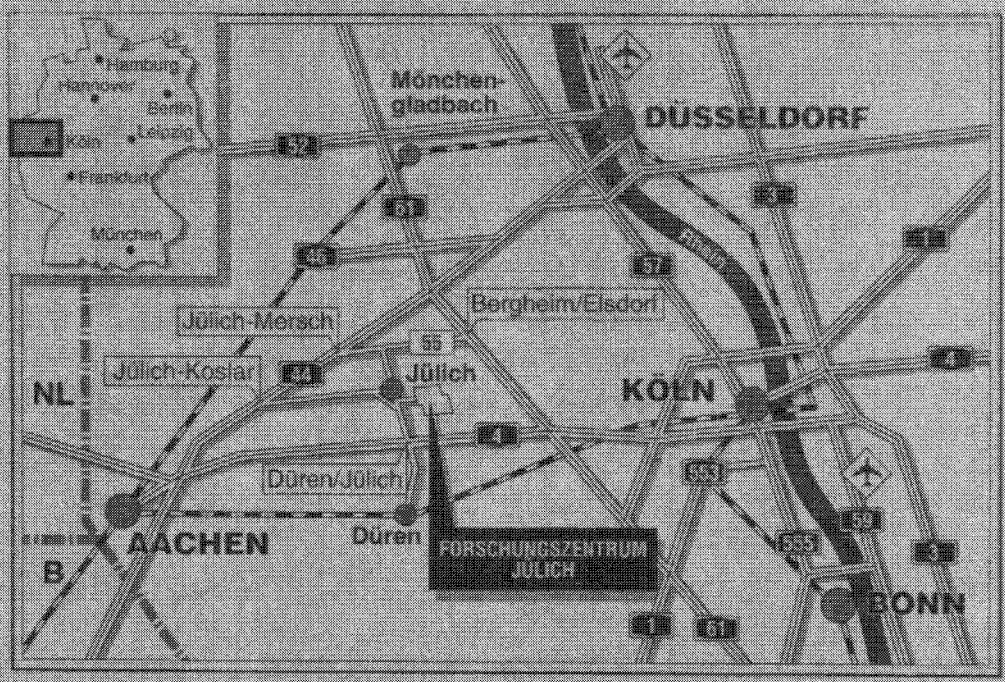
FORSCHUNGSZENTRUM JÜLICH GmbH
KFA

Zentralinstitut für Angewandte Mathematik

Performance-Analyse des DEC Alpha-Chip: Moderne Compiler-Techniken zur Programmoptimierung

Andreas Krumme



Berichte des Forschungszentrums Jülich ; 2912
ISSN 0944-2952
Zentralinstitut für Angewandte Mathematik Jül-2912

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek
D-52425 Jülich · Bundesrepublik Deutschland
Telefon: 02461/61-6102 · Telefax: 02461/61-6103 · Telex: 833556-70 kfa d

Performance-Analyse des DEC Alpha-Chip: Moderne Compiler-Techniken zur Programmoptimierung

Andreas Krumme

Abstract

The DEC Alpha-Chip dealt with is a RISC-processor which leads to a peak-performance of about 150 MFLOPS because of its high clock speed and its internal parallel pipelined structure. Using algorithms with a distinctive locality of reference high performance can be achieved by keeping reused data in fast memory hierarchies. To exploit this means of program optimization, compilers have to respect certain memory characteristics. Unfortunately, the compiler used does not comply with these requirements. This thesis analyzes serveral optimization options of the DEC FORTRAN compiler. Different ways of recognizing reused data in nested loops and different ways of optimizing the source code by appropriate program transformations are explained referring to examples written in FORTRAN code.

Zusammenfassung

Der untersuchte DEC Alpha-Chip ist ein RISC-Prozessor, der durch die hohe Taktfrequenz und die interne parallele Pipeline-Struktur eine theoretische Rechenleistung von bis zu 150 MFLOPS erreicht. Bei Algorithmen mit ausgeprägtem lokalen Verhalten können besonders dann hohe Performance-Werte erzielt werden, wenn häufig wiederverwendete Daten in den schnellen Cache-Speichern gehalten werden. Um diese Möglichkeiten der Programmoptimierung auszunutzen, muß ein Compiler die Merkmale des Speichersystems berücksichtigen, was aber von dem zur Verfügung stehenden Compiler nur in eingeschränktem Maß berücksichtigt wird. In dieser Arbeit wird, nach der Beschreibung des DEC Alpha-Chip, auf die Optimierungsoptionen des benutzten DEC FORTRAN-Compilers eingegangen. Anhand von Beispielen in FORTRAN-Code wird dann erläutert, wie Datenwiederverwendung in Schleifen ermittelt und der Source-Code durch die Anwendung geeigneter Programmtransformationen optimiert werden kann.

Inhalt

Abbildungsverzeichnis

Kapitel 1

Einleitung

Auf dem Weg zu immer höherer Rechenleistung ist die Entwicklung der sogenannten RISC-Architekturen (*Reduced Instruction Set Computer*) ein Meilenstein; durch die Einfachheit des Instruktionssatzes ist bei derartigen Rechnern ein besonders effektives Pipelining möglich. Nach frühen Erfolgen in den 60er Jahren (CDC 6600) ist der kommerzielle Durchbruch dieses Architektur-Prinzips erst Mitte der 80er Jahre, nach Bereitstellung moderner Fertigungstechniken und der daraus resultierenden Massenproduktion von Chip-Komponenten, gelungen [28].

Durch den Einbau von RISC-Prozessoren in massiv-parallelen Rechnern haben die bis dahin vorherrschenden Vektorrechner eine ernsthafte Konkurrenz erhalten, die sowohl durch die theoretisch erreichbare Leistungsfähigkeit als auch durch das günstige Preis/Leistungsverhältnis zu hohen Erwartungen geführt hat [17]. Ihren Einsatz finden massiv-parallele Rechner bevorzugt auf dem Gebiet des technisch-wissenschaftlichen Rechnens, bei dem zur Lösung von komplexen Aufgabenstellungen sehr viele Rechenoperationen ausgeführt werden müssen. Erfahrungen aus dem Betrieb dieser Rechner haben jedoch gezeigt, daß bei der Bearbeitung von realen Anwendungen die theoretisch erreichbare Peak-Performance in beachtlichem Maß unterschritten wird [18].

Der Grund hierfür liegt in der begrenzten Leistungsfähigkeit der Datenkommunikation sowohl zwischen den einzelnen Prozessoren als auch zwischen dem Prozessor und zugehörigem Hauptspeicher. Eine erhöhte Taktrate des Prozessors verlangt eine höhere Datenbandbreite. Die Zugriffszeit der Speicherbausteine ist jedoch relativ zur Prozessortaktzeit gestiegen, was zu einem stärkeren Ungleichgewicht zwischen Verarbeitungs- und Datentransferzeit geführt hat.

Um das Problem der begrenzten Datenbandbreite anzugehen, sind Systementwickler dazu übergegangen, schnelle Cache-Speicher zwischen Prozessor und Hauptspeicher anzuordnen. Obwohl Cache-Speicher in der Regel die effektive Zugriffszeit auf Daten wirkungsvoll verkürzen, zeigen Untersuchungen, daß wissenschaftliche Programme

den Cache-Speicher nur in sehr eingeschränktem Maß nutzen [21]. Der Grund hierfür sind die sehr großen Datenfelder, die für die Berechnungen benutzt werden. Obwohl Daten wiederverwendet werden, sind sie bei erneuter Referenz bereits von anderen Daten überschrieben.

Um eine akzeptable Leistung bei der Ausführung wissenschaftlicher Programme zu erzielen, werden Umstrukturierungen des Programms durchgeführt, die den zeitlichen Abstand aufeinanderfolgender Referenzen desselben Feldelements reduzieren. Von dieser Technik wird insbesondere bei der Erstellung der BLAS-Bibliotheken (*Basic Linear Algebra Subprograms*) Gebrauch gemacht [12], die effiziente Programme für elementare Operationen der linearen Algebra umfassen. Die Optimierung dieser Programme auf die verschiedenen Zielrechner wird üblicherweise manuell von den Herstellern vorgenommen, und die resultierenden algorithmischen Implementationen unterscheiden sich deshalb je nach Architekturprinzip gravierend.

Da nicht für alle Probleme optimierte Unterprogramme verfügbar sind und die Optimierung von Hand sehr zeitaufwendig und fehleranfällig ist, bestehen Bestrebungen, den Optimierungsprozeß auf den Compiler zu übertragen [6, 36, 38]. Die Optimierung durch den Compiler geschieht auf zwei Ebenen: Auf der Instruktionsebene wird die Anordnung von Assembler-Befehlen optimiert, um den internen Parallelismus (*Instruction Level Parallelism, ILP*) effektiv auszunutzen. Optimierungen bezüglich einer effektiveren Ausnutzung der Datenlokalität werden dagegen auf Hochsprachenebene vorgenommen.

In dieser Arbeit wird das Zusammenspiel der drei Elemente Prozessor, Speichersystem und Compiler am Beispiel des DEC Alpha-Chip und des DEC FORTRAN-Compilers untersucht. Nach einer Einführung in die charakteristischen Merkmale von RISC-Prozessoren folgt in Kapitel 3 die Beschreibung des DEC Alpha-Chip. Auf dieser Grundlage werden im folgenden Kapitel die Optimierungsoptionen des DEC FORTRAN-Compilers vorgestellt und weitere Optimierungsmöglichkeiten ermittelt. Kapitel 5 beschäftigt sich mit der Programmoptimierung auf Hochsprachenebene, die auf eine effektivere Nutzung des Speichersystems abzielt. Abschließend wird am Beispiel der Matrixmultiplikation gezeigt, wie mit Hilfe dieser Optimierungen eine erhebliche Verbesserung der Performance erzielt werden kann.

Kapitel 2

Architekturmerkmale von RISC-Prozessoren

Dieses Kapitel beschreibt die wesentlichen Konzepte zur Leistungssteigerung von Prozessorsystemen, d.h. von Prozessoren und der Speicherumgebung. Zu Beginn werden Gründe angegeben, die zur Entwicklung von Rechnern mit verkürztem Instruktionssatz (*Reduced Instruction Set Computer, RISC*) geführt haben. Auf der Grundlage einer qualitativen Definition der Performance werden dann Komponenten beschrieben, die vor allem in RISC-Prozessorsystemen zu einem Performance-Gewinn führen.

2.1 Performance

RISC-Prozessoren wurden aus dem Bestreben entwickelt, die Ressourcen eines Prozessors bei der Abarbeitung von Programmen optimal auszuschöpfen. Grundlagen für die Entwicklung waren dabei Performance-Messungen an Betriebssystemen und Testmodulen, die von optimierenden Compilern übersetzt wurden. Analysen der erzeugten Maschinen-Codes ergaben, daß auch Compiler für Prozessoren mit komplexem Instruktionssatz (*Complex Instruction Set Computer, CISC*) häufig einfache Maschinenbefehle benutzen, da sich Befehlskonstrukte von Hochsprachen selten durch komplexe Instruktionen des Befehlssatzes ausdrücken lassen. Der Befehlssatz von RISC-Prozessoren enthält daher keine komplexen Operationen, sondern überläßt die maschinennahe Programmierung komplexer Befehle der Software [22].

Die Verwendung eines reduzierten Instruktionssatzes trägt noch nicht zur Performance-Verbesserung bei, zumal die Ersetzung von komplexen Operationen durch eine Folge einfacher Maschinenbefehle die Anzahl der notwendigen Instruktionen für ein Programm erhöht. Die beobachteten Performance-Verbesserungen

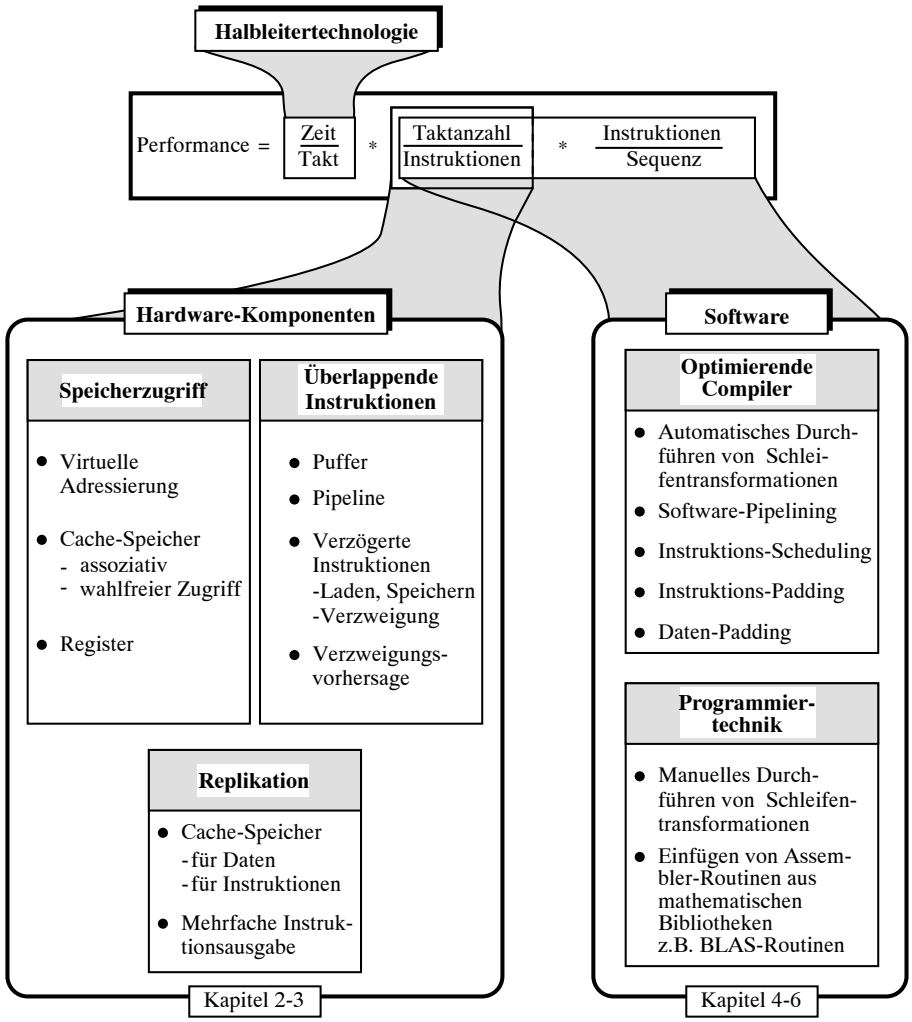


Abbildung 2.1: Definition der Performance und Zuordnung leistungssteigernder Konzepte

beim Einsatz von RISC-Prozessoren werden durch die bessere Unterstützung von leistungssteigernden Hardware-Komponenten erreicht. Ein quantitativer Vergleich, der die Überlegenheit eines RISC-Prozessorsystems bezüglich eines vergleichbaren CISC-Rechners belegt, ist in [4] angegeben.

Um die Randbedingungen, wie unterschiedliche Taktrate und Compiler-Effizienz bei der Leistungsbewertung zu berücksichtigen, kann nach [4, 22] die Performance eines Prozessorsystems wie folgt definiert werden:

$$Zeit\ pro\ Instruktionssequenz = \frac{Zeit}{Takt} \cdot \frac{Taktanzahl}{Instruktionen} \cdot \frac{Instruktionen}{Sequenz}$$

Jede Reduzierung der einzelnen Faktoren bedeutet einen geringeren Zeitaufwand

pro Sequenzberechnung und damit eine Erhöhung der Rechenleistung. Wird beispielsweise die Taktfrequenz um den Faktor 2 erhöht, ist der erste Term $\frac{Zeit}{Takt}$ halbiert, während die anderen Terme ihren Wert beibehalten. Die notwendige Zeit zur Berechnung der Sequenz ist damit auf die Hälfte reduziert. Der erste Term in Abbildung 2.1 ist abhängig von der Halbleitertechnologie und wird im Rahmen dieser Arbeit nicht weiter untersucht. Das Produkt $\frac{Taktanzahl}{Instruktionen} \cdot \frac{Instruktionen}{Sequenz}$ kann automatisch durch den Einsatz von optimierenden Compilern oder manuell durch Anwendung von Programmiertechniken in der jeweils verwendeten Hochsprache minimiert werden. Die Untersuchung von Programmierregeln zur optimalen Ausnutzung der Prozessorkomponenten ist zentraler Punkt der vorliegenden Arbeit und folgt in den Kapiteln 4 bis 6. Der mittlere Term ist auch abhängig von den einzelnen Hardware-Komponenten des Prozessorsystems, wie z.B. Pipeline und Cache-Speicher. Die Erläuterung dieser und anderer Komponenten in den folgenden Abschnitten dient als Grundlage für die Beschreibung des DEC Alpha-Chip AA-21064 in Kapitel 3 und für das Verständnis der darauf folgenden Kapitel über Compiler-Techniken zur Optimierung von Instruktionssequenzen.

Bei der Beschreibung der Hardware wird im folgenden auf die

- Aufgabe,
- Funktion und
- Performance

der Hardware-Komponenten eingegangen. Quantitative Untersuchungen zur Performance oder Dimensionierung verschiedener Implementationen von Cache-Speichern, Pipelines etc. werden nicht angestellt.

2.2 Speicherzugriff

Das Speicher-System wird in vielen Literaturstellen als Flaschenhals der Prozessorarchitektur bezeichnet, weil die Verarbeitungsgeschwindigkeit des Prozessors durch die begrenzte Bandbreite des Speichers limitiert wird [34]. Die charakterisierenden Merkmale des Speichers sind Zugriffszeit, Kapazität und Herstellungskosten, wobei die Forderung nach kürzeren Zugriffszeiten und größerer Kapazität im allgemeinen mit höheren Kosten verbunden ist. Bei der Implementation von Prozessorsystemen werden unterschiedliche Speichertypen verwendet, die nach ihrer Zugriffszeit verschiedenen *Speicherhierarchien* zugeordnet werden.

Eine für die Auswahl von Speichertypen wichtige Programm-Charakteristik ist die *Lokalität der Referenz*: In der Regel werden Daten und Instruktionen referenziert, auf die bereits in der gerade zurückliegenden Vergangenheit zugegriffen wurde [15, 34]. Diese Eigenschaft hat zur Folge, daß zur Laufzeit jeweils nur ein Teil der Programmdaten und Instruktionen benutzt wird, der in kleinen schnellen Speichern

ein- und ausgelagert werden kann. Im Idealfall befinden sich dort alle benutzten Daten und Instruktionen, so daß die Zugriffszeit dann der des schnellen Speichers entspricht.

2.2.1 Virtuelle Adressierung

Die Ausführung von Programmen, deren Adreßraum den Umfang des zur Verfügung stehenden physikalischen Speichers übersteigt, verlangt die Benutzung von *virtuellen Adressen.* Wie in Abbildung 2.2 dargestellt, wird vom Prozessor eine virtu-

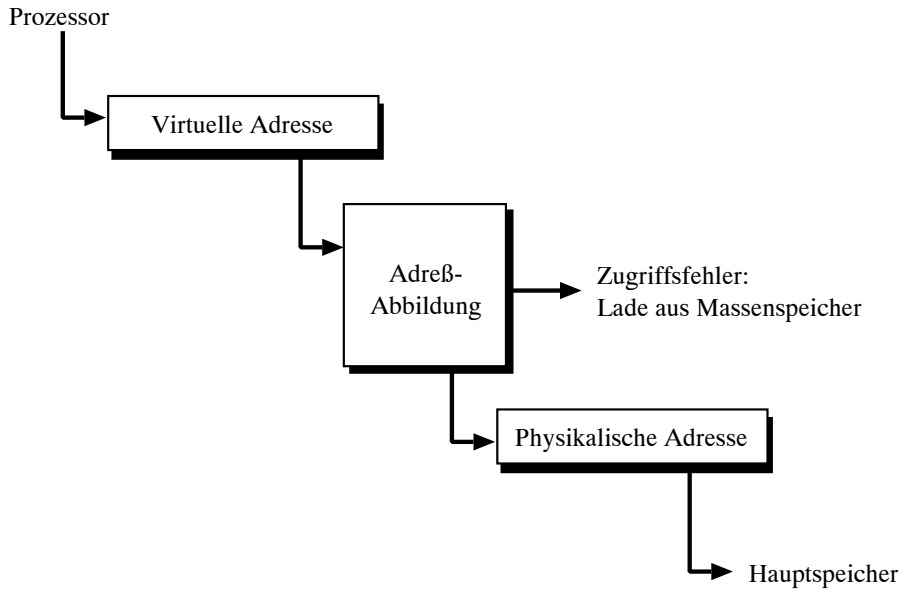


Abbildung 2.2: Abbildung von virtuellen in physikalische Adressen

elle Adresse generiert, die ein Abbildungsmechanimus in eine physikalische Adresse transformiert. In einer Tabelle im Hauptspeicher ist festgehalten, ob eine Zuordnung zwischen der virtuellen und der physikalischen Adresse besteht. Ist dies der Fall, so wird das entsprechende Datum aus dem Hauptspeicher gelesen. Ist kein Eintrag für die virtuelle Adresse gespeichert, so muß das Datum aus dem Massenspeicher in den Hauptspeicher geladen werden. Da aufgrund der oben erwähnten Lokalität anzunehmen ist, daß in der nächsten Zeit auch auf benachbarte Daten zugegriffen wird und ein Nachladen von Daten in den Hauptspeicher zeitaufwendig ist, wird neben dem gesuchten Datum auch eine bestimmte Anzahl benachbarter Daten in den Hauptspeicher geladen. Der Umfang dieser Datenmenge wird Seitengröße genannt und ist bei dem untersuchten DEC Alpha-Chip AA-21064 8 KByte.

Ein weiterer Vorteil der Unterteilung des virtuellen Adreßraums ist die Verminderung der zu speichernden Adreßpaare um den Faktor der Seitengröße. Gespeichert

wird nur noch das Adreßpaar der Seitenbasisadresse. Die einzelnen Maschinenworte innerhalb der Seite werden durch einen Versatzwert adressiert, der die Distanz zum Seitenanfang angibt. Eine zusätzliche Verminderung des Speicheraufwands kann durch eine Unterteilung der Seiten in Segmente vorgenommen werden. Eine Darstellung dieser Technik ist in [24, 34] zu finden.

Eine weitere Möglichkeit, die auch der untersuchte DEC Alpha-Chip AA-21064 vorsieht, ist die Verwendung eines assoziativen Cache-Speichers, der häufig verwendete Seitenadressen zwischenspeichert. Bevor also der große Datensatz der Adressen durchsucht werden muß, wird im *Translation Lookaside Buffer* (TLB) nachgesehen, ob die gesuchte Seitenadresse gespeichert ist. Ist dies der Fall, so wird die entsprechende physikalische Adresse ausgegeben und die Adreßabbildung ist nach Addition des Versatzwertes beendet.

Performance

Die effektive Zugriffszeit auf den Hauptspeicher berechnet sich nach der Formel:

$$Zugriffszeit \;=\; h \cdot t_h + (1-h) \cdot t_m$$

wobei t_h die Zugriffszeit auf den Hauptspeicher, t_m die Zugriffszeit auf den Massenspeicher und h die Trefferwahrscheinlichkeit eines Hauptspeicherzugriffs ist. In der Literatur sind für das Verhältnis $t_m : t_h$ Werte von 100 bis 1000 angegeben, die durch die Trägheit der mechanischen Bauteile des Massenspeichers bestimmt sind.

2.2.2 Cache-Speicher

Der Cache-Speicher ist das Bindeglied zwischen den Registern und dem langsameren Hauptspeicher. Seine Aufgabe ist es, die Latenzzeit beim Zugriff auf Daten des Hauptspeichers zu verkürzen, indem Hauptspeicherdaten, die das laufende Programm referenziert, im Cache-Speicher gehalten werden. Anhand der Abbildung 2.3 soll die Funktion eines assoziativen Cache-Speichers erläutert werden. Das Auffinden eines Datums geschieht in einem assoziativen Speicher allgemein durch Angabe des Inhaltes oder eines Teils davon (auch Schlüssel genannt) [11]. Ausgehend von dieser Definition ist in der Abbildung 2.3 die Adresse als Teil des Inhaltes einer Zeile des Cache-Speichers aufzufassen. Die Adressen und Daten sind in diesem Beispiel oktal kodiert, wobei ein gespeichertes Datenwort aus einer zweistelligen Oktalzahl (6 Bit) besteht und jeweils ein Feld einer Zeile belegt. So belegen beispielsweise die Daten mit dem Adreßteil 0375x die Speicherzellen mit den Adressen von 03750 bis 03757. Jeder Aufruf einer Hauptspeicheradresse wird dem Cache-Speicher zugeführt und initiiert dort eine Suche nach dem entsprechenden Datum.

