auch am strukturellen Aufbau aus Abbildung 5.2 und der Folgegrafik nur die Aufgabe der Funktionsblöcke verändert und erweitert. Da die Transformationen bei COMPAX allerdings im Vergleich relativ umfangreich ausfallen, hat sich der kritische Pfad verlängert, obwohl weiterhin die Aufteilung in Pipelineblöcke eingesetzt wird.

5.5 WAH KOMPRIMIERUNG

Neben der Möglichkeit zur Beschleunigung der Verarbeitung gibt es die Option zur Laufzeitverbesserung der Kompression und Dekompression. Da diese Funktionalitäten im gesamten Verarbeitungsalgorithmus nach der eigentlichen Operation die wesentlichen Hot-Spots sind, ist deren Optimierung ein weiterer konsequenter Schritt im Rahmen der TIE Optimierungsstrategie. Dabei kann unter anderem die Busbreite der Load/Store Units zum Speicher wesentlichen Einfluss auf die Geschwindigkeitssteigerung haben. Diese Optimierung erfolgte nur für WAH, um die grundsätzlichen Möglichkeiten dieses Schrittes auszuloten. Analog zu den Schritten der Verarbeitungsoptimierung wird die Kompression zunächst an einem Beispiel demonstriert. Danach folgt die Umsetzung im Rahmen einer Ablaufanalyse, die zur C-Implementierung führt Abschließend werden die TIE Optimierungen vorgestellt

5.5.1 Beispielhafter Ablauf

Die Tabelle 5.12 enthält ein Beispiel zur Kompression mit WAH. Hierbei liegen die beliebig gewählten Ausgangsdaten zunächst als ein zusammenhängender Bitstrom vor, der dem Algorithmus zugeführt und wie folgt verarbeitet wird:

Schritt	Beschreibung	Beispielhafte Umsetzung
	124 Bit unbearbeitete Ausgangsdaten in 32 Bit Blöcken vor der Kompression	0000000000 0000000000 0000000010 00 0000000000 0000000000 0000000000 00 0000000000 0000100000 0000000000 00 0000000000 0010000000 00000000
1	Aufteilen der Ausgangsdaten in 31 Bit Blöcke	0000000000 0000000000 0000000010 0 0000000000 0000000000 0000000000 0 0000000000 0000001000 0000000000 0 0000000000 0000010000 0000000000 0
2	Identifizieren von Fills (im Beispiel hervorgehoben)	0000000000 0000000000 0000000010 0 **0000000000 0000000000 0000000000 0** **0000000000 0000000000 0000000000 0** 0000000000 0000010000 0000000000 0
3	Kodieren der Gruppen zu Fills und Literalen mit 32 Bit Wortlänge	0 0000000000 0000000000 00000000100 **10 0000000000 0000000000 0000000010** 0 0000000000 0000010000 00000000000

Tabelle 5.12: Konzeptioneller Ablauf der WAH-Komprimierung

Der geschilderte Ablauf zeigt anhand von 124 Bit eines Eingangsdatenstroms die Kompression mit WAH. Der Eingangsdatenstrom liegt dabei in 32 Bit Wortgröße vor und muss für die Kompression zunächst in 31 Bit Chunks aufgeteilt werden. Im Anschluss werden die 31 Bit Chunks einzeln gelesen und auf den Ausgangsdatenstrom geschrieben. Das erfolgt konform zum RLE Prinzip, sodass die schrittweise gelesenen Daten entweder an die Ergebnisdaten angehangen, oder zusammengeführt werden. Da der Beispieldatenstrom 2x31 Bit Nullen enthält, werden diese entsprechend zu einem 0-Fill der Länge zwei (Lauflängenzähler: $10_2 = 2_{10}$) konvertiert. Im Ergebnis erfolgte eine Komprimierung des belegten Speicherplatzes um ein Wort. Dabei wird davon ausgegangen, dass zur ursprünglichen Speicherung des Eingangsdatenstroms 128 Bit Speicherplatz für die 124 Bit allokiert wurde.
Die bitgenaue Aufschlüsselung ist in den folgenden Gleichungen sichtbar. So ist der unkomprimierte Datenstrom in Gleichung (5.4) dargestellt und nach Durchlaufen des WAH Algorithmus die komprimierte Form in Gleichung (5.5).

$$B_{unkompr.} = \underbrace{\underbrace{0000000000\ 0000000000\ 00000000}_{28 \times 0}\,1\,\underbrace{0\ldots0}_{79 \times 0}\,1\,\underbrace{0\ldots0}_{15 \times 0}}_{\textbf{124 Bit}} \tag{5.4}$$

$$B_{WAH-kompr.} = \underbrace{\underbrace{0\ldots0}_{29 \times 0}\,1001\,\underbrace{0\ldots0}_{29 \times 0}\,1\,\underbrace{0\ldots0}_{17 \times 0}\,1\,\underbrace{0\ldots0}_{15 \times 0}}_{\textbf{96 Bit}} \tag{5.5}$$

Im Ergebnis konnte also trotz der Notwendigkeit des Einfügens von Indikatorbits eine Kompression um 28 Bit bzw. 21,9% erreicht werden.

5.5.2 Ablaufanalyse

Das Beispiel aus dem vorhergehenden Abschnitt hat gezeigt, dass sich das Prinzip der Iteration über die Elemente auch für den Kompressionsalgorithmus eignet. Um die spätere Implementierung vorzunehmen, wird der Algorithmus zunächst als Ablaufanalyse formuliert.

Ablauf der WAH-Kompression
Setze `POS=0`
Für alle unkomprimierten 32 Bit Wörter aus dem Eingangsdatenstrom:

1. Laden
   - Extrahiere 31 Bit des Eingangsdatenstromes beginnend bei `POS` in das Wort `w` und inkrementiere `POS` um 31
2. Verarbeiten
   - Wenn `w` nur aus Nullen besteht, setze `w=80000001`
   - Oder wenn `w` nur aus Einsen besteht, setze `w=C0000001`
   - sonst ändere nichts an `w`
3. Speichern
   - Wenn `w` ein Fill ist und das Ergebnis der vorherigen Iteration ein Fill des gleichen Typs war und beide Lauflängen bei der Addition keinen Überlauf erzeugen, dann speichere die addierten Lauflängenzähler im alten Element
   - Sonst hänge `w` an den Ausgangsdatenstrom an

Nachdem dieser Ablauf in C Implementiert wurde stellte sich heraus, dass es zwei besonders laufzeitintensive Abschnitte gibt. Dazu gehört zum einen das Laden, da hier nicht auf das übliche 32 Bit Alignment zurückgegriffen wird. Das hat zur Folge, dass in vielen Fällen eine Fensterung über zwei Elementen vorgenommen werden muss, was das Laden verlangsamt. Den zweite Abschnitt mit erhöhter Laufzeit stellt, aufgrund der notwendigen Vergleiche zum komprimierten Schreiben, das Speichern dar.

5.5.3 Optimierung mit TIE

Die Ablaufanalyse hat verdeutlicht, dass die Kompression insbesondere für die SIMD Optimierung geeignet ist, da die Verarbeitung selbst relativ einfach ist und somit mehrere Elemente auf einmal verarbeitet werden können, ohne das der kritische Pfad zu lang wird. Aus diesem Grund eignet sich natürlich auch Fusion als Optimierungstechnik. Auf Basis der Erkenntnisse aus den Optimierungen der Verarbeitungsalgorithmen entstanden die in Tabelle 5.13 angeführten Instruktionen:

Instruktion	Beschreibung
`ldBaseData`	Lädt 128 Bit unkomprimierte Basisdaten für die Kompression.
`TIEcompWAH`	Komprimiert 31-124 Bit der Eingangsdaten, verknüpft sie mit dem vorherigen Ergebnis und schreibt sie zurück in den Speicher.

Tabelle 5.13: Instruktionsbeschreibung für die WAH-Kompression

Der folgende Quellcode ist ein Beispiel für den funktional korrekten Einsatz der Instruktionen:

```
for(i=0; i<MIN_LENGTH; i++)
{
    ldBaseData();
    TIEcompWAH();
};
```

Die mit TIE optimierte Version der Kompression nutzt wieder Loop Unrolling, um die Anzahl der Takte zu minimieren. Mit jeder Iteration der Schleife werden im besten Fall 124 Bit verarbeitet. Die 124 Bit werden dabei über Prefetching berechnet. Wenn also mindestens 31 Bit einem 1-Fill oder einem 0-Fill entsprechen, werden die folgenden 31 Bit überprüft, ob sie die gleiche Charakteristik wie das erste Element aufweisen. Dieser schrittweise Vergleich kann bis zu 4 Elemente auswerten, sodass im Idealfall – also bei 4x31x0 Bits oder 4x31x1 Bits – 124 Bits gleichzeitig in das Ergebniswort geschrieben werden können. Diese Optimierung mit Hilfe von SIMD ist vor allem für Basisdaten interessant, die viele gleiche Bits enthalten und somit ein hohes Optimierungspotential aufweisen. Wenn B die Anzahl der Bits im Eingangsdatenstrom darstellt, muss die Schleife folglich mindestens $\frac{B}{128}$ Iterationen machen um alle Daten zu laden (im besten Fall). Dieser Anzahl entspricht `MIN_LENGTH`. Die anderen Fälle werden durch eine nachgelagerte Schleife bearbeitet, sodass in jedem Fall alle Eingangsbits verarbeitet werden. Das Laden der Operanden wird im aufgezeigten Quelltext von dem `ldBaseData` Intrinsic übernommen und Verarbeitung sowie die Schreibvorgänge durch `TIEcompWAH` realisiert.

Um auch bei der Kompression den kritischen Pfad zugunsten der Taktfrequenz möglichst kurz zu halten, wurde wieder eine Unterteilung in mehrere Teile vorgenommen, die in unterschiedlichen Takten der Pipeline verarbeitet werden können. Aufgrund der Ausgangsbedingungen konnte eine Unterteilung in drei Schritte vorgenommen werden, die im Folgenden als `CO_x` bezeichnet werden, wobei $x \in \{1,2,3\}$ Auskunft über den jeweiligen Teilschritt gibt. Um die Funktionalität der einzelnen Teile zu verdeutlichen, werden diese separat in der folgenden Tabelle 5.14 erklärt:

Teilschritt	Beschreibung
`CO_1`	Berechnet das Kompressionsergebnis von 31-124 Bits auf Basis des Prefetchings, aktualisiert den Eingangsdatenstrom und aktualisiert zugehörige Indizes.
`CO_2`	Kombiniert ggf. das Ergebnis mit dem vorherigen und schreibt das Ergebnis in einen Ergebnisstate.
`CO_3`	Schreibt 4 Ergebniswörter zurück und aktualisiert zugehörige Indizes.

Tabelle 5.14: Beschreibung der Pipelinestufen der WAH-Kompression

Nachdem die Instruktion auf mehrere Takte aufgeteilt wurde, folgt die Betrachtung der Pipeline wie in Abbildung 5.4 dargestellt. Dabei wurde wieder der eingeschwungene Zustand der Pipeline angenommen und die Ladeinstruktion aus Gründen der Übersichtlichkeit in `LD_W` umbenannt.

	CO_2 LD_W	CO_3		
. . .		CO_1	CO_2 LD_W	. . .
Takt	n	n+1	n+2	

Abbildung 5.4: Pipelineausschnitt bei der Verarbeitung von Instruktionen der WAH-Kompression

Abbildung 5.4 macht deutlich, dass bei der Durchführung der Kompression ein hoher Speicherdurchsatz erreicht werden kann. In jedem Takt können entweder Daten geladen oder gespeichert werden, sodass eine effiziente Nutzung der Load/Store Units möglich ist, sofern die die Daten das hierfür benötigte Prefetching ermöglichen.

5.6 WAH DEKOMPRIMIERUNG

Auf Basis der erfolgreichen Umsetzung der WAH-Kompression können die gewonnen Erkenntnisse auch auf die Dekompression angewandt werden. Zunächst wird an einem Beispiel der Herleitung der Ablaufanalyse verdeutlicht, um im Anschluss die Implementierung in C und die Optimierung mit TIE vorzunehmen.

