Technische Universität Dresden

Beschleunigerarchitekturen zur energieeffizienten Datenbank-Anfrageverarbeitung in Mehrprozessorsystemen

Sebastian Haas

von der Fakultät Elektrotechnik und Informationstechnik
der Technischen Universität Dresden

zur Erlangung des akademischen Grades

Doktoringenieur

(Dr.-Ing.)

genehmigte Dissertation

Vorsitzender:	Prof. Dr.-Ing. habil. Christian Georg Mayr (TU Dresden)
1. Gutachter:	Prof. Dr.-Ing. Dr. h.c. Gerhard Fettweis (TU Dresden)
2. Gutachter:	Prof. Dr.-Ing. Holger Blume (Leibniz Universität Hannover)

Tag der Einreichung:	18.12.2018
Tag der Verteidigung:	18.04.2019

Danksagung

Diese Arbeit ist das Resultat meiner Tätigkeiten als wissenschaftlicher Mitarbeiter am Vodafone Stiftungslehrstuhl Mobile Nachrichtensysteme der Technischen Universität Dresden. Mein herzlicher Dank geht dabei an den Lehrstuhlleiter und meinen Doktorvater Prof. Dr.-Ing. Dr. h. c. Gerhard Fettweis, der mir die Möglichkeit zur Promotion gegeben hat. Ohne seine sachkundige Betreuung und stetige Unterstützung wäre diese Arbeit nicht möglich gewesen. Des Weiteren bedanke ich mich bei Herrn Prof. Dr.-Ing. Holger Blume für die Anfertigung des Zweitgutachtens und sein hilfreiches Feedback.

Während meiner Zeit am Lehrstuhl entstand eine großartige und intensive Zusammenarbeit mit den Kollegen und Projektpartnern, sowie vor allem in der CATS-Gruppe während der dynamischen Zeiten bei der Chipentwicklung. Ich bedanke mich bei Emil Matúš, Oliver Arnold, Benedikt Nöthen, Mattis Hasler, Robert Wittig, Friedrich Pauls, Sadia Moriam und Annett Ungethüm, sowie bei zahlreichen Kollegen vom HPSN-Lehrstuhl.

Nicht vergessen möchte ich die Studenten, deren Abschlussarbeiten ich betreuen durfte und die mir damit einen anderen Blickwinkel auf meine Arbeit eröffneten. Ein spezieller Dank geht an Heiner Bauer, Wenqian Zhai, Michael Schöbel, Anh Khoa Nyguen, Suyi Zhou, Md Mizanur Rahman, Tarjina Islam, Shaown Mojumder und Lin Ma.

Zuletzt gebührt ein herzlicher Dank meiner Familie und besonders meinen Eltern, die mich stets seelisch und moralisch unterstützt und damit indirekt zum Gelingen dieser Arbeit beigetragen haben.

Kurzfassung

Die Datenverarbeitung auf einer weltweit stetig wachsenden Informationsmenge und die hohen Anforderungen an die Energieeffizienz der Rechensysteme sind allgegenwärtige Herausforderungen der heutigen Zeit. Dabei werden zunehmend Datenbanken und deren Funktionalitäten eingesetzt, um diese großen Datenmengen effizient zu verwalten, abzuspeichern und zu verarbeiten. Auf Grund ihrer universellen Anwendbarkeit und der hohen Leistungsfähigkeit werden zumeist hoch-performante General-Purpose (GP) Prozessoren für administrative als auch für die Anfrageverarbeitung in Datenbanksystemen eingesetzt. Die Anfrageverarbeitung führt dabei eine Reihe von Operatoren wie z. B. das Suchen, Sortieren, oder Hashing aus, die signifikant die Gesamtleistung des Datenbanksystems beeinflussen. Um die weiter steigenden Anforderungen an Durchsatz, Latenz und Verlustleistung zu erfüllen, wurden bisher die Taktfrequenzen und damit die Leistungsfähigkeit von GP-Prozessoren kontinuierlich erhöht. In Zukunft werden jedoch die physikalischen Eigenschaften der verwendeten Halbleitermaterialien die Rechenleistung begrenzen.

Diese Arbeit entwickelt und analysiert Beschleunigerarchitekturen für die Datenbank-Anfrageverarbeitung, um die Leistungsfähigkeit der zugrundeliegenden Datenbankoperatoren zu steigern. Die Datenbankbeschleuniger (DBA) werden als anwendungsspezifische Hardwareblöcke (ASIC) und als Prozessor mit erweiterten Befehlssatz (ASIP) implementiert, die eine Parallelisierung der Algorithmen auf Bit-, Daten- und Befehlsebene ermöglichen. Der erste Ansatz erlaubt eine hohe Beschleunigung bei gleichzeitig niedrigem Flächen- und Leistungsverbrauch der Hardware. Im Gegensatz dazu steht beim ASIP-Ansatz bereits ein konfigurierbarer Basisprozessor zur Verfügung, der die Befehlssteuerung übernimmt und damit eine einfache Anpassung des DBAs an zahlreiche Datenbankoperatoren ermöglicht. Die vorgestellten DBAs erreichen damit die Leistungsfähigkeit von optimierten GP-Prozessoren bei einer um bis zu drei Größenordnungen höheren Energie- und Flächeneffizienz.

Für die Parallelisierung der Datenbankoperatoren auf Taskebene werden die DBAs in das Tomahawk Multiprozessorsystem auf einem Chip (MPSoC) integriert, das ein skalierbares Network-on-Chip und DMA-Controller für einen intelligenten Datentransfer bereitstellt. Eine zentrale Scheduling-Einheit arbeitet dabei den Anfrageausführungsplan ab und steuert die Zuweisung der Tasks auf die Verarbeitungseinheiten und den Transfer der Daten zu einem externen Speicher. Des Weiteren ist die Skalierung von Taktfrequenzen und Versorgungsspannungen möglich, um Durchsatz und Leistungsverbrauch an die Lastanforderungen anzupassen und damit den Energieverbrauch zu minimieren. Darüber hinaus wird das Tomahawk MPSoC mit Hilfe von Simulationen in einem virtuellen Prototyp und mit analytischen Modellen der Datenbankoperatoren hinsichtlich der Skalierbarkeit untersucht. Diese Auswertungen zeigen das Verhalten der Algorithmen bei steigender Prozessoranzahl und wachsenden Kardinalitäten sowie in Abhängigkeit der Speicherbandbreiten und relevanter algorithmusspezifischer Parameter.

Abstract

Nowadays, data processing on a worldwide continuously growing amount of information and the high demands on energy efficiency of the computing systems have become ubiquitous challenges. Hence, databases and their functionalities are increasingly deployed to manage, store, and process these large data sets. General-purpose (GP) processors provide high performances in a wide range of applications and, thus, are mostly used in database systems for administrative tasks as well as for query processing. Query processing performs on a series of operators such as searching, sorting, or hashing which have a significant impact on the overall performance of the database system. So far, continuously increasing the clock frequency of GP processors leads to higher performances to meet the ever-growing requirements on throughput, latency, and power consumption. However in the future, processing power will be limited by physical properties of the semiconductor materials.

This thesis develops and analyses accelerator architectures for database query processing to speed up the underlying operators. The database accelerators (DBA) are implemented as application-specific hardware blocks (ASIC) and as a processor including instruction-set extensions (ASIP) allowing a parallelization on bit, data, and instruction level. The first approach enables high speedups of the algorithms by achieving low area and power consumptions of the hardware. In contrast, the ASIP approach uses a configurable basic processor which manages the control flow. This allows to simply adapt the DBA to a high variety of database operators. The presented DBAs achieve the performances of GP processors while reducing the area and energy consumption by three orders of magnitude.

The DBAs will be integrated into the Tomahawk multiprocessor system-on-chip (MPSoC) to parallelize the database operators on task level. The Tomahawk MPSoC provides a scalable network-on-chip as well as DMA controllers for an intelligent data transfer. A central scheduling unit follows the query execution plan and is responsible for assigning tasks to the processing elements and to initiate the data transfer to an external memory. According to the system's load, the MPSoC allows for scaling the clock frequencies and supply voltages to adapt throughputs and power consumption to achieve the highest energy efficiency. Furthermore, the scalability of the Tomahawk MPSoC is investigated by simulations with a virtual prototype and by means of analytical models of the database operators. The evaluations show the behavior of the algorithms when changing the number of processors, cardinalities, memory bandwidths, and other relevant algorithmic-specific parameters.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

3GPP	3rd Generation Partnership Project
AAP	Anfrage-Ausführungsplan
ACM	Application Control Module
ADPLL	All-digital Phase Locked Loop
ALU	Arithmetic Logic Unit
APEX	ARC Processor Extensions
ARC	Argonaut RISC Core
ARM	Advanced RISC Machine
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-Set Processor
AVS	Adaptive Voltage Scaling
AVX	Advanced Vector Extension
BBPM	Baseband Processing Module
BMI	Bit Manipulation Instruction
CDN	Content Delivery Network
CM	CoreManager
CMOS	Complementary Metal-Oxid-Semiconductor
CodAL	Codasip Architecture Language
COMPAX	Compressed Adaptive Index
CPU	Control Processing Unit
DAG	Directed Acyclic Graph
DAX	Data Analytics Accelerator
DB	Datenbank
DBA	Database Accelerator
DBMS	Datenbankmanagementsystem
DMA	Direct Memory Access
DMAC	DMA-Controller
DMEM	Data Memory
DMS	Data Movement System
DPM	Data Processing Module
DPU	Data Processing Unit
DRAM	Dynamic Random Access Memory
DSP	Digitaler Signalprozessor
DVFS	Dynamic Voltage and Frequency Scaling
Eclat	Equivalence Class Transformation
FIFO	First in – First out
FLIX	Flexible Length Instruction Extension
FMA	Fused Multiply-Add

FPGA	Field-Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
GALS	Globally Asynchronous - Locally Synchronous
GE	Gate Equivalent
GOLD	GSM One-Chip Logic Device
GP	General-Purpose
GPIO	General-Purpose Input/Output
GPU	Graphics Processing Unit
GSM	Global System for Mobile communications
GUI	Graphical User Interface
HPM	Hardware Performance Monitor
iDMA	Intelligenter DMA-Controller
ILP	Instruction-Level Parallelism
IMEM	Instruction Memory
IoT	Internet of Things
ISA	Instruktionssatzarchitektur
ISE	Instruktionssatzerweiterung
LF	Lastfaktor
LISA	Language for Instruction-Set Architecture
LPDDR	Low-Power Double Data Rate
LSU	Load/Store-Unit
MIMD	Multiple Instructions, Multiple Data
MIPS	Microprocessor without Interlocked Pipeline Stages
MM	Memory Manager
MMU	Memory Management Unit
MMX	Multimedia Extension
MPSoC	Multiprocessor System-on-Chip
NoC	Network-on-Chip
OLAP	Online Analytical Processing
PE	Processing Element
PEXT	Parallel Extraction of Bits
PLWAH	Position List Word-Aligned Hybrid
PM	Processing Module
PMC	Power Management Controller
PPE	PowerPC Processing Element
RAPL	Running Average Power Limit
RDBMS	Relationales Datenbankmanagementsystem
RID	Row-Identifier
RISC	Reduced Instruction-Set Computer
RLE	Run-Length-Encoding
RTL	Register Transfer Level
SIMD	Single Instruction - Multiple Data
SPE	Synergistic Processing Element
SoC	System-on-Chip

SQL	Structured Query Language
SRAM	Static Random Access Memory
SSE	Streaming SIMD Extension
TDP	Thermal Design Power
TIE	Tensilica Instruction Extension
TLB	Translation Lookaside Buffer
TPC	Transaction Processing Performance Council
UART	Universal Asynchronous Receiver Transmitter
VLIW	Very Large Instruction Word
VLSI	Very Large Scale Integration
WAH	Word-Aligned Hybrid
WUR	Write User Register

Symbolverzeichnis

$\|R\|$	Kardinalität der Liste/Relation R
α	Verhältnis aus Datenverarbeitungs- und Transferzeit
β	Verhältnis aus sequentiellem und parallelem Anteil der Datenverarbeitungszeit
A	Chipfläche
B_{DRAM}	Bandbreite des globalen Speichers
B_{NoC}	Bandbreite eines Links des Network-on-Chips
D_{Build}	Durchsatz der Build-Phase beim Hash-Join
D_{Hash}	Durchsatz des Hashings
D_{Hist}	Durchsatz der Berechnung von Histogramm und Präfixsumme beim Hash-Join
D_{Probe}	Durchsatz der Probing-Phase beim Hash-Join
D_{Set}	Durchsatz der Sorted-Set Algorithmen
D_{MS}	Durchsatz des Merge-Sorts
$D_{MS,presort}$	Durchsatz des Merge-Sorts bei Vorsortierung
D_{HJ}	Durchsatz des Hash-Joins
f	Taktfrequenz
f_{max}	Maximale Taktfrequenz
h	Hashwert
k	(SIMD-)Parallelitätsfaktor
K	Optimierungskosten
m	Anzahl Buckets
n_{Block}	Anzahl Datenblöcke
n_{Task}	Anzahl Tasks
N	Anzahl Elemente einer Menge
p	Anzahl Prozessorelemente
P	Verlustleistung
r_{DB}	Laufzeitverhältnis aus gleichzeitig auftretender Datenverarbeitung und Transfer
r_{LF}	Lastfaktor
s_{Block}	Größe eines Datenblocks
s_{Mem}	Größe eines Speichers
sel	Selektivität
S	Speedup
t_{Ges}	Gesamtausführungszeit
t_{Join}	Zeit zur Ausführung des Join-Operators
t_{Par}	Paralleler Anteil der Datenverarbeitungszeit
t_{Proc}	Datenverarbeitungszeit

t_{Sched}	Laufzeit des Schedulers
t_{Seq}	Sequentieller Anteil der Datenverarbeitungszeit
t_{Takt}	Zeit in Taktzyklen
t_{Task}	Zeit eines Tasks
t_{Trf}	Datentransferzeit
t_{Opt}	Optimierungszeit
U_{DD}	Versorgungsspannung
W_{Elem}	Datenbreite eines Elements
W_{Mem}	Datenbreite eines Speichers

1 Einleitung

1.1 Motivation

Der Fortschritt von Prozessorsystemen wird auf Grund einer stetig wachsenden Datenmenge, höheren Anforderungen hinsichtlich Datendurchsatz, Latenz und Energieeffizienz vorangetrieben. Die Anwendungsbereiche sind dabei vielfältig: Car-to-Car Kommunikation, virtuelle Realität und hochauflösendes Video- und Audiostreaming benötigen zunehmend den Datenaustausch über drahtlose Netzwerke. Zum einen liegen dabei die Herausforderungen bei einer energieeffizienten und flexiblen Datenverarbeitung wie sie z. B. bei der Auswertung von Sensoren von Geräten des Internet der Dinge (IoT), Audio- und Videocodierung sowie bei Internet-Aktivitäten benötigt werden. In gleichem Maße muss eine leistungsstarke Umsetzung der zur drahtlosen Übertragung zugrundeliegenden Signalverarbeitungsalgorithmen erfolgen.

Zellulare Mobilfunksysteme verzeichnen ein weltweites Wachstum des Datenverkehrs um durchschnittlich 46 % pro Jahr [150]. Mit der fünften Generation des zellularen Mobilfunks 5G werden deshalb Datenraten im GBit-Bereich bei Latenzen unter 5 ms angestrebt [115]. Dazu bedarf es einer dezentral aufgebauten Netzwerkarchitektur, in der sich die Rechensysteme örtlich näher am Anwender befinden. Dieser als *Mobile Edge Computing* (MEC, [77]) bekannte Ansatz hat sich bereits als Vorabversion in 4G Technologien als ein mobiles *Content Delivery Network* (CDN) [146] etabliert und wird Hauptbestandteil der fünften Generation sein. MEC ermöglicht verkürzte Signalwege auf Grund der verkürzten örtlichen Distanzen. Es erlaubt aber auch das Caching, die Synchronisierung und Vorverarbeitung von Daten, um den Datenaustausch zum Hauptserver zu reduzieren. Auch die auf Grund unterschiedlichster Anwendungen erzeugten unstrukturierten Daten können so zuvor aggregiert und gefiltert werden. Aktuelle Mobilfunksysteme sind weitestgehend auf die Spitzenauslastung dimensioniert, obwohl die Rechenleistung als auch die Leistungsaufnahme innerhalb von Stunden um mehrere Größenordnungen schwanken [103]. MEC ermöglicht damit eine dynamische Anpassung der Last, ohne die Beeinflussung der Gesamtleistung jedoch mit einer höheren Energieeffizienz. Dazu bedarf es auch einer Anpassung der zugrunde liegenden Hardware, um eine flexible Anpassung von Taktfrequenz und Versorgungsspannung zu erreichen.

Diese Arbeit beschäftigt sich deshalb mit der Entwicklung von energieeffizienten Beschleunigerarchitekturen und Prozessorelementen. Der Fokus liegt dabei auf der Vorverarbeitung von Daten und der Umsetzung rechenintensiver Operationen. Dabei wird das Modell der Anfrageverarbeitung aus Datenbanksystemen zu Grunde gelegt, um auf deren verwendeten Datenstrukturen und Operatoren aufzubauen.

1.2 Einordnung von Datenbanksystemen

Das weltweite Datenaufkommen von 2017 wird auf etwa 20 ZB geschätzt und verdoppelt sich alle zwei Jahre [51]. Datenbanksysteme sind heutzutage ein essentieller Bestandteil, um diese große Datenmengen effizient zu verwalten, abzuspeichern und zu verarbeiten. Die Verwaltungssoftware des Datenbanksystems ist das Datenbankmanagementsystem (DBMS), das neben den Aufgaben der Administration, Datensicherheit und Integrität auch die Anfrageverarbeitung steuert [43]. Die Anfrageverarbeitung beeinflusst dabei signifikant die Gesamtleistung des DBMS. Anfragen werden von Benutzern an das DBMS gestellt, um Informationen aus der Datenbank zu erhalten.

Für die Umsetzung des zuvor angesprochenen MEC-Ansatzes ist auch die Integration solcher DBMS-bezogenen Funktionen in drahtlose Kommunikationssysteme notwendig [41, 81]. Mit diesem Hintergrund ist es möglich, ein zellulares Mobilfunksystem als ein verteiltes Datenbanksystem zu abstrahieren, in dem jede oder jede Gruppe von Mobilfunkzellen ein vereinfachtes DBMS integriert [164]. Damit entstehen neue Herausforderungen wie z. B. standortbezogene Anfragen, die auch sich dynamisch ändernde Echtzeitdaten einbeziehen müssen [15, 80].

Das DBMS verwendet standardmäßig hoch-performante General-Purpose Mehrkernprozessoren für administrative Aufgaben als auch für die Anfrageausführung. Um jedoch die Anforderungen an Anfragedurchsatz und -latenz zu erfüllen und gleichzeitig eine hohe Energieeffizienz zu erzielen, bedarf es neuartiger Prozessorsysteme. In dieser Arbeit werden deshalb Beschleunigerarchitekturen zur Datenbank-Anfrageverarbeitung untersucht. Diese *Datenbankbeschleuniger* (Database Accelerator, DBA) sind auf grundlegende Datenbankalgorithmen spezialisiert und stellen damit den erforderlichen Datendurchsatz bei einer vergleichsweise um mehrere Größenordnungen niedrigeren Leistungsaufnahme bereit. Die vorgeschlagene Grundidee ist dabei die Integration des DBA als Coprozessor neben den General-Purpose CPUs im DBMS. Der Coprozessor führt die ausgelagerten Datenbankoperationen mit höchster Energieeffizienz aus und entlastet zusätzlich die CPU des DBMS, die damit Rechenzeit für andere administrative Aufgaben erhält.

1.3 Entwicklung moderner Prozessorsysteme

Innerhalb der letzten Jahrzehnte hat sich die Single-Thread Prozessorleistung pro Jahr um etwa 50 % gesteigert [71], sodass eine Anpassung an die wachsenden Anforderungen garantiert war. Das ist vor allem auf die immer kleiner werdenden Herstellungstechnologien der Transistoren zurückzuführen, die eine stetige Erhöhung der Taktfrequenz und eine Verringerung der Versorgungsspannung ermöglichten. So wird prognostiziert, dass bis zum Jahr 2033 Transistoren in einer 1 nm Technologie gefertigt werden [79]. Diese von Dennard u. a. [37] charakterisierte Skalierung von Taktfrequenz und Versorgungsspannung verliert im Laufe der Zeit jedoch ihren Vorteil, da sich mit kleineren Strukturbreiten bei gleichbleibender Chipgröße und größerer Transistorzahl die statische Leistungsaufnahme (Leckstrom) erhöht. Eine weitere Erhöhung der Taktfrequenz hätte eine zu große thermische Verlustleistung zur Folge, die den Chip irreparabel schädigen würde. Die Auswirkungen sind an der durchschnittlichen Steigerung der Rechenleistung von einzelnen

Prozessorkernen zu erkennen, die seit Anfang der 2000er auf etwa 20 % pro Jahr gesunken ist.

Um weiterhin die Leistungsfähigkeit der Rechensysteme zu steigern, werden immer mehr einzelne und verschiedenartige Kerne zusammen mit Speicher-, Peripherie- und I/O-Komponenten auf einem Chip integriert (Multiprocessor System-on-Chip, MPSoC) [45]. Die Verarbeitungseinheiten enthalten dann applikationsspezifische Hardware, die es ermöglichen, mit vergleichsweise geringen Taktfrequenzen den notwendigen Datendurchsatz zu erzielen. Niedrige Taktfrequenzen und Versorgungsspannungen sowie geringe Chipgrößen erlauben damit die Reduzierung der Verlustleistung.

Der in dieser Arbeit verwendete Tomahawk MPSoC integriert die angesprochenen Datenbankbeschleuniger, die mit Hilfe ihrer anwendungsspezifischer Hardware eine energieeffiziente Abarbeitung von Datenbankanfragen erlauben. Tomahawk ist als heterogene Hardwareplattform in zahlreichen Forschungsprojekten eingebunden. Zum Beispiel forscht das Exzellenzcluster „Center for Advancing Electronics Dresden“ (cfaed) der TU Dresden an neuen zur CMOS-Technologie komplementären Materialien (z. B. Silicon Nanowires, Carbon Nanotubes). Das Ziel ist, wild heterogene Systeme aufzubauen, die eine energieeffiziente, adaptierbare und robuste Datenverarbeitung bereitstellen. Als CMOS-Basisplattform dient das Tomahawk MPSoC, das die Untersuchung der Leistungsfähigkeit zahlreicher Anwendungsbereiche wie z. B. der Datenbank-Anfrageverarbeitung erlaubt [30, 151].

1.4 Beiträge und Gliederung dieser Arbeit

Diese Arbeit beschäftigt sich mit der Entwicklung und Analyse von Beschleunigerarchitekturen für die Datenbank-Anfrageverarbeitung, um die Leistungsfähigkeit der zugrundeliegenden Datenbankoperatoren zu steigern. Dazu werden unterschiedliche Algorithmen auf ihre Komplexität und Parallelisierbarkeit analysiert und mit anwendungsspezifischer Hardware implementiert. Eine Beschleunigung der Algorithmen wird dabei durch vier Ansätze ermöglicht, d. h. durch eine Parallelität auf

- Bit- bzw. arithmetisch-logischer Ebene (Fusion),
- Datenebene (SIMD),
- Befehlsebene (ILP) und
- Task- bzw. Systemebene (MIMD).

Die Parallelität auf Bit- bzw. arithmetisch-logischer Ebene ist eine Form der Parallelisierung auf niedrigstem Level. Es werden mehrere verschiedene bitweise oder elementare arithmetisch-logische Operationen gleichzeitig auf unterschiedlichen Daten ausgeführt. Diese werden zu einer einzigen Instruktion verschmolzen (Fusion). Im Vergleich zur sequentiellen Abarbeitung der einzelnen Operationen ermöglicht die Ausführung von nur einem kombinierten Befehl eine Reduzierung der Gesamtausführungszeit. Erweitert man nun Befehle auf die Verarbeitung von Vektordaten, erhält man eine Parallelität auf Datenebene. Diese parallele Ausführung einer Instruktion mit verschiedenartigen Daten wird nach der Taxonomie nach Flynn [46] als Single Instruction, Multiple Data (SIMD) bezeichnet. Ist z. B. auf Grund der Anforderungen des Algorithmus oder der Datenanordnung

im Speicher keine Datenparallelität möglich, können mehrere unabhängige Instruktionen gleichzeitig ausgeführt werden (Instruction-Level Parallelism, ILP). Dazu sind mehrere parallele funktionale Einheiten notwendig, die die Instruktionen aus dem Speicher lesen und anschließend dekodieren. Hier kann auch die Parallelität durch Pipelining eingeordnet werden, bei der Registerstufen in den Datenfluss sequentiell abzuarbeitender Operationen oder Instruktionen eingeführt werden. Eine Reduzierung der Ausführungszeit ist dann durch eine nebenläufige Ausführung der einzelnen Stufen möglich. Mit mehreren parallelen Verarbeitungseinheiten, die voneinander unabhängige Tasks ausführen, kann eine Beschleunigung auf Systemebene erzielt werden (Multiple Instructions, Multiple Data, MIMD). Im Vergleich zur Datenparallelität ermöglicht die Taskparallelität eine höhere Flexibilität, da den Tasks allgemeine Aufgaben zugewiesen werden können, während SIMD-Instruktionen auf bestimmte Operationen spezialisiert sind. Die dargestellte Unterteilung beschreibt ein weitestgehend ideales Modell für Rechnerarchitekturen. Typischerweise sind wie auch in dieser Arbeit immer Kombinationen der vier Ansätze realisierbar, um die Leistungsfähigkeit der Algorithmen zu steigern.

Die verschiedenen Parallelisierungsmethoden werden in den einzelnen Kapiteln dieser Arbeit angewendet und deren Einfluss auf die Beschleunigung der Datenbank-Anfrageverarbeitung analysiert. Das Ziel ist, die Leistungsfähigkeit der Algorithmen zu steigern und dabei die Hardwarekosten zu minimieren. Die Vor- und Nachteile der Ansätze können durch eine geeignete Wahl der Parameter wie z. B. Taktfrequenz, Registerbreite und Speicherbandbreite charakterisiert werden. Zusätzlich sind weitere Nebenbedingungen wie die Verteilung der Daten im Speicher oder die Komplexität der Algorithmen zu berücksichtigen, da sie ebenfalls die Leistungsfähigkeit der Datenbankoperatoren beeinflussen. Die nachfolgenden Paragraphen stellen kurz die Inhalte der Kapitel vor.

Zu Beginn erfolgt in Kapitel 2 eine detaillierte thematische Einordnung dieser Arbeit mit einem Überblick über verwandte Arbeiten sowie Erläuterungen von Grundbegriffen. Die einzelnen Beiträge sind:

- Einordnung der Datenbank-Anfrageverarbeitung mit Diskussion über eine potentielle Integration des vorgeschlagenen Datenbankbeschleunigers (DBAs) in ein Datenbanksystem
- Überblick über den aktuellen Stand der Technik zur Ausführung von Datenbankoperatoren
- Erläuterungen der zum Grundverständnis notwendigen ausgewählten Datenbankoperatoren und Grundbegriffe

Kapitel 3 präsentiert die Implementierung des Datenbankbeschleunigers als anwendungsspezifischen integrierten Schaltkreis (DBA ASIC). Der ASIC enthält dedizierte Hardwareblöcke, die an die Datenbankoperatoren angepasst sind. Die Anpassung erfolgt durch die parallele Datenverarbeitung als auch mit einer festgelegten Ablaufsteuerung. Konkrete Beiträge sind:

- Diskussion der Implementierungs- und Parallelisierungsaspekte von anwendungsspezifischen Hardwareblöcken

- Konzept, Entwicklung und Vorstellung der Hardwareblöcke für Implementierung von Sorted-Set Algorithmen
- Evaluierung der Hardwareblöcke hinsichtlich Durchsatz, Flächen- und Leistungsverbrauch sowie Vergleich mit weiteren Rechnerarchitekturen

Kapitel 4 stellt die Implementierung des Datenbankbeschleunigers als Prozessor mit anwendungsspezifischem Instruktionssatz vor (DBA ASIP). Die Steigerung der Leistungsfähigkeit wird durch eine Anpassung der Prozessorarchitektur als auch durch die Erweiterung des Befehlssatzes erreicht. Im Detail umfasst das Kapitel die folgenden Beiträge:

- Vorstellung eines Systemmodells für erweiterbare Prozessoren und Diskussion von Techniken zur Anpassung von Architektur und Befehlssatz
- Entwicklung neuartiger Befehlssatzerweiterungen für die energieeffiziente Ausführung von Sorted-Set Algorithmen, Sortieren und Hashing
- Evaluierung der Befehlssatzerweiterungen hinsichtlich Durchsatz, zusätzlichen Flächen- und Leistungsverbrauchs, Einfluss des erweiterten Befehlssatzes und der Speicherbandbreite sowie Vergleich mit weiteren Rechnerarchitekturen
- Design des DBA ASIP in einem Forschungschip und Präsentation der Messergebnisse
- Untersuchungen zur Leistungsfähigkeit der Datenbankbeschleuniger für die Anfrageoptimierung am Beispiel der Selektivitätsabschätzung

Kapitel 5 zeigt die Integration des DBA ASIP in das Tomahawk MPSoC. Die in den Verarbeitungseinheiten integrierten DBAs sind über ein Network-on-Chip miteinander verbunden und erhalten mittels eines DMA-Controllers Zugriff auf die Daten in einem externen Speicher. Die Beiträge des Kapitels sind:

- Vorstellung der Tomahawk-Architektur sowie deren Umsetzung als Forschungschips
- Implementierung der Datenbankoperatoren für die Ausführung auf mehreren Prozessoren des Tomahawk
- Präsentation der Messergebnisse zu Durchsatz und Leistungsaufnahme sowie Vergleich mit anderen Rechnerarchitekturen

In Kapitel 6 werden Untersuchungen zur Skalierbarkeit der vorgeschlagenen parallelen Implementierungen auf dem Tomahawk MPSoC vorgestellt. Diese beinhalten im Einzelnen:

- Simulation der parallelen Implementierungen der Datenbankoperatoren in einer virtuellen Multiprozessor-Plattform
- Aufstellen mathematischer Modelle der Algorithmen zur Betrachtung der Skalierbarkeit von Speedup und Ausführungszeit

- Vergleich der Resultate aus der Simulation und den mathematischen Modellen mit den Messergebnissen des Tomahawk

Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick auf weiterführende Arbeiten in Kapitel 7.

2 Datenbanken: Grundlagen und verwandte Arbeiten

Dieses Kapitel beschreibt die erforderlichen Grundlagen zum Verständnis des in dieser Arbeit verwendeten Datenbankmodells und die Algorithmen zur Anfrageverarbeitung. Zusätzlich wird ein erster allgemeiner Überblick über aktuelle Forschungsarbeiten von Datenbankbeschleunigern vorgestellt.

2.1 Einführung

2.1.1 Datenbankmodell

Jede Datenbank ist durch ein Datenbankmodell spezifiziert, das die Struktur der Daten beschreibt. Dazu zählen Datentypen, Integritätsbedingungen, sowie Operationen, die auf den Datenstrukturen angewendet werden können. Die Verwaltungssoftware einer Datenbank, das Datenbankmanagementsystem (DBMS), basiert direkt auf dem verwendeten Datenbankmodell [122]. Das am weitesten verbreitete DBMS ist das relationale Datenbankmodell (Relationales Datenbankmanagementsystem, RDBMS). Die Systeme unterstützen dabei die Anfragesprache *Structured Query Language* (SQL). In einem RDBMS befinden sich die Daten in Tabellen, sogenannten *Relationen*, wobei die Spalten als *Attribute* und die Zeilen als *Tupel* bezeichnet werden [43]. Der *Grad* und die *Kardinalität* beschreiben die Anzahl der Attribute bzw. die Anzahl der Tupel. Zulässig sind nur atomare Werte in den Relationen, d. h. Zahlen, Zeichenketten und andere nicht strukturierte Wertebereiche. Weiterhin sind einzelne Attribute verschiedener Relationen miteinander verknüpft, um Beziehungen zwischen den Tabellen aufzubauen.

Weitere nicht-relationale DBMS, sogenannte *Not-only-SQL* (NoSQL) Modelle, sind z. B. objekt-orientierte Datenbanksysteme, bei denen die Tupel durch Objekte definiert werden. Mehrere Objekte stellen eine Klasse dar, wobei die Klassen dann in Hierarchien organisiert sind. Im Gegensatz zum RDBMS, können objekt-orientierte Programmiersprachen und komplexere Datentypen verwendet werden. Oft werden relationale Systeme erweitert, um zusätzlich objekt-orientierte Datenbanksysteme zu unterstützen. Weitere Beispiele für NoSQL-Datenbanken sind hierarchische und Netzwerkdatenbanken, die die Daten in Baumstrukturen bzw. als Graphen abspeichern. Diese Datenstrukturen erlauben im Vergleich zu den Relationen eine einfachere Verknüpfung von Daten. Damit kann die Leistungsfähigkeit von Anfragen verbessert werden, die häufig Anfragen nach Beziehungen zwischen den Daten verarbeiten (z. B. Joins).

Graphdatenbanken leiten sich aus den Netzwerkdatenbanken ab, arbeiten aber auf einem höheren Abstraktionsniveau [4]. Während Netzwerkdatenbanken Zeiger und Werte

auf physischer Ebene nutzen, speichern und verarbeiten Graphdatenbanken die Knoten und Kanten und folgen dabei den logischen Verknüpfungen des Graphen.

Diese Arbeit stützt sich auf ein relationales Datenbankmodell, das am weitesten verbreitet ist und in den marktführenden Datenbanksystemen von Oracle und Microsoft eingesetzt wird [35]. Weitere Beispiele für populäre kommerzielle RDBMS sind IBM DB2 [78] sowie Open-Source Systeme wie MySQL [113] oder PostgreSQL [119]. In einem RDBMS wird jedes Tupel durch sogenannte *Schlüssel* (auch als Key, Zeilenkennung oder Row-Identifier (RID) bezeichnet) eindeutig identifiziert, die ein oder mehrere Attribute der Relation darstellen. In dieser Arbeit werden Algorithmen betrachtet, die ausschließlich auf Schlüsseln (z. B. Sorted-Set Algorithmen) als auch auf mehreren Attributen (z. B. Hash-Join) arbeiten. Für Letzteres kann der Sonderfall auftreten, dass eine Relation genau zwei Attribute besitzt. Die Tupel werden dann als *Schlüssel-Wert-Paare* (*Key-Payload Pair*) bezeichnet.

Datenbankanfragen dienen grundsätzlich zum Filtern von relevanten Daten aus der Gesamtinformationsmenge der Datenbank und bestehen aus einer Abfolge mehrerer elementarer Operatoren wie z. B. Sortieren, Suchen oder Scannen. Diese Grundoperationen werden aus den in SQL beschriebenen komplexen Anfragen extrahiert. Die Ausführung der Anfrage basiert auf der Abarbeitung dieser Operatoren, wobei in der Anfrage nur das Ergebnis genau spezifiziert ist. Die eigentliche Systemarchitektur und Programmierschnittstelle der Ausführungseinheiten ist nicht vorgegeben. Der in dieser Arbeit entwickelte Datenbankbeschleuniger benötigt deshalb keine Unterstützung der Hochsprache SQL, sondern verarbeitet die Algorithmen mit der Programmiersprache C. Lediglich ist die Anordnung der Daten im physischen Speicher vorzugeben. In dieser Arbeit wird ein nach Spalten geordnetes System vorausgesetzt, d. h. die Werte eines Attributs der Relation liegen an aufeinanderfolgenden Speicheradressen. Dies erlaubt den gleichzeitigen Zugriff auf mehrere Elemente einer Spalte z. B. mittels SIMD-Operationen.

2.1.2 Stand der Technik

Die Beschleunigung der Datenbank-Anfrageverarbeitung hat ihre Anfänge in den 1980er Jahren. Erste sogenannte Datenbankmaschinen hatten zusätzliche Verarbeitungslogik direkt in den Magnetköpfen der Festplatten integriert, um z. B. bestimmte Tupel beim Lesen der Daten zu selektieren oder zu suchen [25]. Später wurden Systeme mit mehreren Verarbeitungseinheiten entwickelt, die über ein Verbindungsnetzwerk verbunden sind. Die Einheiten enthalten jeweils einen Prozessor, Cache und Festplattenspeicher und sind oft in einer *Shared-Nothing Architektur* integriert, d. h. sie nutzen keinen gemeinsamen Speicher. Beispiele solcher Datenbankmaschinen sind *Gamma* [39], *GRACE* [94] oder *SABRE* [147]. Die Prozessoren sind mit spezieller Hardware ausgestattet, um die Ausführung von Datenbankoperationen wie Join, Aggregation oder Sortieren zu beschleunigen [11, 93, 106, 129].

Die ersten Entwicklungen von Datenbankbeschleunigern zur Anfrageverarbeitung haben bereits das Potential anwendungsspezifischer Hardware für Datenbanksysteme nachgewiesen. Die Forschungen in diesem Bereich werden deshalb heutzutage noch aktiv weiter verfolgt. Die Arten der Hardwarebeschleuniger lassen sich dabei in zwei Gruppen einteilen. Die erste und zunächst triviale Methode ist die Entwicklung von Hardwareblöcken, die ausgewählte Datenbankoperatoren beschleunigen und als ASIC oder auf einem FPGA (Field-

Programmable Gate Array) implementiert werden [29, 34, 108, 132, 155, 160, 163]. Auch diese Arbeit untersucht die Leistungsfähigkeit ausgewählter Operatoren mit Hilfe datenbankspezifischer Hardwareblöcke, die in einem ASIC integriert sind. Die Implementierungen und Auswertungen sowie weitere verwandte Arbeiten werden dazu in Kapitel 3 vorgestellt.

Im Gegensatz zu FPGAs ist die ASIC-Funktionalität nach der Chipfertigung nicht mehr veränderbar. Allerdings liegt der Nachteil des FPGA in einer größeren Chipfläche, einer höheren Verlustleistung und niedrigeren Taktfrequenzen. Kuon u. a. [97] berichten, dass ein FPGA im Durchschnitt 40× größer ist, 12× mehr dynamische Leistung aufnimmt und etwa 3× langsamer arbeitet als der ASIC. Zudem bietet die Rekonfigurierbarkeit der FPGAs keinen entscheidenden Vorteil, da Datenbankoperatoren im Vorfeld eindeutig für das jeweilige System definiert werden können.

Ein weiterer Ansatz zur Beschleunigung von Datenbankanfragen sind anwendungsspezifische Befehlssatzerweiterungen. Zahlreiche Forschungsarbeiten nutzen dabei General-Purpose (GP)-Prozessoren und deren bereits vorkonfigurierte Instruktionen, wie z. B. die Intel *Streaming SIMD Extensions* (SSE), um Basisoperatoren wie die Sorted-Set Intersektion oder Joins zu beschleunigen [130, 167]. Andere Arbeiten hingegen erweitern die Hardware von x86-Prozessoren mit Vektorinstruktionen [66], oder entwickeln datenbankspezifische Hardwaremodule, die über zusätzliche Instruktionen in der Software eines GP-Prozessors angesprochen werden [95, 159]. Gegenüber der optimierten Softwareimplementierung kann mit Hilfe dieser Befehlssatzerweiterungen die Ausführungszeit um Faktor 3,1 bis 7,8 und gleichzeitig der Energieverbrauch um durchschnittlich 85 % reduziert werden. Diese Arbeit untersucht ebenfalls das Potential von erweiterten Instruktionen, um die Leitungsfähigkeit und Energieeffizienz ausgewählter Datenbankoperatoren zu verbessern und gleichzeitig eine hohe Flexibilität zu erreichen. Im Unterschied zu den existierenden Lösungen, geht diese Arbeit noch einen Schritt weiter. Neben der Entwicklung neuer Instruktionen wird auch die Architektur des zugrundeliegenden Prozessors an die Algorithmen angepasst. Es entsteht so ein Datenbankbeschleuniger (DBA) als vollständiger Prozessor mit anwendungsspezifischen Instruktionssatz (ASIP). Kapitel 4.1.3 gibt dazu einen weiteren Einblick in die verwandten Arbeiten mit Befehlssatzerweiterungen.

Der Einsatz von Befehlssatzerweiterungen spezialisiert den zugrundeliegenden Prozessor auf die Ausführung ausgewählter Datenbankoperatoren. In modernen kommerziellen Datenbanksystemen müssen GP-Prozessoren jedoch die breite Funktionalität des DBMS unterstützen. Eine Lösung ist, das bestehende System mit eigenständigen hardwarebeschleunigten Coprozessoren zu erweitern, um rechenintensive Datenbankoperationen auszulagern. Zum Beispiel integriert IBM Netezza [47] sogenannte *Snippet-Blades* in den Datenpfad zwischen Festplatte und Host-Prozessor, um Daten bereits beim Laden vorzuverarbeiten. Die Snippet-Blades enthalten jeweils einen FPGA, Mehrkern-CPUs sowie zusätzlichen Speicher und dienen damit zum Vorhalten der Daten, Filtern und zur Kompression. Ein weiteres Beispiel ist der Oracle Sparc Prozessor [100], der anwendungsspezifische Hardware neben hochperformanten CPUs auf einem gemeinsamen System-on-Chip integriert. Der *Sparc M8* [114] enthält 32 Prozessoren mit insgesamt 256 Threads, die in acht Vierkern-Cluster mit jeweils einem gemeinsamen 64 MB L3 Cache aufgeteilt sind. 32 sogenannte *Data Analytics Accelerators* (DAX) dienen zur Anfrageverarbeitung und

ermöglichen damit das Auslagern von Filteroperationen, Dekompression und Vergleichsoperatoren.

Ein weiterer Forschungsbereich zur Beschleunigung von Datenbankanfragen fokussiert sich auf den Einsatz von Grafikprozessoren (Graphics Processing Unit, GPU). Die massive Thread-Parallelität von GPUs wird ausgenutzt, um die Leistungsfähigkeit grundlegender Datenbankoperatoren wie Join [89, 101, 162], die Intersektion [5, 156], Sortieren [128] oder die Indizierung mittels Bitmaps [50] zu steigern. Gegenüber GP-CPUs erreichen die GPU-Implementierungen einen bis zu Faktor 10 höheren Durchsatz. Breß u. a. [27] geben einen Überblick über GPU-beschleunigte Datenbanksysteme und diskutieren deren Herausforderungen bei der Integration von GPUs in existierende DBMS. Auch für den in dieser Arbeit entwickelten Datenbankbeschleuniger entstehen Herausforderungen bei einer potentiellen Integration in ein DBMS, die in Abschnitt 2.2.1 analysiert werden.

Einen grundlegend anderen Ansatz zur effizienten Verarbeitung von Datenbankanfragen verfolgen die Autoren in [67]. Sie präsentieren einen Entwurf von *ApproxiDB*, ein hybrides DBMS, das neben der konventionellen Hardware auch *Approximate Hardware* enthält. Diese ungenauen Berechnungen profitieren von einer höheren Leistungsfähigkeit und einem niedrigeren Energieverbrauch. Die Arbeit in [32] verwendet Sortieralgorithmen wie Quick-Sort und Merge-Sort und führt diese auf approximativem Speicher aus. Um trotzdem fehlerfreie Ergebnisse zu erhalten, findet eine nachträgliche Korrektur der Daten statt. Die Latenz der Speicherzugriffe kann im konkreten Fall bis zu 11 % reduziert werden.

2.2 Anfrageverarbeitung

Anfragen werden von Benutzern an das Datenbanksystem gestellt, um Information aus der Datenbank zu erhalten bzw. abzufragen. Um relevante Daten effizient aus der großen Gesamtinformationsmenge der Datenbank zu filtern und zu extrahieren, durchläuft die DBMS-Software eine komplexe Prozedur, die in Abschnitt 2.2.1 beschrieben wird. Zu den einzelnen Schritten dieser Anfrageverarbeitung gehören das Parsen und Übersetzen der Anfrage, sowie die Anfrageoptimierung und Ausführung. Danach werden in Abschnitt 2.2.2 die für diese Arbeit wichtigen Grundbegriffe zu Datenbanken erläutert. Wie zuvor beschrieben, umfasst die Anfrageausführung die Abarbeitung einzelner Datenbankoperatoren. Abschnitt 2.2.3 führt dazu die in dieser Arbeit untersuchten Operatoren ein und motiviert deren Relevanz in der Anfrageverarbeitung.

2.2.1 Ablauf

Der Zugang der Benutzer zum Datenbanksystem erfolgt typischerweise über SQL-Anfragen, die im DBMS verarbeitet werden. Die DBMS-Software läuft dabei auf hochperformanten CPUs, die die Anfragen bearbeiten und ausführen. Die Abbildung 2.1 zeigt solch ein Datenbanksystem, das dem Stand der Technik entspricht sowie dessen potentielle Erweiterungen, um den in dieser Arbeit vorgestellten DBA zu integrieren (graue Komponenten). Der DBA agiert als Coprozessor, um Datenbankoperatoren zu beschleunigen und damit die CPU des DBMS von rechenintensiven Operationen zu entlasten. Das System

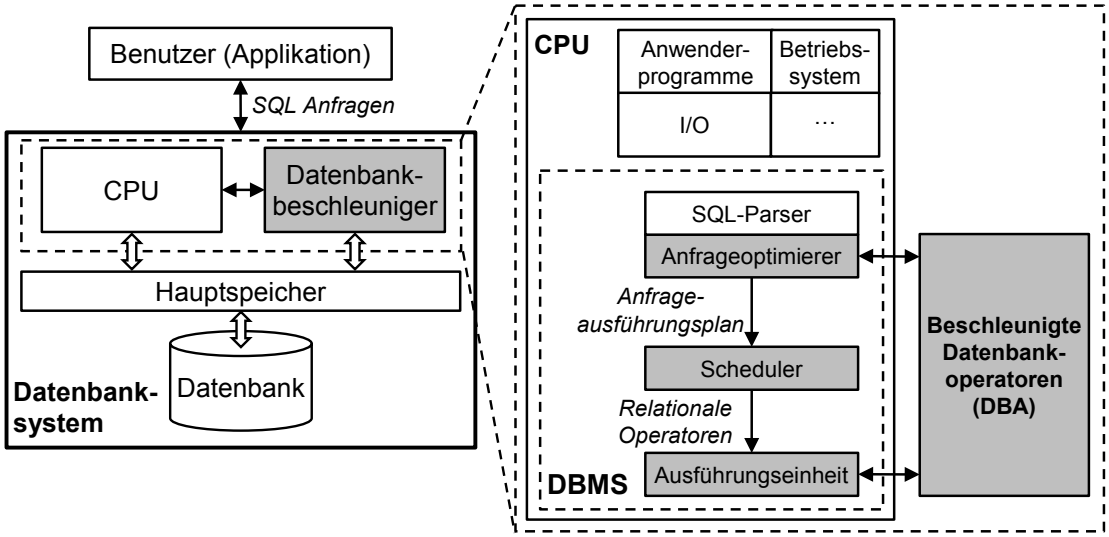


Abbildung 2.1: Übersicht eines Coprozessor-beschleunigten Datenbankensystems. Die grau markierten Komponenten kennzeichnen die für diese Arbeit erforderlichen Erweiterungen eines modernen DBMS, um den Datenbankbeschleuniger vollständig integrieren zu können.

enthält zur besseren Überschaubarkeit nur eine CPU, könnte aber auf mehrere CPUs erweitert werden, die in einem shared oder distributed Speichersystem verbunden sind [122]. Jedoch sollten die CPU und der DBA wie dargestellt einen gemeinsamen Hauptspeicher nutzen, da in einem verteilten Speicher zusätzliche Datentransfers notwendig wären, die die Leistungsfähigkeit einschränken. Der DBA kann wie in dieser Arbeit vorgeschlagen als ASIC, ASIP oder als Mehrkernsystem eingesetzt werden.

Die folgenden Schritte beschreiben den allgemeinen Ablauf der Anfrageverarbeitung [127], wobei jeweils die notwendigen Anpassungen für die Integration des DBA in das Coprozessor-beschleunigte Datenbanksystem erläutert werden.

1. Gültigkeitsprüfung: Die Anfrage wird zunächst geparst und syntaktisch auf Korrektheit geprüft. Dabei werden die Existenz der referenzierten Tabellen sowie die Zugriffsrechte kontrolliert. Es entsteht ein Parse-Baum oder Anfrage-Graph, der SQL-Syntaxstrukturen enthält.
2. Übersetzung und Vereinfachung: Im nächsten Schritt wird der Parse-Baum in einen noch unoptimierten Anfrageausführungsplan (AAP) überführt, der aus Operationen der relationalen Algebra besteht. Eine nachfolgende Vereinfachung des Anfrageplans erfolgt z. B. durch das Entfernen redundanter Ausdrücke oder das Auflösen verschachtelter Operationen. Dieser und der vorhergehende Schritt sind in Abbildung 2.1 im SQL-Parser zusammengefasst, der noch keine Anpassung in Bezug auf die Integration des DBA benötigt.
3. Optimierung: Der Anfrageoptimierer wählt den effizientesten AAP durch Aufstellen und Vergleichen von alternativen Plänen aus. Dabei wird zwischen einer logischen und physischen Optimierung unterschieden. Zum einen kann durch das logische Umordnen der Operationen mittels algebraischer Gesetze oder durch Einbeziehen

von Selektivitätsabschätzungen die Ausführungszeit der Anfrage verkürzt werden. Die Optimierung auf physischer Ebene bezieht Kenntnisse des verfügbaren Speicherplatzes, Datenbreiten und -typen sowie Verarbeitungskosten (Ausführungszeit, Leistungsaufnahme) ein. Dieser Optimierungsschritt benötigt bereits Kenntnisse über den verwendeten DBA und dessen unterstützte Operatoren. Zudem kann der DBA zur Abschätzung der Selektivität verwendet werden. Sind die Selektivitäten der Eingangsdaten bekannt, ist eine Abschätzung der Ausführungszeit, Datentransferzeit und des benötigten Speicherbedarfs möglich. Kapitel 4.4 präsentiert dazu die Ergebnisse des DBA zur Anfrageoptimierung.

4. Codegenerierung und Ausführung: Der ausgewählte AAP wird im letzten Schritt in ausführbaren Code umgewandelt und zum Scheduler weitergeleitet, welcher die Operatoren physisch auf die Ausführungseinheiten zuweist. Auch hier sollte die Anordnung der Daten im Speicher beachtet werden, um diese an die Speicherarchitektur der jeweiligen Ausführungseinheit anzupassen. Die Anfrage kann nun von der CPU ausgeführt, oder falls möglich, zum Coprozessor ausgelagert werden.

2.2.2 Grundbegriffe

Dieser Abschnitt gibt einen kurzen Überblick über die in dieser Arbeit verwendeten Grundbegriffe aus dem Bereich der Datenbanken.

Menge Der Begriff der *Menge* wurde von Georg Cantor [166] definiert: „Unter einer Menge verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objekten m [...] zu einem Ganzen.“ Diese Objekte m nennt Cantor dann die Elemente der Menge M. In der Informatik wird die Menge auch als Set oder Liste bezeichnet und bildet eine Datenstruktur, um Objekte zu speichern und zu organisieren. Die Elemente haben dann einen bestimmten Datentyp.

Relation Eine *Relation* R ist im Sinne der Datenbanken eine Teilmenge des kartesischen Produkts der n Wertebereiche (Domänen) $D_1, D_2, ..., D_n$: Die Domänen dürfen keine Mengen darstellen, da diese auch nicht atomare bzw. weiter zerlegbare Werte enthalten können. Zulässig sind nur Zahlen, Zeichenketten und andere nicht strukturierte Wertebereiche. Die Relation ist das Basiselement eines RDBMS (siehe Abschnitt 2.1.1).

Kardinalität Die *Kardinalität* $|M|$ der Menge M gibt die Anzahl der Elemente dieser Menge an. Die Definition gilt genauso für Tupel in Relationen.

Selektivität Die *Selektivität* sel_M gibt den relativen Anteil aller qualifizierenden Elemente einer Menge M an, die ein gegebenes Selektionsprädikat erfüllen. Zum Beispiel berechnet sich für einen Intersektionsalgorithmus auf zwei Mengen M und N die Selektivität aus dem Verhältnis der Kardinalitäten der Schnittmenge zur kleineren Menge:

$$sel_{MN} = \frac{|M \cap N|}{\min(|M|, |N|)}. \tag{2.1}$$

Hashtabelle Hashverfahren bieten die Möglichkeit, im Gegensatz zu einfachen Listen oder Bäumen, in einer konstanten Zeitkomplexität $\mathcal{O}(1)$ bestimmte Datensätze zu finden [91]. Dabei wird mit Hilfe einer *Hashfunktion* jedes Objekt des Datensatzes auf eine Position in einer *Hashtabelle* abgebildet. Bei der Suche nach einem bestimmten Objekt, muss deshalb nur die berechnete Position in der Hashtabelle und nicht der gesamte Datensatz durchlaufen werden. Die Hashtabelle stellt dabei ein lineares Feld mit den Indizes $0, ..., m-1$ dar. m ist die Anzahl der *Buckets* (Behälter) und bestimmt die Größe der Hashtabelle. Die Buckets nehmen die zu speichernden Datensätze auf. Gemäß der Abbildung $h : K \to \{0, ..., m-1\}$ weist die Hashfunktion jedem Schlüssel k aus der Menge aller Schlüssel K einen Hashwert $h(k)$ mit $0 \leq h(k) \leq m-1$ zu. Eine Kollision tritt auf, wenn unterschiedliche Schlüssel einen identischen Hashwert bekommen ($h(k_i) = h(k_j)$ mit $0 \leq i, j < |K|$, $i \neq j$). Kollisionen erhöhen damit die Zeitkomplexität, da für das Suchen eines Datensatzes zusätzliche Vergleiche innerhalb eines Buckets notwendig sind. Die mittlere Anzahl der Kollisionen ist durch den *Lastfaktor* r_{LF} gegeben und ist das Verhältnis aus Tupelanzahl zur Bucketanzahl:

$$r_{LF} = \frac{|K|}{m}. \tag{2.2}$$

2.2.3 Operatoren

Datenbanksysteme müssen eine Vielzahl von Datenbankoperatoren zur Anfrageverarbeitung unterstützen. Wie häufig bestimmte Operatoren verwendet werden, ist dabei sehr von der gewählten Anwendung abhängig. Eine Einordnung in verschiedene Anwendungsbereiche ermöglichen standardisierte Benchmarks, die ein angepasstes Datenschema für beliebige Datengrößen, sowie die zugehörigen Anfragen und Operatoren bereitstellen. Tabelle 2.1 zeigt dazu einen Überblick mit ausgewählten Anwendungsbereichen und Benchmarks für relationale Datenbanksysteme, die als Grundlage für die in dieser Arbeit untersuchten Algorithmen dienen. Zum einen ist hier der in der Literatur sehr ausführlich untersuchte TPC-H Benchmark (Transaction Processing Performance Council, TPC) zu nennen [141]. TPC-H besteht aus insgesamt 22 industrie- und geschäftsorientierten Anfragen, die grundlegende SQL-Operatoren wie `SELECT`, `JOIN`, `GROUP-BY` und `ORDER BY` enthalten. Diese SQL-Operatoren basieren auf elementaren Algorithmen wie Scan, Join, Aggregation und Sortieren. Star Schema [112], BigBench [54] und BigDataBench [152] sind modifizierte Versionen des TPC-H Benchmarks und nutzen ebenfalls die zuvor beschriebenen SQL-Operatoren.

Zahlreiche Forschungsarbeiten analysieren unter anderem die Verteilung und Häufigkeit der Operatoren im TPC-H Benchmark. Zum Beispiel stellen Vaidya u. a. [144] und Boncz u. a. [24] fest, dass der Join-Operator in fast allen Anfragen verwendet wird. Als zweithäufigster Operator wird das Sortieren verwendet. Andere Arbeiten wie [31] und [66] untersuchen auch den Einfluss der Benchmarks auf die Auslastung der Rechensysteme. Sie berichten ebenfalls, dass der Hash-Join im TPC-H und Star Schema Benchmark bis zu 61 % der gesamten Rechenzeit einnimmt. Eine Beschleunigung des Hash-Joins hat damit nicht nur einen großen Einfluss auf die Laufzeit des Benchmarks, sondern würde auch signifikant die Gesamtleistungsfähigkeit des zugrundeliegenden Datenbanksystems erhöhen. In Abschnitt 2.1.2 wurden bereits zahlreiche Forschungsarbeiten genannt, die

Tabelle 2.1: Überblick über moderne Datenbankanwendungen, Benchmarks und deren Operatoren für relationale Datenbanksysteme, die in dieser Arbeit untersucht werden.

Applikation/Benchmark	Operator	Algorithmus	Beschreibung
TPC-H [141], Star Schema [112], BigBench [54], BigDataBench [152]	Sortieren	Merge-Sort	Sortieren einer Liste
	Aggregation	Sort-Merge-Aggregation	Anwenden einer Aggregatfunktion (SUM, MIN, MAX, AVG, COUNT) auf Tupel mit identischen Schlüsseln
	Gruppierung	Sort-Merge-Gruppierung	Gruppierung von Tupeln mit identischen Schlüsseln (oft gleichzeitig mit Aggregation ausgeführt)
	Join	Sort-Merge-Join, Hash-Join	Vergleich von zwei oder mehr Relationen
	Scan	Filterung	Selektion/Projektion von Tupeln entsprechend eines Selektionsprädikats, z. B. zur Generierung von Bitmap-Indizes
Webbasierte Suchanfragen in Texten [12, 16], Data-Mining [165]	Sorted-Set Operationen	Intersektion, Vereinigung, Differenz	Vergleich von zwei oder mehreren Listen
OLAP-Anfragen in Data-Warehouses [90, 136]	Kompression, Bitmap-Index-Verarbeitung	WAH, PLWAH, COMPAX	Datenkompression mittels Bitmap-Indizes und Verarbeitung von Bitmap-komprimierten Daten

die Leistungsfähigkeit der in TPC-H relevanten Operatoren durch Anpassung der Hard- und Software verbessern [29, 34, 69, 159]. Auch für diese Arbeit wurden deshalb Algorithmen aus den TPC-H basierten Benchmarks für die Beschleunigung mit angepasster Hardware ausgewählt.

Ein weiterer wichtiger Anwendungsbereich für Datenbanken sind Suchmaschinen, die in der heutigen durch das Internet stark geprägten Zeit von großer Bedeutung sind. Zum einen müssen dabei große Datenmengen mit Texten durchsucht werden. Zum anderen erfolgen ebenfalls Analysen mit Hilfe von statistischen Methoden, um Trends und Suchmuster der Anfragen zu erkennen und damit die Suchergebnisse individuell zu verbessern (Data-Mining) [12, 16, 165]. Typische Algorithmen sind hier die Sorted-Set Operationen wie Intersektion, Vereinigung und Differenz, die jeweils zwei oder mehrere Listen miteinander vergleichen. Die korrespondierenden SQL-Operatoren sind `INTERSECT`, `UNION` bzw. `MINUS`. Detaillierte Erläuterungen zu den Algorithmen und deren parallelen Implementierungen folgen in den nächsten Unterabschnitten.

Ein dritter relevanter Bereich, in dem eine effiziente Abarbeitung von Datenbankoperatoren erforderlich ist, sind *Data-Warehouses*. Data-Warehouses [145], auch Datenlager genannt, sind zentral aufgestellte Datenbanken, die Daten von mehreren externen Quellen speichern. Damit profitieren auch Systeme wie das Cloud-Computing von Data-Warehouses, die zentrale und skalierbare Rechnerressourcen z. B. für mobile Kommunikationssysteme benötigen [96]. Die Arbeit von Kanev u. a. [90] analysierte über einen

Zeitraum von drei Jahren das Zusammenwirken von Serverapplikationen mit der zugrundeliegenden Hardwarearchitektur in Warehouse-Scale Computern, d. h. in den Rechnersystemen eines Data-Warehouses, um Rückschlüsse auf deren Auslastung zu ziehen. Die Auswertungen zeigen, dass etwa ein Drittel der Serverlast auf die Datenverarbeitung zurück geht. Konkret beinhalten diese das Ausführen der Datenbankoperatoren, Datentransfers und die Datenkompression.

Traditionelle relationale Datenbanken (RDBMS) wie sie z. B. für TPC-H-Anfragen eingesetzt werden, arbeiten standardmäßig auf unkomprimierten Daten. Die Kompression ist jedoch ein wichtiger Bestandteil in Data-Warehouses, da hier die Anzahl der Anfragen gering und die zu verarbeitende Datenmenge vergleichsweise größer als in RDBMS ist. Diese Art der Anfragen werden auch OLAP-Anfragen (Online Analytical Processing) genannt [145]. Des Weiteren speichern Data-Warehouses oft nur Kopien aus externen Quellen, die typischerweise nach etwa einem Tag aktualisiert werden. Die Datenkompression in Data-Warehouses ist deshalb sinnvoll, da sich Datensätze seltener ändern und die Komprimierung unabhängig von der Anfrageverarbeitung stattfinden kann (z. B. bei geringer Last des Systems direkt nach der Datenaktualisierung). Aus diesem Grund werden auch in dieser Arbeit Datenbankoperatoren zur Bitmap-Kompression und zur Verarbeitung auf Bitmap-komprimierten Daten untersucht [63]. Bitmap-Indizes sind dabei typische Indizierungsverfahren, die bei OLAP-Anfragen in Data-Warehouses zum Einsatz kommen [136]. Die dabei verwendeten Algorithmen basieren auf der Lauflängencodierung (Run-Length-Encoding, RLE) [18]. Beispiele sind Word-Aligned Hybrid (WAH) [158], Position List Word-Aligned Hybrid (PLWAH) [36] und Compressed Adaptive Index (COMPAX) [49]. Die Algorithmen basieren alle auf der RLE und unterscheiden sich nur durch die Kompression von Wörtern mit einzeln verteilten Bits (Sparse Bits), um eine maximale Datenkompressionsrate zu erhalten.

Die in Tabelle 2.1 aufgelisteten Algorithmen weisen Unterschiede hinsichtlich des Speicherzugriffsverhaltens, der Laufzeitkomplexität und der Parallelisierbarkeit auf Daten- und Befehlsebene auf. Der Erfolg der Beschleunigung der Algorithmen mit Hilfe anwendungsspezifischer Hardware sowie mit mehreren parallelen Verarbeitungseinheiten hängt dabei von den genannten Faktoren ab. Teilweise existieren aber auch Gemeinsamkeiten zwischen den Algorithmen z. B. in Bezug auf die parallele Abarbeitung. Diese Arbeit fokussiert sich deshalb auf drei Bereiche von Datenbankoperatoren, die ausführliche Untersuchungen der oben genannten Einflüsse erlauben:

- Algorithmen auf sortieren Listen: Intersektion, Vereinigung, Differenz
- Sortierverfahren: Merge-Sort
- Hashing/Hash-Join

Die folgenden Abschnitte erläutern die genannten Algorithmen im Detail. Dabei wird auf die ursprüngliche skalare als auch auf die Realisierung hinsichtlich der Datenparallelität eingegangen. Die Implementierungen und Ergebnisse der weiteren in Tabelle 2.1 gelisteten Algorithmen werden hier nur kurz präsentiert. Ausführliche Informationen sind auch in den vom Autor veröffentlichten Publikationen zu finden [61, 62, 63, 64].

Sorted-Set Algorithmen

Die ausgewählten Algorithmen auf sortierten Listen (Sorted-Set Algorithmen) Intersektion, Vereinigung und Differenz sind bekannt aus der Mengenlehre. Sie verknüpfen n Listen nach einem bestimmten Merkmal. In dieser Arbeit werden die Algorithmen auf $n = 2$ aufsteigend sortierten Listen A und B ausgeführt, die jeweils ganzzahlige positive Werte ohne Duplikate enthalten (binäre Operationen). Dieser Anwendungsfall ist typisch für Operatoren, die auf einzelnen Spalten einer Relation z. B. der Zeilenkennung (Schlüsseln) arbeiten, wenn Anfragen mit komplexen Selektionsprädikaten verwendet werden [123]. Wie bereits zuvor erläutert, profitieren ebenso webbasierte Suchanfragen in Texten von einer effizienten Abarbeitung der Sorted-Set Algorithmen [12, 16]. Des Weiteren ist die Intersektion auf zwei Listen für den Eclat-Algorithmus aus dem Bereich des Frequent Itemset Mining essentiell [165].

Im Einzelnen lassen sich die drei Algorithmen wie folgt definieren: Die *Intersektion* (auch Schnittmenge genannt) ist die Menge aller Elemente, die in den Listen A und B enthalten sind. Die *Vereinigung* ist die Menge aller Elemente, die in den Listen A oder B enthalten sind. Die *Differenz* ist die Menge aller Elemente, die in der Liste A aber nicht in der Liste B enthalten sind. Die Implementierungen in Software erfolgen damit nach einem ähnlichen Prinzip. Zwei Indizes `pos_a` und `pos_b`, die auf die Positionen der Elemente der Mengen A und B verweisen, ermöglichen den Zugriff auf die jeweiligen Werte a und b. Sie zeigen zu Beginn jeweils auf die ersten Elemente der Eingangsmengen. Beim Durchlaufen der beiden Mengen wird jeweils der Zeiger inkrementiert, dessen zugehöriger Wert kleiner ist. Bei Gleichheit werden sowohl `pos_a` als auch `pos_b` auf das nächste Element gesetzt. Die Ergebnismenge wird C genannt. Der Algorithmus bricht ab, sobald das Ende einer Menge erreicht ist. Daraus ergibt sich der in Abbildung 2.2a gezeigte Code für die Intersektion.

Abbildung 2.2b stellt den Ablauf anhand eines Beispiels dar. Die Pfeile verbinden dabei die zu vergleichenden Elemente. Die Ergebniselemente sind fett gedruckt. Die Listen enthalten ganzzahlige positive 32-Bit Werte ohne Duplikate und sind aufsteigend sortiert. Der Algorithmus beginnt mit dem Vergleich der ersten beiden Elemente. Das letzte Element der Menge A mit dem Wert 23 wird mit keinem Element aus der Menge B verglichen, da vorher das Ende dieser Menge schon erreicht ist. Der Algorithmus kann demnach sofort beendet werden, sobald ein Zeiger das Ende einer Menge erreicht hat. Es ist, wie sich im Beispiel erkennen lässt, nicht mehr möglich, dass zu dem Wert 23 ein gleicher aus der Menge B gefunden werden kann.

Die Implementierungen für die Vereinigung und Differenz unterscheiden sich von der Intersektion nur durch das Abspeichern im jeweiligen *if-else*-Zweig. Die Vereinigung speichert immer das Element mit dem kleineren Wert von a und b ab (bei Gleichheit a) und die Differenz speichert nur a, wenn $a < b$ gilt. Anders als bei der Intersektion, kann hier ein Kopieren restlicher Elemente am Ende des Algorithmus erforderlich sein (Quellcode im Anhang A, Abbildungen B.1 und B.3).

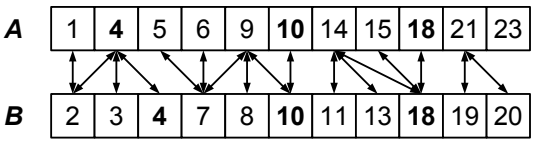
Bei Annahme von gleichmäßig verteilten Werten in den Mengen A und B verringert sich die Anzahl der Vergleiche n_{Comp} linear mit zunehmender Selektivität sel_{AB}:

$$n_{Set,Comp} = (\min(|A|, |B|) - |A| - |B| + 1)sel_{AB} + |A| + |B| - 1. \tag{2.3}$$

Im schlechtesten Fall sind $|A| + |B| - 1$ Vergleiche durchzuführen. Die Selektivität beträgt

```
int intersect(int* A, int l_a,
              int* B, int l_b,
              int* C)){
    int pos_a = 0, pos_b = 0, pos_c = 0;
    while(pos_a < l_a && pos_b < l_b){
        if(A[pos_a] == B[pos_b]){
            C[pos_c++] = A[pos_a];
            pos_a++;
            pos_b++;
        }
        else if(A[pos_a] < B[pos_b])
            pos_a++;
        else
            pos_b++;
    }
    return pos_c;
}
```

(a) Code-Ausschnitt



(b) Zahlenbeispiel

Abbildung 2.2: Softwareimplementierung der Sorted-Set Algorithmen am Beispiel der Intersektion

dann 0 %, die Intersektionsmenge ist leer und die Differenzmenge entspricht der Menge A. Der beste Fall tritt mit $\min(|A|, |B|)$ Vergleichen ein, wenn beide Mengen identisch sind ($sel_{AB} = 100\,\%$). Damit liegt die Zeitkomplexität für die Sorted-Set Algorithmen in $\mathcal{O}(N)$, wenn N die Gesamtanzahl der Elemente $|A| + |B|$ darstellt.

Parallele Sorted-Set Algorithmen Die skalare Ausführung der Sorted-Set Algorithmen weist mehrere Datenabhängigkeiten auf. Zum einen wird die Position der Zeiger durch das Vergleichsergebnis bestimmt. Zum anderen treten Abhängigkeiten der Zeiger zwischen aufeinanderfolgenden Iterationen der Schleife auf. Diese Eigenschaften stellen für die parallele Umsetzung des Algorithmus jedoch keinen wirklichen Nachteil dar. Die Idee einer parallelen Umsetzung basiert darauf, statt ein Element aus der Menge A mit einem Element aus der Menge B zu vergleichen, nun zwei Vektoren a und b mit den jeweils k Elementen a_i und b_i ($i = 0, ..., k-1; k > 1$) zu betrachten. Damit bleibt die allgemeine Struktur der Schleife erhalten, wobei nun jedes Element a_i mit jedem Element b_i in einer Iteration auf Gleichheit geprüft wird. Es sind deshalb k^2 parallele Vergleiche notwendig. Die Anzahl der neu zu ladenden Elemente pro Menge, d. h. die neuen Positionen der Zeiger, ergibt sich ebenfalls aus dem Vollvergleich. Abbildung 2.3 veranschaulicht das Prinzip an einem Beispiel für $k = 4$. Beim ersten Vergleich der Elemente $\{1, 4, 5, 6\}$ mit $\{2, 3, 4, 7\}$ ist der Wert 6 aus der Menge A kleiner als der Wert 7 aus der Menge B. Aus A können demnach vier neue Elemente geladen werden. In dieser 4er-Gruppe der Menge B ist es nicht mehr möglich, dass die Werte 2, 3 und 4 einen identischen Wert aus der Menge A finden. Diese Elemente müssen im nächsten Schritt nicht mehr in den Vergleich einbezogen werden. Neben dem Element mit dem Wert 7 können nun bereits die Werte 8, 10 und 11 im nächsten Schritt bearbeitet werden.

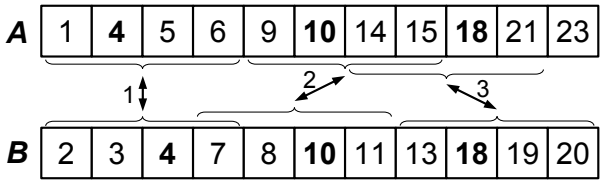


Abbildung 2.3: Zahlenbeispiel für die parallele Sorted-Set Intersektion mit $k = 4$

Genau wie für den skalaren Fall gemäß Gleichung 2.3 ergibt sich die Anzahl der Vergleiche in Abhängigkeit der Selektivität, nun jedoch skaliert mit dem Parallelitätsfaktor k:

$$n_{Set,Comp-Par} = \left(\frac{1}{k}(\min(|A|,|B|) - |A| - |B|) + 1\right) sel_{AB} + \frac{1}{k}(|A| + |B|) - 1. \quad (2.4)$$

Im schlechtesten Fall sind $\frac{1}{k}(|A| + |B|) - 1$ und im besten Fall sind $\frac{1}{k}\min(|A|,|B|)$ Vergleiche durchzuführen. Damit liegt die Zeitkomplexität bei $\mathcal{O}(\frac{N}{k})$, wenn N die Gesamtanzahl der Elemente $|A| + |B|$ darstellt. Die Laufzeit der Sorted-Set Algorithmen ist demnach proportional zur Anzahl der Elemente der Eingangsdaten dividiert mit dem Faktor k. Der Durchsatz des Algorithmus ist prinzipiell nur bei einer Selektivität von $sel_{AB} = 100\,\%$ durch die Bandbreite des Speichers limitiert. In diesem Fall ist $n_{Comp,Par}$ am kleinsten und für jeden Vergleich werden $2k$ neu geladene Elemente benötigt. Für $sel_{AB} < 100\,\%$ überwiegt im Mittel die Verarbeitungszeit.

In ähnlicher Weise können die Algorithmen der Sorted-Set Differenz und Vereinigung parallelisiert werden. Die Differenz nutzt das partielle Laden in gleicher Weise wie bei der zuvor beschriebenen Intersektion. Der Vollvergleich prüft nun, ob $a_i \neq b_i$ und ob $a_i < b_{k-1}$ gilt. Erfüllt das jeweilige Element aus a alle Vergleiche, kann es in die Ergebnismenge geschrieben werden. Bei der Vereinigung entspricht der Vollvergleich der Vektoren a und b einer Kombination aus Sortierung und Duplikateliminierung. Für den duplikatfreien Fall entstehen $2k$ Ergebniselemente, wobei die k kleineren abgespeichert werden und die k größeren Elemente in den Vergleich mit k neu geladenen Elementen gelangen. Sind alle zu vergleichenden Elemente aus a und b identisch, entstehen nur k Ergebniselemente und es können wieder $2k$ Elemente geladen werden. Das Zurückkopieren restlicher Elemente in die Ergebnismenge kann bei der Vereinigung und Differenz ebenfalls vektorbasiert mit k Elementen gleichzeitig erfolgen.