Im dargestellten Cache-Speicher werden die Daten gesucht, die im Hauptspeicher unter der oktalen Adresse 14452 gespeichert sind. Dazu werden die ersten vier Stel-

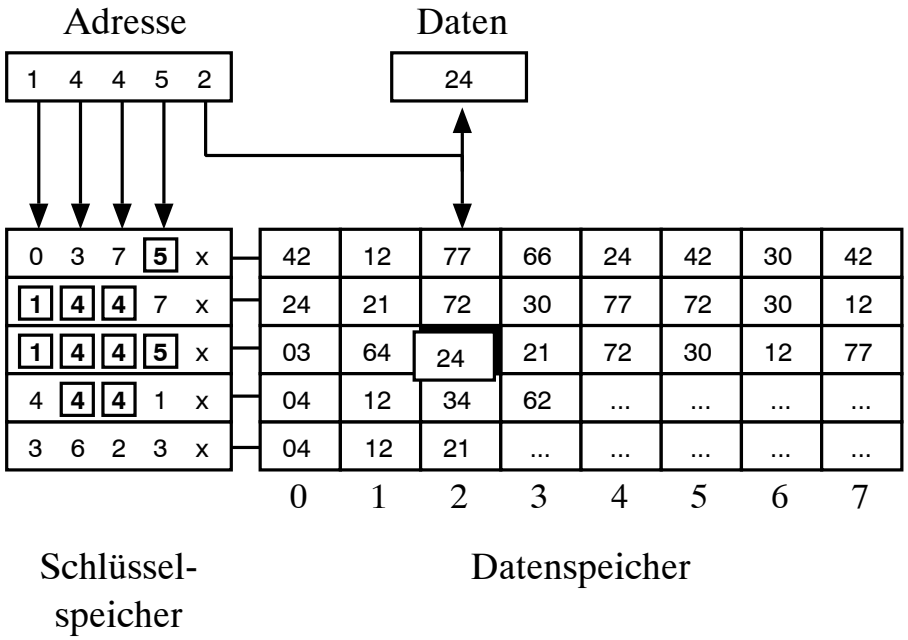


Abbildung 2.3: Aufbau und Adressierung eines assoziativen Cache-Speichers

len aller Adreßfelder im Schlüsselspeicher parallel auf Übereinstimmung geprüft, mit dem Ergebnis, daß in der dritten Zeile das gesuchte Datum zu finden ist. Die fünfte Oktalstelle der angegebenen Adresse mit dem Wert 2 identifiziert die Spalte 2, und das Datum wird ausgegeben. Unter der Suchadresse 1446x würden dementsprechend keine Daten im Cache-Speicher gefunden, und der Hauptspeicher müßte referenziert werden. Die oben beschriebene Überprüfung der Adreßfelder kann mit Hilfe einer großen Anzahl von Komparatoren geschehen, die bitweise parallel alle Adreßeinträge auf Übereinstimmung mit der Zieladresse überprüfen. Da eine solche Implementierung sehr aufwendig ist, werden häufig Cache-Speicher mit wahlfreiem Zugriff bevorzugt.

Wie aus Abbildung 2.4 hervorgeht, verfügen diese Cache-Speicher über dedizierte Adressen für die einzelnen Zeilen. In diesem Beispiel wird zur Vereinfachung eine acht Bit lange physikalische Adresse verwendet, wobei jedes adressierte Datum ebenfalls acht Bit lang ist. In einer Zeile des Datenspeichers werden bei einem Schreibvorgang immer vier Byte gespeichert, deren Adressen sich lediglich in den letzten zwei Bit-Stellen (Teil 1 der Adresse) unterscheiden. In welcher Zeile die Daten abgelegt oder gesucht werden, bestimmt Teil 2 der Adresse. Die drei Bit-Stellen 2 - 4 ermöglichen eine Adressierung von acht Zeilen. Zur Identifikation einer Datenspeicherzeile wird schließlich Teil 3 der Speicheradresse in die entsprechende Zeile des Schlüsselspeichers eingetragen. In Abbildung 2.4 sollen die Daten mit den Adressen 110 001 00 bis 110 001 11 aus dem Cache-Speicher gelesen werden. Dazu wird Teil 1 der Adresse abgeschnitten, und der Inhalt des Schlüssel-Speichers in Zeile 001 wird

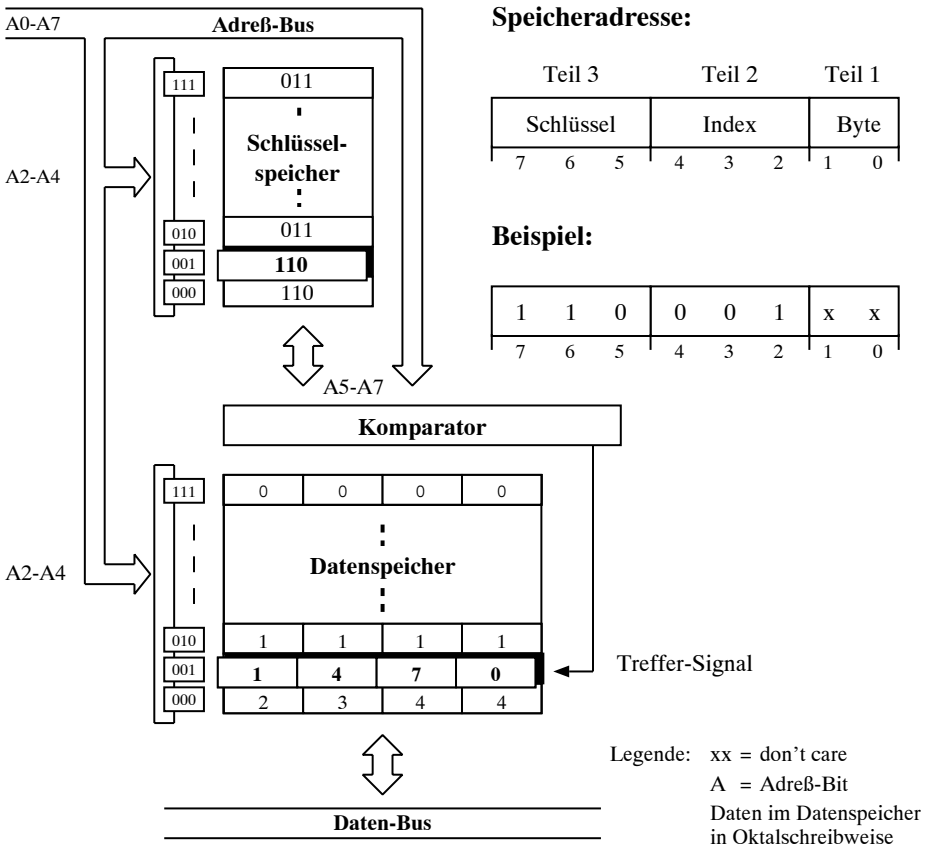


Abbildung 2.4: Laden eines Datums aus einem einfach-assoziativen Cache-Speicher mit wahlfreiem Zugriff

mit Teil 3 der Adresse verglichen. In diesem Falle stimmt der Eintrag mit den gegebenen Bit-Stellen überein, und der Komparator veranlaßt die Ausgabe des Inhaltes der Datenspeicherzeile 001 auf den Daten-Bus. Unter der Annahme, daß der Datenbus in einer Breite von acht Bit implementiert ist, werden die vier gespeicherten Daten nacheinander ausgegeben.

Die oben vorgenommene Adreßeinteilung teilt den zu speichernden Adreßraum in Seiten ein, deren Länge durch die Bit-Stellen von Teil 1 und Teil 2 der Adresse bestimmt ist. Ein Problem tritt auf, wenn die Daten mit den Adressen 11**1** 001 xx gespeichert werden sollen, da die gleiche Zeile wie bei der Adresse 11**0** 001 xx angesprochen werden würde. Dasselbe geschieht bei allen Adressen, die sich lediglich im Teil 3 unterscheiden. Damit ist ausgeschlossen, daß die gleichen Zeilen zweier Seiten gleichzeitig im Cache-Speicher vorhanden sein können. Die Cache-Speicherbelegung schwankt zwischen dem Extrem, daß alle Zeilen einer Seite gespeichert sind, und dem Extrem, daß alle Seiten mit jeweils einer Zeile im Cache-Speicher vertreten sind. Greift ein Programm immer abwechselnd auf Speicheradressen zu, die derselben Zeile zweier unterschiedlicher Seiten entsprechen, so muß die Cache-

Speicherkontrolleinheit die betreffende Zeile immer nachladen.

Dieses Problem kann umgangen werden, wenn man mehrere Paare von Daten- und Schlüsselspeichern implementiert. Man bezeichnet einen solchen Cache-Speicher mit N Paaren als N-fach-assoziativen Cache-Speicher. Für eine Index-Adresse sind dann, entsprechend der Anzahl von Paaren, mehrere Cache-Speicherzeilen vorgesehen, so daß gleiche Zeilen verschiedener Seiten gleichzeitig im Cache-Speicher residieren können. Zur Überprüfung, ob ein Datum im Speicher vorhanden ist, müssen die Schlüsselspeicher parallel auf einen entsprechenden Eintrag geprüft werden. Dies entspricht dem Vorgehen beim anfangs beschriebenen assoziativen Cache-Speicher.

Write-Through- und Write-Back-Cache-Speicher

Der Schreibzugriff und das Verhalten bei einem Lesefehler unterscheidet zwei häufig verwendete Entwurfsstrategien für Cache-Speicher.

- Bei der *Write-Through*-Strategie erfolgt jeder Schreibzugriff des Prozessors direkt auf den Hauptspeicher. Befindet sich die Adresse der neu zu beschreibenden Speicherzellen bereits im Schlüsselspeicher, so wird parallel zum Hauptspeicher auch der Cache-Speicher aktualisiert. Anderenfalls erübrigt sich die Aktualisierung. Beim Lesefehler lädt der Cache-Controller die Daten aus dem Hauptspeicher, aktualisiert den Schlüssel-Eintrag und bedient erst dann den Prozessor.

- Der Schreibzugriff erfolgt im Falle eines Treffers bei der *Write-Back*-Strategie nicht direkt auf den Speicher, sondern nur auf den Cache-Speicher. Parallel dazu verzeichnet der Cache-Controller in einem Markierungs-Bit, daß der Inhalt dieser Cache-Zeile verändert wurde. War das Markierungs-Bit schon gesetzt, werden nur die Daten gespeichert. Es entsteht für diesen Eintrag in beiden Fällen eine zeitweilige Inkonsistenz zwischen Haupt- und Cache-Speicher, die erst nach einem Lesefehler behoben wird. Tritt ein solcher Lesefehler auf, so muß im Gegensatz zur vorher besprochenen Strategie, vor dem Verwerfen der für die neuen Daten benötigten Cache-Zeile geprüft werden, ob diese mit dem Markierungs-Bit gekennzeichnet ist. In diesem Fall muß die oben erwähnte Inkonsistenz durch einen Transfer der betreffenden Zeile in den Hauptspeicher behoben werden. Erst danach kann die gewünschte neue Zeile geladen werden. Ist in der Implementation kein Markierungs-Bit vorgesehen, muß bei jedem Schreibvorgang die zu ersetzende Zeile in den Hauptspeicher geschrieben werden.

Verdrängungsmechanismus

Die zu ersetzenden Daten sind beim einfach-assoziativen Cache-Speicher durch die referenzierte Cache-Zeile eindeutig festgelegt. Demgegenüber kann beim N-fach-assoziativen Cache-Speicher ein Hauptspeicherdatum in einem von N Paaren untergebracht werden. Dies hat zur Folge, daß ein Entscheidungsmechanismus für die

Verdrängung einer der N Zeilen notwendig ist. Eine häufig benutzte Verdrängungsstrategie ist die Ersetzung derjenigen Cache-Speicherzeile, auf die am längsten nicht mehr zugegriffen wurde (Least-Recently-Used-Strategie). Die dazu notwendige Information muß für jede Zeile in separaten Bitstellen gespeichert werden.

Cache-Speicher-Performance

Die effektive Zugriffszeit auf den Cache-Speicher berechnet sich wie folgt:

$$Zugriffszeit = h \cdot t_c + (1 - h) \cdot t_h$$

Dabei ist h die Wahrscheinlichkeit, daß ein Lesetreffer auftritt. Dementsprechend ist $(1 - h)$ die Wahrscheinlichkeit eines Lesefehlers. Unter der Annahme, daß die Zugriffszeit auf den Hauptspeicher das Zehnfache der Zugriffszeit auf den Cache-Speicher beträgt (z.B. $t_c = 10$, $t_m = 100$), bedeutet eine Abnahme der Trefferwahrscheinlichkeit um 10 Prozent von 100% auf 90% ungefähr eine Verdoppelung der effektiven Zugriffszeit:

$$\frac{t_{Zugriff2}}{t_{Zugriff1}} = \frac{0.9 \cdot 10 + 0.1 \cdot 100}{10} = \frac{19}{10} \approx 2$$

Die Dimensionierung von Cache-Speichern beinhaltet die Wahl der Zeilenbreite in Byte, der Zeilenzahl und den Grad der Assoziativität. Das Produkt dieser drei Größen bestimmt die Kapazität des Speichers.

Die Randbedingung für die erreichbare Kapazität eines Cache-Speichers auf dem Prozessor (on-Chip) ist die Größe des zur Verfügung stehenden Raumes. Die Cache-Kapazität auf dem Prozessor beträgt bei dem DEC Alpha-Chip AA-21064 jeweils 8 KByte für Instruktionen und Daten, der Zugriff erfolgt innerhalb von drei Takten. Größere Cache-Speicher werden durch schnelle Speicherbausteine realisiert, die über Datenpfade mit dem Prozessor verbunden werden. Ihre Größe beträgt beispielsweise bei der Alpha-Workstation von Digital Equipment je nach Modell zwischen 512 KByte und 4 MByte. Die größere Kapazität geht jedoch zu Lasten der Zugriffszeit, die ca. acht Takte beträgt [10].

2.2.3 Register

Die schnellsten Speicher eines Prozessors sind die Register, deren kurze Zugriffszeit aus der verwendeten Hardware (schnelle Speicher mit wahlfreiem Zugriff) und aus der Implementation auf dem Prozessor (kurze Datenwege) resultiert.

Die Aufgabe der Register ist es, häufig benutzte Werte präsent zu halten. Die Zugriffszeit beträgt nur einen Takt und entspricht damit der Verarbeitungsgeschwindigkeit des Prozessors. Im Gegensatz zur Belegung des Haupt- und Cache-Speichers ist die Registerbelegung zur Übersetzungszeit festgelegt. In manchen RISC-Architek-

turen sind Assembler-Befehle vorgesehen, die eine Umgehung des Cache-Speichers bei der Belegung der Register ermöglichen. Dies ist bei der gegenwärtigen Alpha-AXP-Architektur, im Gegensatz zur i860-RISC-Architektur von Intel [27], nicht vorgesehen.

2.3 Überlappende Instruktionen

Ein anderer Ansatz, die Performance des Prozessorsystems zu erhöhen, ist die gleichzeitige Nutzung parallel angelegter Hardware-Komponenten. Voraussetzung hierfür sind getrennte Datenpfade und eine ausreichende Zahl von Registern, die die Operanden und Ergebnisse zwischenspeichern. Ausgehend von dem Einsatz von Puffern zur Zwischenspeicherung von Daten wird auf die Technik des Pipelining ausführlich eingegangen.

2.3.1 Puffer

Ein Hilfsmittel für die Realisierung von überlappenden Instruktionen sind Zwischenspeicher, auch Puffer genannt, die zwischen unabhängigen Funktionseinheiten angeordnet werden. In einfachen Systemen aktiviert das zentrale Leitwerk eine Funktionseinheit und wartet auf das Ergebnis. Die Anordnung eines Ein- und Ausgabe-Puffers zwischen den beiden Komponenten erlaubt es dem Leitwerk, während der aktiven Zeit der Funktionseinheit andere Aufgaben wahrzunehmen. Für die Funktionseinheit bestimmte Instruktionen werden im Eingabepuffer, Ergebnisse im Ausgabepuffer abgelegt. Es muß jedoch beachtet werden, daß durch den Einsatz von Puffern die ursprüngliche Reihenfolge der Instruktionen in manchen Fällen nicht eingehalten wird. Wird dies verlangt, müssen Barrieren vereinbart werden, die die Ausführung der Instruktionen in der ursprünglichen Reihenfolge garantieren.

2.3.2 Pipelining

Unter einer Pipeline versteht man die Struktur einer Funktionseinheit, die die Ausführung einer Instruktion in mehreren Schritten realisiert. Dies hat den Vorteil, daß eine Instruktion die Funktionseinheit nicht mehr für den kompletten Ausführungszeitraum blockiert, sondern nur einzelne Segmente für eine verkürzte Zeit. Die Initiierung des folgenden Befehls kann somit unmittelbar nach Ausführung der ersten Teiloperation erfolgen. Abbildung 2.5 verdeutlicht den Zeitvorteil beim Übergang von der sequentiellen zur Pipeline-Verarbeitung. Man erkennt, daß die Abarbeitungsrate nun nicht mehr von der Summe aller Instruktionstakte, sondern von der Initiierungsrate abhängt. Im Idealfall benötigen alle Stufen die gleiche Taktanzahl. Nach der Ladezeit der Pipeline, die der Anzahl der Stufen entspricht, wird nach

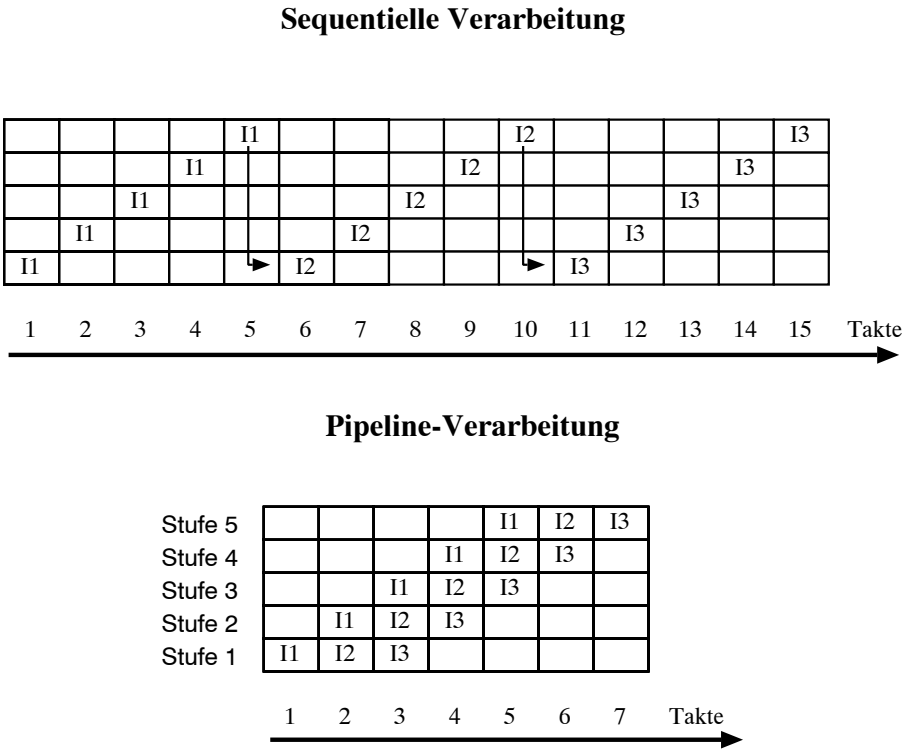


Abbildung 2.5: Zeitvergleich zwischen sequentieller und Pipeline-Verarbeitung von Instruktionen

jedem Takt eine Instruktion fertiggestellt. Aus der Sicht der Hardware besteht jede Stufe aus einem Logikteil und einem oder mehreren Zwischenspeichern. Während des Durchlaufs werden die Zwischenergebnisse mit jedem Zyklus durch die Logik in den Speicher der jeweils folgenden Stufe geschoben. Dauert eine Teilinstruktion länger, so ist sie bestimmend für die Ausgabegeschwindigkeit. Eine Verkürzung der übrigen Bearbeitungszeiten hätte keinen Vorteil zur Folge. Beim Entwurf einer Pipeline werden die Segmente der Instruktionen auf Übereinstimmung geprüft und ähnliche Instruktionen zu Gruppen zusammengefaßt. Für diese Gruppen, z.B. alle Fließpunktoperationen oder alle Integer-Operationen, werden dann Pipelines entwickelt. Ein Beispiel für die Übereinstimmung der Segmente zweier Instruktionen und die Realisierung einer gemischten Pipeline ist in Abbildung 2.6 angegeben. Es gibt Stufen, die von allen Instruktionen benutzt werden, und optionale Stufen, die wahlweise angesprochen werden. Weiterhin ist zu erkennen, daß einige Stufen von bestimmten Instruktionen mehrmals durchlaufen werden und daß eine Instruktion zwei Stufen gleichzeitig belegen kann. Die Nutzung der Pipeline von verschiedenen Instruktionen und das mehrmalige Durchlaufen einer einzelnen Stufe lassen eine beliebige Initiierung neuer Instruktionen nun nicht mehr zu, da einzelne Stufen nicht von mehreren Instruktionen gleichzeitig belegt werden dürfen.

Verdeutlichen läßt sich dies durch die Einführung einer Reservierungstabelle, in der

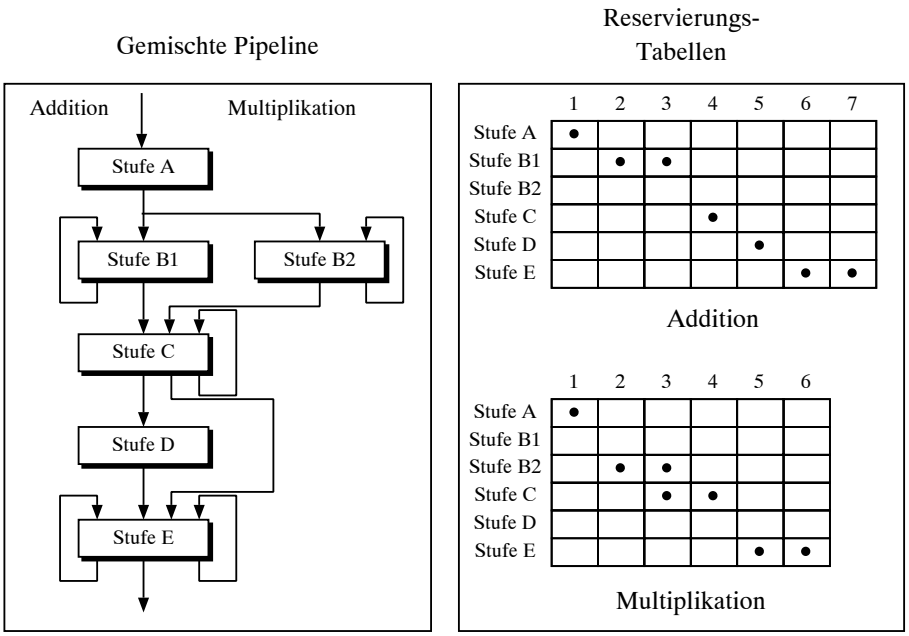


Abbildung 2.6: Gemischte Pipeline und zugehörige Reservierungstabellen

die Stufenbelegung einzelner Instruktionen eingetragen wird. Abbildung 2.6 stellt die Reservierungstabellen zweier Instruktionen dar, die in der nebenstehenden Pipeline abgearbeitet werden. Abbildung 2.7 verdeutlicht durch die graue Hinterlegung die Kollisionen, die bei unterschiedlichen Initiierungsraten der Additionsinstruktion A1 auftreten. Danach ist der früheste Zeitpunkt für eine erneute Initiierung Takt 3. Diese Abhängigkeit läßt sich anhand eines Kollisionsvektors beschreiben. In ihm sind die verbotenen Initiierungstakte durch eine 1 an entsprechender Stelle markiert. Eine 1 an der ersten Stelle bedeutet, daß die Instruktionen nicht gleichzeitig initiiert werden können. Für die Initiierung unterschiedlicher Instruktionen werden dann entsprechende Kollisionsvektoren ermittelt.

Die Überwachung der Pipeline geschieht durch die Pipeline-Kontrolleinheit, die nur dann eine neue Instruktion initiiert, wenn ein konfliktfreier Ablauf garantiert werden kann. Dies hat zur Voraussetzung, daß die nächste Instruktion in dekodierter Form vorliegt und die Ausgabe beim Konfliktfall angehalten werden kann. Eine Möglichkeit, die auch der DEC Alpha-Chip AA-21064 verwendet, ist die Einteilung der Pipeline in statische Dekodier- und dynamische Funktions-Stufen. Erstere können von der Kontrolleinheit angehalten und fortgesetzt werden, letztere sind taktgesteuert und nicht beeinflußbar.