5.6.1 Beispielhafter Ablauf

Die Tabelle 5.15 stellt die Dekomprimierung an einem Beispiel vor:

Schritt	Beschreibung	Beispielhafte Umsetzung

	96 Bit Ausgangsgaten mit allen Codewörtern vor der Dekomprimierung Kodieren der Gruppen zu Fills und Literalen mit 32 Bit Wortlänge	`0 0000000000 0000000000 00000000100`
		`10 0000000000 0000000000 0000000010`
		`11 000000000 0000010000 00000000001`

1	Identifizieren der Codeworttypen (Fills sind fett hervorgehoben)	`0 0000000000 0000000000 00000000100` **`10 0000000000 0000000000 0000000010`** **`11 0000000000 0000000000 0000000001`**

2	Extrahieren der Fills und Zusammenfügen der 31 Bit Gruppen	`0000000000 0000000000 0000000010 0`
		`0000000000 0000000000 0000000000 0`
		`0000000000 0000000000 0000000000 0`
		`1111111111 1111111111 1111111111 1`

3	Schreiben der 124 Bit Ausgangsdaten in 32 Bit Blöcken	`0000000000 0000000000 0000000010 00`
		`0000000000 0000000000 0000000000 00`
		`0000000000 0000000000 0000000001 11`
		`1111111111 1111111111 11111111`

Tabelle 5.15: Konzeptioneller Ablauf der WAH-Dekomprimierung

Der Ablauf nach Tabelle 5.15 macht deutlich, dass auch die Dekompression dem RLE Prinzip folgt. Aus dem Eingangsdatenstrom mit drei 32 Bit Wörtern wird nach dem Einlesen jedes Codewort entsprechend seines Typs klassifiziert und die entsprechende Menge an 31 Bit Gruppen erzeugt. Um diese abschließend wieder 32 Bit aligned im Speicher abzulegen, wird jedes Wort entsprechend um das fehlende Bit vom Nachfolger ergänzt. Die Datenmenge aus dem Beispiel enthält ein Literal, ein 0-Fill der Länge zwei und ein 1-Fill mit der Länge eins. Die 96 Bit der drei Wörter repräsentieren entsprechen 124 Bit unkomprimierter Datenmenge.

5.6.2 Ablaufanalyse

Wie aus dem Beispiel des vorherigen Abschnitts ersichtlich, eignet sich das bisherige Schema mit den Unterpunkten Laden, Verarbeiten und Speichern auch für den Einsatz bei der Dekompression. Die folgende Ablaufanalyse legt hierbei die Grundlage der späteren Implementierung in C und TIE.

Ablauf der WAH-Dekompression
Setze `POS=0`
Für alle komprimierten 32 Bit Wörter `w` aus dem Eingangsdatenstrom `E`:

1. Laden
   - Wenn `w` kein Fill mit vorhandener Lauflänge ist, dann überschreibe es mit einen neuen Operanden aus `E`
2. Verarbeiten
   - Wenn `w` ein 1-Fill ist, dann setze das Ergebnis `ERG=7FFFFFFF`$_{16}$
   - Wenn `w` ein 0-Fill ist, dann setze das Ergebnis `ERG=0000000`$_{16}$
   - sonst setze `ERG=w`
3. Speichern
   - Schreibe `ERG` auf den Ausgangsdatenstrom, beginnend bei `POS` und setze `POS=POS+31`

Diese Ablaufanalyse wurde im Anschluss in C implementiert. Die Analysen während der Profiling-Phase zeigten hierbei wieder, dass die Speicherzugriffe zu den kritischsten Komponenten für die Laufzeit gehören. Insbesondere das Speichern ist durch die 31 Bit großen Ausgangsdaten ineffizient, da auch hier wieder eine Fensterung vorgenommen werden muss, um die Daten auf das 32 Bit Raster des Speichers anzupassen.

5.6.3 Optimierung mit TIE

Die Ablaufdarstellung hat gezeigt, dass die Dekompression der Wörter eine Folge relativ einfacher Operationen ist, die auf alle Elemente nacheinander angewandt werden. Aus diesem Grund eignet sich die Dekompression für die Optimierung mit SIMD Instruktionen, da über diesen Weg mehrere Elemente gleichzeitig verarbeitet werden können. Durch die geringere Komplexität ist zudem sichergestellt, dass der kritische Pfad nicht zu lang wird. Aus diesem Grund eignet sich auch Fusion als Optimierungstechnik für die Dekomprimierung der Daten. Die Kenntnisse der vorherigen Instruktionenserstellungen wurden als Basis für die Erweiterungen genutzt, die in Tabelle 5.16 genauer erklärt werden.

Instruktion	Beschreibung
`ldComprStream`	Lädt 128 Bit komprimierte Daten für die Dekompression.
`TIEdecompWAH`	Dekomprimiert die Eingangsdaten und schreibt zwischen 31 und 124 Bit um sie anschließend in den Speicher zu schreiben.

Tabelle 5.16: Instruktionsbeschreibung für die WAH-Kompression

Wie die Instruktionen funktional korrekt eingesetzt werden, demonstriert das folgende Beispiel:

```
for(i=0; i<MIN_LENGTH; i++)
{
ldComprStream();
TIEdecompWAH();
TIEdecompWAH();
TIEdecompWAH();
TIEdecompWAH();
};
```

Auf Basis der Entwicklungen für die Kompression, werden auch bei der Realisierung der Dekomprimierung Loop Unrolling und Shadow Register eingesetzt. Da die komprimierten Wörter in vielen Fällen nicht nur ein Ergebniswort ergeben, wird die Instruktion zudem so konfiguriert, dass bei Fills bis zu 4 enthaltene Wörter auf einmal geschrieben werden können. Die resultierende Ergebnismenge kann also von Wort zu Wort zwischen 31 Bit und 124 Bit schwanken, je nachdem ob es sich um ein Literal oder ein Fill handelt. Da aber bei jeder Iteration im Maximalfall vier Wörter verarbeitet werden können, reicht das Nachladen der neuen Operanden durch `ldComprStream` alle vier Takte. Im Ergebnis entspricht die minimale Anzahl der Iterationen der Wortanzahl im Eingangsdatenstrom, die durch `MIN_LENGTH` repräsentiert wird. Für den Fall das Fills im Eingangsdatenstrom enthalten sind, die eine größere Lauflänge als vier aufweisen, müssen jedoch mehr Iterationen realisiert werden, weshalb dieser Hauptschleife eine weitere nachgelagert ist.

Auch bei der Dekompression wurde der kritische Pfad in Hinblick auf die Taktfrequenz mit dem Einfügen von Unterabschnitten für die Pipeline beschleunigt. Die Logik lässt sich hierbei in zwei Schritte aufteilen, die im Folgenden als `DE_x` bezeichnet werden, wobei $x \in \{1,2\}$ Auskunft über den jeweiligen Teilschritt gibt, wie in Tabelle 5.17 erläutert.

Teilschritt	Beschreibung
`DE_1`	Berechnet das Ergebnis 31-124 Bit breite Ergebnis der Dekompression des aktuellen Operanden, aktualisiert den Eingangsdatenstrom, schreibt das Ergebnis in einen Zwischenstate und aktualisiert zugehörige Indizes.
`DE_2`	Schreibt 128 Bit des Ausgangsdatenstroms sobald er diese Größe erreicht hat.

Tabelle 5.17: Beschreibung der Pipelinestufen der WAH-Dekompression

Das Verhalten der Pipeline hat sich durch die andere Aufteilung der Instruktionsteile für die Dekompression verändert. Ein Ausschnitt der eingeschwungenen Pipeline ist in Abbildung 5.5 dargestellt. Die Ladeinstruktion wurde dabei aus Übersichtlichkeitsgründen in `LD_C` umbenannt.

	DE_2						
	LD_C						
		DE_1	DE_2				
. . .			DE_1	DE_2			. . .
				DE_1	DE_2		
					DE_1	DE_2	
						LD_C	
Takt	n	n+1	n+2	n+3	n+4	n+5	

Abbildung 5.5: Pipelineausschnitt bei der Verarbeitung von Instruktionen der WAH-Dekompression

Der in Abbildung 5.4 dargestellte Ausschnitt der Pipeline zeigt exemplarisch die Umsetzung der Dekompression. Da bei der Ladeoperation 128 Bit von der Load/Store Unit gelesen werden und potentiell nach jeder Operation 124 Bit Daten auf den Ausgangsdatenstrom geschrieben werden können, erreicht die Dekompression genauso wie die Kompression einen hohen Datenumsatz über den 128 Bit breiten Bus der Load/Store Unit.

5.7 LIMITIERUNGEN UND EINSCHRÄNKUNGEN

Bei der Optimierung der Quellcodes mit Hilfe der TIE Instruktionen wurden einige Annahmen getroffen, die Voraussetzung für den erfolgreichen Ablauf der geschilderten Algorithmen ist. Dabei wird zwischen Voraussetzungen unterschieden, die algorithmenspezifisch sind und Annahmen die für alle betrachteten Algorithmen gelten müssen.

Generelle Voraussetzungen für den erfolgreichen Ablauf sind:

- Anzahl der zu verarbeiteten Bits ist bekannt
- Anzahl der zu verarbeiteten Bits ist größer 0
- Kardinalität der erzeugten Bitmap Indizes ist größer 0
- Gesamtanzahl der zu verarbeiteten (unkomprimierten) Bits ist kleiner 100.001, wobei die zwei 32 kByte Speicher des gegebenen Prozessor diesen Wert vorgeben
- Worte werden grundsätzlich ohne Vorzeichen betrachtet (`unsigned`)
- Wortgröße beträgt immer 32 Bit
- Eingangsdatenstrom liegt im Speicher hintereinander
- Ausgangsdatenstrom kann hintereinander in den Speicher gelegt werden
- Startpointer von Ein- und Ausgangsdatenstrom sind 128 Bit aligned
- Bei Verwendung einer Datenbank, unterstützt diese grundsätzlich die Erstellung der Daten mit den geforderten Ausgangsbedingungen

Darüber hinaus gelten für den erfolgreichen Ablauf der Verknüpfungsoperationen folgende Bedingungen:

- Datenströme der unkomprimierte Basisdaten haben die gleiche Bitanzahl
- Eingangsdatenströme sind auf die zwei Speicherbereiche des Prozessor aufgeteilt

6 EXPERIMENTELLE RESULTATE

Dieses Kapitel dient dem Aufzeigen der Leistungsparameter von den entwickelten Prozessorinstruktionen. Hierzu werden mit Hilfe der Softwaretools von Xtensa, die notwendigen Takte für jeden Verarbeitungsschritt bestimmt und im Anschluss miteinander verglichen. Innerhalb der folgenden Abschnitte zu den einzelnen Algorithmen, werden die wichtigsten Erkenntnisse zusammengefasst und die Daten der Messungen in Graphen dargestellt.. Die Messungsgrafiken der Verarbeitung existieren für jede der möglichen Verknüpfungsarten. Die Grundaussagen bleiben jedoch zwischen den unterschiedlichen Verknüpfungsoperatoren gleich, sodass in den Kapiteln jeweils nur auf eine Messungsgrafik eingegangen wird. Aus Gründen der Vollständigkeit, befinden sich die anderen Grafiken im Anhang dieser Arbeit.

Ausgangsdaten:
Die Menge der unkomprimierten Ausgangsdaten beträgt 50.000 Bit. Für alle Algorithmen gilt, dass gleichförmige Eingangsdaten mit vielen Nullen oder vielen Einsen zu anderen Messergebnissen führt, als variierende Bitmuster. Das ist darauf zurückzuführen, dass alle vorgestellten Kompressionsalgorithmen nach dem RLE Prinzip arbeiten. Aus diesem Grund können gleichförmige Daten besser komprimiert werden als Wechselnde. Die Untersuchungen sind jeweils mit 16 Datenpunkten bei zweidimensionalen Darstellungen und 256 Erhebungen für dreidimensionale Darstellungen angefertigt. Bei den Messungen war anzustreben, die Kompressionsstärke der Algorithmen auf den Ausgangsdaten langsam zu steigern. Entsprechend ist neben den Extrembereichen der Bitdichte bei 1 oder 0 vor allem das Gebiet interessant, bei dem zunehmend die Komprimierung einsetzt. Für das Raster der Untersuchungspunkte ist aus diesem Grund eine höhere Auflösung im Bereich großer Bitdichten anzustreben. Als Ausgangsdaten wurden gleichverteilte Zufallszahlen gewählt, die die Bitmap Indizes simulieren. Dabei kann die Kardinalität der erzeugten Bitmap Indizes frei festgelegt werden. Eine hohe Kardinalität führt zu einer geringen Bitdichte. Die Messungen sind mit den Kardinalitäten von 2^x erzeugt, wobei $x \in \{0,1,...,16\}$.
Diese Nutzung der Kardinalitäten erzeugt die gewünschte Verteilung der unterschiedlichen Bitdichten mit einer großen Auflösung in den Bereichen mit hoher Bitdichte. Alle Untersuchungen erfolgten mit zwei Bitmap Indizes, die verarbeitet wurden. Bei zweidimensionalen Darstellungen wurde die Verteilung eines Index fest konfiguriert, während die Verteilung des anderen Index variiert. Um dabei eine die Notwendigkeit der Verarbeitung jedes Codeworttypen sicherzustellen, besitzt die feste Bitmap eine Kardinalität von 128, während die Bitdichte des zweiten Bitmap Index zwischen 0 und 1 variiert. Die Kardinalität von 128 führt dazu, dass in vier Wörtern mindestens ein Bit auf 1 gesetzt ist. Bei der Verarbeitung mit der anderen Bitmap, führt das zur Verwendung der unterschiedlichen Codewörter.