Sortieren

Das vor allem in TPC-H Benchmarks verwendete Sortieren ist eine oft genutzte Operation in Datenbanksystemen, die z. B. für Sort-Merge-Joins [13] oder den zuvor angesprochenen Sorted-Set Algorithmen vorangestellt wird, um deren Abarbeitung zu vereinfachen. Für diese Arbeit wurde das Sortierverfahren *Merge-Sort* ausgewählt. Wie später noch ersichtlich wird, ist es eines der effizientesten Sortieralgorithmen im Hinblick auf Daten- und Taskparallelität. Das Merge-Sort wurde 1945 von John von Neumann entwickelt und arbeitet nach dem Teile und Herrsche Prinzip [131]. Damit ist die Aufteilung der zu sortierenden Menge in Teilfolgen gemeint, die nachher wieder miteinander verschmolzen werden (engl.: to merge), bis die gesamte Menge sortiert ist. Abbildung 2.4b veranschaulicht das Vorgehen mit $N = 8$ Elementen. Die Eingangsliste wird zuerst in N Teilfolgen

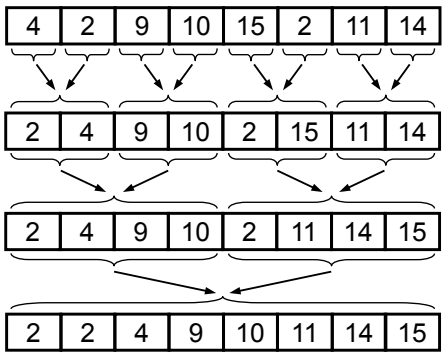
```
void merge(int* A, int l_a,
           int* B){
    int pos_a = 0, pos_b = 0, pos_c = 0;

    //kopiere vordere Folge in Hilfsarray
    while(pos_a <= l_a/2)
        B[pos_b++] = A[pos_a++];

    //verschmelzen
    pos_b = 0;
    while(pos_a < l_a && pos_c < pos_a){
        if(A[pos_a] < B[pos_b])
            A[pos_c++] = A[pos_a++];
        else
            A[pos_c++] = B[pos_b++];
    }

    //kopiere restl. Elemente zurück
    while(pos_c < pos_a)
        A[pos_c++] = B[pos_b++];
}
```

(a) Code-Ausschnitt des Merge-Vorgangs



(b) Zahlenbeispiel

Abbildung 2.4: Softwareimplementierung des Merge-Sort Algorithmus

der Länge 1 aufgeteilt, die bereits sortiert sind. Mit jeder Ebene verdoppelt sich die Länge der zu sortierenden Teilfolgen, bis nach $\log N$[1] Durchgängen die gesamte Menge sortiert ist.

Es existieren zahlreiche rekursive Softwareimplementierung des Merge-Sort, die Sortiernetzwerke nutzen [33, 108]. Der hier verwendete iterative Ansatz orientiert sich an der zuvor beschriebenen Vereinigung, die zwei Mengen verarbeitet und dabei Duplikate beibehält. Die Gemeinsamkeiten sind bei der späteren Entwicklung der Instruktionen zur Beschleunigung der Algorithmen von Vorteil, um Hardwareresourcen einzusparen. Es muss neben der zu sortierenden Menge auch Speicherplatz für mindestens $\frac{N}{2}$ Elemente vorhanden sein, da beim Verschmelzen zweier Teilfolgen, die vordere Folge in ein Hilfsarray kopiert wird. Ist die Gesamtlänge N keine Zweierpotenz, dann ist diese vordere Teilfolge kürzer. Deshalb ist es sinnvoll, von hinten durch die Gesamtmenge zu iterieren. In diesem Fall müssen weniger Elemente in das Hilfsarray kopiert werden. Zusätzlich entsteht der Vorteil, dass Elemente der hinteren Teilfolge schon an ihrer richtigen Position stehen können, da die vordere Teilfolge in das Hilfsarray kopiert wurde.

Das Codebeispiel in Abbildung 2.4a zeigt diesen eigentlichen Merge-Vorgang, bei dem zwei Teilfolgen verschmolzen werden, die sich in Liste `A` befinden. Um die vollständige Menge zu sortieren, ist dieser Merge-Vorgang insgesamt $N - 1$ mal auszuführen (Herleitung siehe Anhang C.1). Zunächst wird die vordere Teilfolge in das Hilfsarray `B` kopiert. Beim Vergleichen kann so das kleinere Element direkt wieder in die Ausgangsliste `A` ein-

[1]Die verwendete Notation der Logarithmusfunktion $\log x$ bezieht sich immer auf den Logarithmus zur Basis zwei: $\log_2 x$.

sortiert werden (in-place Algorithmus). In jeder Ebene müssen dabei $\frac{N}{2}$ Elemente kopiert werden. Mit insgesamt $\log N$ Ebenen benötigt das Kopieren demnach $\frac{N}{2} \log N$ Schritte.

Als nächstes folgt der Vergleich der Elemente beider Teilfolgen. Dieser Vorgang orientiert sich an der Sorted-Set Vereinigung, die zwei Mengen verarbeitet und dabei Duplikate beibehält. Die Anzahl der Vergleiche kann deshalb mit Hilfe von Gleichungen 2.3 ermittelt werden. $|A|$ und $|B|$ stellen dabei die Längen der beiden zu verschmelzenden Teilfolgen dar. Im besten Fall ist die gesamte Menge bereits sortiert. Dann sind pro Ebene $\frac{N}{2}$ Elemente zu vergleichen. Mit insgesamt $\log N$ Ebenen sind demnach mindestens $\frac{N}{2} \log N$ Vergleiche notwendig, um die gesamte Menge zu sortieren. Im schlechtesten Fall sind $N(\log N - 1) + 1$ Vergleiche durchzuführen. Die Elemente der Menge sind dann so verschränkt, dass die Sortierung erst in der letzten Ebene erkennbar wird. Eine solche Folge mit $N = 8$ Elementen ist z. B. $\{10, 15, 11, 3, 12, 4, 1, 6\}$. Die zugehörigen Herleitungen sind im Anhang C.1 angegeben. Stehen am Ende noch Elemente in `B`, müssen diese wieder zurück kopiert werden. Die Anzahl der Kopiervorgänge ist von der Anzahl der Vergleiche und damit ebenfalls datenabhängig.

Die obere Grenze für die Laufzeit des Merge-Sort wird durch den Kopiervorgang und aus der maximalen Anzahl der Vergleiche bestimmt. Das Zurückkopieren restlicher Elemente aus dem Hilfsarray ist in diesem Fall nicht zu betrachten, da jedes Element bereits in den Vergleichsprozess einbezogen wurde. Die Zeitkomplexität für das iterative Merge-Sort lässt sich wie folgt berechnen:

$$\begin{aligned} T(N) &\leq \frac{N}{2} \log N + N(\log N - 1) + 1 \\ &\leq \frac{3}{2} N \log N - N + 1. \end{aligned} \tag{2.5}$$

Folglich ist $T(N) \in \mathcal{O}(N \log N)$. Die Zeitkomplexität des Merge-Sort ist damit eines der effizientesten Sortieralgorithmen im Hinblick auf Daten- und Taskparallelität [33]. So stellen Satish u.a. [128] in ihren Implementierungen fest, dass bei einer 16-fachen Erhöhung der SIMD-Breite Merge-Sort mit dem Faktor 6,5 und andere Sortierverfahren wie Radix-Sort nur um Faktor 3,7 skalieren. Das heißt, obwohl Radix-Sort im Gegensatz zu Merge-Sort eine lineare Zeitkomplexität aufweist, kann ein mit großen SIMD-Breiten (> 2048 Bit) implementiertes Merge-Sort bei hohen Kardinalitäten immer noch leistungsfähiger sein.

Paralleles Sortieren Die parallele Realisierung des Merge-Sort hinsichtlich der Datenparallelität kann dem Konzept der Sorted-Set Algorithmen folgen. Das heißt, die Struktur des Merge-Vorgangs in Abbildung 2.4a bleibt erhalten. Das Kopieren und der Vergleich einzelner Elemente erfolgt nun auf k Elementen gleichzeitig ($k > 1$). Es werden zunächst $\frac{N}{k}$ Teilfolgen der Länge k, dann $\frac{N}{2k}$ Teilfolgen der Länge $2k$, danach $\frac{N}{4k}$ Teilfolgen der Länge $4k$ usw. verschmolzen, bis nach $\log N - \log k$ Durchgängen die gesamte Menge sortiert ist. Im Gegensatz zum skalaren Algorithmus beginnt das parallele Merge-Sort nicht mit Teilfolgen der Länge 1, sondern bereits mit Teilfolgen der Länge k. Die ersten $\log k$ Merge-Vorgänge müssen deshalb mit einem separaten Durchlauf behandelt werden, in dem vorsortierte Datenblöcke der Länge k entstehen. Dazu sind $\frac{N}{k}$ Schritte notwendig.

Die Berechnungen zum skalaren Merge-Sort lassen sich auf eine k-fache Parallelität übertragen. Im Allgemeinen skaliert die Elementanzahl mit dem Parallelitätsfaktor k und

die ersten $\log k$ Sortierebenen entfallen. Das Kopieren der vorderen Teilfolge in das Hilfsarray benötigt demnach insgesamt $\frac{N}{2k} \log \frac{N}{k}$ Schritte. Mit den Herleitungen zu der Anzahl der Vergleichsschritte[2] aus Anhang C.1 berechnet sich die obere Grenze der Laufzeit des parallelen Merge-Sort mit

$$\begin{aligned} T(N,k) &\leq \underbrace{\frac{N}{k}}_{Vorsort.} + \underbrace{\frac{N}{2k} \log \frac{N}{k}}_{Kopieren} + \underbrace{\frac{N}{k}\left(\log \frac{N}{k} - 1\right) + 1}_{max.Vergleichsschritte} \\ &\leq \frac{3N}{2k} \log \frac{N}{k} + 1. \end{aligned} \tag{2.6}$$

Genau wie beim skalaren Merge-Sort gilt weiterhin $\mathcal{O}(\frac{N}{k} \log \frac{N}{k})$, wobei die Laufzeit nun mit dem Parallelitätsfaktor k skaliert. Zusätzlich trägt beim parallelen Merge-Sort die Vorsortierung mit $\frac{N}{k}$ Schritten bei, die aber nur ein lineares Wachstum mit der Elementanzahl aufweist.

Hashing

Hashverfahren sind im Wesentlichen die Alternative zu Ansätzen, die auf sortierten Daten arbeiten. Das Hashing ist dabei nur ein Hilfsmittel, um z. B. den aus TPC-H Benchmarks bekannten Hash-Join zu realisieren. Hashing beschreibt dabei das Berechnen eines Hashwertes mit Hilfe einer gegebenen Hashfunktion. Als Hashfunktionen können z. B. bitweise Operatoren oder die Modulo-Operation eingesetzt werden [10, 116, 118]. Die Auswahl wird durch deren Komplexität, Speicherbedarf und Effizienz bestimmt. Im Rahmen dieser Arbeit wurden zwei Hashfunktionen untersucht: eine konfigurierbare Hashfunktion für ganzzahlige Werte (Integer) und die *CityHash*-Funktion für Zeichenketten (Strings). Der CityHash-Algorithmus wird in der vom Autor veröffentlichten Publikation [8] genauer analysiert. Im Folgenden liegt der Fokus auf der Integer-Hashfunktion, die universell einsetzbar ist und zumeist auf die Schlüssel von Schlüssel-Wert-Paaren angewendet wird. Diese Schlüssel liegen häufig als ganzzahlige Werte vor.

Der zugehörige C-Code ist in Abbildung 2.5 dargestellt. Die Hashfunktion basiert auf einer Bitselektion. Werden n Bits an ausgewählten Positionen eines w Bit breiten Schlüssels extrahiert ($n \leq w \leq 32$), entsteht ein n-Bit Hashwert, der es erlaubt 2^n Buckets in einer Hashtabelle zu adressieren. Der Code realisiert das Extrahieren der Bits, indem jedes Bit soweit nach rechts verschoben wird, sodass nicht selektierte Bits überschrieben werden. Der Algorithmus ist nach $\mathcal{O}(wN)$ Schritten durchlaufen, wobei N die Anzahl der Schlüssel angibt. Da die Bitbreite w konstant ist, weist das Hashing eine lineare Zeitkomplexität auf.

Eine Anpassung der Hashfunktion an die Eingangsdaten und damit eine Minimierung der Kollisionen in der Hashtabelle ist durch eine intelligente Auswahl der Bitpositionen in der Bitmaske möglich. So kann vor dem eigentlichen Hashing mittels *Sampling* eine Häufigkeitsverteilung der in den Eingangsschlüssel gesetzten Bits bestimmt werden [8]. Die Bits mit der höchsten Entropie bilden dann die Bitmaske für die Hashfunktion. Sampling reduziert damit die Ausführungszeit der eigentlichen Anfrageausführung durch ein zuvor

[2] Ein Vergleichsschritt verarbeitet $2k$ Elemente und führt deshalb k^2 elementweise Vergleiche gleichzeitig aus.

```
void int_hash(
    unsigned int* key,          //Input Schlüssel
    unsigned short* hashValue,  //Output Hash-Werte
    int keySize,                //Anzahl Schlüssel
    unsigned int hashFunc       //Hash-Maske
){
    int i, w;
    unsigned int hash;
    unsigned int mask = 0xFFFFFFFF, shVal, shVal_neg;

    for(i=0; i<keySize; i++){
        //Bitselektion
        hash = key[i] & hashFunc;

        //Extrahiere Bits
        for(w=BIT_WIDTH-2; w>=0; w--){
            if(!(hashFunc & (0x1<<w))){
                //teilweiser Rechts-Shift
                shVal = hash & (mask<<w);
                shVal_neg = hash & ~(mask<<w);
                hash = (shVal>>1) | shVal_neg;
            }
        }

        hashValue[i] = hash;
    }
}
```

Abbildung 2.5: Code-Ausschnitt der Integer-Hashfunktion basierend auf einer Bitselektion. Eine Schleife iteriert über alle Bits jeden Schlüssels (`key`) und extrahiert entsprechend einer Hashmaske (`hashFunc`) mittels mehrerer bitweiser Operationen die selektierten Bits, die dann den Hashwert (`hashValue`) bilden.

durchgeführtes Scannen der Daten und kann deshalb der Anfrageoptimierung zugeordnet werden.

Paralleles Hashing Die Parallelität auf Datenebene ist für das Hashing einfach zu erkennen. Die Hashfunktion kann prinzipiell auf jeden Schlüssel gleichzeitig angewendet werden. Es bestehen keine Datenabhängigkeiten. Mit einem Parallelitätsfaktor k reduziert sich die Zeitkomplexität des Hashings damit auf $\mathcal{O}(\frac{wN}{k})$.

Join

Im Gegensatz zu den Sorted-Set Algorithmen ist der Verbund-Operator, oder auch *Join* genannt, für die Verarbeitung von Relationen mit mehreren Attributen vorgesehen wie sie z. B. in TPC-H Benchmarks [66] oder im Bereich des Information Retrieval [121] verwendet werden. Der hier verwendete und am meisten genutzte Equi-Join oder auch innere Join ist durch den algebraischen Ausdruck $R \bowtie_{R.A=S.A} S$ definiert. Das Ergebnis enthält alle Tupel aus den Relationen R und S, deren Attributwerte A identisch sind. Die Relationen

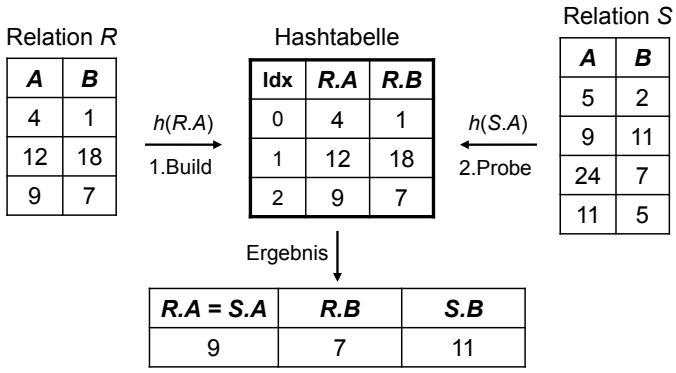


Abbildung 2.6: Allgemeiner Ablauf eines Hash-Joins

können weitere Attribute enthalten, die im Folgenden zusammen als eine Payload des Key-Payload Paars gehandhabt werden. Die bekanntesten Implementierungsstrategien sind der 1) Nested-Loop-Join, 2) Sort-Merge-Join und 3) Hash-Join. Der Nested-Loop-Join iteriert mit Hilfe zweier verschachtelter Schleifen über beide Eingangsrelationen. Obwohl die Partitionierung und Parallelisierung trivial ist, weist der Nested-Loop-Join auf Grund der quadratischen Zeitkomplexität $\mathcal{O}(|R||S|)$ eine mangelhafte Leistungsfähigkeit auf.

Der Sort-Merge-Join berechnet die Ergebnisrelation in einem einzigen Durchlauf ähnlich der Sorted-Set Intersektion. Solche Methoden sind vorzuziehen, wenn ein effizientes Sortierverfahren zur Verfügung steht, oder die Daten bereits sortiert im Speicher abgelegt sind. Die Zeitkomplexität des Algorithmus wächst linear mit der Kardinalität $\mathcal{O}(|R|+|S|)$. Müssen die Daten jedoch zuerst sortiert werden, ist die Komplexität des Sort-Merge-Joins von dem gewählten Sortierverfahren abhängig.

Abbildung 2.6 stellt beispielhaft den Ablauf des Hash-Joins dar, der auf Relationen mit unsortierten Attributen arbeitet. Die als erstes ausgeführte *Build-Phase* bildet die kleinere Relation, hier R, durch Anwenden einer Hashfunktion in die Hashtabelle ab. In der darauffolgenden *Probe-Phase* werden alle Tupel aus S durch Anwenden der gleichen Hashfunktion mit den korrespondierenden Einträgen aus der Hashtabelle verglichen und die gültigen Ergebnistupel abgespeichert. Das jeweilige Ergebnistupel kann bereits nach einem Schritt gefunden sein, da nicht die gesamte Hashtabelle abgesucht werden muss, sondern nur der Bucket, der durch das Tupel aus S bestimmt wird. Es ist deshalb zu erwarten, dass der Algorithmus in linearer Zeitkomplexität $\mathcal{O}(|R|+|S|)$ durchlaufen werden kann. Der Nachteil des Hash-Joins besteht in dem zusätzlich notwendigen Speicherplatz für die Hashtabelle. Zumeist beeinflusst das Hashing während der Build- und Probe-Phase maßgeblich die Gesamtausführungszeit [95]. Der in dieser Arbeit vorgestellte Datenbankbeschleuniger integriert erweiterte Instruktionen für das Hashing. Der Hash-Join erlaubt deshalb, den Einfluss des beschleunigten Operators auf einen komplexen Algorithmus zu untersuchen. Die Parallelisierung des Hash-Joins erfolgt auf Taskebene und wird deshalb in Kapitel 5.3.3 diskutiert. Wie zuvor beschrieben, kann das zugrundeliegende Hashing dabei auf Datenebene parallelisiert werden.

2.3 Herausforderungen

Mit den nun bekannten Grundlagen stellt sich die Frage, welche Herausforderungen bei der Entwicklung von Beschleunigerarchitekturen für die Datenbank-Anfrageverarbeitung zu bewältigen sind. Im Folgenden werden dazu die wichtigsten Punkte diskutiert:

- Die Beschleuniger müssen große Datenmengen verarbeiten und damit optimal an die zugrundeliegenden Algorithmen angepasst sein, um einen hohen Durchsatz und eine niedrige Latenz zu erzielen. In dieser Arbeit werden dazu die Möglichkeiten einer Anpassung mit applikationsspezifischer Hardware untersucht. Je nach Art und Weise der Ansteuerung ist eine ASIC-Implementierung als auch ein flexibler Ansatz mit einem ASIP denkbar. Für Letzteres ist auch die Anpassung der Software an die neue Hardware notwendig, um die Beschleuniger effizient anzusteuern.
- Für den Einsatz der Beschleuniger z. B. in batteriebetriebenen Geräten ist eine niedrige Leistungsaufnahme notwendig. Im Vergleich zu hochperformanten GP-Prozessoren sind deshalb die benötigte Chipfläche und die Taktfrequenz zu reduzieren. Die zu erzielende Energieeffizienz ist dann vor allem durch die anwendungsspezifische Hardware bereitzustellen.
- Die Operatoren der Anfrageverarbeitung sind vielfältig. Die Beschleuniger sollten entweder eine Vielzahl an beschleunigten Algorithmen unterstützen oder eine ausreichende Flexibilität für universelle Aufgaben bereitstellen.
- Die Datenbankbeschleuniger müssen flexibel auf schnelle Änderungen der Last sowie das Umschalten zwischen mehreren Datenbankoperatoren reagieren. Es ist deshalb eine zentrale Scheduling-Einheit notwendig, die die Steuerung des Gesamtsystems erlaubt sowie Taktfrequenz und Versorgungsspannung skaliert, um Durchsatz und Leistungsaufnahme an die Last des Systems anzupassen.
- Die Skalierbarkeit der Beschleunigerarchitekturen ist zu gewährleisten. Dies beinhaltet zum einen die Skalierung innerhalb der Beschleuniger z. B. die Erhöhung von Daten- und Registerbreiten, als auch die Erweiterung des Gesamtsystems durch die parallele Nutzung mehrerer Verarbeitungseinheiten. Dabei stellt jeweils die Speicherbandbreite einen Flaschenhals dar, deren Einfluss mit geeigneten Methoden zu reduzieren ist.
- Die potentielle Integration der Gesamtarchitektur des Datenbankbeschleunigers in ein übergeordnetes Managementsystem (z. B. DBMS) muss garantiert sein. Dabei ist auf eine geeignete Anbindung des Hauptspeichers und weiterer Prozessoreinheiten zu achten. Dabei limitieren ebenfalls die Bandbreiten der Schnittstellen den Datenaustausch.

Eine Diskussion über die Herausforderungen seitens der konkreten Hardwareimplementierungen folgen individuell in den nächsten Kapiteln.

3 DBA als anwendungsspezifischer Hardwareblock

Eine erste offensichtliche Umsetzung eines Datenbankbeschleunigers (DBA) mit dedizierter Hardware ist der Entwurf eines anwendungsspezifischen integrierten Schaltkreises (Application-Specific Integrated Circuit, ASIC). Der in diesem Kapitel untersuchte Ansatz bezieht sich deshalb auf die Implementierung des DBA ASIC. Das Ziel ist es, die Leistungsmöglichkeiten des ASIC auf die Datenbank-Anfrageverarbeitung anzuwenden und deren Herausforderungen zu diskutieren. Ebenfalls ist ein quantitativer Vergleich mit dem DBA ASIP möglich, der in Kapitel 4 vorgestellt wird.

Diese Kapitel beginnt mit einer Einführung in das Design eines ASIC für die Beschleunigung von Datenbankoperatoren und stellt den aktuellen Stand der Technik dar (3.1). Danach wird die ASIC-Implementierung der in Abschnitt 2.2.3 vorgestellten Sorted-Set Algorithmen beschrieben. In Abschnitt 3.3 folgt die Auswertung der Ergebnisse hinsichtlich Durchsatz und Energieverbrauch sowie der Vergleich mit anderen Architekturen.

3.1 ASIC-Design

3.1.1 Implementierungsaspekte

Bei der Entwicklung von anwendungsspezifischer Hardware sind theoretisch keine Grenzen hinsichtlich Fläche und Verlustleistung vorhanden. Die maximale Leistungsfähigkeit kann durch die Eigenschaften der zu beschleunigten Applikation eingeschränkt sein. Zumeist wird aber ein Kompromiss zwischen verschiedenen Implementierungsaspekten eingegangen, die im Folgenden beschrieben sind:

- Taktfrequenz: Wie viele logische Operationen können verknüpft und in einem Taktzyklus ausgeführt werden, ohne den kritischen Pfad in der Hardware bedeutend zu verlängern, sodass die angestrebte Taktfrequenz erreicht wird?
- Parallelität der Hardware: Welche Operationen der Anwendung können parallel ausgeführt werden? Ist das Verwenden von breiteren Registern für die Parallelität auf Wortebene sinnvoll? Inwiefern erhöht Pipelining die Leistungsfähigkeit?
- Skalierbarkeit: Was sind die theoretischen Grenzen der Leistungsfähigkeit bei der Beschleunigung einer Applikation? Mit welcher Komplexität wachsen dabei die Kosten, d. h. Fläche und Verlustleistung der Logik und Register beim Anwenden von Datenparallelität und Pipelining?
- Leistungsaufnahme: Inwieweit kann die Verlustleistung bei veränderter Taktfrequenz und Chipfläche beeinflusst werden, um vorgegebene Parameter noch einzuhalten?

- Speicherzugriff: Kann das Laden und Speichern der Daten parallel ausgeführt werden? Welche Breite sollte dabei die Speicherschnittstelle aufweisen? Welche Speichergröße ist sinnvoll, um eine ausreichende Speicherkapazität bereitzustellen und dabei die Fläche und Verlustleistung zu minimieren?
- Komplexität der Algorithmen: Wie groß ist das Verhältnis von sequentiellen zu parallelem Anteil des Algorithmus? Gibt es eine Abhängigkeit zwischen Durchsatz und Kardinalität der Eingangsdaten?
- Grad der Spezialisierung: Können mehrere unterschiedliche Algorithmen mit der gleichen Hardware verarbeitet werden, ohne dabei die Leistungssteigerung bedeutend zu beeinflussen?
- Potentielle Integration in ein Mehrkernsystem: Kann das angeschlossene Verbindungsnetzwerk die Daten mit mindestens dem Durchsatz bereitstellen, wie diese im ASIC verarbeitet werden?

3.1.2 Stand der Technik

Wie bereits in Abschnitt 2.1.2 erwähnt, existieren zahlreiche Forschungsarbeiten zu Hardwarebeschleunigern für Operatoren zur Datenbank-Anfrageverarbeitung. Zum Beispiel beschleunigen die Arbeiten in [108] und [29] das Sortieren und die Selektion, um einen Median- bzw. Join-Operator aufzubauen. Die Implementierungen werden auf einem FPGA evaluiert und zeigen einen Beschleunigungsfaktor von bis zu einer Größenordnungen gegenüber der Softwareimplementierung mit GP-Prozessoren sowie einer GPU. FPGAs arbeiten üblicherweise mit einem synchronen Design, d. h. die Logikblöcke sind mit einer vorgegebenen Frequenz getaktet. Die Arbeit in [108] untersucht das Sortiernetzwerk ebenfalls als asynchrone Implementierung. Im Vergleich zum synchronen Design wird zwar nur ein Drittel der effektiven Latenz benötigt, jedoch halbiert sich auch der Durchsatz.

Dennl u. a. [38] präsentieren ebenfalls Hardwaremodule, die grundlegende Datenbankoperatoren ausführen und damit einen um 6,2× höheren Durchsatz gegenüber einem Intel Core i7 erreichen. Sie nutzen dabei die partielle Rekonfigurierbarkeit des FPGA aus, um je nach geforderten Operationen der Anfrage die zugehörigen Module bereitzustellen. Die Flexibilität wird erreicht, indem Teile des FPGA innerhalb von Millisekunden zur Laufzeit neu konfiguriert werden.

Die Autoren in [132] beschleunigen die Sorted-Set Intersektion in Verbindung mit dem Eclat-Algorithmus, der in Anwendungen wie Frequent Itemset Mining vorkommt. Im konkreten Beispiel nimmt die auf einem FPGA implementierte Intersektion 70 % der Verarbeitungszeit des Eclat-Algorithmus ein. Im Vergleich zur Softwareimplementierung mit einem GP-Prozessor kann für den Intersektionsoperator und für den vollständigen Algorithmus ein 6× bzw. 4,4× höherer Durchsatz erreicht werden.

Chung u. a. [34] präsentieren LINQits, das die domainspezifische Anfragesprache LINQ mit Hilfe von programmierbaren Hardwaretemplates beschleunigt. LINQits besteht aus einem kompletten Framework, der automatisch verfügbare Hardwaretemplates zur Compilierzeit auf die Software abbildet. Für den in dieser Arbeit entwickelten ASIC wäre eine Integration in das Tomahawk MPSoC von Kapitel 5 möglich. Das MPSoC ermöglicht

dann ähnlich zu LINQits eine dynamische Zuweisung der Systemressourcen mit Hilfe einer zentralen Scheduling-Einheit.

Die zuvor beschriebenen Arbeiten verbessern wie erwartet den Durchsatz und die Leistungsaufnahme. Die Ergebnisse bestätigen bereits die Vorteile anwendungsspezifischer Hardware für Datenbankalgorithmen. Die dabei verwendeten FPGAs arbeiten bei Taktfrequenzen unter 200 MHz mit einer Leistungsaufnahme von bis zu 1 W. Das Ziel der in dieser Arbeit entwickelten ASIC-Lösung soll es deshalb sein, höhere Taktfrequenzen für einen besseren Durchsatz zu erreichen und dabei gleichzeitig die Verlustleistung zu reduzieren.

3.2 Implementierung des ASIC

3.2.1 Architektur des ASIC

Grundsätzlich teilt sich der Aufbau eines ASIC in mehrere Logikblöcke und Speicher (SRAM) auf. Die Anzahl und Größe der unabhängigen Speicherblöcke richtet sich nach den Anforderungen des zu implementierenden Algorithmus. Die Logikblöcke enthalten *Load/Store-Einheiten*, einen *Datenpfad* für die eigentliche Datenverarbeitung, sowie einen *Steuerpfad*, der die *Steuerlogik* und den endlichen Automaten (Finite State Machine, FSM) enthält. Die Load/Store-Einheiten generieren die Adressen und Kontrollsignale für die Zugriffe auf die Speicher. Der Steuerpfad durchläuft mit Hilfe der FSM mehrere Zustände, um den Ablauf des Algorithmus zu festzulegen.

Abbilden der Sorted-Set Algorithmen auf die Hardware

Der DBA ASIC soll zunächst die drei Sorted-Set Algorithmen Intersektion, Vereinigung und Differenz implementieren. Anpassungen, um die weiteren in Tabelle 2.1 aufgelisteten Algorithmen zu unterstützen, sind nachher einfach möglich und werden in Abschnitt 3.2.2 erläutert. Als Basisarchitektur dient die zuvor allgemein beschriebene Struktur mit Load/Store-Einheiten, Datenpfad und Steuerpfad. Hinsichtlich der Algorithmen ist bekannt, dass zwei Eingangs und eine Ausgangsliste existieren. Der DBA ASIC enthält deshalb neben der Verarbeitungslogik insgesamt drei Speicher wie in Abbildung 3.1 schematisch dargestellt. Speicher A und B enthalten die beiden Eingangslisten, während C die Ergebnisliste abspeichert. Die Aufteilung der Daten auf mehrere Speicher erspart das Arbitrieren zwischen Ein- und Ausgangsdaten. Auf Grund der durch den verwendeten SRAM zu berücksichtigenden Latenz bei Lesezugriffen, entlastet diese Struktur ebenfalls das Zeitverhalten auf dem Pfad zwischen Logik und Speicher. Die Speicherbreite $W_{Mem} = kW_{Elem}$ ist entsprechend dem Parallelitätsfaktor k und der Datenbreite eines Elements W_{Elem} zu wählen, um auf jeweils k Elemente gleichzeitig zuzugreifen (Datenparallelität).

Um die parallele Abarbeitung der Sorted-Set Algorithmen entsprechend Abbildung 2.3 aus Kapitel 2.2.3 zu realisieren, befinden sich im Datenpfad Vektorregister mit jeweils k Elementen und die dazugehörigen Vergleichsoperatoren. Die parallele Umsetzung der Sorted-Set Algorithmen verwendet ein partielles Laden, d. h. die Elemente der Vektorregister werden teilweise mit neuen Elementen befüllt wobei auch Elemente nur die Positionen im Register wechseln. Die Anzahl der neu zu ladenden Elemente muss deshalb

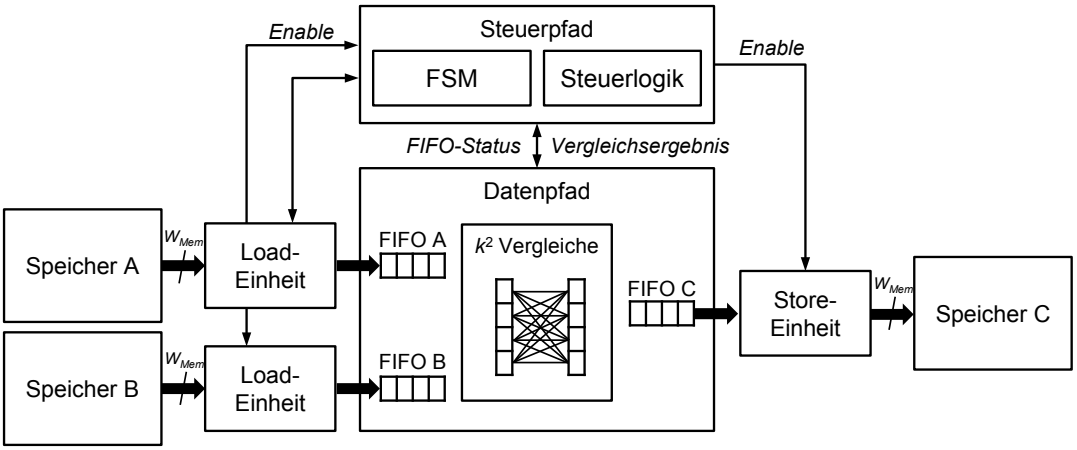


Abbildung 3.1: Blockdiagramm des DBA ASIC

kein Vielfaches von k sein. Der Speicher liefert bei einem Ladevorgang jedoch immer genau k neue Elemente. Aus diesem Grund enthält der Datenpfad zusätzlich insgesamt drei FIFO-Speicher (First in – First out), die als Ringspeicher mit jeweils $2k$ Einträgen implementiert sind. FIFOs A und B halten die geladenen Daten, sodass immer zwei mal k Elemente zu den Vergleichern gelangen. Auf Grund des einzuhaltenden Speicheralignments ist auch FIFO-Speicher C notwendig, der die Ergebniselemente zwischenspeichert.

Im parallelen Algorithmus werden alle Elemente beider Vektoren miteinander verglichen. Es werden deshalb insgesamt k^2 Vergleicher im Datenpfad benötigt. Abbildung 3.2 zeigt dazu beispielhaft den Datenpfad für die Sorted-Set Algorithmen. Die Vergleicher erzeugen gleichzeitig jeweils drei Ergebnisbits (jeweils für $<, =$ und $>$), die als zusammengefasste Bitmaske (Vergleichsergebnis) zur Steuerlogik zurückgeführt wird. Mit dem Vergleichsergebnis berechnet die Steuerlogik die Anzahl der zu ladenden bzw. abzuspeichernden Elemente und aktiviert damit die FIFO-Speicher im Datenpfad. Die Statussignale der FIFO-Speicher beinhalten die Leer- und Voll-Flags, mit denen über die Load/Store-Einheiten die Enable-Signale der drei Speicher angesteuert werden. Damit wird gewährleistet, dass A und B nicht leer laufen bzw. C nicht überläuft.

Bisher wurde angenommen, dass die k Elemente innerhalb eines Taktzyklus geladen werden können. Dieser Ansatz ist unabhängig von der Datenverteilung und beeinflusst damit nicht den Durchsatz. Geht man davon aus, dass reale Datensätze zumeist eine Selektivität von kleiner als 100 % aufweisen, werden nicht immer k neue Elemente pro Takt benötigt. Der Durchsatz des Algorithmus ist dann nicht mehr durch den Speicher sondern durch die Verarbeitungslogik limitiert. Die mittlere Anzahl zu ladender Elemente pro Takt und pro Menge ergibt sich aus der Kardinalität der Menge A bzw. B dividiert mit der Anzahl der Vergleiche $n_{Comp,Par}$ aus Gleichung 2.4. Gilt z. B. $|A| = |B|$, dann beträgt diese über der Selektivität gemittelte Anzahl $\frac{3}{4}k$. In diesem Fall hätte eine Reduzierung der Speicherinterfacebreiten um 25 % im Mittel keinen Einfluss auf den Durchsatz, wenn die FIFO-Speicher der Eingangsdaten ausreichend Elemente vorhalten können.

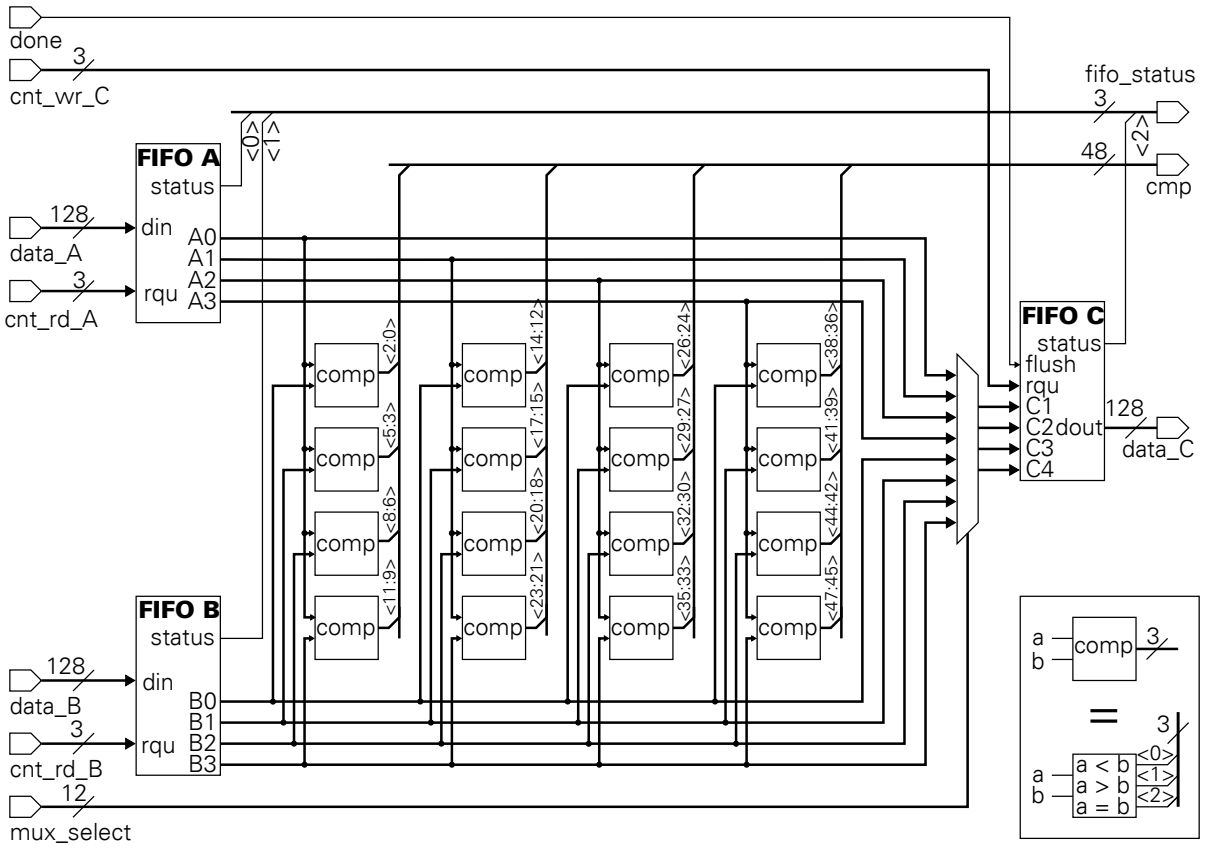


Abbildung 3.2: Datenpfad des DBA ASIC für die Sorted-Set Algorithmen am Beispiel $k = 4$, $W_{Mem} = 128\,\text{Bit}$, $W_{Elem} = 32\,\text{Bit}$

Pipelining

Neben der zuvor beschriebenen Datenparallelität, kann auch eine Parallelität durch Pipelining angewendet werden. Diese ist vor allem notwendig, um den zeitlichen Pfad zwischen SRAM und dem ersten Register zu verkürzen und damit die Latenz der verwendeten SRAMs zu verbergen. Da die FIFO-Speicher zugleich als Register wirken, stellen diese bereits Pipelineregister dar. Damit entsteht eine dreistufige Pipeline, dessen kritischer Pfad sich nun auf den Kontrollsignalen zwischen dem Datenpfad und der Steuerlogik befindet. Dieser verläuft zunächst mit den Eingangsdaten aus den FIFO-Speichern A bzw. B durch den Vollvergleich im Datenpfad. Das Vergleichsergebnis gelangt in die Steuerlogik, die damit wieder die FIFO-Speicher im Datenpfad ansteuert. Das Einfügen einer Pipelinestufe im kritischen Pfad würde allerdings die Berechnung der zu ladenden Elemente um einen Takt verzögern. Damit wäre das Befüllen der Vektorregister nur in jedem zweiten Takt möglich und würde den Durchsatz halbieren.

Ablaufsteuerung

Die Ablaufsteuerung des DBA ASIC wird mit Hilfe der in Abbildung 3.3 dargestellten FSM realisiert. Um zu Beginn in einem definiertem Anfangszustand zu starten, muss von außen ein Reset ausgelöst werden. Danach wird in `IDLE` darauf gewartet, dass mittels

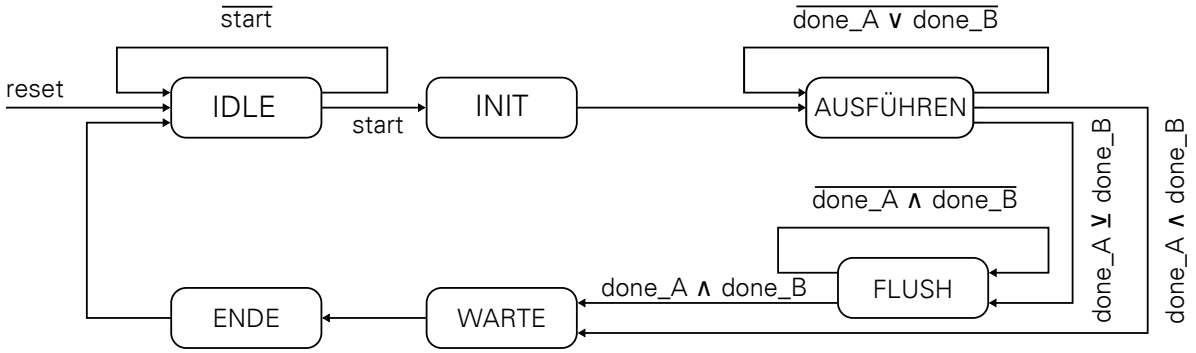


Abbildung 3.3: Endlicher Automat zur Ablaufsteuerung des DBA ASIC

`start` die Abarbeitung angestoßen wird. Zu diesem Zeitpunkt liegen noch keine Daten in den FIFO-Speichern A und B. Erst im Zustand `INIT` werden explizit Daten aus den Eingangsspeichern vorgeladen. Direkt danach folgt der Zustand `AUSFÜHREN`, in dem kontinuierlich die geladenen Daten in den Datenpfad gelangen und dort verglichen werden. Dieser Zustand stellt den Hauptbestandteil des Algorithmus dar und wird solange ausgeführt bis die von der jeweiligen FIFO-Logik erzeugten Signale `done_A` bzw. `done_B` anzeigen, dass die letzten Elemente aus den Speichern gelesen wurden. Ist nur eine der beiden Eingangslisten leer, wird im Zustand `FLUSH` die jeweils andere Liste zu Ende gelesen und mit den verbliebenen Elementen verglichen. Im Zustand `WARTE` erfolgt ein letzter Vergleich der in den FIFO-Speichern A und B verbliebenen Elemente. `FINISH` ermöglicht dann das Zurückschreiben aller übrigen Elemente aus C in den Ergebnisspeicher. Zusätzlich wird hier das für die Sorted-Set Vereinigung und Differenz notwendige Zurückkopieren restlicher Elemente ausgeführt. Danach beginnt der Zyklus von Neuem, indem der Automat wieder auf das Signal `start` wartet.

3.2.2 Weitere Algorithmen

Neben den Sorted-Set Algorithmen wurden vom Autor noch Hardwareblöcke für das Hashing sowie für die Kompression von Bitmap-Indizes entworfen. Diese Operatoren werden für das Arbeiten mit Hashtabellen bzw. die Verarbeitung von komprimierten Daten eingesetzt und beeinflussen ebenfalls die Leistungsfähigkeit der Datenbank-Anfrageverarbeitung (siehe Abschnitt 2.2.3, Tabelle 2.1). Für diese Algorithmen bleibt die Top-Level Architektur des ASIC erhalten bzw. vereinfacht sich überwiegend. Es kann zunächst der zweite Speicher für die Eingangsdaten wegfallen, da nur auf einer Eingangsliste gearbeitet wird. In der FSM entfallen dann die Zustände `FLUSH` und `WARTE`, da die Anzahl der Ladevorgänge nicht datenabhängig bzw. zu Beginn festgelegt ist. Damit entfallen auch die FIFO-Speicher im Datenpfad.

Beim Hashing werden alle Bits eines bis zu 32 Bit breiten Schlüssels durch eine Hashfunktion selektiert. Für jedes Bit existiert nun im Datenpfad eine Verschiebeeinheit die eine Rechtsverschiebung entsprechend der gegebenen Hashmaske durchführt. Auf diese Weise können bis zu k 32-Bit Schlüssel pro Takt geladen und verarbeitet sowie k Hash-

werte abgespeichert werden. Das Sampling erstellt eine Häufigkeitsverteilung der in den Schlüsseln enthaltenen Bits. Der Datenpfad enthält nun für jedes Bit einen Zähler, der jeweils bei einem gesetzten Bit inkrementiert wird. Genau wie beim Hashing können bis zu k 32-Bit Schlüssel pro Takt geladen und verarbeitet werden [20].

Bei der Bitmap-Kompression wird ein Bitstrom in 31-Bit Abschnitte unterteilt, die dann in das durch den jeweiligen Kompressionsalgorithmus vorgegebene Codewort übersetzt werden. Auch hier werden k 31-Bit Abschnitte parallel verarbeitet und k 32-Bit Codewörter gleichzeitig abgespeichert. Da auf Grund des Speicheralignments trotzdem immer Vielfache von 32 Bit geladen werden müssen, ist eine Vorverarbeitungslogik notwendig, die die 31-Bit Abschnitte erzeugt. Dieses Lademuster wiederholt sich nach einer bekannten Anzahl von Takten, sodass keine FIFO-Speicher mit flexibel steuerbaren Zwischenregistern wie bei den Sorted-Set Algorithmen notwendig sind [168].

3.3 Evaluierung des DBA ASIC

Der zuvor beschriebene DBA ASIC implementiert nun die Sorted-Set Algorithmen mit den für Datenbanksystemen typischen Datenbreiten W_{Elem} von 8, 16 und 32 Bit. Damit ist bereits bei einer konstanten Breite des Speicherinterfaces von $W_{Mem} = 128$ Bit eine Untersuchung der Skalierbarkeit mit den Parallelitätsfaktoren $k = \{4, 8, 16\}$ möglich. Aus den Ergebnissen können danach Rückschlüsse auf eine Variation der Speicherinterfacebreite gezogen werden. Für die Realisierung verschiedener Elementbreiten W_{Elem} sind die Bitbreiten der FIFO-Speicher und des Vergleichers im Datenpfad sowie aller Register anzupassen, die eine Abhängigkeit von der Elementanzahl in den Vektoren aufweisen. Bei einer Verdopplung von k vervierfacht sich zwar die Anzahl der Vergleicher, jedoch ist auch die Bitbreite der Daten zu halbieren. Damit ist ein lineares Wachstum der Logikfläche des Datenpfades mit k zu erwarten.

Untersuchungen haben gezeigt, dass SRAM den Flächen- und Leistungsverbrauch der Logik um ein Vielfaches übersteigen kann. Für die Evaluierung des DBA ASIC wird die Größe eines Speicherblocks deshalb mit 32 kB gewählt. Damit entsteht ein Kompromiss zwischen Fläche, Leistungsaufnahme und der erforderlichen Speicherkapazität. So kann ein Speicherblock für die Konfiguration mit $W_{Elem} = 32$ Bit bis zu 8192 Elemente aufnehmen. Diese Elementanzahl ist ausreichend, um den Einfluss von Randeffekten (z. B. Pipelining, Initialisierung der FSM) auf den Durchsatz auszuschließen.

Die RTL-Beschreibung des DBA ASIC wurde mit Mentor Questa simuliert und mit dem Synopsys Design Compiler synthetisiert. Während der Simulation auf Netzliste können die Schaltaktivitäten der Hardware aufgezeichnet werden. Mit Hilfe von Synopsys PrimeTime kann aus diesen Informationen die Verlustleistung der Datenbankoperatoren abgeschätzt werden. Die folgenden Abschnitte präsentieren die Ergebnisse der Leistungsfähigkeit sowie den Flächen- und Leistungsverbrauch des DBA ASIC. Es schließt sich danach ein Vergleich der Resultate mit anderen Prozessorarchitekturen an.

3.3.1 Durchsatz, Flächen- und Leistungsverbrauch

In einem ersten Schritt erfolgt die Synthese der RTL-Beschreibung des DBA ASIC mit dem Ziel, die maximal mögliche Taktfrequenz und die zugehörige Chipfläche zu ermitteln.

Tabelle 3.1: Ergebnisse des DBA ASIC für die Sorted-Set Intersektion bei verschiedenen Datenbreiten, Selektivität: 50 %

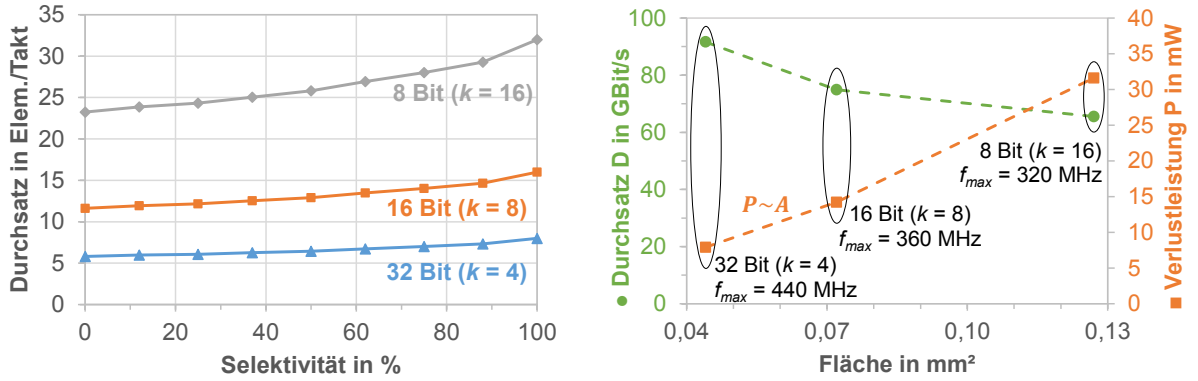
Parallelitätsfaktor (W_{Elem})	$k = 4$ (32 Bit)	$k = 8$ (16 Bit)	$k = 16$ (8 Bit)
Max. Taktfrequenz f_{max} in MHz	440	360	320
Durchsatz in Elem./Takt	6,5	13,0	25,6
Durchsatz bei f_{max} in GBit/s	91,6	74,9	65,5
Fläche bei f_{max} in mm^2	1,004	1,032	1,087
Logik	0,044	0,072	0,127
Speicher	0,960	0,960	0,960
Verlustleistung bei f_{max} in mW	67,8	64,7	77,0
Logik	7,9	14,2	31,6
Speicher	59,9	50,5	45,4
ATE-Produkt			
in mm^2 mW/(Elem./Takt)^2	1,611	0,395	0,128
in mm^2 mW/(GBit/s)^2	0,008	0,012	0,019

Als Standardzellen und SRAM-Makros werden Bibliotheken für einen 65 nm Prozess von TSMC bei typischen Bedingungen (1,2 V, 25 °C) verwendet. Die Ergebnisse sind in Tabelle 3.1 zusammengefasst. Wie zuvor angenommen, wächst die Fläche der Logik linear bei Reduzierung der Datenbreite. Dabei nehmen die drei FIFO-Speicher etwa 50 % der Logikfläche ein. Nimmt man den 96 kB Speicher hinzu, benötigt die Logik nur ca. 10 % der ASIC-Gesamtfläche. Mit der Reduzierung der Datenbreite von 32 auf 8 Bit fällt die maximale Taktfrequenz von 440 MHz auf 320 MHz, da sich die Komplexität der Multiplexer in den FIFO-Speichern und im Datenpfad und somit deren Verzögerungszeiten erhöhen. Ein Beispiel ist das Selektieren der Ergebniselemente für die Sorted-Set Vereinigung: bei einem $k \times k$ Vergleich entstehen 2^k Möglichkeiten, die Elemente anzuordnen.

Im nächsten Schritt wird die Ausführungszeit der Sorted-Set Algorithmen gemessen. Dabei liegt der Fokus auf der Sorted-Set Intersektion, wobei die Auswertungen auf die Ergebnisse der Vereinigung und Differenz übertragen werden können (siehe Anhang Tabelle A.2). Die Elemente in den Eingangslisten werden als vorzeichenlose ganzzahlige Werte interpretiert und sind gleichmäßig über beide Eingangsmengen verteilt. Die Elementanzahl ist dabei durch die Größe der Speicher limitiert. Die Listen für die 8 Bit breiten Eingangsdaten können jedoch nur maximal 255 Elemente enthalten. Hier begrenzt die Bitbreite die Anzahl der Elemente. Der Durchsatz ergibt sich zunächst aus den Listenlängen $|A|$ und $|B|$ sowie der Ausführungszeit in Takten t_{Takt}. Wird die Taktfrequenz f_{max} mit einbezogen, kann der Durchsatz D_{Set} mit

$$D_{Set} = \frac{f_{max}}{t_{Takt}}(|A| + |B|) \tag{3.1}$$

berechnet werden. In Tabelle 3.1 ist zunächst der Durchsatz als Elementanzahl pro Takt angegeben, um den Vorteil der höheren Parallelität zu verdeutlichen. Bei einer Halbierung der Datenbreite verdoppelt sich die Anzahl der parallel zu verarbeitenden Elemente entsprechend dem Parallelitätsfaktor k. Der Durchsatz ist ebenfalls in Abbildung 3.4a



(a) Durchsatz als Funktion der Selektivität

(b) Durchsatz und Verlustleistung über der Fläche (ohne Speicher), Selektivität: 50 %

Abbildung 3.4: Durchsatz und Verlustleistung des DBA ASIC für die Sorted-Set Intersektion für verschiedene Bitbreiten der Eingangsdaten

als Funktion der Selektivität dargestellt. Der Durchsatz nimmt mit steigender Selektivität zu, da die Anzahl der Vergleiche abnimmt. Je mehr identische Elemente sich in den Eingangslisten befinden, desto mehr Elemente können pro Takt nachgeladen werden. Im besten Fall (Selektivität: 100 %) lädt der ASIC insgesamt $2k$ Elemente aus beiden Eingangsmengen und speichert k Elemente pro Takt ab. Der DBA ASIC erreicht hier für die jeweils gewählten Datenbreiten den theoretisch maximalen Durchsatz, der durch die Speicherbandbreite bestimmt ist. Bei einer Selektivität von 0 % und einer ungleichmäßigen Datenverteilung können im schlechtesten Fall nur k Elemente pro Takt geladen werden. In den Experimenten wurde eine gleichmäßige Verteilung der Eingangsdaten angenommen. Dies erlaubt auch bei einer Selektivität von 0 % das Laden von durchschnittlich $1{,}5k$ Elementen pro Takt. In Bezug auf die Verarbeitung einzelner Bits, bleibt der Durchsatz bei unterschiedlichen Datenbreiten theoretisch gleich. Da sich aber die Taktfrequenz mit einer steigenden Parallelität verringert, nimmt auch der Durchsatz entsprechend ab.

Es schließt sich nun die Betrachtung der Verlustleistung an, die ebenfalls in Tabelle 3.1 zu sehen ist. Bei einer Verdopplung der Parallelität verdoppelt sich die Verlustleistung des Logikanteils in gleicher Weise wie der Flächenverbrauch (siehe dazu auch Abbildung 3.4b). Der Anteil der statischen Leistung der Logik ist dabei kleiner als 1 %. Auffallend ist die Abnahme der Verlustleistung des Speichers bei kleineren Datenbreiten. Das ist auf die Eigenschaften der SRAM-Zellen zurückzuführen, deren Leistungsaufnahme durch den Kurzschlussstrom bestimmt wird, der beim Umschalten der Transistoren entsteht.[1] Diese interne Verlustleistung ist nicht von der Aktivität des ASIC sondern von der Frequenz und der gewählten Technologie abhängig. Da die Taktfrequenz des ASIC bei kleineren Datenbreiten auch geringer ist, nimmt die Verlustleistung des Speichers ab.

Mit zunehmenden Parallelitätsfaktor steigt die Fläche und Verlustleistung an, wobei sich der Durchsatz bezogen auf die Elementgröße jeweils verdoppelt. Dieser Einfluss bewirkt eine Reduzierung des ATE-Produkts der ASIC-Implementierungen um Faktor 4,1

[1] Diese interne Leistungsaufnahme der SRAM-Zelle nimmt etwa 70 % der Gesamtverlustleistung unter typischen Prozessparametern ein [107].

bzw. 3,1 (siehe Tabelle 3.1). Dies bestätigt den Vorteil der höheren Parallelität durch Verwenden größerer Vektorregister. In Bezug auf die Verarbeitung einzelner Bits, erhöht sich das ATE-Produkt mit dem Parallelitätsfaktor. Das ist vor allem auf den Rückgang der Taktfrequenzen und des damit eingeschränkten Durchsatzes zurückzuführen. Abbildung 3.4b bestätigt das Verhalten. Die Verarbeitung von 32-Bit Elementen mit $k = 4$ weist den höchsten Durchsatz sowie den niedrigsten Flächen- und Leistungsverbrauch auf.

3.3.2 Vergleich mit anderen Architekturen

Der letzte Teil der Experimente umfasst den Vergleich der Leistungsfähigkeit des DBA ASIC mit anderen relevanten Hardwarearchitekturen. Es wird zunächst mit dem Synopsys `ARC EM4-Mini` und einem Standard-Mikrocontroller von Cadence Tensilica (`108Mini`) verglichen. Beide Prozessoren besitzen eine RISC-Architektur ohne Cache. Der `ARC EM4-Mini` ist je an einen 32 kB lokalen On-Chip Speicher für Instruktionen und Daten angebunden. Die Speichergröße ist konfigurierbar, wurde aber mit 32 kB pro Block gewählt. Damit entsteht ein Flächen- und Leistungsverbrauch des Prozessors, der in der gleichen Größenordnung wie der des DBA ASIC liegt. Die Speicherzugriffslatenz des `ARC EM4-Mini` beträgt nur einen Takt. Im Gegensatz dazu nutzt der `108Mini` einen externen Speicher, dessen Speicherzugriff eine höhere Latenz hat und nicht deterministisch ist. Flächenaufwand und Verlustleistung des `108Mini` liegen aber trotzdem im Bereich des DBA ASIC. Die RTL-Beschreibungen dieser Prozessoren wurden ebenfalls mit den TSMC 65 nm Bibliotheken synthetisiert. Zusätzlich wird mit den Ergebnissen eines modernen x86 GP-Prozessors (`Intel i7-920`) aus [130] verglichen. Die Autoren haben ebenfalls die Leistungsfähigkeit des Prozessors mit der Intel SSE-Befehlssatzerweiterung untersucht. Alle Prozessoren führen den in Abschnitt 2.2.3 gezeigten C-Code für die Sorted-Set Intersektion jeweils mit einem Kern und einem Hardware-Thread aus. Tabelle 3.2 fasst die Vergleichswerte zusammen. Im Folgenden werden die Ergebnisse zunächst für eine Datenbreite von 32 Bit diskutiert.

Im Gegensatz zum DBA ASIC beträgt die Speicherbreite des `ARC EM4-Mini` und `108Mini` nur 32 Bit und ist damit 4× geringer. Des Weiteren haben die Messungen ergeben, dass deren Softwareimplementierung im Mittel nur 0,1 Elem./Takt verarbeiten kann. Der DBA ASIC erreicht bei 32-Bit Elementen einen durchschnittlichen Durchsatz von 6 Elem./Takt. Daraus ergibt sich ein Gesamtspeedup des DBA ASIC von Faktor 83,3 und 76,3 gegenüber dem `ARC EM4-Mini` bzw. `108Mini`.

Der `Intel i7-920` arbeitet mit einer 6× höheren Taktfrequenz und nimmt bis zu 1900× mehr Leistung als der DBA ASIC auf. Es ergibt sich ein 14,5× und 2,6× höherer Durchsatz des DBA ASIC gegenüber der Softwareimplementierung auf dem `Intel i7-920` bzw. der mit den SIMD-Instruktionen beschleunigten Variante. In der SSE-Implementierung aus [130] finden genau wie im DBA ASIC 128-Bit Lade- und Speicherbefehle sowie die Vergleichsinstruktion `PCMPESTRM` Verwendung. `PCMPESTRM` ermöglicht ähnlich wie der ASIC-Datenpfad einen Vollvergleich von zwei 128-Bit Vektoren in ungefähr der Zeit eines einzelnen Vergleichs. Der DBA ASIC profitiert aber von der reinen Hardwareimplementierung, wodurch kein Overhead wie in der Software des x86-Prozessor entsteht.

Die Verringerung des Durchsatzes bei geringeren Datenbreiten ist für den DBA ASIC

Tabelle 3.2: Sorted-Set Intersektion: Vergleich des DBA ASIC mit anderen Prozessoren, Selektivität: 50 %. Wenn nicht anders angegeben, gelten die Werte für $W_{Elem} = 32$ Bit.

	DBA ASIC	ARC EM4-Mini	108Mini	Intel i7-920	Intel i7-920 + SSE4.2
Technologie in nm	65	65	65	45	45
Taktfrequenz in MHz	444	320	442	2670	2670
Durchsatz in GBit/s					
32 Bit	91,6	1,1	1,2	6,3	35,6
16 Bit	74,9	0,5	0,6	3,1	20,0
8 Bit	65,5	0,3	0,3	1,5	14,5
Gesamtfläche in mm^2	1,004	0,388	0,22	263	263
Verlustleistung in W	0,068	0,017	0,024	(1)130	(1)130
Energie in pJ/Bit	0,7	16,5	20,5	21 214	3782
ATE-Produkt in $mm^2\,mW/(GBit/s)^2$	$8{,}1 \cdot 10^{-3}$	5,5	3,7	$8{,}6 \cdot 10^{5}$	$2{,}7 \cdot 10^{4}$

(1)TDP entnommen aus [85].

auf den Rückgang der Taktfrequenz zurückzuführen. Der Durchsatz der Vergleichsprozessoren verringert sich ebenfalls, da trotzdem pro Element immer ein Speicherzugriff notwendig ist. Lediglich die eingesetzten SSE-Instruktionen des `Intel i7-920` erlauben das gleichzeitige Laden mehrerer Elemente. Wie die Durchsätze in Tabelle 3.2 für den x86-Prozessor zeigen, kann die Leistungsfähigkeit trotzdem nicht konstant gehalten werden. Das ist auf den zusätzlichen Overhead beim Zusammenfügen der Ergebniselemente und die hohe Latenz der Vergleichsinstruktion zurückzuführen.

Die berechnete Energie pro Bit ist in Abbildung 3.5 zu sehen. Mit steigender Selektivität verringert sich die benötigte Energie, da sich der Durchsatz erhöht (vgl. Abbildung 3.4a). Das Maß für diesen Anstieg ist dabei vom jeweiligen Compiler abhängig. Eine Ausnahme stellt die Implementierung des `Intel i7-920` mit den SSE-Instruktionen dar, die nicht wie der DBA ASIC ein partielles Laden der Eingangsdaten in die Vektorregister unterstützt. Der Vorteil, dass die Listen auf Grund mehrerer identischer Elemente schneller durchlaufen werden können, ist dann nur bei einer Selektivität von genau 100 % gegeben. Zusammengefasst ergeben sich für den DBA ASIC Energieeinsparungen von einer (`ARC EM4-Mini`) bis zu mehr als vier Größenordnungen (`Intel i7-920`). Berechnet man das ATE-Produkt und bezieht damit auch die Fläche in den Vergleich ein, ergeben sich Verbesserungen im Bereich zwischen drei und acht Größenordnungen (siehe Tabelle 3.2).

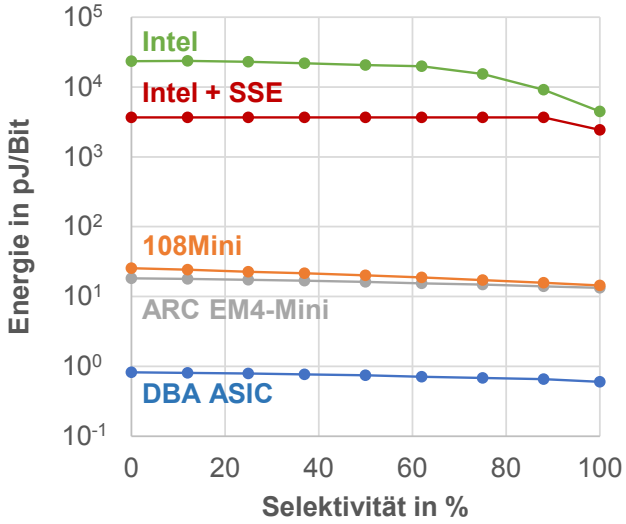


Abbildung 3.5: Energieverbrauch als Funktion der Selektivität für die Sorted-Set Intersektion des DBA ASIC im Vergleich mit anderen Prozessorarchitekturen, $W_{Elem} = 32\,\text{Bit}$

3.4 Schlussfolgerungen

In diesem Kapitel wurde die Entwicklung anwendungsspezifischer Hardwareblöcke für die Beschleunigung der Datenbank-Anfrageausführung präsentiert. Die Ergebnisse des DBA ASIC haben eine Beschleunigung des Durchsatzes um Faktor 2,6 gegenüber GP-Prozessoren und eine Verbesserung der Energieeffizienz um Faktor 21,8 im Vergleich zu RISC-Controllern mit niedriger Leistungsaufnahme gezeigt. Die Untersuchungen verdeutlichen damit, dass der Einsatz von anwendungsspezifischer Hardware auch für Datenbankalgorithmen sinnvoll ist und die Leistungsfähigkeit gegenüber existierenden Lösungen verbessert. Die folgenden Punkte fassen weitere Ergebnisse zusammen:

- Taktfrequenz: Die Untersuchungen haben eine Verringerung der Taktfrequenz um etwa 20 % je Verdopplung des Parallelitätsfaktors gezeigt. Eine weitere Erhöhung der Parallelität ist durch die Vergrößerung der Datenbreite der Speicherinterfaces und der Vektorregister möglich. Um die dabei zu erwartende zusätzliche Verringerung der Taktfrequenz zu verhindern, sollten beispielsweise Pipelineregister in den kritischen Pfad, d. h. auf die Kontrollsignale zwischen dem Daten- und Steuerpfad eingefügt werden. Auf Grund von direkten Abhängigkeiten in der ASIC-Implementierung der Sorted-Set Algorithmen würden die zusätzlich Pipelinestufen den Durchsatz verringern. Dies kann jedoch durch eine höhere Taktfrequenz kompensiert werden.
- Parallelität der Hardware: Bei den Sorted-Set Operationen bietet sich die parallele Ausführung des Vergleichs zweier Vektoren an. In gleicher Weise ist das parallele Anwenden der Hashfunktion beim Hashing und die gleichzeitige Kompression mehrerer Bitmap-Indizes anwendbar. Je breiter die Register, desto höher die Datenparallelität und desto höher der Durchsatz. Das Design des ASIC enthält dabei

Pipelineregister, die die Latenz der verwendeten SRAM-Blöcke verbergen und somit das Zeitverhalten auf dem Pfad zum SRAM entspannen.

- Skalierbarkeit: Grundsätzlich ist die Leistungsfähigkeit der Sorted-Set Algorithmen, des Hashings sowie der Bitmap-Kompression durch die Bandbreite des Speichers limitiert. Lediglich für die Sorted-Set Algorithmen und Datensätzen mit einer Selektivität kleiner 100 % begrenzt die Verarbeitung die Gesamtausführungszeit. Bei einer Skalierung der Verarbeitungslogik, wächst die Fläche linear mit zunehmender Parallelität k, d. h. bei gleichbleibenden Registergrößen aber geringeren Datenbreiten. Würde sich ebenfalls die Breite des Speicherinterfaces und der Vektorregister vergrößern, ist ein quadratisches Flächenwachstum des Datenpfads mit k zu erwarten. Es folgen dazu Untersuchungen in Kapitel 4.3.
- Der Leistungsverbrauch des Logikanteils skaliert linear mit der Logikfläche und der Taktfrequenz, während der Leistungsverbrauch des Speichers nur von der Taktfrequenz abhängig ist.
- Speicherzugriff: Die Anordnung der Daten im Speicher für die Sorted-Set Algorithmen, das Hashing als auch für die Bitmap-Kompression ermöglichen das gleichzeitige Laden mehrerer Elemente, sofern die Datenbreite des Speicherinterfaces den parallelen Zugriff unterstützt.
- Die Laufzeitkomplexität der vorgestellten Algorithmen hängt linear von der Kardinalität der Eingangsdaten ab. Je größer der Parallelitätsfaktor, desto geringer der Einfluss des sequentiellen Anteils auf die Verarbeitungszeit. Es besteht keine Abhängigkeit zwischen Durchsatz und Kardinalität der Eingangsdaten, da die Ausführung der Hardware deterministisch ist. Es besteht lediglich eine Datenabhängigkeit hinsichtlich der Selektivität für die Sorted-Set Algorithmen. Die untersuchten Datenbankoperatoren bearbeiten immer einen fortlaufenden Datenstrom. Andere Algorithmen mit häufigen Verzweigungen im Ablauf (z. B. Sortieren), wären für das ASIC-Design deshalb nicht von Vorteil. Eine mögliche Lösung ist, nur zeitkritische Abschnitte als Hardware zu implementieren (z. B. Sortiernetzwerk) und die komplexe Ablaufsteuerung von einem Prozessor zu übernehmen.
- Grad der Spezialisierung: Mittels einer intelligenten Implementierung können mehrere unterschiedliche Algorithmen mit der gleichen Hardware verarbeitet werden. Beispielsweise sind für die kombinierte Implementierung der drei Sorted-Set Algorithmen in einem ASIC nur Änderungen im Datenpfad und der Steuerlogik vorzunehmen. Auch für das Hashing und die Bitmap-Kompression bleiben grundlegende Elemente in der Steuerlogik sowie bei der Ablaufsteuerung der FSM erhalten.
- Potentielle Integration in ein Mehrkernsystem: Im besten Fall verarbeitet der evaluierte DBA ASIC für die Sorted-Set Algorithmen 2× 128 Bit pro Takt. Das in Kapitel 5 verwendete Network-on-Chip der Tomahawk-Plattform transferiert 64 Bit pro Takt. Für eine Integration in das Mehrkernsystem müsste der Durchsatz des Verbindungsnetzwerks entsprechend durch eine Erweiterung der Datenbreite oder eine im Vergleich zum ASIC erhöhte Taktfrequenz angepasst werden.

Obwohl sich mittels einer intelligenten Implementierung mehrere ähnliche Algorithmen die gleiche Hardware teilen können, ist die Umsetzung weiterer Datenbankalgorithmen zur

Anfrageausführung sehr kostenintensiv im Hinblick auf Zeitaufwand und Verifikation. Deshalb wäre eine Vereinfachung des Ansatzes wünschenswert, indem ein Hardwareblock als Grundlage genutzt und dieser nur mit den Funktionalitäten der zu implementierenden Anwendung ergänzt wird. Genau diesen Ansatz verfolgt der im nächsten Kapitel präsentierte Prozessor, dessen Befehlssatz und Architektur erweitert wird, um Datenbankoperatoren energieeffizient auszuführen. Damit kann nicht nur der Verifikationsaufwand gesenkt sondern auch die Produktivität gesteigert sowie die Zeit zu einer evtl. Markteinführung verkürzt werden.

4 DBA als Prozessor mit anwendungsspezifischem Befehlssatz

Die bisher gezeigte ASIC-Implementierung des DBA nutzt dedizierte Hardwareblöcke um eine energieeffiziente Ausführung der Operatoren für die Datenbank-Anfrageverarbeitung zu erreichen. Dieser hohe Grad an Spezialisierung geht auf Kosten der Flexibilität, d. h. jede Änderung am Steuerungsablauf (FSM) oder das Hinzufügen weiterer Funktionalitäten bedarf immer der Verifikation des gesamten ASIC. Das Konzept des Prozessors mit anwendungsspezifischem Befehlssatz (Application-Specific Instruction-Set Processor, ASIP) stellt eine Kombination aus ASIC und einem universellen RISC dar. Dabei wird ein gegebener Basisprozessor mit zusätzlichen Hardwarekomponenten erweitert, die im Programmcode der Software als neue Instruktionen angesprochen werden. Damit wird ein hohes Maß an Flexibilität zur Verfügung gestellt, ohne einen vollständigen Prozessor und dessen Komponenten von Grund auf neu entwickeln zu müssen. Zudem basiert die Entwicklung größtenteils in Software, deren Simulationsaufwand im Vergleich zur Hardware erheblich geringer ist. Außerdem übernimmt der Prozessor die Befehlssteuerung. Damit ist keine zusätzliche Verifikation der FSM bei einer Änderung des Programmablaufs notwendig. Dieses Konzept des ASIP wird bereits in den Bereichen der digitalen Signalverarbeitung (Digitaler Signalprozessor, DSP) und der Bildverarbeitung (Grafische Verarbeitungseinheit, GPU) erfolgreich angewendet. Auch für die Beschleunigung von Datenbankoperatoren existieren Arbeiten, die spezialisierte Hardware und Standard-Befehlssatzerweiterungen der General-Purpose CPUs einsetzen. Diese folgen jedoch nicht dem vollständigen ASIP-Ansatz, wie er in dieser Arbeit vorgestellt wird.

Dieses Kapitel beschreibt nun das strategische Vorgehen zum Entwurf eines DBA ASIP. Nach einem Überblick über das Modell von konfigurierbaren Prozessoren, werden allgemeine Optimierungsstrategien des ASIP und der Stand der Technik präsentiert. Abschnitt 4.2 erläutert danach die Implementierung und Integration ausgewählter Datenbankoperatoren in den ASIP. Die Evaluierung der Algorithmen erfolgt in Abschnitt 4.3 mit verschiedenen Implementierungen und Prozessorkonfigurationen, die ebenfalls mit anderen Arbeiten verglichen werden. Zusätzlich erlaubt der entwickelte Forschungschip *Titan3D* eine erweiterte Auswertung von Leistungs- und Energieverbrauch des DBA ASIP. Zuletzt werden in Abschnitt 4.4 die Möglichkeiten des DBA hinsichtlich der Anfrageoptimierung untersucht.

4.1 Konfigurierbare Prozessoren

4.1.1 Modell

Obwohl es heutzutage eine Vielzahl an Prozessoren gibt, können alle Funktionalitäten zu einem vereinfachten Prozessormodell reduziert werden. Dieses in Abbildung 4.1 dargestellte Modell beinhaltet mehrere Komponenten und Schnittstellen, die im Folgenden beschrieben werden.