Bei der Ausführung von Instruktionen in einer Pipeline existieren Ausnahmesituationen, die der besonderen Aufmerksamkeit bedürfen:

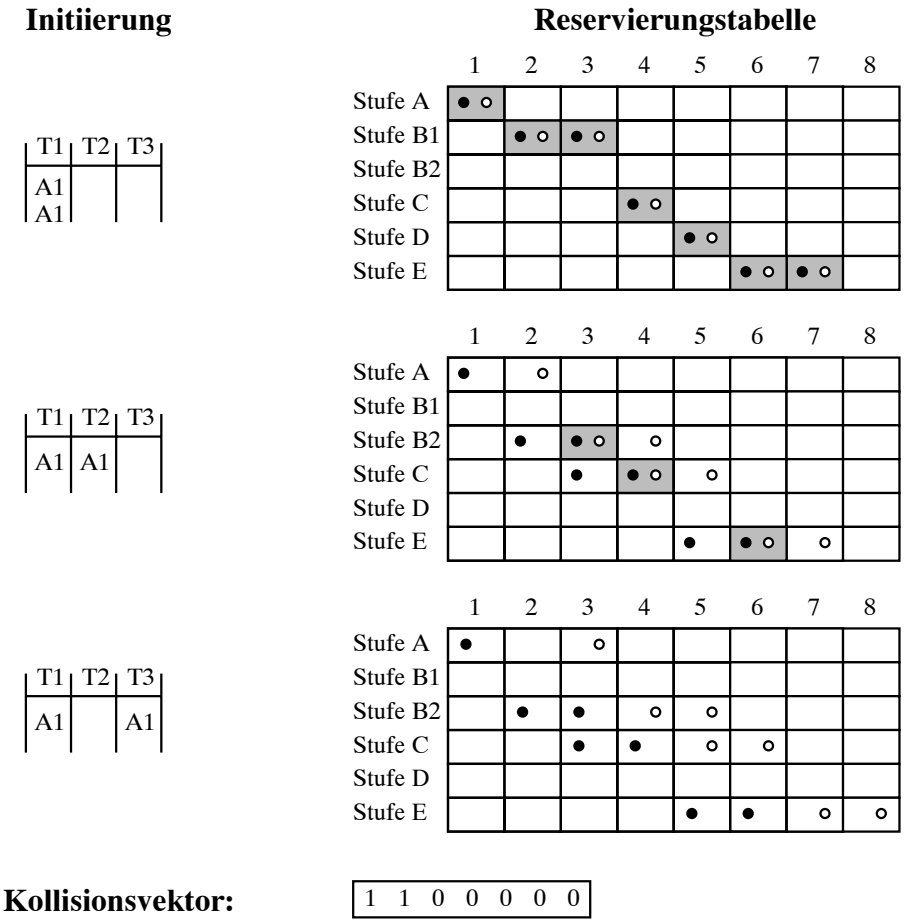


Abbildung 2.7: Verdeutlichung von Kollisionen anhand von Reservierungstabellen

- **Ressourcen-Konflikte**
  Ressourcen-Konflikte liegen vor, wenn zwei in der Ausführung befindliche Instruktionen gleichzeitig auf dieselbe Ressource zugreifen möchten. Dies kann eine Pipeline-Stufe, ein Register oder ein externer Speicher sein. Die Vermeidung von Ressourcen-Konflikten ist Aufgabe der Pipeline-Kontrolleinheit.

- **Datenabhängigkeiten**
  Benutzt eine Instruktion das Ergebnis der vorhergehenden als Eingabe, so kann sie erst dann ausgeführt werden, wenn das Ergebnis vorliegt. Dieses Problem kann auf der Ebene der Programmierung angegangen werden und wird im Kapitel über die Code-Optimierung behandelt.

- **Verzweigungen**
  Verzweigungsinstruktionen verursachen in der Regel Verzögerungen in der Pipeline, da für das Laden der Zielinstruktion die Zieladresse berechnet werden muß. Man kann zwei Arten von Verzweigungen unterscheiden:

- Unbedingte Verzweigungen: es wird immer verzweigt;
- Bedingte Verzweigungen: es wird nur verzweigt, wenn eine Bedingung erfüllt ist, z.B. Registerinhalt = 0.

Bei beiden Verzweigungstypen muß die Zieladresse berechnet werden, was in der Regel zu Verzögerungen führt. Abbildung 2.8 beschreibt den Instruktionsfluß in einer statischen Instruktions-Pipeline anhand einer Beispielsequenz, die eine Verzweigung beinhaltet. Die Instruktions-Pipeline kann nach [19, 34] aus folgenden Stufen bestehen:

- Instruktion laden (fetch)
- Instruktion dekodieren (decode)
- Ausführung (execution)
- Ergebnis speichern (write)

Die erste Stufe lädt die auszuführende Instruktion in die Pipeline, die zweite erkennt, um welchen Befehl es sich handelt, und weist ihn der entsprechenden Funktionseinheit zu (gekennzeichnet durch Stufe 3). Abschließend wird das Ergebnis gespeichert.

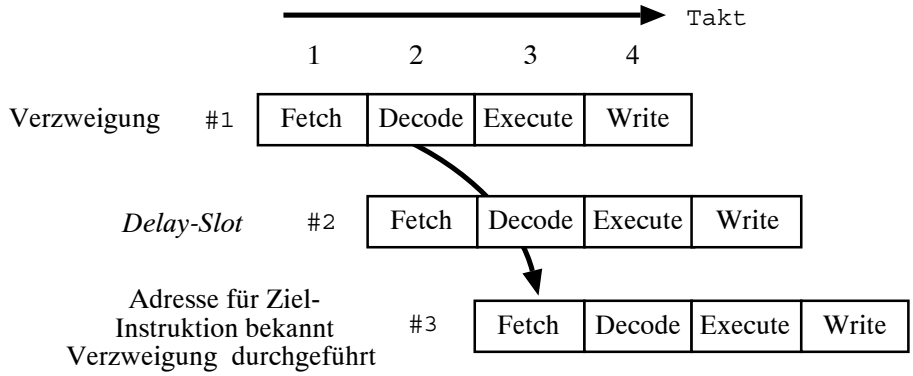


Abbildung 2.8: Darstellung des *Delay-Slots* einer Instruktions-Pipeline

Unter der Annahme, daß die Vergleichsoperation und Adreßberechnung zu Beginn des dritten Taktes abgeschlossen ist, kann die Zielinstruktion erst zu diesem Zeitpunkt geladen werden. Als *Delay-Slot* bezeichnet man den Verzögerungszeitraum, der in den meisten Fällen einen Takt beträgt. Dieser kann entweder mit Befehlen von oberhalb oder unterhalb des Verzweigungsbefehls gefüllt werden.

Um zu vermeiden, daß der Befehl im *Delay-Slot* bei einer falschen Vorhersage einer bedingten Verzweigung Speicherinhalte überschreibt, wird er markiert und arbeitet

auf Zwischenregistern. Stellt sich die Vorhersage als zutreffend heraus, werden die Markierungen zurückgenommen, ansonsten werden die Resultate gelöscht.

Die Güte der Zieladressenvorhersage hat großen Einfluß auf die Taktanzahl, die für die Abarbeitung einer Instruktionssequenz benötigt wird. Dies macht sich besonders bei Schleifenkonstrukten bemerkbar, die als Kontrollstruktur eine bedingte Verzweigung benutzen. Unter der Annahme, daß die Übersetzung einer `FOR`-Schleife am Ende einen bedingten Sprung zum Beginn der Schleife vorsieht, bis die letzte Iteration N erreicht ist, ist es günstig, die Startadresse der Schleife vorherzusagen. Die Vorhersage trifft dann bis auf die letzte Iteration zu, und es geht nur ein Takt verloren.

2.4 Duplizierung von Funktionseinheiten

Eine weitere Möglichkeit, die Performance eines Prozessorsystems zu erhöhen, besteht in dem multiplen Einsatz von Hardware-Komponenten, z.B. von

- Prozessoren,
- Pipelines oder
- Speichern.

Die Verwendung mehrerer Prozessoren führt zu leistungsfähigen Parallel-Architekturen, deren Beschreibung an dieser Stelle zu umfangreich wäre. Es sei darauf hingewiesen, daß der DEC Alpha-Chip AA-21064 Hardware-Einrichtungen vorsieht, die den Einsatz des Prozessors in Parallel-Architekturen begünstigen. So wird in einer Umgebung von mehreren Prozessoren anstelle einer einfachen Lade- und Speicher-Instruktion eine *Mikrosequenz* benutzt, die einen konfliktfreien Zugriff auf den gemeinsamen Speicher ermöglicht. Desweiteren verlangt die Alpha-AXP-Architektur keine explizite Abarbeitung der Befehle in der Reihenfolge ihrer Initiierung. Dies hat den Vorteil, daß beispielsweise bei einem erfolglosen Schreibzugriff auf den Speicher andere Speicher-Befehle schon erfolgen können, bevor der erste nach wiederholten Versuchen beendet ist. Eine strenge Ordnung der Abarbeitungs-Reihenfolge kann durch sogenannte Speicher-Barriere-Instruktionen erzwungen werden [26].

2.4.1 Parallele Verarbeitung von Instruktionen

Die Ausnutzung von Parallelität auf der Instruktionsebene ist ein immer wichtigeres Ziel bei der Entwicklung von Prozessoren und Compilern geworden. Es bestehen unterschiedliche Möglichkeiten, die gleichzeitige Ausgabe mehrerer Instruktionen zu realisieren. Manche Implementationen verlangen eine Kodierung der parallel auszugebenden Instruktionen in einer Anweisung (*Very Large Instruction Word, VLIW*).

Andere, ein Beispiel ist der untersuchte Alpha-Chip, sehen mehrere Instruktions-Pipelines vor, die in jedem Takt entsprechend viele Instruktionen als Block laden. Die Zuordnung der Instruktionen zu einem Block ist durch ihre Lage im Adreßraum bestimmt; dies wird in Kapitel 3 erläutert. In jedem Fall wird bei erfolgreicher paralleler Abarbeitung die Anzahl der Takte pro Instruktionssequenz um den Faktor vermindert, der der Anzahl der parallel ausführbaren Instruktionen entspricht. Können beispielsweise zwei Instruktionen gleichzeitig abgearbeitet werden, so halbiert sich die Anzahl der notwendigen Takte pro Instruktionssequenz.

2.4.2 Zweifache Ausführung des Cache-Speichers

Viele RISC-Prozessoren besitzen sowohl für Daten als auch für Instruktionen einen Cache-Speicher. Dies hat den Vorteil, daß die Speicherbandbreite zum Prozessor erhöht wird und der Datenzugriff parallel zur Instruktionsabarbeitung erfolgen kann. Desweiteren wird die Belegung des Daten-Cache-Speichers nicht durch interferierende Instruktionsblöcke erschwert, was für Optimierungstechniken eine wichtige Voraussetzung ist. Beim untersuchten Alpha-Chip sind auf dem Chip getrennte 8 KByte große Cache-Speicher für Instruktionen und Daten vorgesehen, die jeweils einfach-assoziativ sind. Der externe Cache-Speicher dagegen wird für Daten und Instruktionen gemeinsam benutzt, da außerhalb des Prozessors keine Information für eine Unterscheidung besteht.

Kapitel 3

Der Aufbau des DEC Alpha-Chip AA-21064

Der DEC Alpha-Chip ist der erste 64-Bit RISC-Prozessor, den der Hersteller Digital Equipment in eigener Produktion herstellt. Seiner Entwicklung ging die Überlegung voran, daß in nächster Zeit die bisherigen 32-Bit-Architekturen nicht mehr in der Lage sein werden, die immer größer werdenden Speicher zu adressieren. Neben der realisierten 64-Bit-Architektur tritt insbesondere die hohe Taktrate von bis zu 200 MHz hervor. Aufgrund der parallelen Anordnung der internen Funktionseinheiten kann theoretisch pro Takt eine Fließpunktoperation durchgeführt werden, was eine Peak-Performance von maximal 200 MFLOPS bedeutet. Da jedoch die Zugriffszeit der verwendeten Speicherbausteine keine entsprechende Verbesserung aufweist, muß untersucht werden, welche Rechenleistung tatsächlich erreicht werden kann. Nach einer kurzen Erläuterung der Unterschiede zwischen der DEC Alpha-AXP-Architektur und der Implementation durch den DEC Alpha-Chip AA-21064 wird der Aufbau des Prozessors beschrieben.

3.1 Überblick

Bei der Angabe der Entwicklungsziele des Alpha-Projekts unterscheidet der Hersteller DEC zwischen der Architektur eines Rechners und der Implementation dieser Architektur in der Hardware, z.B. auf einem Chip. Ausgehend von [2] wird die Computer-Architektur als die Summe der Merkmale und das Verhalten eines Computers aus der Sicht der Maschinensprachen-Programmierung definiert. Demgegenüber wird unter einer Implementation die konkrete Hardware-Struktur als spezielle Realisierung der Architektur verstanden. Die sogenannte Alpha-AXP-Architektur soll für einen Zeitraum von ca. 25 Jahren bestehen bleiben, während die Implementationen jeweils nach Maßgabe der technischen Möglichkeiten realisiert

werden. Aufgrund des langen Einsatzzeitraums der Architektur ist die Unabhängigkeit von einzelnen Betriebssystemen eine wichtige Randbedingung. Um dieses zu gewährleisten, wurde für die Formulierung der Betriebssystemroutinen der sogenannte PAL-Code[1] definiert. Er enthält neben dem normalen Instruktionsumfang einige Sonderbefehle, die den Zugriff auf alle Prozessorkomponenten ermöglichen. Für die unterschiedlichen Betriebssysteme existieren damit verschiedene PAL-Code-Routinen, die beim Starten des Systems resident in den Hauptspeicher geladen werden.

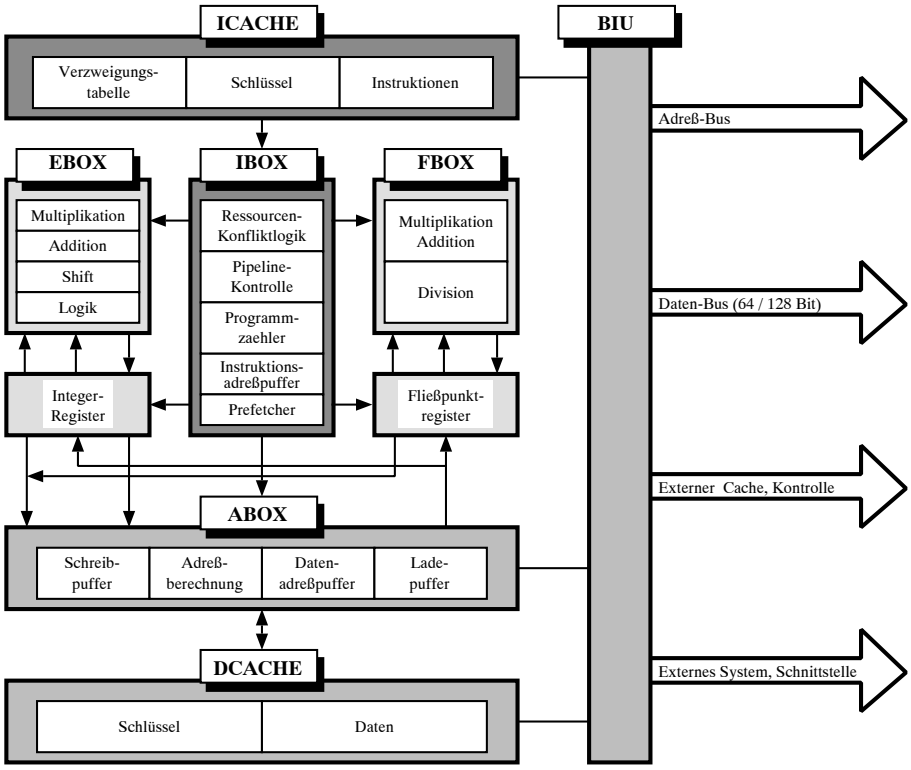


Abbildung 3.1: Komponenten des DEC Alpha-Chip AA-21064

Die Alpha-AXP-Architektur ist eine 64-Bit RISC Load/Store-Architektur: Alle Datenbewegungen zwischen dem Speichersystem und den Registern werden ausschließlich durch Lade- und Speicherinstruktionen ausgeführt, alle Operationen werden auf den Registerinhalten durchgeführt. Der DEC Alpha-Chip AA-21064 ist die erste Implementation der Alpha-Architektur. Er besteht, wie in Abbildung 3.1 dargestellt, aus folgenden Komponenten:

[1] Privileged-Architecture-Language

Das Leitwerk des Prozessors (IBOX) enthält eine doppelt ausgeführte statische Leitwerk-Pipeline, die das Laden, Dekodieren und die Weitergabe von Instruktionen an die entsprechenden Funktionseinheiten (EBOX, FBOX und ABOX) durchführt. Die Ressourcen-Konfliktlogik ermittelt den aktuellen Belegungszustand der jeweils anzusprechenden dynamischen Pipeline der Funktionseinheiten. Die Pipeline-Kontrolleinheit dient zur Steuerung der Leitwerk-Pipeline und verzögert im Konfliktfall die Instruktionsausgabe. Zur schnelleren Zuordnung von virtuellen und physikalischen Instruktionsadressen ist ein Puffer vorgesehen. Der Programmzähler beinhaltet die Adresse der aktuell bearbeiteten Instruktion. Die Prefetching-Einheit steuert das Laden von Instruktionen in die erste Stufe der Leitwerk-Pipeline.

Der interne Instruktions-Cache-Speicher (ICACHE) ist ein 8 KByte großer, einfach-assoziativer, physikalisch adressierter Cache-Speicher für die Zwischenspeicherung der zuletzt durchgeführten Instruktionen.

Die Integer-Funktionseinheit (EBOX) führt arithmetische und logische Integer-Operationen aus. Alle Instruktionen, außer der Division und Multiplikation, werden in einer dreistufigen dynamischen Pipeline ausgeführt und verarbeiten ausschließlich 64-Bit-Daten aus den 32 Integer-Registern. Die Multiplikation ist als Assembler-Befehl vorgesehen, wird aber nicht in der Pipeline abgearbeitet. Die Integer-Division wird durch ein Unterprogramm realisiert, welches vom Compiler zur Übersetzungszeit erstellt wird.

Die Fließpunkt-Funktionseinheit (FBOX) enthält eine sechsstufige dynamische Pipeline zur Durchführung von Fließpunktaddition und -multiplikation. Divisionsinstruktionen werden nicht in der Pipeline abgearbeitet und beanspruchen einen hohen Zeitaufwand zur Durchführung. Alle Fließpunktoperationen verarbeiten ausschließlich 64-Bit-Daten aus den 32 Fließpunktregistern

Der Integer-Registersatz verfügt über zwei Schreibpfade und vier Lesepfade zur ABOX und EBOX.

Der Fließpunkt-Registersatz verfügt über zwei Schreib- und drei Lesepfade, zwei davon zur EBOX. Es sind keine vier Lesepfade wie bei den Integer-Registern notwendig, da kein Instruktionspaar existiert, welches auf vier Fließpunktregister lesend zugreift.

Der interne Daten-Cache-Speicher (DCACHE) ist ein 8 KByte großer, einfach-assoziativer Cache-Speicher für die Zwischenspeicherung der zuletzt referenzierten Daten. Er ist physikalisch adressiert und nach der *write-through* Strategie ausgelegt.

Die Adreß-Funktionseinheit (ABOX) führt in einer dreistufigen Pipeline die Adreßberechnung und den Datenaustausch zwischen den Registern und dem übrigen Speichersystem aus. Die in Abbildung 3.1 angeführten Komponenten Lesefehler- und Schreibpuffer dienen als Zwischenspeicher zur Erweiterung der Datentransfer-Bandbreite. Im Datenadreßpuffer sind die zuletzt genutzten virtuellen und zugehörige physikalischen Speicheradressen abgelegt. Die *External Bus Interface Unit* (BIU) bildet die Schnittstelle zum externen Speichersystem.

Modellbezeichnung	**DEC 3000 Modell 400S AXP**	**DEC 3000 Modell 300 AXP**
Betriebssystem	Open VMS AXP, DEC OSF/1 AXP	
CPU	DECchip 21064 RISC Microprocessor	
Taktfrequenz	133 MHz (7.5 ns)	150 MHz (6.6 ns)
CPU Datenbusbreite	128 Bit	64 Bit
Instruktionsinitiierung	Maximal 2 Instruktionen pro Takt	
Maschinenwortlänge	64 Bit	
Adreßräume	34 Bit physikalisch, 43 Bit virtuell	
Hauptspeicher	128 MByte	32 MByte
ICache-Speicher	8 KByte, einfach-assoziativ	
DCache-Speicher	8 KByte, einfach-assoziativ, write-through	
Externer Cache-Speicher	512 KByte einfach-assoziativ write-back	256 KByte einfach-assoziativ write-back
Speicherbusbreite	256 Bit	64 Bit
Speicherlatenzzeit		
int. DCache-Speicher	3 Takte	3 Takte
ext. Cache-Speicher	8 Takte	8 Takte
Hauptspeicher	24-30 Takte	30-67 Takte

Tabelle 3.1: Hardware-Daten der untersuchten Rechner

Für die Ausführung von Verzweigungsinstruktionen ist keine eigene Funktionseinheit eingezeichnet, da die Abarbeitung von Fließpunkt- und Integer-Verzweigungen in verschiedenen Funktionseinheiten stattfindet.

Die Performance-Analysen wurden auf den Modellen

- DEC 3000, Modell 400S AXP (Server) und
- DEC 3000, Modell 300 AXP

durchgeführt, deren Daten in Tabelle 3.1 aufgelistet sind. Die Unterschiede zwischen beiden Rechnern bestehen im wesentlichen in der unterschiedlichen Taktfrequenz, Speichergröße und Breite der Datenwege.

3.2 Datenformate und Instruktionstypen

Die folgende Beschreibung soll einen kurzen Überblick über die von der Alpha-Architektur vorgesehenen Datenformate und Instruktionstypen geben. Zum leichteren Verständnis der in den folgenden Kapiteln verwendeten Assembler-Befehle ist im Anhang eine tabellarische Auflistung der benutzten Instruktionen angefügt [7].

Datenformate

Der Basis-Datentyp, die sogenannte Maschinenwortlänge, ist in der Alpha-Architektur zu 64 Bit definiert. Daten dieses Formats werden sowohl im Speicher als auch in den Registern abgelegt. Darüber hinaus können aus Gründen der Kompatibilität auch 32- Bit-Daten im Speicher abgelegt und, auf 64 Bit expandiert, im Register bearbeitet werden.

Es existieren die drei fundamentalen Datenformate:

- Integer,
- IEEE Fließpunkt,
- VAX Fließpunkt,

die jeweils in der 32-Bit- und 64-Bit-Form vorgesehen sind.

In Abbildung 3.2 sind die Integer- und IEEE-Fließpunktformate für den Speicher und die Register dargestellt. In der kanonischen Form[2] eines 32-Bit-Wertes in einem Integer-Register (64 Bit) sind alle höchstwertigen 33 Bitstellen gleich dem Bit 31. Die kanonische Form eines 32-Bit-Wertes in einem Fließpunkt-Register besteht aus dem von 8 auf 11 Bit expandierten Exponenten und der von 23 auf 52 Bit expandierten Mantisse. Dies erlaubt die Verwendung von einfach-genauen Fließpunktwerten in Arithmetik- und Verzweigungsbefehlen, die doppelt genaue Fließpunktwerte benutzen. Die darstellbaren Zahlenbereiche für VAX- und IEEE-Fließpunktzahlen sind in der Tabelle 3.2 angegeben.

Format	Genauigkeit	Min	Max
VAX	Single Precision (32 Bit)	0.294E-38	1.700E38
VAX	Double Precision (64 Bit)	0.560E-308	0.899E308
IEEE	Single Precision (32 Bit)	1.175E-38	3.400E38
IEEE	Double Precision (64 Bit)	2.225E-308	1.798E308

Tabelle 3.2: Darstellbare Zahlenbereiche im VAX- und IEEE-Format

[2]Die kanonische Form ist eine standardisierte Darstellung von redundant kodierten Daten [26].