Einordnung der Ergebnisse:
Die relevante Messgröße der Vergleichsmessungen ist der Speedup. Dieser berechnet sich aus

dem Quotienten der Prozessorcycles ohne TIE Instruktionen und der für die Verarbeitung genutzten Takte mit TIE Instruktionen. Dabei wird eine möglichst geringe Anzahl benötigter Takte angestrebt, da sich die Laufzeit dadurch reduziert und somit eine Optimierung eintritt. Das Ergebnis der Division stellt dann den Geschwindkeitsfaktor zwischen den beiden Vergleichsgrößen dar. Hierzu werden jeweils die Simulation auf dem `oli_imdata_6` Prozessor mit und ohne TIE Instruktionen verglichen, um eine Aussage über die erzielte Optimierung machen zu können. Die dreidimensionalen Messungsgrafiken sollten dabei weitgehend symmetrisch sein, wobei leichte Abweichungen durch unterschiedliche Verzweigungsgrade innerhalb des nicht-optimierten Codes auftreten können.

6.1 PROZESSORMODELLE

Die Simulationen wurden auf der Basis unterschiedlicher Prozessormodelle durchgeführt. Der Prozessor, für den die Instruktionen optimiert wurden, wird als *oli_imdata_6* bezeichnet. Diese, auf der Basis des Xtensa LX4 Prozessors entstandene Recheneinheit, hat Änderungen erfahren. Der resultierende Prozessor ist in Abbildung 6.1 dargestellt.

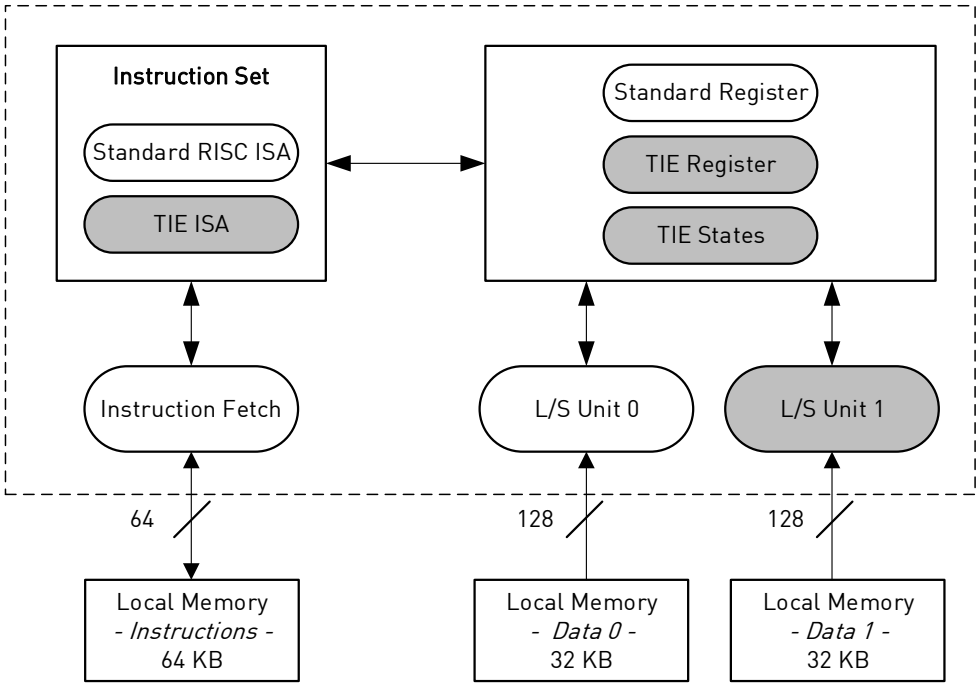

Abbildung 6.1: Architekturdiagramm des oli_imdata_6 Prozessors

Zu den wesentlichen Merkmalen des Prozessors gehört die Instruction Fetch Unit mit einer Breite von 64 Bit. Die breiten Instruktionen sind notwendig, um von dem Einsatz der FLIX Instruktionen profitieren zu können. Weiterhin verfügt der Prozessor über zwei Load/Store Units, mit einer Speicheranbindung von jeweils 128 Bit Breite. Diese Architektur ermöglicht den parallelisierten Einsatz von Lade- und Speicheroperationen sowie die Verarbeitung von mehrerer Wörtern mit einer Instruktion. Die Änderungen des Instruktionssatzes mit zugehörigen Speicherbauteilen und der zweiten Load/Store Unit sind in Abbildung 6.1 grau hervorgehoben.

Um den Vergleich mit einem anderen Prozessor zu ermöglichen, wurde ein fertig konfigurierter

Standardprozessor von Tensilica verwendet – der *Diamond Standard 108mini* [Ten10]. Im Gegensatz zu dem angepassten Prozessor, verfügt er nur über eine Load/Store Unit und die Wortbreite zur Übertragung ist hier, genauso wie bei der Instruction Fetch Unit, auf 32 Bit beschränkt. Der schematische Aufbau ist in Abbildung 6.2 dargestellt.

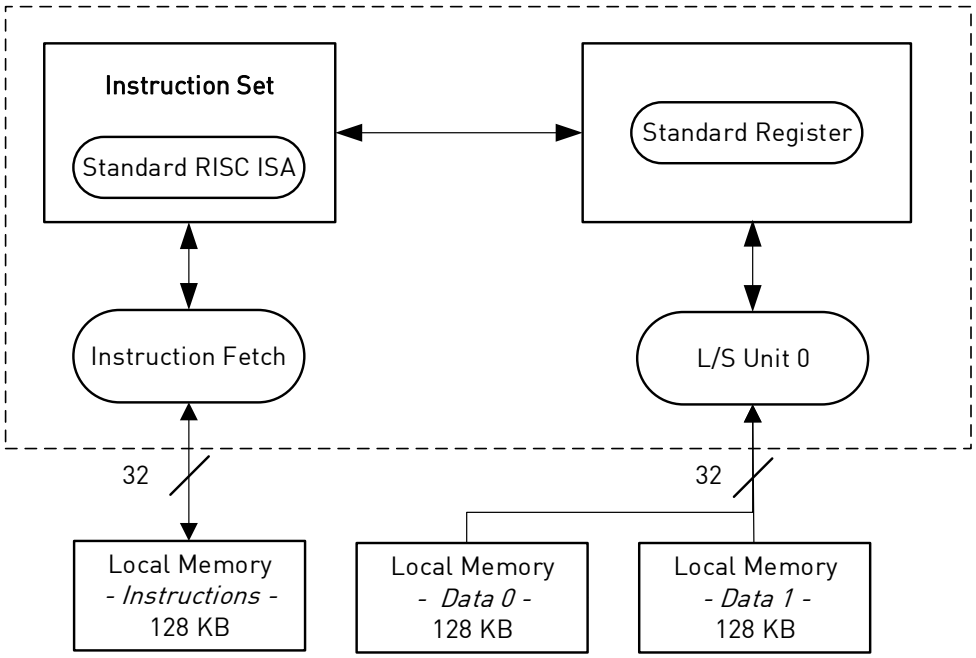

Abbildung 6.2: Architekturdiagramm des Diamond Standard 108mini

Der 108mini kann also weder die FLIX Instruktionen, noch den Speicherzugriff mit 128 Bit Breite nutzen, weshalb er einen guten Vergleich zu einem Standardprozessor ermöglicht.
Die folgenden Abschnitte greifen für jeden Algorithmus den Vergleich der zwei Prozessoren auf. Weiterhin wird die Simulation auf dem `oli_imdata_6` Prozessor sowohl mit, als auch ohne die hinzugefügten TIE Instruktionen durchgeführt.

6.2 SYNTHESEDETAILS

Der Xtensa LX Prozessor wurde im Rahmen der vorliegenden Arbeit in mehreren Schritten durch Instruktionen erweitert, die die Funktionalität des Prozessors vergrößert haben. Die Instruktionen wurden in den `oli_imdata_6` eingebracht und schrittweise synthetisiert. Dafür wurde der *Synopsys Design Compiler* verwendet, der die Ergebnisse für einen *65 nm low-power TSMC* Prozess mit Standardbedingungen ($25°C$, $1{,}25V$) erstellt. Um die Veränderungen von Chipfläche und maximaler Taktfrequenz zwischen den Entwicklungsstufen unterscheiden zu können, wurde nach der Fertigstellung jedes Algorithmus eine Synthese vorgenommen. Die Tabelle 6.1 enthält die relevanten Werte der Synthese. Dabei enthält der Basissprozessor keinerlei hinzugefügte Instruktionen. In der WAH-Version sind hingegen die Kompression, Verarbeitung und Dekompression von Bitmap Indizes auf Basis des WAH Algorithmus enthalten. Die folgenden Ergebnisse für PLWAH und COMPAX enthalten dabei immer auch die Funktionalität der vorherigen Synthesen. Weiterhin ist der *Diamond Standard 108 Mini* Prozessor als Vergleich enthalten.

Funktionalität	Logikfläche [mm^2]	Maximale Frequenz [MHz]
108 Mini	0,220	442
Basisprozessor	0,164	555
WAH	0,419	510
WAH & PLWAH	0,397	455
WAH & PLWAH & COMPAX	0,390	384

Tabelle 6.1: Syntheseergebnisse

Bei der Erstellung der Syntheseergebnisse wird sich bei der Größe lediglich auf die Fläche der Logik bezogen, sodass die notwendigen Speicher in den Ergebnissen keine Berücksichtigung finden. Der Prozessor mit der geringsten Funktionalität ist der Basisprozessor. Daraus ergibt sich sowohl die geringe Chipfläche als auch die hohe Taktfrequenz, da weder Platz für zusätzliche Logik verbraucht wird, noch der kritische Pfad zu lang für die Maximalfrequenz von 555 MHz ist. Der unveränderliche 108 Mini hat hingegen Funktionalitäten für die digitale Signalverarbeitung, weshalb sowohl die Frequenz sinkt als auch die genutzte Fläche steigt. Die größte Chipfläche beansprucht der Prozessor, der durch die TIE Instruktionen für den WAH Algorithmus erweitert wurde. Ein Grund hierfür ist, dass in den Varianten mit PLWAH und COMPAX jeweils weitere Größenoptimierungen an den WAH Instruktionen vorgenommen wurden. Außerdem ergibt sich durch die hohe Taktfrequenz eine Notwendigkeit für zusätzliche Hardwarebausteine, damit jeder Instruktionsteil mit der maximalen Taktfrequenz verarbeitet werden kann. Durch die Erweiterung mit PLWAH und COMPAX steigt die Länge des kritischen Pfades, weshalb die Taktfrequenz gesenkt werden muss. Aus diesem Grund kann auf zusätzliche Hardwarekomponenten für hohe Taktfrequenzen verzichtet werden, was zu einer Reduzierung des Flächenbedarfes führt. Zudem wurde bei der Synthetisierung aller zusätzlichen Instruktionen auf die parallele Verwendung der Ladeinstruktionen verzichtet, was die Chipfläche zusätzlich verkleinert, da einige Hardwarebestandteile geteilt werden können und nicht parallel vorhanden sein müssen.

Neben dem letztendlichen Flächenbedarf nach der Synthetisierung, lässt sich bereits vorab eine Schätzung der Fläche durchführen. Diese wird auf Basis der jeweils verwendeten Logikgatter mit Hilfe der Xtensa Softwaretools ermittelt. Der Basisprozessor verwendet etwa 71.000 Logikgatter und die Erweiterung mit den entwickelten TIE Instruktionen führt zu etwa 142.000 zusätzlichen Gattern. Die Anzahl der verwendeten Logikgattern hat sich mit allen Instruktionen etwa verdreifacht. Tabelle 6.2 enthält die Aufteilung der Gatter auf die einzelnen Funktionalitäten.