Der *Kern* eines Prozessors ist für die Ausführung von Befehlen zuständig, die in einer Instruktionssatzarchitektur (ISA) definiert sind. Dieser Kern besteht aus weiteren Komponenten: die *Arithmetisch-logische Einheit* (ALU) führt Befehle auf ganzen Zahlen (Integer-Arithmetik) u. a. grundlegende mathematische und logische Funktionen, Vergleiche sowie bitweise Operationen. Die *Gleitkommaeinheit* (Floating Point Unit, FPU) führt die Operationen der ALU auf Gleitkommazahlen aus. Die FPU kann zusätzlich spezielle Befehle für komplexe mathematische Funktionen (Logarithmus, trigonometrische Funktionen) unterstützen. In ähnlicher Weise steht die MUL/DIV-Einheit zur Berechnung für Multiplikationen und Divisionen zur Verfügung. Zumeist arbeiten die Befehle auf den lokalen *Registerspeichern* des Prozessors, d. h. Daten werden zuvor in die Register geladen.

Die *Statusregister* (z. B. Zero-, Carry- und Overflow-Flag) werden durch Befehle gesetzt und beinhalten Informationen, die während des Programmablaufs genutzt werden. *Coprozessoren* können eingesetzt werden, um weitere anwendungsspezifische Befehle auszuführen. *Timer* stellen Taktsignale zur Zeitmessung bereit. *Interrupts* und das *Debug-Interface* stehen für die Kommunikation des Prozessors z. B. mit dem Betriebssystem zur Verfügung.

Eine oder mehrere *Load/Store-Einheiten* (Load/Store-Unit, LSU) verbinden den Kern mit dem Speicher oder auch zu anderen Kernen. *Befehls-* und *Datencache* sind direkt mit der LSU verbunden und dienen als Zwischenspeicher, um einen schnelleren Zugriff auf Instruktionen bzw. Daten im externen Speicher zu erreichen, die häufig oder kurzfristig abgerufen werden. Dabei können auch mehrere Caches hierarchisch organisiert sein und unterschiedliche Zugriffszeiten aufweisen. Des Weiteren können Prozessoren einen *lokalen Befehls-* und *Datenspeicher* besitzen, deren Zugriffe die gleiche Latenz wie Caches haben, aber vom Anwendungsprogramm gesteuert werden. Die *Speicherverwaltungseinheit* (Memory Management Unit, MMU) wird zusammen mit dem *Translation Lookaside Buffer* (TLB) für die Bereitstellung des virtuellen Speichers und für die Kommunikation über das *Verbindungsnetzwerk* genutzt.

4.1.2 Optimierungstechniken

Konfigurierbare und erweiterbar Prozessoren, wie sie im Abschnitt 4.1.1 eingeführt wurden, können auf vielfältige Art und Weise im Hinblick auf Durchsatz und Leistungsverbrauch für eine bestimmte Applikation optimiert werden. Dieser Abschnitt diskutiert das Potential und die Einschränkungen für die in dieser Arbeit verwendeten Methoden zur Anpassung von Prozessoren.

Ein vielversprechender Ansatz zur Optimierung eines Prozessors ist die Erweiterung des Basis-Instruktionssatzes durch *Instruktionssatzerweiterungen* (ISE). Solche Befehle

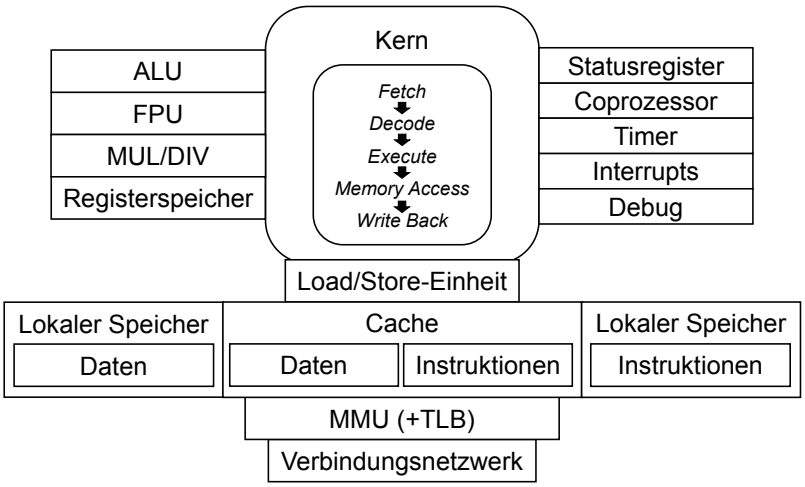


Abbildung 4.1: Allgemeines Prozessormodell

bestehen aus mehreren kombinierten Operatoren, die häufig ausgeführte und rechenintensive Teile des Programmcodes ersetzen und als neue Hardware im Prozessor integriert werden. Damit wird eine Beschleunigung und eine Reduzierung des Energiebedarfs der Applikation erreicht. Dieser Ansatz lässt sich wie bereits in Kapitel 1.4 skizziert in die Parallelität auf arithmetisch-logischer Ebene einordnen. Ein bekanntes Beispiel ist der Multiply-Accumulate-Befehl, der eine Multiplikation und eine Addition in einer Instruktion verknüpft und standardmäßig in modernen Prozessoren vorhanden ist. Weitere Beispiele sind die Berechnung eines Cyclic-Redundency-Check-Wertes (CRC), der Schiebe-, Vergleichs und XOR-Operationen enthält, oder das Vertauschen von einzelnen Bits eines 32-Bit Wortes. Letzteres benötigt kaum zusätzlichen Hardwareaufwand für die ISE, da nur logische Verbindungen der Register vertauscht werden müssen.

Für diese vielversprechende Methode existieren jedoch zwei grundlegende Probleme. Erstens: je mehr Operationen zu einer Instruktion verknüpft werden, desto höher ist deren Spezialisierung bzw. desto weniger wird sie möglicherweise verwendet. Der Entwickler muss demnach einen Kompromiss zwischen Durchsatz und Hardwareaufwand bzw. Chipfläche eingehen. Der zweite Punkt bezieht sich auf den kritischen Pfad in der Prozessorhardware. Das sequentielle Zusammenschalten mehrerer Operatoren in einer Instruktion führt zu längeren kombinatorischen Pfaden. Um nicht die Taktfrequenz zu reduzieren und damit die Leistungsfähigkeit des gesamten Prozessors einzuschränken, können Pipelinestufen innerhalb der Befehle eingeführt werden. Ein oder mehrere Registerstufen zwischen den einzelnen Operatoren einer Instruktion reduzieren den kritischen Pfad, führen jedoch zu einer erhöhten Komplexität der Hardware und zu größeren Latenzen beim Ausführen in Software. Letzteres kann vor allem in kurzen Schleifen, Rekursionen oder bei Pipelinehazards eher einen Anstieg der Ausführungszeit verursachen. Abschnitt 4.2.4 erläutert dazu ein detaillierteres Beispiel.

Ein Ansatz für die Parallelität auf Datenebene ist Single Instruction, Multiple Data (SIMD), das die parallele Verarbeitung mehrerer unterschiedlicher Daten mit einem Befehl bezeichnet [46]. Im einfachsten Fall sind die nebenläufig ausgeführten Berechnungen unabhängig voneinander. Zum Beispiel erzeugt die n-fache Addition zweier Vek-

toren n zusätzliche Addierer. Der Flächenbedarf wächst damit linear mit der SIMD-Breite. Sind jedoch diagonale Abhängigkeiten zwischen den einzelnen Vektorelementen auszuführen (z. B. Vollvergleich zweier Vektoren bei der Sorted-Set Intersektion), wächst der Flächenaufwand mit n^2. Die in dieser Arbeit entwickelte Befehlssatzerweiterung enthält SIMD-Instruktionen, die mehrere Operatoren verknüpfen und kombiniert so die erläuterten Methodiken.

Eine weitere Optimierungstechnik ist die Parallelisierung auf Befehlsebene. Dabei werden mehrere voneinander unabhängige Instruktionen gleichzeitig ausgeführt [124]. Superskalare Prozessoren führen ein dynamisches Scheduling der im Programm sequentiell vorliegenden Instruktionen durch, um Abhängigkeiten zu identifizieren. Im Gegensatz dazu, erfolgt das Scheduling parallel abzuarbeitender Instruktionen in VLIW-Prozessoren (Very Large Instruction Word) zur Compilierzeit. Die Instruktionen werden in einem VLIW-Befehlswort gruppiert und damit explizit auf die Funktionseinheiten des Prozessoren zugewiesen. Das statische Scheduling in VLIW-Prozessoren ist damit einfach umsetzbar, erlaubt aber z. B. keine angepasste Codeausführung bei bedingten Sprüngen oder Cache-Misses [99].

Um die Parallelität auf Daten- und Befehlsebene zu realisieren, sind auch die dafür notwendigen Speicherbandbreiten bereitzustellen. Das betrifft zum einen die in SIMD-Befehlen abzuarbeitenden Daten, die aus dem Datenspeicher zu laden sind. Zum anderen sollte auch die Schnittstelle zum Befehlsspeicher das gleichzeitige Laden mehrerer Instruktionen in das VLIW-Befehlswort unterstützen. Die Optimierungstechniken implizieren deshalb auch eine Anpassung der Prozessorarchitektur. Dazu gehören z. B. die Anzahl der Load/Store- und Ausführungseinheiten, die Speicherbandbreite oder die Anzahl der Pipelinestufen. In Abschnitt 4.2.3 werden dazu die konkreten Architekturerweiterungen des DBA ASIP angesprochen.

4.1.3 Stand der Technik

Wie bereits in Abschnitt 4.1.1 eingeführt, enthält jede Prozessorarchitektur standardmäßig eine Basis-ISA. Beispiele sind die weit verbreitete x86-ISA von Intel und AMD, der ARM-Befehlssatz [7], die MIPS ISA [70] oder die quelloffene RISC-V ISA [153]. Als Befehlssatzerweiterungen stellt Intel z. B. die *Streaming SIMD Extensions* (SSE) [130] vor und die ARM-ISA enthält zusätzliche Thumb und NEON Instruktionen, um die Programmgröße zu verringern bzw. SIMD-Befehle einzuführen [6]. Diese Ansätze basieren auf dem Konzept, neben der Basis-ISA vorgefertigte ISE für kombinierte arithmetische und logische Operationen und SIMD-Verarbeitung zu integrieren. Die Autoren von [135] geben einen Überblick über diese Multimediaerweiterungen für General-Purpose Prozessoren. Viele Arbeiten folgen diesem Konzept, um Datenbankalgorithmen zu beschleunigen. Schlegel u. a. [130] nutzen Intels SSE4.2-Instruktionen für die Verarbeitung der Sorted-Set Intersektion. Der genutzte 8-fach SIMD-Ansatz zeigt eine bis zu 5,3× beschleunigte Ausführung des Algorithmus gegenüber der skalaren Lösung. In ähnlicher Weise verwenden Zhou u. a. [167] Intels SSE2-Instruktionen für die Beschleunigung von Scan, Aggregation, Indexing und Join. Für einen 4-fach SIMD-Ansatz ergibt sich ein Speedup von mehr als Faktor vier im Vergleich zur nicht parallelen Ausführung, da die SIMD-Instruktionen zusätzliche Verzweigungen im Code entfernen und so falsche Sprungvorhersagen eliminie-

ren. Satish u. a. [128] nutzt die Intel SSE2-Instruktionen *min/max* und *shuffle*, um ein Sortiernetzwerk für das Sortierverfahren Merge-Sort aufzubauen. Damit können gleichzeitig zweimal vier 32-Bit Elemente sortiert werden und es ist eine fast optimale Leistungssteigerung von Faktor 3,8 gegenüber der skalaren Version möglich. Die Ansätze erreichen die erwartete Beschleunigung, erlauben aber nicht die Reduzierung des Energieverbrauchs um mehrere Größenordnungen, wie sie in dieser Arbeit mit einem anwendungsspezifischen ASIC und ASIP erzielt werden.

Darüber hinaus sind vielfältige kommerzielle Entwicklungstools erhältlich, mit denen anwendungsspezifische Prozessoren erzeugt werden können, deren vollständige Architektur inklusive des Befehlssatzes an die Applikation angepasst ist. Beispiele solcher Tools sind LISA [73], CodAL [134], APEX [137] oder TIE [28]. Diese Ansätze stellen vordefinierte RISC-ähnliche Prozessoren sowie die zugehörigen Compiler, Simulatoren und Generatoren zur Erzeugung des Prozessors auf Register Transfer Level (RTL) bereit. Mit der *ARC Processor Extension* (APEX) können mit Hilfe der Verilog Hardwarebeschreibungssprache benutzdefinierte Hardware, Instruktionen und Register zum Prozessor hinzufügt werden. In ähnlicher Weise arbeitet die von Cadence Tensilica eigens entwickelte Beschreibungssprache *Tensilica Instruction Extension* (TIE), mit der Register, Instruktionen und Prozessorschnittstellen erweitert werden können. Diese Arbeit nutzt den TIE-Ansatz für die Entwicklung des Datenbankbeschleunigers als ASIP (siehe Abschnitt 4.2). Die *Language for Instruction-Set Architecture* (LISA) und die *Codasip Architecture Language* (CodAL) ermöglichen die Beschreibung der gesamten Prozessorarchitektur auf einem hohen Abstraktionsniveau. Die Sprachkonstrukte unterstützen die Erzeugung eines Speichermodells (Register, Bitbreiten), Hardwareressourcen (Operationen), das Verhaltens- und Timingmodell der Hardwareoperationen sowie Peripherie und Schnittstellen. Beispielsweise nutzt die Arbeit in [42] die Tools von Codasip. Ein Basisprozessor mit einer MIPS-Architektur wird mit Befehlen für eine bitweise Rotation und mehreren verknüpften logischen Operatoren erweitert, um einen *Secure Hash Algorithm* (SHA-3) zu beschleunigen. Die Berechnungszeit von SHA-3 kann so um bis zu 43 % reduziert werden.

Während LISA und CodAL sowohl die Prozessorarchitektur als auch den Instruktionssatz beschreiben, fokussiert sich die Beschreibungssprache *nML* nur auf den Befehlssatz [44]. Dazu gehört das Definieren von Registern, Datentypen, Befehlen aber auch das Speichermodell. Ursprünglich um 1990 von der Technischen Universität Berlin entwickelt, findet nML heute zusammen mit LISA Anwendung in dem Synopsys ASIP-Designer [138].

In der Arbeit von Wu u. a. [160] wird eine entwickelter Instruktionssatz vorgestellt, der keinen Basisprozessor benötigt. Die Instruktionen sind in der sogenannten *Q100* Datenbank-Verarbeitungseinheit integriert. Q100 enthält anwendungsspezifische Hardwareblöcke für die energieeffiziente Abarbeitung von SQL-Operatoren. Der Zugriff auf die einzelnen Hardwarekomponenten erfolgt in einer Simulationsumgebung, die Software mit der Q100-ISA ausführt. Im Vergleich zu einem single-thread Intel Xeon Prozessor, erreichen die mit Q100 implementieren TPC-H Anfragen eine Leistungssteigerung um den Faktor 10.

4.2 Implementierung des ASIP

4.2.1 Verwendete Entwicklungsumgebungen und Tools

Die Entwicklungsmethodik des in dieser Arbeit entworfenen DBA ASIP besteht aus der Kombination von VLSI-Werkzeugen von Cadence Tensilica und Synopsys. Die notwendigen Lizenzen sowie das zugehörige Know-how dieser Tools sind bereits am Lehrstuhl für Mobile Nachrichtensysteme vorhanden und können deshalb direkt zum Einsatz kommen. Des Weiteren werden konfigurierbare Prozessoren von Cadence Tensilica (Xtensa LX) verwendet, deren Systemarchitektur und Instruktionssatz erweitert werden kann (oranger bzw. grüner Kasten in Abbildung 4.2). Die Leistungsmerkmale dieser Systeme wurden mit Hilfe zahlreicher Benchmarks von Start-up-Unternehmen des Lehrstuhls als sehr positiv bewertet und finden deshalb auch in dieser Arbeit Verwendung. Zu den erweiterbaren Architekturkomponenten eines Xtensa Prozessors zählen die lokalen Datenspeicher (Größe, Bitbreite), zusätzliche Load/Store-, Gleitkomma- und Speichermanagementeinheiten, sowie andere Konfigurationsparameter wie die Anzahl der Pipelinestufen, Interrupts und Debug Optionen. Andererseits erlauben die Xtensa Prozessoren die Entwicklung und Integration anwendungsspezifischer Instruktionen, um die zuvor beschriebenen Einheiten und Erweiterungen effizient zu nutzen.

Abbildung 4.2 zeigt den allgemeinen Designprozess. Zu Beginn wird mit Hilfe eines zyklenakkuraten Simulators ein Profil des zu beschleunigenden Algorithmus erstellt, um häufig ausgeführte und rechenintensive Codeabschnitte zu identifizieren. Des Weiteren lassen sich damit algorithmische Eigenschaften und Anforderungen an die Speicherzugriffe analysieren. Tensilica stellt dazu den Simulator, Compiler und Debugger in der Entwicklungsumgebung *Xtensa Xplorer* bereit. Der Prozessor-Generator erzeugt das Simulationsmodell des erweiterten Prozessors und stellt intrinsische Funktionen für die neu entwickelten Instruktionen bereit, die die zuvor identifizierten rechenintensiven Codeabschnitte ersetzen. Das generierte Prozessormodell wird zusammen mit den erweiterten Hardwareeinheiten auf Funktionalität getestet. Durch ein erneutes Profiling können die erwartete Performanz überprüft und evtl. weitere Codeabschnitte zur Beschleunigung identifiziert werden.

Sind die Optimierung des Algorithmus ausgeschöpft, folgt im nächsten Schritt die Generierung eines synthetisierbaren Modells auf *Register-Transfer-Level* (RTL) durch den *Hardware Description Language* (HDL) Prozessor-Generator. Synthese und Place & Route werden mit Synopsys Design Compiler bzw. IC Compiler ausgeführt. Die Syntheseergebnisse liefern Aussagen über das Zeitverhalten, sowie den Flächen- und Leistungsverbrauch des Prozessors. Entsprechen die Ergebnisse nicht den Anforderungen, müssen die neu erzeugten Instruktionen überarbeitet oder die Synthesebedingungen angepasst werden. Zum Beispiel verkürzt das Einfügen von zusätzlichen Registerstufen innerhalb der Instruktionen die kombinatorischen Pfad (kritischen Pfade) und erlaubt damit eine höhere Taktfrequenz des Prozessors. Während der Zeitaufwand für den Iterationszyklus beim Entwickeln der Instruktionen bis zum Simulationsmodell im Minutenbereich liegt, benötigt das Generieren des RTL-Codes und die Synthese mehrere Stunden.

Cadence Tensilica stellt eine eigene Hardwarebeschreibungssprache, genannt *Tensilica Instruction Extension* (TIE), bereit [28]. Die neuen Instruktionen werden vom Compiler in

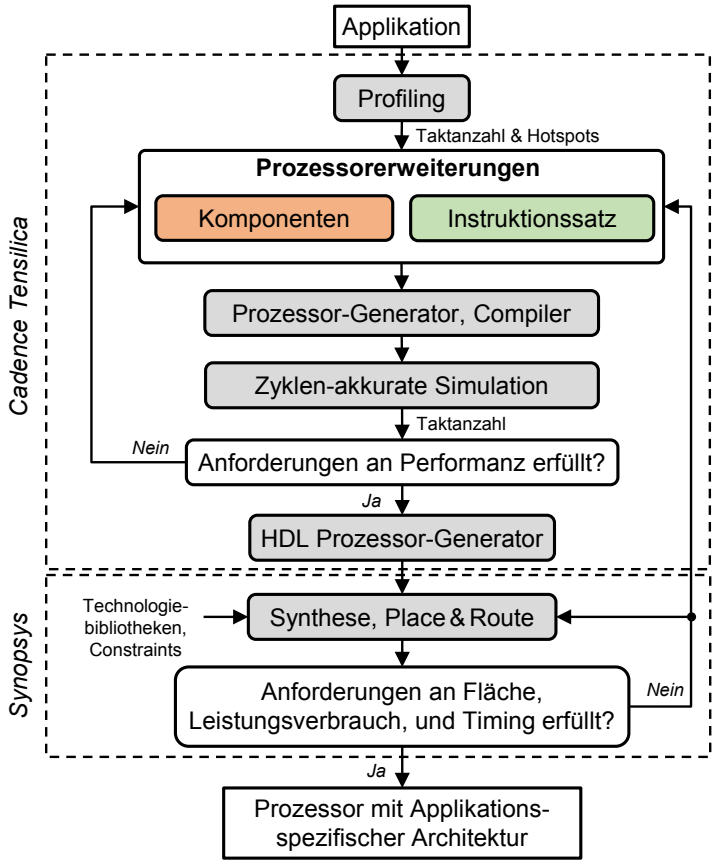


Abbildung 4.2: Entwicklungsablauf des ASIP-Designs: die Architektur- und Instruktionssatzerweiterungen des Xtensa Prozessors sind in orange bzw. grün gekennzeichnet, graue Kästen stellen die zugehörigen Tools dar.

ein bis zu 128 Bit breites VLIW-Format gepackt (Flexible Length Instruction Extension, FLIX) und damit auf mehrere Ausführungseinheiten zugewiesen. Im Unterschied zu superskalaren Prozessoren findet die Zuweisung in der VLIW-Architektur nicht zur Laufzeit sondern statisch statt. Da die benutzerdefinierten Instruktionen bereits bei der Entwicklung hinsichtlich einer festgelegten Ausführung (Reihenfolge und Parallelität) optimiert werden, ist keine dynamische Zuweisung erforderlich.

Neben den bereits vorhandenen Prozessorregistern gibt es zwei weitere Arten von benutzerdefinierten Registerspeichern: TIE-Register und TIE-States. Die Anzahl der Instanzen und die Registerbreite ist frei konfigurierbar und genauso wie die Größe des VLIW-Formats durch die Breite des Daten- bzw. Befehlsspeichers limitiert. Während TIE-Register typische Allzweckregister darstellen, sind TIE-States zum Speichern von anwendungsspezifischen Daten vorzuziehen. States erlauben einen Lese- und Schreibzugriff im gleichen Taktzyklus.

Abbildung 4.3a zeigt die Definition des TIE-Registers `reg32`, des TIE-States `div_value` und der Instruktion `add_div32`. Im Beispiel werden acht 32-Bit Register generiert, die von allen Instruktionen verwendet werden können. Im Gegensatz zu TIE-States, werden die Register-Allokation sowie die Lade- und Speicheroperatio-

```
regfile reg32     32 8 r32
state   div_value  4 add_read_write

operation add_div32
         {out reg32 out1,
          in reg32 in1,
          in reg32 in2}
         {in div_value}
{
    assign out1 = (in1 + in2) >> div_value;
}
```

(a) TIE-Code

```
int a, b, c;

reg32 *r_a = (reg32*)a;
reg32 *r_b = (reg32*)b;
reg32 *r_c = (reg32*)c;
WUR_div_value(1);

*r_c = add_div32(*r_a, *r_b);
```

(b) C-Code

Abbildung 4.3: TIE-Code Beispiel mit zugehörigem C-Code

nen vom Compiler übernommen. Für das 4 Bit breite State-Register übernimmt der Programmierer die Verantwortung für den Registerinhalt. Das optionale Schlüsselwort `add_read_write` erlaubt den Zugriff auf den Inhalt des States vom C-Code. Die Definition der Instruktion wird durch das Schlüsselwort `operation` eingeführt. Danach folgt der Befehlsname, die Argumentenliste und die funktionale Beschreibung der Instruktion. Ein- und Ausgangsargumente können TIE-spezifische Register, States, Konstanten und Interfaces sein. Mit den Schlüsselwörtern `in` und `out` wird spezifiziert, ob vom jeweiligen Argument gelesen oder geschrieben wird. Ähnlich zur Hardwarebeschreibungssprache Verilog, weist das Schlüsselwort `assign` berechnete Werte auf die Ausgangsoperanden zu. Die Instruktion kann in einem Takt, oder falls notwendig, in mehreren Taktzyklen ausgeführt werden, um die Taktfrequenz des Prozessors nicht zu beeinflussen.

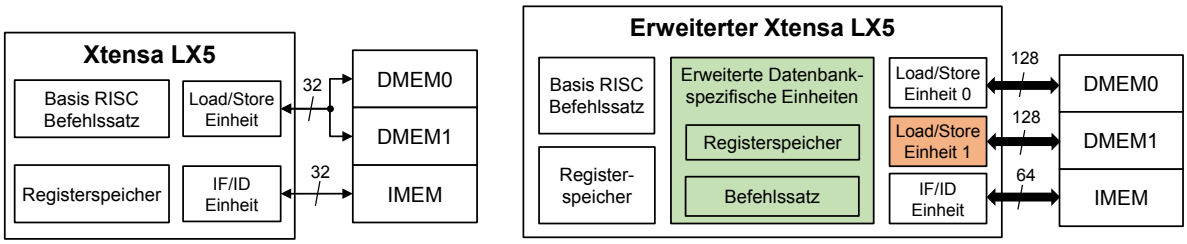
Der TIE-Compiler generiert intrinsische Funktionen, um auf die Register und Instruktionen vom C-Code zuzugreifen (Abbildung 4.3b). Die explizite Typumwandlung der Integer-Variablen ist notwendig, da die Instruktion `add_div32` Argumente vom Typ `reg32` benötigt. Im Beispiel hat `add_div32` nur ein Ausgangsregister, das gleichzeitig der Rückgabewert ist. Der Befehl `WUR_`*State-Name*`()` (Write User Register, WUR) beschreibt den TIE-State mit dem übergebenen Wert.

Das Beispiel in Abbildung 4.3 kann leicht auf SIMD-Funktionalität erweitert werden. Eine Erhöhung der Registerbreite auf 128 Bit, würde die parallele Berechnung von vier Werten erlauben. Dies setzt voraus, dass die Breite des Datenspeichers ebenfalls 128 Bit beträgt.

4.2.2 Basisprozessor

Wie bereits in Abschnitt 4.2.1 erläutert, wird ein Cadence Tensilica Xtensa LX5 Prozessor[1] als Grundlage des DBA ASIP genutzt. Der Basisprozessors ist in Abbildung 4.4a dargestellt und beinhaltet lediglich die notwendigsten Hauptkomponenten. Dazu zählen der Basis-RISC Instruktionssatz und Registerspeicher, sowie die Load/Store-Einheit und

[1]Diese fünfte Version des Xtensa LX Prozessors war zum Zeitpunkt der Entwicklung des DBA ASIP die aktuellste von Cadence Tensilica bereitgestellte Hardwarearchitektur.



(a) Basisprozessor (b) Erweiterter Prozessor

Abbildung 4.4: Verwendete Prozessormodelle des Cadence Tensilica Xtensa LX5

Befehlsabruf- und Dekodiereinheit (Instruction Fetch/Instruction Decode, IF/ID). Die Größe der Speicher ist variabel, zunächst aber auf 32 kB jeweils für DMEM0 und DMEM1 und auf 64 kB für den Befehlsspeicher festgelegt. Die Datenbreite beider Speicherschnittstellen beträgt 32 Bit. Da nur eine Load/Store-Einheit vorhanden ist, kann nur einer der beiden Datenspeicher im gleiche Taktzyklus erreicht werden.

Der Prozessor arbeitet ausschließlich auf den lokalen Speichern. Für die Integration in ein Mehrkernsystem, ist die Anbindung der lokalen Speicher über eine DMA-Einheit an das Verbindungsnetzwerk des MPSoC erforderlich. Diese Implementierungen und Ergebnisse werden in Kapitel 5 präsentiert.

4.2.3 Erweiterter Prozessor

In einem ersten Schritt wird die Architektur des LX5-Prozessors erweitert, um die Ausführung der Datenbankalgorithmen zu beschleunigen. Die vorgenommenen Erweiterungen schöpfen dabei die von Cadence Tensilica bereitgestellten Konfigurationsmöglichkeiten vollständig aus. Die konkreten Erweiterungen sind:

- Das Hinzufügen einer zweiten Load/Store-Einheit ermöglicht den gleichzeitigen Zugriff auf beide lokale Datenspeicher innerhalb eines Prozessortaktes. Die zu verarbeitenden Daten können damit auf die Speicher verteilt und damit die Speicherbandbreite verdoppelt werden.
- Das Erhöhen der Busbreite des Datenspeichers W_{DMEM} erlaubt das Anwenden von SIMD-Befehlen, um die Algorithmen zu beschleunigen. Nachfolgend werden Bitbreiten von 128, 256 und 512 Bit untersucht.
- Um eine weitere Beschleunigung der Algorithmen durch eine Parallelität auf Befehlsebene zu erreichen, wird das Interface zum Befehlsspeicher auf $W_{IMEM} = 64$ Bit erhöht. Es kann damit ein 64-Bit VLIW mit bis zu drei Befehlsslots erzeugt werden. Die Steuerung der zwei Load/Store-Einheiten erfolgt dann z. B. durch das Zuweisen von zwei Ladeinstruktionen in ein VLIW.

Der zweite Schritt umfasst die Entwicklung des erweiterten Befehlssatzes. Die neuen Befehle werden als zusätzliche funktionale Hardwareeinheiten neben der ALU innerhalb der Execute-Stufe der RISC-Pipeline integriert. Abbildung 4.5 zeigt die ersten vier von insgesamt fünf Pipelinestufen des Xtensa Prozessors sowie eine detaillierte Ansicht der

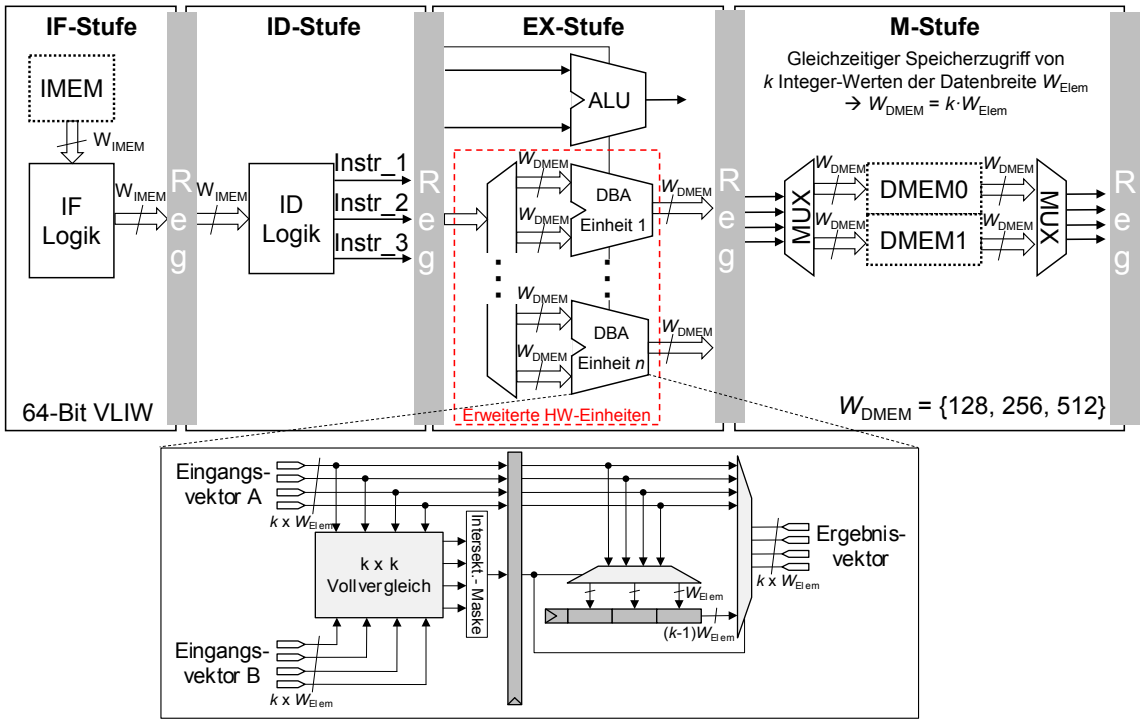


Abbildung 4.5: RISC-Pipeline des erweiterten Xtensa LX5 (DBA) mit zusätzlichen Hardwareeinheiten

Vergleichs- und Speicheroperation der Sorted-Set Intersektion. Zu Beginn wird in der *Instruction-Fetch-Stufe* (IF) ein 64-Bit Befehlswort aus dem Befehlsspeicher (IMEM) geladen und in der *Instruction-Decode-Stufe* (ID) dekodiert. In der *Execute-Stufe* (EX) werden die Daten je nach Befehl in der ALU oder in den DBA-Einheiten gleichzeitig verarbeitet. Der Datenzugriff findet in der *Memory-Stufe* (M) statt. Die Rückführung z. B. von Informationen aus Sprungbefehlen zur IF-Stufe ist Teil der *Write-Back-Stufe* (nicht in Abbildung 4.5).

4.2.4 Befehlssatzerweiterung

Die Architektur des erweiterten Prozessors erlaubt die Ausnutzung der Parallelität auf Daten- und Befehlsebene. Dazu sollen nun Befehlssatzerweiterungen für die drei Sorted-Set Algorithmen Intersektion, Vereingung und Differenz, sowie für das Merge-Sort und das Hashing entwickelt werden, um zusätzlich eine Parallelität auf Bit- bzw. arithmetisch-logischer Ebene zu erreichen. Die Möglichkeiten zur Parallelisierung der Algorithmen wurden in Kapitel 2.2.3 vorgestellt und diskutiert. Das Ziel ist es, die Befehlssatzerweiterung derart aufzubauen, sodass die parallelen Algorithmen effizient auf die erweiterte Prozessorarchitektur abgebildet werden können. Im Folgenden wird die Herangehensweise erläutert, wobei der in Abbildung 4.2 gezeigte Entwicklungsablauf weiterhin als Grundlage dient.

Allgemeiner Aufbau des angepassten Codes

Aus den Untersuchungen der parallelen Algorithmen aus Kapitel 2.2.3 geht eine Gemeinsamkeit hervor. Die Schleifen im Code führen beim Durchlaufen der Mengen kontinuierlich die folgenden Schritte aus: (1) Laden, (2) Verarbeiten, (3) Speichern. Dabei kann das Ergebnis aus dem Verarbeitungsschritt (2) den Zugriff auf den Speicher beeinflussen, indem z. B. Speicheradressen geändert oder das Laden für eine bestimmte Anzahl von Iterationen ausgesetzt wird. Die Hauptschleife für den angepassten C-Code mit den erweiterten Befehlen folgt deshalb dieser geradlinigen Struktur und ist im folgenden Codeausschnitt zu sehen:

```
init();

loadA_incr(); loadB_incr();
while(store_cmp()){
    loadA_incr(); loadB_incr();
}
```

Sie bestimmt maßgeblich das Zeitverhalten des gesamten Algorithmus und muss deshalb bzgl. der benötigten Taktanzahl pro Schleifendurchlauf optimiert sein. Zu Beginn erfolgt eine Initialisierung der Register und TIE-States sowie das initiale Setzen der Start- und Endadressen für die Ein- und Ausgangslisten (`init`). Danach laden die TIE-Operationen `loadA_incr` und `loadB_incr` jeweils k Elemente aus den lokalen Speichern und schreiben diese in die internen State-Register des Prozessors. Das VLIW-Format des Xtensa Prozessors und die beiden Interfaces zum Datenspeicher erlauben das gleichzeitige Ausführen der beiden Instruktionen (Parallelität auf Befehlsebene). Zusätzlich werden in den Ladeoperationen die Adressregister inkrementiert, um beim nächsten Funktionsaufruf die nächsten k Elemente laden zu können. In `store_cmp` erfolgt die Verarbeitung der nun in den Registern enthaltenen Elemente sowie das Abspeichern der Ergebniselemente. Es hat sich als vorteilhaft herausgestellt, die Verarbeitung und das Abspeichern in einer Instruktion zu kombinieren, um die Anzahl der Takte pro Iteration zu reduzieren. Die Trennung von Verarbeitungslogik und das Ansprechen des Speicherinterfaces erfolgt dann durch eine Pipeline innerhalb der Instruktion. Des Weiteren steuert `store_cmp` mit Hilfe des Rückgabewertes die while-Schleife, indem die Adressregister aus den Ladeoperationen mit den Endadressen der Listen verglichen werden. Durch das direkte Ausnutzen des Rückgabewertes in der Schleifenbedingung wird vermieden, zusätzliche lokale Variablen im C-Code anzusprechen. Weitere TIE-States erlauben der Instruktion `store_cmp` die Ladebefehle zu steuern, d. h. entsprechend dem Verarbeitungsergebnis den Speicherzugriff zu beeinflussen. Darüber hinaus ist eine Aufteilung von `store_cmp` ähnlich der Ladebefehle möglich, um gleichzeitig Daten in beiden Datenspeichern abzulegen. Insgesamt benötigt die Hauptschleife drei Takte für eine Iteration: 1. Laden, 2. Verarbeiten und Speichern, 3. Überprüfen der Schleifenbedingung. Es folgen nun weitere notwendige Betrachtungen im Zusammenhang mit der gezeigten Hauptschleife.

Loop Unrolling Die Hauptschleife weist eine entweder minimale bzw. eine zuvor bekannte Anzahl an Iterationen auf, die jeweils durch die Kardinalitäten der Listen und

die Laufzeitkomplexitäten der Algorithmen gegeben sind. Eine for-Schleife, dessen Iterationszahl bereits zur Compilierzeit feststeht, wird dann vor die while-Schleife gesetzt bzw. ersetzt diese komplett. Der benötigte Takt zum Abfragen der Schleifenbedingung entfällt damit.

Behandlung von Datenabhängigkeiten Der als SRAM integrierte lokale Speicher des Xtensa LX5 besitzt eine Verzögerungszeit beim Lesen von einem Taktzyklus. Die geladenen Daten der TIE-Operation `loadA_incr` bzw. `loadB_incr` stehen damit am Ende der M-Stufe in der Xtensa-Pipeline zur Verfügung. Sie werden im nächsten Takt allerdings bereits in der EX-Stufe der TIE-Operation `store_cmp` benötigt. Zu diesem Zeitpunkt sind die Daten noch nicht vorhanden (Daten-Hazard). Um das Einfügen eines Pipelinestalls zu verhindern, schreiben die TIE-Ladeoperationen die aus dem Speicher gelesenen Daten zunächst in ein Zwischenregister. Erst im darauffolgenden Takt werden die Daten aus diesem Register der TIE-Operation `store_cmp` übergeben. Weist der SRAM höhere Verzögerungszeiten beim Lesen der Daten auf, müssten entsprechend mehr Zwischenregister in den TIE-Operationen eingeführt werden.

Nachverarbeitung Genau wie beim Ablauf der Algorithmen, wird die Hauptschleife beendet, sobald das Ende der Eingangslisten erreicht ist. Danach müssen die sich noch in den Zwischenregistern befindenden Elemente verarbeitet und abgespeichert werden. Speziell für die Sorted-Set Vereinigung und Differenz muss sich eine weitere Schleife anschließen, die etwaige restliche Elemente aus den Eingangslisten in die Ergebnisliste kopiert. Bei Annahme von gleichmäßig verteilten Eingangsdaten ist der zeitliche Anteil der Nachverarbeitung im Vergleich zur Gesamtlaufzeit des Algorithmus vernachlässigbar. Eine Anpassung der Nachverarbeitung durch die Befehlssatzerweiterung ist deshalb nicht unbedingt notwendig. Lediglich können SIMD-Instruktionen für das Kopieren restlicher Elemente eingesetzt werden.

Bis jetzt wurden bereits die Möglichkeiten der Datenparallelität für die Algorithmen erläutert und der allgemeine Aufbau der Hauptschleife mit den erweiterten Instruktionen dargestellt. Die folgenden Abschnitte beschreiben nun die konkrete Umsetzung der in Abschnitt 2.2.3 eingeführten Algorithmen mit den Instruktionssatzerweiterungen.

Sorted-Set Algorithmen

Für die Sorted-Set Algorithmen ist bekannt, dass zwei Eingangs- und eine Ausgangsmenge existieren. Im Gegensatz zum DBA ASIC, umfasst der Xtensa LX5 nur zwei Datenspeicher (DMEM0 und DMEM1). Es können damit nur die zwei Eingangsmengen auf die Speicher verteilt werden, um deren gleichzeitigen Zugriff zu ermöglichen. Eine bestimmte Speicherzuordnung der Ergebnismenge hat keinen Einfluss auf die Leistungsfähigkeit. Für die folgenden Betrachtungen befindet sich diese Menge in DMEM1.

Gemäß der parallelen Realisierung der Sorted-Set Algorithmen aus Kapitel 2.2.3 werden gleichzeitig zwei Vektoren a und b mit jeweils k Elementen aus den Mengen A und B verarbeitet. Es ist leicht zu sehen, dass das Laden der Vektoren mit den Instruktionen `loadA_incr` und `loadB_incr` ausgeführt werden kann. Der Vergleich dieser Ele-

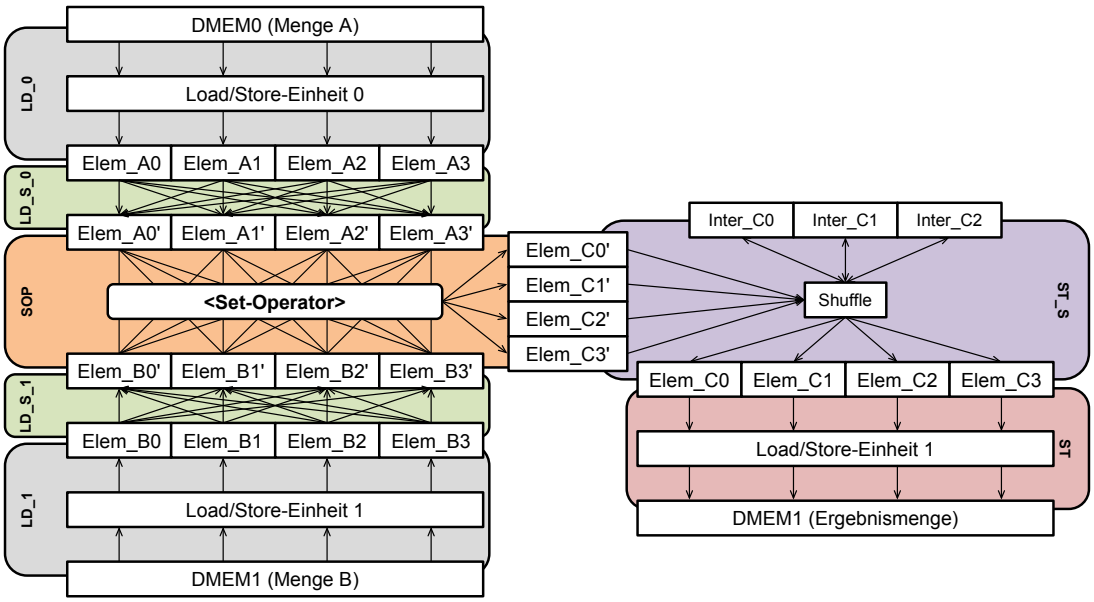


Abbildung 4.6: Lade-, Shuffle- und Speicheroperationen der Sorted-Set Algorithmen am Beispiel $k = 4$

mente und Abspeichern der Ergebniselemente erfolgt dann in `store_cmp`. Abbildung 4.6 zeigt beispielhaft den Datenfluss der Instruktionen und deren interne Operationen. Die Operationen `LD_0` bzw. `LD_1` laden jeweils k Elemente aus den Mengen A und B, die sich in DMEM0 bzw. DMEM1 befinden. Auf Grund des Speicheralignments werden immer genau k Elemente geladen. Um jedoch das partielle Laden der Vektoren a und b zu ermöglichen (siehe Abbildung 2.3), muss eine Möglichkeit für das Umsortieren der Elemente in den Vektoren bereitgestellt werden. Die ASIC-Implementierung aus Kapitel 3.2.1 nutzt dazu FIFO-Speicher. Allerdings stellt die Hardwarebeschreibungssprache TIE keine konkrete FIFO-Implementierung bereit. Aus diesem Grund erfolgt das Umsortieren in einem speziellen Shuffle-Netzwerk. Das Netzwerk ist als $k \times k$ Multiplexer realisiert, d. h. jedes Element des Eingangsregisters ist mit jedem Element des Ausgangsregisters verbunden. Die zugehörigen Shuffle-Operationen in Abbildung 4.6 sind `LD_S_0` bzw. `LD_S_1`. Die Operationen `LD_0` und `LD_S_0` bilden dann zusammen die Instruktion `loadA_incr` für DMEM0 bzw. äquivalent dazu `loadB_incr` für DMEM1.

Danach folgt der Vollvergleich von $k \times k$ Elementen in der `SOP`-Operation, der eine Bitmaske generiert, um die gültigen Ergebniselemente auszuwählen. Die zwei Ladeinstruktionen erhalten ebenfalls Zugriff auf diese in einem TIE-State gehaltenen Bitmaske, um die Anzahl der neu zu ladenden Elemente zu bestimmen sowie das Shuffle-Netzwerk zu konfigurieren. Die Ergebniselemente gelangen in ein weiteres Shuffle-Netzwerk (`ST_S`), um auch das Speicheralignment beim Schreiben in den lokalen Speicher zu beachten. Bei nur drei vorhandenen Ergebniselementen können diese in einem Zwischenregister gehalten werden und die Speicheroperation `ST` wird nicht ausgeführt. Die Operationen `SOP`, `ST_S` und `ST` ergeben zusammen die Instruktion `store_cmp`.

Die Register zwischen den einzelnen Operationen wirken als Pipelineregister, wodurch sich die in Abbildung 4.7 dargestellte Pipeline ergibt. Die Pipeline umfasst fünf Stufen, die über die zwei Takte der Lade- und Speicherinstruktionen verteilt sind. Die Daten

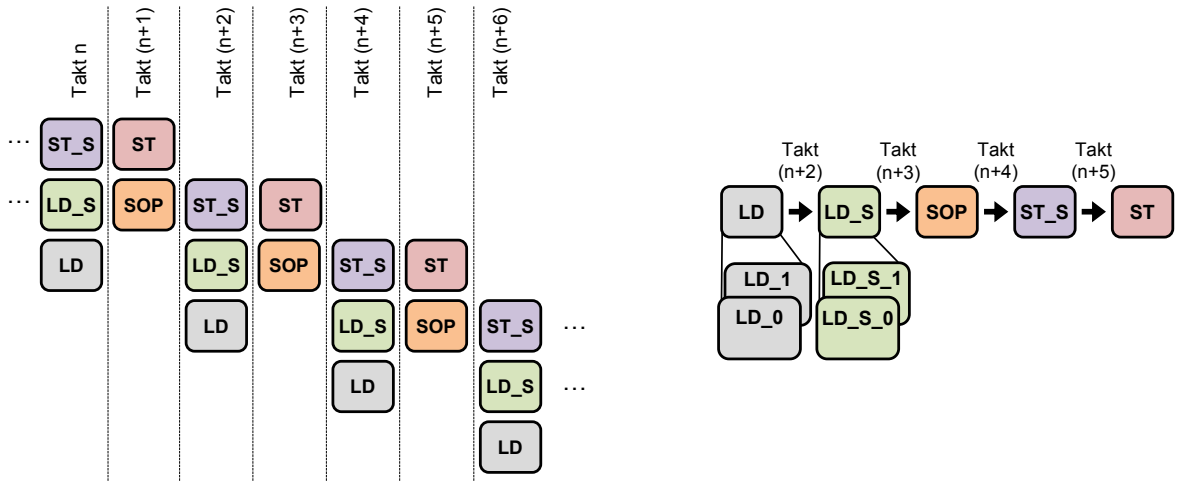


Abbildung 4.7: Ausschnitt der Pipeline für die Sorted-Set Algorithmen

haben deshalb die Software-Pipeline nach insgesamt sechs Takten durchlaufen. In jedem der zwei Takte werden abwechselnd $2k$ Elemente geladen oder k Elemente abgespeichert. Der maximal theoretische Durchsatz beträgt demnach k Elem./Takt und ist durch die Datenbreite des Speicherinterfaces begrenzt.

Das Ladeverhalten der Sorted-Set Vereinigung und Differenz ist identisch zur Intersektion, weshalb die gleichen Ladeinstruktionen `loadA_incr` und `loadB_incr` genutzt werden können. Der Unterschied befindet sich in `store_cmp` im Vollvergleich und bei der Auswertung der resultierenden Bitmaske. Da die Vereinigung die Ergebniselemente sowohl aus A als auch aus B auswählt, sind im duplikatfreien Fall $2k$ Elemente abzuspeichern. Ein vollständiger Vorteil des gleichzeitigen Ladens aus beiden Eingangsmengen entsteht demnach nur wenn A und B identisch sind ($sel_{AB} = 100\,\%$). Beim Kopieren restlicher Elemente in den Eingangsmengen in die Ergebnismenge, werden einfache SIMD-Lade- und Speicherinstruktionen genutzt, um pro Takt k Elemente zu kopieren.

Sortieren

Der in Kapitel 2.2.3 vorgestellte Ansatz für das parallele Merge-Sort besteht aus dem kontinuierlichen Abarbeiten des in Abbildung 2.4a dargestellten Merge-Vorgangs, wobei jeweils andere Teilfolgen unterschiedlicher Länge verarbeitet werden. Die Vorsortierung, um zunächst sortierte Datenblöcke der Länge k zu erzeugen, wird durch eine einfache Schleife mit SIMD-Lade- und Speicherinstruktionen implementiert. Um k Elemente in einem Takt zu sortieren, muss das Bestimmen der Reihenfolge für jedes Element unabhängig erfolgen. Dazu wird jedes Element a_i $(i = 0, ..., k-2)$ mit jedem Element a_j $(j = 1, ..., k-1)$ verglichen, ob $a_i < a_j$ gilt $(i < j)$. Das ergibt insgesamt $\frac{1}{2}k(k-1)$ parallele Vergleiche. Anhand der Anzahl der positiven Vergleiche von jedem Element kann dessen finale Position berechnet werden. k parallele Schieberegister ordnen dann die Elemente entsprechend der ermittelten Positionen an. Ist z. B. das Element a_0 größer als a_1, aber kleiner als alle anderen $k-2$ Elemente, ist a_0 das zweitkleinste Element.

Die drei Abschnitte des Merge-Vorgangs (Kopieren in Hilfsarray, Verschmelzen,

Zurückkopieren) lassen sich nun jeweils auf die Hauptschleife mit erweiterten Instruktionen zurückführen. Das Kopieren zu Beginn und am Ende kann über einfache SIMD-Lade- und Speicheroperationen realisiert werden. Das Verschmelzen entspricht der Vereinigung, bei der Duplikate beibehalten werden. Das bedeutet, dass die zu ladende Elementanzahl konstant bleibt und damit im Unterschied zu den Ladeinstruktionen der zuvor beschriebenen parallelen Sorted-Set Algorithmen beim Merge-Sort keine Shuffle-Netzwerke notwendig sind. Von den immer $2k$ vorhandenen Elementen werden die k kleineren abgespeichert und die k größeren Elemente gelangen mit den k neu geladenen Elementen in den folgenden Vergleich der nächsten Iteration. Die Verarbeitung erfolgt ähnlich der Vorsortierung. Es müssen zwar $2k$ Elemente pro Takt sortiert werden, die k^2 Vergleiche benötigen, jedoch sind bereits jeweils k Elemente sortiert, sodass im Vergleich zur Vorsortierung $k(k-1)$ Vergleiche eingespart werden. Das Ausnutzen beider Datenspeicher ist nicht möglich, da wie zuvor bei der Vereinigung beschrieben das gleichzeitige Laden aus DMEM0 und DMEM1 nur bei einer hohen Anzahl von Duplikaten und deren Eliminierung vorteilhaft ist. Das Merge-Sort kann damit nicht die volle Speicherbandbreite ausnutzen und wird stets durch die Verarbeitungszeit begrenzt.

Hashing

Die in Kapitel 2.2.2 vorgestellte Hashfunktion weist prinzipiell keine Datenabhängigkeiten bei einer Parallelisierung auf Datenebene auf. Die Herausforderung liegt in der Implementierung der inneren Schleife, die über alle $w-1$ Bits iteriert (vgl. Abbildung 2.5). Jede Iteration verschiebt den Hashwert entsprechend der Hashmaske teilweise um ein Bit nach rechts und erzeugt damit voneinander abhängige Iterationen. Eine mögliche Instruktion könnte z. B. nur eine Iteration dieser Schleife ausführen und dabei den teilweisen Rechts-Shift integrieren. Mit diesem Ansatz würde die Verarbeitung $\frac{(w-1)N}{k}$ Schritte benötigen. Um jedoch die Abhängigkeit zur Bitbreite w zu eliminieren, muss die Instruktion die gesamte Schleife integrieren und dabei die einzelnen Schiebeoperationen unabhängig voneinander ausführen. Das setzt voraus, dass die finale Position nicht erst aus der Hashmaske berechnet werden darf, sondern zuvor bekannt sein muss. Aus diesem Grund wird vor dem eigentlichen Hashing per Software eine $n \log w$ breite *Permutationsmaske* generiert, die die n Positionen der gesetzten Bits angibt. Ein Shuffle-Netzwerk nutzt die resultierende Permutationsmaske und verschiebt die ausgewählten Bits, sodass ein n Bit breiter Hashwert entsteht. Die Hashfunktion ist zur Laufzeit konfigurierbar, da die Permutationsmaske je nach Änderung der Hashmaske immer neu generiert werden kann. Abbildung 4.8 zeigt den erläuterten Ablauf des Hashings, der innerhalb eines Takts ausgeführt wird. Ausgehend von der allgemeinen Voraussetzung $n \leq w$, wird im Folgenden ebenfalls $n \leq \frac{w}{2}$ angenommen. Diese Annahme ist sinnvoll, da mit zu vielen selektierten Bits die Hashwerte die Verteilung der Eingangsdaten annehmen. Die Hashfunktion sollte jedoch unabhängig von den Eingangsdaten eine geeignete Lastverteilung der Hashtabelle erzeugen. Die Datenbreite der Hashwerte beträgt deshalb maximale die Hälfte der Schlüsselbreite.

Im Gegensatz zu den Sorted-Set Algorithmen ist beim Hashing nur eine Eingangsliste abzuarbeiten. Um trotzdem den Vorteil der beiden Datenspeicher auszunutzen, wird ein Teil der Eingangsliste mit den Schlüsseln und ein Teil der Ergebnisliste, die die Hashwerte enthält, auf DMEM0 und DMEM1 verteilt. Damit erlauben die beiden Datenspeicher das

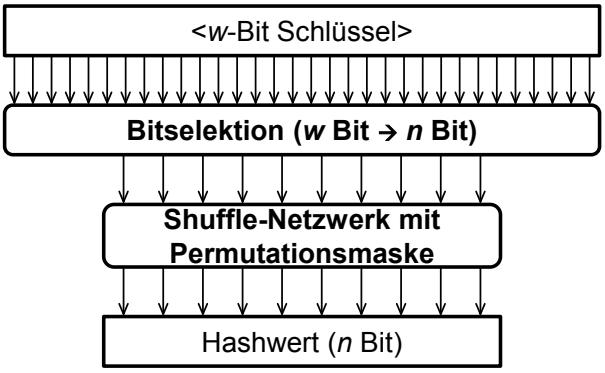


Abbildung 4.8: Ablauf des Hashing-Algorithmus

gleichzeitige Laden von $2k$ Schlüsseln sowie das Abspeichern von $4k$ Hashwerten. Dies erfordert das zweimalige Anwenden der Hashing-Operation pro Iteration der Hauptschleife. Demnach ergibt sich die für das Hashing angepasste Hauptschleife mit:

```
for(i=0; i<key_count/(4*SIMD_WIDTH); i++){
    loadA1_incr(); loadA2_incr();
    loadB1_incr(); loadB2_incr();
    storeA_hop(); storeB_hop();
}
```

Die Instruktionen `loadA1_incr` und `loadA2_incr` laden jeweils k Schlüssel aus DMEM0 bzw. DMEM1. `storeA_hop` verarbeitet diese $2k$ Schlüssel und speichert die berechneten $2k$ Hashwerte in DMEM0 ab. In gleicher Weise ermittelt und speichert die Instruktion `storeB_hop` die mit `loadA1_incr` und `loadA2_incr` geladenen Schlüssel nach DMEM1. Ein VLIW umfasst dabei die beiden Instruktionen, die auf einer Codezeile stehen. Die Hauptschleife führt insgesamt $\frac{N}{4k}$ Iterationen aus und benötigt pro Iteration drei Takte. Damit ist ein maximal theoretischer Durchsatz von $\frac{4}{3}k$ Elementen pro Takt möglich. Die Instruktionen enthalten ebenfalls eine interne Pipeline, um den kritischen Pfad zu verkürzen und die zuvor beschriebenen Daten-Hazards zu umgehen. Die Hashing- und Speicheroperationen verarbeiten damit jeweils die geladenen Daten aus der vorhergehenden Iteration. Es ergibt sich eine Latenz von sechs Takten.

Die hier implementierte Hashfunktion entspricht dem PEXT-Befehl aus der BMI2-Instruktionssatzerweiterung (Bit Manipulation Instruction) von Intel [87]. PEXT kann zwar in einem Takt ausgeführt werden, implementiert jedoch nicht die hier vorgestellte SIMD-Variante. Es ist deshalb zu erwarten, dass der DBA ASIP einen $k\times$ höheren Durchsatz als der Intel erreicht, wenn beide Prozessoren bei gleichen Taktfrequenzen arbeiten würden.

4.3 Evaluierung des DBA ASIP

Der folgende Abschnitt umfasst zunächst die aus Simulationen gewonnen Ergebnisse und Bewertungen des DBA ASIP für die drei Sorted-Set Algorithmen, Merge-Sort und Hashing. Dazu gehört die Analyse von Durchsatz und Flächen- und Leistungsverbrauch der

Algorithmen für verschiedene Prozessorkonfigurationen. Nach dem Vergleich dieser Ergebnisse mit anderen Prozessorarchitekturen, werden die Resultate für den DBA ASIP als Chipimplementierung vorgestellt.

4.3.1 Prozessorkonfigurationen

Die in diesem Kapitel verwendeten Prozessorkonfigurationen von Cadence Tensilica sind der Basisprozessor `LX5_32_1LSU` aus Abbildung 4.4a und der erweiterte Prozessor mit einer LSU `DBA_128_1LSU` bzw. `DBA_128_2LSU` mit zwei LSUs aus Abbildung 4.4b. Dabei erfolgen zunächst Untersuchungen für eine erweiterte Datenbreite des Speicherinterfaces von 128 Bit. Danach schließen sich weitere Betrachtungen mit Prozessorkonfigurationen und Speicherinterfacebreiten von 256 und 512 Bit inklusive der zugehörigen ISE an. Der Basisprozessor `LX5_32_2LSU` mit zwei LSUs dient nur zur Aufwandsabschätzung der zusätzlichen zweiten LSU. Ebenso können die Kosten für das erweiterte Speicherinterface mit Hilfe von `LX5_128_2LSU`, `LX5_256_2LSU` und `LX5_512_2LSU` erfasst werden. Der Compiler kann die zweite LSU und das erweiterte Speicherinterface aber ohne die ISE nicht ausnutzen und damit keine Steigerung der Leistungsfähigkeit erzielen. Als Vergleich dient weiterhin der Cadence Tensilica Diamond Standard-Mikrocontroller `108Mini`. Dieser entspricht dem `LX5_32_1LSU` jedoch mit dem Unterschied, dass dieser nur an einen externen Speicher angebunden ist, dessen Speicherzugriff im Vergleich zum lokalen Speicher eine höhere Latenz aufweist und nicht deterministisch ist. Zudem besitzt der `108Mini` keine VLIW-Architektur.

Entsprechend dem Entwicklungsablauf aus Abbildung 4.2 wurden alle Prozessorkonfigurationen einschließlich der SRAM-Makros mit dem Synopsys Design Compiler für einen 65 nm Prozess von TSMC unter typischen Bedingungen (1,2 V, 25 °C) synthetisiert. Genau wie beim DBA ASIC kann die Verlustleistung mittels Simulation der RTL-Beschreibung der Prozessoren und anschließender Leistungsanalyse abgeschätzt werden.

4.3.2 Einfluss der Befehlssatzerweiterung

Für die Untersuchungen der Leistungsfähigkeit des DBA ASIP im Hinblick auf die Instruktionssatzerweiterung wird zunächst ein SIMD-Parallelitätsfaktor von $k = 4$ gewählt, d. h. die Konfigurationen besitzen ein Datenspeicherinterface mit $W_{Mem} = 128\,\text{Bit}$ und es werden Elemente mit $W_{Elem} = 32\,\text{Bit}$ verwendet. Tabelle 4.1 fasst die aus der Synthese ermittelten Ergebnisse zusammen. Der `108Mini` erreicht die höchste Taktfrequenz mit 442 MHz und besitzt gleichzeitig mit $0{,}220\,\text{mm}^2$ die größte Logikfläche. Der Flächenunterschied ist auf die zusätzliche Hardwareunterstützung der Integer-Division und weiteren Peripherie- und Debuginterfaces zurückzuführen, die nicht in den LX5-Prozessoren enthalten sind. Die zusätzliche LSU und die ISE der LX5-Prozessoren verringern jeweils die Taktfrequenz geringfügig bis auf 410 MHz für den `DBA_128_2LSU` und erhöhen dessen Logikfläche, die aber immer noch geringer als die Fläche des 128-kB Speichers ist.

Bei der Ermittlung der maximalen Taktfrequenz hat sich gezeigt, dass die langen kombinatorischen Pfade, die durch die zusätzlichen Instruktionen entstehen, nicht kritisch sind und durch das Einfügen von Pipelinestufen reduziert werden können. Stattdessen

Tabelle 4.1: Syntheseergebnisse verschiedener Prozessorkonfigurationen

	108Mini	LX5_32_1LSU	LX5_32_2LSU	DBA_128_1LSU	DBA_128_2LSU
Speicherbreite in Bit					
DMEM	32	32	2×32	128	2×128
IMEM	32	32	32	64	64
VLIW-Breite in Bit	–	32	32	64	64
f_{max} in MHz	442	435	429	424	410
Fläche in mm^2	0,220	0,951	0,951	1,234	1,319
Speicher	–	0,774	0,774	0,774	0,774
Logik	0,220	0,177	0,177	0,460	0,545

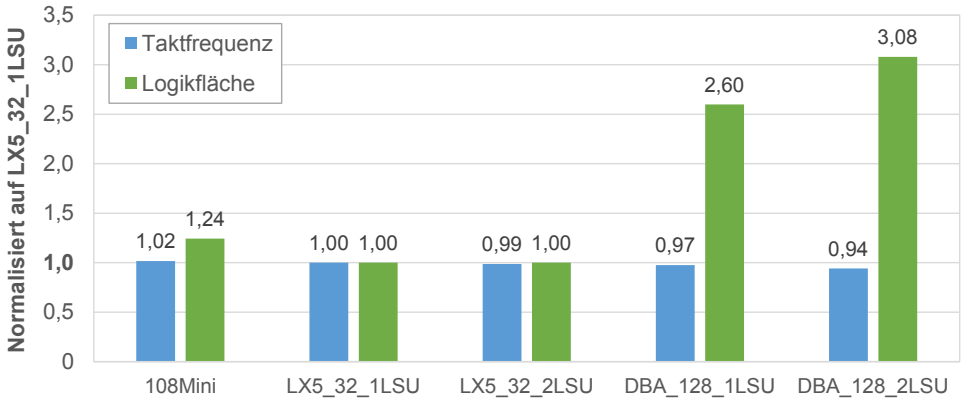


Abbildung 4.9: Syntheseergebnisse verschiedener Prozessorkonfigurationen jeweils normalisiert auf den Basisprozessor LX5_32_1LSU mit 435 MHz, 0,177 mm^2

befindet sich der kritische Pfad immer innerhalb des Prozessors oder auch zwischen Prozessor und Speicher. Es erhöht sich jedoch die Komplexität der Dekodierlogik und der Befehls-Multiplexer auf Grund der zusätzlichen Instruktionen und beeinflusst damit den kritischen Pfad innerhalb des Prozessors. Wie später in Abschnitt 4.3.3 ersichtlich wird, hat dies vor allem Auswirkungen auf Prozessoren mit größeren Speicherinterfaces.

Details zum Flächenverbrauch der einzelnen erweiterten Instruktionen sind in Tabelle 4.2 dargestellt. Die Flächenangaben sind in 10^3 Gate-Equivalents (kGE) angegeben. Dabei entspricht 1 GE einem NAND2-Gatter. Der Prozessor DBA_128_2LSU benötigt insgesamt 225,4 kGE. Der Basisprozessor mit dem Standard-Befehlssatz nimmt dabei 44,4 % der Gesamtfläche ein. Der Anteil der Befehlsdekodierer und Multiplexer aller erweiterter Instruktionen zusammen beträgt 5,4 %. Im Vergleich zum Merge-Sort und Hashing, haben die Instruktionen für die drei Sorted-Set Algorithmen mit insgesamt 60,8 kGE den höchsten Flächenbedarf. Die Ladeinstruktionen und die zugehörigen Register werden dabei von allen drei Algorithmen gemeinsam genutzt. Die Sorted-Set Vereinigung weist gegenüber der Intersektion und Differenz einen erhöhten Verdrahtungsaufwand und damit einen 4,4× höheren Flächenbedarf auf, da die Ergebniselemente nicht nur aus einer sondern aus beiden Eingangsmengen entnommen werden. Den kleinsten Flächenanteil hat das Hashing mit 20,9 kGE, da hier kein partielles Laden oder Umverteilen der Daten notwendig ist. Auch die Verschiebeeinheiten der einzelnen Bits zum Generieren des

Tabelle 4.2: Flächenverbrauch in 10^3 Gate-Equivalents (kGE) des DBA_128_2LSU (Abschätzungen aus Tensilica Xtensa Xplorer entnommen)

	Operationen	Register	Gesamt	Anteil von DBA_128_2LSU in %
DBA_128_2LSU	–	–	225,4	100,0
Basisprozessor	–	–	100,0	44,4
Dekodierung/MUX	–	–	12,2	5,4
Sorted-Set Algorithmen	37,2	23,6	60,8	27,0
Gemein. Ladeinstruktionen	4,7	14,9	19,6	(1)32,2
Intersektion	5,1	2,9	8,0	(1)13,2
Vereinigung	22,3	2,9	25,2	(1)41,4
Differenz	5,1	2,9	8,0	(1)13,2
Merge-Sort	29,9	1,6	31,5	14,0
Hashing	10,9	10,0	20,9	9,2

(1)Bezogen auf Gesamtfläche der Sorted-Set Algorithmen

Hashwertes, die den Hauptbestandteil dieser Instruktionen bilden, sind in Hardware als einfache Verdrahtungen und Multiplexer mit wenig Flächenaufwand realisiert.

Es schließt sich nun die Messung der Ausführungszeiten der Algorithmen an. Die Elemente in den Eingangslisten werden als vorzeichenlose ganzzahlige 32-Bit Werte interpretiert und folgen einer gleichmäßigen Verteilung innerhalb des Wertebereichs $\{0, ..., 2^{32} - 1\}$. Die Elementanzahl ist durch die Größe der Datenspeicher limitiert und beträgt für die Sorted-Set Algorithmen 2500 Elemente pro Eingangsmenge, für das Merge-Sort 7000 Elemente und für das Hashing 8000 Elemente. Die Prozessoren 108Mini, LX5_32_1LSU und LX5_32_2LSU führen den skalaren Code der Algorithmen aus und DBA_128_1LSU bzw. DBA_128_2LSU nutzen den angepassten Code inklusive der jeweiligen ISE. Der C-Code wird mit dem Xtensa C/C++ Compiler und dem Optimierungslevel *-O2* compiliert. Der Durchsatz berechnet sich dann aus der Gesamtlistenlänge N, multipliziert mit der maximalen Taktfrequenz f_{max} und dividiert mit der Ausführungszeit in Takten t_{Takt}.

Während der Messungen wurde festgestellt, dass sich der Speedup der Algorithmen mit steigender Elementanzahl erhöht, da sich der relative zeitliche Anteil der Vor- und Nachverarbeitung im Vergleich zur Hauptschleife reduziert. Das heißt, je mehr Elemente verarbeitet werden, desto höher ist der Vorteil der ISE. Weiterhin stellte sich heraus, dass der Speedup bereits bei den maximalen Listenlängen sein Maximum erreicht, da für diese Fälle der zeitliche Anteil der Hauptschleife dominiert und der Anteil der Nachverarbeitung lediglich etwa 2 % der Gesamtausführungszeit einnimmt. Die für die Messungen genutzten Elementzahlen erlauben damit die Evaluierung der Algorithmen bei höchster Leistungsfähigkeit.

Tabelle 4.3 fasst die Ergebnisse für den Durchsatz, Verlustleistung, Energieverbrauch und ATE-Produkt der drei ausgewählten Datenbankalgorithmen zusammen. Der Leistungsverbrauch des 108Mini wird vollständig durch die dynamische Leistung der Logik für den Zugriff auf den externen Speicher bestimmt. Es ergeben sich deshalb keine Unter-

Tabelle 4.3: Ergebnisse der Datenbankoperatoren für verschiedene Prozessorkonfigurationen. Einheiten: Durchsatz in 10^6 Elem./s, Verlustleistung in mW, Energieverbrauch in pJ/Elem, ATE-Produkt in $mm^2\,mW/(10^6\,Elem./s)^2$

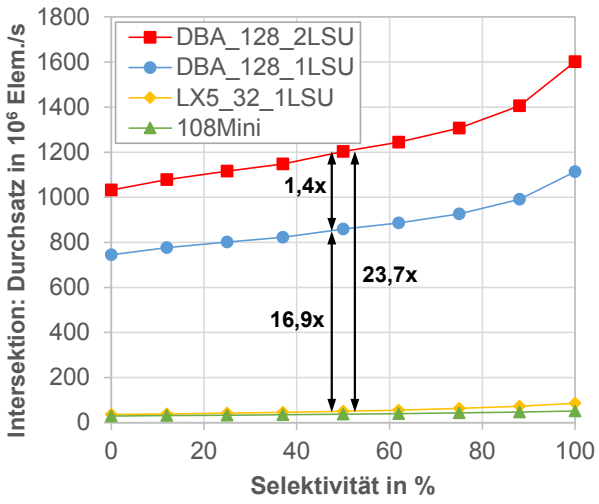
		108Mini	LX5_32_1LSU	LX5_32_2LSU	DBA_128_1LSU	DBA_128_2LSU
Sorted-Set Intersektion	Durchsatz	31,3	50,7	50,0	859,0	1203,0
	Leistung	27,4	43,6	43,4	76,1	91,8
	Logik	27,4	26,2	26,6	40,5	54,4
	Speicher	–	17,4	16,8	26,6	37,4
	Energie	875,4	860,0	868,0	88,6	76,3
	ATE	$6{,}2 \cdot 10^{-3}$	$16{,}1 \cdot 10^{-3}$	$16{,}5 \cdot 10^{-3}$	$12{,}7 \cdot 10^{-5}$	$8{,}4 \cdot 10^{-5}$
Merge-Sort	Durchsatz	1,7	3,2	3,1	29,3	28,3
	Leistung	27,4	44,9	44,5	67,6	63,5
	Logik	27,4	26,9	26,9	37,6	34,4
	Speicher	–	18,0	17,6	30,0	29,1
	Energie	16 117,6	14 031,3	14 354,8	2307,2	2243,8
	ATE	2,1	4,2	4,4	$97{,}2 \cdot 10^{-3}$	$104{,}6 \cdot 10^{-3}$
Hashing	Durchsatz	1,1	2,0	2,0	1122,4	2156,5
	Leistung	27,4	32,6	30,0	65,9	84,1
	Logik	27,4	25,0	24,1	34,0	41,3
	Speicher	–	7,6	5,9	31,9	42,8
	Energie	24 909,1	16 300,0	15 000,0	58,7	39,0
	ATE	4,9	7,8	7,1	$6{,}5 \cdot 10^{-5}$	$2{,}4 \cdot 10^{-5}$

schiede im Leistungsverbrauch für die verschiedenen Datenbankoperatoren. Im Folgenden wird nun detailliert auf die Resultate von jedem Algorithmus eingegangen.

Sorted-Set Algorithmen

Der Durchsatz für die Sorted-Set Intersektion ist im Diagramm der Abbildung 4.10 dargestellt (siehe auch Abbildung A.2 im Anhang A). Genau wie beim DBA ASIC erhöht sich der Durchsatz mit steigender Selektivität, da die Anzahl der Vergleiche abnimmt und damit mehr Elemente pro Takt nachgeladen werden können (siehe auch Kapitel 3.3.1). Der Durchsatz für die Intersektion für den 108Mini beträgt 31,3 Mio. Elem./s für eine Selektivität von 50 %. Bereits der Prozessor LX5_32_1LSU ermöglicht durch die Nutzung eines lokalen Speichers eine annähernde Verdopplung des Durchsatzes. Bei Hinzunahme der ISE und das Ausnutzen des erweiterten Speicherinterfaces mit dem DBA_128_1LSU ist eine Steigerung des Durchsatzes um Faktor 16,9 zu erkennen. Wird auch die zweite LSU genutzt und damit das gleichzeitige Laden aus beiden Eingangsmengen ermöglicht, erreicht der Prozessor DBA_128_2LSU einen Durchsatz von 1203,0 Mio. Elem./s. Das entspricht einem Speedup von 23,7 gegenüber dem Basisprozessor LX5_32_1LSU. Dabei trägt die Datenparallelität als auch die Reduzierung des Codes auf die einfache Hauptschleife zur Beschleunigung bei. Damit entfallen z. B. die Takte zum Setzen der Zeiger und das Ausführen der bedingten Sprünge. Der Unterschied des Durchsatzes zwischen LX5_32_1LSU und LX5_32_2LSU entsteht nur auf Grund der unterschiedlichen Taktfrequenzen der Prozessoren.