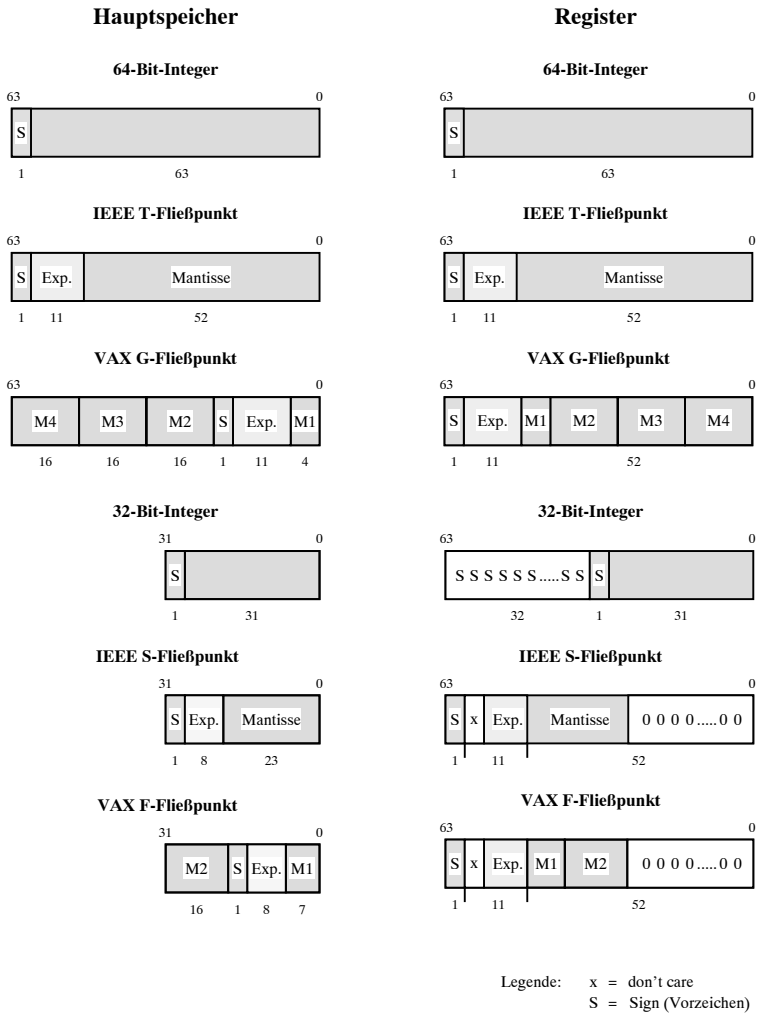


Abbildung 3.2: Datenformate der Alpha-Architektur

Die Anordnung der Byte-Blöcke innerhalb eines Datenwortes unterscheidet *big-endian* und *little-endian* Adressierung:

big-endian:	Byte 0 ist das höchstwertige Byte
little-endian:	Byte 0 ist das niederwertigste Byte

Die Bit-Anordnung innerhalb eines Byte ist in beiden Fällen *little-endian*. Der Default für die Alpha-Architektur ist *little-endian*.

Instruktionstypen

Alle in der Alpha-Architektur definierten Instruktionen haben eine Länge von 32 Bit und sind im Hauptspeicher auf 32 Bit Grenzen (*longword aligned*) abgelegt. In Abbildung 3.3 sind die Assembler-Formate der vier vorgesehenen Instruktionstypen

- Operation,
- Speicher,
- Verzweigung und
- Betriebssystem (CALL_PAL)

dargestellt. Sie beinhalten einen 6 Bit langen Operations-Code zur Identifizierung und maximal drei Registerfelder RA, RB und RC. Die restlichen Bitstellen enthalten Konstanten für Arithmetikinstruktionen, Verschiebewerte für Sprunginstruktionen oder Erweiterungen des Operations-Codes durch das Funktionsfeld zur Identifizierung von bereits bestehenden und zukünftig vorgesehenen Instruktionen.

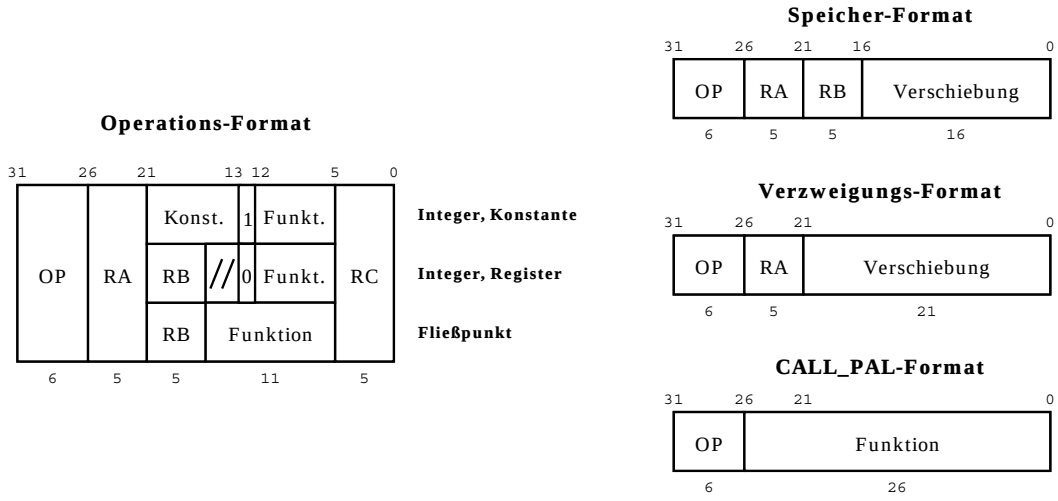


Abbildung 3.3: Instruktionstypen der Alpha-Architektur

Operation

Alle Operationsinstruktionen sind Register-Register Befehle, die die drei Register RA, RB und RC in der Weise **RA operate RB** $\rightarrow$ **RC** benutzen. Integer-Befehle werden durch den Operations-Code und das Funktionsfeld identifiziert. An die Stelle der Registerbenennung RB kann eine 8 Bit lange Konstante treten, deren höherwertige nicht belegte Stellen mit Nullen aufgefüllt sind. Integer-Divisionen werden durch Unterprogrammaufrufe realisiert, die vom Compiler erstellt und in den Assembler-Code eingefügt werden. Fließpunktoperationen werden anhand des Funktionsfeldes identifiziert. Die Verwendung von Konstanten ist nicht vorgesehen.

Speicher

Bei Lade- und Speicherinstruktionen steht in Register RA der zu speichernde Wert, während in Register RB die Konstante zur Berechnung der virtuellen Adresse abgelegt ist. Zur Adreßberechnung wird die 16-Bit-Konstante des Verschiebungsfeldes auf 64 Bit expandiert und zum Inhalt des Registers RB addiert. Ein eventuell auftretender Überlauf wird ignoriert.

Da alle Register-Register Operationen[3] auf 64-Bit-Daten durchgeführt werden, sind für das Laden und Speichern von 32-Bit-Hauptspeicher-Daten spezielle Instruktionen vorgesehen: Ladeinstruktionen expandieren 32-Bit-Daten auf 64 Bit in kanonischer Form, Speicherinstruktionen komprimieren 64-Bit-Werte auf 32 Bit. Lade- und Speicherinstruktionen, die nur auf Teile eines Registers wirken, sind nicht vorgesehen. Sie können durch kurze Instruktionssequenzen ersetzt werden, die in [7] angegeben sind.

Verzweigung

Allgemein können drei verschiedene Arten von Verzweigungsbefehlen unterschieden werden:

1. *Bedingte Sprünge:* Zur Durchführung von bedingten Sprüngen wird der Inhalt des Registers RA getestet und bei Erfolg die virtuelle Zieladresse berechnet. Dazu wird die Versatzkonstante um zwei Bit nach links verschoben und zum Wert des Programmzählers addiert. Die Verschiebung um zwei Bit ermöglicht die Adressierung von aufeinanderfolgenden Instruktionen. Ist die Verzweigungsbedingung nicht erfüllt, wird der Programmzähler mit der sequentiell folgenden Adresse geladen.

2. *Unbedingte Sprünge:* Bei unbedingten Sprüngen entfällt der Test des Registers RA. Dafür wird nach Berechnung der virtuellen Adresse der aktuelle Wert des Programmzählers nach RA geschrieben. Die 21 Bit des Verschiebungsfeldes machen Sprünge um ± 4 MByte möglich.

3. *Unterprogramm-Sprünge:* Bei Unterprogramm-Sprüngen wird zuerst die Adresse der sequentiell folgenden Instruktion in das Register RA geschrieben. Danach erfolgt das Laden des Programmzählers mit der Unterprogrammadresse.

Für alle Verzweigungsinstruktionen besteht die Möglichkeit, die Zieladresse vor der endgültigen Berechnung vorherzusagen. In Abschnitt 3.3.3 wird beschrieben, wie diese Technik das frühzeitige Laden von Instruktionen ermöglicht.

Eine Besonderheit des Instruktionssatzes der Alpha-Architektur sind die sogenannten *bedingten Verschiebeinstruktionen*, die Daten zwischen Registern nach Abfrage

[3] Nur die Lade- und Speicherinstruktionen sind keine Register-Register Befehle.

einer Bedingung verschieben. Hierdurch wird Compilern die Möglichkeit gegeben, Verzweigungsinstruktionen einzusparen und damit die Performance zu erhöhen.

Betriebssystem (CALL_PAL)

Diese Instruktionen rufen Betriebssystemroutinen (PAL-Code) auf, die durch den Inhalt des Operations- und Funktionsfeldes identifiziert werden. Der PAL-Code wird im sogenannten *Kernel-Mode* durchgeführt und kann von Benutzer-Programmen nicht beeinflußt werden. PAL-Code-Routinen benutzen neben den oben beschriebenen Instruktionen eine Reihe von Sonderbefehlen, die das Referenzieren von internen Kontrollregistern, das Füllen von Adreßpuffern etc. ermöglichen. Bei der Initialisierung dieser Routinen wird die Pipeline des Prozessors geleert, der aktuelle Programmzähler in einem internen Register zwischengespeichert und die entsprechende PAL-Code-Adresse geladen. Mit einem reservierten Befehl wird nach Beendigung der Routine wieder in den Benutzer-Modus verzweigt und der Zustand vor der Verzweigung wieder hergestellt. Mit Hilfe von PAL-Code-Sequenzen können verschiedene Betriebssysteme an die Implementationen angepaßt werden. Konkrete Anwendungen werden im Rahmen dieser Arbeit nicht besprochen.

Multiprozessor-Unterstützung

Die Alpha-Architektur sieht bereits einige Instruktionen für die Implementation von Multiprozessorsystemen zur Unterstützung eines Shared-Virtual-Memory Systems vor:

- *load-locked*,
- *in-register modify*,
- *store-conditional* und
- *test*.

Wenn diese Sequenz ohne *Interrupt, Exception* oder einen gleichzeitigen Schreibzugriff eines anderen Prozessors beendet werden kann, ist ein sogenannter *atomic update* durchgeführt. Im anderen Fall muß der Prozeß wiederholt werden. Außerdem wird keine strenge Lade/Speicher-Reihenfolge verlangt: Wenn ein Prozessor sequentiell auf die Speicherstellen A und B schreibt, der Zugriff auf A erst nach einem erneuten Versuch gelingt, so wird ein zweiter Prozessor den Schreibvorgang in der Reihenfolge B, A registrieren. Ist eine strenge Lade/Speicher-Reihenfolge notwendig, so werden Speicher-Barriere-Instruktionen eingefügt, die die ursprüngliche Zugriffsreihenfolge garantieren.

3.3 Leitwerk (IBOX)

Kernstück des Leitwerks ist die in Abbildung 3.4 dargestellte vierstufige Leitwerk-Pipeline, die alle Instruktionen durchlaufen, bevor sie auf die einzelnen Funktionseinheiten verteilt werden. Sie ist doppelt ausgeführt, so daß in jedem Takt gleichzeitig zwei Instruktionen (ein Instruktionspaar) geladen werden können[4]

Sämtliche Stufen dieser Pipeline sind statisch ausgeführt, damit bei einer Konfliktsituation die Ausgabe der Instruktionen an die Funktionseinheiten verzögert werden kann. Da die Pipelines der anliegenden Funktionseinheiten dynamisch sind und aus diesem Grund nicht angehalten werden können, muß vor der Ausgabe (Stufe 4) die Konfliktfreiheit garantiert werden. Desweiteren besteht die Möglichkeit, daß die Stufen innerhalb der jeweiligen Pipeline unabhängig voneinander ihren Zustand behalten können, während die jeweils vorherige Stufe weitergeschaltet wird. Der Vorteil, den diese Technik bietet, wird bei der Behandlung von sogenannten Pipeline-Blasen deutlich. Dies sind leere Pipeline-Stufen, die durch die Verzweigungsvorhersage und das Pipeline-Löschen infolge von Abbruchsituationen.

Man kann zwei Klassen von Abbruchsituationen unterscheiden:

- exceptions
- non-exceptions

Die erste Klasse verlangt das komplette Löschen der Pipeline, bevor die Instruktionen ab einer neuen Startadresse neu geladen werden. In beiden Fällen müssen alle Befehle gelöscht werden, die nach der Instruktion, die den Abbruch verursacht hat, geladen werden. Dies beinhaltet auch das Stoppen der gleichzeitig ausgegebenen Instruktion.

Im Fall der *non-exceptions* können die ausstehenden Instruktionen beendet werden, bevor die Pipeline neu initialisiert wird. Beispiele für diese Gruppe von Ausnahmen sind nicht zutreffende Vorhersagen von Verzweigungsinstruktionen und Lesefehler beim Referenzieren des Instruktions-Cache-Speichers. Blasen entstehen auch bei jeder vorhergesagten Verzweigung in Stufe 2 der Leitwerk-Pipeline, da das bereits geladene Instruktionspaar in Stufe 1 gelöscht werden muß.

Der Programmzähler, der dem sogenannten *prefetcher* zugeordnet ist, bestimmt das nächste zu ladende Instruktionspaar und wird bei jeder durchgeführten Instruktionsausgabe in Stufe 4 der Leitwerk-Pipeline um 8 Byte erhöht (2 Instruktionen). Er wird neu initialisiert durch die Zieladresse einer Verzweigungsvorhersage in Stufe 2, oder durch die neue Startadresse nach einer Abbruchsituation.

[4]Zur vereinfachten Beschreibung wird folgende Notation verwendet:
- Stufen der Leitwerk-Pipeline A: 1a, 2a, 3a, 4a
- Stufen der Leitwerk-Pipeline B: 1b, 2b, 3b, 4b

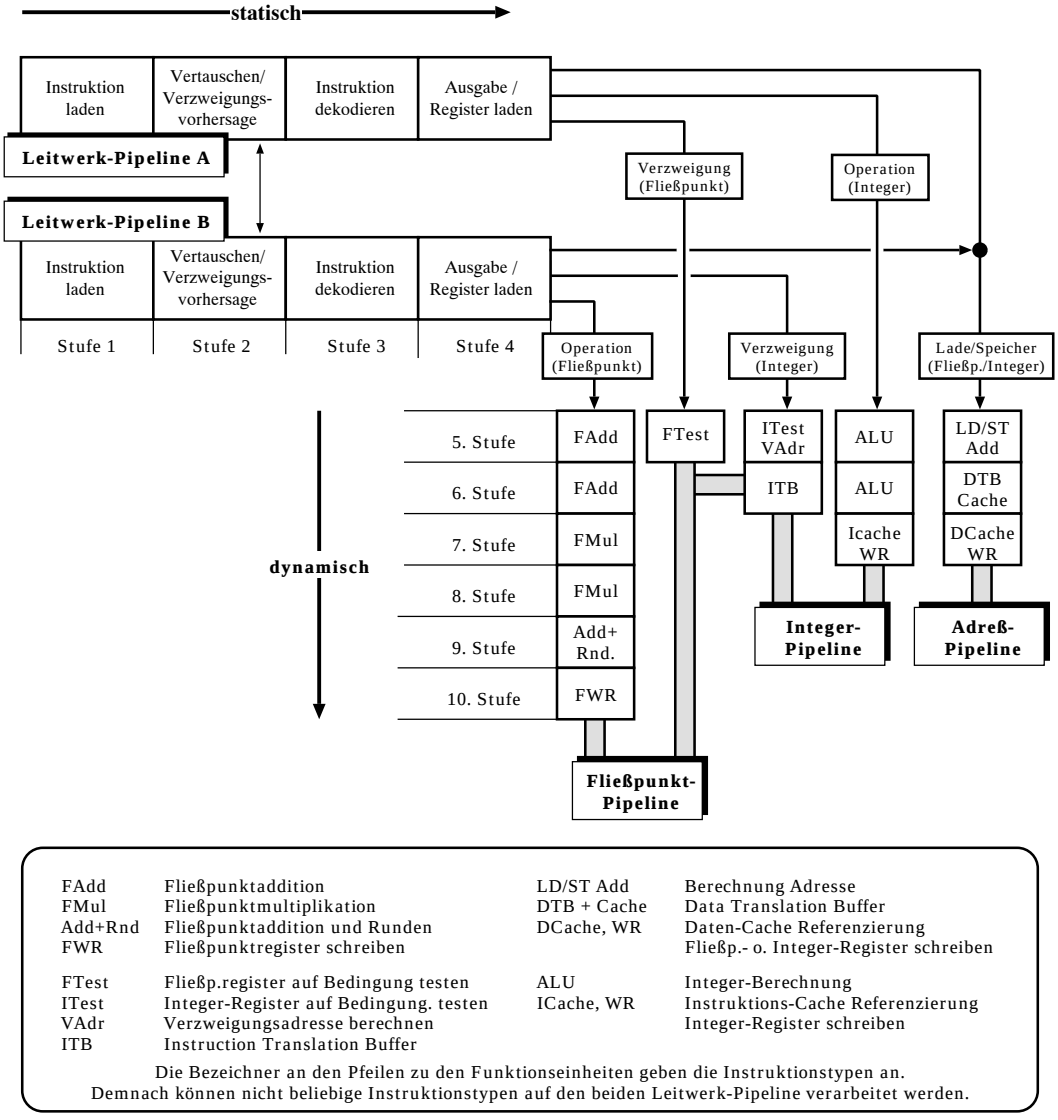


Abbildung 3.4: Leitwerk-Pipeline mit zugeordneten Komponenten

3.3.1 Leitwerk-Pipeline

Stufe 1: Instruktion laden

Wie in der Abbildung 3.4 dargestellt, wird in der ersten Stufe ein Instruktionspaar geladen. Die notwendige Adresse liefert der Programmzähler der Prefetch-Einheit. Die Adresse der ersten Instruktion eines Paares liegt immer auf einer 8-Byte-Grenze, d.h. die letzten drei Bit-Stellen der Byte-orientierten Adresse sind alle gleich null. Soll die zweite Instruktion eines Paares mit dem unmittelbar nachfolgenden Befehl

parallel eingelesen werden, so muß der Compiler, wie in Abbildung 3.5 verdeutlicht, einen NOP-Befehl[5] einfügen.

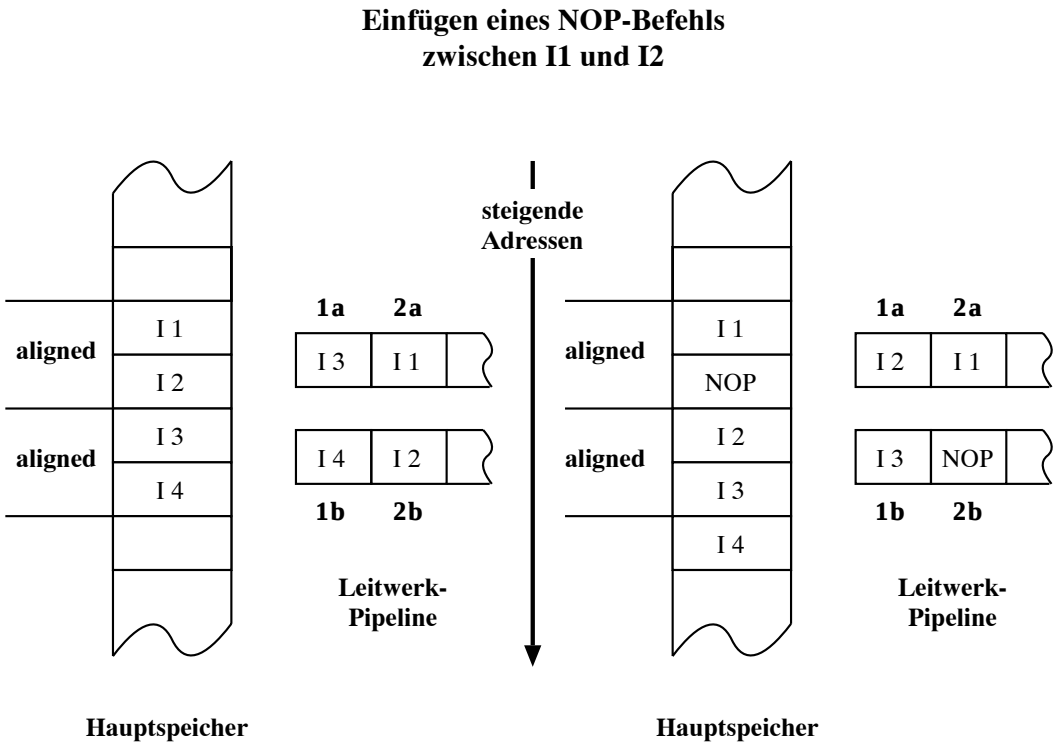


Abbildung 3.5: Einfügen von NOP-Befehlen für Instruktions-Scheduling

Ein Nachteil dieser Vorgehensweise ist das Anwachsen des Code-Umfangs durch das Einfügen von zusätzlichen Befehlen, was unter Umständen zu erhöhten Zugriffsfehlern im Instruktions-Cache-Speicher führen kann. Der Vorteil eines geschickt plazierten NOP-Befehls wird im nächsten Abschnitt deutlich.

Stufe 2: Instruktionen Vertauschen / Verzweigungsvorhersage

Während in die Stufen 1a und 1b beliebige Instruktionstypen geladen werden können, sind in den Stufen 2 bis 4 Restriktionen zu beachten. Zur Vereinfachung der Hardware wurden die Instruktionen, wie in Tabelle 3.3 dargestellt, in zwei Klassen aufgeteilt und der jeweiligen Pipeline zugeordnet.

[5] Beispielsweise ld r31,r31. Das Integer-Register r31 und das Fließpunktregister f31 besitzen immer den Wert Null, so daß der vorstehende Befehl nichts bewirkt.

Pipeline A	Pipeline B
Operation (Integer)	Operation (Fließpunkt)
Verzweigung (Fließpunkt)	Verzweigung (Integer)
Speicher (Integer)	Speicher (Fließpunkt)
Laden, Fließpunkt/Integer	

Tabelle 3.3: Zuordnung von Instruktionstypen und Leitwerk-Pipeline

Infolge dieser Maßnahme können Situationen entstehen, die das Austauschen der Instruktionen eines Paares oder die Serialisierung notwendig machen. Folgende Fälle werden unterschieden:

- Die Instruktionen gehören jeweils der falschen Klasse an: Die Instruktionen werden vertauscht.
- Die Instruktionen gehören derselben Klasse an: Die Instruktionen werden serialisiert und der korrekten Pipeline zugewiesen.
- Die Instruktionen benutzen dieselben Register: Dies wird in Stufe 3 (Dekodierung) erkannt. Die Instruktionen werden serialisiert und der korrekten Pipeline zugewiesen.

Die Ausgabereihenfolge wird dabei immer garantiert, d.h. I1 wird nie nach I2 ausgeben. Hierauf wird bei der Beschreibung von Stufe 4 noch einmal eingegangen.

Abbildung 3.6 verdeutlicht die Auswirkungen einer ungünstig ausgerichteten Instruktionssequenz: Im Takt 4 muß das Instruktionspaar der dritten Stufe serialisiert werden, da beide das Register `f21` referenzieren. Durch die Serialisierung entstehen sogenannte Pipeline-Blasen, die in der Abbildung durch waagerechte Striche gekennzeichnet sind. Im Takt 5 können die beiden Ladeinstruktionen zwar in beiden Pipelines abgearbeitet werden, jedoch in Stufe 4 nicht parallel ausgegeben werden. Daher wird auch dieses Paar serialisiert. In Takt 6 müssen die Instruktionen der Stufe 2 ausgetauscht werden, um sie der richtigen Pipeline zuzuführen. Diese Operation verursacht keine Verzögerung. Die oben beschriebenen Serialisierungen jedoch haben vier leere Pipeline-Stufen zur Folge, was einem Verlust von zwei Takten entspricht.