Funktionalität	Anteil am Gesamtprozessor [%]
Basisprozessor	20,3
WAH	37,1
PLWAH	21,2
COMPAX	21,4
Summe	100,0

Tabelle 6.2: Abschätzung der Größenanteile

Aus Tabelle 6.2 ergibt sich, dass die Verarbeitungsoperationen etwa 20 Prozent der zusätzlichen Chipfläche ausmachen. Zwar stellt die WAH Verarbeitung den einfachsten Algorithmus mit der geringsten Gatteranzahl dar, aber durch die zusätzliche Integration der Instruktionen für Kompression und Dekompression steigt der insgesamt verwendete Anteil.

6.3 WAH OPERATION

6.3.1 Vergleich der WAH Entwicklungsstufen

In Abschnitt 5.2 wurden die unterschiedlichen Stufen der Instruktionsentwicklung am Beispiel von WAH aufgezeigt. Neben der bereits erfolgten Beschreibung der unterschiedlichen Versionen, folgen an dieser Stelle die erzeugten Messdaten der Implementierungen. In Abbildung 6.3 sind die benötigten Takte der unterschiedlichen Verarbeitungen dargestellt.

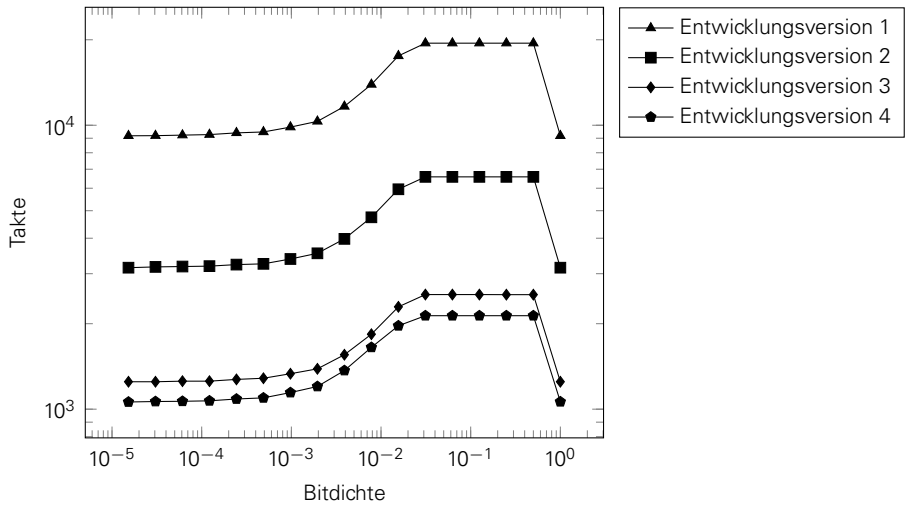

Abbildung 6.3: Vergleich der benötigten Takte für die WAH Verarbeitung - AND Verknüpfung

Alle Entwicklungsstufen zeigen bei Extremwerten der Bitdichte um 0 und 1 die Charakteristik der Verwendung von Lauflängenzählern bzw. Fills. Bitströme die fast ausschließlich aus Nullen bestehen (Bitdichte von 10^{-5}) lassen sich, genauso wie Bitströme die nur aus Einsen bestehen (Bitdichte von 10^0), gut komprimieren. Deswegen sind aus den stark komprimierten Bitmaps zu Verarbeitung nur wenige Ladeoperationen notwendig, weshalb die komprimierten Bitmaps schneller verknüpft werden können. Aus diesem Grund sind alle Implementierungsversionen in den Randbereichen besonders effizient, was zu einer geringen Taktanzahl und somit der schnelleren Abarbeitung führt. Da jedoch einer der verarbeiteten Indizes eine feste Verteilung mit der Kardinalität 128 hat, enthält dieser Index einige unterschiedliche Codeworte, sodass zur Verarbeitung der beiden Bitmap Indizes generell mehrere Iterationen notwendig sind. Die Anzahl der Iterationen zur Verarbeitung lässt sich an der Summe der Verarbeitungstakte erkennen. Dabei wird deutlich, dass im Intervall von 10^{-2} bis kurz vor 10^0 die höchste Taktanzahl notwendig ist. Das ist darauf zurückzuführen, dass der Index mit variierender Verteilung in diesem Bereich die meisten Codewörter enthält, da sich durch mehr gesetzte Bits die Möglichkeit der Kompression mit Fills reduziert. Aus diesem Grund müssen in dem Intervall besonders viele Codewörter einzeln verarbeitet werden, sodass die Anzahl der benötigten Takte insgesamt steigt. Da die Anzahl der Wörter jedoch nicht über die Anzahl der maximal möglichen Literale steigen kann, kommt es in diesem Bereich zur Ausbildung eines Plateaus.
Zwischen den TIE Entwicklungsstufen Eins und Zwei spiegelt sich das Reduzieren der benötigten Instruktionen sowie die Vergrößerung der Ladeinstruktionen von 32 Bit auf 128 Bit wieder. Für die gleiche Verarbeitung werden die benötigten Takte über den gesamten Bereich etwa geviertelt. Die weitere Reduzierung der notwendigen Instruktionen sowie die Nutzung der FLIX Optimierung führt etwa zu einer Halbierung der benötigten Takte zwischen den Versionen zwei und drei. Die vierte und finale Entwicklungsstufe kann durch Loop Unrolling und weitere Zusammenführung von Instruktionen die benötigten Takte noch weiter reduzieren, sodass über den gesamten Untersuchungsbereich etwa 10-mal weniger Takte benötigt werden, als bei der ersten Implementierungsvariante.

6.3.2 Messergebnisse der finalen Implementierung

Nachdem die Ergebnisse der vierten Implementierungsstufe bereits vorgestellt wurden, folgt nun ein Vergleich der neuen Instruktionen mit anderen Prozessoren sowie die Untersuchungen zum Speedup. Abbildung 6.4 stellt die benötigten Takte der einzelnen Prozessoren dar.

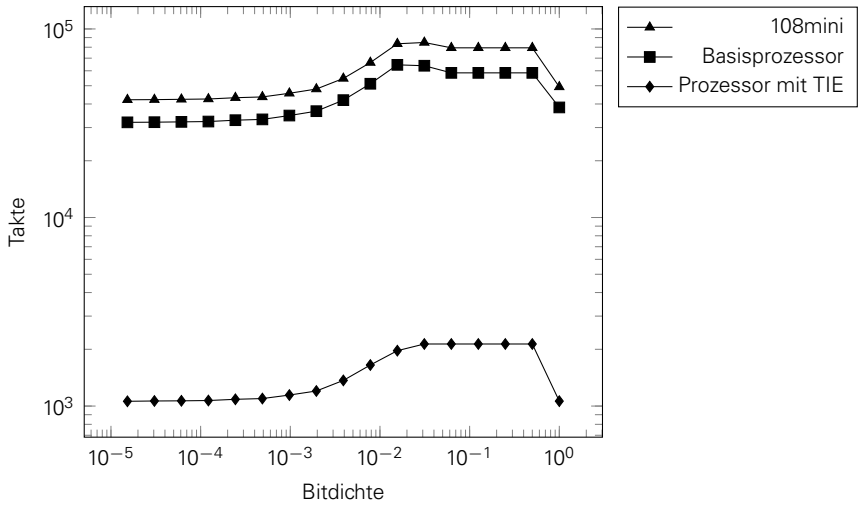

Abbildung 6.4: Vergleich der Entwicklungsstufen der WAH Verarbeitung - OR-Verknüpfung

Während der Basisprozessor nur eine kleine Verbesserung der benötigten Takte gegenüber dem 108 Mini aufweist, verwendet der angepasste Prozessor weniger als ein Zehntel der Takte für die gleiche Operation. Der Unterschied der beiden Standardprozessoren ist auf die Möglichkeit von FLIX Instruktionen bei dem Basisprozessor zurückzuführen. Dadurch können Instruktionen parallelisiert werden, was gegenüber dem 108 Mini eine Reduzierung der benötigten Takte zur Folge hat. Durch die spezielle Anpassung mit TIE Instruktionen kann der erweiterte Prozessor jedoch deutlich mehr Takte einsparen. Die Grenzwerte der Bitdichten von 0 und 1 zeigen in allen Graphen das typische Verhalten der stark komprimierten Bitmap Indizes. Im Intervall zwischen 10^{-2} und 10^0 liegt die benötigte Taktanzahl hingegen am höchsten, da hier die meisten Codewörter geladen und verarbeitet werden müssen. Die leichten Abweichungen der Messpunkte ohne TIE Implementierung können auf die Notwendigkeit unterschiedlicher Codewörter zurückgeführt werden. Da jeder Codeworttyp mit unterschiedlichen Funktionen verarbeitet werden muss ist der Verzweigungsgrad bei vielen unterschiedlichen Codewörtern am höchsten. Das führt zu einer erhöhten Menge benötigter Takte, die sich jeweils in dem Plateau wiederspiegeln, dass sich im oben genannten Intervall ausbildet.

Die oben dargestellt Grafik hat gezeigt, dass die benötigten Takte der Verarbeitung mit den TIE Instruktionen reduziert werden konnten. Diese Verbesserung resultiert in dem Speedup der in Abbildung 6.5 dargestellt ist.

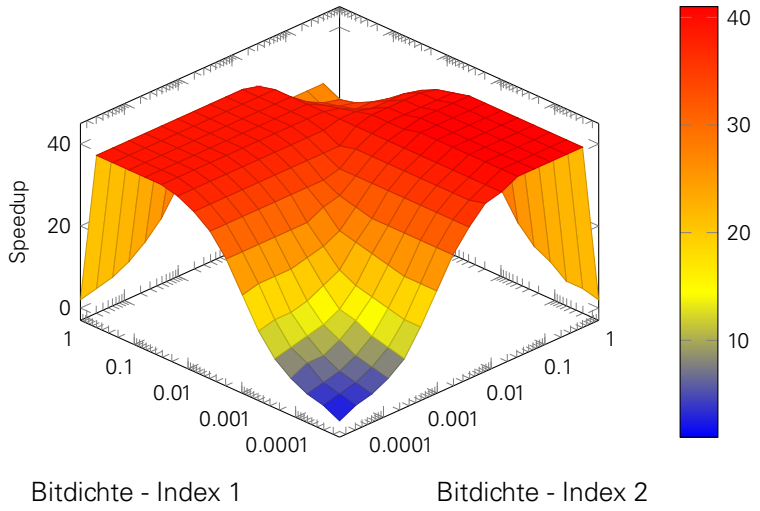

Abbildung 6.5: Speedup bei der WAH Operation - Intersektion

Die Grafik zeigt das Verhältnis der benötigten Takte für die Verarbeitung mit und ohne TIE Instruktionen. Da die Verarbeitung zwei Bitmap Indizes als Eingangsdatenströme benötigt, wird der Speedup in Abhängigkeit der Bitdichten beider Indizes dargestellt. Die Charakteristiken der Grafik sind im Wesentlichen auf die Größe der verarbeiteten Datenströme zurückzuführen. So enthalten die Bitmap Indizes mit geringer Bitdichte (Vordergrund der Grafik) wenige Elemente. Das gilt auch für die Bitdichte von 1 (Hintergrund der Grafik) und die Seitenbereiche der Grafik, bei denen Bitmaps verarbeitet werden, die jeweils aus 1- oder 0-Fills bestehen. In den Fällen einer kleinen Eingangsdatenmenge von komprimierten Bitmap Indizes, sind auch bei der Verarbeitung ohne zusätzliche Instruktionen nur wenige Operationen notwendig. Der Speedup sinkt in diesen Bereichen, da die verarbeitete Datenmenge klein ist und die zusätzlich notwendigen Vor- und Nachbereitungen bei der Verwendung von TIE Instruktionen in einem kleineren Speedup resultieren. In den Intervallen höherer Bitdichte (etwa zwischen 0,5 und 0,01) liegen hingegen viele Eingangsdaten vor. Da das Programm mit den Standardinstruktionen in diesem Bereich viele unterschiedliche Operationen durchführen muss, ist die Reduzierung der Takte mit den TIE Instruktionen besonders deutlich und der Speedup daher besonders groß.

6.4 PLWAH OPERATION

Um bei PLWAH die Optimierung durch hinzugefügte Instruktionen zu messen, wurde zunächst ein Vergleich der unterschiedlichen Prozessortypen vorgenommen. Die Ergebnisse der Messung sind in Abbildung 6.6 dargestellt.