Wie in Abschnitt 4.2.4 analysiert wurde, wird der theoretisch maximale Durchsatz bei



Durchsatz in 10^6 Elem./s, Selektivität: 50 %

Prozessor	Intersektion	Vereinigung	Differenz
108Mini	31,3	26,4	35,7
LX5_32_1LSU	50,7	47,7	50,4
LX5_32_2LSU	50,0	47,0	49,7
DBA_128_1LSU	859,0	574,2	859,0
DBA_128_2LSU	1203,0	780,4	1192,6

Abbildung 4.10: Sorted-Set Algorithmen: Durchsatz für verschiedene Prozessorkonfigurationen

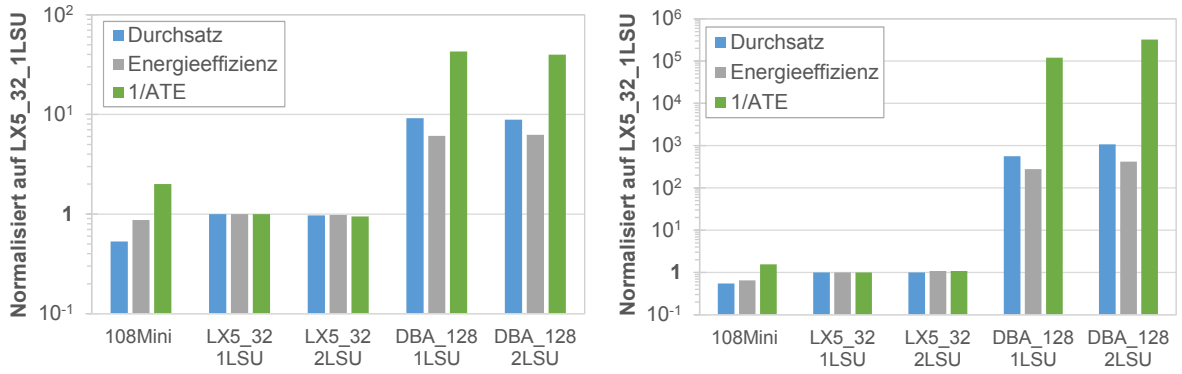
einer Selektivität von 100 % erreicht, d. h. wenn die Hauptschleife pro Iteration $2k$ Elemente verarbeitet. Dieser wird für den DBA_128_2LSU durch die Speicherbandbreite begrenzt und beträgt $k \cdot f_{max} = 1640$ Elem./s. Der maximal gemessene Durchsatz liegt bei 1602 Elem./s und erreicht damit annähernd das theoretische Maximum.

Die Ergebnisse für die Sorted-Set Differenz entsprechen in etwa der Intersektion, da hier nur der Vergleich der Elemente verändert ist. Dies hat keinen signifikanten Einfluss auf die Leistungsfähigkeit des DBA ASIP. Lediglich die Sorted-Set Vereinigung weist einen geringeren Durchsatz auf, da Elemente aus beiden Eingangsmengen abgespeichert werden müssen und damit die Hauptschleife des angepassten C-Codes verhältnismäßig mehr Iterationen zu durchlaufen hat.

Aus Tabelle 4.3 ist ersichtlich, dass die Verlustleistung des DBA_128_2LSU mit 91,8 mW für die Sorted-Set Intersektion am größten ist. Das ist auf die erhöhte Aktivität der zweiten LSU und der ISE zurückzuführen. Der Speicher benötigt dabei im Mittel 45 % der Gesamtleistungsaufnahme. Auch hier entsteht der Unterschied zwischen LX5_32_1LSU und LX5_128_2LSU nur auf Grund der unterschiedlichen Taktfrequenzen der Prozessoren, da der skalare C-Code ohne die ISE nicht von der zweiten LSU profitiert. Im Vergleich zum 108Mini ermöglicht der DBA_128_2LSU auf Grund des höheren Durchsatzes einen 11,5× geringeren Energieverbrauch.

Sortieren

Jeder Datenbankoperator weist Unterschiede in der Parallelität und in den Speicherzugriffen auf, die die maximal mögliche Leistungsfähigkeit beeinflussen. So iteriert Merge-Sort $\log N$ mal über die zu sortierende Menge, während z. B. die Sorted-Set Algorithmen nur einmal über die Eingangslisten laufen. Im Vergleich zum Basisprozessor LX5_32_1LSU erreicht der DBA_128_2LSU einen Speedup von 8,8 bei einer 1,4× Erhöhung der Verlustleistung. Wie in Abbildung 4.11a zu sehen, ergibt sich damit ein Energiegewinn von etwa Fak-



(a) Merge-Sort (b) Hashing

Abbildung 4.11: Ergebnisse verschiedener Prozessorkonfigurationen für ausgewählte Datenbankoperatoren jeweils normiert auf den Basisprozessor `LX5_32_1LSU`. Energieverbrauch und ATE-Produkt sind als reziproke Werte dargestellt (höher ist besser).

tor 6, sowie eine ca. 40-fache Verbesserung des ATE-Produkts. Obwohl der `LX5_32_1LSU` einen höheren Durchsatz als der `108Mini` erreicht, ist der Flächen- und Leistungsbedarf des `108Mini` geringer und damit das ATE-Produkt etwa um den Faktor 2 besser.

Der zeitlich dominierende Anteil des Merge-Vorgangs ist das eigentliche Verschmelzen entsprechend dem Algorithmus der Vereinigung (siehe auch Gleichung 2.6). Gemäß der Tabelle in Abbildung 4.10 erreicht der `DBA_128_2LSU` für die Sorted-Set Vereingung einen Speedup von Faktor 16,4. Da dieser Anteil der Vereingung im Merge-Sort etwa die Hälfte der Gesamtsortierzeit benötigt, entsteht hier ein etwa 8× höherer Durchsatz.

Wie in Abschnitt 4.2.4 erläutert wurde, hat das Anwenden der zweiten LSU keinen Vorteil und es ist kein Leistungsgewinn zwischen den entsprechenden Prozessoren möglich. Zusätzlich ist anzumerken, dass in weiteren Untersuchungen zur Leistungsfähigkeit des hier implementierten Merge-Sort kein signifikanter Einfluss auf die Verteilung oder evtl. Vorsortierung der Eingangsdaten festgestellt wurde.

Hashing

Der für das Hashing implementierte SIMD-Ansatz und die in einer Instruktion kombinierten Schiebeoperatoren erlauben einen Speedup von 560 gegenüber der skalaren Ausführung mit dem `LX5_128_1LSU`. Die Beschleunigung wird hier durch das Kombinieren der zahlreichen bitweisen Operatoren in einer Instruktion erreicht, deren Ausführung mit einem GP-RISC sehr zeitaufwendig ist. Bei Hinzunahme der zweiten LSU verdoppelt sich die Anzahl der verarbeitenden Elemente pro Schleifendurchlauf, da nun auch der zweite Datenspeicher Eingangs- und Ergebniselemente enthält. Wie in Abbildung 4.11b dargestellt, erhöht diese zusätzliche Parallelität den Speedup auf 1080. Der Durchsatz des `DBA_128_2LSU` liegt dann bei 2156,5 Mio. Elem./s bzw. 5,26 Elem./Takt und entspricht damit annähernd dem maximal theoretisch möglichen Durchsatz von 5,33 Elem./Takt. Auf Grund des hohen Beschleunigungsfaktors ergibt sich auch

ein 420× Energiegewinn des DBA_128_2LSU gegenüber dem LX5_32_1LSU. Das ATE-Produkt verbessert sich dann um fünf Größenordnungen.

Weitere Messungen haben gezeigt, dass der skalare als auch der parallele Hashing-Algorithmus nicht von den Eingangsdaten abhängt. Ebenso besteht kein Einfluss auf die Leistungsfähigkeit der Prozessoren bzgl. der Position oder Anzahl der gesetzten Bits in der Bitmaske der Hashfunktion. Lediglich beim Abbilden von Tupeln in eine Hashtabelle entscheidet die Hashfunktion über die Verteilung der Daten in den Buckets, der Kollisionen und damit auch über die Ausführungszeit.

Weitere Algorithmen

Tabelle 2.1 zeigt noch weitere Datenbankoperatoren, die essentiell für eine effiziente Anfrageverarbeitung sind. Diese Operatoren wurden deshalb auch mit Instruktionssatzerweiterungen beschleunigt und ebenfalls in den DBA ASIP integriert. Zum einen sind das der Sort-Merge-Join und die Sort-Merge-Aggregation, die auf Relationen mit sortieren Tupeln arbeiten [64]. Der Sort-Merge-Join ermittelt ähnlich der Intersektion alle Tupel zweier Relationen, deren Schlüssel identisch sind. Die Sort-Merge-Aggregation gruppiert und aggregiert entsprechend einer Aggregatfunktion alle Tupel einer Relation, die den gleichen Schlüssel aufweisen. Die ISE ermöglichen einen bis zu 15× höheren Speedup im Vergleich zum nicht beschleunigten Algorithmus und erreichen eine um drei Größenordnungen höhere Energieeffizienz gegenüber FPGA-Implementierungen.

Ergänzend zu dem vorgestellten Hashing-Algorithmus wurden das Sampling und das String-Hashing implementiert [8]. Sampling gehört zu den Anfrageoptimierungsalgorithmen und bestimmt eine Häufigkeitsverteilung der in den Eingangsschlüsseln gesetzten Bits, um eine an den Datensatz angepasste Hashfunktion zu erhalten. *CityHash* [58] ist eine String-Hashfunktion, die das Abbilden von Zeichenketten in eine Hashtabelle erlaubt. Der DBA ASIP erreicht für das Sampling einen 760× und 180× höheren Beschleunigungsfaktor im Vergleich zur skalaren Ausführung mit dem Xtensa Basisprozessor bzw. gegenüber einem Intel Prozessor. Eine Beschleunigung des CityHash mit den erweiterten Instruktionen ist gegenüber x86-Prozessoren jedoch kaum möglich. Zum einen ist der Algorithmus auf Grund vieler im Voraus berechenbarer Operationen gut für Speicherarchitekturen mit Caches geeignet. Auf der anderen Seite besteht der Algorithmus aus mehreren sequentiell ausgeführten mathematischen Operatoren, die nicht durch SIMD-Operationen parallelisiert werden können. Es besteht deshalb nicht die Möglichkeit, alle Optimierungstechniken des ASIP vollständig anzuwenden.

Des Weiteren wurde eine ISE für die Kompression und Verarbeitung von Bitmap-Indizes entwickelt [63]. Die zusätzlichen Befehle beschleunigen die Kompression einer Bitfolge, die auf einer Lauflängencodierung basiert. Datenbankalgorithmen wie die Mengenoperationen (Intersektion, Vereinigung, Differenz) können dann direkt auf diesen komprimierten Daten ausgeführt werden. Im Vergleich zu einem hoch-performanten x86-Prozessor erreichen die mit dem DBA ASIP ausgeführten Algorithmen einen Speedup bis zu Faktor 5 und benötigen mehr als 530× weniger Energie. Wie die zuvor erläuterten Sorted-Set Algorithmen nutzen diese Implementierungen ähnliche Methodiken beim gleichzeitigen Laden und Speichern der Daten und zur Parallelisierung auf Bit-, Daten- und Befehlsebene.

Detaillierte Erläuterungen und Ergebnisse sind in den jeweils angegebenen vom Autor publizierten Veröffentlichungen zu finden.

4.3.3 Einfluss der Speicherbusbreite

Zur Untersuchung der Skalierbarkeit der entwickelten ISE, wird das Datenspeicherinterface des Xtensa Prozessors auf jeweils 256 Bit und 512 Bit erweitert. Mit Datenbreiten von $W_{Elem} = 32$ Bit ermöglicht dies die Entwicklung von erweiterten Instruktionen für SIMD-Ansätze mit $k = 8$ und $k = 16$. Um Randeffekte zu vermeiden, die durch die Integration verschiedenartiger ISE in einen Prozessor entstehen, wurden dazu speziell die Prozessorkonfigurationen `DBAi_128_2LSU`, `DBAi_256_2LSU` und `DBAi_512_2LSU` implementiert. Diese enthalten ausschließlich erweiterte Instruktionen für die Sorted-Set Intersektion, die die jeweiligen Speicherbusbreiten unterstützen. Der `DBAi_128_2LSU` entspricht demnach dem `DBA_128_2LSU` ohne zusätzliche Instruktionen für Merge-Sort und Hashing.

Zunächst erfolgt eine Synthese der generierten RTL-Beschreibung der Prozessorkonfigurationen. Tabelle 4.4 fasst die Ergebnisse zusammen. Für die Prozessoren mit einem 128 Bit und 256 Bit breiten Speicherinterface kann eine Taktfrequenz von 410 MHz erreicht werden. Bei dem Prozessor `DBAi_512_2LSU` reduziert sich die Taktfrequenz um etwa 21 % auf 322 MHz. Dies ist auf das teilweise nicht lineare Wachstum der Logikfläche zurückzuführen, das im folgenden Paragraphen erläutert wird.

Die Sorted-Set Intersektion benötigt für den Vollvergleich eines k-fach SIMD-Ansatzes k^2 Vergleicher. Hält man die Datenbreite konstant, ergibt sich demnach für den Vollvergleich ein quadratisches Wachstum der Logikfläche mit k. Das gilt ebenfalls für den Verdrahtungsaufwand der Shuffle-Netzwerke in den Lade- und Speicheroperationen, die jeweils k mit k Elementen verbinden. Ausgenommen sind die SIMD-Register, die jeweils k Elemente aufnehmen und deren Fläche damit linear mit k wächst. Abbildung 4.12a stellt die Flächenaufteilung der Prozessoren mit unterschiedlichen Datenspeicherbreiten dar. Im Vergleich zum `DBAi_128_2LSU` erhöht sich z. B. die Fläche der Register und der Operationen des `DBAi_256_2LSU` um Faktor 1,8 bzw. 3,8. Die quadratische Skalierung des Flächenverbrauchs für die erweiterten Instruktionen wirkt sich bei den betrachteten SIMD-Breiten noch nicht auf die Gesamtfläche der Prozessoren aus, da hier noch die Fläche des Basisprozessors dominiert. Dies ist erst bei SIMD-Breiten von mehr als 512 Bit zu erwarten, wenn der Flächenverbrauch des Basisprozessors weniger als 20 % der Gesamtfläche einnimmt. Die Unterschiede in der Speicherfläche in Tabelle 4.4 entstehen durch das Nutzen von verschiedenen Speichermakros, um die erforderlichen Bitbreiten bereitzustellen. Bei gleichbleibender Gesamtgröße des Speichers, hat ein großes Speichermakro eine höhere Flächeneffizienz als mehrere kleinere.

Im nächsten Schritt wird die Ausführungszeit der Sorted-Set Intersektion gemessen, um den Durchsatz zu erhalten. Dazu werden weiterhin die Elementanzahl und Datenverteilungen aus Abschnitt 4.3.2 verwendet. Die Prozessoren `DBAi_256_2LSU` und `DBAi_512_2LSU` erreichen einen Durchsatz von 5,5 Elem./Takt bzw. 9,5 Elem./Takt bei einer Selektivität von 50 %. Das entspricht einem Speedup von 1,9 bzw. 3,3 gegenüber dem Prozessor `DBAi_128_2LSU`. Wie erwartet ergibt sich eine ungefähre Verdopplung bzw. Vervierfachung des Durchsatzes bezogen auf die Taktanzahl. Berechnet man den

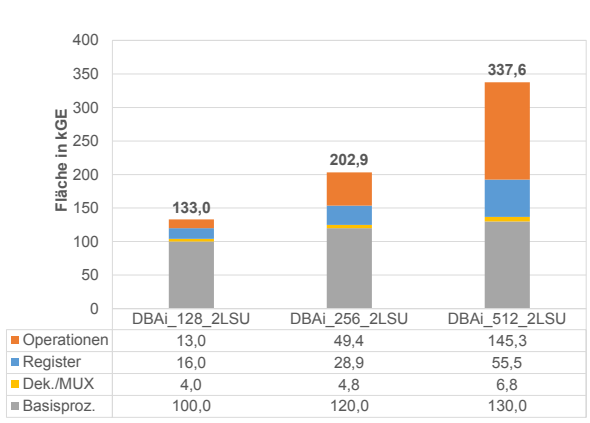
Tabelle 4.4: Ergebnisse der Sorted-Set Intersektion für Prozessorkonfigurationen mit verschiedenen Datenspeicherbreiten, Selektivität: 50 %

	DBAi_128_2LSU	DBAi_256_2LSU	DBAi_512_2LSU
Taktfrequenz f_{max} in MHz	410	410	322
Durchsatz in Elem./Takt	2,9	5,5	9,5
Durchsatz bei f_{max} in 10^6 Elem./s	1203,0	2235,6	3070,3
Fläche bei f_{max} in mm^2	1,088	1,167	1,852
Logik	0,314	0,394	0,727
Speicher	0,774	0,773	1,125
Leistung bei f_{max} in mW	91,8	128,6	198,7
Logik	54,4	63,1	92,2
Speicher	37,4	65,5	106,5
ATE-Produkt			
in mm^2 mW/(Elem./Takt)^2	11,9	5,0	4,1
in mm^2 mW/(10^6 Elem./s)^2	$6{,}9 \cdot 10^{-5}$	$3{,}0 \cdot 10^{-5}$	$3{,}9 \cdot 10^{-5}$

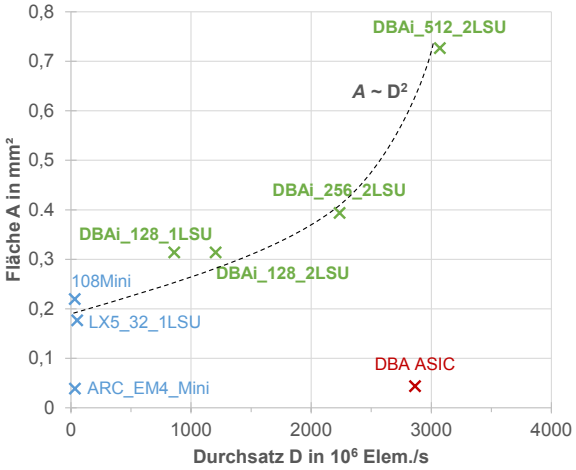
Durchsatz bei Hinzunahme der Taktfrequenz, erreicht der Prozessor DBAi_512_2LSU auf Grund der niedrigeren Taktfrequenz nur einen 2,6× Speedup gegenüber DBAi_128_2LSU.

Die Leistungsaufnahme der Hardwarelogik als auch für den Speicher erhöht sich ebenfalls mit steigender Datenspeicherbreite. Der DBAi_512_2LSU nimmt mit insgesamt 198,7 mW etwa 1,5× und 2,2× mehr Leistung als die Prozessoren DBAi_128_2LSU bzw. DBAi_256_2LSU auf. Die relative Steigerung der Verlustleistung entspricht ungefähr der Skalierung der Fläche für die Prozessorkonfigurationen, da für die gewählte 65-nm Technologie die Gesamtleistungsaufnahme hauptsächlich durch die dynamische Leistung bestimmt wird. Die statische Leistung nimmt dabei weniger als 1 % ein. Vergleicht man das ATE-Produkt der drei Prozessorkonfigurationen, erkennt man eine leichte Abnahme mit größeren Datenbreiten des Speicherinterfaces. Die reduzierte Taktfrequenz des DBAi_512_2LSU lässt dann jedoch keine weitere Verringerung des ATE-Produkts zu.

Abbildung 4.12b vergleicht den Durchsatz der Sorted-Set Intersektion für die verschiedenen Prozessorkonfigurationen in Bezug auf deren Flächenverbrauch. Als Referenz sind ebenfalls der DBA ASIC und der aus aus Kapitel 3.3.2 bekannte Synopsys ARC EM4-Mini eingetragen. Alle nicht optimierten Prozessoren bilden zusammen eine Gruppe bei einem Durchsatz von weniger als 50 Mio. Elem./s. Davon benötigt der 108Mini mit 0,220 mm^2 die größte und der ARC_EM4_Mini mit 0,039 mm^2 die kleinste Logikfläche. Betrachtet man die erweiterten Xtensa Prozessoren mit einem 128 Bit breiten Datenspeicherinterface, ist bei einem 1,8× Flächenzuwachs ein Speedup für die Sorted-Set Intersektion von mindestens Faktor 16,9 möglich. Die Vergrößerung des Datenspeicherinterfaces der Prozessoren auf 256 Bit und 512 Bit erlaubt eine annähernde Verdopplung bzw. Vervierfachung des Durchsatzes, die mit einer 1,3× bzw. 2,3× Erhöhung der Fläche einhergeht. Mit größeren Speicherbandbreiten des DBA ASIP geht die Erhöhung des Durchsatzes tendenziell mit einem quadratischen Wachstum der Fläche einher. Der DBA ASIC erreicht mit einem 128 Bit breiten Speicherinterface einen Durchsatz



(a) Flächenvergleich der Prozessoren mit unterschiedlichen Datenspeicherbreiten



(b) Mit einer Erhöhung der Datenspeicherbreite und des Durchsatzes zeigt sich ein quadratisches Wachstum der Logikfläche.

Abbildung 4.12: Ergebnisse der Sorted-Set Intersektion für unterschiedliche Prozessorkonfigurationen (Fläche ohne Speicher), Selektivität: 50 %

von 2864 Mio. Elem./s. Die Leistungsfähigkeit des DBA ASIC liegt damit annähernd im Bereich des `DBAi_512_2LSU`, benötigt aber 16,5× weniger Fläche.

4.3.4 Vergleich mit anderen Architekturen

Es schließt sich nun ein Vergleich der Leistungsfähigkeit des DBA ASIP (`DBA_128_2LSU`) mit hoch-performanten x86-Prozessoren, anderen RISC-Architekturen sowie mit dem DBA ASIC aus Kapitel 3 an. Schlegel u. a. [130] führen die Sorted-Set Intersektion mit einem `Intel i7-920` aus. Der Prozessor arbeitet bei einer Taktfrequenz von 2,67 GHz und einer mittleren Verlustleistung (TDP) von 130 W [85]. Die Messungen werden zum Vergleich immer mit einem Kern ausgeführt. Des Weiteren wurden in [130] SSE-Instruktionen zur Beschleunigung der Intersektion genutzt, die 128-Bit Lade- und Speicherbefehle sowie die SIMD-Vergleichsinstruktion `PCMPESTRM` verwenden. Weitere Vergleichsarchitekturen sind die bereits in Abschnitt 4.3.2 verwendeten Prozessoren `108Mini` und `LX5_32_1LSU` (LX5 RISC) sowie der Synopsys `ARC EM4-Mini`. Der `ARC EM4-Mini` arbeitet genau wie der LX5 RISC auf einem lokalen Speicher mit einer Speicherzugriffslatenz von einem Takt, nutzt jedoch keine 5-stufige sondern eine 3-stufige Prozessorpipeline. Damit reduziert sich die Taktfrequenz des `ARC EM4-Mini` im Vergleich zum Basisprozessor LX5 RISC um Faktor 1,3 auf 320 MHz.

Abbildung 4.13 stellt den Energieverbrauch über dem invertierten AT-Produkt für verschiedene Prozessorarchitekturen dar. Die Anpassung der Prozessoren an die Applikation wird damit durch eine positive x-Richtung und eine negative y-Richtung gekennzeichnet. Die x86-CPUs weisen dabei das höchste AT-Produkt und den größten Energieverbrauch auf. Im Vergleich zum Intel+SSE erzielt der DBA ASIP einen 1,1× höheren Durchsatz und einen um drei Größenordnungen niedrigeren Energieverbrauch. Dies re-

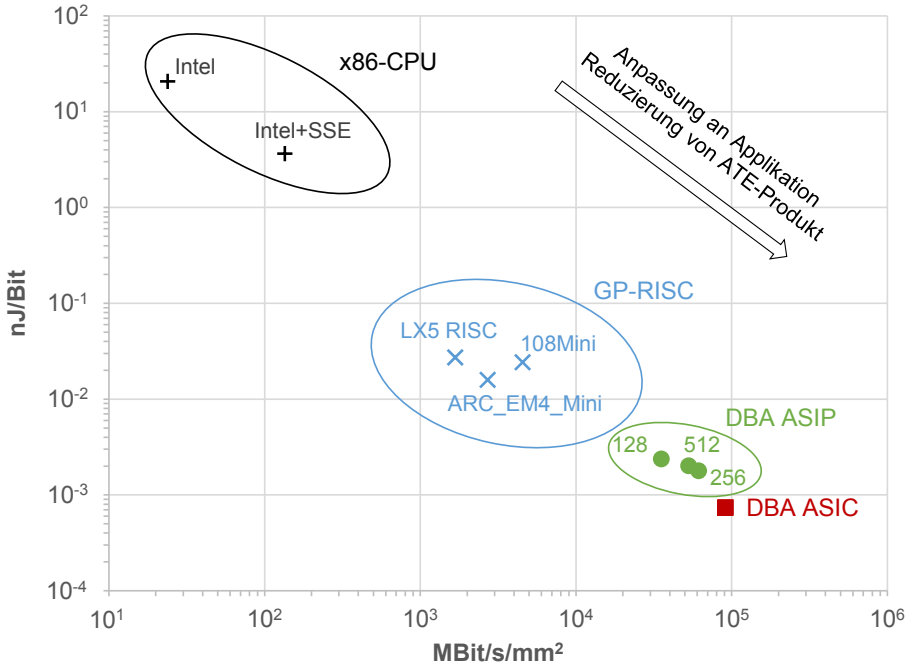


Abbildung 4.13: Durchsatz-Energie-Fläche-Zusammenhang der DBA-Implementierungen im Vergleich mit anderen relevanten Prozessorarchitekturen für die Sorted-Set Intersektion, Selektivität: 50 %

sultiert vor allem aus der etwa 920× geringeren Leistungsaufnahme des DBA ASIP. Der DBA ASIC enthält keinen GP-RISC wie der ASIP und ist damit in seiner Flexibilität eingeschränkt. Im Vergleich zum Intel+SSE erlaubt das dem DBA ASIC den Durchsatz zu verdoppeln und den Energieverbrauch zusätzlich zum DBA ASIP um Faktor 5 zu senken. Die Leistungsfähigkeit der weiteren RISC-Prozessoren `108Mini`, LX5 RISC, und `ARC EM4-Mini` ist bis zu Faktor 6,3 geringer als der des `Intel i7-920`. Da sowohl der Intel als auch die GP-RISC den gleichen C-Code ausführen, sind die Unterschiede auf den Compiler und die bis zu 8,3× geringere Taktfrequenz zurückzuführen. Die niedrige Taktfrequenz erlaubt jedoch eine mittlere Leistungsaufnahme von weniger als 100 mW und damit eine im Vergleich zum x86-Prozessor etwa drei Größenordnungen bessere Energiebilanz. Weitere Informationen zu den Prozessorarchitekturen sind im Anhang in Tabelle A.3 zusammengefasst.

Für die Algorithmen Merge-Sort und Hashing wird nun ebenfalls die Leistungsfähigkeit des DBA ASIP (`DBA_128_2LSU`) mit x86-CPUs verglichen (Tabelle 4.5). Die Arbeit von Chhugani u. a. [33] untersucht das Merge-Sort mit einem `Intel Q9550` [82] und nutzt SSE-Instruktionen zur Beschleunigung des Algorithmus. Der Prozessor hat eine 7,9× höhere Taktfrequenz, nimmt aber trotz der kleineren Herstellungstechnologie etwa 1500× mehr Leistung als der DBA ASIP auf. Mit Hilfe von SSE-Instruktionen erreicht der `Intel Q9550` einen fast doppelt so hohen Durchsatz als der DBA ASIP. Eine Angleichung des Durchsatzes wäre z. B. durch eine kleinere Technologiegröße möglich, die eine Erhöhung der Taktfrequenz des DBA ASIP erlaubt. Zudem kann das Datenspeicherinterface, wie in Abschnitt 4.3.3 gezeigt, vergrößert und damit ein höherer Durchsatz

Tabelle 4.5: Merge-Sort und Hashing: Vergleich des DBA ASIP (DBA_128_2LSU) mit anderen Prozessoren

	Merge-Sort		**Hashing**		
	DBA ASIP	Intel Q9550	DBA ASIP	Intel i7-4550U	Intel i7-3960X
Technologie in nm	65	45	65	22	32
Taktfrequenz in GHz	0,41	3,22	0,41	1,5	3,3
Gesamtfläche in mm^2	1,3	214,0	1,3	181,0	434,7
Durchsatz in 10^6 Elem./s	28,3	16,3	2156,5	20,6	14,9
		[1]60,0		[2]2063,0	
Verlustleistung in W	0,063	[3]95,0	0,084	[4]27,9	[4]24,3
Energie in nJ/Elem.	2,2	1583,3	0,04	13,5	1630,9

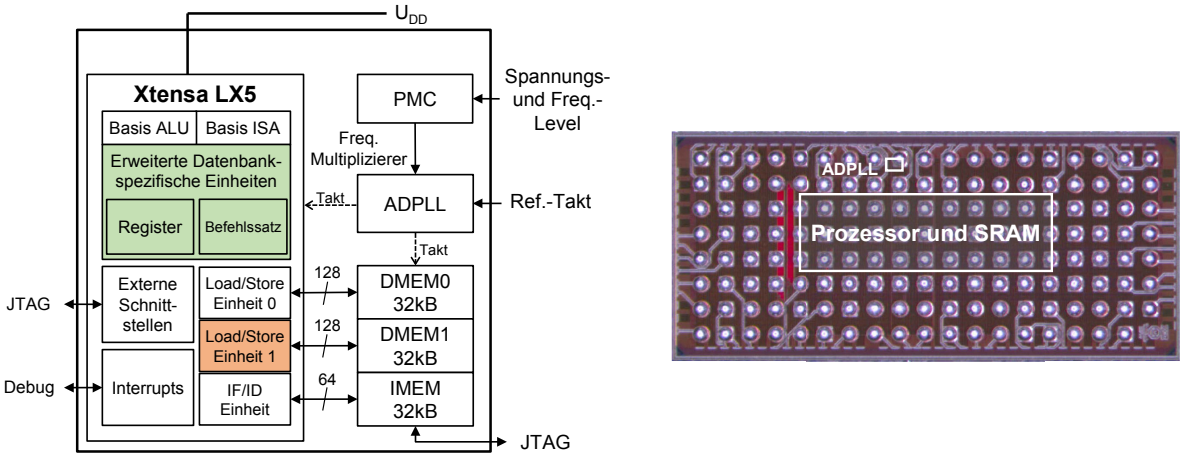
[1]Mit SSE-Instruktionssatz von Intel.
[2]Mit PEXT-Befehl aus dem BMI2-Instruktionssatz von Intel.
[3]TDP entnommen aus [82].
[4]Prozessorleistungsaufnahme PP0 gemessen mit RAPL-Counter.

erzielt werden. Trotz des geringeren Durchsatzes weist der DBA ASIP immer noch einen 720× niedrigeren Energieverbrauch auf.

Zum Vergleich der Leistungsfähigkeit des Hashing-Algorithmus werden der Intel i7-4550U [84] und der Intel i7-3960X [83] verwendet, die auf einer Haswell bzw. Sandy-Bridge Architektur basieren. Der Intel i7-4550U ist im Gegensatz zum Intel i7-3960X auf Grund seines niedrigen Leistungsverbrauchs für batteriebetriebene mobile Geräte geeignet. Zudem unterstützt dieser den PEXT-Befehl aus dem BMI2-Instruktionssatz, der ebenfalls eine Bitselektion ausführt und damit den Hashing-Algorithmus um Faktor 100 beschleunigt. Trotzdem erreicht der DBA ASIP mit seiner deutlich niedrigeren Taktfrequenz immer noch einen besseren Durchsatz. Zudem ermöglicht dessen niedrige Verlustleistung einen 340× und 54 300× geringeren Energieverbrauch gegenüber dem Intel i7-4550U bzw. Intel i7-3960X.

4.3.5 Chipintegration: Titan3D

Der DBA ASIP konnte in einer ersten Version in dem Forschungschip *Titan3D* integriert werden [62]. Die Entwicklung erfolgte am Vodafone Lehrstuhl für Mobile Nachrichtensysteme der TU Dresden unter Mitwirkung des Autors. Der Chip wurde in einer 28 nm Super Low-Power (SLP) CMOS Technologie von Globalfoundries hergestellt und integriert 1,11 Mio. NAND2-Gatter bei einer Chipfläche von 3,3 mm×1,5 mm = 4,95 mm^2. Abbildung 4.14 zeigt das Blockschaltbild und das Chipfoto des Titan3D. Auf dem Foto ist die Unterseite des Chips abgebildet, der in einer Flip-Chip-Technologie gefertigt und montiert wurde. Der DBA ASIP ist mit den drei 32-kB Single-Port Speichern (SRAM), einem Taktgenerator und einer Steuereinheit für das Leistungsmanagement (Power Management Controller, PMC) [75] in das Chip-Toplevel integriert. Der Prozessortakt wird mit einer digitalen Phasenregelschleife (All-digital Phase Locked Loop, ADPLL) [74] aus einem exter-



(a) Blockdiagramm (b) Chipfoto

Abbildung 4.14: Titan3D Halbleiter-Chip

nen Referenztakt erzeugt. Die ADPLL ermöglicht eine Taktfrequenz von 83 bis 2000 MHz und der PMC unterstützt Versorgungsspannungen im Bereich zwischen 0,8 und 1,1 V. Die Chipfläche der Logik und des 96-kB Speichers beträgt 0,286 mm^2 bzw. 0,232 mm^2.

Der DBA ASIP entspricht der in Abschnitt 4.3.1 eingeführten Prozessorkonfiguration `DBA_128_2LSU`, d. h. mit einem 128 Bit breiten Datenspeicherinterface und zwei LSUs. Die Befehlssatzerweiterung unterstützt dabei die folgenden Datenbankoperatoren: Sorted-Set Algorithmen (Intersektion, Vereinigung, Differenz), Merge-Sort, Sort-Merge Join und Sort-Merge Aggregation. Bei einer Versorgungsspannung von 1,1 V erreicht der Prozessor eine maximale Taktfrequenz von 200 MHz für die Sorted-Set Algorithmen und 250 MHz für die übrigen Operatoren. Die Messung des durchschnittlichen Leistungsbedarfs des RISC ohne und mit Nutzung der ISE ergibt 18,1 mW bzw. 27,7 mW (siehe auch Tabelle A.1 im Anhang A). Im Vergleich mit der zuvor in den Simulationen verwendeten 65-nm Technologie, reduziert sich die Verlustleistung damit um etwa Faktor 2,5. Da der Titan3D jedoch auch bei einer um bis zu mehr als die Hälfte reduzierten Taktfrequenz arbeitet, ist für einen besseren Vergleich die Betrachtung der benötigten Energie notwendig. Abbildung 4.15 stellt dazu den für jeden Datenbankoperator individuellen Energiebedarf für die Ausführung ohne (RISC) und mit (DBA ASIP) aktivierter ISE dar. Die Ergebnisse resultieren aus den Messungen von Ausführungszeit und Leistungsaufnahme auf dem Titan3D-Chip. Dabei werden weiterhin die in Abschnitt 4.3.2 beschriebenen Kardinalitäten der Algorithmen verwendet.

Im Vergleich zur Ausführung des skalaren Codes mit dem RISC, erhöht sich bei angewendeter ISE die Leistungsaufnahme um 55 % (Mittelwert über alle integrierten Datenbankoperatoren). Der Durchsatz steigt um durchschnittlich Faktor 16,7. Wie in Abbildung 4.15 zu sehen ist, resultiert dies in einem Energiegewinn von ca. einer Größenordnung. Die Skalierung der 65-nm Technologie aus der Simulation auf den 28-nm Prozess des Titan3D erlaubt eine weitere Reduzierung des gesamten Energiebedarfs um bis zu 50 %.

Der im Chip integrierte PMC unterstützt die Skalierung von Taktfrequenz und Versorgungsspannung, um eine Steigerung der Energieeffizienz zu erreichen. Als Referenz dient

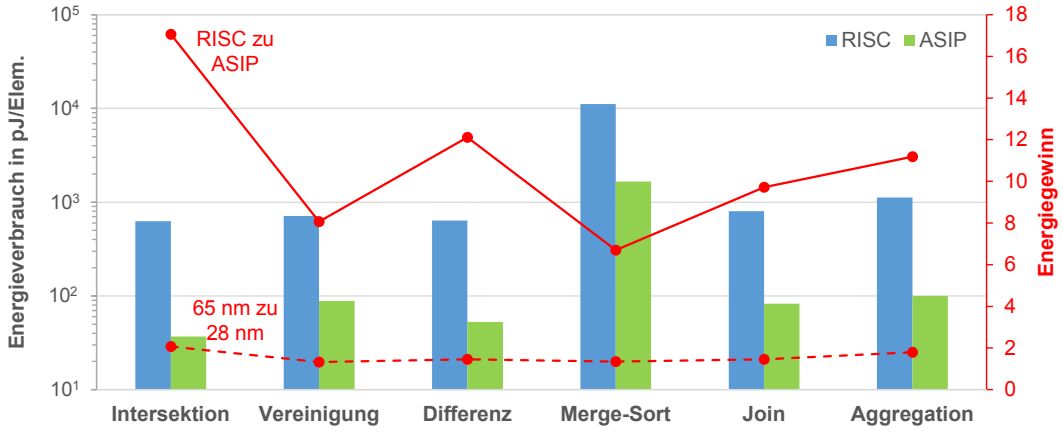


Abbildung 4.15: Titan3D: Energieverbrauch für ausgewählte DB Algorithmen ohne (RISC) und mit (DBA ASIP) aktivierter ISE

die Ausführung der Sorted-Set Intersektion ohne der ISE bei 200 MHz und 1,1 V, wobei mit dieser Konfiguration der höchste Durchsatz erreicht wird. Es werden zwei weitere Arbeitspunkte bei {150 MHz, 1,0 V} und {100 MHz, 0,9 V} definiert und ebenfalls der Energiebedarf bestimmt. Die Messungen werden für den RISC mit aktivierter ISE in gleicher Weise wiederholt. Abbildung 4.16 präsentiert die Ergebnisse. Der Energieverbrauch des DBA ASIP liegt nun unter 6 % im Vergleich zur zur Ausführung mit dem RISC (Referenz bei 100 %). Für den RISC-Fall ergibt sich bei verringerter Versorgungsspannung eine Reduzierung der Energie um bis zu 33 %, wenn die Halbierung der Taktfrequenz und die damit einhergehende Halbierung des Durchsatzes akzeptiert werden kann. Die Trendlinien im Diagramm der Abbildung 4.16 zeigen eine quadratische Skalierung des Energieverbrauchs bei Änderung des Arbeitspunktes, da die Verlustleistung quadratisch von der Versorgungsspannung abhängt.

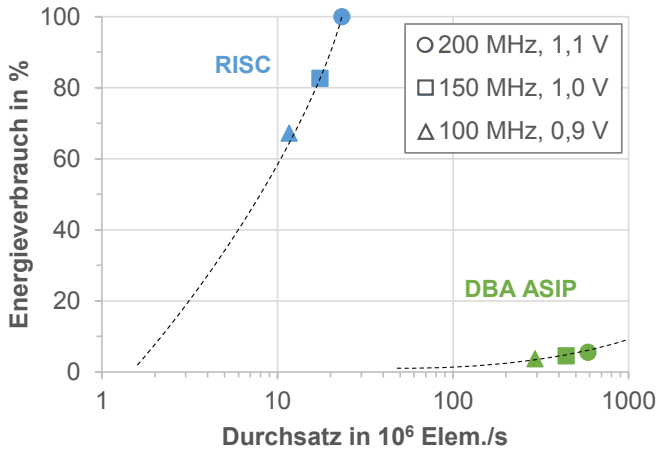


Abbildung 4.16: Titan3D: Energieverbrauch der Sorted-Set Intersektion bezogen auf die Ausführung ohne ISE (RISC) bei Skalierung von Taktfrequenz und Versorgungsspannung

4.4 Anfrageoptimierung

Der in dieser Arbeit vorgestellte Datenbankbeschleuniger ist auf die Anfrageverarbeitung, d. h. auf die energieeffiziente Ausführung der in Abschnitt 2.2.3 beschriebenen Operatoren spezialisiert. Im Gegensatz zu GP-Prozessoren sind Hardwarebeschleuniger auf die effiziente Ausführung einer begrenzten Anzahl von Anwendungen beschränkt. Auch der DBA unterstützt die beschleunigte Abarbeitung einer limitierten Anzahl von Datenbankoperatoren und benötigt zudem vordefinierte Anforderungen hinsichtlich der Verteilung und Anordnung der Daten. In diesem Abschnitt soll deshalb untersucht werden, inwieweit der DBA nicht nur für die Ausführung der Operatoren, sondern zusätzlich für weitere Aufgaben in einem Datenbanksystem zum Einsatz kommen kann. Dazu wird die in Abschnitt 2.2 eingeführte Anfrageoptimierung betrachtet, die vor der eigentlichen Anfrageausführung erfolgt (siehe auch Anfrageoptimierer in Abbildung 2.1). Die Anfrageoptimierung stellt ebenfalls einen wichtigen Bestandteil bei der Anfrageverarbeitung dar, um die Leistungsfähigkeit des Datenbanksystems zu steigern. Für die Untersuchungen in diesem Abschnitt werden Optimierungsalgorithmen aus dem Stand der Technik verwendet, deren Abarbeitung analysiert und auf die Ausführung mit dem DBA angepasst.

4.4.1 Algorithmen zur Anfrageoptimierung

Wie in Abschnitt 2.2 beschrieben, führen Datenbankanfragen eine Reihe von Anfrageoperatoren aus. Dabei kann das Umordnen oder Kombinieren dieser Operatoren die Leistungsfähigkeit des Datenbanksystems signifikant steigern. So können z. B. Datentransfers eingespart, vereinfachte Datenstrukturen verwendet und die Anzahl an Speicherzugriffen verringert werden. Einen Überblick in die Grundlagen der Anfrageoptimierung geben z. B. die Publikationen [59] und [104]. Die Methoden der logischen Optimierung schätzen wichtige Parameter der Anfragen und Datensätzen, wie die Selektivität, Sortierung oder Kompressionsverfahren. Weiterhin werden physische Optimierungsansätze verwendet, die z. B. Anforderungen an die Ausführungszeit, Speicherbedarf und Leistungsaufnahme der Verarbeitungseinheiten einbeziehen. Der Anfrageoptimierer berechnet und vergleicht nun mit Hilfe von Kostenfunktionen und den zuvor beschriebenen Parametern den Aufwand für mehrere potentielle Datenbankanfragepläne. Zur Anfrageausführung wird schließlich der Anfrageplan mit den geringsten Kosten verwendet.

In einem konkreten Beispiel von He u. a. [68] wird eine Schätzung der Selektivität durchgeführt, um den Durchsatz eines Join-Operators zu optimieren. Dazu werden die Daten vor dem eigentlichen Join durchlaufen, ohne dabei Ergebniselemente abzuspeichern, sondern nur um die Anzahl der Ergebniselemente zu bestimmen. Dieser Ansatz zur Schätzung der Selektivität wird oft in Datenbanksystemen angewendet und wird deshalb auch in dieser Arbeit untersucht (Abschnitt 4.4.2).

Die Schätzung der Selektivität gibt demnach Auskunft über die Verteilung der Daten, um damit die Ausführung der Datenbankoperatoren zu optimieren. In ähnlicher Weise arbeitet das bereits in Abschnitt 2.2.3 eingeführte Sampling. Dabei wird ein Teil der Daten vor der eigentlichen Anfrageausführung durchlaufen und ein Histogramm erzeugt, das die Verteilung der in den Elementen gesetzten Bits angibt. Mit diesen Informationen kann eine an die Daten angepasste Hashfunktion ermittelt werden, um den Aufbau und das

zeitliche Zugriffsverhalten einer Hashtabelle zu optimieren (z. B. Reduzierung der Anzahl von Kollisionen). Der Sampling-Algorithmus wurde in dieser Arbeit in den DBA ASIC integriert sowie mit Hilfe des DBA ASIP und der Befehlssatzerweiterung beschleunigt (siehe Abschnitt 4.3.2).

Eine andere Methode verfolgt der bereits in Kapitel 3.1.2 erwähnte LINQits Framework [34], der mit Hilfe des Anfrage-Compilers *Dandelion* [125] eine Optimierung durchführt. Der Compiler bestimmt mittels eines Datenflussgraphen den besten Anfrageausführungsplan hinsichtlich der gewählten Datenbankoperatoren und Verarbeitungseinheiten. Dabei werden Operatoren des Ausführungsplans umgeordnet oder zusammengefasst, um den Kommunikationsaufwand und die Zeit für Datentransfers zu minimieren. Diese logische Optimierung wird auch in dieser Arbeit auf einen mehr-wege Join angewendet. Das Berechnen der Kosten für die zu vergleichenden Anfragepläne kann dann mit Hilfe des DBA beschleunigt werden. Stehen mehrere Verarbeitungseinheiten zur Verfügung, erstellt Dandelion einen optimierten Anfrageplan für die parallele Ausführung auf mehreren CPUs und einer GPU (physische Optimierung). Das optimierte Abbilden des Anfrageplans auf ein Mehrkernsystem wird in dieser Arbeit in Kapitel 5 untersucht.

4.4.2 Optimierung mittels Schätzung von Selektivitäten

Im Folgenden soll nun konkret untersucht werden, inwieweit der DBA die Anfrageoptimierung unterstützen kann. Dazu wird ein in der Literatur oft verwendetes Prinzip zur Schätzung der Selektivität verwendet [104]. Das Ziel ist, dass der DBA in einem zusätzlichem Durchlauf die Selektivitäten der Datensätze berechnet, ohne dabei die ursprüngliche Gesamtausführungszeit zu erhöhen. Mit Hilfe der bekannten Selektivität ist dann die Optimierung der Reihenfolge nacheinander abzuarbeitender Datenbankoperatoren möglich, indem Datensätze, die verhältnismäßig kleine Ergebnismengen erzeugen, zu Beginn der Anfrage ausgeführt werden. Um das Konzept genauer zu erläutern und das Potential der Anfrageoptimierung zu zeigen, wird ein r-wege Join als Beispiel genutzt [56, 149]. Die Anfrage arbeitet auf r Eingangsrelationen, die im Folgenden als R_1 bis R_r bezeichnet werden. Jede Relation kann aus Tupeln beliebiger Breite bzw. mehreren m Spalten bestehen, wobei im folgenden der in der Literatur am meisten verbreitete Fall von 64-bit Tupeln ($m = 2$) angenommen wird. Ein Tupel repräsentiert damit ein Schlüssel-Wert-Paar (Key-Payload) mit jeweils einem 32-bit Ganzzahlwert, die in zwei separaten Listen a und b abgespeichert sind. Weiterhin wird angenommen, dass die Relationen keine Duplikate enthalten und nach den jeweiligen Schlüsseln sortiert sind.

Der mehr-wege Join wird nun in mehrere zwei-wege Joins unterteilt. Die Ergebnisrelation eines zwei-wege Joins enthält alle Tupel aus zwei gegebenen Eingangsrelationen, die die gleichen Schlüssel aufweisen (siehe auch Sort-Merge-Join in Kapitel 2.2.3). Der r-wege Join ergibt sich dann aus der Aneinanderreihung von mehreren zwei-wege Joins mit der Standardreihenfolge $(((R_1 \bowtie R_2) \bowtie R_3)...) \bowtie R_r$. Gleiche Tupel werden durch identische Schlüssel $R_i.a$ und $R_j.a$ mit $i, j = 1, ..., r\,(i \neq j)$ erkannt. Jedes Ergebnistupel enthält den Schlüssel und alle Payloads aus beiden Eingangsrelationen. Das bedeutet, dass eine Zeile der Ergebnisrelation den gemeinsamen Schlüssel und alle zugehörigen r Payloads aufnimmt. Die Idee ist nun, die Reihenfolge der einzelnen zwei-wege Joins über die Selektivitäten der Eingangsdaten zu bestimmen und danach den optimierten Join-Operator

```
// Test für Ausführungreihenfolge (R1 = R2) = R3
cost12 = intersect(R1.a, R2.a);

// Test für Ausführungreihenfolge R1 = (R2 = R3)
cost23 = intersect(R2.a, R3.a);

// Test für Ausführungreihenfolge (R1 = R3) = R2
cost13 = intersect(R1.a, R3.a);

// Wähle Plan 1: (R1 = R2) = R3
if(cost12 <= cost23 && cost12 <= cost13){
    joinSW(R1, R2, R_tmp);
    joinSW(R_tmp, R3, R_res);
}

// Wähle Plan 2:  R1 = (R2 = R3)
else if(cost23 < cost12 && cost23 <= cost13){
    joinSW(R2, R3, R_tmp);
    joinSW(R_tmp, R1, R_res);
}

// Wähle Plan 3: (R1 = R3) = R2
else if(cost13 < cost12 && cost13 < cost23){
    joinSW(R1, R3, R_tmp);
    joinSW(R_tmp, R2, R_res);
}
```

Abbildung 4.17: Anfrage eines drei-wege Joins mit zuvor erfolgter Optimierung

in Software auszuführen. Die Ausführung in Software ist notwendig, da der DBA keine beschleunigte Implementierung für einen zwei-wege oder mehr-wege Join enthält. Der in Abschnitt 4.3 gezeigte zwei-wege Join (Sort-Merge-Join) wurde zwar mit Hilfe einer Instruktionssatzerweiterung beschleunigt, speichert jedoch nur Indizes zu den Ergebniselementen ab und ist auf die Verarbeitung von Key-Payload-Paaren limitiert. Die Selektivitäten erhält man dann mit Hilfe des DBA und durch Ausführen der beschleunigten Sorted-Set Intersektion. Dabei werden nur die Anzahl der Ergebniselemente jedoch nicht die Ergebniswerte selbst benötigt.

Die Bestimmung der Reihenfolge der zwei-wege Joins aus den den abgeschätzten Selektivitäten wird im Folgenden am Beispiel mit $r = 3$ Relationen erläutert. Der erste von zwei Joins muss auf die beiden Relationen mit der geringsten Selektivität angewendet werden, da so die Länge der resultierenden Relation reduziert wird, die mit der verbleibenden Relation verarbeitet werden muss. Die Sorted-Set Intersektion wird dazu jeweils auf den Schlüsseln der drei Relationen ausgeführt, um damit die jeweilige Anzahl der Ergebniselemente und damit die Selektivitäten zu erhalten. In Datenbanksystemen erfolgt dieser Scan zumeist nur auf einem Teil der Daten, wobei das Ergebnis dann auf den gesamten Datensatz hochgerechnet wird. In dieser Arbeit werden die kompletten Listen einbezogen, da die lokalen Speicher des DBA nur eine Teilmenge der Daten enthalten.

In der Literatur existieren ebenfalls andere Methoden, um die Leistungsfähigkeit eines mehr-wege Join zu verbessern. Beispielsweise durchläuft der *Leapfrog Triejoin* [148]

alle Relationen gleichzeitig und verfolgt dabei immer den aktuell kleinsten und größten Schlüssel bis ein gemeinsamer Schlüssel gefunden wird. Der Algorithmus setzt voraus, dass die Daten als Baum im Speicher liegen. Dies würde zusätzliche Kosten auf Grund des Datentransfers erfordern, um die Daten aus den Relationen zu überführen. Die in dieser Arbeit implementierte Anfrageoptimierung ist nicht an solche bestimmte Datenstrukturen gebunden, da mit Hilfe der Selektivitätsabschätzung allgemeine Informationen über die Größe der Ergebnislisten und Speicheranforderungen vor der eigentlichen Ausführung des Join-Operators bestimmt werden.

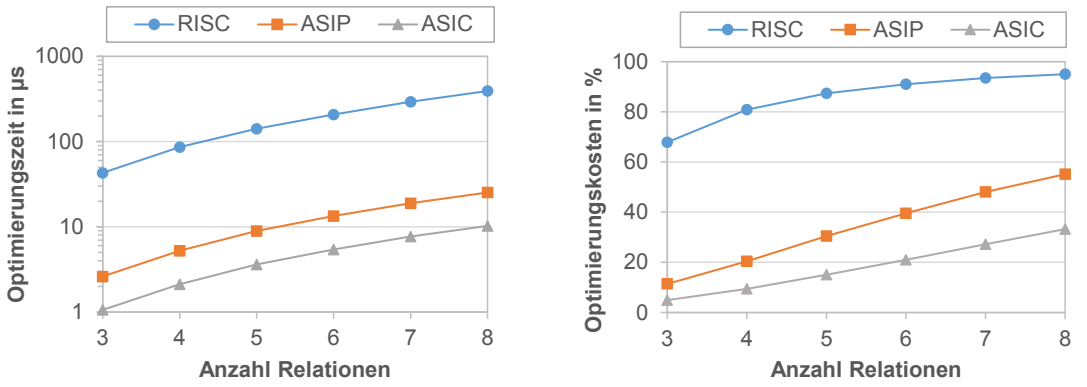
Abbildung 4.17 zeigt den Code der Selektivitätsabschätzung durch den Intersektionsoperator auf einzelnen Listen (`intersect`) und die darauffolgenden softwarebasierten Joins (`joinSW`). `intersect` gibt jeweils die berechnete Selektivität zurück, die nun ein Maß für die Ausführungszeit des Joins und damit auch für die Kosten ist. Schließlich wird der Anfrageplan mit den geringsten Kosten ausgeführt.

Erweitert man das Beispiel wieder zu r Relationen, erhält man $\frac{1}{2}r(r-1)$ Selektivitäten, die sortiert werden, um die optimale Reihenfolge der zwei-wege Joins zu bestimmen. Das bedeutet, dass die Anzahl der Intersektionen qudratisch mit r wächst. Da die Anzahl der zwei-wege Joins $r-1$ beträgt und damit nur linear wächst, vergrößert sich mit zunehmender Anzahl an Relationen r der relative Optimierungsaufwand. Dieses Verhalten wird im folgenden Abschnitt weiter ausgewertet.

4.4.3 Ergebnisse

Die Evaluierung des mehr-wege Joins inklusive der zuvor ausgeführten Selektivitätsabschätzung erfolgt nun für $r = \{3, ..., 8\}$ Relationen, um das Verhalten bei wachsender Komplexität zu beurteilen. Die Optimierung basiert wie zuvor beschrieben auf der Umordnung der einzelnen zwei-wege Joins. Da die Ergebnisse eines Joins wieder als Eingangsrelationen der nachfolgenden Joins genutzt werden, sind Relationen, die geringe Selektivitäten aufweisen, zuerst zu verarbeiten. Die nachfolgenden Eingangskardinalitäten sind damit bereits reduziert und die Ausführungszeit eines zwei-wege Joins verkürzt sich. Daraus lässt sich aber auch schlussfolgern, dass bei höheren Selektivitäten, d. h. je mehr sich die Relationen gleichen, der Vorteil der Optimierung zunehmend verloren geht. Aus diesem Grund wird die Gesamtselektivität des Joins auf 0 % gesetzt. Die Ergebnisrelation ist damit leer. Weiterhin wird jedoch angenommen, dass zwischen einzelnen Relationen identische Elemente vorkommen können. Die Selektivität zwischen den ersten $\lfloor \frac{r}{2} \rfloor$ Relationen wird deshalb auf 50 % und die der verbleibenden auf 0 % gesetzt. Die Kardinalität aller Relationen beträgt 512 Tupel und ist durch die Größe der lokalen Datenspeicher des DBA limitiert. Der softwarebasierte Join wird mit dem Basisprozessor Xtensa LX5 RISC (`LX5_32_1LSU`) ausgeführt. Für die Sorted-Set Intersektion kann der Basis-RISC, der DBA ASIP mit dem erweiterten Befehlssatz (`DBA_128_2LSU`) als auch der DBA ASIC aus Kapitel 3 eingesetzt werden.

In einem ersten Schritt wird die Zeit der Anfrageoptimierung gemessen. Abbildung 4.18a stellt diese über der Anzahl der Relationen dar. Der Graph bestätigt den zuvor angesprochenen quadratischen Aufwand zur Bestimmung aller Selektivitäten und damit die Laufzeit der ausgeführten Intersektionen. Der DBA ASIC benötigt für acht Relationen eine



(a) Optimierungszeit

(b) Optimierungskosten (Optimierungszeit relativ zur Gesamtausführungszeit des Joins).

Abbildung 4.18: Anfrageoptimierung des r-wege Joins: Ergebnisse für unterschiedliche Konfigurationen und als Funktion der Anzahl von Eingangsrelationen

Optimierungszeit von 10,2 µs. Das entspricht einem Speedup von 2,5 und 40 gegenüber dem DBA ASIP bzw. dem RISC.

Der nächste Schritt umfasst die Untersuchung der Gesamtausführungszeit der Anfrage inklusive Optimierung, wenn der Anfrageplan mit den geringsten Kosten ausgewählt wird. Abbildung 4.18b zeigt den Optimierungsoverhead, d. h. die Optimierungszeit dividiert mit der Gesamtausführungszeit der Anfrage. Der Basis-RISC benötigt bereits für drei Relationen 65 % der Zeit, um die Selektivitäten zu bestimmen. Damit würde die Optimierung mehr Zeit beanspruchen als die eigentliche Anfrageverarbeitung. Im Gegensatz dazu profitiert der DBA ASIC von der beschleunigten Sorted-Set Operation und der Optimierungsoverhead beträgt für acht Relationen nur 32 %.

4.4.4 Auswertung

Wie bereits zuvor erläutert, lohnt eine Optimierung nicht, wenn die Selektivitäten zwischen den Eingangsrelationen genau 0 % oder 100 % entsprechen. Sobald jedoch zwei Relationen wenige identische Schlüssel enthalten, kann die darauffolgende Optimierung die Leistungsfähigkeit der Anfrage bereits verbessern. Sind z. B. die Selektivitäten zwischen den Relationen R_1 und R_2, R_2 und R_3 sowie R_1 und R_3 50 %, 20 % bzw. 0 %, dann ergibt sich die optimierte Ausführungsreihenfolge des drei-wege Joins mit $R_1 \bowtie R_3 \bowtie R_2$. Der erste Join $R_1 \bowtie R_3$ produziert keine Ergebnistupel, sodass der zweite Join mit R_2 entfällt. Diese optimierte Ausführung ist etwa 2× schneller als die standardmäßige Reihenfolge $R_1 \bowtie R_2 \bowtie R_3$, da ein zusätzlicher Join auszuführen ist. Der Effekt wird noch deutlicher, wenn der Unterschied zwischen den einzelnen Selektivitäten ansteigt.

Des Weiteren stellt sich die Frage, wie hoch der Durchsatz des DBA mindestens sein muss, um für eine bestimmte Anzahl an Relationen r einen vorgegebenen relativen Optimierungsaufwand α nicht zu überschreiten. Die relativen Optimierungskosten K ergeben sich aus dem Verhältnis von Optimierungszeit t_{Opt} zur Gesamtausführungszeit t_{Ges} der

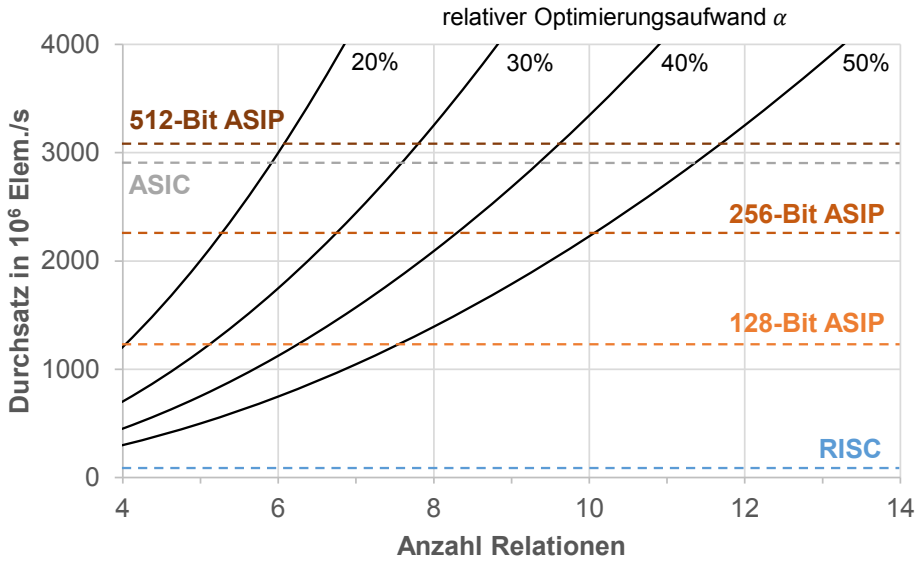


Abbildung 4.19: Anfrageoptimierung des r-wege Joins: Untere Schranke des Durchsatzes für verschiedene vorgegebene relative Optimierungskosten. Die gestrichelten Linien geben den Durchsatz der jeweiligen Konfigurationen an (RISC, ASIP mit drei verschiedenen Datenspeicherbreiten, ASIC).

Anfrage und sollen kleiner als der gegebene Optimierungsaufwand α sein:

$$K(r) = \frac{t_{Opt}(r)}{t_{Ges}(r)} = \frac{t_{Opt}(r)}{t_{Opt}(r) + t_{Join}(r)} < \alpha \tag{4.1}$$

Die Gesamtzeit ist dabei die Summe aus t_{Opt} und der Zeit zur Berechnung des Joins t_{Join}. Die Optimierungszeit t_{Opt} berechnet sich aus der Anzahl der pro Intersektion zu verarbeitenden Elemente N, dem Durchsatz D_{Inter} des jeweiligen Prozessors für die einmalige Ausführung der Sorted-Set Intersektion und der Anzahl der zu schätzenden Selektivitäten:

$$t_{Opt}(r) = \frac{Nr(r-1)}{2D_{Inter}} \tag{4.2}$$

Setzt man Gleichung 4.2 in 4.1 ein und formt nach D_{Inter} um, erhält man die untere Schranke für den Durchsatz des Prozessors, der die Intersektion zur Optimierung ausführt:

$$D_{Inter}(r) > \frac{Nr(r-1)(1-\alpha)}{2\alpha t_{Join}(r)} \tag{4.3}$$

Abbildung 4.19 stellt den Durchsatz in Abhängigkeit der Anzahl der Relationen für verschiedene α graphisch dar. Dabei wurde weiterhin $N = 512$ sowie die in Abschnitt 4.4.3 gemessene Ausführungszeit t_{Join} für den softwarebasierten Join verwendet. Es ist zu erkennen, dass der Durchsatz des Basisprozessors LX5 RISC nicht ausreicht, um eine der vorgegebenen maximalen Optimierungskosten einzuhalten. Jedoch kann z.B. der DBA ASIP mit einem 128 Bit breiten Datenspeicherinterface (`DBA_128_2LSU`) für die Optimierung

eines sechs-wege Joins eingesetzt werden, wenn der Optimierungsaufwand 40 % der Gesamtausführungszeit nicht übersteigen soll. Dürfen die relativen Optimierungskosten nur etwa 20 % betragen, muss der DBA ASIC oder der DBA ASIP mit einem 512 Bit breiten Speicherinterface (siehe `DBAi_512_2LSU` aus Abschnitt 4.3.3) eingesetzt werden. Die Resultate sind weitestgehend unabhängig von der Kardinalität N. Die Laufzeit t_{Join} des verwendeten Sort-Merge-Joins steigt linear mit N und kompensiert dadurch die Abhängigkeit von der Kardinalität des Mindestdurchsatzes (siehe Gleichung 4.3).

4.5 Schlussfolgerungen

Dieses Kapitel präsentierte den Datenbankbeschleuniger als Prozessor mit anwendungsspezifischen Befehlssatz für die energieeffiziente Verarbeitung von Datenbankoperatoren. Die erweiterten Befehlssatzerweiterungen des DBA ASIP zeigen eine bis zu 1080× Beschleunigung des Durchsatzes im Vergleich zur skalaren Ausführung mit dem Basisprozessor von Cadence Tensilica. Das Ergebnis wird mit einer acht-fach SIMD-Parallelität (2× 128-Bit breites Datenspeicherinterface) sowie mit einem 64-Bit VLIW mit drei Befehlsslots erreicht. Im Vergleich zum DBA ASIC aus Kapitel 3, dessen Hardware ausschließlich für die ausgewählten Datenbankoperatoren genutzt werden kann, ist der Durchsatz des DBA ASIP mit gleichen Datenbreiten des Speicherinterfaces etwa 2,4× geringer. Der zusätzliche dritte Datenspeicher des DBA ASIC erlaubt den gleichzeitigen Zugriff auf Eingangs- und Ausgangsmenge und damit einen höheren Durchsatz. Bei einer Vervierfachung der Speicherbandbreite auf 512 Bit gleicht sich die Leistungsfähigkeit des ASIP mit dem ASIC an. Die hohe Spezialisierung erlaubt dem DBA ASIC in diesem Fall eine Flächeneinsparung von Faktor 16,5×. Des Weiteren erzielt der DBA ASIP mit einem 512 Bit breiten Datenspeicherinterface eine 2,8× bessere Leistungsfähigkeit gegenüber hoch-performanten x86-Prozessoren. Zusätzlich wird die Energieeffizienz um bis zu drei Größenordnungen verbessert. Damit ist neben dem DBA ASIC auch der DBA ASIP als Beschleuniger in einem Coprozessor-erweiterten Datenbanksystem geeignet, um die GP-Prozessoren bei der Anfrageverarbeitung zu entlasten.

Entsprechend der Wahl der Parameter, d. h. dem SIMD-Parallelitätsfaktor k, der damit benötigten Speicherbandbreite und VLIW-Größe sowie der Aufteilung der Daten im Speicher können unterschiedliche Beschleunigungsfaktoren durch Ausnutzen verschiedener Parallelisierungsansätze erreicht werden. Der Einfluss auf die Kosten der Hardware sowie weitere Erkenntnisse und Implementierungsergebnisse sind in den nächsten Punkten zusammengefasst:

- Taktfrequenz: Die zusätzliche Load/Store-Einheit und die Befehlssatzerweiterungen für das auf 128-Bit vergrößerte Datenspeicherinterface verringern die Taktfrequenz geringfügig um knapp 6 %. Bei Speicherbreiten bis 512 Bit muss die maximale Taktfrequenz nochmal um etwa 21 % reduziert werden, da in diesem Fall der größte Teil der Logikfläche quadratisch mit der Datenbreite wächst. Speziell für noch größere Bitbreiten sollten beispielsweise zusätzliche Pipelineregister in die Hardware der erweiterten Befehle integriert werden, die zwar die Latenz erhöhen, aber eine höhere Taktfrequenz für den gesamten Prozessor erlauben.
- Parallelität der Hardware: Für die Sorted-Set Algorithmen und das Merge-Sort

kann der Vergleich von mehreren Elementen mit SIMD-Befehlen parallel erfolgen. Ebenso können für das Hashing und Sampling, Sort-Merge-Join und Sort-Merge-Aggregation sowie für die Bitmap-Verarbeitung alle Optimierungstechniken des ASIP angewendet werden. Lediglich bei Algorithmen wie z. B. dem CityHash, das einen fast vollständigen sequentiellen Ablauf aufweist, ist keine Beschleunigung mit SIMD-Operationen möglich.

- Skalierbarkeit: Eine Erhöhung Speicherbandbreite ermöglicht auch eine Steigerung des Durchsatzes der untersuchten Algorithmen, solange die zugehörigen erweiterten Instruktionen ebenfalls an die verfügbare Speicherbandbreite angepasst werden. In diesem Fall, d. h. bei einer Skalierung der Datenspeicherbreite um den Faktor k und einer konstanten Bitbreite der Daten, wächst die Fläche für die SIMD-Register linear mit k. Jedoch erhält man ein quadratisches Wachstum der Fläche für Operationen, die auf $k \times k$ Elementen arbeiten. Dazu zählen z. B. die Vergleicher und Shuffle-Netzwerke für die Sorted-Set Algorithmen. Eine quadratische Flächenerhöhung ist für den DBA ASIP erst bei SIMD-Breiten von mehr als 512 Bit zu erwarten, wenn der Flächenverbrauch der Befehlssatzerweiterungen die Gesamtfläche mit Basisprozessor und Speicher dominiert. Es müssen dann weitere Hardwareoptimierungen wie z. B. Pipelining eine zu starke Absenkung der maximalen Taktfrequenz verhindern, um die Leistungsfähigkeit des Gesamtprozessors nicht zu beeinflussen.
- Bei gleichbleibender Taktfrequenz verdoppelt sich die Leistungsaufnahme der Prozessorlogik und des Speichers auf Grund der zusätzlichen Fläche der Befehlssatzerweiterungen sowie durch die Verdopplung der Datenspeicherbreite.
- Grad der Spezialisierung: Eine gemeinsame Nutzung von ähnlichen Instruktionen der Befehlssatzerweiterungen für die Einsparung von Chipfläche ist für die betrachteten Algorithmen teilweise möglich. Zum Beispiel nutzen die drei Sorted-Set Algorithmen gemeinsame Ladeinstruktionen. Allerdings enthalten diese Ladebefehle auch algorithmusspezifische Operationen wie das Vorhalten von Daten, weshalb eine weitere gemeinsame Nutzung mit anderen Algorithmen wie z. B. dem Hashing nur eingeschränkt möglich ist. Eine Verallgemeinerung der Instruktionen hätte eine Reduzierung der Leistungsfähigkeit zur Folge.

Die bisherigen Analysen zum DBA beschränken sich auf die Ausführung mit einem Prozessor. Zudem werden Daten verarbeitet, die sich bereits im lokalen Speicher befinden. Das nächste Kapitel untersucht nun, inwieweit eine weitere Leistungssteigerung der Algorithmen durch mehrere Prozessoren (MIMD) möglich ist und welchen Einfluss dabei der Datentransfer zwischen den Prozessoren und zu einem globalen Speicher hat.

5 Parallelisierung auf dem Tomahawk MPSoC

Die vorherigen Kapitel haben erfolgreich gezeigt, wie Operatoren aus der Datenbank-Anfrageverarbeitung durch die Anpassung der Hardware beschleunigt werden können. Die Implementierungen des Datenbankbeschleunigers (DBA) als ASIC und ASIP erreichen die hohe Leistungsfähigkeit durch eine Parallelisierung der Algorithmen auf algorithmisch-logischer als auch auf Daten- und Befehlsebene. Dabei werden mehrere Elemente der Datensätze parallel verarbeitet sowie voneinander unabhängige Instruktionen gleichzeitig ausgeführt.

Dieses Kapitel untersucht nun, inwieweit eine Leistungssteigerung der Datenbankoperatoren auf Task- bzw. Systemebene in einem Mehrkernsystem (Multiprocessor System-on-Chip, MPSoC) möglich ist. In den bisherigen Betrachtungen befinden sich die zu verarbeitenden Daten bereits im lokalen Speicher des Datenbankbeschleunigers. Die Herausforderung beim Zusammenschalten mehrerer Verarbeitungseinheiten liegt deshalb besonders bei dem Verbindungsnetzwerk und dem globalen Speicher, die die notwendige Bandbreite für den Datenaustausch zur Verfügung stellen müssen. Zudem ist auf eine geeignete Verteilung und Partitionierung der Daten zu achten, um ein effizientes Laden und Speichern zu realisieren.

Es stellt sich die Frage, wie das Design eines MPSoC aussieht und welche Komponenten entscheidend für eine energieeffiziente Datenbank-Anfrageverarbeitung sind. Dazu werden zu Beginn des Kapitels Implementierungsaspekte diskutiert, um ein allgemeines Systemmodell eines MPSoC zu erhalten, das an die leistungsstarke Ausführung von Datenbankoperatoren angepasst ist. Für die anschließende Realisierung des Modells dient das am Vodafone Lehrstuhl für Mobile Nachrichtensysteme der TU Dresden entwickelte *Tomahawk* MPSoC. Die Plattform ermöglicht die Integration mehrerer Datenbankbeschleuniger (DBA), die über ein Network-on-Chip (NoC) untereinander als auch mit einem globalen Speicher verbunden sind. Die Untersuchungen in dieser Arbeit fokussieren sich dabei auf das In-Memory Computing, d. h. auf den Einfluss der unterschiedlichen Parallelitäten innerhalb der Verarbeitungseinheiten und durch mehrere Prozessoren als auch auf die Auswirkungen des notwendigen Datentransfer zu einem globalen Speicher. Die Beurteilung der Ergebnisse in Bezug auf ein gesamtes Datenbanksystem ist Gegenstand zukünftiger Arbeiten.

Das Kapitel beginnt mit einer Diskussion über ein MPSoC-Modell zur Datenbank-Anfrageverarbeitung und präsentiert dazu die in dieser Arbeit verwendete Tomahawk-Plattform sowie verwandte Mehrkernsysteme. In Abschnitt 5.2 werden die Methoden zum Scheduling der ausgewählten Datenbankalgorithmen beschrieben. Danach erläutert Abschnitt 5.3 die parallele Umsetzung der Algorithmen auf dem Tomahawk MPSoC

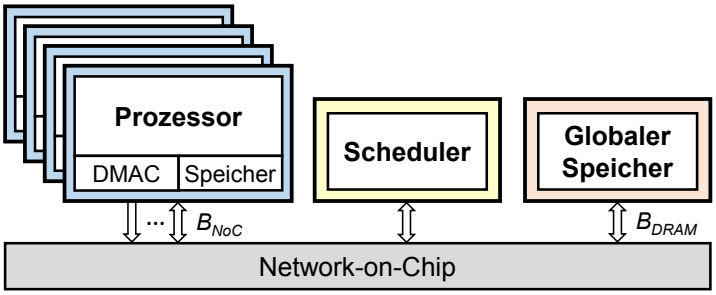


Abbildung 5.1: Allgemeines Systemmodell eines MPSoC mit mehreren Verarbeitungseinheiten bzw. Prozessoren, einer Managementeinheit für das Task-Scheduling (Scheduler) sowie ein Shared-Memory (globaler Speicher mit Bandbreite B_{DRAM}). B_{NoC} entspricht der Bandbreite eines Links zwischen den Routern des NoC bzw. dem Link zwischen Router und Verarbeitungseinheit.

durch Anwenden der erläuterten Scheduling-Konzepte. Die Ergebnisse werden dann in Abschnitt 5.4 präsentiert.

5.1 Mehrkernsysteme

Dieser Abschnitt diskutiert in einer Einführung die Herausforderungen beim Entwickeln eines Mehrkernsystems für die energieeffiziente Datenbank-Anfrageverarbeitung. Danach folgt ein Überblick über den Stand der Technik von Multiprozessorsystemen sowie die Präsentation der in dieser Arbeit verwendeten Tomahawk-Plattform.

5.1.1 Einführung

Im Allgemeinen besteht ein Multiprozessorsystem aus mehreren Verarbeitungseinheiten bzw. Prozessoren, die über ein Verbindungsnetzwerk verbunden sind. Dabei setzt sich zunehmend das Network-on-Chip [22] durch, da es im Gegensatz zu Bussystemen oder Punkt-zu-Punkt-Verbindungen mit der Anzahl der Prozessoren skaliert. Die Verarbeitungseinheiten sind zumeist durch spezialisierte Hardware oder einem anwendungsspezifischen Instruktionssatz an die Applikation angepasst. Sie besitzen vorzugsweise jeweils einen lokalen Datenspeicher, um die limitierte Bandbreite eines globalen Shared-Memory zu überwinden. DMA-Controller (Direct Memory Access Controller, DMAC), die durch die Applikation gesteuert werden, ermöglichen dann den Datentransfer zwischen den Prozessoren sowie zwischen Prozessor und globalen Speicher. Die Steuerung und Taskzuweisung kann durch ein oder mehrere Managementeinheiten übernommen werden (Scheduler). Das Ziel ist, eine möglichst gleichmäßige Auslastung der verfügbaren Prozessoren zu erreichen. Dies wird mit einem dynamischen Task-Scheduling aber auch durch die Anpassung des Leistungsverbrauchs durch eine individuelle Skalierung von Frequenz und Versorgungsspannung (Dynamic Voltage and Frequency Scaling, DVFS) der Prozessoren ermöglicht. Das beschriebene Systemmodell ist in Abbildung 5.1 dargestellt.