Abbildung 3.7 zeigt, daß durch geschicktes Einfügen einer NOP-Operation durch den Compiler die Serialisierung der Instruktionspaare vermieden werden kann (die Verzögerungszeit durch Registerabhängigkeit zwischen der Instruktion `ldf f21` und `addf f21,f22,f23` in Stufe 4 soll im folgenden nicht berücksichtigt werden). Das Einfügen der NOP-Instruktion hinter dem ersten Ladebefehl hat eine Veränderung der folgenden Instruktionspaare zur Folge, die keine Serialisierung mehr notwendig macht. Anstelle der vier NOP-Instruktionen ist nur noch die eingefügte Füllinstruktion vorhanden. Läßt man Datenabhängigkeiten außer Acht, so bestimmt die

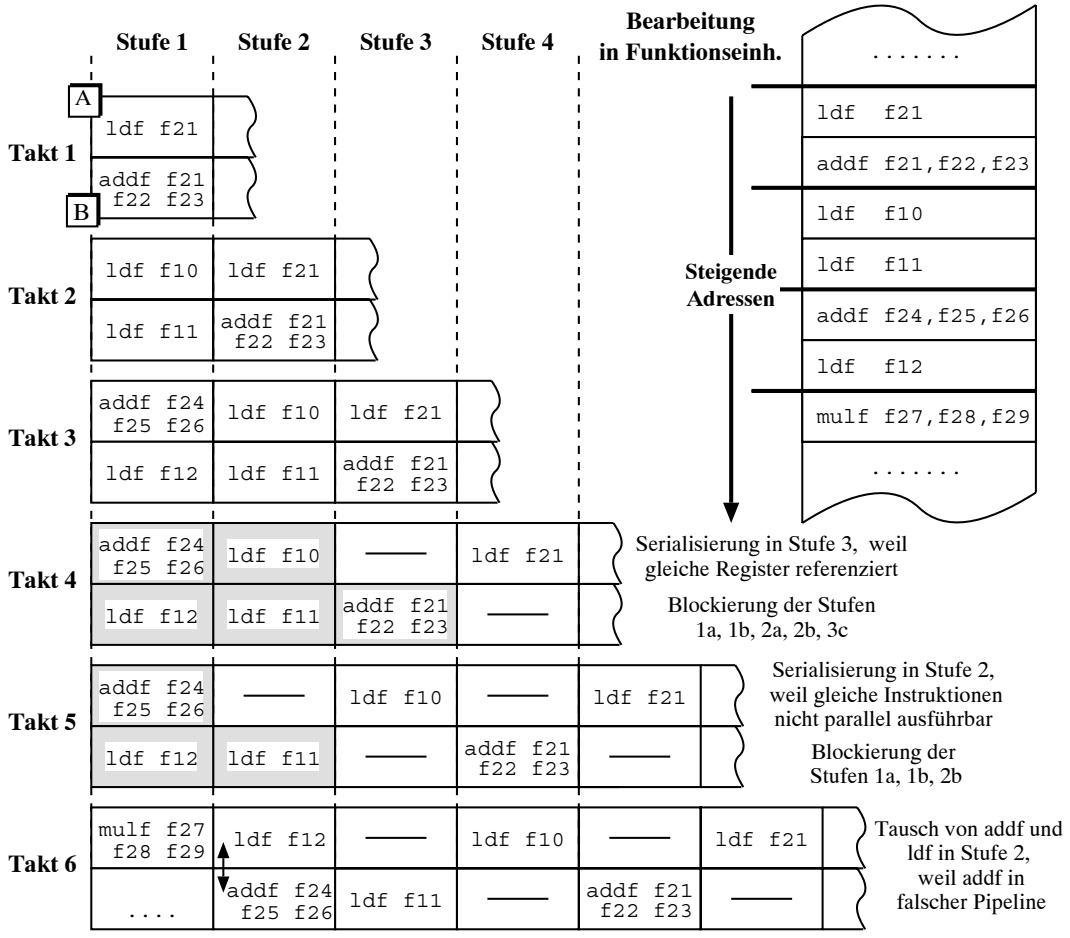


Abbildung 3.6: Auswirkung von ungünstig ausgerichteten Instruktionen

Anzahl der Takte, die zum vollständigen Einlesen der Instruktionssequenz notwendig sind, auch gleichzeitig die notwendige Taktanzahl zur Durchführung. Daraus läßt sich im angegebenen Beispiel ein Gewinn von zwei Takten durch Einführung des NOP-Befehls ableiten.

Die Verzweigungsvorhersage dient bei Verzweigungsbefehlen zur frühzeitigen Bestimmung der Zieladresse. Wird eine Verzweigung als durchgeführt vorhergesagt, so wird der Programmzähler mit dem Wert der Zieladresse geladen und das bereits geladene Instruktionspaar in Stufe 1 gelöscht. Die entstehende Pipeline-Blase kann beseitigt werden, indem bei einem Stillstand der Leitwerk-Pipeline, verursacht durch einen Konfliktfall, die nachrückenden Instruktionen das leere Stufenpaar auffüllen. Auf die verschiedenen Verfahren der Vorhersage wird in Abschnitt 3.3.3 eingegangen.

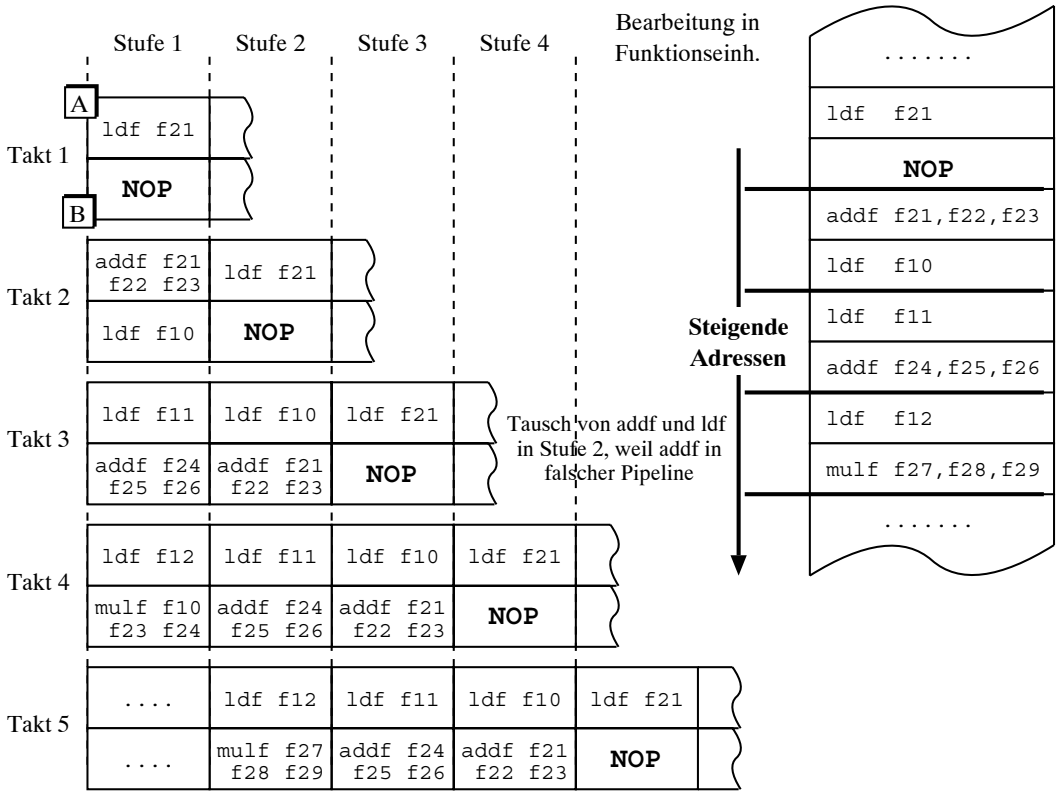


Abbildung 3.7: Einfügen eines NOP-Befehls zur Vermeidung von Pipeline-Blasen

Stufe 3: Instruktion dekodieren

In dieser Stufe wird das Instruktionspaar dekodiert. Hierbei wird erkannt, welche Register ggf. doppelt angesprochen werden. Ist dies der Fall, so wird, wie oben beschrieben, das betreffende Instruktionspaar serialisiert.

Stufe 4: Instruktionsausgabe / Register laden

Die vierte Stufe ist die Schnittstelle zu den dynamischen Pipeline-Stufen in den einzelnen Funktionseinheiten. Es werden keine Instruktionen ausgegeben, solange nicht alle benötigten Ressourcen[6] frei sind und die korrekte Ausgabereihenfolge, d.h. I1 vor I2, garantiert werden kann. Die Ausgabe erfolgt damit nach folgenden Regeln:

- I1 und I2 gleichzeitig, wenn alle Ressourcen frei sind.
- Nur I1, wenn Ressourcen nur für I1 frei sind.
- Keine Instruktion, wenn alle Ressourcen belegt oder nur Ressourcen für I2 frei sind.

[6]Unter Ressourcen fallen Funktionseinheiten und angesprochene Register.

Erfolgt die Ausgabe von I1 vor I2 (Fall 2), wird I2 danach allein ausgegeben, d.h. es wird keine Parallelausgabe mit einer folgenden Instruktion ausgeführt. Neben der Ausgabekontrolle werden in der 4. Stufe die Registerinhalte in die Pipeline-Register dieser Stufe geschrieben und bei der Ausgabe an die jeweilige Funktionseinheit weitergegeben.

3.3.2 Instruktionsausgabe und Bearbeitungszeiten

Datenabhängigkeiten bei der Instruktionsausgabe entstehen, wenn aufeinanderfolgende Befehle dieselben Register referenzieren. Instruktionen, die ein Register beschreiben, nennt man Erzeuger bezüglich dieses Registers. Instruktionen die diesen Wert aus dem Register lesen, werden Benutzer genannt. Die notwendige Zeit, um die die Ausgabe der Benutzerinstruktion verzögert werden muß, hängt von der Bearbeitungszeit der Erzeugerinstruktion in der jeweiligen Funktionseinheit ab.

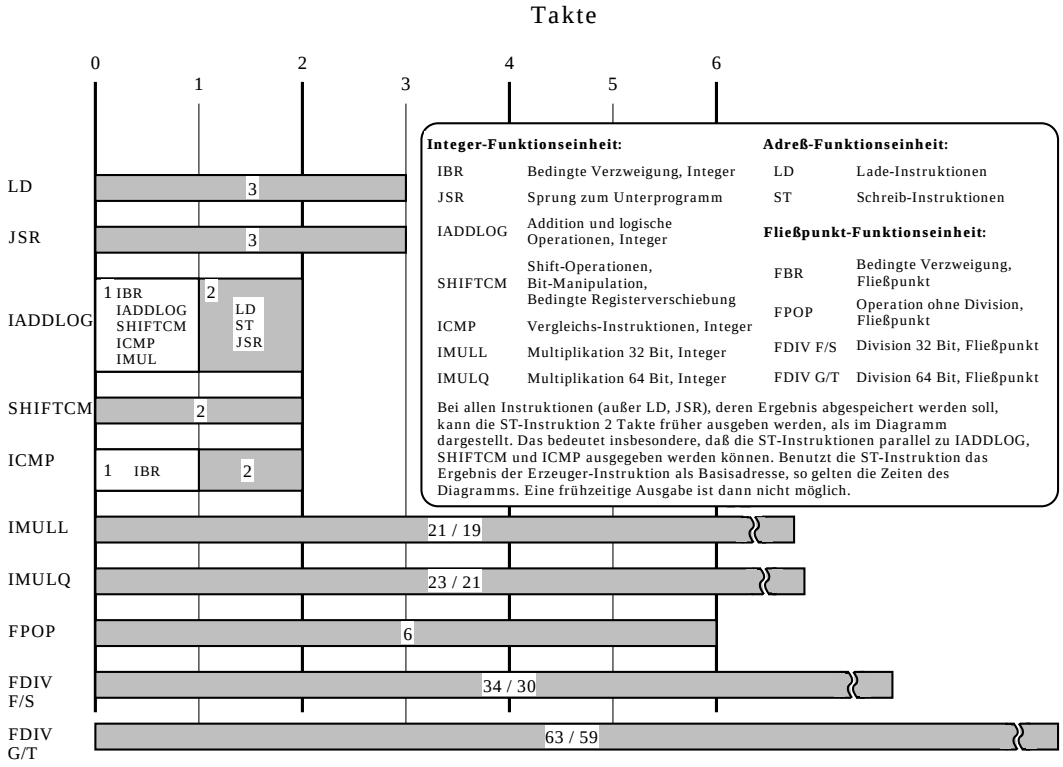


Abbildung 3.8: Ausführungszeiten von Instruktionen

In Abbildung 3.8 sind die Ausführungszeiten der verschiedenen Instruktionstypen dargestellt. Um diese Zeit muß die Instruktionsausgabe verzögert werden, wenn die gerade ausgegebene und die folgende Instruktion dasselbe Register benutzen.

```
ldq     r2,(r0)     Erzeugerinstruktion bezüglich r2
addq    r2,r3,r4    Erzeuger/Benutzer-Konflikt, verzögert Ausgabe
ldq     r2,(r1)     Erzeuger/Erzeuger-Konflikt,
                    kann mit ADDQ parallel ausgeben werden
```

Im allgemeinen sind nur Erzeuger/Benutzer-Konflikte von Interesse, da zwischen zwei interferierenden Erzeugerinstruktionen meistens eine Benutzerinstruktion vorkommt, was im obigen Beispiel gezeigt ist. Aufeinanderfolgende Benutzer/Benutzer- und Benutzer/Erzeuger-Zugriffe bedürfen keiner besonderen Aufmerksamkeit, weil die Benutzerzugriffe in Stufe 4 (Lese Registerinhalte) der Leitwerk-Pipeline erfolgen, also vor der Ausgabe.

Die nicht markierten Bereiche in Abbildung 3.8 kennzeichnen die Möglichkeit für einige folgende Instruktionen, frühzeitig auf das Ergebnis einer vorangehenden Instruktion zuzugreifen. Grund hierfür sind sogenannte *Bypass*-Kanäle zwischen einzelnen Stufen, die eine Kopie des Ergebnisses in die Stufe 4 der Pipeline ablegen, bevor der berechnete Wert im Registerblock abgespeichert wird. In der Legende der Abbildung 3.8 wird dies für die Speicherbefehlsklasse ST erläutert.

Instruktionsausgabe

Bei der Ausgabe einer Instruktion in Stufe 4 der Pipeline müssen folgende Bedingungen erfüllt sein:

- Alle benötigten Register sind frei;
- Eine Integer-Multiplikation ist nur möglich, wenn die Multipliziereinheit frei ist;
- Eine SHIFT-, IADDLOG-, ICMP- oder ICMOV-Instruktion kann frühestens drei Takte vor Beendigung einer IMUL-Instruktion ausgegeben werden;
- Eine LD-Instruktion kann frühestens zwei Takte nach der Ausgabe einer STC-Instruktion ausgegeben werden;
- Eine FDIV-Instruktion kann nur ausgeben werden, wenn die Fließpunktdivisionseinheit frei ist;
- Eine Fließpunktinstruktion kann frühestens sechs Takte vor der Beendigung einer Fließpunktdivision ausgegeben werden.

Die benutzten Ausdrücke der Instruktionstypen entsprechen denjenigen in Abbildung 3.8.

Parallele Instruktionsausgabe

Die parallele Ausgabe wird dann ausgeführt, wenn alle benötigten Ressourcen nach der oben beschriebenen Erzeuger/Benutzer Abhängigkeit frei sind und folgende Bedingungen erfüllt sind:

- Das Instruktionspaar liegt in einem ausgerichteten 8-Byte-Block.
- Die Instruktionen sind verschiedenen Gruppen zugeordnet (siehe Tabelle 3.3).
- Nicht mehr als eine Verzweigungsinstruktion kommt vor (Bsp.: Die parallele Ausgabe eines Unterprogrammaufrufes und einer Fließpunktverzweigung ist nicht möglich).
- Die Instruktionen gehören nicht beide den folgenden Instruktionstypen an: Lade / Speicher / Verzweigung.

3.3.3 Verzweigungs-Pipeline und Verzweigungsvorhersage

Für Verzweigungsbefehle wird in der Stufe 5 mit Hilfe der angegebenen Verzweigungsdistanz der neue virtuelle Programmzählerwert berechnet. Danach wird der Instruktionsadreßpuffer referenziert, um die physikalische Adresse der nächsten Instruktion zu generieren. Im letzten Schritt wird schließlich der Instruktions-Cache-Speicher auf das Vorhandensein dieser Instruktion überprüft und diese in die erste Stufe der Leitwerk-Pipeline geladen.

Die grundlegende Struktur von Verzweigungsbefehlen wurde bereits in Abschnitt 3.2 dargelegt. Die Alpha-Architektur unterscheidet zwischen Verzweigungstypen und Datenformaten des zu testenden Datums, so daß Teile der Instruktion entweder in der Fließpunkt- oder in der Integer-Funktionseinheit ausgeführt werden:

- Unterprogrammsprünge EBOX;
- Unbedingte Sprünge EBOX;
- Bedingte Sprünge
  - -Integer EBOX;
  - -Fließpunkt FBOX;

Dementsprechend ist in Abbildung 3.1 keine spezifische Box für Verzweigungen vorgesehen. Alle Verzweigungsinstruktionen beruhen auf dem Test eines einzigen Registerinhaltes, der je nach Verzweigungstyp aus dem Integer- oder Fließpunkt-Registersatz gelesen wird. Wie in Abbildung 3.4 dargestellt, erfolgt die Berechnung der Zieladresse für jeden Typ in der Integer-Funktionseinheit in Pipeline-Stufe 5 (VAdr). Zu diesem Zeitpunkt muß der Programmzähler der Prefetch-Einheit

zurückgesetzt werden, wenn die berechnete Adresse nicht mit der vorhergesagten aus Stufe 2 übereinstimmt. Weiterhin müssen alle bereits geladenen Instruktionen gelöscht werden, was einen Verlust von vier Takten bedeutet. In Stufe 6 wird schließlich die physikalische Adresse des zu ladenden Instruktionspaares durch Referenzierung des Instruktionsadreßpuffers bestimmt und aus dem Instruktions-Cache-Speicher in die erste Stufe der Leitwerk-Pipeline geladen. Ist die physikalische Adresse nicht im Puffer enthalten, so wird der Programmablauf gestoppt, schon ausgegebene Instruktionen werden beendet und die Instruktionen der Leitwerk-Pipeline zwischengespeichert. Anschließend wird in den PAL-Code verzweigt, der die Füllung des Puffers realisiert.

Der Instruktions-Cache-Speicher enthält die zuletzt referenzierten Instruktionspaare. Dies bewirkt besonders für wiederholt durchlaufene serielle Programmteile (z.B. Schleifenkonstrukte) eine Verminderung der Zugriffsdauer. Der Cache-Speicher ist physikalisch adressiert, einfach-assoziativ und besitzt eine Kapazität von 8 KByte, was die Speicherung von 1024 Instruktionspaaren ermöglicht. In einer Zeile des Cache-Speichers werden vier Instruktionspaare zwischengespeichert, die auf einer 32-Byte-Grenze ausgerichtet sind.

Bei einem Lesefehler im internen Instruktions-Cache-Speicher wird der externe Cache-Speicher referenziert. Dies geschieht über die Aktivierung der BIU, die bei einem Treffer mit einer Latenzzeit von fünf Takten (Werte für SRAM mit 20 ns Zugriffszeit, [10]) einen ausgerichteten 32 Byte Instruktionsblock in den Instruktions-Puffer lädt. Von hier gelangt das gewünschte Instruktionspaar in die Stufen 1a und 1b der Leitwerk-Pipeline. Erfolgt danach ein Zugriff auf ein anderes Paar im Puffer, so werden die vier Instruktionspaare in die zugehörige Zeile des Instruktions-Cache-Speichers übertragen. Der Puffer bietet mehrere Vorteile: Zum einen muß die Lade-Stufe der Leitwerk-Pipeline nicht auf die Beendigung des Füllvorganges der gesamten Zeile des Cache-Speichers warten. Zum anderen kann das Füllen des Cache-Speichers mit dem Laden des sequentiell folgenden Instruktionsblockes überlagert werden. Die Zwischenspeicherung ist vor allem bei ineinander geschachtelten Verzweigungen vorteilhaft. Folgt nach einem Sprung sofort ein weiterer, so stünden bei direkter Abspeicherung drei Instruktionspaare im Cache-Speicher, ohne daß sie referenziert werden würden. Da der Speicherplatz für Instruktionen im internen Speicher knapp bemessen ist und das Einfrieren der Pipeline bei Ladeverzögerungen durch Lesefehler erhebliche Performance-Einbußen bedeutet, ist eine hohe Trefferquote im internen Cache-Speicher sehr erstrebenswert.

Nach der Übertragung wird, wie oben schon erwähnt, parallel zur Ausführung der Instruktionen der Instruktions-Puffer mit dem sequentiell folgenden Block geladen. Versagt der Datenzugriff bei beiden Cache-Speichern, so realisiert die BIU die Übertragung eines ausgerichteten 512-Byte-Blockes (64 Instruktionspaare) aus dem Hauptspeicher in den externen Cache-Speicher sowie die oben beschriebene Übertragung der referenzierten Instruktionen.

Zur Beschleunigung der Adreßtransformation sind Instruktionsadreßpuffer (Instruc-

tion Translation Buffer, ITB) vorgesehen:

ITB 1:	8 Einträge für 8 KByte Seitengröße voll assoziativ LRU-Ersetzungs Algorithmus
ITB 2:	4 Einträge für 4 MByte Seitengröße voll assoziativ LRU-Ersetzungs-Algorithmus

Da die Abweichung vom seriellen Verhalten des Instruktionsflusses (Verzweigungen) in tiefen Pipeline-Strukturen Performance-Einbußen von mehreren Takten bedeutet, ist die frühzeitige Erkennung der wahrscheinlichen Zieladresse und damit der zu bearbeitenden Instruktion von großer Bedeutung. Dementsprechend sind in der Alpha-Architektur einige Verfahren zur Verzweigungsvorhersage vorgesehen, die sich in die Klassen statische und dynamische Vorhersage einordnen lassen.

Statische Verzweigungsvorhersage

Die sogenannte statische Verzweigungsvorhersage basiert auf Informationen in den Instruktionen selbst[7]. Für die drei verschiedenen Klassen von Verzweigungsbefehlen Bedingter, Unbedingter Sprung und Sprung zum Unterprogramm existieren jeweils verschiedene Möglichkeiten:

1. *Bedingte Sprünge:* Vorwärtsgerichtete Sprünge werden als nicht durchgeführt, rückwärtsgerichtete Sprünge als durchgeführt vorhergesagt. Dieser Hinweis setzt die weiter unten beschriebene dynamische Vorhersage nicht außer Kraft, kann aber ihre Notwendigkeit abmindern. Voraussetzung zur Ausnutzung dieser Festlegung ist eine entsprechende Konzeption des Compilers und der Hardware.

2. *Unbedingte Sprünge:* Nicht genutzte Bit-Stellen der Sprunginstruktionen werden so belegt, daß sie die niederwertigsten Ziffern der wahrscheinlichsten Zieladresse beinhalten. Die vorgesehenen 14 Bit sind ausreichend, die Adreßverschiebung innerhalb einer Speicherseite (8 KByte) zu spezifizieren.

3. *Unterprogramm-Sprünge:* Für Unterprogrammsprünge ist ein last-in, first-out Stack-Puffer mit vier Einträgen für die aktuellsten Rücksprungadressen vorgesehen.

Dynamische Verzweigungsvorhersage

Zusätzlich zu den oben erwähnten Möglichkeiten zur Verzweigungsvorhersage ist ein 2 KByte großer Puffer vorgesehen, der für jede Instruktion des Instruktions-Cache-Speichers ein sogenanntes *History-Bit* beinhaltet. Das *History-Bit* zeichnet

[7]Statisch deshalb, weil Instruktionen im Programmablauf nicht verändert werden.

das Sprungverhalten der ersten Durchführung einer Sprunganweisung auf. Daher ist auch die Bezeichnung *Dynamische Verzweigungsvorhersage* üblich. Diese Information kann als Vorhersage für spätere Ausführungen dieser Instruktion benutzt werden. Ist noch keine Information gespeichert, so kann das Vorzeichen-Bit des Verschiebungsfeldes interpretiert werden:

- Bit ist negativ: Die Sprungadresse wird vorhergesagt
- Bit ist positiv: Die seriell folgende Adresse wird vorhergesagt

Alternativ zur Benutzung des Verzweigungs-Puffers kann ausschließlich aufgrund des Vorzeichen-Bits das Resultat eines bedingten Sprunges vorhergesagt werden. Das Ein- und Ausschalten dieser Option geschieht durch Überschreiben eines internen Prozessorregisters durch PAL-Code. Es entstehen keine Extra-Kosten durch die Verzweigungsvorhersage, da das Prefetching mit der Berechnung der tatsächlichen Adresse überlagert wird.

3.4 Integer-Funktionseinheit (EBOX)

Die EBOX enthält zur Durchführung von Integer-Addition, logischen Operationen und Verzweigungsbefehlen eine dreistufige Pipeline. Multiplikationsoperationen werden nicht in der Pipeline ausgeführt und benötigen vier Takte zur Durchführung. Wie in der Abbildung 3.4 dargestellt, werden die Integer-Operationen in den ersten zwei Stufen durchgeführt und das Ergebnis während der dritten Stufe in eines der 32 Integer-Register geschrieben. Eine folgende Integer-Operation kann den berechneten Wert bereits vor dem Zurückschreiben benutzen. Der Integer-Registerblock enthält vier Lese- und zwei Schreibeingänge, die eine parallele Durchführung von Integer-Operationen, Lade-, Speicher- und Verzweigungsoperationen ermöglichen.