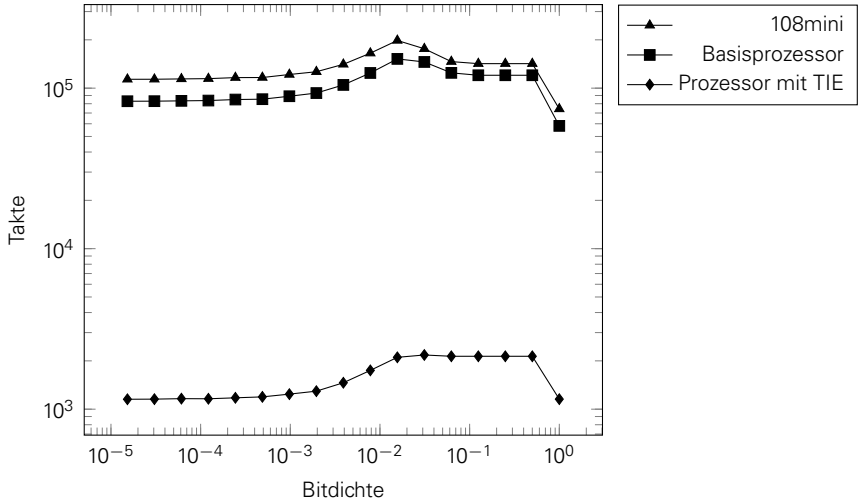

Abbildung 6.6: Vergleich der benötigten Takte für die PLWAH Verarbeitung - OR-Verknüpfung

In der Grafik lassen sich die bereits genannten Charakteristiken der RLE Komprimierung erkennen. So ist bei einer Bitdichte von 1 und nahe 0 die Kompression besonders effektiv, sodass die geringe Datenmenge in einer kleineren Anzahl benötigter Verarbeitungstakte resultiert. Weiterhin ist ab einer Bitdichte von 0,5 die Länge der Bitmap mit variierender Verteilung ausschlaggebend für die zusätzlich benötigten Takte. Dieser Einfluss auf die Anzahl der Takte sinkt jedoch bei Bitdichten kleiner als 0,01, sodass in diesem Bereich die Länge der Bitmap mit fester Verteilung am relevantesten für die benötigten Verarbeitungstakte wird. In der Abbildung lässt sich zudem das Maximum der Verarbeitungstakte bei etwa 0,02 erkennen. Das gilt jedoch nur für die Quellcodes ohne TIE Erweiterung. Der Grund hierfür liegt in den Sparse Codewörtern von PLWAH. Bei der OR-Verknüpfung der Bitmap Indizes entstehen in diesem Bereich besonders viele Fills, die sich mit Sparse Codewörtern zusammenführen lassen. Aus diesem Grund werden zusätzliche Takte benötigt, um die Spezialwörter von PLWAH zu erzeugen, was in einem Maximalwert der insgesamt verwendeten Takte resultiert. Die Erzeugung dieser Spezialwörter ist bei den TIE Instruktionen jedoch mit in den normalen Verarbeitungsinstruktionen eingebracht, sodass keine zusätzlichen Instruktionen notwendig werden. Entsprechend führt die Erstellung der Spezialwörter bei TIE zu keinen zusätzlichen Takten, sodass sich in diesem Bereich kein neues Maximum ausbildet.

Um die erzielten Optimierungen mit den zusätzlichen Instruktionen darzustellen, wurde die Verarbeitungstakte mit und ohne TIE Instruktionen ins Verhältnis gesetzt. Das Ergebnis der Messungen ist in Abbildung 6.7 dargestellt.

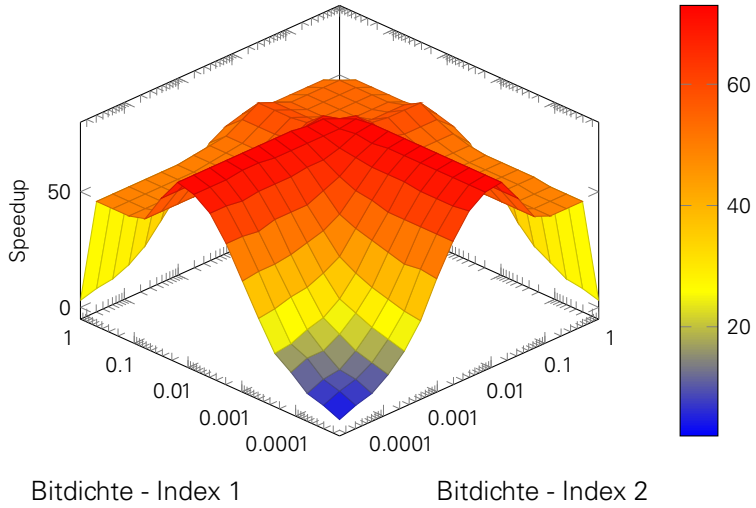

Abbildung 6.7: Speedup bei der PLWAH Operation - Vereinigung

In den Grenzbereichen besonders geringer oder besonders hoher Bitdichten liegt bei PLWAH die stärkste Kompressionsrate vor, weshalb die Menge der zu verarbeitenden Daten geringer wird und sich damit der Speedup verringert. Dieses Verhalten ist an allen vier Eckpunkten der Grafik zu beobachten und es entspricht den Ausführungen, die bereits bei WAH gemacht wurden. Bei PLWAH liegt durch den Einsatz der Spezialwörter eine höhere Komplexität im Vergleich zu WAH vor. Das führt zu mehr benötigten Takten bei der Verarbeitung ohne TIE Instruktionen, sodass der Speedup größer wird, als bei WAH. Insbesondere im Bereich der Bitdichten von 0,02 tritt deswegen ein erhöhter Speedup von bis zu Faktor 73 auf.

6.5 COMPAX OPERATION

Den komplexesten der implementierten Algorithmen stellt COMPAX dar. Um auch hier die Optimierungen ins Verhältnis zu setzen, stellt Abbildung 6.8 die benötigten Takte der Abarbeitung für den Prozessor mit TIE Instruktionen genauso wie für dessen Basisprozessor und den 108 Mini dar.

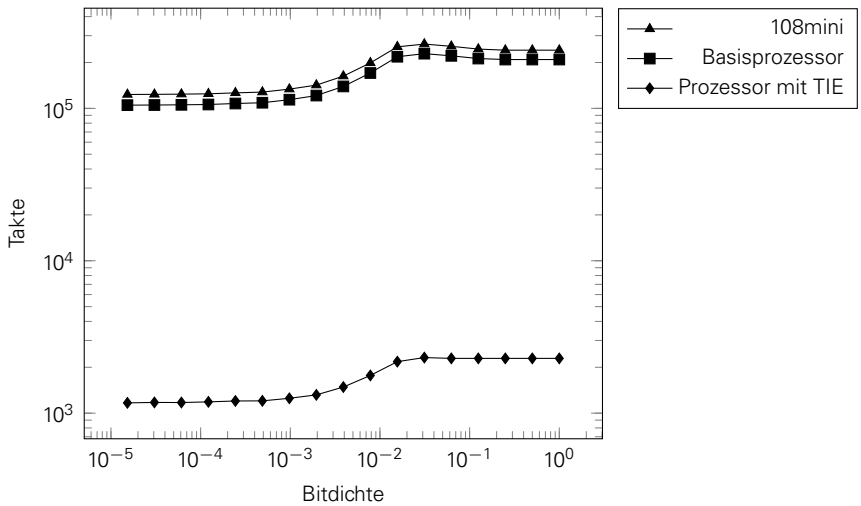

Abbildung 6.8: Vergleich der benötigten Takte für die COMPAX Verarbeitung - OR-Verknüpfung

Der Verlauf der Graphen kann wieder als zwei Plateaus und deren Übergang betrachtet werden. Äquivalent zu den Schilderungen der vorherigen Algorithmen, ist das auf die Länge der komprimierten Eingangsvektoren zurückzuführen. Allerdings besitzen die Graphen kein Minimum bei einer Bitdichte von 1, da COMPAX für 1-Fills kein separates Codewort vorsieht und die Länge des Vektors somit auch für den Fall von vielen 1-Bits gleich bleibt. Der Maximalwert der benötigten Takte liegt wieder im Bereich von 0,02, da hier die meisten Codewörter zu Spezialwörtern zusammengeführt werden, was zu mehr benötigten Takten führt. Die zusätzliche Komplexität der Spezialwörter von COMPAX sorgt dafür, dass die Implementierungen ohne TIE etwa 100-mal mehr Takte benötigten, als die Variante mit erweiterten Instruktionen.

Um diesen Speedup wieder für möglichst viele Verteilungskombinationen der Bitmap Indizes darzustellen, wurden auch für COMPAX die Messungen des Speedups mit unterschiedlichen Kombinationen durchgeführt. Die Ergebnisse der Messung mit 256 Datenpunkten ist in Abbildung 6.9 dargestellt.

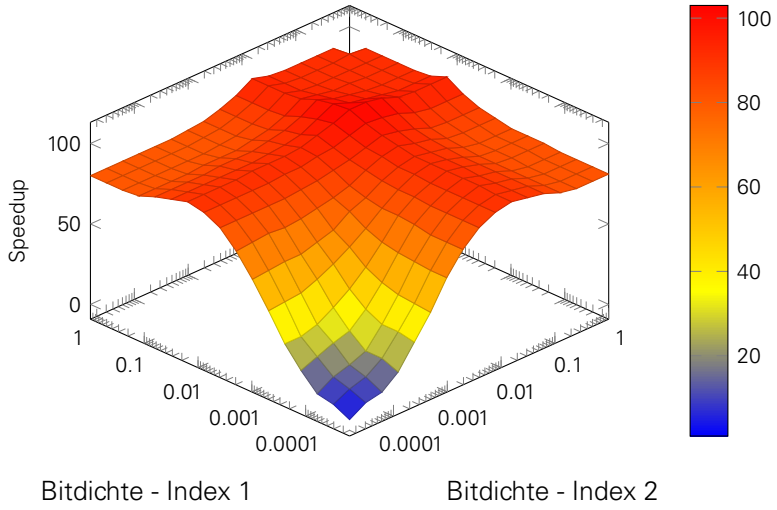

Abbildung 6.9: Speedup bei der COMPAX Operation - symmetrische Differenz

Die Grafik zeigt den geschilderten Speedup von bis zu Faktor 103. In den Intervallen, in denen viele Eingangselemente vorliegen, ist der Speedup höher als im stark komprimierten Bereich mit den Bitdichten um 0,0001 (Vordergrund der Grafik). Insbesondere im Bereich der Bitdichten von 0,02 ist eine Erhöhung des Speedups zu erkennen. Diese ist wieder auf die Kombination mehrerer Codewörter zu Spezialwörtern zurückzuführen. Die C Implementierung benötigt dazu zusätzliche Takte, während die TIE Implementierung diese Funktion im Rahmen der Fusion Strategie bereits in der Verarbeitungsinstruktion integriert hat, was zu dem zusätzlichen Speedup führt

6.6 WAH KOMPRESSION

Neben den Instruktionen zur Verknüpfung der komprimierten Bitmap Indizes wurden im Fall von WAH auch Instruktionen zur Beschleunigung von Kompression und Dekompression entworfen. Die Vergleiche der unterschiedlichen Prozessoren für den Fall der Kompression sind in Abbildung 6.10 dargestellt.