Um ein MPSoC zu entwickeln, das an eine leistungsstarke Abarbeitung der Datenbankoperatoren angepasst ist, sind verschiedene Implementierungsaspekte zu diskutieren:

- Applikation: Die Anfrageverarbeitung benötigt zahlreiche Datenbankoperatoren. Welche Operatoren sollen dabei vom MPSoC unterstützt werden? Die gleichzeitige Entwicklung der Architektur und der Algorithmen ist von Vorteil, um Entscheidungen hinsichtlich der notwendigen Leistungsfähigkeit und Verlustleistung direkt während der Designphase zu treffen. Die Anpassbarkeit des MPSoC an die Applikation ist zur Compilierzeit nur noch über Software möglich.
- Prozessoranzahl: Wie viele Prozessoren sind notwendig, um die gewünschte Parallelität auf Taskebene zu erreichen? Dabei muss auch sichergestellt werden, dass die Komplexität des Task-Schedulings nicht maßgeblich erhöht sowie die Bandbreite zum globalen Speicher nicht zum Flaschenhals wird.
- Skalierbarkeit: Wie skaliert das Mehrkernsystem bei zunehmender Prozessoranzahl? Die Skalierbarkeit wird dabei von der Parallelisierbarkeit der Applikation sowie von die Bandbreite des Verbindungsnetzwerkes beeinflusst.
- Homogenes oder heterogenes System: Sind Prozessoren mit unterschiedlichen Befehlssätzen oder Hardwarebeschleunigern notwendig, um die vielfältigen Datenbankalgorithmen zu unterstützen? Kann durch die Heterogenität auch der Flächenbedarf reduziert werden?
- Kommunikation: Datenbankanwendungen arbeiten auf großen Datenmengen. Wie sieht ein dazu angepasstes Verbindungsnetzwerk aus? Welche Topologie des NoC ist sinnvoll, um die gewünschte Bandbreite zur Datenkommunikation bereitzustellen? Sind DMA-Controller vorhanden, die einen intelligenten Datentransfer ermöglichen?
- Programmiermodell: Eine hohe Parallelität ist entscheidend für einen hohen Datendurchsatz. Welches Modell zur parallelen Ausführung der Datenbankoperatoren kann dazu angewendet werden? Wie ist dabei die Interaktion von Hardware und Software?
- Partitionierung: Wie können die Algorithmen bzw. die Daten auf mehrere Prozessoren aufgeteilt werden, sodass Datenabhängigkeiten minimiert werden und damit eine maximale parallele Ausführung ermöglicht wird?
- Speichermodell: Die Verarbeitung großer Datenmengen ist charakteristisch für Datenbankanwendungen. Sollte deshalb jeder Prozessor einen eigenen lokalen Speicher oder Cache besitzen? Wie viel Speicherplatz wird insgesamt innerhalb des MPSoC benötigt?
- Sicherheit: Wie kann die Isolation mehrerer nebenläufig ausgeführter Algorithmen erfolgen? Wie kann das Gesamtsystem vor Schadsoftware geschützt werden?

Aus den oben genannten Punkten lassen sich drei wesentliche Hauptschwerpunkte für die Entwicklung eines MPSoC zur Datenbank-Anfrageverarbeitung ableiten:

1. Es sind mit Hilfe von Hardwareerweiterungen angepasste Verarbeitungseinheiten zu nutzen, die den lokalen Datendurchsatz der Algorithmen entsprechend der gegebenen Parameter maximieren. Wie in den vorherigen Kapiteln gezeigt wurde, erlaubt das Anwenden einer Parallelität auf arithmetischer-logischer, sowie auf Daten- und Befehlsebene eine Leistungssteigerung.

2. Es werden hohe Bandbreiten für das NoC als auch zwischen lokalem Speicher und Prozessor benötigt. Ein Maß für die Beurteilung der Bandbreite eines Verbindungsnetzwerkes ist die *Bisektionsbandbreite*. Teilt man ein Netzwerk mit einer minimalen Anzahl von Schnitten in zwei annähernd gleich große Teile, dann ergibt sich die Bisektionsbandbreite aus der Summe aller Bandbreiten der Schnittverbindungen. Zum Beispiel hat ein NoC mit einer 2D Stern-Gitter-Topologie mit n Knoten und einer Bandbreite pro Link von B_{NoC} eine Bisektionsbandbreite von $\sqrt{n} \cdot B_{NoC}$. Im Vergleich dazu beträgt die Bisektionsbandbreite für eine hexagonale Topologie $(2\sqrt{n} - 1)B_{NoC}$ und ist damit fast doppelt so hoch.
3. Für die parallele Steuerung mehrere Datentransfers ist ein intelligenter DMA-Controller nötig, um Methoden wie das Double-Buffering anzuwenden und damit die vorhandene Speicherbandbreite effizient auszunutzen. Das Double-Buffering orientiert sich dabei am Prinzip eines Ping-Pong-Speichers: Der Speicher wird in zwei Hälften aufgeteilt. Ein Teil wird benutzt, um Daten aus dem globalen Speicher zu laden, während der Prozessor auf dem anderen Teil arbeitet. Danach verarbeitet der Prozessor die Daten der ersten Speicherhälfte und der zweite Teil wird für das Laden genutzt, usw.

5.1.2 Stand der Technik

Die ersten Multiprozessorsysteme wurden in den 1970er Jahren entwickelt. Beispiele sind die Systeme *Illiac IV* [26] oder *C.mmp* [161], die bereits bis zu 64 Verarbeitungseinheiten für die parallele Berechnung von arithmetisch-logischen Befehlen enthalten. Die Prozessoren besitzen einen lokalen Speicher und sind als Array angeordnet über eine Crossbar an einen gemeinsamen Speicher angebunden. Die ersten im Bereich der Datenbanken eingesetzten Multiprozessorsysteme sind die bereits in Kapitel 2.1.2 angesprochenen Datenbankmaschinen [39], die mit spezieller Hardware ausgestattet sind, um die Ausführung von Datenbankoperationen zu beschleunigen.

Mehr als zwanzig Jahre später entstanden die ersten Multiprozessorsysteme auf einem Chip, wie z. B. die von Siemens entwickelten GOLD-Chips (GSM One-Chip Logic Device) [133]. Der Chipsatz unterstützt alle benötigten Funktionalitäten der GSM-Technologie (Global System for Mobile communications). Dabei ist speziell der GOLD-Signalprozessor für die Sprach- und Kanalkodierung sowie zur Digitalfilterung und Modulation vorgesehen. Der Chip umfasst zwei Prozessorkerne mit On-Chip Speicher, die über ein Bussystem mit einem Mikrocontroller für Steuerfunktionen verbunden sind. Ein weiteres Beispiel ist der *Daytona* Multiprozessor-DSP [1], der ebenfalls auf Signalverarbeitungsalgorithmen spezialisiert ist. Daytona besteht aus vier Verarbeitungseinheiten, die jeweils einen RISC, einen Vektor-Coprozessor und einen 8-kB Cache enthalten. Der RISC verarbeitet Multimediaanwendungen wobei der Hardwarebeschleuniger DSP-Funktionalitäten mit Hilfe von Vektoroperationen unterstützt.

Darauf folgten weitere MPSoCs, die z. B. für Multimediaanwendungen, Basisbandoperationen oder als Netzwerkprozessoren eingesetzt werden. Wolf u. a. [154] geben einen detaillierten Überblick in diese Systeme. Ebenso existieren heutzutage zahlreiche hochmoderne MPSoCs, wie z. B. die SpiNNaker-Architektur zur Simulation neuronaler Netzwerke [48], die PULP-Plattform für energieeffiziente DSP-Anwendungen [126], oder die heterogenen

Multiprozessorsysteme des Forschungsinstituts CEA-Leti, die zur Abarbeitung komplexer Algorithmen in der digitalen Basisbandverarbeitung angewendet werden [21].

Anwendungsspezifische MPSoCs für die effiziente Ausführung der Datenbank-Anfrageverarbeitung sind dagegen selten in der Literatur zu finden. Ein Beispiel ist der Oracle Sparc Prozessor [114], der anwendungsspezifische Hardware neben hochperformanten CPUs auf einem gemeinsamen Chip integriert (siehe Kapitel 2.1.2). Des Weiteren werden oft General-Purpose Chip-Multiprozessoren wie z. B. Netzwerk- [57] oder Multimediaprozessoren [52] für die Abarbeitung von Datenbankoperationen eingesetzt. Die Systeme verknüpfen einfache Prozessorkerne, die im Vergleich zu x86-Prozessoren einen geringeren Flächen und Leistungsbedarf aufweisen. Die Prozessoren enthalten keine anwendungsspezifische Hardware. Wie die folgenden zwei Beispiele zeigen, kann jedoch mit Hilfe der Parallelität durch mehrere Prozessoren und mehrere Threads eine Leistungssteigerung der Datenbankalgorithmen erreicht werden.

Das erste Beispiel ist die Cell Broadband Engine (CellBE) [88], deren Konzept und Architektur zu der in dieser Arbeit genutzten Tomahawk-Plattform ähnlich ist. CellBE ist ein heterogener MPSoC, der einen GP-Prozessor, genannt *PowerPC Processing Element* (PPE), und acht anwendungsspezifische sogenannte *Synergistic Processing Elements* (SPE) enthält. Die PPE ist über einen zwei-level Cache an einen externen Speicher angebunden und bildet die Hauptverarbeitungseinheit, die auch als zentrale Steuereinheit für das Task-Scheduling verwendet wird. Die SPEs sind 128-Bit RISC Prozessoren, die jeweils einen lokalen 256-kB Speicher umfassen. Sie unterstützen vordefinierte SIMD- und Vektorinstruktionen für allgemeine arithmetische, logische und Vergleichsoperationen. Der Zugriff der SPEs auf den externen Speicher ist nur über den integrierten DMA-Controller möglich. Gedik u.a. implementieren das Sortieren [52] und den Join-Operator [53] auf der CellBE. Die Parallelisierung auf mehrere SPEs und das Ausnutzen von SIMD-Befehlen, um gleichzeitig vier 32-Bit Elemente zu vergleichen, erlaubt einen 4× und 8,3× höheren Durchsatz für das Sortieren bzw. den Join gegenüber einem Intel Xeon Prozessor.

Ein weiteres MPSoC zur Verarbeitung großer Datenmengen ist die von Oracle Labs vorgestellte *Data Processing Unit* (DPU) [2], die mehrere Verarbeitungseinheiten, genannt dpCores, integriert, die jeweils einen lokalen 32-kB Speicher sowie einen 16-kB Datencache und einen 8-kB Instruktionscache umfassen. Die dpCores enthalten einen 64-Bit MIPS-ähnlichen Basisbefehlssatz sowie vektorbasierte Lade- und Filterinstruktionen zur effizienten Abarbeitung von Datenbankoperatoren. Das Schlüsselelement ist das *Data Movement System* (DMS), welches komplexe Speicherzugriffsmuster zur Datenpartitionierung während des Datentransfers vom DRAM in die lokalen Speicher der dpCores erlaubt. Die DMS-Hardware wird über Software programmiert und ermöglicht somit die Beschleunigung von Datenbankoperatoren wie Filtern, Aggregation und Hashing. Die Chipimplementierung der DPU integriert insgesamt 32 dpCores, einen 256-kB L2 Cache, das DMS und eine Steuereinheit für das Leistungsmanagement bestehend aus zwei ARM Cortex-A9 und einem Cortex-M0 Prozessor. Mit der verwendeten 40-nm Technologie benötigt ein dpCore eine durchschnittliche Leistung von 51 mW bei einer Taktfrequenz von 800 MHz. Für eine Reihe von Standard SQL-Anfragen erreicht die DPU einen im Mittel 15× geringeren Energieverbrauch verglichen mit einem Intel Xeon Prozessor.

Gerade auf Grund der enormen Leistungsfähigkeit von x86-Prozessoren, beziehen sich viele Forschungsarbeiten auf die Implementierung der Datenbankoperatoren in solchen

hoch-performanten GP-CPUs. Die Programme sind dann durch intelligente Softwareanpassungen auf die Speicherarchitekturen mit Cache optimiert. Beispielsweise kann so die Leistungsfähigkeit von Sorted-Set Operationen [139, 142], Sortieren [33, 128], Joins [13, 23], Bitmap Operationen [60, 157, 117] und die Aggregation [120] um etwa eine Größenordnung gegenüber der nicht optimierten Variante verbessert werden.

Die Reduzierung der Speicherlatenz durch Cache-optimierte Datenbankalgorithmen und die intelligente Anordnung der Daten im Speicher ist ein erster Schritt zur Erhöhung der Leistungsfähigkeit der Systeme. Jedoch müssen auch die zugrunde liegende Prozessorhardware und das Verhalten des Speichers in die Implementierungen einbezogen werden, um zukünftige Leistungsanforderungen der datenintensiven Algorithmen zu erfüllen. MPSoCs bieten deshalb den Vorteil, nicht nur die Software hinsichtlich effizienter Speicherzugriffe zu optimieren, sondern die gesamte Hardwarearchitektur an die Anwendung anzupassen und so die Energieeffizienz zu steigern. Der in dieser Arbeit vorgestellte Datenbankbeschleuniger und dessen Integration in das Tomahawk MPSoC trägt dazu bei, eine Erhöhung der Energieeffizienz der Datenbank-Anfrageverarbeitung gegenüber den für x86-Prozessoren optimierten Algorithmen zu ermöglichen.

5.1.3 Tomahawk MPSoC

Konzept

Tomahawk [9] ist ein heterogenes Mehrkernsystem, das applikationsspezifische Hardware, Task-Scheduling und dynamisches Energiemanagement integriert. Die Steuerung der Plattform erfolgt über einen Host-Prozessor, genannt *Applikationsprozessor*, und einer zentralen Steuereinheit, die im Tomahawk-Konzept als *CoreManager* (CM) bezeichnet wird. Alle Komponenten sind mit einem externen Speicher und weiterer Peripherie über ein Network-on-Chip miteinander verbunden. Die eigentliche Datenverarbeitung wird durch verschiedenartige *Verarbeitungsmodule* (Processing Module, PM) realisiert, die ein oder mehrere *Verarbeitungseinheiten* (Processing Elements, PEs) enthalten können. Ein PE kann dabei ein RISC, DSP oder anderer Hardwarebeschleuniger (ASIC, ASIP) sein. Jedes PM enthält einen lokalen Speicher (SRAM), der von allen PEs innerhalb des Moduls erreichbar ist. Im Gegensatz zu einem Cache, ist die Anwendung für den genauen Speicherinhalt zuständig. Es ist auch möglich, mittels einer intelligenten Speicherarbitrierung, den Speicher in mehrere Blöcke aufzuteilen. Der Zugriff der PEs erfolgt dann nur auf den Blöcken, die durch Zuweisen eines bestimmten Adressraums definiert werden. Damit kann die Isolation der in den PEs verarbeitenden Daten gewährleistet werden. Der Speicherzugriff vom NoC wird durch einen DMA-Controller realisiert, der direkt über die Software der PEs oder den CM gesteuert wird.

Der Applikationsprozessor definiert den Ablauf der Anwendung mit Hilfe eines Datenflussgraphen (DFG) und den zugehörigen Tasks. Dazu zählen z. B. synchrone DFGs [98] oder Kahn Prozessnetzwerke [55]. Der CM erhält den DFG vom Applikationsprozessor und bildet diesen zur Laufzeit auf die Plattform ab, indem die Tasks den PEs zugewiesen werden. Zusätzlich führt der CM eine Datenabhängigkeitsanalyse durch, um die Parallelität der Tasks zu gewährleisten. Zudem passt der CM individuell für jedes PM die Versorgungsspannung und Taktfrequenz entsprechend der Gesamtlast des Systems an (DVFS), um die Energieaufnahme zu minimieren. Das Konzept des CM ist vergleichbar mit dem

Tabelle 5.1: Systemspezifikationen der Tomahawk-Generationen

	Tomahawk1 (2007)	Tomahawk2 (2013)	Tomahawk3 (2014)	Tomahawk4 (2015)
Applikationen	Videocodierung, Signalverarbeitung/SDR	Signalverarbeitung/SDR	DB-Anfrageverarbeitung	DB-Anfrageverarbeitung, Signalverarbeitung/SDR
NoC-Topologie	Bus (Crossbar)	Stern-Gitter	Stern-Gitter	Hexagon
Anzahl Kerne	12	20	5	15
Technologie	130 nm	65 nm	28 nm	28 nm
Chipfläche	100 mm^2	36 mm^2	18 mm^2	18 mm^2
Flächenanteil Speicher	67 %	27 %	20 %	35 %
Speichergröße pro PM	64 kB	64 kB	96 kB	128 kB
Ext. Speicherinterface	3× DDR	DDR2 (2× 128 MB)	LPDDR2 (2× 64 MB)	LPDDR2 (128 MB)
Taktung	Globaler Takt	GALS	GALS	GALS
Energiemanagement	–	DVFS, AVS	DVFS, AVS	DVFS, AVS
Max. Taktfrequenz	175 MHz	500 MHz	500 MHz	667 MHz
Max. Spannung	1,3 V	1,2 V	1,1 V	1,1 V
Mittlere Leistung	1,5 W	0,5 W	0,2 W	0,2 W

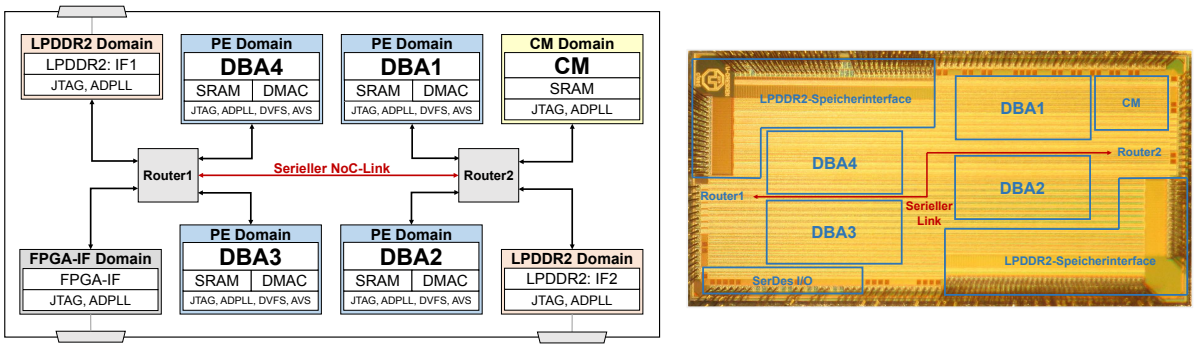
eines superskalaren Prozessors, gesehen auf einer höheren Abstraktionsebene. Ähnlich zu der Ausführung nebenläufiger Befehle, verarbeitet der CM mehrere parallele Tasks, indem diese auf die PEs zugewiesen werden.

Der ursprüngliche Schwerpunkt der Tomahawk-Plattform liegt bei der Abarbeitung von Algorithmen aus dem Bereich der digitalen Signalverarbeitung für drahtlose Kommunikationssysteme. Das Konzept kann jedoch einfach auf die Datenbank-Anfrageverarbeitung angewendet werden, die in gleicher Weise eine hohe Leistungsfähigkeit und Energieeffizienz des Systems benötigt. Datentransfers, Task-Scheduling und Energiemanagement können genauso vom CM verarbeitet werden. Der Datenflussgraph enthält nun die Datenbankoperatoren des Anfrageausführungsplans. Abschnitt 5.2 beschreibt später das genaue Vorgehen.

Tabelle 5.1 stellt die verschiedenen Generationen der Tomahawk MPSoCs gegenüber. Tomahawk1 [102] und Tomahawk2 [111] wurden bereits 2011 bzw. 2014 entwickelt und enthalten keine Datenbankbeschleuniger. In diesem Abschnitt werden deshalb nur die dritte und vierte Tomahawk-Generation präsentiert, die jeweils mehrere Instanzen des in Kapitel 4 vorgestellten DBA ASIP integrieren. Die Entwicklung dieser zwei Forschungschips erfolgte am Vodafone Lehrstuhl für Mobile Nachrichtensysteme der TU Dresden unter Mitwirkung des Autors.

Energiemanagement

Um eine individuelle Skalierung der Taktfrequenz zu erreichen, ist jedes PM in einer separaten Domain eingebettet, in der sich eine digitale Phasenregelschleife (ADPLL [74]) befindet, die den Takt für den Prozessor und die weitere Hardware bereitstellt. Damit nutzen die Domains einen lokalen synchronen Takt und sind global asynchron über das NoC verbunden sind (Globally Asynchronous - Locally Synchronous, GALS). Für die



(a) Blockdiagramm (b) Chipfoto

Abbildung 5.2: Tomahawk3 MPSoC

Skalierung der Versorgungsspannung ist jede PM-Domain an mindestens zwei separate Spannungsversorgungsleitungen angebunden, die im Bereich zwischen 0,6 und 1,1 V eingestellt werden können. Eine Hardwareeinheit (Power Management Controller, PMC), die vom CM angesteuert wird, ermöglicht das Umschalten zwischen den zwei vordefinierten Spannungsleitungen innerhalb von 50 ns [75]. Auf Grund der niedrigen Umschaltzeiten sind auch nur kurze Abschaltzeiten der PMs notwendig. Des Weiteren beeinflussen auftretende Prozess- und Temperaturschwankungen die Spannungspegel. Eine adaptive Spannungsskalierung (Adaptive Voltage Scaling, AVS) [111] wertet diese von *Hardware Performance Monitors* (HPM) detektierten Schwankungen aus und führt die Spannungspegel dynamisch nach.

Tomahawk3

Der Tomahawk3 MPSoC [61] ist ausschließlich zur Abarbeitung von Datenbankalgorithmen aus der Anfrageverarbeitung vorgesehen. Tomahawk3 enthält fünf heterogene Prozessoren, zwei LPDDR2-Speicherschnittstellen und ein FPGA-Interface, die über ein paketvermitteltes NoC miteinander verbunden sind. Der Chip wurde in einer 28 nm SLP CMOS Technologie von Globalfoundries hergestellt und integriert 16,85 Mio. NAND2-Gatter sowie 480 kB SRAM bei einer Chipfläche von $3\,\text{mm} \times 6\,\text{mm} = 18\,\text{mm}^2$. Abbildung 5.2 zeigt das Blockdiagramm und das Chipfoto des Tomahawk3.

Der CM ist ein Cadence Tensilica Xtensa LX5 Prozessor (aus [111]) und die PEs integrieren den DBA ASIP aus Kapitel 4. Ein PE enthält zwei Dual-Port Datenspeicher (DMEM) und einen Dual-Port Instruktionsspeicher (IMEM) mit einer Größe von jeweils 32 kB. Die zwei Ports der Speicher ermöglichen den gleichzeitigen Zugriff von PE- und von NoC-Seite. Die Speichergrößen wurden gewählt, um zunächst eine mit dem zuvor entwickelten Titan3D-Chip aus Kapitel 4.3.5 einheitliche Konfiguration zu erhalten. Für den Datentransfer über das NoC integriert jedes PE einen DMA-Controller mit einem Transferslot, der direkt vom PE oder CM konfiguriert wird. Die zwei LPDDR2-Speicherschnittstellen erreichen eine Bandbreite von jeweils 12,5 GBit/s und verbinden zwei 64 MB SDRAMs. Es können mehrere Chips über eine SerDes-Schnittstelle mit 5 GBit/s, die an einen FPGA angebunden ist, verbunden werden.

Die NoC-Architektur folgt einer Gitternetz-Topologie mit zwei Routern. Diese Topo-

logie ermöglicht eine hohe Skalierbarkeit und Flexibilität für zukünftige erweiterte Architekturen mit mehr Prozessoren. Zur Minimierung der Routinglatenzen sind die Chipkomponenten sternförmig an die Router angebunden. Ein serieller NoC-Link verbindet die zwei Router auf dem Chip und ermöglicht im Vergleich zu parallelen Links eine Überbrückung von größeren On-Chip-Distanzen ($> 5\,\text{mm}$) bei gleichzeitig geringen Energieverbrauch [76]. Da nur eine Verbindung zwischen den beiden Routern existiert, entspricht die Bisektionsbandbreite der Bandbreite des NoC B_{NoC}.

Tomahawk4

Gemäß der in Abschnitt 5.1.1 definierten Hauptkriterien für ein MPSoC zur effizienten Datenbank-Anfrageverarbeitung erfüllt die zuvor beschriebene Tomahawk3-Architektur noch nicht alle Punkte in vollem Maße. Zwar ist der DBA als angepasste Verarbeitungseinheit im Tomahawk3 integriert, jedoch kann das NoC mit nur einem Link zum Flaschenhals beim Datentransfer werden. Zudem unterstützt der im Tomahawk3 enthaltene DMA-Controller nur einen Transferslot. Damit ist das Laden der Daten mittels Double-Buffering nicht möglich. Aus diesen Gründen wurde ein weiterer Forschungschip entwickelt: Tomahawk4.

Der Tomahawk4 MPSoC [65] hat eine Chipfläche von $18\,\text{mm}^2$ und wurde ebenfalls in der 28 nm SLP CMOS Technologie von Globalfoundries hergestellt. Der Tomahawk4 folgt genau wie der Tomahawk3 dem allgemeinen Tomahawk-Konzept und integriert vier DBAs, einen CoreManager (CM), eine LPDDR2-Speicherschnittstelle und ein FPGA-Interface. Des Weiteren umfasst der Chip eine Cadence Tensilica 570T CPU als Applikationsprozessor und zusätzliche Peripherie (GPIO, UART). Wie später in Abschnitt 5.2 noch ersichtlich wird, kann der CM einen bereits vordefinierten Anfrageausführungsplan abarbeiten, ohne dass der Applikationsprozessor diesen explizit erstellen muss. Der Applikationsprozessor dient hier deshalb nur als GP-CPU mit Cache und erlaubt so einen Vergleich der Leistungsfähigkeit mit den DBAs und deren lokalen Speichern. Das Basisband-PM (BBPM) umfasst mehrere Hardwarebeschleuniger für rechenintensive Signalverarbeitungsalgorithmen und wird im Rahmen anderer Forschungsarbeiten untersucht. Das zugehörige Blockdiagramm des Tomahawk4-MPSoC ist in Abbildung 5.3 dargestellt. Die für die Verarbeitung von Datenbankoperatoren wesentlichen Änderungen bestehen in einer Weiterentwicklung des DMA-Controllers sowie in einem erweiterten NoC. Die folgenden Paragraphen beschreiben die Änderungen im Detail.

Das NoC ist in einer hexagonalen Topologie mit insgesamt acht Routern implementiert, die im Gegensatz zu der im Tomahawk3 vorhanden Zwei-Router Stern-Gitter-Topologie eine $5\times$ höhere Bisektionsbandbreite aufweist. Damit erlauben die zusätzlichen diagonalen NoC-Verbindungen eine verbesserte Aufteilung des Datenverkehrs und eine höhere Robustheit bei blockierten oder fehlerhaften NoC-Links. Die Router sind über parallele Links untereinander verbunden. Der NoC-Link zwischen den Routern R7 und R8 überbrückt eine größere Distanz von 4,5 mm und wurde deshalb als serieller Link implementiert (siehe auch Chipfoto in Abbildung A.4 im Anhang A).

Jedes PM integriert den DBA ASIP aus Kapitel 4 und zwei 32-kB Datenspeicher und einen 64-kB Instruktionsspeicher. Die Speicherkonfiguration wurde in Anlehnung an den Tomahawk3 gewählt. Die Speicher besitzen nun ebenfalls zwei Ports: ein Port wird zwi-

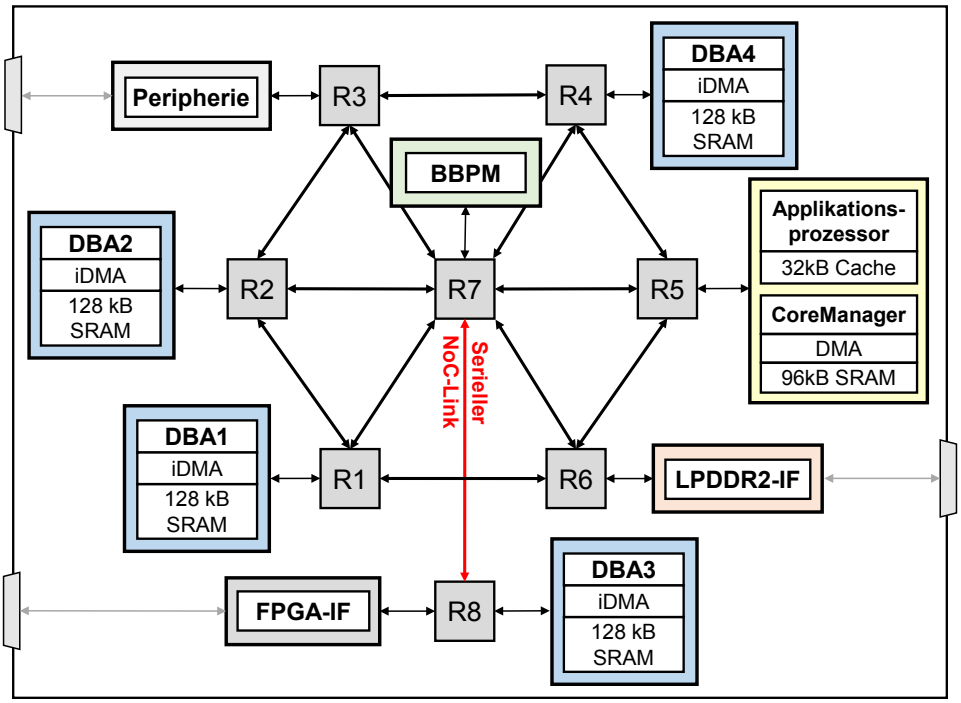


Abbildung 5.3: Tomahawk4 MPSoC Blockdiagramm

schen den beiden Prozessoren geteilt wobei der zweite Port an das NoC angebunden ist. Die Schnittstelle zwischen NoC und Speicher wird über einen *intelligenten DMA-Controller* (iDMA) realisiert, der die konventionelle Funktionalität des DMAC aus dem Tomahawk3 enthält, jedoch nun insgesamt acht Transferslots für die parallele Steuerung mehrere Datentransfers unterstützt. Der iDMA konfiguriert die Slots baut damit virtuelle Kanäle zwischen den PEs oder zwischen PE und globalen Speicher auf. Dieses Kanalkonzept gewährleistet die Isolation mehrerer gleichzeitiger Datentransfers und der damit einhergehenden Speicherbereiche. Die Datenpuffer sind per FIFO-Speicher in den jeweiligen Speicher des PM abgebildet. Eine auf Credits basierte Flusskontrolle verhindert das Überlaufen der FIFO-Speicher [110]. Zusätzlich kann der aktuelle Füllstand der FIFO-Speicher abgefragt werden, um den Fortschritt beim Datentransfer nachzuverfolgen. Das ermöglicht z. B. eine zeitliche Überlappung von Datentransfer und Datenverarbeitung und damit eine Reduzierung der gesamten Ausführungszeit.

5.2 Scheduling- und Parallelisierungsansatz

In Abschnitt 2.2.1 wurde der allgemeine Ablauf der Datenbank-Anfrageverarbeitung beschrieben. Dabei erzeugt das DBMS aus der SQL-Anfrage einen Anfrageausführungsplan, der aus einzelnen aufeinander folgenden relationalen Operatoren besteht. Im Folgenden wird nun das allgemeine Vorgehen beim Abbilden der gegebenen Operatoren auf die Tomahawk-Plattform beschrieben.

5.2.1 Datenflussgraphen

Wie bereits in Abschnitt 5.1.3 angesprochen, sieht das Tomahawk-Konzept den CoreManager (weiterhin als Scheduler bezeichnet) als einen zentralen dedizierten Prozessor für das Task-Scheduling vor. Der Scheduler arbeitet dabei einen *Datenflussgraphen* ab, dessen Knoten die Tasks und die Kanten die Datentransfers repräsentieren. Ein Task führt Berechnungen auf den gegebenen Eingangsdaten aus und produziert Ergebnisdaten. Zudem sind Datenabhängigkeiten zwischen den Tasks direkt aus dem Graphen ablesbar. Prinzipiell stellt der Anfrageausführungsplan mit den enthaltenen Datenbankanfragen ebenfalls einen Datenflussgraphen dar. Dieser Graph ist gerichtet, d. h. jeder Task wird nur einmal ausgeführt. In der Graphentheorie entspricht dies einem sogenannten *azyklischen Datenflussgraph* (Directed Acyclic Graph, DAG) [140]. Der Anfrageausführungsplan kann demnach einfach in einen DAG mit Tasks überführt werden, den der Scheduler abarbeitet.

Es wird angenommen, dass dem Scheduler für jeden Datenbankoperator ein vordefinierter DAG zur Verfügung steht. Jedoch kann der Scheduler den gegebenen DAG weiter modifizieren, da z. B. durch eine bestimmte Anordnung der Eingangsdaten neue Tasks hinzukommen oder bestehende wegfallen. Diese zusätzlichen Modifikationen sind aber nicht zur Laufzeit, sondern nur vor der eigentlichen Abarbeitung des DAG notwendig (Scheduler-Overhead). Zudem müssen dem Scheduler zu Beginn die Anzahl verfügbarer DBAs, deren durch die ISE unterstützten Datenbankoperatoren, sowie die Speicherbereiche der Ein- und Ausgangsdaten bekannt sein. Eine detaillierte Beschreibung der parallelen Implementierung für die DAGs ausgewählter Datenbankoperatoren folgt in Abschnitt 5.3.

5.2.2 Scheduling der Datenbankoperatoren

Die Informationen über den abzuarbeitenden Datenbankoperator aus dem Anfrageausführungsplan werden dem Scheduler über eine übergeordnete Instanz bekannt gegeben. Diese beinhalten den auszuführenden Operator, die Speicherbereiche der Ein- und Ausgangsdaten und deren Kardinalitäten. Der Scheduler arbeitet nun den DAG ab, indem voneinander unabhängige Tasks auf gerade frei verfügbare DBAs zugewiesen und dort gleichzeitig abgearbeitet werden. Die Zuweisung erfolgt durch das Senden von Kontrollinformationen. Diese umfassen den auszuführenden Datenbankoperator, die für den jeweiligen Prozessor relevante Speicherbereiche im globalen Speicher und die Anzahl der zu verarbeitenden Daten. Dieses hier verwendete Programmiermodell und die Mehrkernarchitektur des Tomahawk ist vergleichbar mit dem OpenCL-Standard [109]. Das zugrundeliegende Modell von OpenCL geht ebenfalls von einem Host-Prozessor aus, der die parallele Abarbeitung einer Anwendung auf mehreren heterogenen Verarbeitungseinheiten steuert.

Es wird angenommen, dass sich die Eingangsrelationen der Datenbank oder die relevante Teilmenge zu Beginn nicht wie zuvor in den lokalen sondern im globalen Speicher befinden. Deshalb ist zunächst der Transfer der Eingangsdaten zu den lokalen Speichern der DBAs notwendig. Dieser Transfer wird mit Hilfe des zum jeweiligen Prozessor zugehörigen DMA-Controllers realisiert, der entweder durch den Scheduler oder direkt durch den DBA selbst konfiguriert werden kann. Beispielsweise ist die Steuerung des Datentransfers durch den Scheduler von Vorteil, wenn eine globale Kenntnis erforderlich ist, d. h. wenn Daten teilweise für die Dauer mehrerer Tasks bis zum Ende des gesamten Algorithmus immer

zwischen den lokalen Speichern der DBAs verbleiben. In diesem Fall kann die DMA-Einheit des Tomahawk3 MPSoC verwendet werden. Besteht hingegen keine Möglichkeit auf Datenlokalität und die Daten werden in einer Softwarepipeline und einem Double-Buffering-Ansatz geladen, wird der iDMA aus dem Tomahawk4 verwendet. Der iDMA unterstützt mehrere Slots für die gleichzeitige Steuerung mehrerer Datentransfers. Die Prozessoren übernehmen dann selbst die Ansteuerung ihres zugehörigen iDMA, da die zusätzlichen Verzögerung, die beim Senden von Kontrollinformationen über das NoC zum Scheduler entstehen, die Leistungsfähigkeit des Algorithmus beeinflussen würden. In gleicher Weise erfolgt das Zurücksenden der Ergebnisdaten von den lokalen in den globalen Speicher am Ende eines jeden Tasks bzw. am Ende des gesamten Algorithmus. Ist bereits bekannt, dass nachfolgende Datenbankoperatoren auf den Ergebnisdaten des aktuellen Algorithmus arbeiten, können die Daten auch direkt in den lokalen Speichern der DBAs verbleiben.

5.2.3 Parameter

Wie zuvor beschrieben, arbeitet der Scheduler überwiegend online. Die Menge der Tasks steht zwar vor der Ausführung fest, jedoch nicht die genauen Zuweisungen auf die Prozessoren. Der Scheduler reagiert dann nur auf Statussignale, die die Prozessoren per Interrupts senden bzw. die der Scheduler aktiv zyklisch abfragt. Diese Informationen umfassen die folgenden Parameter:

- Status: Leerlauf, Verarbeitung
- Tasklaufzeit
- Spannungs- und Frequenzlevel

Anhand dieser Parameter und dem vorliegenden DAG trifft der Scheduler Entscheidungen über den Zeitpunkt der nächsten Task-Zuweisungen. Die Tasklaufzeiten werden dabei mit Hilfe von Timern ermittelt, die in den DBAs integriert sind. Durch Einstellen von Spannungs- und Frequenzlevel der einzelnen Prozessoren kann der Scheduler ebenfalls den Energieverbrauch des Gesamtsystems minimieren. Beispielsweise kann die Taktfrequenz der Prozessoren reduziert werden, die sich im Leerlauf befinden oder vergleichsweise kürzer laufende Tasks abarbeiten.

5.3 Parallelisierung der Datenbankoperatoren auf Taskebene

Dieser Abschnitt erläutert die Implementierungen der Datenbankoperatoren für deren parallele Ausführung auf mehreren DBAs des Tomahawk MPSoC. Dabei existieren zwei Methoden der Parallelisierung: mit einem *Intra-Operator-Ansatz* werden die Daten partitioniert und auf mehrere Prozessoren aufgeteilt, wobei jeder Prozessor auf dem gleichen Algorithmus arbeitet. Im Gegensatz dazu führt der *Inter-Operator-Ansatz* voneinander unabhängige Algorithmen nebenläufig auf mehreren Prozessoren mit unterschiedlichen

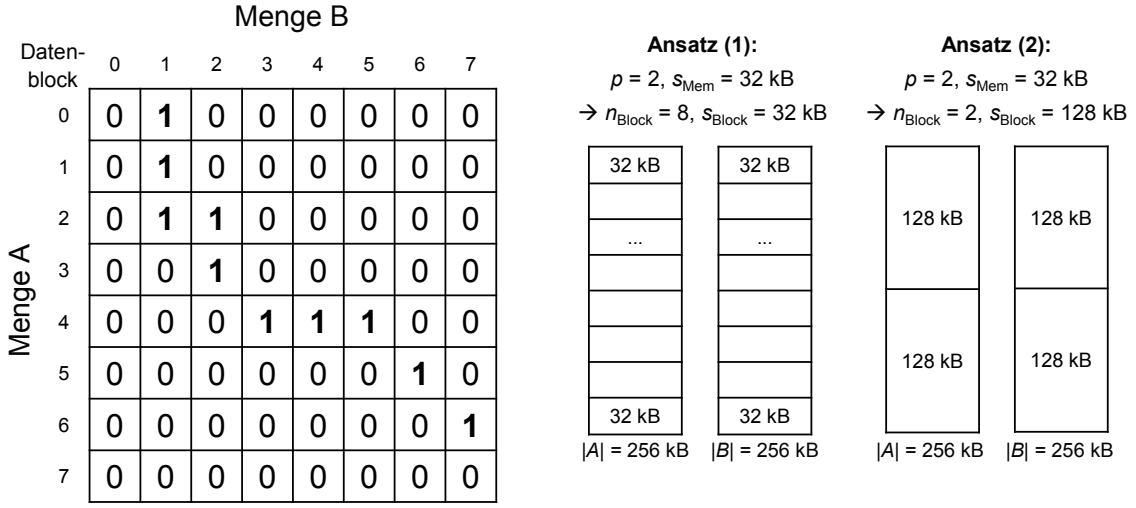
Daten aus. Letzteres ist für komplexe Anfragen oder z. B. für den in Kapitel 4.4 verwendeten Algorithmus zur Anfrageoptimierung geeignet. Die aufeinander folgenden i Intersektionen zur Schätzung der Selektivitäten können auf p Prozessoren gleichzeitig ausgeführt werden. Da in diesem Fall keine Datenabhängigkeiten existieren, ist ein Speedup von $i/(\lfloor \frac{i}{p} \rfloor + 1)$ möglich, der für große i gegen p geht.

In dieser Arbeit wird der Inter-Operator-Ansatz verwendet, um die Möglichkeiten der Parallelisierung für bereits in den vorherigen Kapiteln verwendeten Datenbankoperationen zu untersuchen: Sorted-Set Operationen, Merge-Sort und Hash-Join. Diese Algorithmen verlangen jeweils unterschiedliche Methoden zur Parallelisierung. Für die Sorted-Set Operationen ist das Bilden der Datenpartitionen nicht trivial. Es ist deshalb ein zusätzlicher Zeitaufwand des Schedulers zu erwarten. Für das eigentliche Merge-Sort ist eine beliebige Erhöhung der Elementanzahl nicht vorgesehen. Der gewählte parallele Sortieransatz muss deshalb höhere Elementanzahlen unterstützen sowie eine effiziente Lastverteilung auf den parallelen Prozessoren ermöglichen. Der Hash-Join weist theoretisch keine Limitierungen bei der Parallelisierung auf. Lediglich liegt die Herausforderung bei der Implementierung der Hashtabelle, da diese die Größe der Eingangsrelationen einnimmt sowie eine verhältnismäßig hohe Anzahl von gleichzeitigen Zugriffen unterstützen soll. Die folgenden Abschnitte erläutern nun die parallelen Implementierungen der Datenbankoperatoren.

5.3.1 Sorted-Set Algorithmen

Die parallele Abarbeitung der beiden sortierten Eingangslisten für die Sorted-Set Algorithmen ist zunächst nicht trivial. Eine einfache Einteilung der Listen in mehrere gleich große Blöcke führt dazu, dass auch nicht gegenüberliegenden Partitionen der beiden Mengen übereinstimmenden Elemente enthalten können. Das bedeutet, dass die Listen vor der eigentlichen Ausführung analysiert werden müssen, um die Grenzen der Partitionen entsprechend zu verschieben. Ebenso wäre eine Nachverarbeitung denkbar, die die restlichen Ergebniselemente in der Ausgangsliste findet. In anderen Arbeiten existieren zahlreiche Ansätze für die parallele Abarbeitung der Intersektion, z. B. mit Hilfe einer Hashtabelle [132] oder durch Definieren von variablen Trennelementen, die die Partitionen markieren [40]. Die Hashtabelle bietet sich vorzugsweise bei unsortierten Datensätzen an. Der zweite Ansatz hätte einen hohen Aufwand für den Scheduler zur Folge, da das Suchen der Trennelemente datenabhängig ist. Dies erfordert das Laden einer unbestimmten Anzahl von Elementen aus dem globalen Speicher.

Die Methode aus [40] kann jedoch für die Tomahawk-Plattform adaptiert werden, sodass die Anzahl der Speicherzugriffe des Schedulers auf ein Minimum reduziert wird. Dazu werden die Positionen der Trennelemente fest vorgegeben, d. h. die beiden Eingangsmengen A und B mit den Listenlängen $|A|$ und $|B|$ sind in jeweils n_{Block} Datenblöcke einzuteilen, die alle die gleiche Größe s_{Block} aufweisen. Die Sorted-Set Operationen müssen nun auf alle Datenblöcke angewendet werden, deren Wertebereiche sich überlappen. Nur in diesen Datenblöcken befinden sich Ergebniselemente. Alle anderen Blöcke müssen nicht verarbeitet werden, da hier die Ergebnismenge leer ist. Der Scheduler erzeugt dazu eine sogenannte *Adjazenzmatrix* wie am Beispiel der Intersektion in Abbildung 5.4a zu sehen. Dazu werden jeweils die ersten und letzten Elemente der Datenblöcke aus den Mengen A und B verglichen, um die überlappenden Blöcke zu finden. Die Elemente der Matrix geben dann



(a) Adjazenzmatrix am Bsp. $n_{Block} = 8$, $|A| = |B|$: eine „1“ kennzeichnet voneinander abhängige Datenblöcke

(b) Einteilung der Eingangslisten in Datenblöcke nach zwei Ansätzen (T3 und T4)

Abbildung 5.4: Parallele Implementierung der Sorted-Set Intersektion

mit einer „1“ an, wenn die jeweiligen Datenblöcke Abhängigkeiten aufweisen und deshalb zu verarbeiten sind.

Da die Daten in den Mengen sortiert sind, ergibt sich immer eine zusammenhängende Kette von Einsen in der Adjazenzmatrix. Überlappen sich wie in der Beispielmatrix die Blöcke $(0, 1)$, jedoch nicht die Blöcke $(0, 2)$, kann keine weitere Überlappung in Zeile 0 folgen. Der Scheduler wird deshalb als nächstes direkt den Block $(1, 1)$ überprüfen. Im Folgenden soll nun zur Vereinfachung weiterhin $|A| = |B|$ gelten. Die Annahme ändert nicht die Implementierungsmethodik, lediglich ist die Adjazenzmatrix immer quadratisch.

Bei der Auswahl von Anzahl und Größe der Datenblöcke muss ein Kompromiss zwischen verfügbaren Speicher und Parallelität eingegangen werden. Entweder passen die Blöcke in den lokalen Speicher der Prozessoren, oder soll sich die Anzahl der Blöcke nach der Anzahl der Prozessoren richten, um eine maximale Parallelität zu erreichen. Zusammengefasst ergeben sich damit zwei mögliche Implementierungsansätze:

(1) Die Größe eines Blocks s_{Block} ist kleiner oder gleich der Größe des lokalen Datenspeichers[1] s_{Mem} ($s_{Block} \leq s_{Mem}$). Damit bestimmt die Größe des Speichers die Anzahl der Blöcke. Es ist ein einfacher Datentransfer zu den Prozessoren durch den Scheduler möglich. Hier kann deshalb das Tomahawk3 MPSoC (T3) für die Evaluierung verwendet werden. Die Vorverarbeitungskosten (Scheduler-Overhead) sind von der Größe der Eingangslisten abhängig.

(2) Die Größe eines Blocks s_{Block} ist durch das Verhältnis der Gesamtlistenlänge $|A|$ zur Anzahl der Prozessoren p gegeben ($s_{Block} = \frac{|A|}{p}$). Die Anzahl der Blöcke entspricht dann der Anzahl der Prozessoren ($n_{Block} = p$). Das Laden der Daten erfolgt mit einer Softwarepipeline und durch Double-Buffering innerhalb der Prozessoren, da die zu

[1] s_{Mem} entspricht der notwendigen Größe des lokalen Datenspeichers für eine der beiden Mengen A oder B.

verarbeitenden Listen größer als der lokale Speicher sind. Damit ist die Verwendung des iDMA und des Tomahawk4 MPSoC (T4) notwendig. Die Vorverarbeitung durch den Scheduler ist von der Anzahl der Prozessoren abhängig.

Abbildung 5.4b skizziert die verschiedenen Blockeinteilungen am Beispiel für zwei Mengen mit jeweils 256 kB (64 000 32-Bit Elemente). Die Parallelisierung der Sorted-Set Vereinigung und Differenz kann dabei ohne Einschränkungen auf diese Implementierungen der Intersektion angewendet werden.

Die Anzahl der Tasks entspricht der Anzahl der Einsen in der Adjazenzmatrix. Demnach enthält nun der zugehörige DAG nach der Vorverarbeitung durch den Scheduler n_{Task} unabhängige Tasks, die nebenläufig auf den p Prozessoren ausgeführt werden können. Die Taskanzahl für die beiden Ansätze (1) und (2) ergibt sich bei gleichmäßig verteilten Werten (nur Einsen in der Hauptdiagonalen der Adjazenzmatrix) in den Eingangsdaten mit

$$n_{Task(1)} = \frac{|A|}{s_{Mem}} \quad \text{bzw.} \quad n_{Task(2)} = p. \tag{5.1}$$

Die maximale Taskanzahl wird erreicht, wenn die Hauptdiagonale und eine erste Nebendiagonale der Adjazenzmatrix Einsen enthält:

$$n_{Task(1),max} = \frac{2|A|}{s_{Mem}} - 1 \quad \text{bzw.} \quad n_{Task(2),max} = 2p - 1. \tag{5.2}$$

Die Daten sind dann auch gleichmäßig verteilt, wobei zwischen den Werten der Mengen ein Offset besteht, der zu den zusätzlichen Überlappungen führt. Mit dem Durchsatz für die Sorted-Set Algorithmen D_{Set} ist die Tasklaufzeit gegeben mit

$$t_{Task(1)} = \frac{2s_{Mem}}{D_{Set}} \quad \text{bzw.} \quad t_{Task(2)} = \frac{2|A|}{pD_{Set}}. \tag{5.3}$$

Die Gesamtverarbeitungszeit (ohne Datentransfer) ist dann bei einer gleichmäßigen Datenverteilung für beide Ansätze identisch und beträgt

$$t_{Set(1)} = t_{Set(2)} = n_{Task} t_{Task} = \frac{2|A|}{D_{Set}}. \tag{5.4}$$

Der Durchsatz D_{Set} hängt linear mit der Anzahl der Elemente ab. Demnach ergibt sich auch für die parallele Umsetzung der Sorted-Set Algorithmen eine lineare Abhängigkeit mit der Elementanzahl. Trotz der gleichen Gesamtverarbeitungszeiten beider Ansätze ergeben sich individuelle Vorteile. Bei Ansatz (1) können die Daten direkt in den lokalen Speichern verbleiben, wenn aufeinanderfolgende Tasks aus der gleichen Zeile oder Spalte der Adjazenzmatrix abgearbeitet werden (Datenlokalität). Ansatz (2) hat hingegen den Vorteil, dass der Datentransfer und die Verarbeitung teilweise gleichzeitig stattfinden können.

5.3.2 Merge-Sort

Betrachtet man das Vorgehen beim Sortieren mittels Merge-Sort in Abbildung 2.4b von Kapitel 2.2.3, dann ist eine offensichtliche Taskparallelität zu erkennen. Das Verschmelzen

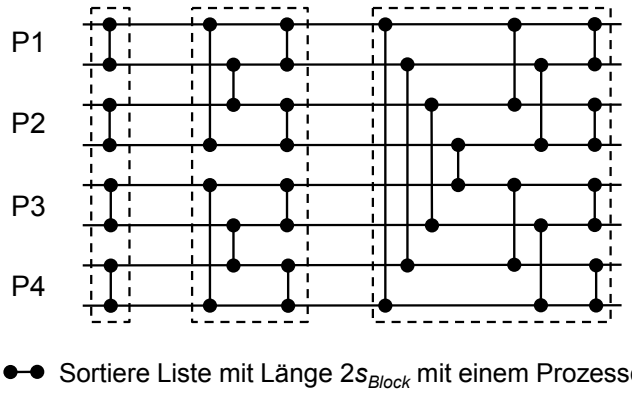


Abbildung 5.5: Bitonisches Merge-Sort mit Lastverteilung für $p = 4$ Prozessoren. Der Graph umfasst acht Datenblöcke. Der Sortiervorgang ist in drei Untergruppen und sechs Teilstufen aufgeteilt. Jede Teilstufe hat vier unabhängige Tasks.

zweier Teilfolgen entspricht dem Sortieren von zwei Datenblöcken mit einem Prozessor und kann nun für alle Blöcke einer Ebene gleichzeitig durchgeführt werden. Der Nachteil ist, dass sich die Anzahl der zu sortierenden Datenblöcke mit jeder Ebene halbiert. Am Ende existieren noch zwei Blöcke, die von nur einem Prozessor sortiert werden müssen. Die Gesamtanzahl der Tasks beträgt damit $2(2n_{Block} - 1)$. Des Weiteren verlangt das Merge-Sort im Gegensatz zu den Sorted-Set Algorithmen, dass sich der gesamte zu sortierende Block im lokalen Speicher des Prozessors befinden muss. Damit lässt sich das Prinzip des Double-Bufferings nicht mehr anwenden und die maximale Listengröße wäre durch den lokalen Speicher des Prozessors limitiert.

Dies motiviert die Entwicklung eines weiteren parallelen Merge-Sorts, dass die Last gleichmäßig auf die Prozessoren verteilt. Solche Methoden existieren bereits in anderen Arbeiten [13, 33], die auf einem bitonischen Sortierverfahren nach Batcher [19] aufbauen. Dieses Konzept kann für die Implementierung auf dem Tomahawk angepasst werden. Dazu wird die zu sortierende Gesamtliste der Länge N in n_{Block} Datenblöcke der Größe s_{Block} eingeteilt. Es sind nun immer zwei Datenblöcke mit je s_{Block} Elementen von einem Prozessor entsprechend dem Sortiergraphen in Abbildung 5.5 zu verarbeiten. Dieser Graph stellt gleichzeitig den im Scheduler abgespeicherten DAG dar und kann nach einem gegebenen Algorithmus auf eine beliebige Anzahl von Prozessoren erweitert werden. Die Gesamtanzahl an Tasks ergibt sich aus der Anzahl von Datenblöcken:

$$n_{Task} = \frac{n_{Block}}{4}(\log^2 n_{Block} + \log n_{Block}), \tag{5.5}$$

Die Tasks sind dabei in $\log n_{Block}$ Untergruppen bzw. in $\frac{2n_{Task}}{n_{Block}}$ Teilstufen aufgeteilt. In jeder Teilstufe existieren dann immer $\frac{1}{2}n_{Block}$ unabhängige Tasks, die nebenläufig abgearbeitet werden können. Im Vergleich zur linearen Komplexität der Taskanzahl beim einfachen parallelen Merge-Sort, wächst hier die Taskanzahl mit $\mathcal{O}(n_{Block} \log^2 n_{Block})$.

Für eine ideale Parallelität, wie in Abbildung 5.5 zu sehen, muss demnach $N \leq ps_{Mem}$ gelten, d. h. die Größe der Gesamtliste ist durch die Anzahl der Prozessoren p und der Größe des lokalen Speichers s_{Mem} begrenzt. Es gilt dann weiterhin $n_{Block} = 2p$ und $s_{Block} = \frac{N}{n_{Block}} < \frac{1}{2}s_{Mem}$. Es ist damit eine absolute Lastverteilung über alle Prozessoren vorhanden. Dieser Fall hat auch den Vorteil, dass Datenlokalität ausgenutzt werden

kann. Beispielsweise kann Prozessor P1 für die Dauer des gesamten Algorithmus den ersten Block im lokalen Speicher halten. Kann die Gesamtmenge nicht mehr auf die lokalen Speicher aufgeteilt werden ($N > ps_{Mem}$), muss trotzdem $s_{Block} < \frac{1}{2}s_{Mem}$ gelten. In diesem Fall muss ein Prozessor mehrere unterschiedliche Blöcke der gleichen Teilstufe verarbeiten. Im Beispiel der Abbildung 5.5 tritt dieser Fall auf, wenn bei gleichbleibender Gesamtlistenlänge weniger als vier Prozessoren für den Graphen verfügbar wären.

Zusammenfassend kann das parallele Merge-Sort für die folgenden drei Fälle implementiert werden:

(1) Einfaches Merge-Sort ohne Lastverteilung und mit Datenlokalität: Die Anzahl der zu sortierenden Elemente ist durch die Größe des lokalen Speichers limitiert ($N \leq s_{Mem}$). Mit jeder Stufe halbiert sich die Anzahl der parallel abzuarbeitenden Tasks. In der letzten Stufe ist keine Parallelität mehr vorhanden und es muss nur noch ein Prozessor sortieren. Beim Wechseln in die nächste Stufe können die Daten teilweise in den lokalen Speichern der Prozessoren verbleiben. Auf Grund der Datenlokalität konfiguriert der Scheduler die DMA-Controller in den PMs und initiiert die Datentransfers. Es ist eine Verarbeitung mit dem DMAC aus dem Tomahawk3 möglich.

(2) Merge-Sort mit Lastverteilung und Datenlokalität: Es wird das bitonische Sortierverfahren angewendet. Es sind in jeder Stufe mindestens p unabhängig Tasks vorhanden. Um jedoch noch die Datenlokalität auszunutzen, ist die Gesamtlistenlängen mit $N \leq ps_{Mem}$ beschränkt. Auch hier übernimmt der Scheduler die Konfiguration der DMA-Controller. Es ist ebenfalls eine Verarbeitung mit dem Tomahawk3 möglich.

(3) Merge-Sort mit Lastverteilung und ohne Datenlokalität: Dieses Sortierverfahren ist identisch zu Fall (2). Die Daten müssen jedoch nach jedem Sortiervorgang wieder in den globalen Speicher zurückgeschrieben werden. Die Gesamtlistenlänge ist nicht durch den Algorithmus limitiert ($N > ps_{Mem}$). Der Scheduler bildet die Tasks auf die PMs ab, indem diesen nur der Index auf die jeweiligen Datenpartitionen zugewiesen wird. Mit Hilfe des iDMA aus dem Tomahawk4 initiiert dann jede PM den Datentransfer, wobei eine zeitliche Überlappung von Datenverarbeitung und dem Senden der Daten möglich ist, um die Ausführungszeit zu verkürzen.

Vergleicht man die Komplexität der Implementierungen von Fall (1) mit Fall (2) bzw. (3) in Bezug auf die Anzahl der Prozessoren, liegt der Vorteil bei dem einfachen Merge-Sort. Setzt man voraus, dass die Größe der zu sortierenden Menge konstant bleibt und beliebig viele Prozessoren verfügbar sind, die Gesamtmenge also ideal auf die lokalen Speicher verteilt werden kann ($n_{Block} = 2p$), dann beträgt die Anzahl der noch sequentiell abzuarbeitenden Tasks für das einfache und das Merge-Sort mit Lastverteilung $\log p + 1$ bzw. $\frac{1}{2}(\log^2 p + 3\log p + 2)$. Erhöht man die Anzahl der Prozessoren, wächst damit die Anzahl der Tasks für das Merge-Sort mit Lastverteilung quadratisch mit dem Logarithmus und beim einfachen Merge-Sort nur logarithmisch. Da die Größe der Gesamtmenge in diesem Fall konstant bleibt, ist das Einbeziehen der logarithmischen Zeitkomplexität für die eigentliche Sortierung nicht notwendig (siehe Kapitel 4.2.4). Es folgen weitere Auswertungen in Kapitel 5.4.1 und 6.2.2.

5.3.3 Hash-Join

Wie in Kapitel 2.2.3 beschrieben, soll der Equi-Join auf zwei Relationen R und S ausgeführt werden. Die Relationen enthalten jeweils unsortierte Tupel mit einem 32-Bit Schlüssel und einer 32-Bit Payload. Zu Beginn befinden sich alle Tupel im globalen Speicher. Es wird $|R| \leq |S|$ angenommen, sodass die Relation R als kleinere von beiden in die Hashtabelle abgebildet werden kann. Aus diesem Grund weist die Hashtabelle $|R|$ Bucketeinträge auf und muss wegen ihrer Größe auch im globalen Speicher abgelegt werden. Dies stellt die Implementierung auf dem Tomahawk vor mehrere Herausforderungen:

DRAM-Zugriffe Im Vergleich zum lokalen Speicher der PMs, sind DRAM-Zugriffe in Bezug auf Durchsatz und Latenz weitaus weniger performant. Deshalb ist es nicht möglich, dass sich alle Prozessoren eine Hashtabelle teilen, da dies eine Synchronisation[2] der einzelnen Zugriffe voraussetzt. Jeder Prozessor muss deshalb eine eigene Hashtabelle zugewiesen bekommen.

Speicher-Mapping Der DRAM des Tomahawk ist für die PMs nur über DMA-Zugriffe zu erreichen und kann deshalb nicht in deren Adressraum abgebildet werden. Es sind deshalb zu Beginn die Adressen der individuellen Hashtabellen den Prozessoren durch den Scheduler bekanntzugeben.

Eingangsdaten Es wird angenommen, dass der Scheduler keine vorherige Kenntnis über die Verteilung der Eingangsdaten oder die Anzahl der Duplikate hat. Es ist daher nicht bekannt, wie viele Kollisionen in der Hashtabelle auftreten. Deshalb müssen die Bucketgrößen dynamisch anpassbar sein.

Kollisionsbehandlung Das Abbilden mehrerer Tupel auf den gleichen Bucket verursacht eine Kollision in der Hashtabelle. In der Literatur existieren bereits eine Reihe von Standardverfahren zur Kollisionsbehandlung [43]. Zum Beispiel können alle Einträge eines Buckets durch eine verkettete Liste verknüpft werden (Chaining). Der Aufwand für das Suchen eines Tupels ist dann proportional zum Lastfaktor r_{LF}, d. h. dem Verhältnis aus Tupelanzahl $|R|$ zur Bucketanzahl m (siehe Gleichung 2.2). Bei einer Erhöhung von $|R|$ ist auch gleichzeitig m zu erhöhen, um die konstante Zeitkomplexität $\mathcal{O}(1)$ für das Suchen eines Tupels zu gewährleisten. Des Weiteren liegen auf Grund der verketteten Liste die Bucketeinträge im Speicher nicht auf aufeinanderfolgenden Adressen. Es sind deshalb höhere Zugriffszeiten beim Lesen der Hashtabelle im DRAM zu erwarten. Außerdem benötigt die verkettete Liste zusätzlichen Speicherplatz für die Zeiger, die auf das nächste Listenelement verweisen.

Eine weitere Methode zur Kollisionsbehandlung berechnet vor dem eigentlichen Einfügen der Tupel in die Hashtabelle die Anzahl der Kollisionen [17, 92]. Damit sind die Bucketgrößen und die Strukturierung der kompletten Hashtabelle im Voraus bekannt, sodass alle Bucketeinträge auf benachbarten Speicheradressen abgelegt werden können. Der Aufbau der Hashtabelle wird mit Hilfe eines Histogramms und einer Präfixsumme

[2] Zur Synchronisation können sogenannte *Latches* verwendet werden, die über ein gesetztes Bit angeben, ob der zugehörige Bucket gerade beschrieben wird. [23]

realisiert. Die konstante Zeitkomplexität für das Suchen eines Tupels ist damit immer gegeben.

Ein weiteres Verfahren zur Kollisionsbehandlung nutzt die *offene Adressierung*. Jeder Bucket kann nur ein Tupel aufnehmen. Tritt eine Kollision auf, wird mit Hilfe einer zweiten Hashfunktion eine alternative Position in der Hashtabelle gesucht. Ist auch dieser Bucket belegt, wird eine dritte Hashfunktion verwendet, usw. Die Anzahl der Hashfunktionen entspricht dann der Bucketanzahl m. Daraus folgt, dass immer mindestens so viele Buckets wie Tupel vorhanden sein müssen: $|R| \leq m$ bzw. $r_{LF} \leq 1$. Die Zeitkomplexität für das Suchen beträgt $\mathcal{O}(\frac{1}{1-r_{LF}})$ und steigt damit mit dem Lastfaktor. Spezialfälle für die offene Adressierung sind das Cuckoo-Hashing [118] oder das Hopscotch-Hashing [72], die im Falle einer Kollision bereits eingefügte Tupel auf eine andere Position in der Hashtabelle verschieben.

Die Methoden der offenen Adressierung weisen viele Nachteile im Hinblick auf eine mögliche Umsetzung auf der Tomahawk-Plattform auf. Zum einen sollten die Anzahl der Zugriffe auf die Hashtabelle auf Grund der hohen Latenz des DRAM minimiert werden. Zum anderen ermöglicht der Datenbankbeschleuniger die effiziente Ausführung einer zuvor konfigurierten Hashfunktion, deren dynamische Anpassung ebenfalls zusätzlichen Aufwand verursachen würde. Anhand der zuvor erfolgten Analysen sind für das Erreichen einer hohen Leistungsfähigkeit des Hash-Joins auf der gegebenen Tomahawk-Architektur die Kollisionsbehandlungen basierend auf einer verketteten Listen sowie der Hashtabelle mit Histogramm und Präfixsumme vorzuziehen.

Datenpartitionierung Für eine parallele Abarbeitung des Hash-Joins müssen die Eingangsrelationen partitioniert werden. Bei einer naiven Einteilung der nicht sortierten Relationen R und S in p Datenblöcke müssten alle Blöcke aus R mit allen Blöcken aus S verarbeitet werden. Dies hätte einen quadratischen Aufwand $\mathcal{O}(p^2)$ zur Folge. Ähnlich dem *Radix-Clustering* aus [105] kann aber die Partitionierung anhand der Hashwerte vorgenommen werden. Die Partitionen enthalten dann nur die Tupel, deren Hashwerte in einen vorgegebenen Bereich fallen. Damit wird jeder Datenblock aus R mit genau einem Datenblock aus S verarbeitet, wobei sich ein Aufwand von $\mathcal{O}(p)$ ergibt.

Um den Implementierungsansatz zu finden, der für die Tomahawk-Architektur im Hinblick auf die Leistungsfähigkeit des Hash-Joins am besten geeignet ist, sollen im Folgenden zwei Ansätze untersucht und verglichen werden, die sich durch den Aufbau der Hashtabelle unterscheiden:

- Hash-Join mit Radix-Clustering und Hashtabelle mit Kollisionsbehandlung durch verketteter Liste
- Hash-Join mit Radix-Clustering und Hashtabelle basierend auf Histogramm und Präfixsumme

Die folgenden Abschnitte erläutern die beiden Implementierungsansätze im Detail.

Hashtabelle mit verketteter Liste

Der erste Ansatz orientiert sich an der Arbeit in [23] und verwendet eine Hashtabelle, deren Buckets über eine verkettete Liste verbunden sind (`LL-HT`). Ein Bucketeintrag speichert

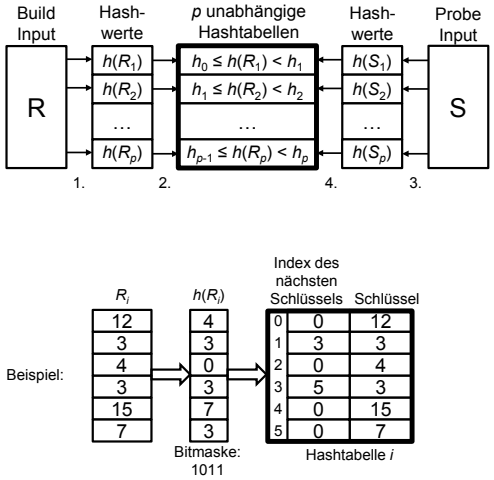
den Schlüssel aus der Eingangsrelation und einen Zeiger (Index) auf den nächsten Eintrag. Dieser Index dient als Referenz, um die verkettete Liste aufzubauen. Weiterhin speichert jeder Prozessor Informationen über den ersten und letzten Bucketeintrag sowie die aktuelle Gesamtanzahl der Einträge im lokalen Speicher ab.

Im Folgenden wird der Hash-Join für die Ausführung auf p Prozessoren (DBAs) beschrieben. Der Algorithmus baut auf dem Standardverfahren des Hash-Joins auf, das in Kapitel 2.2.3 erläutert wurde. Die Prozessoren laden die Schlüssel der Eingangsrelationen R und S mittels Double-Buffering, damit Datentransfer und Hashing teilweise gleichzeitig stattfinden kann. Jeder Prozessor i ($i = 1, ..., p$) arbeitet die folgenden Schritte ab, die auch in Abbildung 5.6a verdeutlicht sind:

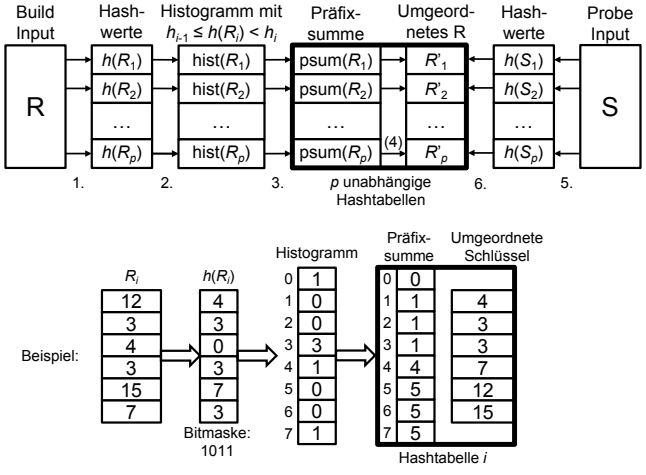
1. Hashing von R: Iteriere über R und berechne die Hashwerte $h(R)$. Das Laden und Verarbeiten der Daten vom globalen Speicher wird dabei abwechselnd auf zwei Hälften im lokalen Speicher durchgeführt.
2. Füge R in Hashtabelle ein: Überprüfe jeden Schlüssel in R, ob dieser in den für Prozessor i zugewiesenen Bereich von Hashwerten $h_{i-1} \leq h(R_i) < h_i$ passt. R_i und $h(R_i)$ bezeichnen dabei den für Prozessor i zugehörigen Schlüssel bzw. Hashwert. Füge nun den Schlüssel R_i in die Hashtabelle i ein, die zu Prozessor i gehört. Damit entstehen p unabhängige Hashtabellen. Ein Schlüssel wird dabei immer als ein neuer Bucketeintrag aufgenommen. Ist bereits ein Schlüssel im Bucket vorhanden, addiere den aktuellen Index auf den Index des zuvor eingefügten Schlüssels.
3. Hashing von S: Iteriere über S und berechne die Hashwerte $h(S)$ auf gleiche Weise wie mit R im 1. Schritt.
4. Join: Suche mit den Hashwerten $h(S_i)$ die korrespondierenden Schlüssel R_i in der Hashtabelle, vergleiche diesen mit dem Schlüssel S_i und speichere übereinstimmende Schlüssel ab. Schreibe dabei fortlaufend die Ergebnislisten in den globalen Speicher zurück.

Die beschriebene Partitionierung ähnelt dem Radix-Join aus [105], wobei die Radix-Bits hier nun durch die Hashfunktion ausgewählt werden. Damit ist sicher gestellt, dass keine Datenabhängigkeiten zwischen den Partitionen auftreten. Diese Lösung skaliert auch mit der Tupelanzahl, da die Relation R nur einmal durchlaufen werden muss, um die Hashtabelle aufzubauen. Außerdem speichert die Hashtabelle alle Einträge mit aufeinanderfolgenden Speicheradressen. Mit Ausnahme von Kollisionen, ist für das Beschreiben eines Buckets nur ein Speicherzugriff notwendig, der ein deterministischen Zeitverhalten aufweist.

Abbildung 5.6a veranschaulicht auch anhand eines Beispiels das Erzeugen der Hashtabelle der Partition R_i des Prozessors i. Die Hashfunktion basiert auf einer Bitselektion des Schlüssels (siehe Kapitel 2.2.2) und berechnet die Hashwerte von jedem Tupel. Der Bereich für gültige Hashwerte liegt zwischen h_{i-1} und $h_i = 16$, ansonsten würde der jeweilige Schlüssel nicht in die Hashtabelle eingefügt werden. Die gewählte Hashmaske führt zu einer Kollision in der Hashtabelle, da die Schlüssel 3 und 7 auf Grund identischer Hashwerte in den gleichen Bucket abgebildet werden. Der Bucket mit dem Schlüssel 3 an Position 1 ist deshalb mit dem nächsten Bucketeintrag bei Index 3 verkettet. Der Index bei Position 3 zeigt nun wieder auf Index 5, da der Schlüssel 7 ebenfalls zu diesem Bucket gehört.



(a) Hashtabelle mit verketteter Liste (`LL-HT`). Das Beispiel darunter zeigt die Schritte Hashing (1.) und Einfügen (2.) des Prozessors i.



(b) Hashtabelle mit Histogramm (`Hist-HT`). Das Beispiel darunter zeigt alle Schritte vom Hashing (1.) bis zum Einfügen (4.) des Prozessors i.

Abbildung 5.6: Hash-Join ausgeführt mit p Prozessoren

Hashtabelle mit Histogramm

Der zweite Ansatz (`Hist-HT`) erzeugt eine Hashtabelle anhand eines Histogramms und einer Präfixsumme (adaptiert von [17] und [92]), die eine nun umgeordnete Relation R enthält. Im Gegensatz zum `LL-HT` Hash-Join, müssen hier die Kardinalitäten der Eingangsrelationen bekannt sein, da für den Aufbau der Hashtabelle zweimal über R iteriert werden muss. Prozessor i erhält mittels Double-Buffering alle Schlüssel aus der Relation R, die sich im globalen Speicher befindet und führt, wie in Abbildung 5.6b veranschaulicht, die folgenden Schritte aus:

1. Hashing von R: Dieser Schritt ist identisch zum 1. Schritt des `LL-HT` Hash-Joins.
2. Bestimme Histogramm von R: Wähle anhand der berechneten Hashwerte $h(R)$ alle Schlüssel R_i aus, die in den für Prozessor i zugewiesenen Bereich von Hashwerten $h_{i-1} \leq h(R_i) < h_i$ passt, um das Histogramm i zu erhalten.
3. Berechne Präfixsumme aus dem Histogramm: Iteriere über das Histogramm i und bestimme die Präfixsummen für alle R_i.
4. Füge R in Hashtabelle ein: Lade R erneut und berechne die Hashwerte wie im 1. Schritt. Füge die Schlüssel R_i in die Hashtabelle i ein, deren Hashwerte im zugewiesenen Bereich liegen. Die Position in der Hashtabelle ist dabei durch die für diesen Schlüssel berechnete Präfixsumme bestimmt. Alle Schlüssel des gleichen Buckets sind nun nacheinander in der Hashtabelle abgespeichert. Es ergeben sich wieder p unabhängige Hashtabellen für die p Prozessoren.
5. Hashing von S: Iteriere über S und berechne die Hashwerte $h(S)$ auf gleiche Weise wie mit R im 1. Schritt.
6. Join: Identisch zum 4. Schritt des `LL-HT` Hash-Joins. Beim Suchen der jeweiligen Schlüssel in der Hashtabelle werden jetzt nur die Präfixsummen benötigt.