3.5 Fließpunkt-Funktionseinheit (FBOX)

Die Fließpunkt-Funktionseinheit beinhaltet zur Durchführung von Addition und Multiplikation eine fünfstufige Pipeline. Zur Bewahrung der Kompatibilität zu anderen Rechnerarchitekturen von DEC sind sowohl VAX- als auch IEEE-Datenformate für einfache und doppelte Genauigkeit vorgesehen. Ergebnisse von Fließpunktaddition und -multiplikation werden in der Pipeline-Stufe 9 in die Fließpunktregister geschrieben. Nachfolgende Operationen, die das Ergebnis benutzen, können bereits einen Takt früher auf das Ergebnis zugreifen.

3.6 Adreß-Funktionseinheit (ABOX)

Die Aufgabe der ABOX ist der Datentransfer zwischen dem Prozessor und dem angeschlossenen Speichersystem, das in den meisten Implementationen aus dem Haupt- und Cache-Speicher besteht. Virtuelle Adressen umfassen maximal 64 Bit, was bei einer Byte-orientierten Adressierung eine Adreßraumgröße 1.8^19 Byte ergibt. Die minimale Adreßraumgröße für Implementationen ist auf 2^{43} Byte festgelegt. Aus Gründen der Aufwärtskompatibilität zu zukünftigen Anwendungen müssen jedoch alle Bit-Stellen überprüft werden.

Ladeinstruktionen

Bei Ladeinstruktionen wird, wie in Abbildung 3.4 dargestellt, in der fünften Stufe die effektive virtuelle Adresse des zu ladenden Datums berechnet. Zur Adreßberechnung ist ein in der ABOX enthaltenes Addierwerk vorgesehen, das die effektive virtuelle Adresse durch Addition des aktuellen Programmzählerwertes und des im Befehl enthaltenen Versatzwertes erzeugt. Dies ist eine Voraussetzung für die Fähigkeit der Adreß-Funktionseinheit eine Instruktion pro Takt zu akzeptieren.

In der sechsten Stufe der Pipeline wird die zugehörige physikalische Adresse generiert. Dazu wird der Datenadreßpuffer auf die entsprechende Seiten-Basisadresse überprüft und bei Zugriffserfolg die Adreßbildung vollendet. Ist die physikalische Seitenadresse nicht gespeichert, so wird in den PAL-Code verzweigt, der den Puffer mit der gewünschten Adresse füllt. Der Puffer ist voll assoziativ und kann bis zu 32 Einträge enthalten, was einem maximalen Speicherbereich von 32 mal 8 KByte bis 32 mal 64 KByte entspricht, je nach gewählter Seitengröße. Die Standardseitengröße der untersuchten Rechner ist 8 KByte.

Im letzten Schritt wird das Datum mit der berechneten physikalischen Adresse aus dem internen Daten-Cache-Speicher in das angegebene Register geladen. Der Cache-Speicher ist ein 8 KByte großer, physikalisch adressierter, einfach-assoziativer write-through Cache-Speicher mit einer Zeilenbreite von 32 Byte (vier Maschinenworte a 64 Bit) und 256 Zeilen.

Tritt beim Referenzieren des internen Daten-Cache-Speichers ein Lesefehler auf, so muß das entsprechende Datum aus dem externen Speicher geladen werden. In diesem Fall stoppt die Adreß-Funktionseinheit die Annahme von weiteren Instruktionen, die die Leseeingänge der Register benutzen. Dies sind alle Lade- und Speicherbefehle, Sprungbefehle zu Unterprogrammen sowie einige PAL-Code Befehle zum Füllen des Daten-Cache-Speichers und Laden interner Prozessorregister. Zum Zeitpunkt der Erkennung eines Lesefehlers können sich bereits zwei nachfolgende Instruktionen in der Pipeline befinden. Diese werden wie folgt behandelt:

- Ladeinstruktionen, die keinen Lesefehler verursachen, werden ausgeführt.
- Ladeinstruktionen, die einen Lesefehler verursachen, werden im Ladepuffer

zwischengespeichert und wieder in die Pipeline eingefügt, wenn der erste Lesefehler behoben ist.

- Speicherinstruktionen werden zwischengespeichert und nach Behebung der Lesefehler in der ursprünglichen Reihenfolge (in Bezug auf ausstehende Ladeinstruktionen) dem Schreibpuffer zugeführt.

Das Referenzieren eines Datums aus dem externen Speicher beinhaltet immer das Laden eines ausgerichteten 32-Byte-Blocks (vier Maschinenworte). Das Speichersystem bei den benutzten Rechnern besteht aus einem zweiten Cache-Speicher und dem Hauptspeicher. Die expliziten Hardware-Daten sind in Tabelle 3.1 angegeben. Die Steuerung übernimmt die BIU (Bus Interface Unit), die mit der CPU je nach Rechnermodell über einen 64 Bit bzw. 128 Bit breiten Datenpfad verbunden ist. Der Ladevorgang folgt der Strategie, das gewünschte Maschinenwort als erstes Datum zu senden, so daß der betreffende Ladebefehl abgeschlossen werden kann, bevor die ganze Zeile geladen ist. Dies ist immer gewährleistet, wenn sich das entsprechende Datum im externen Cache-Speicher befindet. Die übertragenen Daten werden nicht direkt in den internen Daten-Cache-Speicher geschrieben, sondern in einem zweitem Puffer zwischengespeichert. Ist das angeforderte Datum angekommen, müssen noch die ausstehenden Ladebefehle im Lesefehler-Puffer abgearbeitet werden, deren Daten auch in den zweiten Puffer geschrieben werden. Erst wenn der Lesefehler-Puffer geleert ist und das angeforderte Datum der letzten ausstehenden Lade-Instruktion eingetroffen ist, werden weitere Instruktionen vom Leitwerk in die Adreß-Funktionseinheit ausgegeben. Ist die gesamte Zeile des Cache-Speichers von der BIU empfangen, erfolgt immer dann der Transfer aus dem Zwischenspeicher in den internen Daten-Cache-Speicher, wenn keine aktuellen Lade- und Speicherinstruktionen anliegen.

Speicherinstruktionen

Da der Prozessor im Idealfall ein Fließpunktergebnis pro Takt erzeugen kann, das angeschlossene Speichersystem jedoch mehrere Takte zum Abspeichern benötigt, ist ein Schreibpuffer vorgesehen, der die Speicherbandbreite zwischen Prozessor und Speichersystem erhöht. Desweiteren kann durch die Einführung des Puffers erreicht werden, daß Daten aus ausgerichteten 32-Byte-Blöcken in nur einem Transfervorgang in den externen Daten-Cache-Speicher geschrieben werden können. Außerdem besteht die Möglichkeit, Speicherinstruktionen mit derselben Zieladresse in einem Speichervorgang zusammenzufassen. Dies hat den Vorteil, daß die Datenpfade zum Speichersystem für andere Lade- und Speichervorgänge entlastet werden.

Wie in Abbildung 3.9 dargestellt, besteht der Schreibpuffer aus vier 32 Byte langen Blöcken, die jeweils über einen Anfangs- und Ende-Zeiger verfügen. Aus der Adreß-Pipeline gelangen Speicherinstruktionen auf die Positionen, die von den Ende-Zeigern markiert sind. An die BIU wiederum werden die Instruktionen gesendet, auf die die Anfangszeiger weisen. Die Zeiger werden nach diesen Vorgängen aktualisiert.

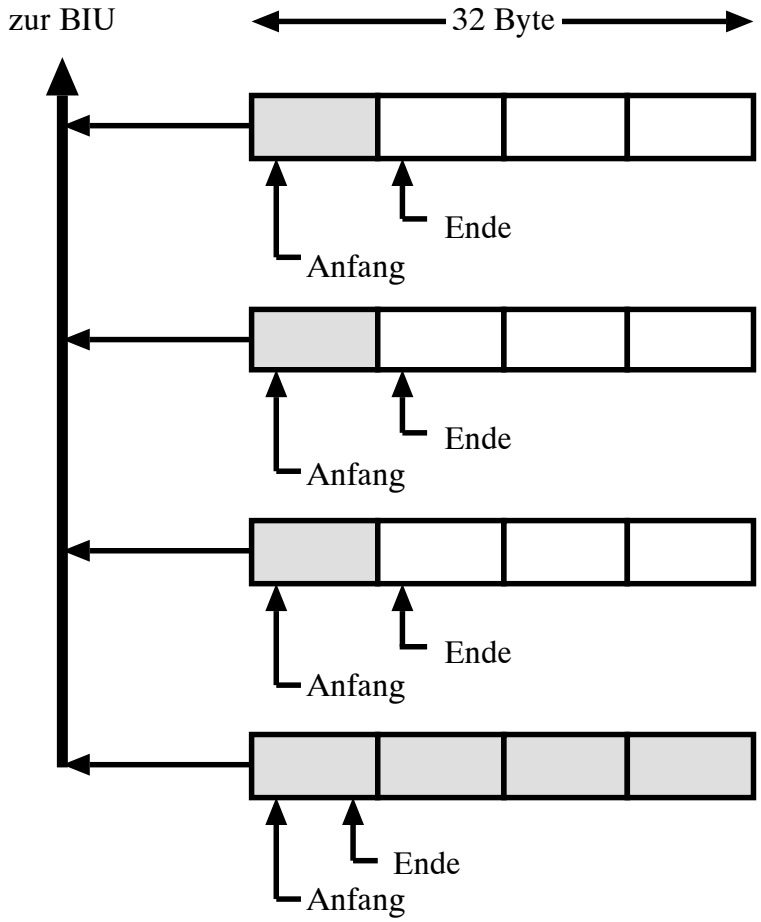


Abbildung 3.9: Aufbau des Schreibpuffers

Sie zeigen nur dann auf dieselbe Position, wenn der Block komplett gefüllt oder leer ist. Unter den folgenden Bedingungen versucht die Logik des Schreib-Puffers die zwischengespeicherten Schreibbefehle an die BIU zu senden:

- Mindestens zwei Einträge liegen vor;
- Ein Eintrag liegt vor, und seit der Ausführung der letzten Speicher-Instruktion in der Adreß-Pipeline sind 256 Takte verstrichen;
- Eine Speicher-Barriere-Instruktion ist im Puffer;
- Ein Lesefehler ist aufgetreten, der ein Datum im Schreib-Puffer betrifft. Daraufhin werden alle Schreibbefehle zur BIU transferiert;

Schon in Kapitel 2 wurde erwähnt, daß sich durch die Einführung von Puffern die Bearbeitungsreihenfolge der zwischengespeicherten Befehle ändern kann. Soll die ursprüngliche Zugriffsreihenfolge bewahrt bleiben, so sind explizite Speicher-Barriere-Instruktionen notwendig.

Kapitel 4

Optimierende Compiler

Für die Erstellung von Programmen werden heutzutage in der Regel Hochsprachen verwendet und die direkte Kodierung mit Assembler-Befehlen tritt mehr und mehr in den Hintergrund. Aus diesem Grund ist die Verfügbarkeit von effektiven Compilern zur Übersetzung von Hochsprachenprogrammen in Maschinenbefehle besonders wichtig. Um die hohe theoretische Leistungsfähigkeit *superskalarer* und *superpipelined* Prozessoren zu erreichen, muß das übersetzte Programm sowohl den internen Parallelismus des Prozessors als auch das hierarchische Speichersystem effektiv ausnutzen.

Die Herausforderung bei der Implementation eines optimierenden Compilers besteht darin, daß die Optimierungen für das Speichersystem und für die parallelen Prozessorkomponenten zum Teil unterschiedliche Informationen benötigen, jedoch nicht unabhängig voneinander betrachtet werden können. Existierende parallelisierende FORTRAN-Compiler benutzen in der Regel zwei verschiedene Programmrepräsentationen: Auf Hochsprachenebene werden Optimierungen für das Speichersystem durchgeführt, auf der Instruktionsebene Optimierungen für eine bessere Ausnutzung des prozessorinternen Parallelismus. Die Übersetzung des Programms erfolgt dann in zwei Stufen. Dies hat den Nachteil, daß die notwendigen Programmanalysen zweifach durchgeführt werden müssen und die Optimierungen nur eingeschränkt aufeinander abgestimmt werden können. Zur Zeit wird daran gearbeitet, die Optimierungen auf einer gemeinsamen Abstraktionsebene durchzuführen[35].

Der zur Verfügung stehende DEC FORTRAN-Compiler [9] ist nur auf die Optimierung der internen Prozessorkomponenten ausgerichtet, was besonders bei Algorithmen mit hoher Datenlokalität zu Performance-Werten führt, die weit unterhalb der erreichbaren Performance liegen. Im folgenden werden die vorhandenen Optimierungsfunktionen des Compilers untersucht und die notwendigen Informationen für die Durchführung der Optimierungen beschrieben. Zum Ende des Kapitels werden dann weitere Möglichkeiten zur Performance-Steigerung ermittelt, die vom verwendeten Compiler nicht unterstützt werden.

4.1 Datenabhängigkeit

Die Voraussetzung für die Anwendbarkeit von Programmtransformationen ist die Beibehaltung der Programmsemantik, d.h. es dürfen nur solche Änderungen vorgenommen werden, die das Ergebnis des ursprünglichen Programms nicht verändern. Beschränkungen existieren aufgrund bestehender Datenabhängigkeiten: Wird ein Wert in einem Programmteil benutzt, der vorher erzeugt worden ist, so dürfen diese Anweisungen nicht parallel ausgeführt werden. Die dazu notwendige Information liefert die Abhängigkeitsanalyse. Nach Wolfe [37] können zwei Arten von Abhängigkeiten unterschieden werden:

- die Kontrollflußabhängigkeit und
- die Datenflußabhängigkeit.

Kontrollflußabhängigkeiten entstehen bei der Benutzung von bedingten Anweisungen.

Beispiel 4.1:

```
S1   IF (L.GE.I) THEN
S2     A(L) = I
     ENDIF
```

Die Ausführung der Anweisung S2 ist abhängig von der Erfüllung der Bedingung in Anweisung S1.

Datenabhängigkeiten treten auf, wenn in Anweisungen Daten benutzt werden, die zu einem anderen Zeitpunkt erzeugt worden sind.

Beispiel 4.2

```
S1   A = B + D
S2   C = A * 3
```

In der Anweisung S1 wird der Wert der Variable `A` erzeugt und in Anweisung S2 benutzt. Eine Vertauschung von S1 und S2 würde die Semantik des Programmteils verändern und wäre demnach unzulässig.

Aus dem Blickwinkel der Wiederverwendung erscheint die Datenabhängigkeit nicht als Einschränkung, sondern als Möglichkeit, den Wert der Variable `A` im schnellen Teil der Speicherhierarchie zu halten und damit eine Referenzierung des langsameren Hauptspeichers zu vermeiden. Um die Zugriffszeit auf `A` in S2 zu verringern, sollte daher der Wert von `A` nach Ausführung der Anweisung S1 nicht sofort in den Hauptspeicher zurückgeschrieben, sondern in einem Register gehalten werden.

In dieser Arbeit werden die Definitionen für Datenabhängigkeiten benutzt, deren detaillierte Herleitung in [3, 36, 37] zu finden ist. Zur Beschreibung der Datenabhängigkeit definieren wir folgende Mengen:

Definition 4.1:

Die **Eingabemenge** IN(S) einer Anweisung S ist die Menge aller Variablen, deren Wert in dieser Anweisung benutzt wird, also auf der rechten Seite einer FORTRAN-Anweisung stehen.
Die **Ausgabemenge** OUT(S) einer Anweisung S ist die Menge aller Variablen, deren Wert in dieser Anweisung erzeugt wird und damit auf der linken Seite einer FORTRAN-Anweisung stehen.

Beispiel 4.3

```
S1      A = B + D
S2      C = A * 3
```

IN(S1) = {B, D} OUT(S1) = {A}
IN(S2) = {A} OUT(S2) = {C}

Betrachtet man Feldreferenzen in einer Schleife, so gehören der Ein- bzw. der Ausgabemenge nur diejenigen Elemente an, die in der Schleife benutzt oder erzeugt werden.

Beispiel 4.4

```
      DO I = 1, 3
S1      C(I) = A(I+1) * B
      ENDDO
```

IN(S1) = {A(2), A(3), A(4), B} OUT(S1) ={C(1), C(2), C(3)}

Werden in einer Schleife bedingte Anweisungen, wie z.B. eine IF-Anweisung benutzt und liegt keine weitere Information über das Ergebnis der Bedingung vor, so muß der Compiler die Ein- und Ausgabemenge der betreffenden Anweisung so bestimmen, als ob diese sowohl für den einen als auch für den anderen Fall durchgeführt werden würde. Folgende Typen von Datenabhängigkeiten werden unterschieden:

Definition 4.2:

Es existiert eine **echte Abhängigkeit** (true dependence) von der Anweisung S1 zur Anweisung S2,

$$\delta_{\mathtt{X}}\text{: S1} \rightarrow \text{S2},$$

wenn ein in S1 erzeugter Wert in eine Variable X abgelegt ($\mathtt{X} \in$ OUT(S1)) und später von S2 gelesen wird ($\mathtt{X} \in$ IN(S2)).

Beispiel 4.5

```
S1   X = 1
S2   Y = X
```

Definition 4.3:

Eine **Antiabhängigkeit** existiert von S1 nach S2,

$$\hat{\delta}_{\mathtt{X}}: \text{S1} \rightarrow \text{S2},$$

wenn eine Variable X von S1 gelesen wird (X $\in$ IN(S1)) und benutzt werden muß, bevor sie von S2 überschrieben wird (X $\in$ OUT(S2)).

Beispiel 4.6

```
S1   Y = X
S2   X = 5
```

S1 und S2 dürfen nicht parallel oder in umgekehrter Reihenfolge ausgeführt werden, weil sonst Y der Wert 5 und nicht der ursprüngliche Wert von X zugewiesen wird.

Definition 4.4:

Eine **Ausgabeabhängigkeit** existiert von S1 nach S2,

$$\delta^{O}_{\mathtt{X}}: \text{S1} \rightarrow \text{S2}.$$

wenn beide Anweisungen dieselbe Variable X erzeugen (X $\in$ OUT(S1) und X $\in$ OUT(S2)).

Beispiel 4.7

```
S1   X = 6
S2   Y = X
S3   X = 8
```

S1 und S3 dürfen nicht parallel ausgeführt werden, weil sonst die Zuweisung Y = X nicht definiert wäre.

Für die Erkennung der Wiederverwendung von Daten wird ein weiterer Abhängigkeitstyp, die Eingabeabhängigkeit definiert. Im Gegensatz zu den oben beschriebenen Abhängigkeiten schränkt eine Eingabeabhängigkeit eine parallele Ausführung zweier Ausdrücke nicht ein, solange der parallel lesende Zugriff auf die Variable Hardware-technisch möglich ist. **Definition 4.5:**

Eine **Eingabeabhängigkeit** existiert von S1 nach S2,

$$\delta^I_{\mathtt{X}}\colon \mathrm{S1} \rightarrow \mathrm{S2}.$$

wenn beide Anweisungen dieselbe Variable `X` (`X` $\in$ IN(S1) und `X` $\in$ IN(S2)) benutzen.

Beispiel 4.8

```
S1   Y = X
S2   Z = X
```

Die parallele Ausführung von S1 und S2 ist möglich, wenn der gleichzeitige Zugriff auf die Variable `X` Hardware-technisch realisiert werden kann. Auch die Ausführungsreihenfolge kann umgekehrt werden.

Besteht zwischen zwei Anweisungen eine Abhängigkeit, deren expliziter Typ nicht von Interesse ist, so wird die folgende Bezeichnung benutzt:

$$\delta^*_{\mathtt{X}}\colon \mathrm{S1} \rightarrow \mathrm{S2}.$$

Für die Laufzeitreduzierung von Programmen ist die Betrachtung von Datenabhängigkeiten in Schleifenkonstrukten besonders interessant, weil in diesem Fall ein hohes Maß an ausnutzbarer Datenlokalität bestehen kann. Zur Erläuterung von Datenabhängigkeiten in Schleifenkonstrukten sei folgendes Beispiel gegeben:

Beispiel 4.9

```
DO I1 = L1, U1
   DO I2 = L2, U2
      ...
         DO IN = LN, UN
            ...
            S1
            ...
            S2
         ENDDO
      ...
   ENDDO
ENDDO
```

Die Anordnung der geschachtelten Schleifen bestimmt die Reihenfolge, in der die einzelnen Iterationen durchlaufen werden. Für die folgenden Definitionen repräsentieren kleine Buchstaben konkrete Werte der Schleifenvariablen I1 bis IN.

Definition 4.7

Die **lexikographische Reihenfolge** ist die Iterationsfolge des untransformierten Schleifenkonstrukts, in dem die Iteration $\vec{i} = (i_1, i_2, ..., i_n)$ vor der Iteration $\vec{j} = (j_1, j_2, ..., j_n)$ ausgeführt wird,

$$(i_1, i_2, ..., i_n) < (j_1, j_2, ..., j_n),$$

wenn $i_1 < j_1$ oder ein Wert m, $2 \leq m \leq n$, existiert, so daß $i_k = j_k \ \ \forall \ \ k < m$ und $i_m < j_m$. $\vec{i}$ ist **lexikographisch kleiner** als $\vec{j}$, also $\vec{i} < \vec{j}$.

Die lexikographische Reihenfolge bestimmt auch die Auswertungsreihenfolge für die Anweisungen S1 und S2.

Definition 4.8:

Treten zwei Anweisungen $\mathrm{S1}(i_1, i_2, ..., i_n)$ und $\mathrm{S2}(j_1, j_2, ..., j_n)$ innerhalb von n gemeinsam umgebenden Schleifen auf, so wird die Anweisung S1 vor S2 ausgeführt falls gilt,

- $\vec{i} < \vec{j}$, oder
- $\vec{i} = \vec{j}$ und die Anweisung S1 geht der Anweisung S2 im Text voraus. Diese Beziehung wird als **Auswertungsreihenfolge** bezeichnet und wird wie folgt abgekürzt:

$$S1(i_1, i_2, ..., i_n) < S2(j_1, j_2, ..., j_n).$$

Eine andere Auswertungsreihenfolge, beispielsweise durch Schleifenvertauschung hervorgerufen, kann zur ursprünglichen Ausführungsform semantisch äquivalent sein. Welche Änderung semantisch äquivalent und damit gültig ist, bestimmen die Datenabhängigkeiten zwischen den Anweisungen in unterschiedlichen Iterationen, die sogenannten schleifengetragenen Datenabhängigkeiten.

Definition 4.9

Besteht die Datenabhängigkeit

$$\delta^*_{\mathrm{X}}\colon \mathrm{S1}(i_1, i_2, ..., i_n) \rightarrow \mathrm{S2}(j_1, j_2, ..., j_n),$$

dann definieren wir den **Distanzvektor**:

$$\vec{d} = (d_1, d_2, ..., d_n) = (j_1 - i_1, j_2 - i_2, ..., j_n - i_n).$$

Ist dieser Vektor lexikographisch positiv, so existiert eine **schleifengetragene Datenabhängigkeit** mit dem Distanzvektor $\vec{d}$ von Iteration $\vec{i}$ nach $\vec{j}$:

$$\delta^*(\vec{d})\text{: S1} \rightarrow \text{S2}.$$

Alle berechneten Distanzvektoren $\vec{d}$ der ursprünglichen (lexikographischen) Reihenfolge zeigen zeitlich gesehen nach vorn, d.h. die erste Nichtnull-Komponente von $\vec{d}$ ist positiv. Anderenfalls wird der Distanzvektor unter der Annahme $\vec{j} < \vec{i}$ berechnet. Da dies keine plausible Annahme ist, muß der berechnete Distanzvektor aus der weiteren Analyse ausgeschlossen werden. Nicht plausible Datenabhängigkeiten können im transformierten Programm-Code durch Vertauschen der Schleifen entstehen. In diesem Fall verletzt die Schleifentransformation die bestehenden Datenabhängigkeiten und darf daher nicht durchgeführt werden. Zur Erläuterung soll das Beispiel aus Abbildung 4.1 betrachtet werden.