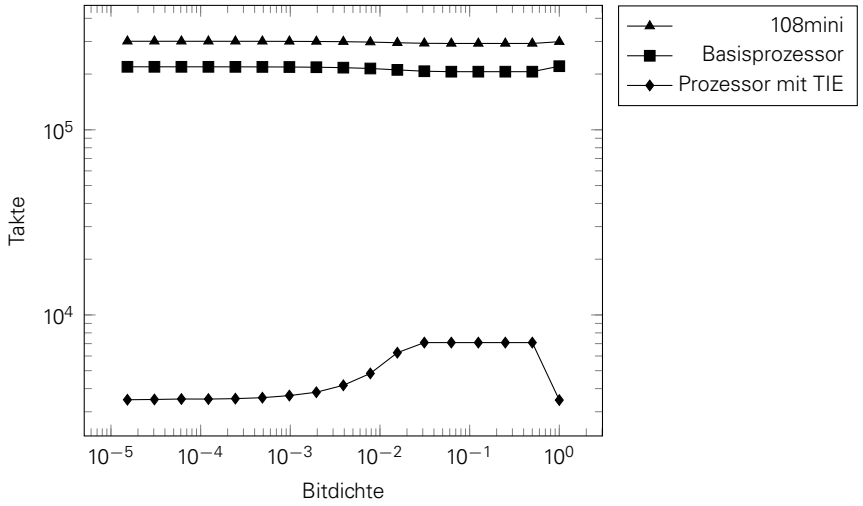

Abbildung 6.10: Vergleich der benötigten Takte für die Kompression mit WAH - OR-Verknüpfung

Bei den oberen beiden Graphen der Abbildung lässt sich ein relativ geradliniger Verlauf feststellen. Das ist darauf zurückzuführen, dass bei der Kompression immer über die gleiche Anzahl von 50.000 Bits iteriert werden muss. Allerdings muss in den Bereichen der Fills, bei besonders hoher oder besonders niedriger Bitdichte, häufig auf das vorherige Element zugegriffen werden. Dieser Zugriff ist mit einigen logischen Vergleichen und eventuell dem Überschreiben des vorherigen Lauflängenzählers verbunden. Da hierfür mehr Takte aufgewendet werden müssen als für das Anhängen an den Ausgangsdatenstrom, liegen die Maxima der beiden Graphen im Bereich der Fills.
Bei der Version mit TIE Implementierung liegen in den Grenzbereichen hingegen die Minima, da die eingesetzte Technik des Prefetchings hier die maximale Effizienz hat. In diesem Fall werden mit jeder Iteration 124 Bit Daten verarbeitet, wohingegen bei wechselnden Bitmustern mit jedem Takt nur 31 Bit verarbeitet werden.

Um den Bedarf der Verarbeitungstakte mit und ohne Instruktionen in Relation zueinander zu setzen, wurde auch für die Kompression eine Messung des Speedups über den unterschiedlichen Eingangsvektoren durchgeführt. Die Ergebnisse sind in Abbildung 6.11 dargestellt.

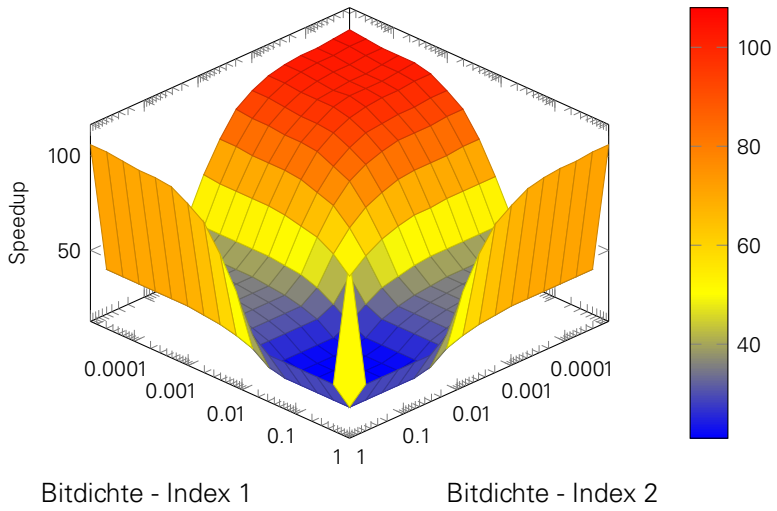

Abbildung 6.11: Speedup bei der WAH Kompression - AND-Verknüpfung

Die Perspektive der Grafiken hat sich im Gegensatz zu den vorherigen Darstellungen um 180 Grad gedreht. Im Vordergrund der Grafik sind besonders hohe Bitdichten dargestellt, während der Hintergrund die Messungen für niedrige Bitdichten enthält. Die beobachteten Geschwindigkeitszuwächse in den Bereichen mit vielen Fills lassen sich auch hier erkennen. So ist der Speedup sowohl bei einer Bitdichte von 1, als auch bei sehr geringen Bitdichten, hoch. In den Übergangsphasen zwischen den Bitdichten von 0,5 und 0,01 mit vielen Literalen ist der Speedup hingegen geringer, da Prefetching hier nicht effizient eingesetzt werden kann. Bei der Kompression handelt es sich um eine unäre Operation, sodass im Prinzip ein Bitmap Index zur Visualisierung des Speedups ausreicht. Allerdings sollte hier der Vergleich zu den vorherigen Graphen ermöglicht werden, weshalb die Ergebnisse eine Kompression von zwei Bitmap Indizes mit den jeweiligen Verteilungen wiederspiegeln.

6.7 WAH DEKOMPRESSION

Der Vergleich der unterschiedlichen Prozessoren wurde auch bei der Dekompression vorgenommen. Generell ist bei den Messergebnissen zu beachten, dass es sich auch bei der Dekompression um eine Operation handelt, die nur eine Eingangsbitmap verarbeitet. Damit die Darstellung trotzdem in der bisherigen Weise erfolgen kann, wurden zunächst zwei komprimierte Bitmap Indizes miteinander verknüpft und das Ergebnis dann mit der Dekompression verarbeitet. Das führt dazu, dass der Verknüpfungsoperator der beiden Indizes wesentlichen Einfluss auf die Messergebnisse hat. Der Vergleich der Prozessoren ist in Abbildung 6.12 dargestellt.

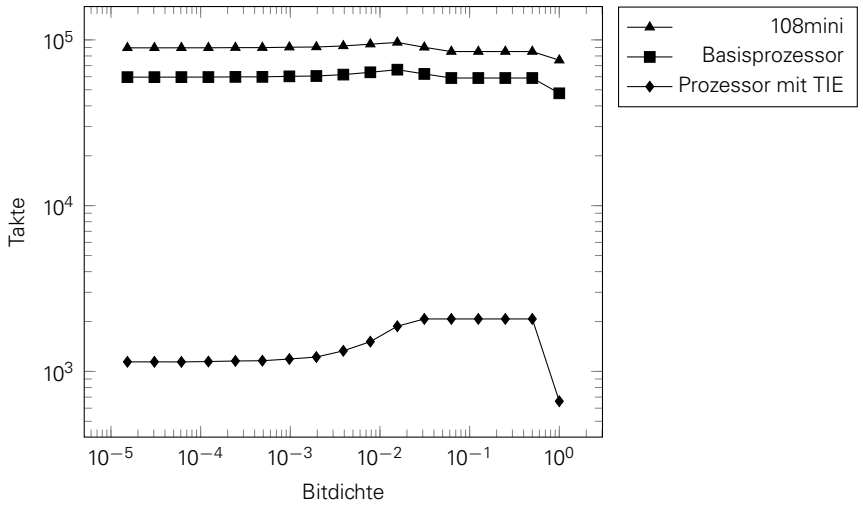

Abbildung 6.12: Vergleich der benötigten Takte für die Dekompression mit WAH - OR-Verknüpfung

Bei dem dargestellten Prozessorvergleich wurde zuvor die Vereinigungsmenge der beiden Bitmap Indizes erzeugt. Die beiden oberen Kennlinien haben ihr Minimum bei einer Bitdichte von 1, da hier ausschließlich 1-Fills geschrieben werden. Im Bereich niedriger Bitdichten nimmt hingegen der Bitmap Index mit fester Kardinalität den größten Einfluss auf die benötigten Takte, da die wechselnden Codewörter komplett in den unkomprimierten Datenstrom übernommen werden und es somit nicht zur Etablierung langer Folgen von 0-Bits kommt. Das Maximum bei einer Bitdichte von etwa 0,02 ist auf die maximale Anzahl unterschiedlichen Codewörtern zurückzuführen, da hier sowohl Literale als auch Fills gebildet werden und dieser Wechsel für zusätzliche benötigte Takte sorgt. Diese Charakteristik ist für die beide Graphen ohne TIE Erweiterung äquivalent,

auch wenn der Basisprozessor mit Nutzung der FLIX Instruktionen wie bei allen vorherigen Ausführungen schneller ist, als der 108 Mini.
Die untere Kurve als Repräsentation der TIE Operation weist hingegen in einigen Bereichen ein anderes Verhalten auf, das auf die Prefetching Funktionalität zurückzuführen ist. So resultiert die Verknüpfung der Bitmaps mit den Bitdichten 128 und 1 ausschließlich in 1-Fills. Im Ergebnis können mit jedem Verarbeitungstakt 124 Bits geschrieben werden, was die Taktanzahl insgesamt minimiert. Ab einer Bitdichte von 0,5 bei der Bitmap mit variabler Kardinalität, können jedoch mit jedem Verarbeitungstakt nur 31 Bit geschrieben werden. Das führt dazu, dass in dem Bereich mit vielen Literalen am meisten Takte für die Verarbeitungsinstruktion verbraucht werden. Bei einer geringen Bitdichte können hingegen alle vorhanden 0-Fills geschrieben werden, sodass das Prefetching im Bereich niedriger Bitdichten für eine verringerte Taktanzahl sorgt. Jedoch wird die Anzahl der 0-Fills durch die OR-Verknüpfung der Bitmap Indizes durch den Index mit fester Bitdichte begrenzt, sodass die benötigten Takte auch im Grenzbereich gleich bleiben.

Nachdem die ermittelten Taktfrequenzen auf Basis eines variierenden Bitmap Index festgestellt wurden, folgt nun die Auswertung der Dekompression bei zwei Indizes mit wechselnden Bitdichten. Der Speedup der TIE Instruktionen ist in Abbildung 6.13 dargestellt.

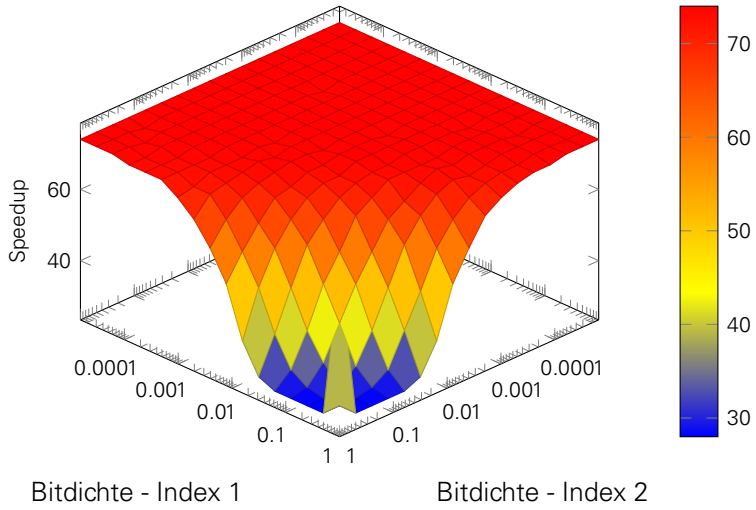

Abbildung 6.13: Speedup bei der WAH Dekompression - AND-Verknüpfung

Die Darstellung basiert auf der Dekompression des Ergebnisses von zwei Bitmap Indizes nach einer AND-Verknüpfung. Demzufolge sind insbesondere bei niedrigen Bitdichten der Eingangsindizes viele 0-Fills zu erwarten. Dieser hintere Bereich der Grafik ist gekennzeichnet durch die höchsten Speedupwerte der Verarbeitung. Das ist auf die Prefetch Funktionalität zurückzuführen, da hier besonders viele Bits je Instruktion geschrieben werden. Der vordere Bereich der Grafik mit höheren Bitdichten ist hingegen durch viele unterschiedliche Codewörter im komprimierten Datenstrom gekennzeichnet, weshalb Prefetching hier nicht zum Einsatz kommt und der Speedup entsprechend geringer ausfällt.

7 ZUSAMMENFASSUNG UND AUSBLICK

Die vorliegende Diplomarbeit hatte das Ziel, Instruktionserweiterungen für Bitmap Indizes zu anlysieren und zu entwickeln. Der eingesetzte Prozessor aus der Xtensa LX Familie stellte, zusammen mit den Erweiterungsmöglichen durch die Tensilica Instruction Extension, die Basis der Untersuchungen dar. Die im Kapitel 4 aufgezeigten Publikationen hatten in der Vergangenheit bereits gezeigt, dass die gezielte Nutzung von Instruktionen abseits der RISC Architektur, vorhandene Algorithmen und Quellcodes beschleunigen kann. Diese Arbeit ging einen Schritt weiter und optimierte nicht nur den Quellcode hardwarespezifisch, sondern passte im Sinne des Hardware-Software Codesigns beide Komponenten aneinander an. Da die untersuchten Algorithmen zur Kompression von Bitmap Indizes noch nicht als Quellcode vorlagen, war das Verständnis der Theorie hinter den Algorithmen der erste Schritt. Nach der in Kapitel 2 beschriebenen Analyse der theoretischen Grundlagen war der nächste Schritt die C-Implementierung von WAH, PLWAH und COMPAX. Die Implementierung umfasste hierbei neben der Funktionalität von Kompression und Dekompression auch die Verarbeitung der komprimierten Bitmap Indizes für jeden Algorithmus. Die Verknüpfung der Bitmap Indizes erfolgte mit den logischen Operatoren AND, OR und XOR, die sich im Datenbankumfeld als besonders relevant herausgestellt haben. Im Anschluss an die allgemeine Realisierung der Algorithmen in C, folgte die systemspezifische Anpassung. Kapitel 3 erläutert zunächst die dafür benötigten Grundlagen von TIE als Programmiersprache. Im Anschluss wurde, wie in Kapitel 5 aufgezeigt, in mehreren Iterationen durch den TIE Entwicklungszyklus für jeden Algorithmus ein spezieller Satz von Instruktionen entwickelt. Die weitere Untersuchung der einzelnen Optimierungen erfolgte in Kapitel 6.