Aus dem beschriebenen Algorithmus lässt sich erkennen, dass Laden und Hashing von R zweimal ausgeführt wird. Da der DBA aber das Hashing durch die Befehlssatzerweiterung unterstützt, kann angenommen werden, dass dieser Schritt immer noch den geringsten Einfluss auf die Ausführungszeit des Hash-Joins haben wird. Allerdings ist auch zu untersuchen, inwieweit der erhöhte Datenverkehr die Laufzeit beeinflusst. Des Weiteren legt die Hashtabelle mit Histogramm alle Schlüssel eines Buckets nacheinander im Speicher ab. Damit kann in den meisten Fällen mit nur einem DMA-Zugriff, der gesamte Bucket vom globalen Speicher geladen werden. Im Vergleich zum `LL-HT` Hash-Join ist deshalb für die Hashtabelle mit Histogramm gerade für eine hohe Anzahl von Kollisionen eine geringere Ausführungszeit für die Join-Phase zu erwarten.

Abbildung 5.6b zeigt ein Beispiel für das Abbilden der Schlüssel R_i des Prozessors i in die Hashtabelle. Es wird angenommen, dass alle Hashwerte innerhalb des Gültigkeitsbereichs liegen. Nach dem Hashing wird das Histogramm aus den Hashwerten bestimmt: Hashwert 4 ist einmal vorhanden, Hashwert 3 gibt es dreimal, Hashwert 0 ist ebenfalls einmal vorhanden, usw. Die Präfixsumme bestimmt nun die Positionen der Schlüssel in der Hashtabelle. Beispielsweise wird Schlüssel 12 auf Position 4 der Hashtabelle abgebildet, da die zugehörige Präfixsumme den Index 4 angibt.

5.3.4 Weitere Algorithmen

Wie bereits in Kapitel 4.3.2 erwähnt wurde, unterstützt der in den Tomahawk-Plattformen integrierte DBA ASIP noch weitere Datenbankoperatoren, die für eine effiziente Anfrageverarbeitung von Bedeutung sind: Sort-Merge-Aggregation, Sampling und die Kompression und Verarbeitung von Bitmap-Indizes. Die Partitionierung der Daten ist dabei trivial, da z. B. beim Sampling keine Datenabhängigkeiten zwischen den Elementen existieren. Die zu durchlaufende Relation der Sort-Merge-Aggregation kann ebenfalls beliebig auf mehrere Prozessoren aufgeteilt werden. Die Grenzen der Partitionen müssen nur soweit verschoben werden, dass sich angrenzende Tupel mit gleichen Schlüsseln nicht in unterschiedlichen Partitionen befinden. Auch für die Algorithmen auf Bitmap-Indizes lassen sich mit Hilfe einer einfachen Vorverarbeitung durch den Scheduler unabhängige Partitionen finden [169]. Die Lösungsansätze sind zu den drei hier beschriebenen parallelen Implementierungen ähnlich und werden deshalb in dieser Arbeit nicht weiter diskutiert.

5.4 Ergebnisse und Auswertung

Dieser Abschnitt präsentiert zunächst die Leistungsfähigkeit der drei ausgewählten Datenbankoperatoren für die beschriebenen parallelen Implementierungen. Danach folgen Ergebnisse zum Leistungs- und Energieverbrauch sowie ein Vergleich mit anderen Arbeiten. Alle Messungen wurden dabei wie in Abschnitt 5.3 beschrieben auf dem Tomahawk3 bzw. Tomahawk4 MPSoC mit den jeweils vier integrierten DBAs durchgeführt. Dies erlaubt den Vergleich der verschiedenen Implementierungsansätze und der beiden Forschungschips. Die Zeitmessung erfolgt über Timer, die in den Prozessoren enthalten sind. Analog-Digital-Wandler messen Versorgungsspannung und Stromaufnahme der Versorgungsleitungen [61, 65]. Aus den Ergebnissen wird dann die Verlustleistung berechnet. Sofern nicht anders angegeben, laufen die DBAs, der Scheduler und das NoC bei einer

Taktfrequenz von 500 MHz. Die gewählten Gesamtkardinalitäten der Mengen und Relationen übersteigen dabei die Größe der lokalen Speicher der PMs um mindestens Faktor 10. Damit ist eine faire Gegenüberstellung des Zeitaufwandes für den Datentransfer und der Verarbeitungszeit möglich, da Randeffekte wie der Initialisierungs-Overhead in der Software oder die Kommunikation zwischen Scheduler und DBAs vernachlässigbar sind. Des Weiteren ist der Scheduler für die Initialisierung der Daten im globalen Speicher sowie für das Starten der DBAs verantwortlich.

5.4.1 Leistungsfähigkeit der Datenbankoperatoren

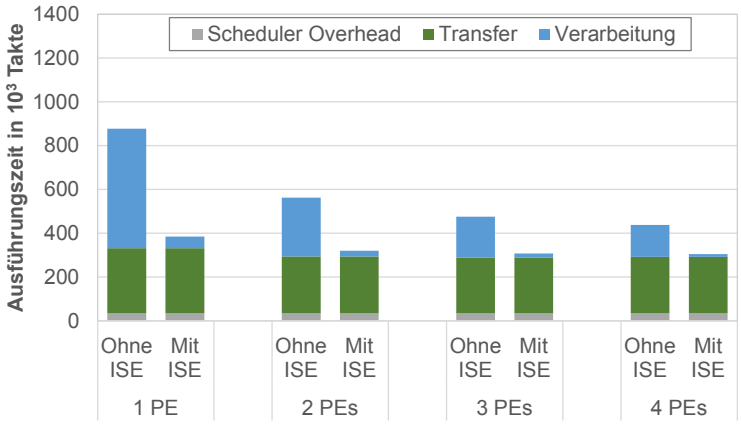
Sorted-Set Intersektion

Für die Evaluierung der Sorted-Set Intersektion werden zwei Datensätze ausgewählt. Datensatz (1) stellt den am häufigsten auftretenden Fall dar, bei dem die zwei Mengen A und B eine Selektivität von 50 % haben und die Adjazenzmatrix nur Einsen in der Hauptdiagonalen aufweist. Der Datensatz (2) deckt den Sonderfall ab, wenn nur ein Datenblock der Menge A mit allen Datenblöcken aus Menge B Ergebniselemente der Intersektion liefert. Die Adjazenzmatrix hat dann nur Einsen in einer Zeile. Um diesen Datensatz zu erzeugen, muss die Selektivität gering sein und wurde deshalb mit 12,5 % angenommen. Beide Datensätze enthalten zufällig erzeugte und gleichmäßig verteilte 32-Bit Werte in den Mengen mit einer Gesamtkardinalität von $|A| + |B| = 100\,000$ Elementen.

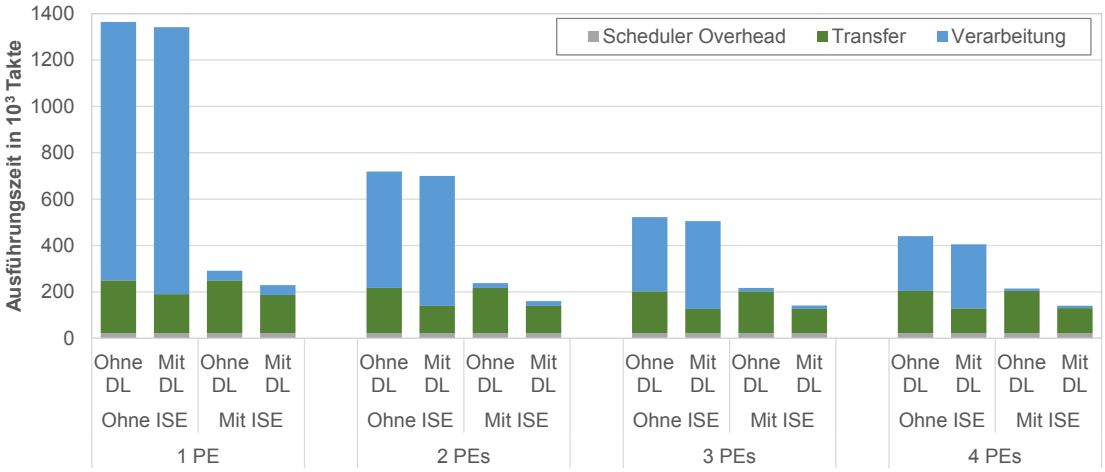
Abbildung 5.7 zeigt die Gesamtausführungszeit für beide Datensätze, wenn die Größe eines Datenblocks der Größe des lokalen Datenspeichers entspricht (siehe Implementierungsansatz (1) aus Abschnitt 5.3.1, Ausführung mit dem Tomahawk3). Ein PE repräsentiert dabei den Xtensa LX5 RISC Prozessor (DBA ASIP), der den Algorithmus entweder ohne Instruktionssatzerweiterung (ISE) oder mit ausführt. Mit dem Datensatz (1) ermöglicht die Nutzung der ISE eine bis zu 2,3× Reduzierung der Ausführungszeit. Im Vergleich zu einem PE ohne ISE, kann die Gesamtausführungszeit mit vier PEs um Faktor 2,0 verringert werden. Insgesamt ist dann bei Hinzunahme der ISE ein Speedup von 3,8 möglich. Der sequentielle Anteil des Algorithmus, d. h. der Scheduler-Overhead und vor allem die Transferzeit, limitiert den Speedup und damit den Vorteil der ISE sowie die Ausführung mit mehreren PEs. Zum Beispiel benötigt der Datentransfer für vier PEs mit ISE etwa 82 % und dominiert damit deutlich die Gesamtausführungszeit. Der Overhead des Schedulers nimmt hier bis zu 11 % ein und erhöht sich linear mit zunehmender Kardinalität, jedoch nicht mit der Anzahl der PEs.

Der Datensatz (2) erlaubt die Nutzung von Datenlokalität, da ein PE für die Dauer des gesamten Algorithmus nur einen Datenblock der Menge A im lokalen Speicher halten muss. Bei vier PEs kann damit die Transferzeit um 30 % reduziert werden. Die im Vergleich zu Datensatz (1) höhere Verarbeitungszeit der PEs wird durch die niedrigere Selektivität der Mengen in Datensatz (2) verursacht (siehe auch Kapitel 4.3.2).

Teilt man nun die Eingangsmengen nicht in Datenblöcke auf, die der Größe der lokalen Speicher entsprechen, sondern richtet sich nach der Prozessoranzahl, erhält man die in Abbildung 5.8a gezeigten Ergebnisse für den Implementierungsansatz (2), d. h. der Ausführung mit dem Tomahawk4 und den gleichmäßig verteilten Daten in Datensatz (1). Damit steigt zwar der Overhead des Schedulers von einer PE auf vier PEs um den Faktor 6,5 an, ändert sich aber nicht bei unterschiedlichen Kardinalitäten. Die Transferzeit



(a) Datensatz (1)

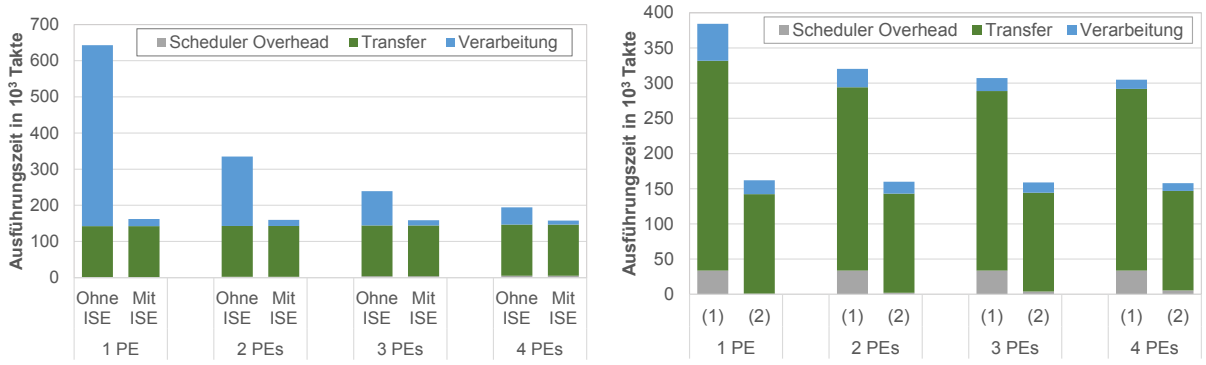


(b) Datensatz (2)

Abbildung 5.7: Sorted-Set Intersektion: Gesamtausführungszeit für Ansatz (1) ausgeführt auf dem Tomahawk3, $s_{Block} = s_{Mem} = 2500$ Elemente, DL: Datenlokalität

dominiert auch hier die Gesamtzeit und damit den Speedup. Der Datensatz (2) zeigt annähernd gleiche Ergebnisse wie in Abbildung 5.8a, da durch das Double-Buffering die Datenlokalität nicht ausgenutzt werden kann.

Ein Vergleich der beiden Ansätze ist in Abbildung 5.8b zu sehen. Die Vorteile liegen dabei bei Ansatz (2). Die Gesamtausführungszeit halbiert sich auf Grund der ebenfalls halbierten Datentransferzeit. Des Weiteren ist der Overhead des Schedulers um mindestens Faktor 6,2 geringer und nimmt damit weniger als 5 % der Gesamtausführungszeit ein. Das liegt daran, dass der Scheduler im konkreten Beispiel eine 20×20 und 4×4 Adjazenzmatrix bei Ansatz (1) bzw. (2) verarbeiten muss. Im Vergleich zu einer genauen Einteilung von Datenblöcken gemäß der Größen des lokalen Speichers wie sie nur im Tomahawk3 möglich ist, reduziert das abwechselnde Laden und Speichern in einer Softwarepipeline mit Hilfe des iDMA aus dem Tomahawk4 die Gesamtlaufzeit um bis zu Faktor 1,9. Dennoch ist bei beiden Ansätzen auf Grund der verhältnismäßig hohen Transferzeit keine Skalierbarkeit mit der Anzahl der PEs gegeben.



(a) Ansatz (2), $s_{Block} = \frac{50\,000\,Elemente}{p}$

(b) Vergleich von Ansatz (1) und (2) für die Ausführung mit ISE

Abbildung 5.8: Sorted-Set Intersektion: Gesamtausführungszeit für Datensatz (1), jeweils ausgeführt auf dem Tomahawk3 und Tomahawk4

Der maximale Gesamtdurchsatz für vier DBAs beträgt 316,9 Mio. Elem./s und ist damit etwa 4,6× geringer als der lokale Durchsatz eines DBAs. Die reduzierte Leistungsfähigkeit wird durch den Datentransfer über das NoC ($B_{NoC} = 1000$ Mio. Elem./s) und vor allem durch die Bandbreite des DRAM ($B_{DRAM} = 400$ Mio. Elem./s) verursacht.

Merge-Sort

Zur Untersuchung der Leistungsfähigkeit des parallelen Merge-Sort werden die in Abschnitt 5.3.2 eingeführten Anwendungsfälle entsprechend der Ausführung auf dem Tomahawk3 und Tomahawk4 ausgewertet. Zusätzlich wird das Merge-Sort mit dem auf dem Tomahawk4 integrierten Applikationsprozessor `APP_570T` ausgeführt. Im Gegensatz zu den DBAs, umfasst der `APP_570T` jeweils einen 16-kB Cache für Daten und Instruktionen. Da beim Sortieren mehrfach auf gleiche Elemente zugegriffen werden muss, ist hier eine Speicherarchitektur mit Cache von Vorteil. Zusätzlich ist der globale Speicher direkt in den Adressbereich des Prozessors abgebildet. Damit entfällt das Konfigurieren eines DMA-Controllers beim Laden und Speichern der Daten. Der `APP_570T` führt den in Kapitel 2.2.3 beschriebenen skalaren Code des Merge-Sort aus, ohne dabei das bitonische Sortiernetzwerk zu nutzen.

Abbildung 5.9 stellt die Gesamtausführungszeit über der Elementanzahl für die unterschiedlichen Anwendungsfälle dar. Dabei wurden jeweils Fall (1) und (2) auf dem Tomahawk3 und Fall (3) auf dem Tomahawk4 ausgeführt. Fall (1) benötigt die geringste Taktanzahl, da hier das einfache Merge-Sort ohne Lastverteilung angewendet wird. Die Elementanzahl ist allerdings durch die Größe des lokalen Speichers limitiert. Für die konkrete Speichergröße der PEs gilt: $s_{Mem} = N = 7000$ Elemente. Für Fall (2) wird das bitonische Sortiernetzwerk verwendet sowie teilweise die Datenlokalität ausgenutzt, sodass bei der Nutzung von vier DBAs die Anzahl der Elemente auf $N \leq 4 \cdot 7000 = 28\,000$ Elemente beschränkt ist. Die Ausführungszeit steigt dann um Faktor 1,3, obwohl das bitonische Sortiernetzwerk eine vollständige Parallelität ermöglicht. Jedoch sind insgesamt mehr Tasks

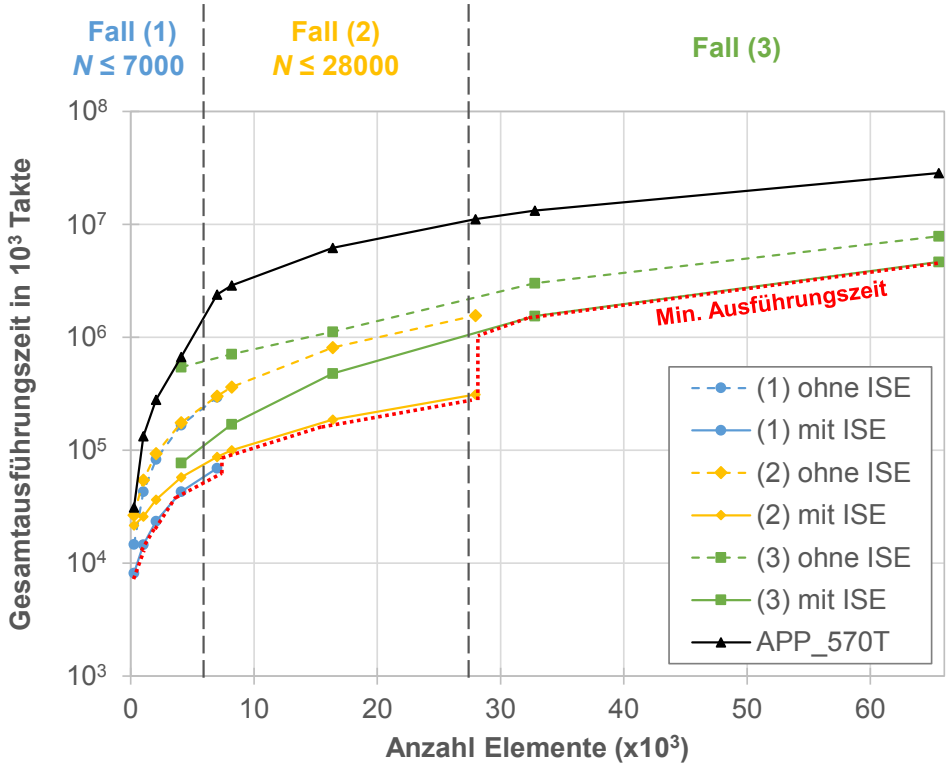


Abbildung 5.9: Merge-Sort: Gesamtausführungszeit mit vier DBAs. Die rote gepunktete Linie markiert die untere Grenze der Ausführungszeit.

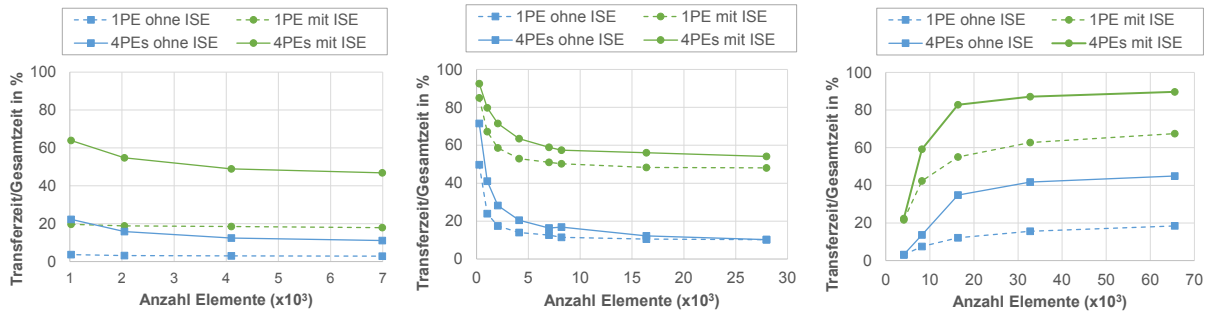
abzuarbeiten (Fall (1): $\mathcal{O}(n_{Block})$, Fall (2): $\mathcal{O}(n_{Block} \log^2 n_{Block})$, siehe Abschnitt 5.3.2) und damit auch mehr Daten zu transferieren.

Berücksichtigt man keine Datenlokalität mehr, gibt es in Fall (3) auch keine Einschränkungen für die Elementanzahl. Im Vergleich zu Fall (2) nimmt damit die Leistungsfähigkeit jedoch um Faktor 5,0 ab, da in Fall (3) die Elemente für das Sortieren jeder Teilstufe immer vom globalen Speicher geladen werden müssen. Der zeitliche Anteil dieser zusätzlichen Datentransfers bestimmt wesentlich das Laufzeitverhalten. Trotz der Vorteile des iDMA (zeitliches Überlappen von Transfer und Datenverarbeitung) und der höheren Bisektionsbandbreite im Tomahawk4 MPSoC weist dieser Sortieralgorithmus keine signifikanten Unterschiede zwischen der Ausführung auf dem Tomahawk3 und Tomahawk4 auf.

Der `APP_570T` zeigt für Fall (3) nur eine akzeptable Leistungsfähigkeit, bis die Anzahl der zu sortierenden Elemente die Größe des Datencaches von 4096 Elementen überschreitet. Im Vergleich zu vier PEs mit ISE, benötigt der `APP_570T` dann etwa 6,1× mehr Takte, um 65 536 Elemente[3] zu sortieren.

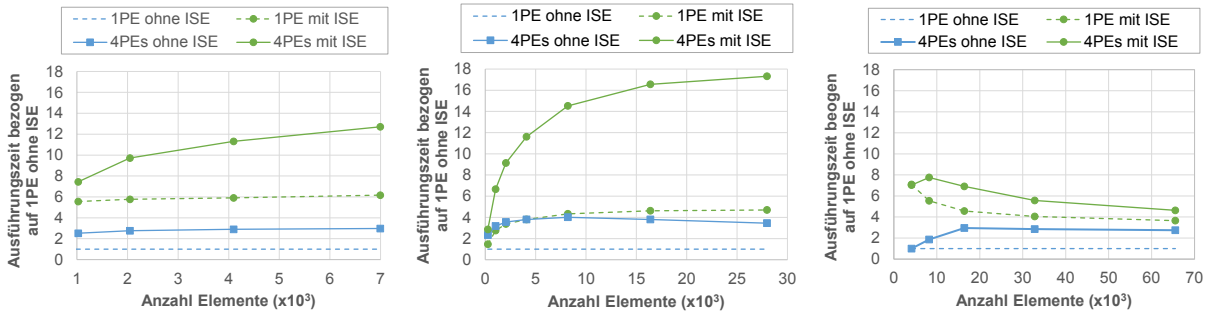
Der Grund für die Erhöhung der Ausführungszeit beim Wechseln der Anwendungsfälle ist die Zunahme der Datentransferzeit gegenüber der Gesamtzeit (Abbildung 5.10). Zum Beispiel erhöht sich der Anteil der Transferzeit bei 7000 Elementen für die drei Anwen-

[3]Die Wahl der Elementanzahl ist nicht auf Vielfache einer Zweierpotenz beschränkt, jedoch ist damit die gleichmäßige Auslastung der DBAs und des Sortiernetzwerkes gewährleistet.



(a) Fall (1) (b) Fall (2) (c) Fall (3)

Abbildung 5.10: Merge-Sort: Verhältnis der Datentransferzeit zu Gesamtausführungszeit für drei Anwendungsfälle



(a) Fall (1) (b) Fall (2) (c) Fall (3)

Abbildung 5.11: Merge-Sort: Speedup für drei Anwendungsfälle

dungsfälle bei vier PEs mit ISE von 47 % auf 59 %. Zudem erhöht sich jeweils der Anteil der Transferzeit, wenn sich die Anzahl der PEs erhöht. Die PEs müssen sich dann die Speicherbandbreite des DRAM teilen. Eine Erhöhung der relativen Transferzeit ist auch zu beobachten, wenn die PEs die Befehlssatzerweiterung nutzen. Das ist lediglich auf eine Verringerung der Verarbeitungszeit der DBAs und damit auch auf eine Verringerung der Gesamtzeit zurückzuführen.

Auffallend ist, dass sich die relative Transferzeit nur für Fall (3) mit steigender Elementanzahl erhöht. Für vier PEs mit ISE benötigt die Transferzeit ca. 90 % der Gesamtausführungszeit. Der Zugriff auf den DRAM findet hier verhältnismäßig häufiger statt. Für diesen Algorithmus hat die Limitierung der Speicherbandbreite damit den höchsten Einfluss und verhindert die Skalierung des Systems mit der Elementanzahl.

Aus den Erkenntnissen zum Verhältnis aus Datentransferzeit zur Gesamtausführungszeit lassen sich auch die Beschleunigungsfaktoren in Abbildung 5.11 erklären. Für die Anwendungsfälle (1) und (2) ist der Anteil der Transferzeit nicht dominant und es wird durch das Nutzen von vier statt einem PE jeweils ein Speedup von 3,0 bzw. 3,9 erreicht. Wird dann die Befehlssatzerweiterung hinzugenommen, sind für 7000 Elemente Speedups von 12,7 bzw. 13,5 möglich, die sich mit steigender Elementanzahl weiter erhöhen. Für den speicher-limitierten Fall (3) verringert sich mit zunehmender Anzahl der Elemente der Speedup gegenüber der Ausführung mit einem PE ohne ISE auf ungefähr

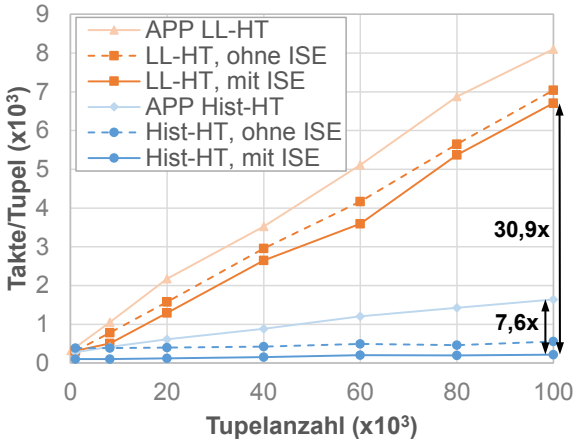
Faktor 4. Besonders ist dies bei der Nutzung von vier PEs mit ISE zu erkennen, da hier der Anteil der Datentransferzeit mit fast 90 % am höchsten ist.

Je nach gegebener Elementanzahl kann der passende Anwendungsfall ausgewählt werden. Sind mehr Prozessoren verfügbar, kann der Fall (2) auch für größere Mengen verwendet werden, da sich die gesamte zu sortierende Menge auf mehr lokale Speicher verteilt. Das reduziert den Flaschenhals zum globalen Speicher. Weitere Untersuchungen zum Einfluss der Speicherbandbreite bei Erhöhung der Prozessoranzahl folgen in Kapitel 6.2.2.

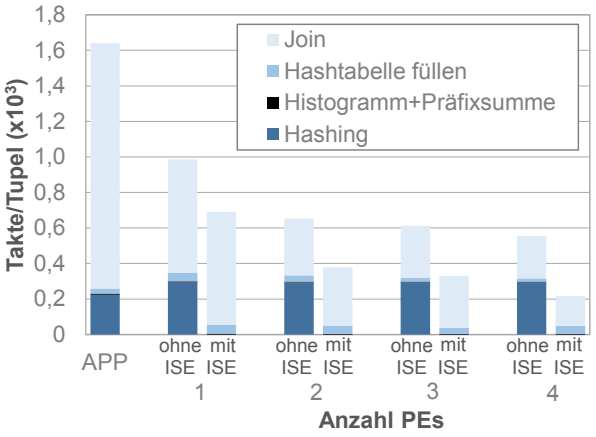
Hash-Join

Für die Evaluierung des Hash-Joins werden die in Abschnitt 5.3.3 beschriebenen zwei Varianten der Hashtabelle mit verketteter Liste `LL-HT` und Histogramm `Hist-HT` genutzt. Beide Varianten nutzen das Double-Buffering mit Hilfe des iDMA beim Laden der Daten und werden deshalb beide auf dem Tomahawk4 MPSoC ausgeführt. Die Vorteile gegenüber dem Laden ohne Double-Buffering wurden bereits zuvor bei den Sorted-Set Algorithmen gezeigt. Die Eingangsrelationen R und S mit $|R| = |S|$ enthalten als Tupel die 64-Bit Schlüssel-Payload-Paare. Dabei sind die 32-Bit Schlüssel zufällig erzeugt und folgen einer gleichmäßigen Verteilung. Die maximale Anzahl der Ergebnistupel beträgt $|R| \cdot |S|$ und wird auf Grund der begrenzten Speicherkapazität auf $\frac{1}{4}|R|$ begrenzt, indem die Schlüssel beider Relationen mit einer Selektivität von 25 % erzeugt werden. Die Hashfunktion basiert auf der in Kapitel 2.2.2 beschriebenen Bitselektion. In der zugehörigen Hashmaske sind dabei die zehn niederwertigsten Bits auf Eins gesetzt. Die Hashtabelle umfasst dann $2^{10} = 1024$ Buckets. Die Wahl dieser Hashmaske erlaubt die Minimierung der Anzahl von Kollisionen in der Hashtabelle, da auch die generierten Werte in den Elementen zumeist die Bits im Bereich des niederwertigsten Bits belegen. Theoretisch hat jede Hashfunktion bzw. Hashmaske für einen gegebenen Datensatz Vor- und Nachteile. Das Ziel dieser Untersuchungen ist aber, allgemein den Hash-Join im Hinblick auf die Eigenschaften der Tomahawk Plattform zu analysieren. Die Auswertung hinsichtlich unterschiedlicher Hashfunktionen wird deshalb für zukünftige Arbeiten offen gelassen.

In einem ersten Schritt werden die Algorithmen auf allen vier PEs ausgeführt und dabei die Kardinalität der Eingangsrelationen variiert. Abbildung 5.12a vergleicht dazu die Anzahl der benötigten Takte pro Eingangstupel, die sich aus der Ausführungszeit in Takten dividiert mit der Gesamttupelanzahl $|R| + |S|$ berechnet. Die Graphen zeigen hier einen linearen Anstieg bei zunehmender Kardinalität, wobei die direkte Ausführungszeit dann quadratisch mit der Tupelanzahl steigt. Dieses Verhalten ist auf die auftretenden Kollisionen in der Hashtabelle zurückzuführen und wird in Kapitel 6.3.3 genauer untersucht. In Abbildung 5.12a erhöht sich die Ausführungszeit des `LL-HT` Hash-Joins schneller als die des `Hist-HT`. Bei 100 000 Tupeln pro Relation benötigt der `Hist-HT` Hash-Join etwa 217 Takte/Tupel und ist damit rund 31× schneller als der `LL-HT` Hash-Join. Die Leistungsfähigkeit erhöht sich um bis zu Faktor 2,6×, wenn die Befehlssatzerweiterung für das Hashing genutzt wird. Die ISE einer PE kann den Durchsatz des Hashings jedoch um mehr als drei Größenordnungen beschleunigen. Dieser Zusammenhang kann erst in den anschließenden Paragraphen mit den zugehörigen Diagrammen begründet werden. Zusätzlich wird der skalare Algorithmus ohne die Partitionierung der Eingangsrelationen mit dem Applikationsprozessor `APP_570T` auf dem Tomahawk4 ausgeführt. Diese



(a) Vergleich der Implementierungen mit verschiedenen Hashtabellen, jeweils ausgeführt mit `APP_570T` (APP) und 4 PEs



(b) Hash-Join basierend auf Histogramm (`Hist-HT`), $|R| = |S| = 100\,000$ Tupel

Abbildung 5.12: Hash-Join: Ausführungszeit in Takte pro Tupel ($\frac{t_{Takt}}{|R|+|S|}$)

Graphen zeigen auch einen linearen Anstieg mit zunehmender Kardinalität. Allerdings ist der `APP_570T` bis zu 1,2× und 7,6× langsamer als der mit vier DBAs ausgeführte `LL-HT` bzw. `Hist-HT` Hash-Join.

Um den Unterschied der beiden Ansätze zu verstehen, wird der `Hist-HT` Hash-Join auf mehreren Prozessoren ausgeführt und dabei die Takte pro Tupel für die einzelnen Schritte des Algorithmus erfasst (Abbildung 5.12b). Die Anzahl der Takte für den Hashing-Schritt enthalten das Hashing aller Schlüssel beider Eingangsrelationen. Diese Zahl bleibt auch für eine unterschiedliche Anzahl von PEs konstant, da jeder Prozessor immer alle Schlüssel in R und S verarbeitet. Damit gehört das Bestimmen des Histogramms und der Präfixsumme sowie das Befüllen der Hashtabelle zur Build-Phase und der Join-Schritt zur Probe-Phase des Hash-Joins. Während in der Build-Phase die PEs nur Daten in den globalen Speicher schreiben, müssen in der Probe-Phase die Tupel aus der Hashtabelle gelesen werden. Auf Grund der höheren Latenz bei DRAM-Lesezugriffen benötigt der Join-Schritt immer mindestens 70 % der Gesamtausführungszeit. Das gleiche Verhalten ist für den Applikationsprozessor in Abbildung 5.12b zu sehen.

Beim `Hist-HT` Hash-Join befinden sich alle Einträge eines Buckets nacheinander im Speicher. Beim Suchen eines Schlüssels in der Hashtabelle ist deshalb idealerweise nur ein einziger Zugriff auf den globalen Speicher notwendig, da mit einem DMA-Transfer mehrere Daten an aufeinanderfolgenden Speicheradressen gelesen werden können. Dies gilt auch bei einer hohen Anzahl von Kollisionen in der Hashtabelle. Im Gegensatz dazu, sind die Bucketeinträge in der Hashtabelle mit verketteter Liste (`LL-HT`) unregelmäßig im Speicher verteilt. Angenommen, es treten n Kollisionen in der Hashtabelle auf, dann müssen Daten an n verschiedenen Adressen im globalen Speicher geladen und damit der DMA-Controller n-mal konfiguriert werden. Das heißt, die Ausführungszeit für den Join-Schritt steigt auch mit zunehmender Anzahl an Kollisionen. Die für den `LL-HT` Hash-Join korrespondierende Abbildung 5.12b würde zeigen, dass immer mehr als 97 % der Gesamtausführungszeit für den Join-Schritt notwendig sind. Des Weiteren lässt sich schlussfol-

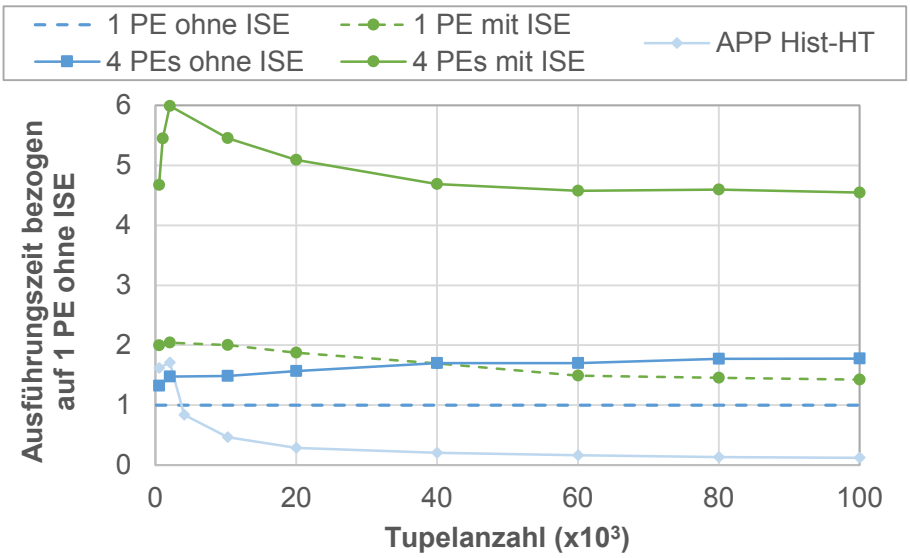
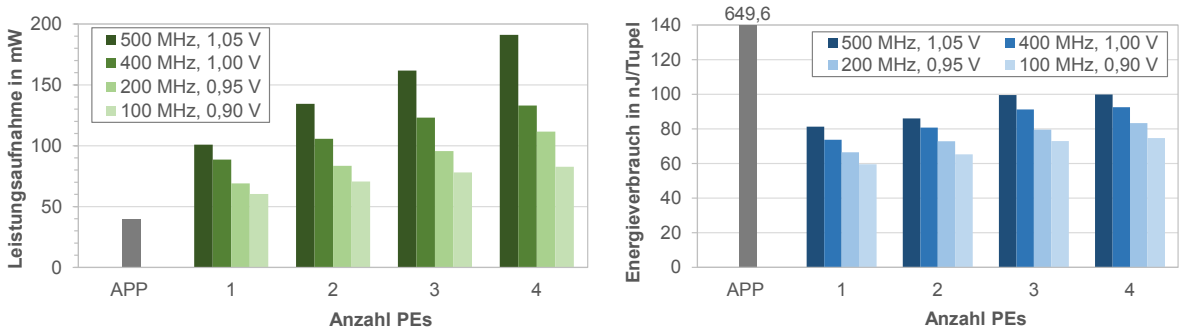


Abbildung 5.13: Hash-Join: Speedup bezogen auf einen Prozessor ohne ISE für die Implementierung der Hashtabelle mit Histogramm (`Hist-HT`)

gern, dass obwohl der `Hist-HT` Hash-Join die Relation R zweimal durchläuft, dieser Ansatz eine bessere Leistungsfähigkeit aufweist, da die Probe-Phase des `LL-HT` Hash-Joins die gesamte Ausführungszeit dominiert. Ein Einfluss des vergleichsweise erhöhten Datenverkehrs auf die Gesamtlaufzeit kann nicht festgestellt werden.

Es folgen nun weitere Untersuchungen zum Einfluss der Befehlssatzerweiterung und der Parallelität auf die Leistungsfähigkeit des Hash-Joins. Abbildung 5.13 zeigt dazu den Speedup für den `Hist-HJ` Hash-Join für unterschiedliche Konfigurationen. Jeder Graph ist auf die Ausführung des Algorithmus mit einer PE ohne ISE normiert. Die lokalen Speicher der PEs enthalten zwei Arrays mit 4096 Elementen, wobei diese abwechselnd zum Laden und Verarbeiten genutzt werden. Es zeigt sich deshalb ein Anstieg der Speedups bis 4096 Elemente. Danach fallen die Speedups wieder ab bzw. bleiben nahezu gleich, da nun die Zeit der Probe-Phase den Algorithmus dominiert und damit auch der Vorteil der Befehlssatzerweiterung mit zunehmender Kardinalität verloren geht. Jedoch zeigt sich, dass der durch vier parallele Prozessoren entstehende Speedup bei der Nutzung der ISE und dem damit beschleunigten Hashing etwa um Faktor 1,8 höher ist. Der zeitliche Anteil des Hashings ist nicht parallelisierbar. Eine Reduzierung des sequentiellen Anteils wirkt sich deshalb positiv auf die Skalierbarkeit aus.

Solange der `APP_570T` von dem Datencache profitiert, arbeitet dieser bis zu 1,8× schneller als ein PE ohne ISE. Ab 2000 8-Byte Tupeln ist die Kapazität des 16-kB Cache erreicht und die zusätzliche Latenz des DRAM führt zu einer höheren Ausführungszeit. Die Speedups für den `LL-HT` Hash-Join werden hier nicht explizit gezeigt, da sie annähernd den Ergebnissen der Implementierung für den `Hist-HT` Hash-Join entsprechen.



(a) Leistungsaufnahme

(b) Energieverbrauch (Leistungsaufnahme von Scheduler und NoC nicht eingerechnet)

Abbildung 5.14: Leistungsmessung des Hash-Joins mit Hashtabelle und Histogramm (`HJ-Hist`) für verschiedene Spannungs- und Frequenzlevel auf dem Tomahawk4. Die PEs nutzen die ISE für das Hashing.

5.4.2 Leistungsaufnahme

Zur Messung der mittleren Verlustleistung werden die Datenbankoperatoren auf dem Tomahawk in einer Endlosschleife ausgeführt. Die Ergebnisse zeigen dabei keine signifikanten Unterschiede zwischen den verschiedenen Algorithmen, da eine mittlere Leistungsaufnahme gemessen wird, die z. B. in dem PM nicht nur den Prozessor sondern die Stromaufnahme aller Logikblöcke einschließt (DMAC, ADPLL, PMC). Es ist deshalb ausreichend, die Leistungsaufnahme von nur einem bestimmten Datenbankoperator auszuwerten, der dann die Resultate mehrerer Algorithmen repräsentiert. Des Weiteren konnte kein signifikanter Unterschied bei der Messung der Leistungsaufnahme zwischen den beiden Forschungschips Tomahawk3 und Tomahawk4 gemessen werden. Dieses Ergebnis ist zu erwarten, da die Chipfläche beider MPSoCs, die DBAs, die Versorgungsspannungen und auch die gewählten Taktfrequenzen identisch sind. Dieser Abschnitt fokussiert sich deshalb auf die Ergebnisse der Leistungsmessung für den Hash-Join, der eine auf einem Histogramm basierende Hashtabelle verwendet. Abbildung 5.14a zeigt die Gesamtverlustleistung für verschiedene Spannungs- und Frequenzlevel. Dabei werden nur Spannung und Frequenz der PMs geändert. Der Scheduler und das NoC laufen weiterhin bei 500 MHz und 1,05 V und nehmen damit immer etwa 51 mW auf. Die Algorithmen erreichen den höchsten Durchsatz bei der maximalen Taktfrequenz von 500 MHz. In diesem Fall liegt die Verlustleistung des Chips einschließlich vier PMs, Scheduler und NoC bei 192 mW.

Abbildung 5.14b stellt den Energieverbrauch für den Applikationsprozessor und die PEs dar. Zur Berechnung der Energie wurde nur die Leistungsaufnahme der PEs einbezogen, um die mögliche Skalierung für verschiedene Spannungs- und Frequenzlevel zu erkennen. Insgesamt steigt der Energieverbrauch um etwa 18 %, wenn die Anzahl der PEs erhöht wird. Der Grund ist, dass sich zwar die Leistungsaufnahme gleichmäßig mit der Prozessoranzahl erhöht, der Durchsatz dabei jedoch nicht perfekt skaliert. Der Energieverbrauch verringert sich aber, wenn Taktfrequenz und Versorgungsspannung reduziert werden, da sich der Durchsatz linear zur Frequenz und die Leistung quadratisch zur Spannung verhält. Der geringste Energiebedarf ergibt sich demnach mit 59,5 nJ/Tupel bei Nutzung von nur

Tabelle 5.2: Systemspezifikationen der Vergleichsprozessoren

	Tomahawk3/4	Intel i7-920	Intel Q9550	Intel Xeon E5-2680
Technologie in nm	28	45	45	32
Taktfrequenz in GHz	0,5	2,67	2,83	2,7
Chipfläche in mm^2	18	263	214	216
DRAM-Bandbreite in GBit/s	12,5	204,8	84,8	409,6

einem PE, das auf dem niedrigsten Frequenz- und Spannungslevel arbeitet. Im Vergleich zur Ausführung mit dem Applikationsprozessor kann die Energieeffizienz um bis zu Faktor 6,5 erhöht werden.

5.4.3 Vergleich mit anderen Architekturen

Es schließt sich nun ein Vergleich der Leistungsfähigkeit der Datenbankoperatoren Sorted-Set Intersektion, Merge-Sort und Hash-Join mit hoch-performanten x86-Prozessoren an. Ein direkter Vergleich mit den anderen in Abschnitt 5.1.2 vorgestellten Systemen, die mit zusätzlicher Hardware an die Verarbeitung der Datenbankoperatoren angepasst sind, ist nur in speziellen Einzelfällen möglich. Zumeist werden unterschiedliche Konfigurationen der zu verarbeitenden Datensätze gewählt, die einen unfairen Vergleich zur Folge hätten. Dieser Abschnitt fokussiert sich deshalb auf den Vergleich mit GP-Prozessoren, die die Basis moderner Datenbanksysteme bilden und damit den aktuellen Stand der Technik entsprechen. Tabelle 5.2 fasst die Spezifikationen der Vergleichssysteme zusammen und Tabelle 5.3 stellt die jeweiligen Messergebnisse gegenüber. Die Messungen mit dem Tomahawk sind dabei aus Abschnitt 5.4.1 und 5.4.2 für maximale Kardinalität und mit der Nutzung der ISE entnommen. Es wird zudem unterschieden, wo sich die Daten zu Beginn des Algorithmus befinden: im lokalen Speicher der PMs (lokal) oder im globalen Speicher des Tomahawk (global).

Für die Sorted-Set Intersektion erreicht ein PE für eine lokale Verarbeitung der Daten einen Durchsatz von 46,9 GBit/s. Müssen die Daten erst vom globalen in den lokalen Speicher transferiert werden, verringert sich der Durchsatz um Faktor 4,8 auf 9,9 GBit/s. Die Bandbreite des DRAM mit 12,5 GBit/s begrenzt hier den Durchsatz und ist mehr als 16× geringer als die des Intel i7-920 mit SSE-Instruktionen aus [130]. Eine Erhöhung der Speicherbandbreite auf dem Tomahawk würde deshalb auch eine Erhöhung des Durchsatzes ermöglichen. Des Weiteren nehmen vier PEs etwa 680× weniger Leistung als der Intel i7-920 auf. Damit ist eine Energieeinsparung von Faktor 192 möglich. Das ATE-Produkt bezieht die Fläche und die Leistungsaufnahme der Systeme sowie das Quadrat des Durchsatzes in den Vergleich ein. Der Tomahawk ist dann um drei Größenordnungen besser als die GP-CPU.

Chhugani u. a. [33] beschleunigen das Merge-Sort mit Hilfe von SSE-Instruktionen auf einem Intel Q9550 und erreichen mit vier Kernen einen Durchsatz von 6,1 GBit/s. Der Tomahawk sortiert mit vier PEs nur 0,22 GBit/s, dafür benötigt dieser aber weniger als 6 % der Energie des optimierten x86-Prozessors. Beim ATE-Produkt ist der Tomahawk etwa Faktor 8 besser als der Intel Q9550. Im Gegensatz zur Sorted-Set Intersektion nähert sich der Durchsatz beim Merge-Sort nicht an die Bandbreite des DRAM an, da hier die

Tabelle 5.3: Vergleich der parallelen Implementierungen der Datenbankoperatoren für den Tomahawk mit anderen Prozessorarchitekturen

		1 PE lokal	1 PE global	4 PEs global	Intel i7-920 [130] 1 Kern
Sorted-Set Intersektion	Durchsatz in GBit/s	46,9	9,9	10,1	35,6
	Leistung in W	0,05	0,1	0,19	(1)130
	Energie in nJ/Bit	0,001	0,010	0,019	3,7
	rel. zu Intel	3700×	370×	195×	1×
	ATE-Produkt in $mm^2\,W/(GBit/s)^2$	$4{,}1 \cdot 10^{-4}$	0,018	0,019	27,0

		1 PE lokal	1 PE global	4 PEs global	Intel Q9550 [33] 1 Kern	Intel Q9550 [33] 4 Kerne
Merge-Sort	Durchsatz in GBit/s	1,1	0,18	0,22	1,9	6,1
	Leistung in W	0,05	0,1	0,19	(2)95	(2)95
	Energie in nJ/Bit	0,045	0,568	0,848	49,5	15,6
	rel. zu Intel	347×	27×	18×	0,3×	1×
	ATE-Produkt in $mm^2\,W/(GBit/s)^2$	0,74	55,6	70,7	$5{,}6 \cdot 10^3$	546,4

		1 PE lokal	1 PE global	4 PEs global	Intel Xeon E5-2680 [14] 1 Thread	Intel Xeon E5-2680 [14] 4 Threads
Hash-Join	Durchsatz in GBit/s	–	0,02	0,07	1,3	3,5
	Leistung in W	–	0,1	0,19	(3)130	(3)130
	Energie in nJ/Bit	–	4,3	2,7	101,6	36,9
	rel. zu Intel	–	9×	14×	0,4×	1×
	ATE-Produkt in $mm^2\,W/(GBit/s)^2$	–	$4{,}5 \cdot 10^3$	$7{,}0 \cdot 10^2$	$1{,}7 \cdot 10^4$	$2{,}3 \cdot 10^3$

(1)TDP entnommen aus [85].
(2)TDP entnommen aus [82].
(3)TDP entnommen aus [86].

zu sortierenden Menge mehrmals aus dem globalen Speicher geladen bzw. abgespeichert wird.

In Bezug auf den Durchsatz des Hash-Joins kann der Tomahawk nicht mit dem GP-Prozessor mithalten. Vor allem die Probe-Phase des Hash-Joins dominiert die Gesamtausführungszeit auf dem Tomahawk, da die PEs mehrfach auf die Hashtabelle und damit über DMA-Transfers auf den globalen Speicher zugreifen müssen. Vergleicht man allerdings wie in Kapitel 4.3.4 (Tabelle 4.5) nur die Verarbeitungszeit des Hashings, dann erreicht ein PE sogar einen leicht höheren Durchsatz als die optimierte x86-CPU. Eine Angleichung der Speicherbandbreite des Tomahawk an die des Intel Xeon E5-2680 (um Faktor 32 auf 409,6 GBit/s) würde deshalb auch einen mit dem Intel ausgeglichenen Gesamtdurchsatz erlauben. Der Intel-Xeon Prozessor mit einer Sandy-Bridge Architektur aus [14] verarbeitet einen Radix-Join, der wie der in dieser Arbeit verwendete Hash-Join die Eingangsrelationen partitioniert, um mehrere unabhängige Hashtabellen zu erhalten. Trotz des niedrigeren Durchsatzes, erreicht der Tomahawk immer noch einen 14× höheren Energiegewinn bzw. ein 3,2× besseres ATE-Produkt gegenüber der Multi-Thread-Ausführung mit dem Intel-Xeon Prozessor. Eine Implementierung der in Abschnitt 5.3.3 beschriebe-

nen parallelen Hash-Joins nur auf den lokalen Speichern ist nicht möglich, da sich die Hashtabelle auf Grund der Größe immer im globalen Speicher befinden muss.

5.5 Schlussfolgerungen

Dieses Kapitel untersuchte die Parallelisierung und die Umsetzung ausgewählter Operatoren für die Datenbank-Anfrageverarbeitung auf dem Tomahawk MPSoC. Dabei konnten die erfolgreiche Integration des DBA ASIP in das Mehrkernsystem präsentiert und mit Hilfe von unterschiedlichen Implementierungsansätzen die daraus resultierenden Vor- und Nachteile analysiert werden. Gegenüber der Ausführung der Algorithmen auf dem Tomahawk3 ist mit Hilfe des iDMA auf dem Tomahawk4 eine bis zu 1,9× Verringerung der Gesamtlaufzeit möglich. Die Erhöhung der Prozessoranzahl von einem auf vier PEs zeigt eine Leistungssteigerung bis zu Faktor 3,5, wenn der Anteil der Datentransferzeit nicht die Gesamtausführungszeit dominiert. Die Skalierbarkeit mit der Elementanzahl ist dabei ebenfalls von der Verteilung der Verarbeitungs- und Transferzeit abhängig. Weiterhin ergab sich, dass die Beschleunigung eines Prozessors durch zusätzliche Instruktionen mindestens im Bereich einer Größenordnung liegen muss, damit trotz des zusätzlichen Datentransfers das gesamte Mehrkernsystem von der reduzierten Verarbeitungszeit profitiert. Im Vergleich zu der flächenmäßig ähnlichen CPU mit Cache von Cadence Tensilica (`APP_570T`), zeigen vier DBAs einen mindestens 6× besseren Durchsatz und Energieverbrauch. Gegenüber hoch-performanten x86-Prozessoren, die erweiterte SSE-Instruktionen nutzen, profitiert die Tomahawk-Plattform hauptsächlich durch eine bessere Energie- und Flächeneffizienz von bis zu drei Größenordnungen. Damit eignet sich die Tomahawk-Plattform als energieeffizienter Coprozessor für die Anfrageverarbeitung in Datenbanksystemen. Zusätzlich erlaubt das Tomahawk MPSoC auch eine Anpassung des Energieverbrauchs indem Taktfrequenz und Versorgungsspannung entsprechend der Lastanforderungen angepasst werden können. Für eine Steigerung des Durchsatzes sollten zukünftige Systeme eine höhere Bandbreite des DRAM und des NoC aufweisen, um den Einfluss des Datentransfers zu reduzieren und damit die Vorteile der Beschleuniger vollständig ausschöpfen zu können. Die folgenden Punkte fassen weitere Resultate zusammen:

- Parallelität auf Taskebene und Partitionierung: Eine Erhöhung des Durchsatzes der Datenbankoperatoren ist durch Ausnutzung mehrerer Prozessoren möglich. Die Limitierung durch den globalen Speicher kann dabei durch Anwenden von Datenlokalität oder das Double-Buffering verringert werden. Ebenso ist die Aufteilung der Eingangsdaten in unabhängige Partitionen entscheidend für eine effiziente Parallelisierung. Zudem sollten zusammenhängende Daten (z. B. Einträge eines Buckets) im DRAM an aufeinanderfolgenden Speicheradressen liegen. Am Beispiel des Hash-Joins erlaubt bereits eine intelligente Datenanordnung im Speicher eine bis zu 30× Leistungssteigerung.
- Speichermodell: Der lokale Speicher eines jeden PMs ist essentiell für die Entlastung des globalen Speichers. Im Gegensatz zum Cache erlaubt ein vom Anwender gesteuerter Speicher eine höhere Leistungsfähigkeit, da so bereits der Datenzugriff in

Software an die Algorithmen angepasst werden kann. Auf Grund der großen Datenmengen bei der Verarbeitung von Datenbankalgorithmen ist ein globaler Speicher unumgänglich.

- Programmiermodell: Für die Ausführung der Operatoren der Datenbank-Anfrageverarbeitung ist ein statisches Scheduling mittels eines gerichteten Datenflussgraphen vorzuziehen, da die Operatoren und deren Parallelität bereits im Voraus feststehen. Die zentrale Scheduling-Einheit steuert das Zuweisen von Tasks, den Datentransfer zwischen den Prozessoren und dem globalen Speicher sowie das Energiemanagement. Der Scheduler benötigt nur Kenntnisse über die Anzahl verfügbarer Prozessoren und die Größe und Speicherbereiche der Daten. Der zusätzliche Zeitaufwand des Schedulers liegt zumeist unter 5 % und beeinflusst damit nicht maßgeblich die Leistungsfähigkeit des Gesamtsystems.
- Vergleich zur Parallelität auf Daten- und Befehlsebene: Das Anwenden von SIMD-Instruktionen und VLIW-Techniken im DBA zeigt zunächst maßgebliche Vorteile gegenüber der Ausführung mit mehreren Prozessoren im Hinblick auf die Leistungsfähigkeit und den Flächenverbrauch. Eine Verdopplung der Anzahl der Verarbeitungseinheiten auf dem MPSoC verdoppelt ebenfalls den Flächenbedarf, wobei sich die Gesamtlaufzeit des Algorithmus im besten Fall halbiert. Hingegen erlaubt das Hinzufügen von erweiterten Instruktionen in einen einzelnen Prozessor für eine vier-fach SIMD-Implementierung eines Datenbankoperators eine nur bis zu 1,7× Flächenerhöhung, während eine Leistungssteigerung von mindestens einer Größenordnung erreicht wird (siehe Kapitel 4.3). In Bezug auf die Skalierbarkeit liegt der Vorteil bei der Parallelität mit mehreren Prozessoren, da hier Durchsatz und Flächenbedarf im Idealfall linear ansteigen. Die Erhöhung der Datenparallelität innerhalb eines Prozessors und die damit verbundene Verringerung der Ausführungszeit hat jedoch ein quadratisches Wachstum der Fläche gezeigt (siehe Abbildung 4.12b in Kapitel 4.3.3). Hinsichtlich der Leistungsaufnahme konnte nur ein lineares Wachstum bei Erhöhung der Daten- als auch der Taskparallelität festgestellt werden.
- Tomahawk als heterogenes System: Die auf dem Tomahawk3 und Tomahawk4 integrierten Datenbankbeschleuniger enthalten jeweils die gleichen Befehlssatzerweiterungen, um die Parallelität mit vier Prozessoren zu untersuchen. Unabhängig davon ist aber das Design eines heterogenen Systems möglich, in dem jeder DBA nur eine Teilmenge der erweiterten Befehle integriert. Bei einem verhältnismäßig geringeren Flächenverbrauch erlaubt dies sowohl eine parallele Verarbeitung der Basisalgorithmen als auch die gleichzeitige Ausführung verschiedenartiger Algorithmen auf mehreren Gruppen von Prozessoren. Diese Aufteilung ist dabei von den benötigten Operatoren der Datenbankanfrage sowie von der Anzahl parallel zu verarbeitender Anfragen abhängig. Zusätzlich muss die Isolation der unterschiedlichen nebenläufig ausgeführten Algorithmen z. B. durch eine intelligente Speicherarchitektur, voneinander unabhängige Scheduling-Einheiten oder ein übergeordnetes Betriebssystem gesichert sein.

Die Untersuchungen in diesem Kapitel sind auf die Systemarchitektur des Tomahawk-Designs mit vier Prozessoren und der festgelegten Bandbreite des DRAM beschränkt.

Das nächste Kapitel analysiert deshalb die Skalierbarkeit des Mehrkernsystems mit Hilfe eines virtuellen Prototypen und mathematischen Modellen.

6 Untersuchungen zur Skalierbarkeit

Die bisherigen Betrachtungen bezogen sich auf die Ausführung der Datenbankoperatoren auf einzelne Prozessoren (Kapitel 4) oder für bis zu vier parallele Kerne auf dem Tomahawk MPSoC (Kapitel 5). Um die Skalierbarkeit des Systems zu untersuchen, ist allerdings die parallele Abarbeitung der Algorithmen auf einer höheren Anzahl von Prozessoren notwendig. Dazu werden Simulationen mit Hilfe eines virtuellen Mehrkernsystems durchgeführt. Das Ziel ist zum einen, weitere Möglichkeiten der Parallelisierung zu untersuchen, andererseits auch ein tieferes Verständnis über das Verhalten des Systems zu bekommen. Damit können Herausforderungen hinsichtlich der Datenparallelität und der Speicherkonfiguration bei der Entwicklung zukünftiger Beschleunigerarchitekturen besser bewältigt werden. Mit zunehmender Komplexität der Systeme steigt jedoch auch der Aufwand der virtuellen Plattformen hinsichtlich Programmierbarkeit sowie Simulationszeit. Um die theoretischen Grenzen der Leistungsfähigkeit für ein gegebenes Mehrkernsystem zu erfassen, ist deshalb die Betrachtung und Auswertung mathematischer Modelle ein wichtiges Vorgehen. Dabei können aus den Messungen und Simulationen der Tomahawk-Plattform analytische Aussagen über das Verhalten der Datenbankalgorithmen abgeleitet werden. Die Abschnitte 6.2 und 6.3 präsentieren diese Modellierung bzw. die daraus gewonnenen Ergebnisse. Danach folgt ein Vergleich der abgeschätzten und gemessenen Ausführungszeiten in Abschnitt 6.4.

6.1 Virtuelle Plattform

Ein virtueller Prototyp ermöglicht die Evaluierung eines komplexen Systems auf einem bestimmten Abstraktionsniveau. Dabei gilt, je höher die Abstraktionsebene, desto geringer die Simulationsdauer, aber desto geringer auch die Genauigkeit der Simulationsergebnisse. In einer Systemebenensimulation kann die korrekte Funktionalität getestet und die Lastverteilung der Anwendung analysiert werden, ohne dabei das exakte zeitliche Verhalten zu berücksichtigen [140]. Im Gegensatz dazu ist für die Untersuchung zur Skalierbarkeit der Tomahawk-Plattform jedoch eine zyklenakkurate Simulationsumgebung für die Abschätzung der Leistungsfähigkeit notwendig. Dabei müssen z. B. Latenzen der Speicherzugriffe oder die Verarbeitungszeiten der Prozessoren Takt für Takt einbezogen werden.

Der in dieser Arbeit verwendete zyklenakkurate Prototyp basiert auf dem bereits in [110] genutzten und weiterentwickelten SystemC-Simulator zur Untersuchung von Kommunikationsmechanismen in Mehrkernsystemen. Der Prototyp enthält mehrere heterogene Verarbeitungseinheiten, die über ein Verbindungsnetzwerk mit einem globalen Speicher (Shared Memory) verbunden sind. Dabei integriert jede Verarbeitungseinheit einen Prozessor, einen lokalen Speicher und einen DMA-Controller. Als Prozessoren werden

dabei die in dieser Arbeit entwickelten Datenbankbeschleuniger mit anwendungsspezifischen Befehlssatz (DBA ASIP) verwendet. Cadence Tensilica stellt dazu die zugehörigen SystemC-Modelle bereit. Der im Tomahawk4 integrierte iDMA-Controller (siehe Kapitel 5.1.3) steht dabei ebenfalls als Softwaremodell zur Verfügung. Zusätzlich wird eine Verarbeitungseinheit als Scheduling-Einheit bestimmt, die für das Zuweisen parallel auszuführender Tasks auf die Verarbeitungseinheiten und das Initiieren der Datentransfers verantwortlich ist. Diese beschriebene Simulationsumgebung stellt die hier definierte Basis des virtuellen Prototyps dar. Weitere Konfigurationsparameter sind die Anzahl der Prozessoren und die Spezifikationen für die jeweiligen angewendeten Datenbankoperatoren. Die Auswertung der Simulationsergebnisse folgt zusammen mit den mathematischen Modellen in Abschnitt 6.3.

6.2 Mathematische Modelle

Für die Betrachtung der mathematischen Modelle wird das allgemeine Systemmodell eines MPSoC aus Abbildung 5.1 in Kapitel 5 zu Grunde gelegt. Mehrere Verarbeitungseinheiten sind über ein Network-on-Chip (NoC) mit einem globalen Speicher (Shared Memory) und einem Scheduler verbunden. Dabei integriert jede Verarbeitungseinheit einen Prozessor, einen lokalen Speicher und einen DMA-Controller. Die Prozessoren arbeiten ausschließlich auf den lokalen Speichern, wobei der Zugriff auf den globalen Speicher nur durch DMA-Controller möglich ist. Die Prozessoren entsprechen dabei den in dieser Arbeit entwickelten Datenbankbeschleuniger, die die lokale Ausführung ausgewählter Algorithmen beschleunigen. Der Scheduler ist genau wie im Tomahawk-Konzept für das Zuweisen paralleler auszuführender Tasks auf die Verarbeitungseinheiten und das initiieren der Datentransfers verantwortlich. Basierend auf dem beschriebenen Systeme, stellen die folgenden Abschnitte die Modelle zum Speedup und zur Ausführungszeit vor.

6.2.1 Speedup

Für die mathematische Betrachtung des Speedups ist der Einfluss der Anzahl der Prozessoren p, der Datentransferzeit t_{Trf} und der Verarbeitungszeit t_{Proc} eines Prozessors auf die Gesamtausführungszeit zu untersuchen. Je nach Abhängigkeit der Datenpartitionen ergeben sich zwei Modelle.

Modell (1)

Zunächst wird angenommen, dass der zu parallelisierende Algorithmus keine Datenabhängigkeiten aufweist. Die Gesamtausführungszeit $t(p)$ in Abhängigkeit der Prozessoranzahl berechnet sich dann mit

$$t(p) = t_{Trf} + \frac{t_{Proc}(p = 1)}{p}. \tag{6.1}$$

Die Datentransferzeit wird als unabhängig von der Anzahl der Prozessoren betrachtet. In Bezug auf das Ahmdahlsche Gesetz stellt diese Transferzeit den sequentiellen Anteil der

Ausführungszeit dar [3]. Der Speedup, d. h. das Verhältnis aus der Gesamtausführungszeit eines einzelnen Prozessor zu p Prozessoren, ist dann

$$\begin{aligned} S_{(1)}(p) &= \frac{t(p=1)}{t(p)} \\ &= \frac{p(1+\alpha)}{p+\alpha} \quad \text{mit} \quad \alpha = \frac{t_{Proc}(p=1)}{t_{Trf}}. \end{aligned} \tag{6.2}$$

Der Speedup konvergiert mit steigender Prozessoranzahl gegen den maximalen Speedup, der gegeben ist mit

$$\begin{aligned} S_{(1),max} &= S_{(1)}(p \to \infty) \\ &= 1+\alpha. \end{aligned} \tag{6.3}$$

Das heißt, je größer der Anteil der Verarbeitungszeit bzw. je kleiner der Anteil der Datentransferzeit an der Gesamtausführungszeit, desto größer der maximale Speedup. Die Hinzunahme von mehr Prozessoren ist also nur dann sinnvoll, wenn die Verarbeitungszeit immer noch größer als die Transferzeit ist. Die Ausnutzung der Befehlssatzerweiterungen verringert die Verarbeitungszeit bei gleicher Transferzeit. In diesem Fall kann der maximal mögliche Speedup bereits bei einer kleineren Prozessoranzahl erreicht werden.

Modell (2)

Ist nun der Algorithmus nicht vollständig parallelisierbar, z. B. durch die zusätzliche Vorverarbeitung durch den Scheduler, entsteht neben dem parallelen Anteil t_{Par} ein sequentieller Anteil t_{Seq}, der nicht parallelisiert werden kann. Mit dieser Verarbeitungszeit $t_{Proc} = t_{Seq} + t_{Par}$ lässt sich wieder die Gesamtausführungszeit in Abhängigkeit der Anzahl der Prozessoren berechnen:

$$t(p) = t_{Trf} + t_{Seq} + \frac{t_{Par}(p=1)}{p}. \tag{6.4}$$

Für den Speedup erhält man dann

$$\begin{aligned} S_{(2)}(p) &= \frac{t(p=1)}{t(p)} \\ &= \frac{p(1+\alpha)(1+\beta)}{p(1+\beta(1+\alpha))+\alpha} \end{aligned} \tag{6.5}$$

mit

$$\alpha = \frac{t_{Proc}(p=1)}{t_{Trf}}, \quad \beta = \frac{t_{Seq}}{t_{Par}(p=1)}.$$

Mit steigender Prozessoranzahl erreicht der Speedup den maximalen Wert

$$\begin{aligned} S_{(2),max} &= S_{(2)}(p \to \infty) \\ &= 1 + \frac{\alpha}{1+\beta(\alpha+1)}. \end{aligned} \tag{6.6}$$

Genau wie in Modell (1) steigt der Speedup, wenn der Anteil der Verarbeitungszeit die Datentransferzeit dominiert. Zusätzlich gilt, je kleiner der sequentielle Anteil des zu parallelisierenden Algorithmus, desto größer ist der maximale Speedup, der mit mehreren Prozessoren erreicht werden kann.

6.2.2 Berechnung der Ausführungszeit

Es schließt sich nun eine mathematische Analyse der Ausführungszeit für die ausgewählten Datenbankoperatoren Sorted-Set Algorithmen, Merge-Sort und Hash-Join an. Dazu werden jeweils die Gleichungen zur Datentransfer- und Verarbeitungszeit aufgestellt, um daraus die Gesamtausführungszeit zu erhalten und mit Hilfe der zuvor eingeführten Speedup-Modelle den Beschleunigungsfaktor zu berechnen. Allgemein wird für jedes Modell die Bandbreite des Network-on-Chip mit B_{NoC} und die des globalen Speichers mit B_{DRAM} angenommen. Sofern nicht anders angegeben, können zunächst die maximal möglichen Werte aus dem Tomahawk MPSoC mit 2 Elem./Takt und 0,8 Elem./Takt für B_{NoC} bzw. B_{DRAM} verwendet werden. Da hier $B_{DRAM} < B_{NoC}$ gilt, ist die Bandbreite des globalen Speichers der limitierende Faktor beim Datentransfer.

Sorted-Set Algorithmen

Die parallele Implementierung der Sorted-Set Algorithmen basiert auf der Auswertung der Adjazenzmatrix aus Abbildung 5.4a, Kapitel 5.3.1. Die beiden Eingangslisten A und B mit der jeweiligen Elementanzahl $|A|$ und $|B|$ sind in $n_{Block,A}$ bzw. $n_{Block,B}$ Blöcke aufgeteilt. Die Anzahl der Einsen in der Matrix gibt die Anzahl der überlappenden Datenblöcke bzw. Tasks an und wird mit n_{Task} bezeichnet. In Kapitel 5.3.1 wurde zwischen zwei Implementierungsansätzen unterschieden, um die Vor- und Nachteile verschiedener Blockeinteilung zu untersuchen. Die Anzahl der Datenblöcke wird in Ansatz (1) durch die Elementanzahl und die Größe der lokalen Datenspeicher und in Ansatz (2) durch die Prozessoranzahl festgelegt. Die Datentransferzeit ist zwar für beide Fälle identische, allerdings nutzt Ansatz (2) das Prinzip des Double-Bufferings, weshalb es zu einer zeitlichen Überschneidung von Transfer- und Verarbeitungszeit kommt. Der Wert r_{DB} gibt dazu den Anteil des Datentransfers an, der während der Verarbeitung auf dem Prozessor auftritt. Es gilt $r_{DB} = 100\,\%$ bei einer vollständigen Überschneidung bzw. $r_{DB} = 0\,\%$, wenn Transfer und Verarbeitung nacheinander erfolgen. Letzteres repräsentiert den Implementierungsansatz (1), der kein Double-Buffering beim Laden nutzt. Die Datentransferzeit verkürzt sich deshalb um den Faktor $1 - r_{DB}$ und lässt sich mit

$$t_{Set,Trf} = \frac{1 - r_{DB}}{B_{DRAM}} \left(\frac{n_{Task}}{n_{Block,A}} |A| + \frac{n_{Task}}{n_{Block,B}} |B| + |C| \right) \tag{6.7}$$

berechnen. Diese Skalierung mit $r_{DB} > 0\,\%$ gilt nur, solange die Verarbeitungszeit größer als die Transferzeit ist. Danach kann im Modell die Transferzeit nur noch um den absoluten Wert der Verarbeitungszeit verkürzt werden. Eine Verringerung der Verarbeitungszeit tritt dabei durch Nutzung der Befehlssatzerweiterung oder durch eine Erhöhung der Prozessoranzahl auf. Die Menge C ist die Ergebnismenge, deren Elementanzahl von dem jeweiligen Algorithmus und der Selektivität sel_{AB} abhängig ist. Konkret ist die Größe der Ergebnisliste für die Operatoren Intersektion, Differenz und Vereinigung gegeben mit

$$\begin{aligned} |C_{Int}| &= sel_{AB} \cdot \min(|A|, |B|), \\ |C_{Diff}| &= |A| - sel_{AB} \min(|A|, |B|), \\ |C_{Verein}| &= |A| + |B| - sel_{AB} \min(|A|, |B|). \end{aligned}$$

Die Selektivität sel_{AB} ist jeweils auf die Intersektion bezogen, d. h. sie definiert das Verhältnis aus den Kardinalitäten der Schnittmenge zur kleineren Eingangsmenge.

Die Verarbeitungszeit ergibt sich ebenfalls aus der Anzahl der Blöcke und der Elemente in den Eingangsmengen, sowie dem Durchsatz eines Prozessors für die Sorted-Set Algorithmen D_{Set}:

$$t_{Set,Proc}(p) = \frac{n_{Task}}{pD_{Set}} \left(\frac{|A|}{n_{Block,A}} + \frac{|B|}{n_{Block,B}} \right). \tag{6.8}$$

Alle n_{Task} zu verarbeitenden Datenblöcke sind unabhängig voneinander und können mit p Prozessoren parallelisiert werden.

Für eine vereinfachte Darstellung der Gleichungen kann eine quadratische Adjazenzmatrix angenommen werden, d. h. es gilt $|A| = |B|$ und $|A| + |B| = N$. Die Gesamtausführungszeit für die parallele Implementierung der Sorted-Set Algorithmen ist dann

$$t_{Set}(N,p) = \underbrace{\frac{1 - r_{DB}}{B_{DRAM}} \left(\frac{Nn_{Task}}{n_{Block}} + |C| \right)}_{Transfer} + \frac{1}{p} \underbrace{\frac{n_{Task}}{n_{Block}} \frac{N}{D_{Set}}}_{par.} + \underbrace{t_{Sched}(n_{Block})}_{seq.}. \tag{6.9}$$

Dabei ist die Elementanzahl der Ergebnisliste gegeben mit

$$|C_{Int}| = \frac{N}{2} sel_{AB}, \quad |C_{Diff}| = \frac{N}{2}(1 - sel_{AB}), \quad |C_{Verein}| = \frac{N}{2}(2 - sel_{AB}).$$

Gleichung 6.9 gilt für beide in Kapitel 5.3.1 erläuterten Implementierungsansätze. Der Unterschied zwischen den Ansätzen wird dabei zum einen durch den Parameter n_{Block} deutlich. Die Anzahl der Datenblöcke wird in Ansatz (1) durch die Elementanzahl und die Größe der lokalen Datenspeicher und in Ansatz (2) durch die Prozessoranzahl festgelegt. Die vom Scheduler einmalig zu Beginn des Algorithmus aufgewendete Zeit t_{Sched}, um die Adjazenzmatrix zu bestimmen, ist nicht parallelisierbar und stellt damit den sequentiellen Anteil der Verarbeitungszeit dar. Da der Scheduler-Overhead von n_{Block} abhängig ist, skaliert dieser je nach gewähltem Implementierungsansatz linear mit N bzw. p. Diese Erkenntnis leitet sich von der Anzahl der Vergleiche ab, die im Mittel (nur Einsen in der Hauptdiagonalen der Adjazenzmatrix) mit $3n_{Block} - 2$ angegeben werden kann. Ein weiterer Unterschied zwischen den Implementierungsansätzen besteht beim Laden der Daten. In Ansatz (1) erfolgt das Laden und Verarbeiten der Datenblöcke zeitlich nacheinander ($r_{DB} = 0\,\%$) und in Ansatz (2) mit Hilfe von Double-Buffering teilweise gleichzeitig ($r_{DB} \geq 0\,\%$).