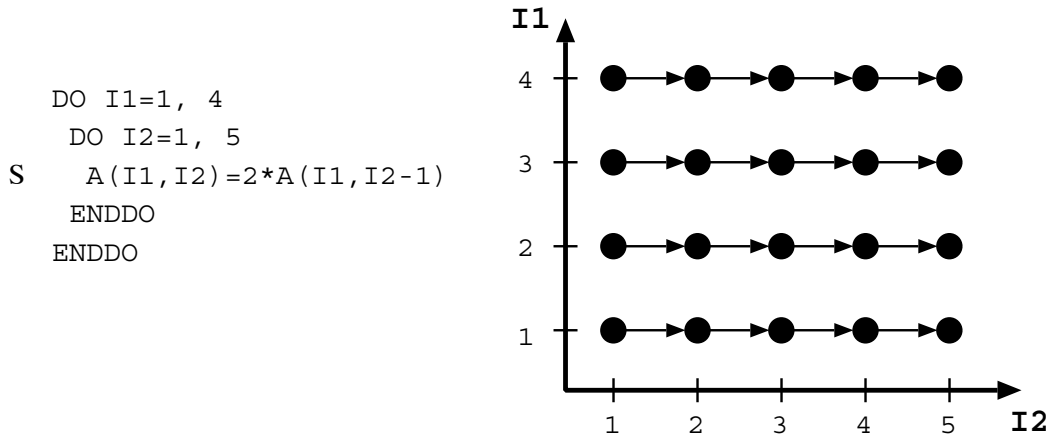


Abbildung 4.1: Darstellung der Datenabhängigkeiten für ein ausgewähltes Beispiel

Zur Berechnung der Datenabhängigkeit muß festgestellt werden, in welchen Iterationen die Anweisungen `A(I1,I2)` und `A(I1,I2-1)` das gleiche Feldelement von `A` referenzieren. Der durch die Anweisung S in Iteration (i_1, i_2) erzeugte Wert ist derselbe, der in Iteration (j_1, j_2) benutzt wird, wenn gilt

$$i_1 = j_1 \text{ und } i_2 = j_2 - 1.$$

Durch Umformung erhält man:

$$j_1 - i_1 = 0 \text{ und } j_2 - i_2 = 1.$$

Damit ist die Annahme $i < j$ bestätigt und für S besteht die durch die Schleife I2 getragene echte Datenabhängigkeit auf sich selbst mit dem Distanzvektor $\vec{d} = (0,1)$:

$$\delta_{\mathrm{A}}(0,1)\text{: S} \rightarrow \text{S}.$$

In dem betrachteten Beispiel gilt dieselbe Datenabhängigkeit für jedes Iterationspaar (i_1, i_2) und (j_1, j_2), für das zusätzlich die Bedingungen $1 \leq i_1, j_1 \leq 4$ und $1 \leq i_2, j_2 \leq 5$ erfüllt ist. Eine Transformation, die dazu führt, daß die Iteration (1,1) nach (1,2) durchlaufen wird, wäre demnach nicht zulässig. Rechnerisch erhielte man einen negativen Distanzvektor $\vec{d} = (0, -1)$.

In Abbildung 4.1 lassen sich alle bestehenden Datenabhängigkeiten durch den einzelnen Distanzvektor $\vec{d} = (0, 1)$ zusammenfassen. Daß dies nicht immer der Fall sein muß, zeigt das folgende Beispiel:

Beispiel 4.10

```
      DO I = 1, N
         DO J = 1, N
S1          C(I)   = 19
S2          A(I,J) = A(I+1,B(J))
         ENDDO
      ENDDO
```

Die Abhängigkeit für die Anweisung S1 lautet:

$$\delta^{O}_{C}(0,1)^{+}\text{: S1} \rightarrow \text{S1}.$$

Der hochgestellte Index '+' zeigt an, daß S1 nicht nur in (1,2) erneut erzeugt wird, sondern auch in (1,3), (1,4) usw. Damit existieren eine ganze Reihe von Abhängigkeiten mit unterschiedlicher Distanz: $\delta^{O}_{C}(0, k)$ für alle ganzzahligen k, die die Bedingung $1 \leq k < N$ erfüllen. Solche Abhängigkeiten werden zyklische Selbstabhängigkeiten genannt und sind sehr wichtig für die optimale Ausnutzung des Cache-Speichers, da sie ein hohes Maß an Wiederverwendung erlauben.

Bei der Bestimmung der Datenabhängigkeit für die Anweisung S2 verhindert die indirekte Adressierung durch `B(J)` eine Bestimmung des Distanzvektors, da `B(J)` sowohl größer als auch kleiner als J sein kann. In diesem Fall wird die kombinierte Darstellung benutzt:

$$\hat{\delta}_{A}(1, \pm)\text{: S2} \rightarrow \text{S2}.$$

Im obigen Fall kann das Vorzeichen der Distanz nicht festgestellt werden, so daß wir anstelle eines konkreten Wertes das Symbol $\pm$ benutzen: $\vec{d} = (1, \pm)$. Für `B(J)` > `J` ist $\vec{d} = (1, +)$, für `B(J)` < `J` ist $\vec{d} = (1, -)$.

4.2 Die Optimierungsoptionen des DEC FORTRAN-Compilers

Der DEC FORTRAN-Compiler wird mit dem Befehl f77 [parameter] progname [parameter] aufgerufen. Programmnamen mit der Endung .f, .for oder .FOR werden als FORTRAN77-Programme interpretiert und übersetzt [9]. Die Programme in dieser Arbeit wurden mit folgendem Compiler-Aufruf übersetzt:

f77 [-O<num>] [-g<num>] <progname.f> [-L DXML]

Die Parameter in den eckigen Klammern sind optional: Werden sie nicht angegeben, so wird der Default-Wert eingesetzt. Die Parameter haben folgende Bedeutung:

- **-O**<num>
  Angabe der Optimierungsstufe: Für <num> können die Werte von 0 bis 4 angegeben werden. Jede höhere Stufe schließt die Optimierungen der vorhergehenden ein:
  0: keine Optimierungen
  1: lokale Optimierungen
  2: globale Optimierungen
  3: Schleifenentfaltung für verbessertes Instruktions-Scheduling
  4: Alignment der Felder im Hauptspeicher (default)

- **-g**<num>
  Erzeugung von Debug-Informationen. Für <num> können die Werte 0 bis 3 angegeben werden:
  0: Kein Einfügen für Debug-Informationen im Objekt-Code:
  Diese Option wurde für die Laufzeitmessungen benutzt.
  1-3: Objekt-Code wird mit Debug-Informationen erweitert (default=1).

- <progname.f>
  Name des Programms, welches übersetzt werden soll.

- **-l** DXML
  Bei Benutzung von Routinen aus der Digital Extended Math Library für DEC OSF/1 AXP muß dieser Parameter mit angegeben werden.

Der Optimierungsprozeß beim DEC FORTRAN-Compiler besteht hauptsächlich aus den zwei Phasen Datenflußanalyse und Optimierung. Die Datenflußanalyse ist der Prozeß zur Sammlung von Informationen über die Art, wie Variablen in einem Programm benutzt werden[1]. Die gesammelten Informationen geben Aufschluß darüber, welche Optimierungen sicher sind und zu einer Verbesserung der Performance führen können. Die Strategie bei der Optimierung durch Compiler besteht in der Inkaufnahme höherer Übersetzungszeiten zur Erreichung kürzerer Laufzeiten. Die Forderung des Programmierers nach einer akzeptablen Wartezeit schränkt

jedoch die Optimierung ein, so daß eine optimale Performance des übersetzten Programms selten erreicht wird.

Direktiven im FORTRAN-Code zur Optimierung, wie etwa bei den vektorisierenden Compilern von der Firma Cray Research, werden vom DEC Compiler in der vorliegenden Version nicht unterstützt. Dagegen sind Direktiven im Assemblercode vorgesehen, mit denen der Programmierer beispielsweise die Lage von Feldern im Hauptspeicher beeinflussen kann. Da in dieser Arbeit das Alignment von Feldern auf Hochsprachenebene vorgenommen wird, sei an dieser Stelle auf die Systemdokumentation verwiesen [8].

4.2.1 Datenfluß- und Abhängigkeitsanalyse

Bei der Datenflußanalyse wird grundsätzlich angenommen, daß alle Referenzierungen eines Feldes dasselbe Element in diesem Feld adressieren, bis ein konstanter Ausdruck im Index benutzt wird. Folgende Datenelemente werden von der Datenanalyse ausgeschlossen, da keine Information über die zugewiesenen Werte ermittelt werden kann:

- Skalare Variable und Felder, die sowohl im aufrufenden als auch im aufgerufenen Unterprogramm vorkommen. Das bedeutet, daß der Datenfluß über Unterprogrammgrenzen hinweg nicht analysiert wird.
- Globale Variable, die in FORTRAN durch sogenannte COMMON-Blöcke definiert sind. Beim Aufruf eines Unterprogramms kann der Wert einer Variable geändert werden, so daß eine Aussage über den Wert der Variable nach dem Aufruf nicht möglich ist.

Nur eingegliederte Unterprogrammanweisungen (*inlining*) werden in der Datenflußanalyse berücksichtigt. Eingegliedert werden nur Unterprogramme, die nicht mehr als 20 Assembler-Instruktionen umfassen.

4.2.2 Lokale Optimierungen (-O1 bis -O4)

Lokale Optimierungen beziehen sich auf Instruktionen innerhalb eines Basisblockes. Ein Basisblock ist eine Folge fortlaufender Anweisungen ohne Verzweigungen im Kontrollfluß. Bei Schleifen ist das der Schleifenrumpf. Dies wird deutlich, wenn man die Übersetzung eines Schleifenkonstruktes in Assembler-Befehle betrachtet.

Beispiel 4.11

```
DO I = 1, 10
   ...
   ...
ENDDO
```

Bei der Übersetzung dieses Progammstücks wird folgender Assembler-Code generiert:

```
     mov   1, I        \* Schleifenzaehler initialisieren
L$1:
     ...
     ...
     addl  I, 1, I     \* Schleifenzaehler I um 1 erhoehen
     addq  I, -11, r5  \* Abfrage vorbereiten r5 = I-11
     blt   r5, L$1     \* Nach L$1 verzweigen, wenn I-11 < 0
```

Der Schleifenkopf mit der Abfrage $I \in \{1...10\}$ wird in eine bedingte Verzweigung am Ende des Basisblockes übersetzt. Trifft die Bedingung zu, so wird zum Anfang des Basisblockes verzweigt, der durch die Marke `L$1` gekennzeichnet ist. Alle lokalen Optimierungen beziehen sich nun auf diesen begrenzten Bereich.

Der DEC FORTRAN-Compiler sieht zahlreiche Optimierungen auf Instruktionsebene vor, die in [9] detailliert erläutert sind. Im weiteren Verlauf soll auf einige eingegangen werden.

Durchführung von Rechenoperationen zur Übersetzungszeit

Diese Optimierung beinhaltet die Berechnung von Anweisungen bestehend aus Variablen und konstanten Ausdrücken deren Werte zur Übersetzungszeit bekannt sind. Desweiteren werden zusammengesetzte logische Ausdrücke in IF-Anweisungen oder in Schleifenköpfen so übersetzt, daß die einfachen Bedingungen zuerst ausgewertet werden:

Beispiel 4.12

```
IF (A(I,J) .GT. B(M,N) .OR. X .GT. Y) GOTO 20
```

`X .GT. Y` wird zur Laufzeit des Programms zuerst ausgeführt, so daß der erste aufwendigere Vergleich gegebenenfalls entfallen kann. Eine Ausnahme bilden Aufrufe von Unterprogrammen in den logischen Ausdrücken, da deren Nichtausführung zu falschen Ergebnissen führen könnte.

Ersetzung von gemeinsamen Teilausdrücken

Wenn derselbe Teilausdruck in mehr als einer Anweisung auftritt und sein Wert sich nicht ändert, wird der Ausdruck einmal berechnet und im folgenden das Resultat benutzt. Besonders effizient ist diese Optimierung bei der Adreßberechung von Elementen mehrdimensionaler Felder:

Beispiel 4.13

```
DIMENSION A(25,25), B(25,25)
A(I,J) = B(I,J)
```

Ohne Optimierung werden die Indizes wie folgt berechnet:

```
T1 = (J-1) * 25 + I
T2 = (J-1) * 25 + I
A(T1) = B(T2)
```

Die Variablen `T1` und `T2` bezeichnen äquivalente Ausdrücke. Diese Redundanz wird folgendermaßen eliminiert:

```
T = (J-1) * 25 + I
A(T) = B(T)
```

Entfernung von überflüssigem Code

Wenn eine Variable erzeugt aber niemals benutzt wird, wird die gesamte Anweisung eliminiert. Dasselbe gilt für mehrfach generierte Variable, die zwischen der ersten und zweiten Erzeugung nicht benutzt werden.

Verbesserte Registerbelegung

Während in der Optimierungsstufe **-O0** alle Registerinhalte nach der Benutzung sofort in den Hauptspeicher zurückgeschrieben werden, wird ab der Stufe **-O1** eine effektivere Registerbelegung durchgeführt, die die Belegung von Registern mit Daten in folgender Rangfolge anstrebt:

1. Feldindizes
2. Variable
3. andere Datenreferenzierungen

Desweiteren werden komplexe Ausdrücke zuerst ausgewertet. Damit sinkt die Wahrscheinlichkeit, daß Registerinhalte im Hauptspeicher zwischengespeichert werden müssen.

4.2.3 Globale Optimierung (-O2 bis -O4)

Globale Optimierungen realisieren Veränderungen der Ausführungsreihenfolge auch über Basisblockgrenzen hinweg. Erst mit der Option **-O2** sind signifikant bessere Performance-Werte zu erzielen, da die internen Architekturmerkmale des Alpha-Chip ausgenutzt werden.

Belegung von Registern mit schleifeninvarianten Werten
Um mehrfache identische Zuweisungen innerhalb von Schleifen zu vermeiden, werden invariante Ausdrücke außerhalb der Schleifen einem Register zugewiesen und müssen damit während der Durchläufe nicht mehr nachgeladen werden. Diese Optimierung ist nicht mehr möglich, wenn alle Register für Berechnungen innerhalb der Schleife benutzt werden.

Auflösung von Eingabe- und Antiabhängigkeiten

Antiabhängigkeiten entstehen durch die oft benutzte Programmiertechnik, dieselben Variablen mehrfach zu verwenden. Unter Berücksichtigung der Lebenszeit dieser Variablen können die dadurch entstandenen Datenabhängigkeiten aufgelöst werden, so daß ein effektiveres Instruktions-Scheduling durchgeführt werden kann. Die Lebenszeit einer Variablen beginnt bei der ersten Erzeugung und endet, wenn alle Berechnungen, die diesen Wert der Variablen benutzen, beendet sind. Die Lebenszeit wird als Information zur effektiven Registerbelegung benutzt, da ein Verbleiben von nicht mehr benutzten Variablen in den Registern die zur Verfügung stehende Registeranzahl unnötig vermindern würde.

Beispiel 4.14

```
      DO I = 1 ,10
S1      X = 3 + I
S2      Y = Y + X
S3      X = 4 * I
S4      A = A + X
      ENDDO
```

In Beispiel 4.14 kann die zweite Erzeugung von `X` durch Umbenennung vermieden werden:

```
      DO I = 1 ,10
S1      X = 3 + I
S2      Y = Y + X
S3      B = 4 * I
S4      A = A + B
      ENDDO
```

Mit der Umbenennung werden die Abhängigkeiten $\hat{\delta}_X$: S2 $\rightarrow$ S3 und δ^O_X: S1 $\rightarrow$ S3 aufgehoben, so daß die Anweisungspaare parallel ausgeführt werden können. Damit ist eine Code-Verschiebung möglich, mit der eine effektivere Auslastung der Pipelines erreicht werden kann[1].

Lebenszeiten von Variablen, die sich über mehrere Iterationen erstrecken, werden vom DEC FORTRAN-Compiler nicht erkannt. Forschungen auf diesem Gebiet [5, 6] haben die Effektivität dieser Registerbelegungsstrategie aufgezeigt. Eine Randbedingung ist dabei die begrenzte Registeranzahl, da die Zwischenspeicherung einer Variablen, beispielsweise über drei Iterationen hinweg, drei zusätzliche Register benötigt. Das von Callahan, Carr und Kennedy in [5] beschriebene Verfahren sieht einen durch Register gebildeten FIFO-Stack vor. Die Anzahl der Register richtet sich nach den Iterationen, die zwischen Erzeugung und Wiederverwendung der Variablen liegen. In jeder Iteration wird der neu berechnete Wert in den Stack geschoben und das Element entnommen, welches vor der entsprechenden Iterationsanzahl erzeugt worden ist und nun benutzt werden kann, ohne daß es aus dem Hauptspeicher geladen werden muß.

Ersetzung von Induktionsvariablen und Senkung der Operationskosten

In Programmen werden Feldreferenzen mit einem Vektorinkrement ungleich 1 in der Regel durch sogenannte Induktionsvariable realisiert, die durch Linearkombination von Schleifenvariable und Konstante gebildet werden. Im Beispiel 4.15 wird auf das Feld `A` in umgekehrter Reihenfolge mit der Schrittweite 2 zugegriffen: `A(20), A(18), ..., A(2)`.

Beispiel 4.15:

```
      J = 11
      DO I = 1, 10
S1      J = J - 1
S2      T = 2 * J
S3      A(T) = 0
       ENDDO
```

Da nach der Anweisung S2 feststeht, daß $T = 2 \cdot J$ ist und T auch an keiner anderen Stelle des Basisblockes geändert wird, kann vereinfacht werden: $T = 2 \cdot J$ wird zu $T = 2 \cdot (J - 1)$ und damit $T = 2 \cdot J - 2 = T - 2$. Da `T` bei der ersten Iteration noch nicht definiert ist, muß die Anweisung `T = 2 * J` vor der Schleife eingefügt werden:

[1] Beim untersuchten Alpha-Chip müssen dafür die Variablen Fließpunktformat haben, da die Integer-Multiplikation nicht in einer Pipeline ausgeführt wird und somit nicht parallel zur Addition erfolgen kann.

```
      J = 11
S2    T = 2 * J
      DO I = 1, 10
S1       T = T - 2
S3       A(T) = 0
      ENDDO
```

Die Nichtausführung einer Integer-Multiplikation bedeutet beim Alpha-Chip eine Einsparung von 21 Takten.

Instruktions-Scheduling

Die bisher betrachteten Optimierungsmethoden verkürzen die Ausführungszeit eines Programms durch die Einsparung von Instruktionen. In Abbildung 2.1 ist darüberhinaus der Faktor $\frac{Taktanzahl}{Instruktionen}$ angeführt, auf den durch Compiler-Optimierung Einfluß genommen werden kann. In Kapitel 2 wurde beschrieben, welche Performance-Steigerung durch die Implementation einer Pipeline-Struktur erreicht werden kann. Wie effektiv diese Hardware-Struktur ausgenutzt wird, ist bestimmt durch die Art und Weise, in der die Assembler-Instruktionen den Funktionseinheiten zugeführt werden. Superskalare Prozessoren, wie etwa der untersuchte Alpha-Chip, erreichen ihre hohen Performance-Werte durch das Ausnutzen des Parallelismus auf Instruktionsebene (*Instruction-Level Parallelism, ILP*). Das Instruktions-Scheduling zielt darauf ab, eine möglichst effiziente Instruktionsreihenfolge zu generieren[29]. Zur Verdeutlichung betrachten wir folgendes Beispiel:

Beispiel 4.16

```
      DO J = 1, N
         DO I = 1, N
S           A(I) = A(I) + B(J)
         ENDDO
      ENDDO
```

Diesen Quelltext übersetzt der DEC FORTRAN-Compiler bei Angabe der Optimierungsstufe **-O2** in folgende Instruktionssequenz:

```
L$8:
1a    ldt    f1, (r4)       \* A(i) in Register f1 laden
1b    lda    r4, 8(r4)      \* Adresse von A(i+1) laden
2a    addl   I, 1, I        \* Schleifenzaehler I um 1 erhoehen
2b    cmple  I, r18, r3     \* I <= N? Ja: r3=1, nein: r3=0
3a    addt   f1, f0, f1     \* A(i) = A(i) + B(j)
3b    stt    f1, -8(r4)     \* A(i) abspeichern
4a    bne    r3, L$8        \* Schleife wird erneut durchlaufen,
                            \* wenn r3 ungleich 0 (I <= N)
```

Die in der ersten Spalte angegebene Numerierung kennzeichnet die parallel eingelesenen Instruktionspaare. Abbildung 4.2 stellt die Ausführungsreihenfolge unter Beachtung der Scheduling-Regeln aus Kapitel 3 dar.

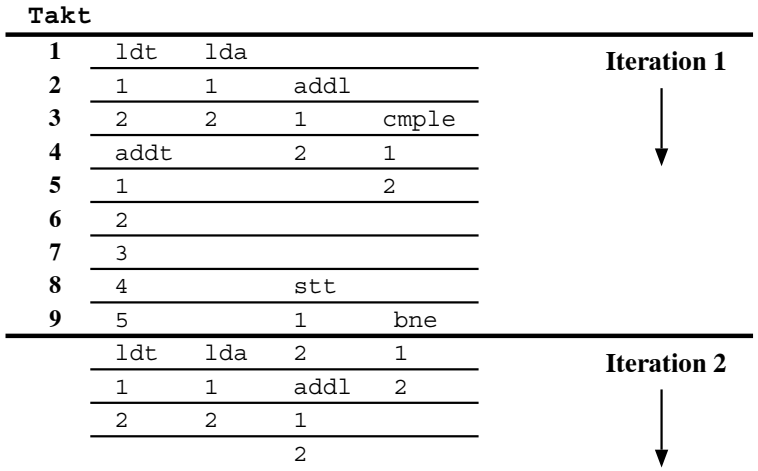


Abbildung 4.2: Beispiel 4.16: Pipelined-Ausführung des Assemblercodes, Optimierungsstufe **-O2**

Die Instruktionen 1a und 1b können parallel an die entsprechenden Funktionseinheiten weitergeleitet werden, da der `lda`- Befehl in der Integer-Funktionseinheit ausgeführt wird [10]. Da zwischen den Instruktionen 2a und 2b eine Erzeuger/Benutzer-Abhängigkeit besteht, werden sie sequentialisiert. Nach den Ausgaberegeln in Abbildung 3.8 kann der `cmple`-Befehl jedoch bereits einen Takt nach dem `addl`-Befehl in der Integer-Funktionseinheit ausgeführt werden. Die Konstante `N` wird außerhalb der beiden Schleifen in das Integer-Register r18 geladen.

Der Befehl 3a kann nicht parallel mit 3b weitergeleitet werden, da bezüglich des Fließpunktregisters `f1` eine Erzeuger/Benutzer Abhängigkeit besteht. `B(J)` wird außerhalb der `I`-Schleife in das Fließpunktregister `f0` geladen, da `J` in der inneren Schleife invariant ist. Durch den *Bypass-Kanal* kann der `stt`-Befehl im vierten Takt nach Initiierung des `addt`-Befehls an die Adreß-Einheit weitergeleitet werden. Einen Takt später wird der `bne`-Befehl in die Integer-Funktionseinheit weitergeleitet. Die Verzweigungsvorhersage in Stufe 2 der Leitwerk-Pipeline hat die Durchführung des rückwärtsgerichteten Sprungs `bne` vorausgesagt, so daß der Programmzähler der Prefetch-Einheit geändert werden muß. Das schon in der ersten Stufe der Leitwerk-Pipeline befindliche Instruktionspaar 5a, 5b und der nicht auszuführende Befehl 4b werden gelöscht, so daß eine Pipeline-Blase entsteht. Diese wird jedoch beseitigt, da die Pipeline durch den Registerkonflikt zwischen den Instruktionen 3a und 3b für mehrere Takte stillsteht. Insgesamt werden für einen Schleifendurchlauf 9 Takte benötigt.