Die Evaluierung der Resultate zeigte insgesamt, dass sich durch die Erweiterung von Programmcode mit angepassenten Instruktionen, eine signifikante Anzahl benötigter Verarbeitungstakte einsparen lässt. Diese Einsparung wächst mit der Komplexität der ersetzten Funktionen bzw. deren Lade- und Speicheraufkommen. So konnte bei WAH als dem einfachsten der untersuchten Algorithmen ein Speedup von maximal Faktor 41 erreicht werden, während PLWAH mit Faktor 77 und COMPAX mit Faktor 104 stärker optimiert werden konnten. Allerdings geht die erhöhte Komplexität in den Instruktionen mit einer sinkenden Taktfrequenz einher. So sind die maximal erreichbaren Taktfrequenzen des erweiterten Prozessors von 510 MHz mit WAH auf 384 MHz mit COMPAX gesunken. Zwar ist COMPAX der komplexeste Algorithmus mit dem größten Potential der Optimierung, aber die Komplexität spiegelt sich auch in steigender Länge des kritischen Pfades wieder, weshalb die Taktfrequenz insgesamt sinkt. Es muss also eine sorgfältige Abwägung vorgenommen werden, welches Ziel der Optimierung das wichtigste ist.
Neben der Komplexität des eigentlichen Algorithmus, stellt die Menge der zu verarbeitenden Daten einen wesentlichen Optimierungsaspekt dar. Die breiten Anbindungen an den Systembus der Load/Store Units können einen wesentlichen Beitrag zur Optimierung leisten. So wurden mit Hilfe der breiteren Lade- und Speicheroperationen die WAH Algorithmen zur Kompression und

Dekompression optimiert. Mit Hilfe dieser Technik konnte die Dekomprimierung mit bis zu 74-mal und die Komprimierung mit bis zu 108-mal weniger Takten auskommen, als die äquivalenten Funktionen ohne Instruktionserweiterung.

Zusammenfassend hat die vorliegende Diplomarbeit gezeigt, dass bei allen untersuchten Algorithmen eine Beschleunigung erzielt werden konnte. Die Anwendungsfälle der Datenbanken bieten somit ein interessantes Umfeld für weitere Forschungsarbeiten mit dem Hintergrund des Hardware- Software Codesigns. Dabei könnten sowohl die vorhandenen Implementierungen erweitert werden, als auch Ausgangsbedingen erforscht werden.
Die Erweiterung der bisherigen Implementierungen kann beispielsweise durch Ergänzung der Kompressions- und Dekompressionsinstruktionen für PLWAH und COMPAX erfolgen. Als ausschlaggebend für die erzielten Optimierungen hat sich unter anderem die Breite der Speicheranbindung gezeigt. Nach einer Erweiterung dieser, könnte der Datendurchsatz weiter gesteigert werden und die Datenintensiven Anwendungsfälle über das aktuelle Maß hinaus beschleunigt werden. Denkbar wäre auch der Einsatz von mehr Pipelinestufen, sodass noch mehr als bisher von nebenläufigen Instruktionen profitiert werden könnte.
Einen limitierenden Faktor der aktuellen Untersuchungen stellt die Größe der Prozessorspeicher dar. Das effiziente Nachladen weiterer Daten in den Speicher könnte somit genauso Schwerpunkt einer Forschungsarbeit sein, wie die Anbindung neuer Speicher an die Prozessorinterfaces und deren Integration. Weiterhin wurde in der vorliegenden Arbeit die Möglichkeiten eines Prozessors mit einem Kern untersucht. Das Verbinden mehrerer Prozessoren zur parallelen Verarbeitung mit den vorhandenen oder neuen Instruktionen könnte ein weiteres Forschungsthema darstellen.

A ANHANG

A.1 WAH - SPEEDUP MESSUNGEN

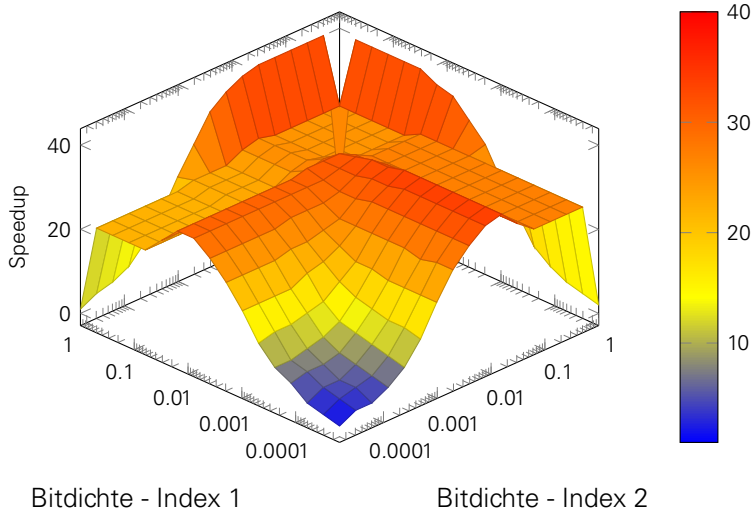

Abbildung A.1: Speedup bei der WAH Operation - Vereinigung

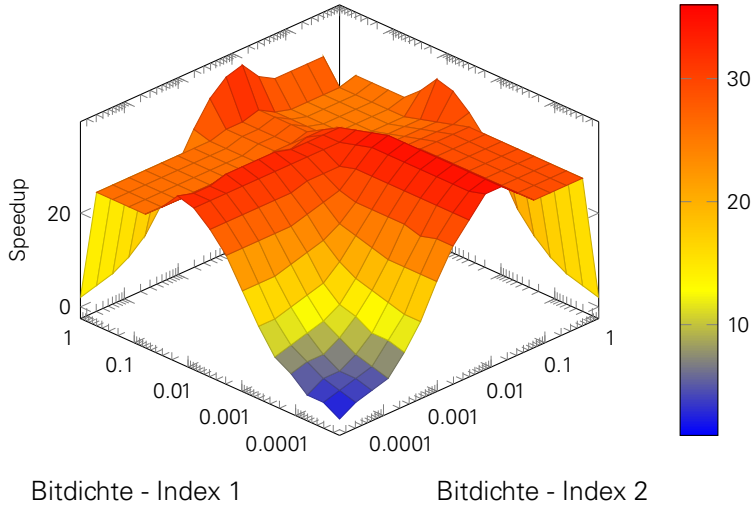

Abbildung A.2: Speedup bei der WAH Operation - symmetrische Differenz

A.2 PLWAH - SPEEDUP MESSUNGEN

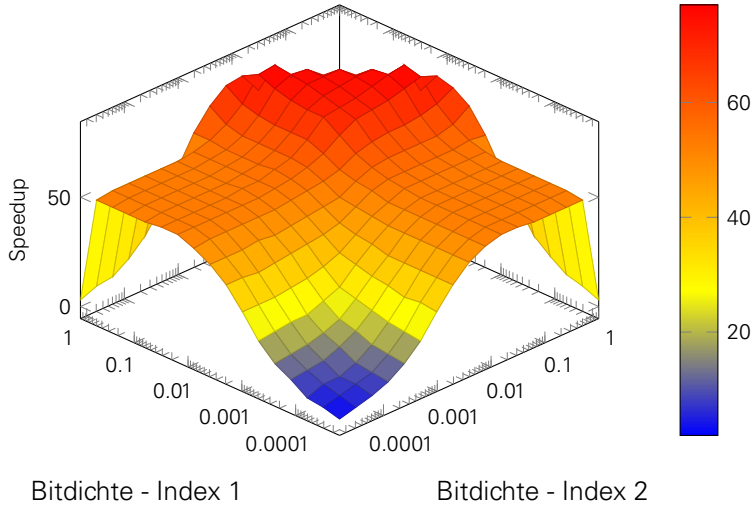

Abbildung A.3: Speedup bei der PLWAH Operation - Intersektion

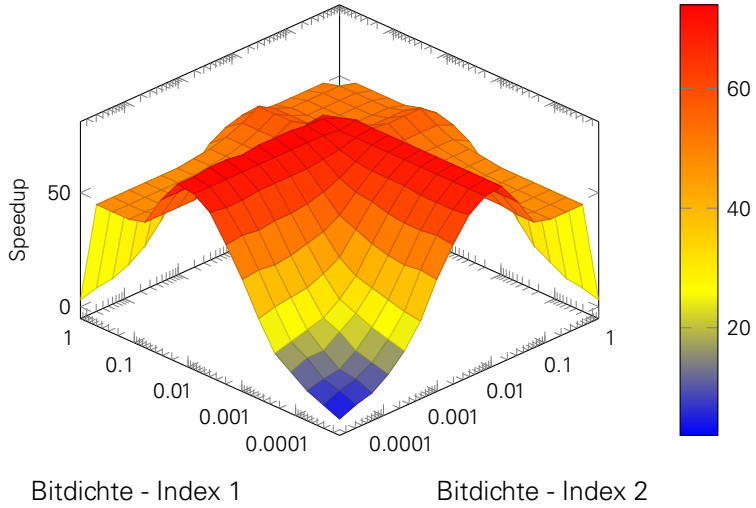

Abbildung A.4: Speedup bei der PLWAH Operation - symmetrische Differenz

A.3 COMPAX - SPEEDUP MESSUNGEN

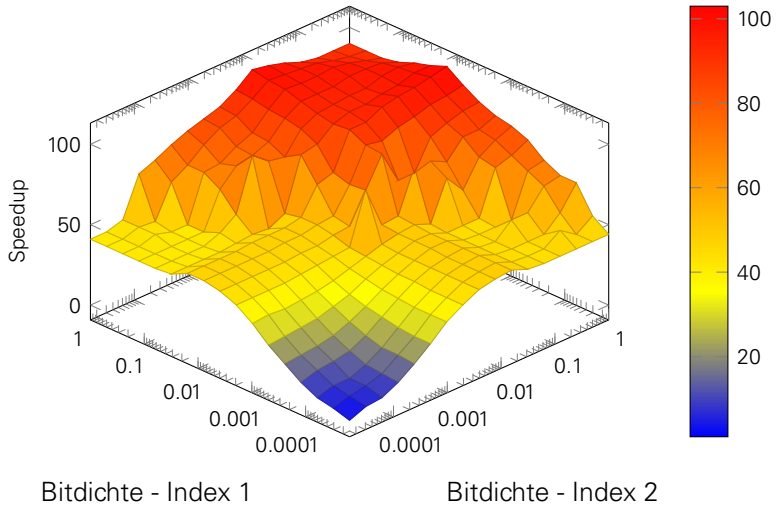

Abbildung A.5: Speedup bei der COMPAX Operation - Intersektion

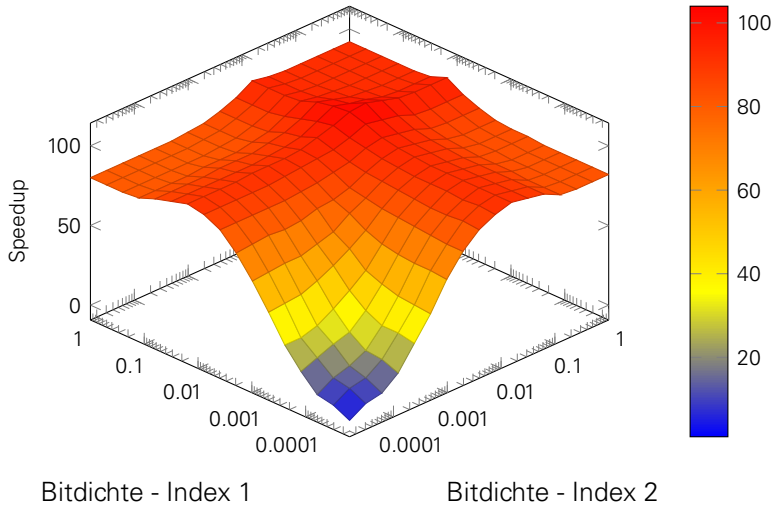

Abbildung A.6: Speedup bei der COMPAX Operation - Vereinigung

LITERATURVERZEICHNIS

[AHF+14] ARNOLD, Oliver ; HAAS, Sebastian ; FETTWEIS, Gerhard ; SCHLEGEL, Benjamin ; KISSINGER, Thomas ; LEHNER, Wolfgang: An application-specific instruction set for accelerating set-oriented database primitives. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* ACM, 2014, S. 767–778