Der maximale Speedup für Ansatz (1) berechnet sich mit Hilfe von Gleichung 6.6. Die zugehörigen Parameter α und β sind

$$\alpha = B_{DRAM} \frac{Nn_{Task} + n_{Block} D_{Set} t_{Sched}(N)}{D_{Set}(Nn_{Task} + n_{Block}|C|)}, \quad \beta = \frac{n_{Block} D_{Set} t_{Sched}(N)}{Nn_{Task}}.$$

Der Speedup für $p \to \infty$ ist damit direkt proportional zur Elementanzahl und zur Bandbreite des globalen Speichers, sowie indirekt proportional zum Durchsatz und zum Scheduling-Overhead. Für Ansatz (2) gilt Gleichung 6.6 nicht mehr, da der Scheduler-Overhead $t_{Sched}(p)$ als sequentieller Anteil von der Prozessoranzahl abhängig ist. Der Speedup konvergiert dann für $p \to \infty$ gegen Null. Die zugehörige Herleitung ist in Anhang C.3 einzusehen.

Merge-Sort

Für die in Kapitel 5.3.2 vorgestellte parallele Implementierung des Merge-Sort ist im Hinblick auf die Skalierbarkeit nur das Verfahren mit ausgeglichener Lastverteilung (bitonisches Merge-Sort, Fall (3)) sinnvoll, da der Algorithmus nicht auf eine bestimmte Elementanzahl N limitiert ist. Zunächst soll aber trotzdem das Merge-Sort ohne ausgeglichener Lastverteilung (Fall (1)) zum Vergleich betrachtet werden. Der Vorteil dieser Implementierung besteht darin, dass nur ein vollständiger Datentransfer aller N Elemente zu Beginn und am Ende des Algorithmus erforderlich ist. Beim Wechseln in die nächste Teilstufe verbleibt eine Hälfte der Daten in den lokalen Speichern der Prozessoren und der andere Teil wird direkt zum nächsten entsprechenden Prozessor gesendet. Mit p Prozessoren existieren $\log p + 1$ Teilstufen und damit $\log p$ Zwischentransfers mit jeweils $\frac{N}{2}$ Elementen. Die Datentransferzeit für das Merge-Sort ohne Lastverteilung ergibt sich dann mit

$$t_{MS(1),Trf}(N,p) = N\left(\frac{2}{B_{DRAM}} + \frac{\log p}{2B_{NoC}}\right). \tag{6.10}$$

Die Transferzeit liegt dann in $\mathcal{O}(\log p)$, da mit steigender Prozessoranzahl die Anzahl der Teilstufen und deren zugehörige Datentransfers zwischen den Prozessoren zunimmt.

Nach der ersten Teilstufe sind die Daten eines Prozessor bereits sortiert. Damit kann Verarbeitungszeit eingespart werden, da in den darauffolgenden Teilstufen nur noch Teilfolgen der Länge $\frac{N}{2p}$ oder größer verschmolzen werden müssen. Im Folgenden wird deshalb der Durchsatz für das lokale Merge-Sort und das Merge-Sort mit vorsortierten Teilfolgen mit D_{MS} bzw. $D_{MS,presort}$ bezeichnet. Die Verarbeitungszeit enthält zum einen die Sortierzeit von p Prozessoren der ersten Teilstufe sowie die Zeit zum Sortieren der restlichen $\log p$ Teilstufen, wobei sich dann die Prozessoranzahl pro Teilstufe halbiert:

$$\begin{aligned} t_{MS(1),Proc}(N,p) &= \frac{N}{pD_{MS}} + \frac{N}{D_{MS,presort}} \sum_{i=1}^{\log p} \frac{1}{p/2^i} \\ &= \underbrace{\frac{2N}{D_{MS,presort}}}_{seq.} + \frac{1}{p} \underbrace{\left(\frac{N}{D_{MS}} - \frac{2N}{D_{MS,presort}}\right)}_{par.}. \end{aligned} \tag{6.11}$$

In Gleichung 6.11 existiert neben dem parallel ausführbarem Anteil ein zusätzlicher sequentieller Anteil, der auf Grund der nicht ausgeglichenen Lastverteilung entsteht. Die Gesamtausführungszeit für das Merge-Sort ohne Lastverteilung $t_{MS(1)}$ ist dann die Summe aus Transfer- und Verarbeitungszeit (Gleichungen 6.10 und 6.11). Der Speedup konvergiert mit $p \to \infty$ gegen Null, da mit zunehmender Prozessoranzahl der Anteil der Transferzeit zur Gesamtausführungszeit mit $\log p$ steigt.

Für die Berechnung der Ausführungszeit des Merge-Sort mit ausgeglichener Lastverteilung wird der Fall (3) aus Kapitel 5.3.2 zu Grunde gelegt, d. h. die Daten müssen nach jeder Teilstufe wieder in den globalen Speicher zurückgeschrieben werden. Der Fall (2) stellt einen Spezialfall von Fall (3) dar und wird deshalb hier nicht betrachtet. Fall (2) basiert auf dem gleichen bitonischen Sortiernetzwerk aber kann zusätzlich Datenlokalität einbeziehen, da die Größe der zu sortierenden Menge limitiert ist. Für den Fall (3) ergibt sich die Datentransferzeit aus der Anzahl der Teilstufen $\frac{1}{2}(\log^2 n_{Block} + \log n_{Block})$

multipliziert mit der Zeit, $2N$ Elemente zu versenden:

$$t_{MS(3),Trf}(N, n_{Block}) = \frac{N}{B_{DRAM}}(\log^2 n_{Block} + \log n_{Block}). \tag{6.12}$$

Die Anzahl der Elemente pro Datenblock s_{Block} ist durch die Größe der lokalen Speicher der Prozessoren vorgegeben. Erhöht man die Anzahl der Elemente N, ist auch die Anzahl der Blöcke mit $n_{Block} = \frac{N}{s_{Block}}$ anzupassen. Der Sortiergraph wächst demnach mit steigenden N und die Datentransferzeit liegt in $\mathcal{O}(N \log^2 N)$.

Bei der Berechnung der Verarbeitungszeit wird wie zuvor zwischen der Sortierung der ersten Teilstufe mit dem Durchsatz D_{MS} und den nachfolgenden Teilstufen mit $D_{MS,presort}$ unterschieden. Die Anzahl der Tasks in der ersten Teilstufe beträgt $\frac{n_{Block}}{2}$ und in den restlichen Teilstufen $n_{Task} - \frac{n_{Block}}{2}$, wobei n_{Task} die Gesamtanzahl der Tasks ist (siehe Gleichung 5.5). Jeder Task sortiert $2s_{Block}$ Elemente. Die Verarbeitungszeit für das Merge-Sort mit Lastverteilung ergibt sich dann mit

$$\begin{aligned} t_{MS(3),Proc}(N, p) &= \frac{n_{Block} s_{Block}}{p D_{MS}} + \frac{2 s_{Block}}{p D_{MS,presort}} \left(n_{Task} - \frac{n_{Block}}{2} \right) \\ &= \frac{N}{p} \left(\frac{1}{D_{MS}} + \frac{\log^2 n_{Block} + \log n_{Block} - 2}{2 D_{MS,presort}} \right). \end{aligned} \tag{6.13}$$

Da die Anzahl der Datenblöcke wie zuvor von der Elementanzahl abhängig ist, ergibt sich eine Komplexität für die Verarbeitungszeit von $\mathcal{O}(\frac{N}{p} \log^2 \frac{N}{p})$. Um entsprechend der Prozessoranzahl eine ausreichende Anzahl unabhängiger Tasks pro Teilstufe bereitzustellen, müssen in jeder Teilstufe mindestens $2p$ Datenblöcke vorhanden sein ($n_{Block} \geq 2p$). Die Gesamtausführungszeit $t_{MS(3)}(N, p)$ ist dann die Summe aus Transfer- und Verarbeitungszeit (Gleichungen 6.12 und 6.13).

Im Gegensatz zum Merge-Sort ohne Lastverteilung steigt hier die Transferzeit nicht mit der Prozessoranzahl. Außerdem weist die Verarbeitungszeit keinen sequentiellen Anteil auf. Der maximale Speedup, der durch die Nutzung mehrerer Prozessoren erreicht werden kann, konvergiert deshalb nicht gegen Null, sondern leitet sich aus Gleichung 6.3 ab:

$$S_{MS(3),max} = 1 + \frac{B_{DRAM}}{D_{MS,presort}} \left(\frac{D_{MS,presort} - D_{MS}}{D_{MS}(\log^2 n_{Block} + \log n_{Block})} + \frac{1}{2} \right). \tag{6.14}$$

Aus Gleichung 6.14 lässt sich ablesen, dass der Speedup für eine sehr große Anzahl von Prozessoren direkt proportional mit der Speicherbandbreite und der Größe des lokalen Speichers sowie indirekt proportional mit der Gesamtelementanzahl und des Sortierdurchsatzes ist. Der höhere Speedup bei weniger Elementen und geringerem Durchsatz lässt sich wieder auf die Aussage in Abschnitt 6.2.1 zurückführen. Je geringer der Anteil der Datentransferzeit und je größer der Anteil der Verarbeitungszeit an der Gesamtausführungszeit, desto mehr steigt der Speedup mit zunehmender Prozessoranzahl.

Hash-Join

Für die Skalierbarkeit des Hash-Joins werden die beiden in Kapitel 5.3.3 eingeführten Ansätze mit unterschiedlichen Implementierungen der Hashtabelle analysiert. Dabei basiert Ansatz (1) auf einer Hashtabelle deren Einträge innerhalb eines Buckets mit einer

verketteten Liste verknüpft sind (`LL-HT`). Die Gesamtausführungszeit des Algorithmus umfasst die vier Hauptbeiträge Datentransfer, Hashing, Füllen der Hashtabelle und Join. Ansatz (2) nutzt eine Hashtabelle, die auf einem Histogramm und einer Präfixsumme aufbaut (`Hist-HT`). Die Zeit für das Berechnen dieser Datenstrukturen trägt hier zusätzlich zur Gesamtausführungszeit bei.

Zunächst ergibt sich die jeweilige Zeit für den Datentransfer aus den Kardinalitäten der Eingangsrelationen R und S, der Ergebnisrelation $T = R \bowtie S$ sowie aus der Bandbreite des globalen Speichers:

$$t_{LL-HT,Trf} = \frac{(1 - r_{DB})(|R| + |S|) + |T|}{B_{DRAM}}, \tag{6.15}$$

$$t_{Hist-HT,Trf} = \frac{(1 - r_{DB})(2|R| + |S|) + |T|}{B_{DRAM}}. \tag{6.16}$$

Genau wie bei den Sorted-Set Algorithmen wird hier das Prinzip des Double-Bufferings angewendet. Das Laden der Daten und das nachfolgende Hashing finden damit teilweise gleichzeitig statt. Ein Maß für diese zeitliche Überschneidung gibt der Faktor $1 - r_{DB}$ an, der die Transferzeit entsprechend verkürzt. Beim `Hist-HT` Hash-Join findet das Laden und das Hashing der in die Hashtabelle abzubildenden Relation R zweimal statt. Die Zeit für das Ausführen der Hashfunktion berechnet sich dann mit

$$t_{LL-HT,Hash} = \frac{|R| + |S|}{D_{Hash}} \tag{6.17}$$

$$t_{Hist-HT,Hash} = \frac{2|R| + |S|}{D_{Hash}}, \tag{6.18}$$

wobei D_{Hash} den Durchsatz für das Hashing angibt und durch Anwenden der zugehörigen Befehlssatzerweiterung erhöht werden kann.

Die Berechnung des Histogramms und der Präfixsumme existiert nur für den `Hist-HT` Hash-Join. Die dafür benötigte Zeit t_{Hist} verringert sich mit zunehmender Prozessoranzahl, da sich auch die Größe des Histogramms und der Präfixsumme für jeden Prozessor verkleinert. Trotzdem muss jeder Prozessor über die gesamte Relation R iterieren, da erst die jeweiligen Hashwerte die Datenpartitionen bestimmen. Dieses Verhalten kann über einen sequentiellen Anteil in der Ausführungszeit modelliert werden. Mit dem Verhältnis aus sequentieller zu paralleler Laufzeit β_{Hist} und dem Durchsatz D_{Hist} für das Erzeugen des Histogramms und der Präfixsumme berechnet sich die Verarbeitungszeit mit

$$t_{Hist}(p) = \frac{|R|}{D_{Hist}} \left(\frac{1}{p} + \beta_{Hist} \right). \tag{6.19}$$

Im nächsten Schritt werden die Tupel der Relation R in die Hashtabelle eingefügt (Build-Phase). Jeder Prozessor muss dabei über die gesamte Relation R iterieren, fügt aber nur Tupel in die Hashtabelle ein, deren Hashwerte im für diesen Prozessor zugewiesenen Bereich liegen. Deshalb entsteht auch hier ein sequentieller Anteil, der durch das Verhältnis zu dem parallelen Anteil mit β_{Build} angegeben wird. Die Verarbeitungszeit berechnet sich dann mit

$$t_{Build}(p) = \frac{|R|}{D_{Build}} \left(\frac{1}{p} + \beta_{Build} \right), \tag{6.20}$$

wobei D_{Build} den Durchsatz der Build-Phase angibt und die Verarbeitungszeit des Prozessors als auch die Zeit der Schreibzugriffe auf die Hashtabelle enthält, die sich im globalen Speicher befindet. Damit ist dieser Durchsatz direkt abhängig von der Bandbreite des globalen Speichers: $D_{Build} \sim B_{DRAM}$. Gleichung 6.20 gilt für den `LL-HT` als auch für den `Hist-HT` Hash-Join, wobei für beide Implementierungen unterschiedliche Werte für die Parameter D_{Build} und β_{Build} gewählt werden.

Im letzten Schritt des Hash-Joins (Probe-Phase) werden die Tupel aus S mit den Einträgen in der Hashtabelle verglichen um die Ergebnisrelation zu erhalten. Die Laufzeit der Probe-Phase wird mit Hilfe des Durchsatzes D_{Probe} bestimmt, der die Leistungsfähigkeit des Prozessors und die Zeit für die Zugriffe auf den globalen Speicher kennzeichnet. Auch hier entsteht durch das Durchlaufen der gesamten Relation S ein sequentieller Zeitanteil, der ähnlich zu Gleichung 6.20 mit Hilfe des Parameters β_{Probe} charakterisiert werden kann. Der Unterschied zur Build-Phase besteht darin, dass nun nicht in die Hashtabelle geschrieben, sondern von der Hashtabelle gelesen wird. Beim Lesen beeinflusst die Anzahl der Kollisionen in der Hashtabelle die Ausführungszeit. Ein Maß für die mittlere Anzahl von Kollisionen ist das Verhältnis aus der Anzahl der eingefügten Tupel $|R|$ zur Anzahl der Buckets m (siehe auch Kapitel 2.2.2). Die Verarbeitungszeit wird deshalb mit diesem Lastfaktor (LF) $\frac{|R|}{m}$ multipliziert, um den Einfluss der Kollisionen zu modellieren:

$$t_{Probe}(p) = \frac{|R|}{m} \frac{|S|}{D_{Probe}} \left(\frac{1}{p} + \beta_{Probe} \right). \tag{6.21}$$

Damit zeigt sich eine quadratische Zeitkomplexität mit der Tupelanzahl, die bereits bei den Messungen mit dem Tomahawk festgestellt wurde (siehe Abbildung 5.12a in Kapitel 5.4.1). In der Hashtabelle, die auf einem Histogramm aufbaut, liegen alle Einträge eines Buckets auf aufeinanderfolgenden Speicheradressen. Im Gegensatz zur Hashtabelle mit verketteter Liste erlaubt dies das Lesen mehrerer Bucketeinträge mit einem DMA-Transfer und damit einen geringeren Zeitaufwand der Probe-Phase. Für den `Hist-HT` Hash-Join kann dieses Verhalten in Gleichung 6.21 durch eine zusätzliche Skalierung des Lastfaktors mit der maximalen Paketlänge des gegebenen NoC n_{Burst} modelliert werden:

$$LF_{Hist-HT} = \frac{|R|}{m n_{Burst}}.$$

Die Gesamtausführungszeit für den `LL-HT` und den `Hist-HT` Hash-Join in Abhängigkeit der Tupelanzahl N mit $N = |R| = |S|$ und der Prozessoranzahl p ist dann die Summe der einzelnen Beiträge:

$$t_{HJ}(N,p) = t_{Trf}(N) + t_{Hash}(N) + t_{Hist}(N,p) + t_{Build}(N,p) + t_{Probe}(N,p). \tag{6.22}$$

Die Laufzeit zur Berechnung des Histogramms existiert dabei nur für den `Hist-HT` Hash-Join. Die Komplexität wird durch das quadratische Wachstum von t_{Probe} mit der Tupelanzahl bestimmt und liegt damit in $\mathcal{O}(\frac{N^2}{p})$.

Die Gesamtausführungszeit enthält sowohl parallele als auch sequentielle Verarbeitungsanteile. Der Speedup berechnet sich deshalb nach dem in Abschnitt 6.2.1 eingeführten Modell (2) und konvergiert mit steigender Prozessoranzahl gegen einen festen Werte (siehe Anhang C.4).

6.3 Ergebnisse und Auswertungen

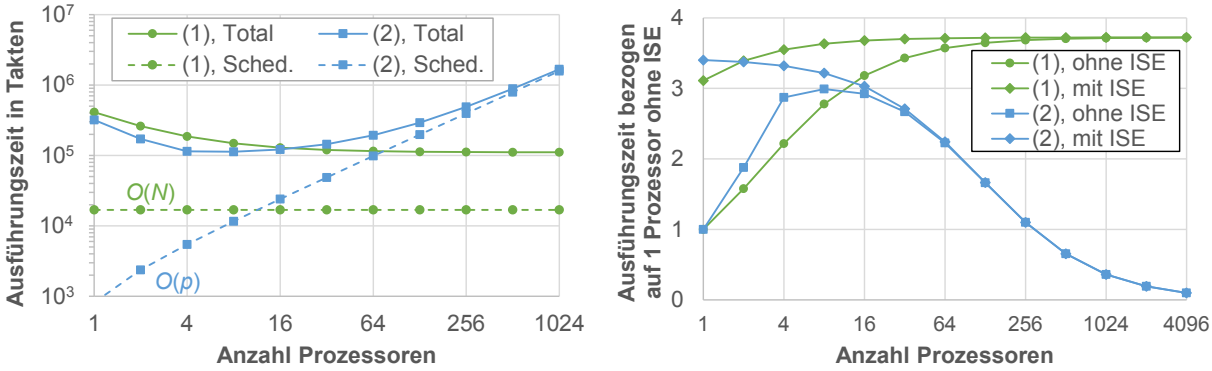
Dieser Abschnitt präsentiert die Ergebnisse aus den Simulationen für die gewählten Datenbankoperatoren und wertet dazu die mathematischen Modelle aus Abschnitt 6.2 aus. Mit Hilfe der Messwerte des Tomahawk MPSoC mit bis zu vier Prozessoren und den Simulationsergebnissen mit bis zu 16 Prozessoren können die Parameter der Modelle ermittelt werden (Fitting). Das ermöglicht den Fehler zwischen Modell und Messung zu minimieren und die Ausführungszeiten für eine größere Anzahl von Prozessoren zu extrapolieren. Des Weiteren kann der Einfluss von Speicherbandbreiten, Datendurchsätzen sowie algorithmusspezifischer Kenngrößen auf die Leistungsfähigkeit untersucht werden. Die ermittelten Parameterwerte für die ausgewählten Datenbankoperatoren sind im Anhang A in Tabelle A.4 zusammengefasst.

6.3.1 Sorted-Set Algorithmen

Für die Sorted-Set Algorithmen werden die beiden in Kapitel 5.3.1 eingeführten Implementierungsansätze untersucht, die sich durch die Größe und Anzahl der Datenblöcke jeder Eingangsmenge unterscheiden. Im Folgenden wird dazu speziell die Sorted-Set Intersektion betrachtet deren Auswertungen ohne Einschränkungen auf die Sorted-Set Vereinigung und Differenz übertragen werden können. Die Verteilung der Eingangsdaten hat hier keinen Einfluss auf die Skalierbarkeit der Leistungsfähigkeit. Es wird deshalb der Standardfall angenommen, d. h. die Elemente in den Eingangsmengen besitzen jeweils gleichförmige Datenverteilungen mit einer Selektivität von 50 %, wobei die Adjazenzmatrix nur Einsen in der Hauptdiagonalen enthält. Wenn nicht anders angegeben, gilt dann $|A| = |B|$ und $|A| + |B| = N$.

Abbildung 6.1a vergleicht die beiden Ansätze hinsichtlich der Ausführungszeit für mehrere Prozessoren. Wie bereits die Messungen auf dem Tomahawk in Kapitel 5.4.1 mit bis zu vier Prozessoren gezeigt haben, ist die Gesamtausführungszeit von Ansatz (2) gegenüber Ansatz (1) etwa um Faktor 2 geringer. Das Double-Buffering ermöglicht hier kürzere Datentransferzeiten. Bei Ansatz (2) steigt jedoch der Scheduler-Overhead mit der Prozessoranzahl linear an. Im Vergleich mit Ansatz (1) führt dies ab 16 Prozessoren zu einer höheren Ausführungszeit, die weiter mit $\mathcal{O}(p)$ steigt. Der Scheduling-Overhead bei Implementierungsansatz (1) ist nur von N abhängig und lässt damit die Gesamtausführungszeit weiterhin skalieren, bis die Datentransferzeit eine weitere Beschleunigung durch mehrere Prozessoren begrenzt.

Um den Einfluss der Befehlssatzerweiterung des Datenbankbeschleunigers auf die Ausführungszeit zu betrachten, ist der Speedup der Sorted-Set Intersektion über der Prozessoranzahl in Abbildung 6.1b dargestellt. Als Referenz dient dabei ein Prozessor, der keine ISE nutzt. Damit wird bereits für die zwei Implementierungsansätze und nur einem Prozessor mit ISE ein Speedup von 3,1 bzw. 3,4 erreicht. Mit zunehmender Prozessoranzahl gleichen sich die Speedups für die Ausführung mit und ohne ISE an, da jeweils die sequentiellen Anteile der Algorithmen die Gesamtausführungszeit dominieren. Ab etwa 1024 Prozessoren nimmt bei Ansatz (1) die Datentransferzeit mehr als 86 % der Gesamtzeit ein und bei Ansatz (2) überwiegt der Scheduler-Overhead mit 92 %. Da sich für Ansatz (2) die benötigte Zeit des Schedulers linear mit der Prozessoranzahl erhöht (sie-



(a) Gesamtlaufzeit (Total) und Anteil der Laufzeit des Schedulers (Sched.), ohne ISE

(b) Speedup bez. auf einen Prozessor

Abbildung 6.1: Ergebnisse der Sorted-Set Intersektion für zwei Implementierungsansätze (1) und (2) in Abhängigkeit der Prozessoranzahl, $N = 50\,000$ Elemente. Der Scheduling-Anteil ist bei Ansatz (1) von der Elementanzahl ($\mathcal{O}(N)$) und bei Ansatz (2) von der Prozessoranzahl ($\mathcal{O}(p)$) abhängig. Ansatz (2) skaliert deshalb nicht mit der Prozessoranzahl.

he Abschnitt 6.2.2), geht der Speedup bei Ansatz (2) gegen Null. Die Nutzung der ISE verliert zwar mit zunehmender Prozessoranzahl ihren Vorteil, jedoch kann damit z. B. für Ansatz (1) bereits bei 16 Prozessoren ein Speedup von 3,7 erreicht werden, der ohne ISE erst bei einem achtfachen der Prozessoranzahl möglich ist. Die zusätzliche Logik der ISE für die Sorted-Set Algorithmen erhöht die Fläche und die Leistungsaufnahme eines Prozessors mit Speicher um Faktor 1,1 bzw. 2,1 (siehe Kapitel 4.3.2). Im Vergleich zur Ausführung mit acht Prozessoren, ist damit durch Ausnutzung der ISE eine Flächen- und Leistungseinsparung von Faktor 7,0 bzw. 3,8 bei gleichem Durchsatz möglich.

Es stellt sich nun die Frage, ob die Ausführungszeit von Ansatz (1) durch den Scheduler-Overhead mit $\mathcal{O}(N)$ bei steigender Elementanzahl limitiert wird. Abbildung 6.2 stellt dazu die Gesamtausführungszeit für mehrere Kardinalitäten der Eingangsmengen dar. Zunächst ist festzuhalten, dass sich die Datentransferzeit linear mit der Anzahl der Elemente erhöht. Bei Ansatz (2) findet der Datentransfer und die Verarbeitung teilweise gleichzeitig statt. Die Balken für die Transferzeit sind in Abbildung 6.2 deshalb entsprechend verkürzt. Der Scheduler-Overhead für Ansatz (1) steigt gleichermaßen mit der Gesamtausführungszeit, sodass dieser prozentuale Anteil unabhängig von der Elementanzahl immer zwischen 13 und 15 % liegt. Ansatz (1) skaliert damit mit der Element- als auch mit der Prozessoranzahl.

Der Datentransfer benötigt im Mittel etwa 70 % der Gesamtausführungszeit. Eine Reduzierung ist nur durch die Erhöhung der Bandbreite des globalen Speichers und des Verbindungsnetzwerkes möglich. Abbildung 6.3 zeigt diesen Einfluss auf die Ausführungs- und Datentransferzeit. Bei einer Verzehnfachung der Speicherbandbreite des Tomahawk MPSoC auf 125 GBit/s erreicht man Bandbreiten der in Kapitel 5.4.3 verwendeten x86-Vergleichsprozessoren. In diesem Fall ist für Implementierungsansatz (1) eine bis zu 5,5× geringere Ausführungszeit möglich. Der Anteil der Datentransferzeit sinkt dabei von 86 auf 38 %. Der Scheduler profitiert nicht von der größeren Bandbreite des globalen Spei-

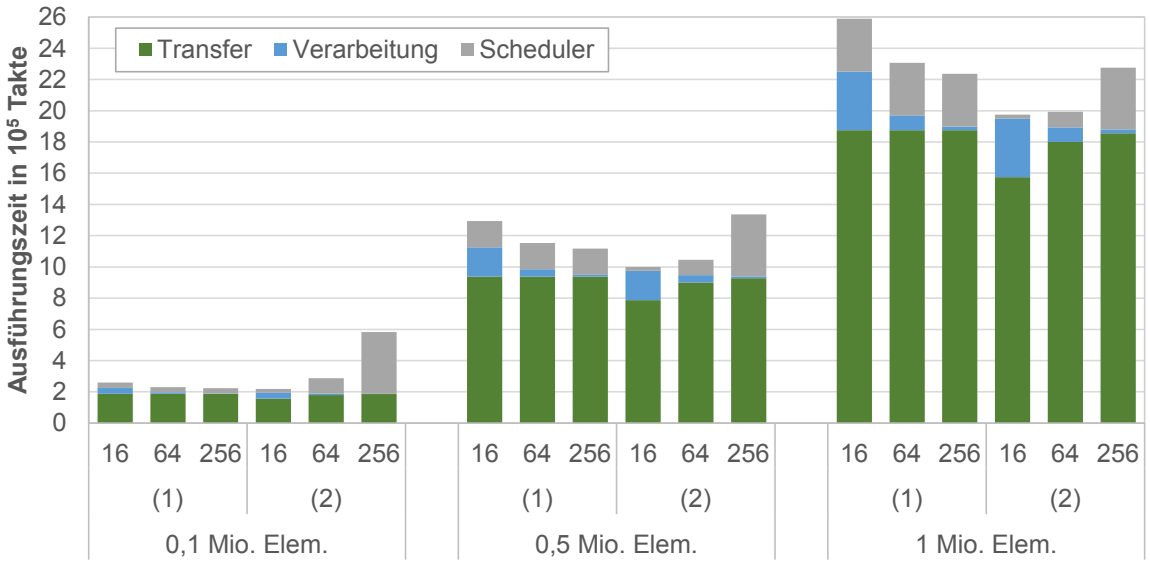
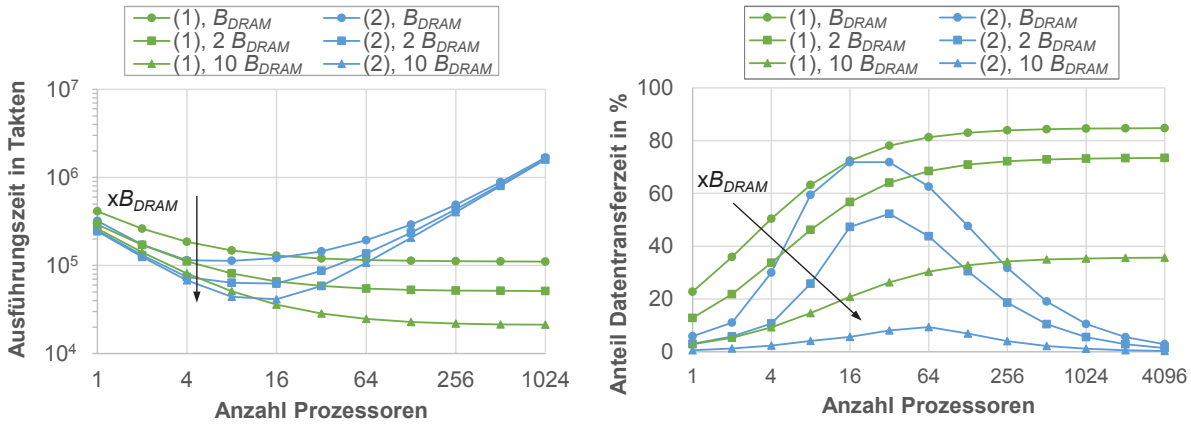


Abbildung 6.2: Ausführungszeit der Sorted-Set Intersektion ohne ISE für 16, 64 und 256 Prozessoren, Implementierungsansätze (1) und (2), sowie für verschiedene Listengrößen ($|A| = |B|$).

chers, da aufeinanderfolgende Lesezugriffe immer an unterschiedlichen Adressen erfolgen. Für Ansatz (2) gleichen sich deshalb die Ausführungszeiten mit zunehmender Prozessoranzahl für unterschiedliche Speicherbandbreiten wieder an. Der zugehörige prozentuale Anteil des Datentransfers in Abbildung 6.3b verringert sich jedoch auf Grund des dominierenden Scheduler-Overheads. Unabhängig davon kann aber für Ansatz (2) die Datentransferzeit im Bereich zwischen 16 und 64 Prozessoren von 68 auf 6 % reduziert werden. In Bezug auf den Speedup und den Scheduler-Overhead ist eine uneingeschränkte Erhöhung der Prozessoranzahl nicht sinnvoll. Bezieht man alle zuvor präsentierten Erkenntnisse ein, dann sind für die Ausführung der Sorted-Set Algorithmen etwa 16 parallele Prozessoren vorzuziehen.

6.3.2 Merge-Sort

In diesem Abschnitt werden die parallelen Implementierungen für das Sortierverfahren Merge-Sort jeweils ohne und mit Lastverteilung ausgewertet (siehe auch Kapitel 5.3.2). Die zu sortierende Gesamtmenge des Merge-Sort ohne Lastverteilung (Fall (1)) ist dabei auf die Größe des lokalen Speichers eines Prozessors beschränkt. Abbildung 6.4 stellt die Ausführungszeit jeweils mit und ohne Nutzung der ISE zunächst mit der maximalen Elementanzahl von $N = 7000$ für eine variable Anzahl von Prozessoren dar. Das zugehörige mathematische Modell wurde in Gleichungen 6.10 und 6.11 gezeigt. Zum einen besteht ein Anstieg der Datentransferzeit mit der Prozessoranzahl ($\mathcal{O}(\log p)$). Bei einer Verdopplung der Prozessoranzahl erhöht sich die Transferzeit um ca. 10 %. Zum anderen teilt sich die Verarbeitungszeit auf Grund der nicht ausgeglichenen Lastverteilung in einen sequentiellen und einen parallelisierbaren Anteil auf. Der prozentuale sequentielle Anteil bleibt auch ohne die Nutzung der ISE bis 512 Prozessoren unter 48 %. Der parallele Anteil halbiert



(a) Ausführungszeit

(b) Datentransferzeit

Abbildung 6.3: Einfluss der Bandbreite des globalen Speichers auf die Ausführungszeit der Sorted-Set Intersektion für zwei Implementierungsansätze (1) und (2) jeweils ohne ISE, $N = 50\,000$ Elemente. Die initiale Bandbreite entspricht der des Tomahawk mit $B_{DRAM} = 0{,}8$ Elem./Takt. Die Bandbreiten des DRAM und des NoC werden gleichermaßen erhöht.

sich mit jeder Verdopplung der Prozessoranzahl. Bei 512 Prozessoren erreicht der Speedup das Maximum und liegt bei 14,3. Eine weitere Erhöhung der Anzahl der Prozessoren ist aber zum einen auf Grund der geringen Elementanzahl nicht sinnvoll. Des Weiteren konvergiert der Speedup für $p \to \infty$ gegen Null, da sich auch die Transferzeit mit der Prozessoranzahl erhöht. Dieser Implementierungsansatz des Merge-Sort ist deshalb für Sortierung von kleinen Datenmengen vorzuziehen und dient hier als Vergleich mit dem folgenden Ansatz des Merge-Sort mit Lastverteilung.

Betrachtet man das mathematische Modell des Merge-Sort mit Lastverteilung (Gleichungen 6.12 und 6.13), so ist eine Skalierbarkeit mit der Element- und Prozessoranzahl möglich. Der Datentransfer ist von der Elementanzahl ($\mathcal{O}(N \log^2 N)$) aber nicht von der Prozessoranzahl abhängig und die Verarbeitungszeit enthält keinen sequentiellen abzuarbeitenden Anteil. Die Gesamtausführungszeit liegt damit in $\mathcal{O}(\frac{N}{p} \log^2 \frac{N}{p})$ wie in Abbildung 6.5 zu sehen. Der Vorteil der Befehlssatzerweiterung wirkt sich nur bis etwa 32 Prozessoren aus, da dann der Anteil der Datentransferzeit mit über 80 % das Laufzeitverhalten dominiert. Der Algorithmus folgt damit genau dem Speedup-Modell (1) aus Abschnitt 6.2.1. Mit Hilfe von Gleichung 6.14 ergibt sich für $p \to \infty$ und $N = 1$ Mio. Elemente ein maximaler Speedup bezogen auf einen Prozessor von 5,9. Dieser wird bereits bei der Ausführung mit 512 Prozessoren erreicht.

Bei den bisherigen Untersuchungen des Merge-Sort wurde die Größe des lokalen Speichers mit $s_{Mem} = 2s_{Block} = 4096$ Elementen konstant gehalten. Nach den Gleichungen 6.12 und 6.13 ist jedoch auch eine Verringerung der Ausführungszeit bei einer Vergrößerung des lokalen Speichers zu erwarten. Dieser Einfluss ist bereits bei weniger als 16 Prozessoren zu erkennen, weshalb im Folgenden die Simulationsergebnisse analysiert werden können. Die in Abbildung 6.6a dargestellte Ausführungszeit verringert sich um bis zu Faktor 1,2 und 1,4, wenn die Speichergröße verdoppelt bzw. vervierfacht wird. Der Grund für die

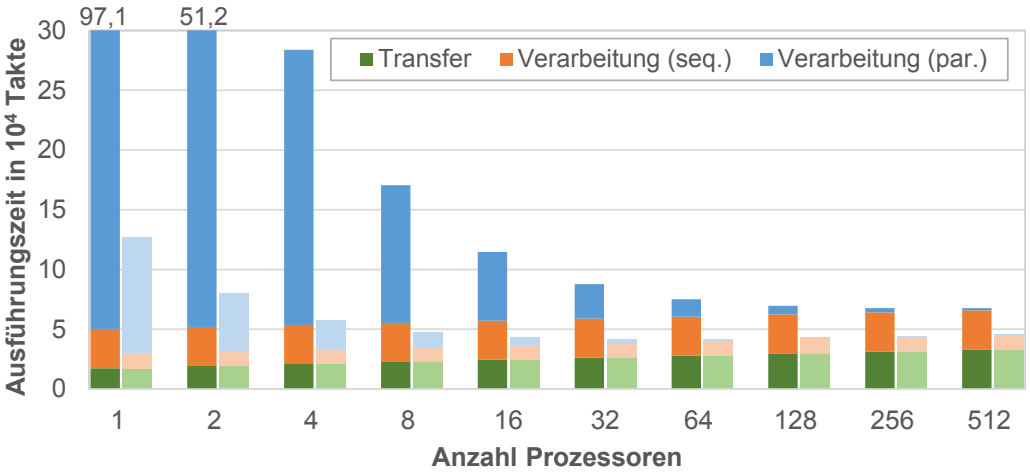
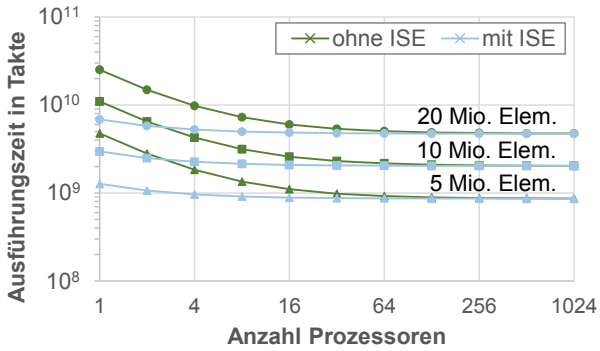


Abbildung 6.4: Ausführungszeit des Merge-Sort ohne Lastverteilung über der Prozessoranzahl, ohne und mit ISE (jeweils dunkle bzw. helle Balken), $N = 7000$ Elemente

zusätzliche Beschleunigung ist zum einen auf eine effizientere Bündelung der Datenblöcke beim Transfer über das Verbindungsnetzwerk sowie auf eine Reduzierung des zeitlichen Overheads zurückzuführen, der durch die Kommunikation der Prozessoren mit dem Scheduler und die Konfiguration der DMA-Einheiten entsteht. Zum anderen verkleinert sich der Sortiergraph. Damit verringert sich zwar die Anzahl der zu sortierenden Datenblöcke, deren Größe steigt aber mit gleicher Komplexität an. Nur der nichtlineare Zusammenhang zwischen der Anzahl der Tasks und der Datenblöcke (siehe Gleichung 5.5) erlaubt dann eine Reduzierung der Gesamtausführungszeit.

Ein Nachteil bei der Vergrößerung des lokalen Speichers zeigt sich jedoch bei Betrachtung der Leerlaufzeit der Prozessoren in Abbildung 6.6b. Ein Prozessor befindet sich im Leerlauf (Idle-Modus), wenn dieser weder Daten vom globalen Speicher lädt oder dort hin speichert noch diese verarbeitet. Der Leerlauf ist damit das Komplement zur Auslastung und tritt auf, wenn der Prozessor auf den nächsten abzuarbeitenden Task wartet. Da der Scheduler die Tasks nacheinander auf die Prozessoren zuweist, erhöht sich die Leerlaufzeit im Allgemeinen mit steigender Prozessoranzahl. Des Weiteren sind die jeweiligen Teilstufen im Sortiergraphen des Merge-Sort mit Lastverteilung immer nacheinander abzuarbeiten. Diese teilweisen Datenabhängigkeiten zwischen den Tasks erzwingen bei unterschiedlichen Verarbeitungszeiten der Prozessoren einen Leerlauf. Die Unterschiede in den Verarbeitungszeiten treten auf, da die Prozessoren niemals gleichzeitig Daten aus dem globalen Speicher laden können. Wie in Abbildung 6.6b zu sehen, erhöht sich die relative Leerlaufzeit zusätzlich, wenn die Anzahl der Prozessoren kein Vielfaches der Anzahl der Datenblöcke ist. Zum Beispiel ist die Menge mit $N = 65\,536$ zu sortierenden Elementen in 16 Datenblöcke pro Teilstufe und mit jeweils $2s_{Block} = 4096$ Elementen pro Datenblock eingeteilt. Eine optimale Lastverteilung ist dann nur bei Verwendung von 1, 2, 4, 8 oder 16 Prozessoren gegeben. Für kleinere Datenblockgrößen entstehen mehr abzuarbeitende Tasks, die dem Scheduler eine höhere Flexibilität bei der Taskzuweisung erlauben. Damit ist die mittlere Auslastung bei einer Verdopplung und Vervierfachung der Speichergröße



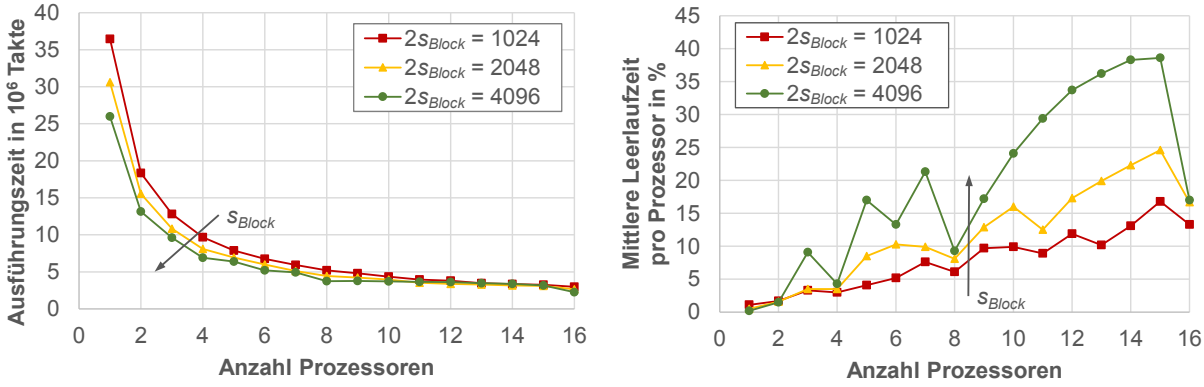
(a) in Abhängigkeit der Prozessoranzahl

(b) in Abhängigkeit der Elementanzahl, ohne ISE, für 1, 16 und 1024 Prozessoren.

Abbildung 6.5: Ausführungszeit des Merge-Sort mit Lastverteilung, $s_{Block} = 2048$ Elemente. Die Laufzeit zeigt eine logarithmische Abhängigkeit von der Prozessor- und Elementanzahl ($\mathcal{O}(\frac{N}{p} \log^2 \frac{N}{p})$).

insgesamt 3,4 bzw. 3,7 % geringer. Das Verhalten wird ebenfalls beim Betrachten des Speedups, der durch die Nutzung mehrerer Prozessoren entsteht, bestätigt (siehe Abbildung A.5 im Anhang A).

Genau wie bei den Sorted-Set Algorithmen profitiert auch das Merge-Sort von einer größeren Bandbreite des globalen Speichers. Nach Gleichung 6.14 wirkt sich der Wert für die Speicherbandbreite direkt linear auf den Speedup aus. Der Einfluss auf die Gesamtausführungszeit erhöht sich mit zunehmender Prozessoranzahl, da dann der zeitliche Anteil des Datentransfers die Gesamtlaufzeit dominiert. Zum Beispiel würde die Verdopplung der Bandbreite des globalen Speichers auf dem Tomahawk MPSoC den Speedup für vier Prozessoren von 2,7 auf 3,4 erhöhen.



(a) Die Ausführungszeit verringert sich mit steigender Größe der Datenblöcke.

(b) Die mittlere Zeit, die ein Prozessor auf den nächsten Task wartet, erhöht sich mit steigender Größe der Datenblöcke.

Abbildung 6.6: Simulationsergebnisse des Merge-Sort mit Lastverteilung bei unterschiedlichen Größen der zu sortierenden Blöcke s_{Block} (Größe des lokalen Speichers), ohne ISE, $N = 65\,536$ Elemente

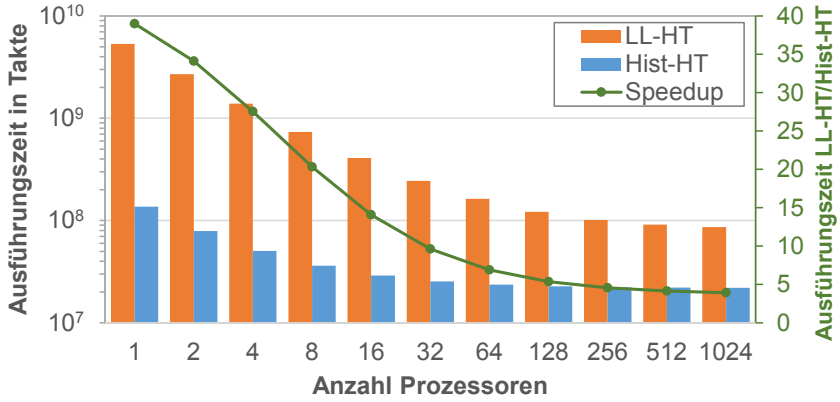


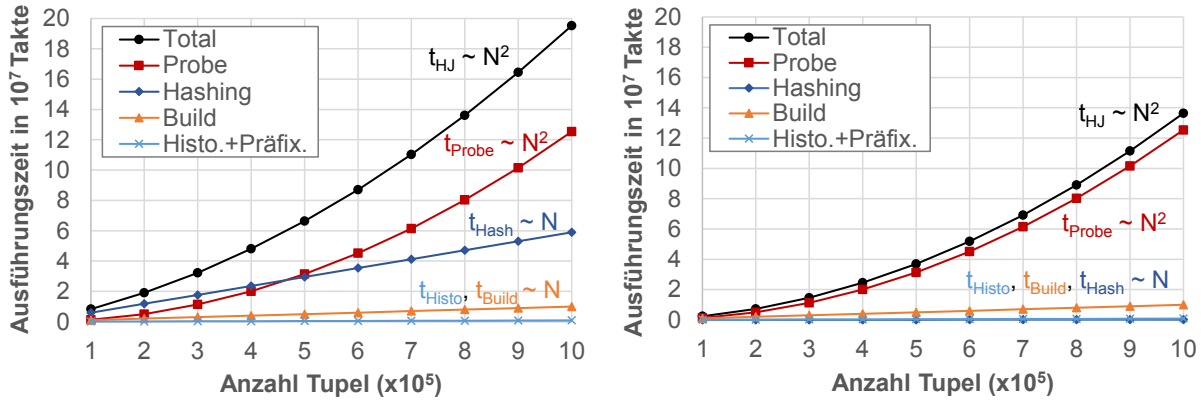
Abbildung 6.7: Ausführungszeit des `Hist-HT` und `LL-HT` Hash-Joins in Abhängigkeit der Prozessoranzahl, $|R| = |S| = 100\,000$ Tupel, mit ISE

6.3.3 Hash-Join

Das Laufzeitverhalten des Hash-Joins, der auf die Tomahawk-Architektur angepasst ist, wird überwiegend durch die Zugriffe auf die Hashtabelle beeinflusst, die sich im globalen Speicher befindet. In Kapitel 5.3.3 wurden dazu zwei Ansätze für die Implementierung der Hashtabelle vorgestellt, die auch bei den folgenden Untersuchungen zur Skalierbarkeit verglichen werden sollen. Mit Hilfe der Messungen auf dem Tomahawk konnte bereits die zeitliche Aufteilung der einzelnen Phasen des Hash-Joins analysiert, sowie der Einfluss der Befehlssatzerweiterung auf das Hashing ausgewertet werden (siehe Abbildung 5.12 in Kapitel 5.4.1). Darauf aufbauend ist nun mit Hilfe der Modelle aus Abschnitt 6.2.2 eine Untersuchung der Ausführungszeit für eine größere Prozessoranzahl möglich.

Abbildung 6.7 vergleicht dazu die Laufzeiten der Hash-Join-Implementierungen mit Hashtabelle, die auf einem Histogramm (`Hist-HT`) und auf einer verketteten Liste (`LL-HT`) aufbauen. Ab etwa 64 und 512 Prozessoren skaliert die Ausführungszeit für den `Hist-HT` bzw. `LL-HT` Hash-Join nicht mehr, da hier der globale Speicher zum Flaschenhals wird. Die Probe-Phase nimmt dann 87 bzw. 90 % der Gesamtlaufzeit ein, die im Modell durch den sequentiellen Anteil in Gleichung 6.21 repräsentiert wird. Zusätzlich ist in Abbildung 6.7 zu erkennen, dass sich die Ausführungszeit von `LL-HT` gegenüber dem `Hist-HT` Hash-Join mit steigender Prozessoranzahl schneller reduziert. Der Speedup sinkt dabei von Faktor 39,0 für einen Prozessor auf Faktor 3,9 für 1024 Prozessoren. Beim `LL-HT` Hash-Join sind zwar mehr Zugriffe auf die Hashtabelle und damit auf den globalen Speicher notwendig, auf Grund der insgesamt höheren Laufzeit ist aber eine bessere parallele Abarbeitung möglich.

Die Gleichungen zum Hash-Join zeigen ein insgesamt quadratisches Wachstum der Ausführungszeit mit der Tupelanzahl. Abbildung 6.8 bestätigt das Verhalten für den `Hist-HT` Hash-Join, wobei das Ausnutzen der ISE lediglich die Gesamtlaufzeit reduziert. Zwar steigt die Laufzeit des Hashings, die Berechnungszeit für das Histogramm und der Build-Phase linear, jedoch wächst die Laufzeit der Probe-Phase quadratisch mit der Tupelanzahl. Dies ist auf die in der Hashtabelle entstehenden Kollisionen zurückzuführen, die die zusätzlichen Lesezugriffe pro Bucket verursachen. Gegenüber dem `Hist-HT` Hash-Join ist der Anstieg der quadratischen Kurve für die Probe-Phase des `LL-HT` Hash-Joins



(a) ohne ISE (b) mit ISE

Abbildung 6.8: Ausführungszeit des `Hist-HT` Hash-Joins in Abhängigkeit der Tupelanzahl für einen Prozessor

mit den gewählten Parametern aus Tabelle A.4 um Faktor 2034 höher. Die verkettete Liste bewirkt im Falle einer Kollision einen zusätzlichen DMA-Zugriff des Prozessors, während bei der Hashtabelle mit Histogramm bereits mehrere aufeinanderfolgende Bucketeinträge mit einem Lesezugriff geladen werden können.

Eine Möglichkeit, um vor allem die Leistungsfähigkeit des `LL-HT` Hash-Joins zu steigern, besteht in einer Reduzierung der Anzahl der Kollisionen bzw. in einer Erhöhung der Anzahl der Buckets in der Hashtabelle. Abbildung 6.9 verdeutlicht diese Abhängigkeit der Ausführungszeit von der Bucketanzahl. Für die zuvor gezeigten Ergebnisse wurden 1024 Buckets angenommen. Das entspricht einer Hashfunktion, die insgesamt zehn Bits selektiert. Im Allgemeinen reduziert sich die Ausführungszeit bei einer Erhöhung der Bucketanzahl. Sind mindestens genauso viele Buckets wie Tupel vorhanden (Lastfaktor ≤ 1), treten im Mittel keine Kollisionen mehr auf und das quadratische Wachstum der Laufzeit mit der Tupelanzahl für die Probe-Phase geht zunehmend in einen linearen Anstieg über. Die Laufzeiten beider Hash-Join-Implementierungen gleichen sich dann an. Der Zeitunterschied für Lastfaktoren größer als Eins ist damit auf die effizientere Kollisionsbehandlung des `Hist-HT` Hash-Joins zurückzuführen.

Eine weitere Möglichkeit, die Skalierbarkeit des Hash-Joins zu steigern, ist die Erhöhung der Bandbreite des globalen Speichers B_{DRAM}. Abbildung 6.10 stellt dazu die Ausführungszeit für verschiedene Bandbreiten sowie den Einfluss der Befehlssatzerweiterung dar. Mit einer Verzehnfachung der Speicherbandbreite auf 125 GBit/s liegt der Wert im Bereich der Bandbreiten moderner GP-Prozessoren. Alle Graphen sind auf den Referenzpunkt für die Ausführung von einem Prozessor ohne ISE mit $1 \times B_{DRAM}$ bezogen. Im Vergleich zu `Hist-HT`, profitiert der `LL-HT` Hash-Join erheblich mehr von einer Bandbreitenerhöhung, da hier mehr Zugriffe auf den DRAM notwendig sind. Bei einer Verzehnfachung der Speicherbandbreite reduziert sich die Laufzeit von `LL-HT` ebenfalls um nahezu Faktor zehn, während der `Hist-HT` Hash-Join nur einen Speedup von 2,7 erreicht. Ohne die Nutzung der ISE geht der Vorteil der Bandbreitenerhöhung mit zunehmender Prozessoranzahl verloren, da nun die Verarbeitungszeit des Hashings die Gesamtlaufzeit dominiert. Beispielsweise benötigt das Hashing bei 1024 Prozessoren und $1 \times B_{DRAM}$ et-

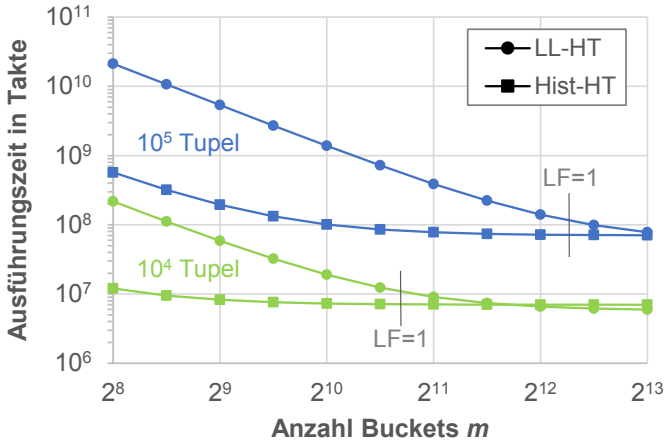
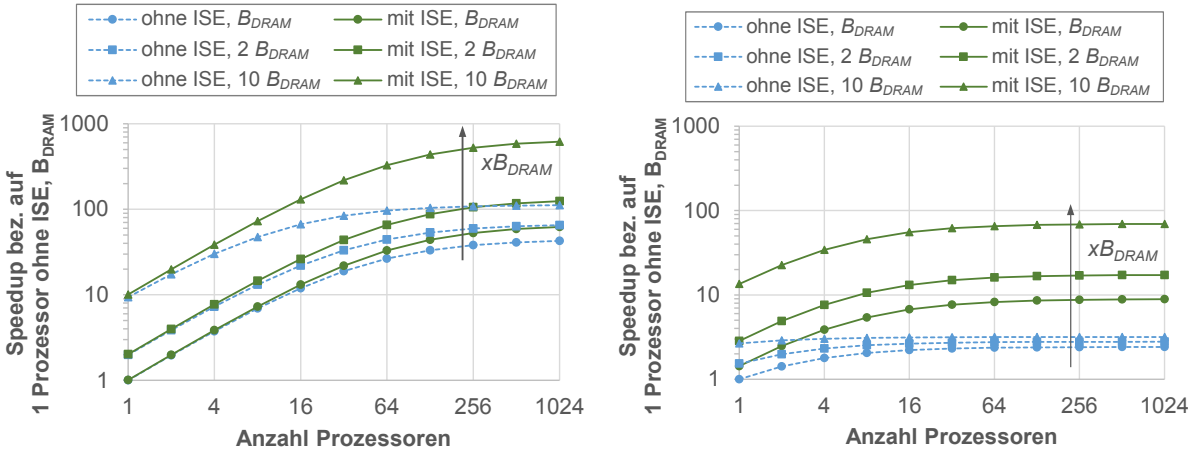


Abbildung 6.9: Ausführungszeit des Hash-Joins in Abhängigkeit der Bucketanzahl. Der Wert des Lastfaktors erhöht sich mit sinkender Bucketanzahl.

wa 73 % und für $10 \times B_{DRAM}$ fast 96 % der Gesamtausführungszeit. Bei Anwenden der ISE für das Hashing dominiert weiterhin die Zeit der Probe-Phase. Die Befehlssatzerweiterung erhält damit die Möglichkeit der Skalierung des Hash-Joins bei zunehmender Prozessoranzahl und für größere Speicherbandbreiten.



(a) `LL-HT` Hash-Join (b) `Hist-HT` Hash-Join

Abbildung 6.10: Einfluss der Bandbreite des lokalen Speichers auf die Ausführungszeit des Hash-Joins, $|R| = |S| = 100\,000$ Tupel, $m = 1024$. Die initiale Bandbreite entspricht der des Tomahawk mit $B_{DRAM} = 25{,}6$ Bit/Takt. Die Bandbreiten des DRAM und des NoC werden gleichermaßen erhöht.

Tabelle 6.1: Vergleich der Ausführungszeiten (in 10^3 Takte) des mathematischen Modells mit den Messwerten des Tomahawk MPSoC für einen und vier Prozessoren jeweils mit ISE

	1 Prozessor			4 Prozessoren		
	Messung	Modell	Abweichung	Messung	Modell	Abweichung
Sorted-Set Intersektion: $N = 100\,000$ Elem.						
$t_{Int(1)}$	312,9	264,6	–15,4 %	231,3	232,0	–0,3 %
t_{Proc}	52,6	43,4	–17,5 %	13,1	10,8	–17,6 %
t_{Trf}	226,6	187,5	–17,3 %	184,5	187,5	+1,6 %
t_{Sched}	33,7	33,7	0,0 %	33,7	33,7	0,0 %
$t_{Int(2)}$	162,1	188,3	+13,9 %	157,8	192,9	+18,2 %
t_{Proc}	43,7	43,4	–0,6 %	10,9	10,9	0,0 %
t_{Trf}	117,5	144,0	+18,4 %	141,4	176,6	+19,9 %
t_{Sched}	0,84	0,81	–3,6 %	5,47	5,46	–0,2 %
Merge-Sort: (1) $N = 7000$ Elem., (3) $N = 65\,536$ Elem.						
$t_{MS(1)}$	142,8	126,7	–11,3 %	69,3	57,8	–16,6 %
t_{Proc}	117,3	109,3	–6,8 %	36,9	36,9	0,0 %
t_{Trf}	25,5	17,4	–31,8 %	32,4	20,9	–35,5 %
$t_{MS(3)}$	5911,6	4380,1	–25,9 %	4658,9	2938,2	–36,9 %
t_{Proc}	1922,5	1922,5	0,0 %	480,6	480,6	0,0 %
t_{Trf}	3989,1	2457,6	–38,4 %	4178,3	2457,6	–41,2 %
Hash-Join: $N = 100\,000$ Tupel						
$t_{HJ,LL–HT}$	5 359 425,2	5 318 826,3	–0,9 %	1 341 521,1	1 390 512,9	–3,5 %
t_{Trf}	97,8	98,4	+0,6 %	95,7	98,4	+2,7 %
t_{Hash}	43,2	43,2	0,0 %	43,3	43,3	0,0 %
t_{Build}	17 765,8	18 091,4	+1,8 %	6134,5	6472,5	+5,2 %
t_{Probe}	5 341 518,4	5 300 593,2	–0,8 %	13 335 379,6	1 383 898,7	+3,5 %
$t_{HJ,Hist–HT}$	138 412,9	136 376,2	–1,5 %	52 198,9	50 485,9	–3,3 %
t_{Trf}	130,6	130,0	–0,5 %	128,7	134,0	+4,0 %
t_{Hash}	65,2	64,5	–1,1 %	65,2	64,5	–1,1 %
t_{Hist}	852,4	859,9	+0,9 %	652,7	652,6	0,0 %
t_{Build}	9857,3	9959,2	+1,0 %	3765,6	3959,2	+4,9 %
t_{Probe}	127 507,4	125 362,6	–1,7 %	47 586,7	45 679,6	–4,0 %

6.4 Fehlerabschätzung

Die Modelle aus Abschnitt 6.2.2 beschreiben die Datentransferzeiten und die Verarbeitungszeiten für die einzelnen Beiträge der Algorithmen. Eine Anpassung an die Messergebnisse erfolgt durch eine geeignete Auswahl der Parameter (siehe Tabelle A.4), sodass der relative Fehler minimiert werden kann. Tabelle 6.1 gibt einen Überblick über die Abweichungen für die Ausführung mit einem und vier Prozessoren bei Nutzung der Befehlssatzerweiterung. Für die Laufzeiten der Datenverarbeitung, die ausschließlich lokal auf den Prozessoren ausgeführt wird, sind überwiegend sehr genaue Voraussagen mit einem relativen Fehler von unter 1 % möglich. Im Mittel weichen die Modelle trotzdem um etwa 13 % von den Messergebnissen ab. Das ist vor allem auf die Unterschiede bei der Modellierung der Datentransferzeit zurückzuführen. Die Modelle sind hinsichtlich der Speicherarchitektur und des Ausführungskonzepts mit einem zentralen Scheduler an das Tomahawk MPSoC individuell angepasst. Jedoch berücksichtigen diese nicht die indivi-

duelle Topologie des NoC sowie damit auftretende Anstauungen (Kongestion) der Datenpakete auf den begrenzten NoC-Ressourcen. Die damit getroffenen Rückschlüsse auf die Skalierbarkeit für eine höhere Prozessoranzahl sind deshalb nur gültig, wenn das NoC nicht die Leistungsfähigkeit der Algorithmen limitiert.

6.5 Schlussfolgerungen

Dieses Kapitel untersuchte die Skalierbarkeit der Sorted-Set Algorithmen, des Merge-Sort und des Hash-Joins hinsichtlich der Ausführung mit einer steigenden Anzahl von Prozessoren sowie bei wachsenden Kardinalitäten. Dabei wurden die Ergebnisse aus Simulationen mit einem virtuellen Prototyp sowie mit analytischen Modellen ausgewertet. Zum einen konnte der Einfluss der Befehlssatzerweiterungen des Datenbankbeschleunigers auf die Laufzeit analysiert werden. Der durch mehrere Prozessoren mögliche Beschleunigungsfaktor ist durch den Anteil des Datentransfers an der Gesamtlaufzeit limitiert. Je größer dieser Anteil, desto weniger kann von der Befehlssatzerweiterung profitiert werden. Bei einer Skalierung der vorhandenen Tomahawk-Plattform durch Hinzufügen von Datenbankbeschleunigern entsteht deshalb eine maximale Prozessoranzahl, mit der noch eine zusätzliche Leistungssteigerung möglich ist. Für die hier untersuchten Algorithmen liegt dieser Wert zwischen 16 und 32 Prozessoren. Zum anderen verringert die Erhöhung der Bandbreite des globalen Speichers die Laufzeiten des Datentransfers sowie alle Verarbeitungsanteile mit Zugriffen auf den DRAM. In diesem Fall beeinflusst der Durchsatz der Datenverarbeitung wieder maßgeblich die Gesamtlaufzeit, der durch Nutzung der Befehlssatzerweiterungen erhöht werden kann.

Für die untersuchte Tomahawk-Plattform ist wie beschrieben die Integration von bis zu 32 Verarbeitungseinheiten sinnvoll. Damit wird durch die Taskparallelität ein maximaler Speedup von Faktor 32 erreicht, während die Fläche um den gleichen Faktor steigt. Mit Hilfe von Befehlssatzerweiterungen und der dabei ausgenutzten Parallelität auf Daten- und Befehlsebene kann jedoch mit einem erweiterten Prozessor, wie in Kapitel 4 gezeigt, ein wesentlich größerer Beschleunigungsfaktor bei einem nur gering ansteigenden Flächenbedarf erzielt werden. Obwohl die Logikfläche für die Erweiterung der SIMD-Register und Instruktionen quadratisch mit dem Durchsatz skaliert (siehe Abbildung 4.12b in Kapitel 4.3.3), ist deshalb ein einzelner Datenbankbeschleuniger hinsichtlich der Leistungsfähigkeit und des Flächenverbrauchs gegenüber dem Mehrkernsystem vorzuziehen.

Um skalierbare Architekturen für die Datenbankanfrageverarbeitung im Sinne der Tomahawk-Plattform zu entwickeln, müssen demnach die Bandbreite des globalen Speichers und des Verbindungsnetzwerkes sowie der lokale Durchsatz der Verarbeitungseinheiten an die Prozessoranzahl angepasst werden können. Wie dazu die Untersuchungen in diesem Kapitel gezeigt haben, ist eine Skalierung der Prozessoranzahl durch eine entsprechende Skalierung der Bandbreiten möglich. Eine weitere Voraussetzung ist, dass der Aufwand des Schedulers nicht die Gesamtlaufzeit dominiert. Die hier vorgeschlagenen analytischen Modelle für die untersuchten Algorithmen zeigen dabei keinen signifikanten Einfluss des Scheduler-Overheads auf das Laufzeitverhalten. Lediglich für einen konkreten Implementierungsansatz der Sorted-Set Operationen weist der zeitliche Scheduling-Anteil ein lineares Wachstum mit der Prozessoranzahl auf, das die Skalierung des Algorithmus

verhindert. Die Modelle stützen sich dabei auf die Annahme, dass der Kommunikationsaufwand zwischen Scheduler und Verarbeitungseinheiten auch auf skalierbaren Architekturen vernachlässigbar bleibt. Ist diese Voraussetzung nicht mehr gegeben, sind erweiterte Scheduling-Konzepte aufbauend auf mehreren Hierarchien oder Clustern zu wählen.

Die hier verwendeten Datensätze enthalten gleichmäßig verteilte Werte. Typischerweise können ungleichmäßige Datenverteilungen zu einer nicht angepassten Lastverteilung führen. Während das hier verwendete Sortierverfahren Merge-Sort grundsätzlich nicht datenabhängig ist, ermöglicht bereits der gewählte Scheduling-Ansatz eine teilweise dynamisch angepasste Lastverteilung für die Sorted-Set Algorithmen. Der Scheduler weist die abzuarbeitenden Tasks auf die jeweils frei verfügbaren Prozessoren zu. Dabei ist nur eine Menge von parallelen Tasks vorgegeben, nicht aber die konkrete Zuweisung der Tasks auf bestimmte Prozessoren. Dies erlaubt auch bei unterschiedlichen Tasklaufzeiten eine teilweise dynamisch angepasste Lastverteilung. Lediglich bei dem hier untersuchten Hash-Join kann eine ungleichmäßige Datenverteilungen dazu führen, dass die Partitionen der Hashtabelle keine annähernd gleiche Größe aufweisen und damit den Durchsatz reduzieren. In diesem Fall erlaubt z. B. das Sampling eine Anpassung der Hashfunktion an die Eingangsdaten und damit eine gleichmäßigere Lastverteilung.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die zunehmenden Anforderungen an Datendurchsatz, Latenz und Verlustleistung sowie die stetig wachsenden Datenmengen erfordern eine permanente Anpassung der Leistungsfähigkeit und Energieeffizienz bestehender Rechnerarchitekturen. Im Bereich der Datenbanken beeinflusst dabei die Anfrageverarbeitung signifikant die Leistungsfähigkeit des Datenbankmanagementsystems. Diese Arbeit untersuchte deshalb die Beschleunigung der für die Datenbank-Anfrageverarbeitung wichtigen Basisoperatoren mit Hilfe von anwendungsspezifischen Hardwarearchitekturen. Zum Beispiel unterstützen die Datenbankbeschleuniger eine energieeffiziente Ausführung von Mengenoperationen, Sortieren und Hashing. Des Weiteren wurde gezeigt, dass neben der Beschleunigung der Anfrageverarbeitung auch die Optimierung des Anfrageplans mit Hilfe der Beschleuniger möglich ist.

Die Datenbankbeschleuniger wurden als ASIC-Blöcke sowie als Prozessor mit erweitertem Befehlssatz (ASIP) realisiert und hinsichtlich ihrer Leistungsfähigkeit, Flächenbedarf und Verlustleistung untersucht. Der erste Ansatz erlaubt eine hohe Beschleunigung der Algorithmen bei gleichzeitig niedrigem Flächen- und Leistungsverbrauch der Hardware. Die Umsetzung ist jedoch aufwendig und nicht flexibel auf weitere Datenbankoperatoren erweiterbar, da eine zusätzliche Verifikation der Ablaufsteuerung der FSM notwendig ist. Im Gegensatz dazu steht beim ASIP-Ansatz bereits ein Basisprozessor zur Verfügung, der die Befehlssteuerung der FSM übernimmt. Die erweiterten Hardwarekomponenten können dann im Programmcode der Software mit neuen Instruktionen angesprochen werden.

Der Datenbankbeschleuniger basierend auf dem ASIP konnte in den Forschungschips Titan3D sowie in den Tomahawk MPSoCs der dritten und vierten Generation implementiert und untersucht werden. Neben den zuvor nur durch Simulationen gewonnenen Ergebnissen, erlaubt der Titan3D erste Messungen eines einzelnen Datenbankbeschleunigers. Die Chipintegration mit Hilfe der Tomahawk-Plattform ermöglicht dann die Parallelisierung der Datenbankoperatoren auf mehreren Verarbeitungseinheiten. Dabei wurden grundlegende Verfahren zur Datenpartitionierung angewendet und auf die Tomahawk-Architektur angepasst. Eine zentrale Scheduling-Einheit steuert dabei das Gesamtsystem durch die Zuweisung der abzuarbeitenden Tasks auf die Verarbeitungseinheiten. Zusätzlich erlaubt das Scheduling eine Anpassung von Durchsatz und Leistungsverbrauch an die Lastanforderungen, indem Taktfrequenzen und Versorgungsspannungen der Verarbeitungseinheiten dynamisch skaliert werden können. Der globale Speicher des Tomahawk MPSoC stellt bei der Ausführung der Datenbankoperatoren den Flaschenhals dar, dessen Limitierung durch Anwenden von Datenlokalität und Double-Buffering verringert werden kann.

Um Aussagen zur Skalierbarkeit des Tomahawk MPSoC treffen zu können, wurden Simulationen mit Hilfe eines virtuellen Prototyps ausgeführt sowie analytische Modelle der

Datenbankoperatoren entworfen und ausgewertet. Die Erkenntnisse zeigen die Komplexität und das Verhalten der Algorithmen bei steigender Prozessoranzahl und wachsenden Kardinalitäten in Abhängigkeit der Speicherbandbreiten und weiterer algorithmusspezifischer Parameter, wie z. B. Größe und Struktur von Datenblöcken oder den Aufbau der Hashtabellen. Des Weiteren erlaubt dies eine Analyse des Scheduling-Aufwandes und dessen Einfluss auf die Gesamtlaufzeit des parallel abzuarbeitenden Algorithmus.

Die in dieser Arbeit entwickelten Beschleunigerarchitekturen zur Datenbank-Anfrageverarbeitung erweitern den Stand der Technik, indem die Vorteile aus bestehenden Realisierungen mit FPGAs und GP-Prozessoren kombiniert werden. Zum einen ermöglicht die anwendungsspezifische Hardware eine bis zu 2,8× Erhöhung des Datendurchsatzes und eine um drei Größenordnungen bessere Energieeffizienz gegenüber optimierten Softwareimplementierungen mit x86-Prozessoren. Der beim ASIP-Ansatz zugrundeliegende RISC-Prozessor stellt dabei zusätzlich die Flexibilität von GP-Prozessoren bereit. Des Weiteren erlaubt der Datenbankbeschleuniger eine Leistungsersparnis von bis zu Faktor 20 im Vergleich zu FPGA-Implementierungen [64].

Die Arbeit lässt sich ebenfalls mit den folgenden als Thesen formulierten Kernaussagen zusammenfassen:

1. Die zunehmenden Anforderungen an Datendurchsatz, Latenz und Verlustleistung sowie die weltweit stetig wachsenden Datenmengen verlangen nach einer permanenten Anpassung der Leistungsfähigkeit und Energieeffizienz bestehender Rechnerarchitekturen.
2. Im Bereich der Datenbanken beeinflussen elementare Operatoren der Anfrageverarbeitung wie z. B. das Suchen, Sortieren oder Hashing signifikant die Leistungsfähigkeit der Prozessorsysteme von Datenbanken.
3. Beschleunigerarchitekturen mit datenbankspezifischer Hardware ermöglichen eine Erhöhung der Leistungsfähigkeit sowie der Energie- und Flächeneffizienz gegenüber existierenden Lösungen für die untersuchten Datenbankoperatoren.
4. Die Implementierungen des Datenbankbeschleunigers können als anwendungsspezifische Hardwareblöcke (ASIC) und als Prozessor mit erweiterten Befehlssatz (ASIP) erfolgen. Dabei wird eine Parallelisierung auf Bit, Daten- und Befehlsebene ausgenutzt. Der ASIC-Ansatz profitiert von einem höheren Durchsatz und einem kleineren Flächenverbrauch, während der im ASIP vorhandene und in Software programmierbare RISC eine höhere Flexibilität bei Änderungen des Programmablaufs bereitstellt.
5. Die vorgeschlagene Architektur des ASIC-Beschleunigers und die erweiterten Befehle des ASIP sind zum einen auf die effiziente Ausführung ausgewählter Datenbankoperatoren spezialisiert, zum anderen kann die verwendete Hardwarelogik auch auf weitere Algorithmen angepasst werden.
6. Die im Programmcode des ASIP ausgeführte Hauptschleife beeinflusst signifikant den Gesamtdurchsatz der Datenbankalgorithmen und ist entsprechend der benötigten Taktanzahl pro Schleifendurchlauf zu minimieren.

7. Die Steigerung der Datenparallelität durch Erhöhen der Datenbreite des Speicherinterfaces bzw. Reduzieren der Breite der Datenelemente verursacht eine Verringerung der maximalen Taktfrequenz.
8. Bei einer Skalierung der Datenbreite des Speicherinterfaces wächst die Fläche der Vektorregister linear, während die Fläche der Verarbeitungslogik quadratisch steigt.
9. Der Leistungsverbrauch des Logikanteils skaliert dann linear mit der Logikfläche und der Taktfrequenz, während der Leistungsverbrauch des Speichers nur von der Taktfrequenz abhängig ist.
10. Die vorgestellten Datenbankbeschleuniger erreichen die Leistungsfähigkeit von optimierten x86-Prozessoren bei einer um bis zu drei Größenordnungen höheren Energie- und Flächeneffizienz.
11. Des Weiteren ermöglichen die Beschleuniger eine Verringerung der Optimierungskosten für die Anfrageoptimierung mittels Selektivitätsabschätzung.
12. Bei der Integration der Datenbankbeschleuniger in das Tomahawk MPSoC ist eine Erhöhung des Durchsatzes durch Ausnutzung der Parallelität auf Taskebene möglich. Dabei stellt der globale Speicher einen Flaschenhals dar, dessen Limitierung durch das Nutzen von lokalen Datenspeichern und das abwechselnde Laden und Speichern in einer Softwarepipeline (Double-Buffering) verringert werden kann.
13. Eine weitere Entlastung des Datenverkehrs zum globalen Speicher ist durch eine intelligente Datenanordnung möglich, bei der verknüpfte Datensätze auf aufeinanderfolgenden Adressbereichen abgelegt werden.
14. Der Ausführungsplan einer Datenbankanfrage entspricht einem azyklischen Datenflussgraphen, der durch Tasks und Datenabhängigkeiten charakterisiert ist. Der Datenflussgraph kann dann mittels einer Scheduling-Einheit auf die parallelen Prozessoren des MPSoC abgebildet werden.
15. Solange die Datentransferzeit im Verhältnis zur Datenverarbeitungszeit gering bleibt, skaliert die Leistungsfähigkeit der Datenbankoperatoren mit steigender Prozessoranzahl und die Befehlssatzerweiterung weist dann den höchsten Einfluss auf den Durchsatz des Gesamtsystems auf.
16. Beeinflusst die Datentransferzeit maßgeblich die Gesamtlaufzeit der Algorithmen, entsteht eine maximale Prozessoranzahl, mit der noch eine zusätzliche Leistungssteigerung durch Hinzufügen von Datenbankbeschleunigern möglich ist. In diesem Fall ist ein einzelner Datenbankbeschleuniger hinsichtlich der Leistungsfähigkeit und des Flächenverbrauchs gegenüber dem Mehrkernsystem vorzuziehen.
17. Zur Untersuchung der Skalierbarkeit der parallelen Implementierungen der Datenbankalgorithmen können analytische Modelle erstellt werden, die die jeweiligen Datentransfer- und Verarbeitungszeiten berechnen. Die Analysen zeigen, dass der theoretisch maximale Beschleunigungsfaktor in Bezug auf die Anzahl der Prozessoren proportional zur Kardinalität der Datensätze und zur Bandbreite des globalen Speichers, sowie indirekt proportional zum lokalen Durchsatz eines Prozessors und zur Vorverarbeitungszeit der Scheduling-Einheit ist.

7.2 Ausblick

Während der Anfertigung der Arbeit entstanden stetig neue Fragen und Forschungsthemen, die im Folgenden kurz diskutiert werden. Ein Thema ist dabei die Auswahl der zu untersuchenden Basisoperatoren in Datenbankanfragen. Die Untersuchungen in dieser Arbeit fokussieren sich auf einen Teil der wichtigsten Algorithmen. Darüber hinaus sind auch die folgenden Erweiterungen für die Beschleunigung der Datenbank-Anfrageverarbeitung von Interesse:

- Weitere Sortieralgorithmen mit unterschiedlichen Komplexitäten und deren Anpassung an die Hardware
- Sortieren von Schlüssel-Wert-Paaren basierend auf dem in dieser Arbeit implementierten Merge-Sort auf einfachen Listen
- Untersuchung und Vergleich des Sort-Merge-Joins als Pendant zum Hash-Join
- Aggregations- und Gruppieralgorithmen auf unsortierten Relationen mit Hilfe von Hashtabellen

Neben der Integration von anwendungsspezifischer Hardware als ASIC oder ASIP in den Verarbeitungseinheiten eines MPSoC ist ebenfalls die Anpassung der DMA-Einheiten denkbar. So könnten während des Datentransfers bereits Operatoren wie die Filterung oder das Hashing ausgeführt werden [143]. Beim Arbeiten mit Relationen können z. B. nur Tupel transferiert werden, deren Schlüssel oder sogar deren Hashwert ein bestimmtes Kriterium erfüllt. In diesem Fall würde das Hashing und die Partitionierung durch den Prozessor für den Hash-Join aus Kapitel 5 entfallen. Es müsste untersucht werden, inwieweit dieser Ansatz die Prozessoren entlastet und damit die Gesamtlaufzeit des Algorithmus verkürzt.

Des Weiteren bietet sich die Möglichkeit der Leistungssteigerung durch eine intelligente Speicherarchitektur auf Modulebene an. Mehrere Verarbeitungseinheiten sind in einem Verarbeitungsmodul integriert und können innerhalb eines Takts auf den gemeinsamen lokalen Speicher zugreifen. Eine Arbitrierungslogik weist nun jeder Verarbeitungseinheit einen bestimmten Adressbereich aus dem lokalen Speicher zu, der zur Laufzeit konfigurierbar ist. Dies ermöglicht eine zusätzliche Datenlokalität, da z. B. die Verarbeitungseinheiten innerhalb des Moduls bereits Zugriff auf die Ergebnisdaten des vorher abgearbeiteten Tasks haben. Weiterführende Untersuchungen müssten deshalb klären, inwieweit die Datenoperatoren von der Datenlokalität profitieren können. Dieses neuartige Konzept ist eine Weiterentwicklung der Speicherarchitektur der Tomahawk-Plattform und ist erstmalig in dem Forschungschip *Kachel* integriert. Die Chipentwicklung erfolgte Anfang 2018 am Vodafone Lehrstuhl für Mobile Nachrichtensysteme der TU Dresden unter Mitwirkung des Autors.

Die Gegenüberstellung der Leistungsfähigkeit und Energieeffizienz des in dieser Arbeit vorgestellten Datenbankbeschleunigers mit modernen Prozessorsystemen zeigte die eindeutigen Vorteile anwendungsspezifischer Hardware. Zukünftige Untersuchungen müssen nun klären, wie diese Vorteile auf das gesamte DBMS übertragen werden können, wenn der Datenbankbeschleuniger als Coprozessor neben den hoch-performanten Prozessoren arbeitet. Die wichtigsten zu untersuchenden Punkte sind:

- Schnittstelle und Kommunikation zwischen Beschleuniger und GP-Prozessor des DBMS
- Anbindung und Architektur des Hauptspeichers: benötigte Bandbreite, Art des Speichersystems
- Anfrageoptimierung und Operatorplatzierung: Welcher Operator kann wann am effizientesten mit dem Beschleuniger ausgeführt werden?
- Anfrageverarbeitung: einheitliche Datenstrukturen zwischen DBMS und Beschleuniger, Scheduling, Lastverteilung bei gleichzeitig laufenden Anfragen

Für das in dieser Arbeit zugrunde gelegte Modell einer relationalen Datenbank, konnte die Leistungssteigerung der Anfrageoperatoren durch den Datenbankbeschleuniger erfolgreich gezeigt werden. Dabei liegen die Daten in Relationen bzw. Tabellen vor. Die Techniken für Vektoroperationen profitieren von diesen geordneten Speicherstrukturen, da mehrere zusammenhängende Elemente gleichzeitig vom Speicher geladen werden können. Darüber hinaus stellt sich die Frage, ob diese Konzepte auch auf alternative Datenstrukturen angewendet werden können, wie sie z. B. in graphenorientierten Datenbanken vorkommen. In Graphdatenbanken liegen die Daten als Knoten und Kanten des Graphen im Speicher vor und sind deshalb für Anfragen angepasst, die häufig Traversierungen verwenden oder nach Beziehungen zwischen Datensätzen suchen. Eine Herausforderung ist dabei, einen geeigneten Ansatz zur Beschleunigung auf Daten- und Befehlsebene zu entwickeln, um die Parallelität der Hardware auszunutzen.

Eigene Publikationen

Journal-Publikationen

Jeronimo Castrillon, Matthias Lieber, Sascha Klüppelholz, Marcus Völp, Nils Asmussen, Uwe Aßmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andres Goens, Sebastian Haas, Dirk Habich, Hermann Härtig, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Akash Kumar, Wolfgang Lehner, Linda Leuschner, Siqi Ling, Steffen Märcker, Christian Menard, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt, Sascha Wunderlich:
A Hardware/Software Stack for Heterogeneous Systems
In: IEEE Transactions on Multi-Scale Computing Systems (TMSCS), Juli 2018

Sebastian Haas, Stefan Scholze, Sebastian Höppner, Annett Ungethüm, Christian Mayr, René Schüffny, Wolfgang Lehner, Gerhard Fettweis:
Application-Specific Architectures for Energy-Efficient Database Query Processing and Optimization
In: Microprocessors and Microsystems (MICPRO), Nov. 2017

Konferenz-Publikationen

Sebastian Haas, Tobias Seifert, Benedikt Nöthen, Stefan Scholze, Sebastian Höppner, Andreas Dixius, Esther Pérez Adeva, Thomas Augustin, Friedrich Pauls, Sadia Moriam, Mattis Hasler, Erik Fischer, Yong Chen, Emil Matúš, Georg Ellguth, Stephan Hartmann, Stefan Schiefer, Love Cederström, Dennis Walter, Stephan Henker, Stefan Hänzsche, Johannes Uhlig, Holger Eisenreich, Stefan Weithoffer, Norbert Wehn, René Schüffny, Christian Mayr, Gerhard Fettweis:
A Heterogeneous SDR MPSoC in 28nm CMOS for Low-Latency Wireless Applications
In: Proceedings of the 54th Annual Design Automation Conference (DAC'17), Austin/Texas, USA, Juni 2017

Annett Ungethüm, Dirk Habich, Tomas Karnagel, Sebastian Haas, Erik Mier, Gerhard Fettweis, Wolfgang Lehner:
Overview on Hardware Optimizations for Database Engines
In: Proceedings of the Datenbanksysteme für Business, Technologie und Web (BTW'17), Stuttgart, Deutschland, März 2017

Sebastian Haas, Oliver Arnold, Stefan Scholze, Sebastian Höppner, Georg Ellguth, Andreas Dixius, Annett Ungethüm, Erik Mier, Benedikt Nöthen, Emil Matúš, Stefan Schiefer, Love Cederström, Fabian Pilz, Christian Mayr, René Schüffny, Wolfgang Lehner, Gerhard Fettweis:
A Database Accelerator for Energy-Efficient Query Processing and Optimization
In: Proceedings of the Nordic Circuits and Systems Conference (NORCAS'16), Kopenhagen, Dänemark, Nov. 2016

Sebastian Haas, Tomas Karnagel, Oliver Arnold, Erik Laux, Benjamin Schlegel, Gerhard Fettweis, Wolfgang Lehner:
HW/SW-Database-CoDesign for Compressed Bitmap Index Processing
In: Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'16), London, England, Juli 2016

Sebastian Haas, Oliver Arnold, Benedikt Nöthen, Stefan Scholze, Georg Ellguth, Andreas Dixius, Sebastian Höppner, Stefan Schiefer, Stephan Hartmann, Stefan Henker, Thomas Hocker, Jörg Schreiter, Holger Eisenreich, Jens-Uwe Schlüßler, Dennis Walter, Tobias Seifert, Friedrich Pauls, Mattis Hasler, Yong Chen, Hermann Hensel, Sadia Moriam, Emil Matúš, Christian Mayr, René Schüffny, Gerhard Fettweis:
An MPSoC for Energy-Efficient Database Query Processing
In: Proceedings of the Design Automation Conference (DAC'16), Austin/Texas, USA, Juni 2016

Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, Wolfgang Lehner:
An Application-Specific Instruction Set for Accelerating Set-Oriented Database Primitives
In: Proceedings of the SIGMOD'14 Special Interest Group on Management of Data (SIGMOD'14), Snowbird/Utah, USA, Juni 2014

Workshop-Publikationen

Sebastian Haas, Gerhard Fettweis:
Energy-Efficient Hash Join Implementations in Hardware-Accelerated MP-SoCs
In: Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS'17), München, Deutschland, Sept. 2017

Marcus Völp, Sascha Klüppelholz, Jeronimo Castrillon, Hermann Härtig, Nils Asmussen, Uwe Aßmann, Franz Baader, Christel Baier, Gerhard Fettweis, Jochen Fröhlich, Andres Goens, Sebastian Haas, Dirk Habich, Mattis Hasler, Immo Huismann, Tomas Karnagel, Sven Karol, Wolfgang Lehner, Linda Leuschner, Matthias Lieber, Siqi Ling, Steffen Märcker, Johannes Mey, Wolfgang Nagel, Benedikt Nöthen, Rafael Peñaloza, Michael Raitza, Jörg Stiller, Annett Ungethüm, Axel Voigt:
The Orchestration Stack: The Impossible Task of Designing Software for Unknown Future Post-CMOS Hardware
In: Proceedings of the International Workshop on Post-Moores Era Supercomputing (PMES'16), Salt Lake City/Utah, USA, Nov. 2016

Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, Tomas Karnagel, Wolfgang Lehner:
HASHI: An Application-Specific Instruction Set Extension for Hashing
In: Proceedings of the International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS'14), Hangzhou, China, Sept. 2014

Literaturverzeichnis

[1] Ackland, B., A. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. J. Nicol, J. H. O'Neill, J. Othmer, E. Sackinger, K. J. Singh, J. Sweet, C. J. Terman und J. Williams: *A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP*. IEEE Journal of Solid-State Circuits, 35(3):412–424, März 2000.

[2] Agrawal, S. R., S. Idicula, A. Raghavan, E. Vlachos, V. Govindaraju, V. Varadarajan, C. Balkesen, G. Giannikis, C. Roth, N. Agarwal und E. Sedlar: *A Many-core Architecture for In-memory Data Processing*. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50'17, S. 245–258, New York, NY, USA, 2017. ACM.

[3] Amdahl, G. M.: *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS'67 (Spring), S. 483–485, April 1967.

[4] Angles, R. und C. Gutierrez: *Survey of Graph Database Models*. ACM Comput. Surv., 40(1):1:1–1:39, Feb. 2008.

[5] Ao, N., F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu und S. Lin: *Efficient Parallel Lists Intersection and Index Compression Algorithms Using Graphics Processing Units*. Proc. VLDB Endow., 4(8):470–481, Mai 2011.

[6] ARM: *Introducing NEON: Development Article*, Juni 2009.

[7] ARM Lim.: *Arm v8-M Architecture Reference Manual*, März 2018. `https://static.docs.arm.com/ddi0553/ah/DDI0553A_h_armv8m_arm.pdf`.

[8] Arnold, O., S. Haas, G. Fettweis, B. Schlegel, T. Kissinger, T. Karnagel und W. Lehner: *HASHI: An Application-Specific Instruction Set Extension for Hashing*. In: *Proceedings of the Fifth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, ADMS'14, S. 25–33, 2014.

[9] Arnold, O., E. Matus, B. Noethen, M. Winter, T. Limberg und G. Fettweis: *Tomahawk: Parallelism and Heterogeneity in Communications Signal Processing MPSoCs*. ACM Trans. Embed. Comput. Syst., 13(3s):107:1–107:24, März 2014.

[10] Azar, Y., A. Z. Broder, A. R. Karlin und E. Upfal: *Balanced Allocations*. SIAM Journal on Computing, 29(1):180–200, 1999.

[11] Babb, E.: *Implementing a Relational Database by Means of Specialized Hardware*. ACM Trans. Database Syst., 4(1):1–29, März 1979.

[12] Baeza-Yates, R.: *A Fast Set Intersection Algorithm for Sorted Sequences*. In: Sahinalp, S. C., S. Muthukrishnan und U. Dogrusoz (Hrsg.): *Combinatorial Pattern Matching*, S. 400–408. Springer Verlag, 2004.

[13] Balkesen, C., G. Alonso, J. Teubner und M. T. Özsu: *Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited*. Proc. VLDB Endow., 7(1):85–96, 2013.

[14] Balkesen, C., J. Teubner, G. Alonso und M. T. Özsu: *Main-Memory Hash Joins on Modern Processor Architectures*. IEEE Transactions on Knowledge and Data Engineering, 27(7):1754–1766, Juli 2015.

[15] Barbara, D.: *Mobile Computing and Databases - A Survey*. IEEE Transactions on Knowledge and Data Engineering, 11(1):108–117, Jan. 1999.

[16] Barbay, J., A. López-Ortiz und T. Lu: *Faster Adaptive Set Intersections for Text Searching*. In: *Experimental Algorithms*, S. 146–157. Springer Verlag, 2006.

[17] Barber, R., G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone und D. Sharpe: *Memory-Efficient Hash Joins*. Proc. VLDB Endow., 8(4):353–364, Dez. 2014.

[18] Bassiouni, M. A.: *Data Compression in Scientific and Statistical Databases*. IEEE Transactions on Software Engineering, SE-11(10):1047–1058, Oct 1985.

[19] Batcher, K. E.: *Sorting Networks and Their Applications*. In: *Spring Joint Computer Conference, 30.April – 2.May 1968*, AFIPS'68 (Spring), S. 307–314, New York, NY, USA, 1968. ACM.

[20] Bauer, H.: *Entwicklung und Analyse von ASIC-Implementierungen für die Verarbeitung von Algorithmen aus dem Bereich Big-Data*. Studienarbeit, TU Dresden, Juni 2015.

[21] Beigne, E., F. Clermidy, D. Lattard, I. Miro-Panades, Y. Thonnart und P. Vivet: *Fine-grain DVFS and AVFS techniques for complex SoC design: An overview of architectural solutions through technology nodes*. In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, S. 1550–1553, Mai 2015.

[22] Benini, L. und G. D. Micheli: *Networks on Chips: A New SoC Paradigm*. Computer, 35(1):70–78, Jan. 2002.

[23] Blanas, S., Y. Li und J. M. Patel: *Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs*. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, S. 37–48. ACM, 2011.

[24] Boncz, P., T. Neumann und O. Erling: *TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark*. In: Nambiar, R. und M. Poess (Hrsg.): *Performance Characterization and Benchmarking*, S. 61–76. Springer International Publishing, 2014.

[25] Boral, H. und D. J. DeWitt: *Database Machines: An Idea Whose Time has Passed? A Critique of the Future of Database Machines*. In: Leilich, H.-O. und M. Missikoff (Hrsg.): *Database Machines*, S. 166–187. Springer Verlag, 1983.

[26] Bouknight, W. J., S. A. Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh und D. L. Slotnick: *The Illiac IV System*. Proceedings of the IEEE, 60(4):369–388, April 1972.

[27] Bress, S., M. Heimel, N. Siegmund, L. Bellatreche und G. Saake: *GPU-Accelerated Database Systems: Survey and Open Challenges*, S. 1–35. Springer Verlag, 2014.

[28] Cadence: *TIE Language–The Fast Path to High-Performance Embedded SoC Processing*. Techn. Ber., Cadence Design Systems, Inc, Jan. 2016.

[29] Casper, J. und K. Olukotun: *Hardware Acceleration of Database Operations*. In: *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA'14, S. 151–160, 2014.

[30] Castrillon, J., M. Lieber, S. Klüppelholz, M. Voelp, N. Asmussen, U. Assmann, F. Baader, C. Baier, G. Fettweis, J. Fröhlich, A. Goens, S. Haas, D. Habich, H. Härtig, M. Hasler, I. Huismann, T. Karnagel, S. Karol, A. Kumar, W. Lehner, L. Leuschner, S. Ling, S. Märcker, C. Menard, J. Mey, W. Nagel, B. Nöthen, R. P. naloza, M. Raitza, J. Stiller, A. Ungethüm, A. Voigt und S. Wunderlich: *A Hardware/Software Stack for Heterogeneous Systems*. IEEE Transactions on Multi-Scale Computing Systems, 4(3):243 – 259, Juli 2018.

[31] Chavan, S., A. Hopeman, S. Lee, D. Lui, A. Mylavarapu und E. Soylemez: *Accelerating Joins and Aggregations on the Oracle In-Memory Database*. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, S. 1441–1452, April 2018.

[32] Chen, S., S. Jiang, B. He und X. Tang: *A Study of Sorting Algorithms on Approximate Memory*. In: *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD'16, S. 647–662. ACM, 2016.

[33] Chhugani, J., A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar und P. Dubey: *Efficient Implementation of Sorting on Multi-core SIMD CPU Architecture*. Proc. VLDB Endow., 1(2):1313–1324, 2008.

[34] Chung, E. S., J. D. Davis und J. Lee: *LINQits: Big Data on Little Clients*. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA'13, S. 261–272. ACM, 2013.

[35] DB-Engines: *DB-Engines Ranking*, Juli 2018. Online verfügbar: `https://db-engines.com/de/ranking`.

[36] Deliège, F. und T. B. Pedersen: *Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps*. In: *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT'10, S. 228–239, 2010.

[37] Dennard, R. H., F. H. Gaensslen, V. L. Rideout, E. Bassous und A. R. LeBlanc: *Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions*. IEEE Journal of Solid-State Circuits, 9(5):256–268, Okt. 1974.

[38] Dennl, C., D. Ziener und J. Teich: *On-the-fly Composition of FPGA-Based SQL Query Accelerators Using A Partially Reconfigurable Module Library*. In: *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, S. 45–52, April 2012.

[39] DeWitt, D. J., S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao und R. Rasmussen: *The Gamma Database Machine Project*. IEEE Transactions on Knowledge and Data Engineering, 2(1):44–62, 1990.

[40] Ding, S., J. He, H. Yan und T. Suel: *Using Graphics Processors for High Performance IR Query Processing*. In: *Proceedings of the 18th International Conference on World Wide Web*, WWW'09, S. 421–430. ACM, 2009.

[41] Dunham, M. H. und A. Helal: *Mobile Computing and Databases: Anything New?*. SIGMOD Rec., 24(4):5–9, Dez. 1995.

[42] Eissa, A. S., M. A. Elmohr, M. A. Saleh, K. E. Ahmed und M. M. Farag: *SHA-3 Instruction Set Extension for A 32-bit RISC Processor Architecture*. In: *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, S. 233–234. IEEE, 2016.

[43] Elmasri, R. und S. B. Navathe: *Fundamentals of Database Systems*. Pearson, 7. Aufl., 2017.

[44] Fauth, A., J. V. Praet und M. Freericks: *Describing Instruction Set Processors Using nML*. In: *Proceedings of the 1995 European Conference on Design and Test*, EDTC'95, S. 503–507, März 1995.

[45] Fettweis, G. P. und E. Matus: *Scalable 5G MPSoC Architecture*. In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*, S. 613–618, Okt. 2017.

[46] Flynn, M. J.: *Some Computer Organizations and Their Effectiveness*. IEEE Transactions on Computers, C-21(9):948–960, Sept. 1972.

[47] Francisco, P.: *The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics*, 2011. Online verfügbar: `https://www.ibmbigdatahub.com/sites/default/files/document/redguide_2011.pdf`.

[48] Furber, S. B., D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple und A. D. Brown: *Overview of the SpiNNaker System Architecture*. IEEE Transactions on Computers, 62(12):2454–2467, Dez. 2013.

[49] Fusco, F., M. P. Stoecklin und M. Vlachos: *NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic*. Proc. VLDB Endow., 3(1-2):1382–1393, Sept. 2010.

[50] Fusco, F., M. Vlachos, X. Dimitropoulos und L. Deri: *Indexing Million of Packets per Second using GPUs*. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC'13, S. 327–332, Okt. 2013.

[51] Gantz, J. und D. Reinsel: *The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East*. IDC iView: IDC Analyze the Future, S. 1–16, Dez. 2012.

[52] Gedik, B., R. R. Bordawekar und P. S. Yu: *CellSort: High Performance Sorting on the Cell Processor*. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB'07, S. 1286–1297. VLDB Endowment, 2007.

[53] Gedik, B., R. R. Bordawekar und P. S. Yu: *CellJoin: A Parallel Stream Join Operator for the Cell Processor*. The VLDB Journal, 18(2):501–519, April 2009.

[54] Ghazal, A., T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte und H.-A. Jacobsen: *BigBench: Towards an Industry Standard Benchmark for Big Data Analytics*. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, S. 1197–1208, 2013.

[55] Gilles, K.: *The semantics of a simple language for parallel programming*. Information Processing, 74:471–475, 1974.

[56] Golab, L. und M. T. Özsu: *Processing Sliding Window Multi-joins in Continuous Queries over Data Streams*. In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB'03, S. 500–511. VLDB Endowment, 2003.

[57] Gold, B., A. Ailamaki, L. Huston und B. Falsafi: *Accelerating Database Operators Using a Network Processor*. In: *Proceedings of the 1st International Workshop on Data Management on New Hardware*, DaMoN'05, 2005.

[58] Google Inc.: *CityHash v1.1.1*. `http://code.google.com/p/cityhash/`, Juni 2013.

[59] Graefe, G.: *Query Evaluation Techniques for Large Databases*. ACM Computing Surveys (CSUR), 25(2):73–169, 1993.

[60] Guzun, G., G. Canahuate, D. Chiu und J. Sawin: *A Tunable Compression Framework for Bitmap Indices*. In: *2014 IEEE 30th International Conference on Data Engineering*, S. 484–495, März 2014.

[61] Haas, S., O. Arnold, B. Nöthen, S. Scholze, G. Ellguth, A. Dixius, S. Höppner, S. Schiefer, S. Hartmann, S. Henker, T. Hocker, J. Schreiter, H. Eisenreich, J.-U. Schlüssler, D. Walter, T. Seifert, F. Pauls, M. Hasler, Y. Chen, H. Hensel, S. Moriam, E. Matúš, C. Mayr, R. Schüffny und G. P. Fettweis: *An MPSoC for Energy-efficient Database Query Processing*. In: *Proceedings of the 53rd Annual Design Automation Conference (DAC'16)*, S. 112:1–112:6, 2016.

[62] Haas, S., O. Arnold, S. Scholze, S. Höppner, G. Ellguth, A. Dixius, A. Ungethüm, E. Mier, B. Nöthen, E. Matúš, S. Schiefer, L. Cederstroem, F. Pilz, C. Mayr, R. Schüffny, W. Lehner und G. P. Fettweis: *A Database Accelerator for Energy-Efficient Query Processing and Optimization*. In: *2016 IEEE Nordic Circuits and Systems Conference (NORCAS)*, S. 1–5, Nov. 2016.

[63] Haas, S., T. Karnagel, O. Arnold, E. Laux, B. Schlegel, G. Fettweis und W. Lehner: *HW/SW-Database-CoDesign for Compressed Bitmap Index Processing*. In: *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, S. 50–57, Juli 2016.

[64] Haas, S., S. Scholze, S. Höppner, A. Ungethüm, C. Mayr, R. Schüffny, W. Lehner und G. Fettweis: *Application-specific architectures for energy-efficient database query processing and optimization*. Microprocessors and Microsystems (MICPRO), 55:119 – 130, 2017.

[65] Haas, S., T. Seifert, B. Nöthen, S. Scholze, S. Höppner, A. Dixius, E. Pérez Adeva, T. Augustin, F. Pauls, S. Modiam, M. Hasler, E. Fischer, Y. Chen, E. Matúš, G. Ellguth, S. Hartmann, S. Schiefer, L. Cederström, D. Walter, S. Henker, S. Hänzsche, J. Uhlig, H. Eisenreich, S. Weithoffer, N. Wehn, R. Schüffny, C. Mayr und G. Fettweis: *A Heterogeneous SDR MPSoC in 28nm CMOS for Low-Latency Wireless Applications*. In: *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*, S. 47:1–47:6, 2017.

[66] Hayes, T., O. Palomar, O. Unsal, A. Cristal und M. Valero: *Vector Extensions for Decision Support DBMS Acceleration*. In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, S. 166–176, 2012.

[67] He, B.: *When Data Management Systems Meet Approximate Hardware: Challenges and Opportunities*. Proc. VLDB Endow., 7(10):877–880, Juni 2014.

[68] He, B., K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo und P. Sander: *Relational Joins on Graphics Processors*. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, S. 511–524. ACM, 2008.

[69] Heimel, M., M. Saecker, H. Pirk, S. Manegold und V. Markl: *Hardware-Oblivious Parallelism for In-memory Column-Stores*. Proc. VLDB Endow., 6(9):709–720, Juli 2013.

[70] Hennessy, J., N. Jouppi, F. Baskett und J. Gill: *MIPS: A VLSI Processor Architecture*. In: *VLSI Systems and Computations*, S. 337–346. Springer Verlag, 1981.

[71] Hennessy, J. L. und D. A. Patterson: *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5. Aufl., 2011.

[72] Herlihy, M., N. Shavit und M. Tzafrir: *Hopscotch Hashing*. In: Taubenfeld, G. (Hrsg.): *Distributed Computing*, S. 350–364. Springer Verlag, 2008.

[73] Hoffmann, A., H. Meyr und R. Leupers: *Architecture Exploration for Embedded Processors with LISA*. Springer Verlag, 2002.

[74] Höppner, S., S. Hänzsche, G. Ellguth, D. Walter, H. Eisenreich und R. Schüffny: *A Fast-Locking ADPLL With Instantaneous Restart Capability in 28-nm CMOS Technology*. IEEE Transactions on Circuits and Systems II: Express Briefs, 60(11):741–745, Nov. 2013.

[75] Höppner, S., C. Shao, H. Eisenreich, G. Ellguth, M. Ander und R. Schüffny: *A Power Management Architecture for Fast Per-Core DVFS in Heterogeneous MPSoCs*. In: *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*, S. 261–264, Mai 2012.

[76] Höppner, S., D. Walter, T. Hocker, S. Henker, S. Hänzsche, D. Sausner, G. Ellguth, J.-U. Schlüssler, H. Eisenreich und R. Schüffny: *An Energy Efficient Multi-Gbit/s NoC Transceiver Architecture With Combined AC/DC Drivers and Stoppable Clocking in 65 nm and 28 nm CMOS*. IEEE Journal of Solid-State Circuits, 50(3):749–762, März 2015.

[77] Hu, Y. C., M. Patel, D. Sabella, N. Sprecher und V. Young: *Mobile Edge Computing A key technology towards 5G*. ETSI White Paper No. 11, Sept. 2015.

[78] IBM Corp.: *IBM DB2 12 for z/OS Technical Overview*, 2016. Online verfügbar: `http://www.redbooks.ibm.com/redbooks/pdfs/sg248383.pdf`.

[79] IEEE International Roadmap for Devices and Systems (IRDS): *Executive Summary*, 2017. Online verfügbar: `https://irds.ieee.org/images/files/pdf/2017/2017IRDS_ES.pdf`.

[80] Ilarri, S., E. Mena und A. Illarramendi: *Location-dependent Query Processing: Where We Are and Where We Are Heading*. ACM Comput. Surv., 42(3):12:1–12:73, März 2010.

[81] Imielinski, T. und B. R. Badrinath: *Querying in highly mobile distributed environments*. In: *Proc. VLDB Endow.*, Bd. 92, S. 41–52, 1992.

[82] Intel Corp.: *Intel Core 2 Quad Processor Q9550*. `https://ark.intel.com/products/33924`.

[83] Intel Corp.: *Intel Core i7-3960X Processor Extreme Edition*. https://ark.intel.com/products/63696.

[84] Intel Corp.: *Intel Core i7-4550U Processor*. https://ark.intel.com/products/75112.

[85] Intel Corp.: *Intel Core i7-920 Prozessor*. https://ark.intel.com/de/products/37147.

[86] Intel Corp.: *Intel Xeon Prozessor E5-2680*. https://ark.intel.com/de/products/64583.

[87] Intel Corp.: *Intel 64 and IA-32 Architectures Software Developer's Manual*, März 2018.

[88] Kahle, J. A., M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer und D. Shippy: *Introduction to the Cell multiprocessor*. IBM Journal of Research and Development, 49(4.5):589–604, Juli 2005.

[89] Kaldewey, T., G. Lohman, R. Mueller und P. Volk: *GPU Join Processing Revisited*. In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN'12, S. 55–62, 2012.

[90] Kanev, S., J. P. Darago und K. Hazelwood: *Profiling a warehouse-scale computer*. In: *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA'15, Juni 2015.

[91] Kemper, A. und A. Eickler: *Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag, 10. Aufl., 2015.

[92] Kim, C., T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. S. J. Chhugani, A. D. Blas und P. Dubey: *Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs*. PVLDB, 2009.

[93] Kim, W., D. Gajski und D. J. Kuck: *A Parallel Pipelined Relational Query Processor*. ACM Trans. Database Syst., 9(2):214–235, Juni 1984.

[94] Kitsuregawa, M., H. Tanaka und T. Moto-Oka: *Application of Hash to Data Base Machine and Its Architecture*. New Generation Computing, 1(1):63–74, März 1983.

[95] Kocberber, O., B. Grot, J. Picorel, B. Falsafi, K. Lim und P. Ranganathan: *Meet the Walkers: Accelerating Index Traversals for In-memory Databases*. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, S. 468–479, 2013.

[96] Kumar, K. und Y. Lu: *Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?*. Computer, 43(4):51–56, April 2010.

[97] Kuon, I. und J. Rose: *Measuring the gap between FPGAs and ASICs*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 26(2):203–215, 2007.

[98] Lee, E. A. und D. G. Messerschmitt: *Synchronous Data Flow*. Proceedings of the IEEE, 75(9):1235–1245, Sept. 1987.

[99] Lenell, J. und N. Bagherzadeh: *A performance comparison of several superscalar processor models with a VLIW processor*. Microprocessors and Microsystems, 18(3):131 – 139, 1994.

[100] Li, P., J. L. Shin, G. Konstadindis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. D. Lin, P. Loewenstein, H. Park, V. Srinivasan, D. Huang, C. Hwang, W. Hsu und C. McAllister: *A 20nm 32-Core 64MB L3 Cache SPARC M7 Processor*. In: *ISSCC Dig. Tech. Papers*, ISSCC'15, S. 72–73, Feb. 2015.

[101] Lieberman, M. D., J. Sankaranarayanan und H. Samet: *A Fast Similarity Join Algorithm Using Graphics Processing Units*. In: *2008 IEEE 24th International Conference on Data Engineering*, S. 1111–1120, April 2008.

[102] Limberg, T., M. Winter, M. Bimberg, R. Klemm, M. Tavares, H. Ahlendorf, E. Matúš, G. Fettweis, H. Eisenreich, G. Ellguth und J.-U. Schlüssler: *A Heterogeneous MPSoC with Hardware Supported Dynamic Task Scheduling for Software Defined Radio*. In: *Design Automation Conference*, Bd. 2009, S. 97–98, 2009.

[103] Lorincz, J., T. Garma und G. Petrovic: *Measurements and Modelling of Base Station Power Consumption under Real Traffic Loads*. Sensors, 12(4):4281–4310, 2012.

[104] Mamoulis, N. und D. Papadias: *Selectivity Estimation of Complex Spatial Queries*. In: *International Symposium on Spatial and Temporal Databases*, S. 155–174. Springer Verlag, 2001.

[105] Manegold, S., P. Boncz und M. Kersten: *Optimizing Main-Memory Join on Modern Hardware*. IEEE Transactions on Knowledge and Data Engineering, 14(4):709–730, Juli 2002.

[106] McGregor, D. R., R. G. Thomson und W. N. Dawson: *High Performance Hardware for Database Systems*. In: *Proceedings of the Second International Conference on Systems for Large Data Bases*, VLDB'76, S. 103–116. VLDB Endowment, 1976.

[107] Mert, Y. M. und O. S. Simsek: *Employing Dynamic Body-Bias for Short Circuit Power Reduction in SRAMs*. In: *Sixteenth International Symposium on Quality Electronic Design*, S. 267–271, März 2015.

[108] Mueller, R., J. Teubner und G. Alonso: *Data Processing on FPGAs*. Proc. VLDB Endow., 2(1):910–921, 2009.

[109] Munshi, A.: *The OpenCL specification*. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*, S. 1–314, Aug. 2009.

[110] Nöthen, B.: *Untersuchungen von Kommunikationsmechanismen in heterogenen Mehrprozessorsystemen*. Dissertation, TU Dresden, Okt. 2015.

[111] Nöthen, B., O. Arnold, E. P. Adeva, T. Seifert, E. Fischer, S. Kunze, E. Matus, G. Fettweis, H. Eisenreich, G. Ellguth, S. Hartmann, S. Höppner, S. Schiefer, J.-U. Schlüssler, S. Scholze, D. Walter und R. Schüffny: *A 105GOPS 36mm^2 Heterogeneous SDR MPSoC with Energy-Aware Dynamic Scheduling and Iterative Detection-Decoding for 4G in 65nm CMOS*. In: *ISSCC Dig. Tech. Papers*, ISSCC'14, S. 188–189, Feb. 2014.

[112] O'Neil, P., E. O'Neil, X. Chen und S. Revilak: *The Star Schema Benchmark and Augmented Fact Table Indexing*. In: Nambiar, R. und M. Poess (Hrsg.): *Performance Evaluation and Benchmarking*, S. 237–252. Springer Verlag, 2009.

[113] Oracle Corp.: *MySQL 5.7 Reference Manual*, 2017. Online verfügbar: `https://downloads.mysql.com/docs/refman-5.7-en.a4.pdf`.

[114] Oracle Corp.: *SPARC M8 Processor*. Oracle Data Sheet, Sept. 2017.

[115] Osseiran, A., F. Boccardi, V. Braun, K. Kusume, P. Marsch, M. Maternia, O. Queseth, M. Schellmann, H. Schotten, H. Taoka, H. Tullberg, M. A. Uusitalo, B. Timus und M. Fallgren: *Scenarios for 5G Mobile and Wireless Communications: The vision of the METIS Project*. IEEE Communications Magazine, 52(5):26–35, Mai 2014.

[116] Ottmann, T. und P. Widmayer: *Algorithmen und Datenstrukturen*. Springer Verlag, 6. Aufl., 2017.

[117] Owen, K. und L. Daniel: *Compressed bitmap indexes: beyond unions and intersections*. Software: Practice and Experience, 46(2):167–198, Sept. 2014.

[118] Pagh, R. und F. F. Rodler: *Cuckoo Hashing*. Journal of Algorithms, 51(2):122 – 144, 2004.

[119] PostgreSQL Global Development Group: *PostgreSQL 9.6.4 Documentation*, 2017. Online verfügbar: `https://www.postgresql.org/files/documentation/pdf/9.6/postgresql-9.6-A4.pdf`.

[120] Psaroudakis, I., T. Kissinger, D. Porobic, T. Ilsche, E. Liarou, P. Tözün, A. Ailamaki und W. Lehner: *Dynamic Fine-grained Scheduling for Energy-efficient Main-memory Queries*. In: *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN'14, S. 1:1–1:7, 2014.

[121] Rahman, N. A. und T. S. Saad: *Hash Join Algorithms Used in Text-Based Information Retrieval: Guidelines for Users*. In: *2008 International Symposium on Information Technology*, Bd. 2, S. 1–7, Aug. 2008.

[122] Ramakrishnan, R. und J. Gehrke: *Database Management Systems*. McGraw-Hill, 3. Aufl., Jan. 2003.

[123] Raman, V., L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang und F.-L. Ling: *Lazy, Adaptive Rid-list Intersection, and Its Application to Index Anding*. In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD'07, S. 773–784, 2007.

[124] Rauber, T. und G. Rünger: *Parallele Programmierung*. Springer Verlag, 3. Aufl., 2012.

[125] Rossbach, C. J., Y. Yu, J. Currey, J.-P. Martin und D. Fetterly: *Dandelion: a Compiler and Runtime for Heterogeneous Systems*. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, S. 49–68. ACM, 2013.

[126] Rossi, D., A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Bartolini, P. Flatresse und L. Benini: *A 60 GOPS/W, -1.8V to 0.9V body bias ULP cluster in 28nm UTBB FD-SOI technology*. Solid-State Electronics, 117:170 – 184, 2016.

[127] Saake, G., K.-U. Sattler und A. Heuer: *Datenbanken: Implementierungstechniken*. mitp Verlag, 3. Aufl., 2011.

[128] Satish, N., C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim und P. Dubey: *Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort*. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, S. 351–362. ACM, 2010.

[129] Satoh, T., H. Takeda, U. Inoue und H. Fukuoka: *Acceleration of Join Operations by a Relational Database Processor, RINDA*. In: *2nd International Symposium on Database Systems for Advanced Applications*, S. 243–248, 1991.

[130] Schlegel, B., T. Willhalm und W. Lehner: *Fast Sorted-Set Intersection using SIMD Instructions*. In: *Proceedings of the Second International Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures*, ADMS'11, 2011.

[131] Sedgewick, R.: *Algorithms in Java, Parts 1–4 (Fundamental Algorithms, Data Structures, Sorting, Searching)*. Addison-Wesley, 3. Aufl., Juli 2002.

[132] Shi, S., Y. Qi und Q. Wang: *Accelerating Intersection Computation in Frequent Itemset Mining with FPGA*. In: *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, S. 659–665, Nov. 2013.

[133] Siemens AG: *Mobile Communication ICs*, 1996. https://datasheet.datasheetarchive.com/originals/library/Datasheets-DAV2/DSADA0026587.pdf.

[134] Šimková, M., Z. Přikryl, Z. Kotásek und T. Hruška: *Automated Functional Verification of Application Specific Instruction-set Processors*, S. 128–138. Springer Verlag, 2013.

[135] Slingerland, N. T. und A. J. Smith: *Multimedia Extensions for General Purpose Microprocessors: A Survey*. Microprocessors and Microsystems, 29(5):225 – 246, 2005.

[136] Stockinger, K. und K. Wu: *Bitmap Indices for Data Warehouses*. In: *Data Warehouses and OLAP: Concepts, Architectures and Solutions*, S. 157–178. IGI Global, 2007.

[137] Synopsys: *DesignWare Processor IP Portfolio*. Techn. Ber. 05/17.CE.CS9217, Synopsys, Inc, 2017.

[138] Synopsys, Inc.: *ASIP Designer: Design Tool for Application-Specific Instruction-Set Processors*, April 2018. https://www.synopsys.com/dw/doc.php/ds/cc/asip-designer-ds.pdf.

[139] Tatikonda, S., F. Junqueira, B. B. Cambazoglu und V. Plachouras: *On Efficient Posting List Intersection with Multicore Processors*. In: *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, S. 738–739. ACM, 2009.

[140] Teich, J. und C. Haubelt: *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer Verlag, 2. erw. Aufl., 2007.

[141] Transaction Processing Performance Council: *TPC Benchmark H, Standard Specification*, 2017. Online verfügbar: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf.

[142] Tsirogiannis, D., S. Guha und N. Koudas: *Improving the Performance of List Intersection*. Proc. VLDB Endow., 2(1):838–849, Aug. 2009.

[143] Ungethüm, A., D. Habich, T. Karnagel, S. Haas, E. Mier, G. Fettweis und W. Lehner: *Overview on Hardware Optimizations for Database Engines*. In: Mitschang, B., D. Nicklas, F. Leymann, H. Schöning, M. Herschel, J. Teubner, T. Härder, O. Kopp und M. Wieland (Hrsg.): *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, S. 383–402. Gesellschaft für Informatik, Bonn, 2017.

[144] Vaidya, P. und J. J. Lee: *Characterization of TPC-H Queries for a Column-oriented Database on a Dual-Core AMD Athlon Processor*. In: *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM'08, S. 1411–1412, 2008.

[145] Vaisman, A. und E. Zimányi: *Data Warehouse Systems: Design and Implementation*. Springer Verlag, 2014.

[146] Vakali, A. und G. Pallis: *Content Delivery Networks: Status and Trends*. IEEE Internet Computing, 7(6):68–74, Nov. 2003.

[147] Valduriez, P. und G. Gardarin: *Join and Semijoin Algorithms for a Multiprocessor Database Machine*. ACM Trans. Database Syst., 9(1):133–161, März 1984.

[148] Veldhuizen, T. L.: *Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm*. In: *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014.*, S. 96–106, 2014.

[149] Viglas, S. D., J. F. Naughton und J. Burger: *Maximizing the Output Rate of Multi-way Join Queries over Streaming Information Sources*. In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB'03, S. 285–296. VLDB Endowment, 2003.

[150] VNI, C.: *Cisco Visual Networking Index: Forecast and Methodology, 2016–2021*, Juni 2017.

[151] Völp, M., S. Klüppelholz, J. Castrillon, H. Härtig, N. Asmussen, U. Assmann, F. Baader, C. Baier, G. Fettweis, J. Fröhlich, A. Goens, S. Haas, D. Habich, M. Hasler, I. Huismann, T. Karnagel, S. Karol, W. Lehner, L. Leuschner, M. Lieber, S. Ling, S. Märcker, J. Mey, W. Nagel, B. Nöthen, R. Peñaloza, M. Raitza, J. Stiller, A. Ungethüm und A. Voigt: *The Orchestration Stack: The Impossible Task of Designing Software for Unknown Future Post-CMOS Hardware*. In: *Proceedings of the 1st International Workshop on Post-Moore's Era Supercomputing (PMES), Co-located with The International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*, Salt Lake City, USA, Nov. 2016.

[152] Wang, L., J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li und B. Qiu: *BigDataBench: a Big Data Benchmark Suite from Internet Services*. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, S. 488–499, Feb. 2014.

[153] Waterman, A., Y. Lee, D. A. Patterson und K. Asanović: *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Techn. Ber. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, Mai 2014.

[154] Wolf, W., A. A. Jerraya und G. Martin: *Multiprocessor System-on-Chip (MPSoC) Technology*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 27(10):1701–1713, Okt. 2008.

[155] Woods, L., Z. István und G. Alonso: *Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading*. Proc. VLDB Endow., 7(11):963–974, Juli 2014.

[156] Wu, D., F. Zhang, N. Ao, F. Wang, X. Liu und G. Wang: *A Batched GPU Algorithm for Set Intersection*. In: *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, S. 752–756. IEEE, 2009.

[157] Wu, K., E. J. Otoo und A. Shoshani: *Compressing Bitmap Indexes for Faster Search Operations*. In: *Proceedings 14th International Conference on Scientific and Statistical Database Management*, S. 99–108, 2002.

[158] Wu, K., E. J. Otoo und A. Shoshani: *An Efficient Compression Scheme for Bitmap Indices*. Techn. Ber., ACM Transactions on Database Systems, 2004.

[159] Wu, L., R. J. Barker, M. A. Kim und K. A. Ross: *Navigating Big Data with High-throughput, Energy-efficient Data Partitioning*. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA'13, S. 249–260, 2013.

[160] Wu, L., A. Lottarini, T. K. Paine, M. A. Kim und K. A. Ross: *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*, ASPLOS'14, S. 255–268. ACM, 2014.

[161] Wulf, W. A. und S. P. Harbison: *Reflections in a Pool of Processors: An Experience Report on C.mmp/Hydra*. Techn. Ber., Carnegie-Mellon University, Pittsburgh, PA, Feb. 1978.

[162] Yabuta, M., A. Nguyen, S. Kato, M. Edahiro und H. Kawashima: *Relational Joins on GPUs: A Closer Look*. IEEE Transactions on Parallel and Distributed Systems, PP(99):1–1, 2017.

[163] Yoshimi, M., Y. Oge und T. Yoshinaga: *Pipelined Parallel Join and Its FPGA-Based Acceleration*. ACM Trans. Reconfigurable Technol. Syst., 10(4):28:1–28:28, Dez. 2017.

[164] Zain, S., S. Sulong, N. M. M. Hashim und Z. I. Rizman: *Environment for agent-based model in mobile database transaction: A review*. ARPN Journal of Engineering and Applied Sciences, 10:9496–9507, 2015.

[165] Zaki, M. J.: *Scalable Algorithms for Association Mining*. IEEE Transactions on Knowledge and Data Engineering, 12(3):372–390, Mai 2000.

[166] Zermelo, E.: *Georg Cantor: Gesammelte Abhandlungen mathematischen und philosophischen Inhalts*. Springer Verlag, 1932.

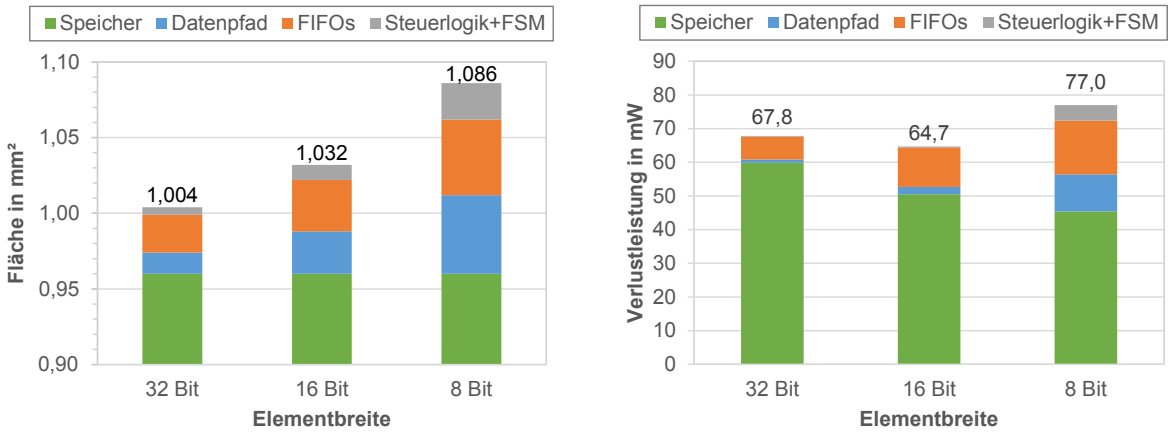
[167] Zhou, J. und K. A. Ross: *Implementing Database Operations Using SIMD Instructions*. In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD'02, S. 145–156, 2002.

[168] Zhou, S.: *Development and Analysis of ASIC Implementations for Data Intensive Algorithms on Compressed Bitmap Indices*. Studienarbeit, TU Dresden, Okt. 2016.

[169] Zhou, S.: *Analysis and Implementation of a Hardware Scheduler for Data Intensive Algorithms on Compressed Bitmap Indices*. Masterarbeit, TU Dresden, Sept. 2017.

Anhang

A Weitere Abbildungen und Tabellen



(a) Fläche

(b) Verlustleistung

Abbildung A.1: Aufteilung von Flächenbedarf und Verlustleistung auf die einzelnen Logikkomponenten des DBA ASIC und den Speicher. Es wird die Datenbreite W_{Elem} variiert und erzeugt damit bei konstanter Datenbreite des Speicherinterfaces unterschiedliche Parallelitätsfaktoren.

Tabelle A.1: Titan3D: Messergebnisse der Datenbankoperatoren. Die Messungen erfolgten immer bei der angegeben max. Taktfrequenz f_{max} und bei einer Versorgungsspannung von $U_{DD} = 1{,}1\,\mathrm{V}$.

Operator	Max. Taktfreq. f_{max} in MHz	Durchsatz in 10^6 Elem./s RISC	Durchsatz in 10^6 Elem./s RISC+ISE	Leistung in mW RISC	Leistung in mW RISC+ISE	Energie in pJ/Elem. RISC	Energie in pJ/Elem. RISC+ISE
Intersektion	200	24,7	586,9	15,5	21,6	627,8	36,8
Vereinigung	200	23,3	380,7	16,6	33,7	713,8	88,5
Differenz	200	24,6	581,7	15,7	30,7	639,0	52,8
Merge-Sort	250	1,8	16,1	20,4	26,8	11 137,4	1663,9
Join	250	21,1	243,9	16,9	20,1	800,9	82,4
Aggregation	250	20,8	329,6	23,3	33,0	1120,2	100,1

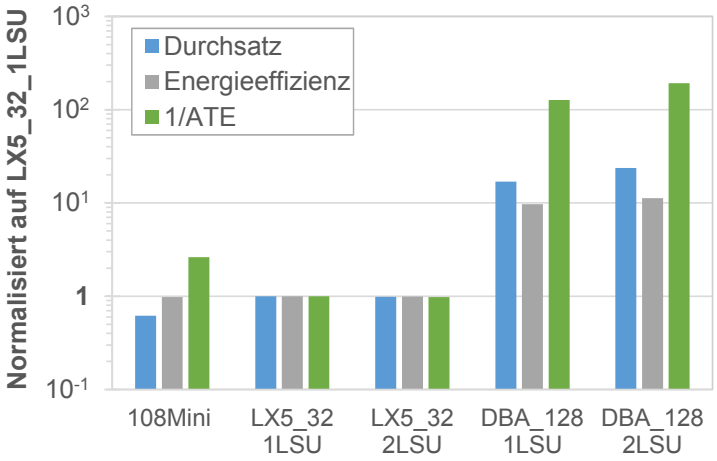


Abbildung A.2: Ergebnisse verschiedener Prozessorkonfigurationen für die Sorted-Set Intersektion jeweils normiert auf `LX5_32_1LSU`. Energieverbrauch und ATE-Produkt sind als reziproke Werte dargestellt (höher ist besser).

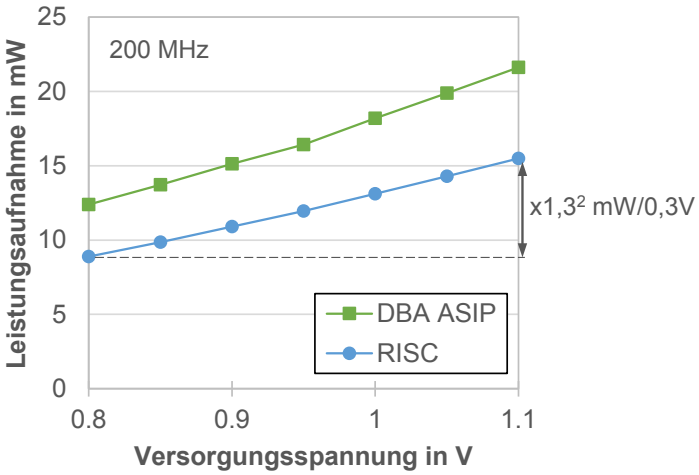


Abbildung A.3: Titan3D: Gesamtleistungsaufnahme der Sorted-Set Intersektion ohne (RISC) und mit (DBA ASIP) aktivierter Befehlssatzerweiterung bei skalierter Versorgungsspannung, Taktfrequenz: 200 MHz.

Tabelle A.2: Ergebnisse des DBA ASIC für die Sorted-Set Vereinigung und Differenz bei verschiedenen Datenbreiten, Selektivität: 50 %

Parallelitätsfaktor (W_{Elem})	$k = 4$ (32 Bit)	$k = 8$ (16 Bit)	$k = 16$ (8 Bit)
Max. Taktfrequenz f_{max} in MHz	440	360	320
Vereinigung			
Durchsatz in Elem./Takt	5,4	10,8	21,3
Durchsatz bei f_{max} in GBit/s	76,0	62,2	54,5
Verlustleistung bei f_{max} in mW	77,7	74,1	88,2
Logik	8,6	15,4	34,2
Speicher	69,1	58,7	54,0
Differenz			
Durchsatz in Elem./Takt	6,5	13,0	25,6
Durchsatz bei f_{max} in GBit/s	91,6	74,9	65,5
Verlustleistung bei f_{max} in mW	67,6	64,5	76,8
Logik	7,8	14,1	31,5
Speicher	59,8	50,4	45,3

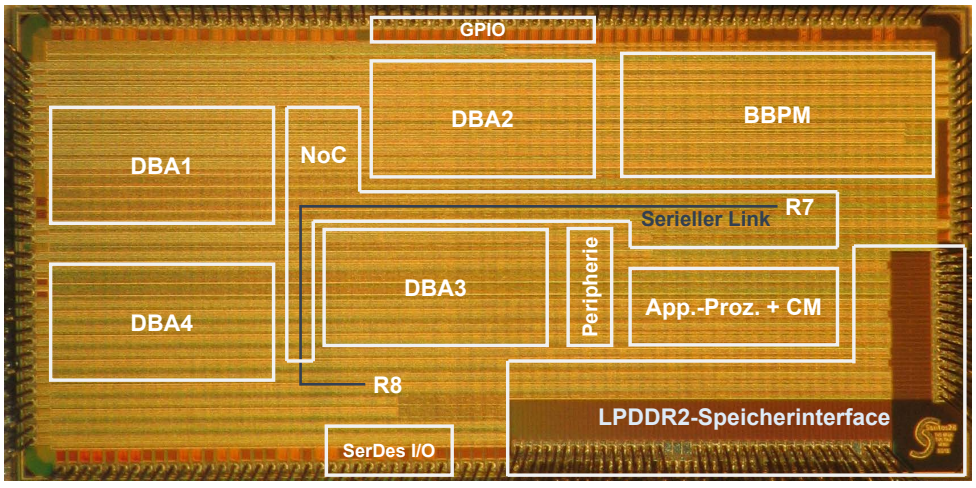


Abbildung A.4: Tomahawk4 MPSoC Chipfoto

Tabelle A.3: Sorted-Set Intersektion: Vergleich der DBA-Implementierungen mit anderen Prozessoren, Selektivität: 50 %, $W_{Elem} = 32\,\text{Bit}$

	DBAi 128_2LSU	DBAi 256_2LSU	DBAi 512_2LSU	DBA ASIC	ARC EM4-Mini	108Mini	Intel i7-920	Intel i7-920 + SSE4.2
Technologie in nm	65	65	65	65	65	65	45	45
Taktfrequenz in MHz	410	410	322	444	320	442	2670	2670
Durchsatz in 10^6 Elem./s	1203	2236	3070	2931	32,9	31,3	196	1100
Durchsatz in GBit/s	37,6	69,9	95,9	91,6	1,1	1,2	6,3	35,6
Gesamtfläche in mm^2	1,088	1,167	1,852	1,004	0,388	0,22	263	263
Verlustleistung in W	0,092	0,129	0,199	0,068	0,017	0,024	(1)130	(1)130
Energie in pJ/Bit	2,4	1,8	2,1	0,7	16,5	20,5	21 214	3782
ATE-Produkt in $mm^2\,mW/(10^6\,\text{Elem./s})^2$	$6{,}9 \cdot 10^{-5}$	$3{,}0 \cdot 10^{-5}$	$3{,}9 \cdot 10^{-5}$	$7{,}9 \cdot 10^{-6}$	$6{,}0 \cdot 10^{-3}$	$5{,}5 \cdot 10^{-3}$	$8{,}9 \cdot 10^{2}$	$2{,}8 \cdot 10^{1}$

(1)TDP entnommen aus [85].

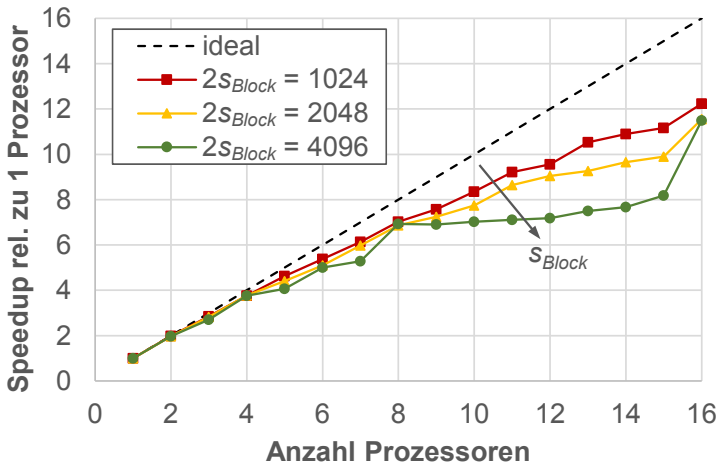


Abbildung A.5: Simulationsergebnisse (Speedup) des Merge-Sort mit Lastverteilung bei verschiedenen Größen der zu sortierenden Blöcke s_{Block} (Größe des lokalen Speichers), ohne ISE, $N = 65\,536$ Elemente

Tabelle A.4: Gewählte Standardwerte der Modellparameter

B_{DRAM}	0,8 Elem./Takt	
B_{NoC}	2,0 Elem./Takt	
Sorted-Set Algorithmen	ohne ISE	mit ISE
D_{Int}	0,166 Elem./Takt	2,3 Elem./Takt
D_{Diff}	0,166 Elem./Takt	2,3 Elem./Takt
D_{Verein}	0,166 Elem./Takt	1,53 Elem./Takt
r_{DB}	80 %	
$t_{Sched}(p = 1, N = 10\,000)$	3370 Takte	
Merge-Sort	ohne ISE	mit ISE
D_{MS}	0,0073 Elem./Takt	0,0640 Elem./Takt
$D_{MS,presort}$	0,1057 Elem./Takt	1,0211 Elem./Takt
Hash-Join	ohne ISE	mit ISE
D_{Hash}	0,0051 Elem./Takt	3,1 Elem./Takt
β_{Hist}	2,111	
D_{Hist}	0,361 76 Elem./Takt	
$\beta_{Hist-HT,Build}$	0,2449	
$D_{Hist-HT,Build}$	0,0125 Elem./Takt	
$\beta_{LL-HT,Build}$	0,1678	
$D_{LL-HT,Build}$	0,006 45 Elem./Takt	
$\beta_{Hist-HT,Probe}$	0,179 95	
$D_{Hist-HT,Probe}$	0,001 46 Elem./Takt	
$\beta_{LL-HT,Probe}$	0,015	
$D_{LL-HT,Probe}$	0,001 87 Elem./Takt	
m	1024 Buckets	
n_{Burst}	63	
r_{DB}	68 %	

B Weitere Quellcodes

```
int difference(int* A, int l_a,
               int* B, int l_b,
               int* C)){
    int pos_a = 0, pos_b = 0, pos_c = 0;
    while(pos_a < l_a && pos_b < l_b){
        if(A[pos_a] == B[pos_b]){
            pos_a++;
            pos_b++;
        }
        else if(A[pos_a] < B[pos_b])
            C[pos_c++] = A[pos_a++];
        else
            pos_b++;
    }

    //Nachverarbeitung
    while(pos_a < l_a))
        C[pos_c++] = A[pos_a++];

    return pos_c;
}
```

Abbildung B.1: Quellcode der Sorted-Set Differenz

```
void MergeSort(int *A, int len,
               int *aux){
    int mw;    //Länge der jew. Teilfolge
    int mid;   //Pos. des mittleren Elem.

    for(mw = 1; mw < len; mw = mw*2){
        for(mid = len-mw-1; mid >= 0; mid -= mw*2){
            merge(&A[mid-mw+1],  //zu sort. Teilliste
                  mid+mw,        //Länge der Teilliste
                  aux);          //Hilfsarray
        }
    }
}
```

Abbildung B.2: Quellcode des Merge-Sort. Der Code der Funktion `merge` ist in Abbildung 2.4a dargestellt.

```
int union(int* A, int l_a,
          int* B, int l_b,
          int* C)){
    int pos_a = 0, pos_b = 0, pos_c = 0;
    while(pos_a < l_a && pos_b < l_b){
        if(A[pos_a] == B[pos_b]){
            C[pos_c++] = A[pos_a];
            pos_a++;
            pos_b++;
        }
        else if(A[pos_a] < B[pos_b])
            C[pos_c++] = A[pos_a++];
        else
            C[pos_c++] = B[pos_b++];
    }

    //Nachverarbeitung
    if(pos_a == l_a)
        while(pos_b < l_b)
            C[pos_c++] = B[pos_b++];
    else
        while(pos_a < l_a))
            C[pos_c++] = A[pos_a++];

    return pos_c;
}
```

Abbildung B.3: Quellcode der Sorted-Set Vereinigung

C Weitere Herleitungen

C.1 Merge-Sort mit Datenparallelität

Anzahl ausgeführter Merge-Vorgänge

In der Abbildung 2.4b ist das Verschmelzen der jeweiligen Teilfolgen einer zu sortierenden Menge mit $N = 8$ Elementen zu sehen. Die Anzahl der Merge-Vorgänge lässt sich nun pro Ebene ableiten. In der ersten Ebene existieren $\frac{N}{2} = 4$, in der zweiten Ebene $\frac{N}{4} = 2$ und in der dritten Ebene $\frac{N}{8} = 1$ Merge-Vorgänge. Allgemein gilt dann für die Anzahl n_{Merge} der insgesamt ausgeführten Merge-Vorgänge, um N Elemente zu sortieren:

$$\begin{aligned} n_{Merge}(N) &= \frac{N}{2} + \frac{N}{4} + \frac{N}{8} + ... + \frac{N}{2^{\log N}} \\ &= N \sum_{i=1}^{\log N} \frac{1}{2^i} \end{aligned} \tag{C.1}$$

Bei einer k-fachen Datenparallelität existieren nur $\log N - \log k$ Ebenen (für den skalaren Fall ist $k = 1$). Außerdem wird immer gleichzeitig auf Datenblöcken der Größe k gearbeitet. Gleichung C.1 lässt damit erweitern zu

$$n_{Merge}(N, k) = \frac{N}{k} \sum_{i=1}^{\log N - \log k} \frac{1}{2^i}. \tag{C.2}$$

Durch Anwenden der endlichen geometrischen Reihe

$$\sum_{i=0}^{n} x^i = 1 + x + x^2 + ... + x^n = \frac{1 - x^{n+1}}{1 - x}$$

ergibt sich mit den Substitutionen $n = \log N - \log k$ und $x = \frac{1}{2}$

$$\begin{aligned} n_{Merge}(N, k) &= \frac{N}{k} \left(\frac{1 - \left(\frac{1}{2}\right)^{(\log N - \log k + 1)}}{1 - \frac{1}{2}} - 1 \right) \\ &= \frac{N}{k} \left(1 - \frac{2}{2^{\log \frac{N}{k} + 1}} \right) \\ &= \frac{N}{k} \left(1 - \frac{k}{N} \right) \\ &= \frac{N}{k} - 1 \end{aligned} \tag{C.3}$$

Die Anzahl der Merge-Vorgänge wächst demnach linear mit der Größe der Menge und skaliert mit dem Parallelitätsfaktor k. Das Ergebnis gilt auch, wenn N keine Zweierpotenz ist.

Anzahl der Vergleichsschritte

Die Vergleiche beim Verschmelzen zweier Teilfolgen entsprechen den Vergleichen beim Bestimmen der Sorted-Set Vereinigung. Für jede Ebene des Merge-Sort kann nun die Anzahl der Vergleichsschritte berechnet werden. Ein Vergleichsschritt bezeichnet den Vergleich von zwei Vektoren der Länge k. Ein Vergleichsschritt benötigt deshalb insgesamt k^2 Vergleiche. Für $k = 1$ entspricht ein Vergleichsschritt einem einzigen Vergleich zweier Elemente. Ist N keine Zweierpotenz, sind die jeweiligen zu verschmelzenden Teilfolgen nicht gleich lang. In diesem Fall zeigen die Gleichungen aber nur geringe Abweichungen zum tatsächlichen Ergebnis.

Minimale Anzahl Vergleichsschritte Wie in Gleichung 2.3 gezeigt wurde, müssen bei der Sorted-Set Vereinigung mindestens $\min(|A|, |B|)$ Vergleiche durchgeführt werden. Beim Merge-Sort entsprechen nun $|A|$ und $|B|$ den Längen der zu verschmelzenden Teilfolgen. Es folgt die Betrachtung am Beispiel $N = 8$. In der ersten Ebene ($|A| = |B| = 1$) sind mindestens $\min(1, 1) \cdot \frac{N}{2} = 4$ Vergleiche, in der zweiten Ebene ($|A| = |B| = 2$) mindestens $\min(2, 2) \cdot \frac{N}{4} = 4$ Vergleiche und in der dritten Ebene ($|A| = |B| = 4$) mindestens $\min(4, 4) \cdot \frac{N}{8} = 4$ Vergleiche durchzuführen. Man erkennt, dass in jeder Ebene mindestens $\frac{N}{2} = 4$ Vergleiche auszuführen sind. Mit insgesamt $\log N$ Ebenen, ergibt sich die minimale Anzahl von Vergleichen mit $\frac{N}{2} \log N$. Für eine k-fache Datenparallelität sind pro Ebene nur noch $\frac{N}{2k}$ Vergleichsschritte notwendig. Außerdem entfallen die ersten $\log k$ Ebenen durch die Vorsortierung. Damit gilt

$$n_{MS,Comp,Min}(N, k) = \frac{N}{2k} \log \frac{N}{k}. \tag{C.4}$$

Maximale Anzahl Vergleichsschritte Des Weiteren ist aus in Gleichung 2.3 bekannt, dass bei der Sorted-Set Vereinigung höchstens $|A| + |B| - 1$ Vergleiche durchgeführt werden. Es folgt wieder die Betrachtung am Beispiel $N = 8$. In der ersten Ebene ($|A| = |B| = 1$) sind höchstens $(1 + 1 - 1) \cdot \frac{N}{2} = 4$ Vergleiche, in der zweiten Ebene ($|A| = |B| = 2$) höchstens $(2 + 2 - 1) \cdot \frac{N}{4} = 6$ Vergleiche und in der dritten Ebene ($|A| = |B| = 4$) höchstens $(4 + 4 - 1) \cdot \frac{N}{8} = 7$ Vergleiche durchzuführen. Man erkennt, dass der Wert für $|A| + |B| - 1$ dem Wert im Nenner mit eins subtrahiert entspricht. Die maximale Anzahl der Vergleiche pro Ebene ist demnach

$$n^i_{MS,Comp,Max}(N) = N \frac{|A| + |B| - 1}{|A| + |B|}.$$

Mit $|A| + |B| = 2^i$ gilt

$$n^i_{MS,Comp,Max}(N) = N \left(1 - \frac{1}{2^i}\right), \tag{C.5}$$

wobei i die jeweilige Sortierebene angibt. Erweitert man Gleichung C.5 auf k-fache Datenparallelität, sind pro Ebene nur noch $\frac{N}{k}$ Vergleichsschritte notwendig. Zusätzlich entfallen wieder die ersten $\log k$ Ebenen durch die Vorsortierung. Damit berechnet sich die maxi-

male Anzahl der Vergleichsschritte beim iterativen Merge-Sort mit

$$\begin{aligned} n_{MS,Comp,Max}(N,k) &= \frac{N}{k}\sum_{i=1}^{\log N-\log k} n^i_{MS,Comp,Max} \\ &= \frac{N}{k}\sum_{i=1}^{\log N-\log k}\left(1-\frac{1}{2^i}\right) \\ &= \frac{N}{k}(\log N-\log k)-\underbrace{\frac{N}{k}\sum_{i=1}^{\log N-\log k}\frac{1}{2^i}}_{=n_{Merge}} \\ &= \frac{N}{k}\log\frac{N}{k}-\left(\frac{N}{k}-1\right) \\ &= \frac{N}{k}\left(\log\frac{N}{k}-1\right)+1. \end{aligned} \tag{C.6}$$

C.2 Speedup-Modell (3)

Es wird angenommen, dass der parallel abzuarbeitende Algorithmus eine lineare Abhängigkeit der Datentransferzeit von der Prozessoranzahl aufweist. Dies tritt z. B. bei dem Fall (1) des Merge-Sort auf. Das Speedup-Modell (2) wird deshalb auf folgende Weise erweitert:

$$t(p) = pt_{Trf}(p=1)+t_{Seq}+\frac{t_{Par}(p=1)}{p}. \tag{C.7}$$

Für den Speedup erhält man dann

$$\begin{aligned} S_{(3)}(p) &= \frac{t(p=1)}{t(p)} \\ &= \frac{p(1+\alpha)(1+\beta)}{p^2(1+\beta)+\alpha\beta p+\alpha} \end{aligned} \tag{C.8}$$

mit

$$\alpha=\frac{t_{Proc}(p=1)}{t_{Trf}(p=1)},\quad \beta=\frac{t_{Seq}}{t_{Par}(p=1)}.$$

Mit steigender Prozessoranzahl konvergiert der Speedup gegen Null. Das Maximum des Speedups ergibt sich durch Berechnen der ersten Ableitung und Einsetzen des erhaltenen

Wertes p_{max} in $S_{(3)}$:

$$\begin{aligned}
\frac{\partial S_{(3)}}{\partial p} &= \frac{(1+\alpha)(1+\beta)(p^2(1+\beta)+\alpha)}{(p^2(1+\beta)+\alpha\beta p+\alpha)^2} \\
\frac{\partial S_{(3)}}{\partial p} &\stackrel{!}{=} 0 \\
\rightarrow p_{max} &= \sqrt{\frac{\alpha}{1+\beta}} \qquad \text{(C.9)} \\
S_{(3),max} &= S_{(3)}(p_{max}) \\
&= \frac{\sqrt{\alpha}(1+\alpha)(1+\beta)}{2\alpha\sqrt{1+\beta}+\alpha\beta\sqrt{\alpha}} \qquad \text{(C.10)}
\end{aligned}$$

C.3 Speedup-Modell (4)

Es wird angenommen, dass der parallel abzuarbeitende Algorithmus eine lineare Abhängigkeit mit dem sequentiellen Anteil der Verarbeitungszeit von der Prozessoranzahl aufweist. Beispielsweise ist dies der Fall für den Implementierungsansatz (2) der Sorted-Set Algorithmen. Das Speedup-Modell (2) wird deshalb auf folgende Weise erweitert:

$$t(p) = t_{Trf}(p=1) + pt_{Seq} + \frac{t_{Par}(p=1)}{p}. \qquad \text{(C.11)}$$

Für den Speedup erhält man dann

$$\begin{aligned}
S_{(4)}(p) &= \frac{t(p=1)}{t(p)} \\
&= \frac{p(1+\alpha)(1+\beta)}{\alpha\beta p^2+p(1+\beta)+\alpha} \qquad \text{(C.12)}
\end{aligned}$$

mit

$$\alpha = \frac{t_{Proc}(p=1)}{t_{Trf}}, \quad \beta = \frac{t_{Seq}(p=1)}{t_{Par}(p=1)}.$$

Mit steigender Prozessoranzahl konvergiert der Speedup gegen Null, da in Gleichung C.12 der Nenner schneller steigt als der Zähler. Der Speedup durchläuft aber ein Maximum, das sich durch Berechnen der ersten Ableitung und Einsetzen des erhaltenen Wertes p_{max} in $S_{(4)}$ ermitteln lässt:

$$\begin{aligned}
\frac{\partial S_{(4)}}{\partial p} &= \frac{(1+\alpha)(1+\beta)(\alpha-p^2\alpha\beta)}{(\alpha\beta p^2+(1+\beta)p+\alpha)^2} \\
\frac{\partial S_{(4)}}{\partial p} &\stackrel{!}{=} 0 \\
\rightarrow p_{max} &= \frac{1}{\sqrt{\beta}} \qquad \text{(C.13)} \\
S_{(4),max} &= S_{(4)}(p_{max}) \\
&= \frac{(1+\alpha)(1+\beta)}{1+\beta+2\alpha\sqrt{\beta}} \qquad \text{(C.14)}
\end{aligned}$$

C.4 Speedup des Hash-Joins

Die Verarbeitungszeit des Hash-Joins enthält sowohl parallele als auch sequentiell abzuarbeitende Anteile. Der Speedup berechnet sich deshalb nach Modell (2) in Gleichung 6.5. Zur Vereinfachung der Gleichungen wird $N = |R| = |S|$ angenommen. Des Weiteren erzeugen reale Datensätze selten die maximale Kardinalität der Ergebnisrelation mit $|T| = N^2$. Eine Reduzierung auf $|T| = \frac{1}{4}N$ ist deshalb eine sinnvolle Annahme. Die zugehörigen Parameter ergeben sich nun aus der Gesamtlaufzeit in Gleichung 6.22 für den `Hist-HT` Hash-Join:

$$\alpha = \frac{B_{DRAM}}{2(1 - r_{DB}) + 0{,}25} \left(\frac{3}{D_{Hash}} + \frac{1 + \beta_{Hist}}{2D_{Hist}} + \frac{1 + \beta_{Build}}{2D_{Build}} + \frac{N(1 + \beta_{Probe})}{4D_{Probe}m} \right), \tag{C.15}$$

$$\beta = \frac{\frac{3}{D_{Hash}} + \frac{\beta_{Hist}}{2D_{Hist}} + \frac{\beta_{Build}}{2D_{Build}} + \frac{N\beta_{Probe}}{4D_{Probe}m}}{\frac{1}{2D_{Hist}} + \frac{1}{2D_{Build}} + \frac{N}{4D_{Probe}m}}. \tag{C.16}$$