[Ant94] ANTOSHENKOV, G.: *Byte aligned data compression*. November 8 1994. – US Patent 5,363,098

[AW10] ANDRZEJEWSKI, Witold ; WREMBEL, Robert: GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid. In: *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part II*. Berlin, Heidelberg : Springer-Verlag, 2010 (DEXA'10). – ISBN 3–642–15250–3, 978–3–642–15250–4, S. 315–329

[BD83] BORAL, Haran ; DEWITT, DavidJ.: Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines. Version: 1983. http://dx.doi.org/10.1007/978-3-642-69419-6. In: LEILICH, H.-O. (Hrsg.) ; MISSIKOFF, M. (Hrsg.): *Database Machines*. Springer Berlin Heidelberg, 1983. – DOI 10.1007/978–3–642–69419–6. – ISBN 978–3–540–12959–2, S. 166–187

[Cod70] CODD, Edgar F.: A relational model of data for large shared data banks. In: *Communications of the ACM* 13 (1970), Nr. 6, S. 377–387

[CP10] COLANTONIO, Alessandro ; PIETRO, Roberto D.: Concise: Compressed 'n' Composable Integer Set. In: *Information Processing Letters* 110 (2010), Nr. 16, S. 644 – 650. – ISSN 0020–0190

[DMG97] DE MICHELL, G ; GUPTA, Rajesh K.: Hardware/software co-design. In: *Proceedings of the IEEE* 85 (1997), Nr. 3, S. 349–365

[DP10] DELIÈGE, François ; PEDERSEN, Torben B.: Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. In: *Proceedings of the 13th International Conference on Extending Database Technology* ACM, 2010, S. 228–239

[Fly66] FLYNN, M.: Very high-speed computing systems. In: *Proceedings of the IEEE* 54 (1966), Dec, Nr. 12, S. 1901–1909. http://dx.doi.org/10.1109/PROC.1966.5273. – DOI 10.1109/PROC.1966.5273. – ISSN 0018–9219

[FSV10] FUSCO, Francesco ; STOECKLIN, Marc P. ; VLACHOS, Michail: Net-fli: on-the-fly compression, archiving and indexing of streaming network traffic. In: *Proceedings of the VLDB Endowment* 3 (2010), Nr. 1-2, S. 1382–1393

[FVDD13a] FUSCO, Francesco ; VLACHOS, Michail ; DIMITROPOULOS, Xenofontas ; DERI, Luca: Indexing Million of Packets Per Second Using GPUs. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. New York, NY, USA : ACM, 2013 (IMC '13). – ISBN 978–1–4503–1953–9, S. 327–332

[FVDD13b] FUSCO, Francesco ; VLACHOS, Michail ; DIMITROPOULOS, Xenofontas ; DERI, Luca: Indexing Million of Packets Per Second Using GPUs. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. New York, NY, USA : ACM, 2013 (IMC '13). – ISBN 978–1–4503–1953–9, S. 327–332

[GLW+04] GOVINDARAJU, Naga K. ; LLOYD, Brandon ; WANG, Wei ; LIN, Ming ; MANOCHA, Dinesh: Fast Computation of Database Operations Using Graphics Processors. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2004 (SIGMOD '04). – ISBN 1–58113–859–8, S. 215–226

[Gon00] GONZALEZ, Ricardo E.: Xtensa: A configurable and extensible processor. In: *IEEE micro* 20 (2000), Nr. 2, S. 60–70

[Häb11] HÄBERLEIN, Tobias: Technische Informatik. In: *Technische Informatik: Ein Tutorium der Maschinenprogrammierung und Rechnertechnik, ISBN 978-3-8348-1372-5. Vieweg+ Teubner Verlag | Springer Fachmedien Wiesbaden GmbH, Wiesbaden, 2011* 1 (2011)

[HOM08] HASSABALLAH, M. ; OMRAN, Saleh ; MAHDY, Youssef B.: A Review of SIMD Multimedia Extensions and Their Usage in Scientific and Engineering Applications. In: *Comput. J.* 51 (2008), November, Nr. 6, S. 630–649. `http://dx.doi.org/10.1093/comjnl/bxm099`. – DOI 10.1093/comjnl/bxm099. – ISSN 0010–4620

[Hsi79] HSIAO, D.K.: Data Base Machines are Coming, Data Base Machines are Coming! In: *Computer* 12 (1979), March, Nr. 3, S. 7–9. `http://dx.doi.org/10.1109/MC.1979.1658647`. – DOI 10.1109/MC.1979.1658647. – ISSN 0018–9162

[Joh99] JOHNSON, Theodore: Performance measurements of compressed bitmap indices. In: *Proceedings of the 25th International Conference on Very Large Data Bases* Morgan Kaufmann Publishers Inc., 1999, S. 278–289

[KLA08] KASER, Owen ; LEMIRE, Daniel ; AOUICHE, Kamel: Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes. In: *Proceedings of the ACM 11th international workshop on Data warehousing and OLAP* ACM, 2008, S. 1–8

[KR07] KUON, Ian ; ROSE, J.: Measuring the Gap Between FPGAs and ASICs. In: *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 26 (2007), Feb, Nr. 2, S. 203–215. `http://dx.doi.org/10.1109/TCAD.2006.884574`. – DOI 10.1109/TCAD.2006.884574. – ISSN 0278–0070

[Lan11] LANGI, AZ.R.: An FPGA implementation of a simple lossless data compression coprocessor. In: *Electrical Engineering and Informatics (ICEEI), 2011 International Conference on*, 2011. – ISSN 2155–6822, S. 1–4

[LB12] LEMIRE, Daniel ; BOYTSOV, Leonid: Decoding billions of integers per second through vectorization. In: *CoRR* abs/1209.2137 (2012)

[LBK14] LEMIRE, Daniel ; BOYTSOV, Leonid ; KURZ, Nathan: SIMD Compression and the Intersection of Sorted Integers. In: *CoRR* abs/1401.6399 (2014)

[Leh03] LEHNER, Wolfgang: *Datenbanktechnologie für Data-Warehouse-Systeme*. dpunkt-Verlag, 2003

[Lei06] LEIBSON, Steve: *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*. Academic Press, 2006

[LV04] LYMAN, Peter ; VARIAN, Hal: *How much information?* `sims.berkeley.edu/research/projects/how-much-info-2003/`. Version: 2004, Abruf: 06.06.2014

[MT09] MUELLER, Rene ; TEUBNER, Jens: FPGA: What's in It for a Database? In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2009 (SIGMOD '09). – ISBN 978–1–60558–551–2, S. 999–1004

[MTA09] MUELLER, Rene ; TEUBNER, Jens ; ALONSO, Gustavo: Data Processing on FPGAs. In: *Proc. VLDB Endow.* 2 (2009), August, Nr. 1, S. 910–921. `http://dx.doi.org/10.14778/1687627.1687730`. – DOI 10.14778/1687627.1687730. – ISSN 2150–8097

[OFW99] OBERMAN, Stuart ; FAVOR, Greg ; WEBER, Fred: AMD 3DNow! technology: Architecture and implementations. In: *Micro, IEEE* 19 (1999), Nr. 2, S. 37–48

[OH05] OLUKOTUN, Kunle ; HAMMOND, Lance: The Future of Microprocessors. In: *Queue* 3 (2005), September, Nr. 7, S. 26–29. `http://dx.doi.org/10.1145/1095408.1095418`. – DOI 10.1145/1095408.1095418. – ISSN 1542–7730

[O'N89] O'NEIL, Patrick E.: Model 204 architecture and performance. In: *High Performance Transaction Systems*. Springer, 1989, S. 39–59

[OOW07] O'NEIL, E. ; O'NEIL, P. ; WU, Kesheng: Bitmap Index Design Choices and Their Performance Implications. In: *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, 2007. – ISSN 1098–8068, S. 72–84

[PW96] PELEG, Alex ; WEISER, Uri: MMX technology extension to the Intel architecture. In: *Micro, IEEE* 16 (1996), Nr. 4, S. 42–50

[PWW97] PELEG, Alex ; WILKIE, Sam ; WEISER, Uri: Intel MMX for multimedia PCs. In: *Communications of the ACM* 40 (1997), Nr. 1, S. 24–38

[Sal07] SALOMON, David: *Data Compression: The Complete Reference*. Berlin, Germany / Heidelberg, Germany / London, UK / etc. : Springer-Verlag, 2007. – xxv + 1092 S. – ISBN 1–84628–602–6. – With contributions by Giovanni Motta and David Bryant.

[SGL10] SCHLEGEL, Benjamin ; GEMULLA, Rainer ; LEHNER, Wolfgang: Fast Integer Compression Using SIMD Instructions. In: *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. New York, NY, USA : ACM, 2010 (DaMoN '10). – ISBN 978–1–4503–0189–3, S. 34–40

[SM13] SAECKER, Michael ; MARKL, Volker: Big data analytics on modern hardware architectures: A technology survey. In: *Business Intelligence*. Springer, 2013, S. 125–149

[SQW13] SHI, Shaobo ; QI, Yue ; WANG, Qin: FPGA Acceleration for Intersection Computation in Frequent Itemset Mining. In: *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013 International Conference on*, 2013, S. 514–519

[SW07] STOCKINGER, Kurt ; WU, Kesheng: Bitmap indices for data warehouses. In: *Data Warehouses and OLAP: Concepts, Architectures and Solutions* 5 (2007), S. 157–178

[Ten10] TENSILICA: *Diamond Standard 108Mini Controller*. `http://ip.cadence.com/uploads/pdf/108Mini.pdf`. Version: 2010, Abruf: 23.07.2014

[Ten11] TENSILICA: *Tensilica Instruction Extension (TIE) Language User's Guide*. 2011

[Ten12] TENSILICA: *Xtensa LX4 Microprocessor Data Book*. 2012

[TGRM14] TURNER, Vernon ; GANTZ, John ; REINSEL, David ; MINTON, Stephen: The Digital Universe of Opportunities: Rich Data and the The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. In: *IDC iView: IDC Analyze the Future* (2014)

[WLP+14] WU, Lisa ; LOTTARINI, Andrea ; PAINE, Timothy K. ; KIM, Martha A. ; ROSS, Kenneth A.: Q100: The Architecture and Design of a Database Processing Unit. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2014 (ASPLOS '14). – ISBN 978–1–4503–2305–5, S. 255–268

[WOS01] WU, Kesheng ; OTOO, Ekow J. ; SHOSHANI, Arie: A Performance Comparison of Bitmap Indexes. In: *Proceedings of the Tenth International Conference on Information and Knowledge Management*. New York, NY, USA : ACM, 2001 (CIKM '01). – ISBN 1–58113–436–3, S. 559–561

[WOS04] WU, Kesheng ; OTOO, Ekow J. ; SHOSHANI, Arie: An efficient compression scheme for bitmap indices. In: *Lawrence Berkeley National Laboratory* (2004)

[WOSN01] WU, Kesheng ; OTOO, Ekow J. ; SHOSHANI, Arie ; NORDBERG, Henrik: Notes on design and implementation of compressed bit vectors. In: *Lawrence Berkeley National Laboratory, Tech. Rep* (2001)

[WSO04] WU, K. ; SHOSHANI, A. ; OTOO, E.: *Word aligned bitmap compression method, data structure, and apparatus*. Dezember 14 2004. – US Patent 6,831,575

[ZR02] ZHOU, Jingren ; ROSS, Kenneth A.: Implementing Database Operations Using SIMD Instructions. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA : ACM, 2002 (SIGMOD '02). – ISBN 1–58113–497–5, S. 145–156

[ZZSY13] ZHANG, Xudong ; ZHAO, W.X. ; SHAN, Dongdong ; YAN, Hongfei: Group-Scheme: SIMD-based compression algorithms for web text data. In: *Big Data, 2013 IEEE International Conference on*, 2013, S. 525–530