Das Prinzip des *Software-Pipelining* erkennt man an der Anweisung `lda r4, 8(r4)`: Die Klammer steht für den Inhalt des Registers r4, die vorstehende '8' bedeutet eine Addition von 8 Byte, da 64-Bit-Fließpunktwerte benutzt werden. Mit dem Befehl wird die Adresse von `A(I)`, die vor der `I`-Schleife in `r4` geladen wurde, mit der Adresse von `A(I+1)` überschrieben. Dieser Wert wird jedoch erst in der nächsten Iteration `I+1` verwendet, um dann das Register f1 mit dem Wert von `A(I+1)` zu laden. Das Software-Pipelining ermöglicht also das Ausführen von Instruktionen, die eigentlich anderen Iterationen zugeordnet sind. Dies hat den Vorteil, daß die Ladeanweisung `ldt f1, (r4)` in der nächsten Iteration nicht auf die Adresse von `A(I+1)` warten muß. Damit wird auch verständlich, warum die Speicherinstruktion in 4a nicht die Adresse `(r4)`, sondern die Adresse `-8(r4)` benutzen muß, um `A(I)` abzuspeichern. Es stellt sich nun die Frage, warum nur das Laden der Adresse von `A(I+1)` durchgeführt wird und beispielsweise nicht der `stt`-Befehl für `A(I-1)`. In diesem Fall könnte nämlich die Verzögerung von Takt 5 bis Takt 8 vermieden werden. Die Antwort hierauf liegt in der Optimierungsstrategie des DEC FORTRAN-Compilers begründet: Anstelle des Software-Pipelining wird für die bessere Ausnutzung der Pipeline-Struktur die innere Schleife mehrmals ausprogrammiert. Dadurch vergrößert sich der Basisblock der inneren Schleife, und die nun zur Verfügung stehenden Instruktionen ermöglichen ein besseres Instruktions-Scheduling. Dieses Vorgehen wird im DEC Compiler-Handbuch [9] als zusätzliche globale Optimierung beschrieben und wird in den Optimierungsstufen **-O3** und **-O4** angewendet. Bevor hierauf im nächsten Abschnitt eingegangen wird, ist festzuhalten, daß bereits mit der Optimierungsstufe **-O2** das Software-Pipelining und das Instruktions-Scheduling durchgeführt wird. Dies läßt erwarten, daß auch manuell vom Programmierer durchgeführte Schleifentransformationen, die eine Vergrößerung des Basisblockes bewirken, mit der Optimierungsstufe **-O2** übersetzt zu erheblichen Performance-Verbesserungen führen können.

4.2.4 Zusätzliche globale Optimierungen (-O3 und -O4)

Die zusätzlichen globalen Optimierungen beinhalten eine Vergrößerung des Basisblockes, was ein verbessertes Instruktions-Scheduling ermöglicht. Mit der Option **-O3** oder **-O4** wird das Programmstück aus Beispiel 4.16 in folgende Assembler-Befehle übersetzt:

```
L$11:
1a      ldt   f1, (r5)        \* A(I) laden
1b      addl  I, 4, I         \* Schleifenzaehler um 4 erhoehen
2a      ldt   f10, 8(r5)      \* A(i+1) laden
2b      cmple I, r3, r6       \* I <= N-3? ja: r6=1, nein: r6=0
3a      ldt   f11, 16(r5)     \* A(I+2) laden
3b      cmple I, r18, r7      \* I <= N? ja: r7=1, nein: r7=0
4a      addt  f1, f0, f1      \* A(I) = A(I) + B(J)
```

```
4b      ldt   f12, 24(r5)     \* A(I+3) laden
5a      addt  f10, f0, f11    \* A(I+1) = A(I+1) + B(J)
5b      addt  f11, f0, f11    \* A(I+2) = A(I+2) + B(J)
6a      addt  f12, f0, f12    \* A(I+3) = A(I+3) + B(J)
6b      stt   f1, (r5)        \* A(I) speichern
7a      stt   f10, 8(r5)      \* A(I+1) speichern
7b      stt   f11, 16(r5)     \* A(I+2) speichern
8a      lda   r5, 32(r5)      \* Adresse von A(I+4) laden
8b      stt   f12, -8(r5)     \* A(I+3) speichern
9a      bne   r6, L$11        \* Verzweigen nach L$11, wenn I<=N-3
9b      beq   r7, L$7         \* Verlasse innere Schleife, wenn I>N,
                              \* sonst Korrekturschleife ausfuehren

                              \* Korrekturschleife, wenn N nicht
L$14:                         \* ganzzahlig durch 4 teilbar
10a     ldt   f1, (r5)        \* A(I) laden
10b     lda   r5, 8(r5)       \* Adresse von A(I+1) laden
11a     addl  I, 1, I         \* Schleifenzaehler um 1 erhoehen
11b     cmple I, r18, r7      \* I <= N? ja: r7=1, nein: r7=0
12a     addt  f1, f0, f1      \* A(I) = A(I) + B(J)
12b     stt   f1, -8(r5)      \* A(I) speichern
13a     bne   r7, L$14        \* Verzweigen nach L$14, wenn I<=N
13b     nop                   \*$
```

Dasselbe Ergebnis erhält man bei der Übersetzung des folgenden Programmstücks mit der Optimierungsoption **-O2**:

```
DO J = 1, N
   DO I = 1, N-3, 4            \* innere Schleife vierfach
      A(I)   = A(I)   + B(J)   \* entrollt
      A(I+1) = A(I+1) + B(J)
      A(I+2) = A(I+2) + B(J)
      A(I+3) = A(I+3) + B(J)
   ENDDO
   DO I = I, N                 \* Korrekturschleife, wenn
      A(I) = A(I) + B(J)       \* MOD(N, 4) ungleich 0
   ENDDO
ENDDO
```

Zur Vergrößerung des Basisblockes wird die innere Schleife vierfach entfaltet. Da in diesem Fall zur Übersetzungszeit nicht festgestellt wird welchen Wert `N` annimmt, muß für den Fall, daß `N` nicht ganzzahlig durch vier teilbar ist, eine Korrekturschleife angefügt werden, die die restlichen Iterationen durchführt (*clean-up*). In Abbildung 4.3 ist die Ausführung der Instruktionen dargestellt, die die erreichte

Performance-Steigerung verdeutlicht: Vernachlässigt man die Korrekturschleife und die Instruktionen außerhalb des Basisblockes, so werden für vier Iterationen nur 16 Takte benötigt. Da die Anweisung S in Beispiel 4.16 N^2 mal ausgeführt wird, hat die Ausführungszeit der Instruktionen außerhalb des Schleifenkonstrukts bei den betrachteten Problemgrößen von `N` > 100 keinen Einfluß auf die gemessene Performance. Aus der Abbildung 4.3 ergibt sich folgende Vorhersage: Auf 16 Takte kommen 4 Fließpunktoperationen, was auf der Alpha-Workstation Modell 300 (150 MHz) zu einer theoretischen Performance von $\frac{4}{15} \cdot 150$ MFLOPS $= 40$ MFLOPS führt. Verglichen mit $\frac{1}{9} \cdot 150$ MFLOPS $= 16,7$ MFLOPS für die ursprüngliche Form aus Abbildung 4.2 bedeutet dies eine Steigerung um mehr als das Doppelte. Die tatsächlich gemessene Performance des Programmstücks aus Beispiel 4.16 mit den verschiedenen Optimierungsoptionen ist in Abbildung 4.4 dargestellt.

Takt							
1	ldt i	addl					Iteration 1 - 4
2	1	1	ldt i+1	cmple			
3	2	2	1	1	ldt i+2	cmple	
4	addt i	ldt i+3	2	2	1	1	
5	1	1	addt i+1		2	2	
6	2	2	1	addt i+2			
7	3	addt i+3	2	1			
8	4	1	3	2	stt i		
9	5	2	4	3	1	stt i+1	
10	stt i+2	3	5	4	2	1	
11	1	4	lda	5		2	
12	2	5	1				
13			2				
14	stt i+3						
15	1	bne					
	2	1	ldt i+4	addl			Iteration 5 - 8
		2	1	1			
			2	2			

Abbildung 4.3: Beispiel 4.16: Pipelined-Ausführung des Assemblercodes, Optimierungsoption **-O3**

Es fällt auf, daß im Gegensatz zur nichtentfalteten Version (**-O2**), die erwartete Leistung von 40 MFLOPS nicht erreicht wird. Dieser Sachverhalt kann darauf zurückgeführt werden, daß die angenommene Verzögerungszeit von zwei Takten für Ladeinstruktionen zu klein gewählt ist. Die Messungen auf der mit 133 MHz getakteten Alpha-Workstation Modell 400, deren Ergebnis in Abbildung 4.5 dargestellt ist, stützen diese Aussage: Anstelle der 40 MFLOPS aus der obigen Berechnung ist in diesem Fall eine geringere Performance von $\frac{4}{15} \cdot 133$ MFLOPS $= 35.5$ MFLOPS zu erwarten. Trotzdem werden größere Werte als im obigen Fall gemessen, was auf eine kleinere Speicherlatenzzeit durch den breiteren Datenbus dieses Rechners hinweist.

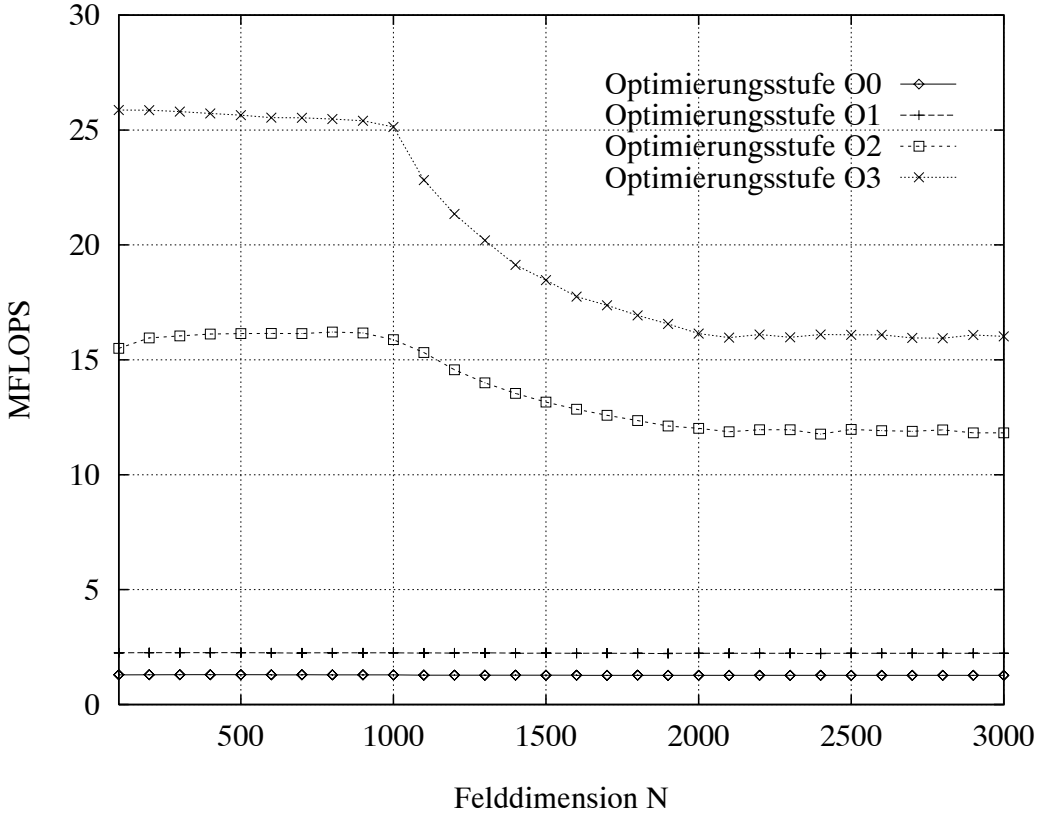


Abbildung 4.4: Performance-Messung, Beispiel 4.16: Alpha-Workstation, Modell 300 (150 MHz)

Ab der Problemgröße $N > 1000$ kann der der interne Daten-Cache-Speicher nicht mehr alle referenzierten Werte von `A(I)` zwischenspeichern. Dies hat zur Folge, daß aufgrund der Zeilenlänge des Cache-Speichers jeder vierte Zugriff auf das Feld `A(I)` eine Referenzierung des externen Speichersystems notwendig macht. Bei den betrachteten Problemgrößen sind die Daten im externen Cache-Speicher enthalten, was zu einer zusätzlichen Verzögerungszeit von sechs Takten für jeweils eine von vier Fließpunkt-Ladeinstruktionen führt.

Vergleicht man die Messungen für die verschiedenen Workstation-Modelle miteinander, so führt im Fall der nicht entfalteten Version (**-O2**) bei Problemgrößen N < 2000 die schnellere Taktrate zu besseren Performance-Werten. Oberhalb dieses Wertes gleicht die kürzere Speicherlatenzzeit diesen Laufzeitvorteil aus, und es werden nahezu identische Laufzeiten gemessen. Bei der Betrachtung der entfalteten Version ist die kürzere Speicherlatenzzeit bestimmend für das Laufzeitverhalten, so daß für alle Problemgrößen auf dem Rechner mit der niedrigeren Taktfrequenz, aber breiterer Datenbusbreite, bessere Performance-Ergebnisse gemessen werden. Das bedeutet, daß insbesondere für Programmversionen, die eine bessere Auslastung der internen Pipeline-Struktur bewirken, eine größere Speicherlatenzzeit erheblichen Einfluß auf die erreichte Performance hat. Da der DEC FORTRAN-Compiler keine Programmtransformationen unterstützt, die die eine verbesserte Ausnutzung

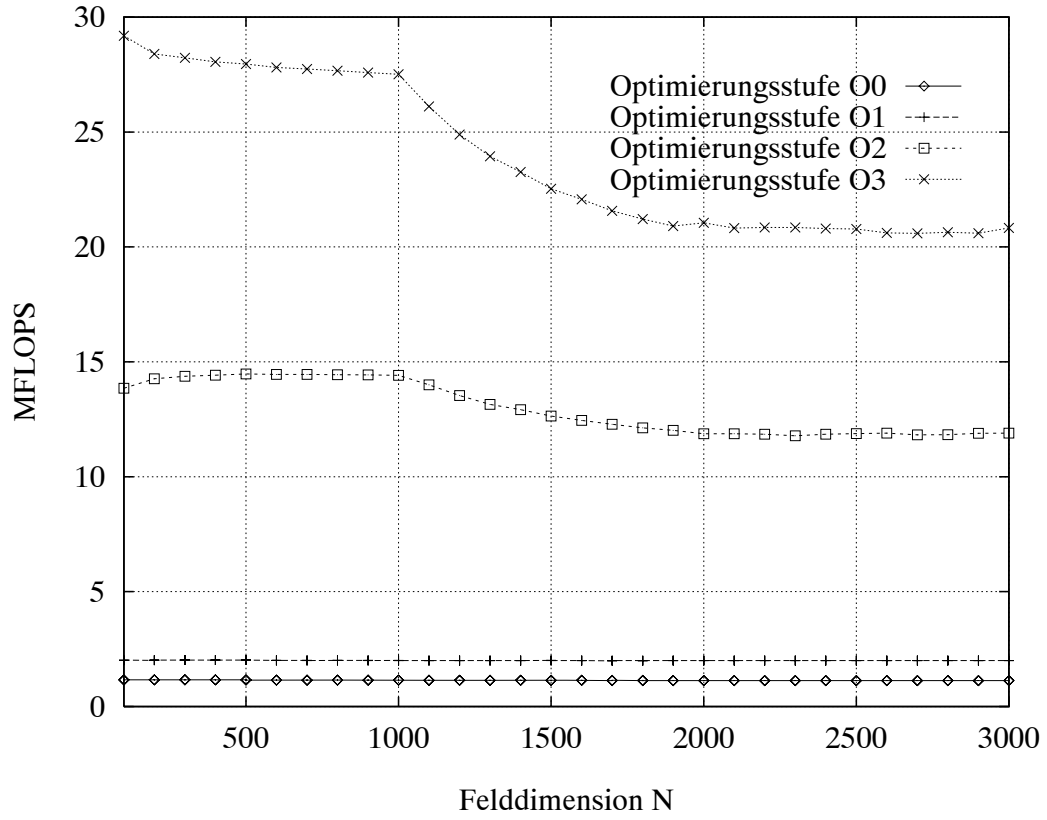


Abbildung 4.5: Performance-Messung, Beispiel 4.16: DEC Alpha-Workstation, Modell 400 (133 MHz)

der Datenlokalität im Cache-Speicher ermöglichen, ist die durchgeführte Entfaltung in den Optimierungsstufen **-O3** und **-O4** für Probleme mit großen Datenfeldern und großer Wiederverwendung nicht ausreichend.

Die Anzahl der ausprogrammierten Iterationen richtet sich nach der Differenz zwischen Start- und Endwert der inneren Schleife. Um die Ausführung der Korrekturschleife zu vermeiden, muß der Entfaltungswert ein Vielfaches dieser Differenz bilden. Der DEC FORTRAN-Compiler führt entweder eine vier- oder fünffache Entfaltung der inneren Schleife durch. Es wird jedoch nur dann eine fünffache Entfaltung vorgenommen, wenn die Anzahl der Iterationen ganzzahlig durch fünf und nicht durch vier teilbar ist[2]. Kann die obere Grenze durch die Datenflußanalyse nicht bestimmt werden, so wird ebenfalls eine vierfache Entfaltung vorgenommen und die Korrekturschleife angefügt. Für die Laufzeitverbesserung von Programmen sollte daher, wenn möglich, ein konkreter Wert für die obere Grenze einer Schleife angegeben werden. Da die Datenflußanalyse nicht über Unterprogrammgrenzen hinausgeht, muß die Definition dieses Wertes in dem Programmteil erfolgen, in dem auch die Schleife enthalten ist. Eine Parameterangabe im Hauptteil des Programms ist nicht ausreichend. Das Auftreten von Kontrollflußabhängigkeiten innerhalb des Schlei-

[2]So wird beispielsweise eine Schleife mit 20 Iterationen vierfach entrollt.

fenrumpfes (z.B. IF-Anweisungen) verhindert generell die automatische Entfaltung. Neben dem besseren Instruktions-Scheduling wird im folgenden Beispiel 4.17 durch die vollständige Entfaltung der inneren Schleife auch die Registerbelegung verbessert.

Beispiel 4.17

```
DO I = 1, N
   DO J = 1, 5
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

Nach der Ausprogrammierung erkennt der Compiler, daß die Indizes der Feldreferenzen `B(J)`,...,`B(J+4)` bezüglich der `I`-Schleife konstant sind und somit außerhalb der Schleife geladen werden können. Dieses vorzeitige Laden wird vom Compiler jedoch nur für invariante Feldreferenzen auf der rechten Seite von Zuweisungen vorgenommen. Übersetzt man beispielsweise das Programmstück aus Beispiel 4.17a, so wird auch in diesem Fall die innere Schleife vollständig entfaltet, die Inhalte der Feldelemente `A(I)`,...,`A(I+4)` jedoch nicht außerhalb der `J`-Schleife, sondern in jeder Iteration in die Register geladen und in den Speicher geschrieben werden.

Beispiel 4.17a

```
DO J = 1, N
   DO I = 1, 5
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

Diese unnötigen Schreibbefehle können durch den Programmierer durch das sogenannte *scalar replacement* vermieden werden. Darunter versteht man die Zuweisung von invarianten Feldreferenzen auf skalare Variable, was vor der Schleife durchgeführt werden muß. Einen entsprechend modifizierter Code ist in Beispiel 4.17b angegeben.

Beispiel 4.17b

```
A1 = A(1)
A2 = A(2)
A3 = A(3)
A3 = A(4)
A5 = A(5)
DO J = 1, N
```

```
      A1 = A1 + B(J)
      A2 = A2 + B(J)
      A3 = A3 + B(J)
      A4 = A4 + B(J)
      A5 = A5 + B(J)
   ENDDO
   A(1) = A1
   A(2) = A2
   A(3) = A3
   A(4) = A4
   A(5) = A5
```

Durch die Zuweisung weist der Compiler den Feldelementen `A(I)`,...,`A(I+4)` Register zu, so daß die Schreib- und Ladebefehle innerhalb des Basisblocks entfallen. Nach Durchlauf der Schleife müssen die Registerinhalte in den Hauptspeicher zurückgeschrieben werden. Durch die programmierten Zuweisungen werden keine zusätzlichen Assembler-Befehle generiert, sondern die Lade- und Speicherbefehle aus dem Basisblock der Schleife hinausgeschoben.

Da im Kapitel 6 auch mehrdimensionale Felder betrachtet werden, soll an dieser Stelle noch die Auswirkung der Indexvariablen auf das Instruktions-Scheduling beschrieben werden.

Beispiel 4.18:

```
DO J = 1, N
   DO I = 1, N
      A(I,J) = A(I,J) + B(J)
   ENDDO
ENDDO
```

Übersetzt man das Programmstück aus Beispiel 4.18 mit den Optionen **-O3** oder **-O4**, so erhält man für den Basisblock den identischen Assembler-Code wie schon aus dem vorhergehenden Beispiel. Da in FORTRAN Felder spaltenweise abgelegt werden, werden durch die gegebene Iterationsfolge Feldelemente referenziert, die sequentiell im Hauptspeicher abgelegt sind. Damit erfolgt die Adreßberechnung innerhalb des Schleifenrumpfes in der gleichen Weise wie bei eindimensionalen Feldern.

Vertauscht man jedoch die Schleifen und ändert die Feldreferenz `B(J)` nach `B(I)`, damit dieses Element wiederum außerhalb der inneren Schleife in ein Register geladen werden kann, so wird mit dem Vektorinkrement N auf das Feld `A(I,J)` zugegriffen. Der Compiler erkennt dies und generiert einen anderen Assembler-Code:

```
L$23:
1a    ldt     f1, (r7)      \* A(I,J) laden
1b    addl    J, 4, J       \* Schleifenzaehler um 4 erhoehen
2a    cmple   J, r8, r19    \* J <= N-3? Ja: r19=1, nein: r19=0
2b    cmple   J, r18, r20   \* J <= N? Ja: r20=1, nein: r20=0
3a    addt    f1, f0, f1    \* A(I,J) = A(I,J) + B(I)
3b    stt     f1, (r7)      \* A(I,J) speichern
4a    addq    r7, r6, r7    \* Adresse von A(I,J+1) berechnen
4b    ldt     f1, (r7)      \* A(I,J+1) laden
5a    addt    f1, f0, f1    \* A(I,J+1) = A(I,J+1) + B(I)
5b    stt     f1, (r7)      \* A(I,J+1) speichern
6a    addq    r7, r6, r7    \* Adresse von A(I,J+2) berechnen
6b    ldt     f1, (r7)      \* A(I,J+2) laden
7a    addt    f1, f0, f1    \* A(I,J+2) = A(I,J+2) + B(I)
7b    stt     f1, (r7)      \* A(I,J+2) speichern
8a    addq    r7, r6, r7    \* Adresse von A(I,J+3) berechnen
8b    ldt     f1, (r7)      \* A(I,J+3) laden
9a    addt    f1, f0, f1    \* A(I,J+3) = A(i+3) + B(I)
9b    stt     f1, (r7)      \* A(I,J+3) speichern
10a   addq    r7, r6, r7    \* Adresse von A(I,J+4) berechnen
10b   bne     r19, L$23     \* Verzweigen nach L$23, wenn J<=N-3
11a   beq     r20, L$19     \* Verlasse innere Schleife, wenn j>N,
                            \* sonst Korrekturschleife ausfuehren
```

Die Korrekturschleife ist, bis auf die Berechnung der Adresse, identisch mit dem vorhergehenden Beispiel und ist daher nicht angegeben. Die Adreßberechnung durch die Addition des konstanten Wertes N erhöht die Instruktionsanzahl um drei. Desweiteren werden die Anweisungen nicht überlappt sondern sequentiell ausgeführt, was zu einem erheblichen Anstieg der benötigten Taktanzahl führt. Der Compiler führt dieses Scheduling durch, weil anzunehmen ist, daß die einzelnen Elemente `A(I,J)`, `A(I,J+1)` usw. ausschließlich aus dem externen Speichersystem geladen werden müssen. Da die Adreßfunktionseinheit bei ausstehenden Ladeoperationen keine weiteren Befehle annimmt und jeder weitere Befehl ebenfalls zu einer Blockierung führt, ist keine Performance-Steigerung durch das Instruktions-Scheduling nach Beispiel 4.16 zu erzielen. Aus diesem Grund ist die spaltenweise Referenzierung von mehrdimensionalen Feldern anzustreben.

4.2.5 Data-Alignment (-O4)

Manche Programme referenzieren Datenfelder in der Art, daß bei ungünstiger Ausrichtung der Felder im Hauptspeicher eine Häufung von Lesefehlern im Cache-Speicher auftritt. Dieses Problem kann durch effizientes Ausrichten der Felder zur Übersetzungszeit gelöst werden. Zur Erläuterung soll folgendes Beispiel betrachtet werden:

Beispiel 4.19

```
REAL*8 A(512), C(512), B(512)
DO I = 1, 512
   A(I) = A(I) + B(I)
ENDDO
```

Die eindimensionalen Felder `A`, `C` und `B` stehen hintereinander im Hauptspeicher und sind jeweils 4 KByte groß, so daß bei sequentieller Lage im Hauptspeicher, die Ausdrücke `A(I)` und `B(I)` in derselben Iteration Feldelemente referenzieren, die exakt 8 KByte weit auseinander liegen. Bei der Ausführung dieses Programmstücks ist bei der Übersetzung mit niedrigeren Optimierungsoptionen als **-O4** eine starke Erhöhung der Lesefehler im internen Cache-Speicher zu beobachten, da die Referenzierungen von `A(I)` und `B(I)` dieselbe Cache-Speicherzeile adressieren und damit `B(I)` in jeder Iteration aus dem externen Speicher geladen werden muß.

Eine Möglichkeit zur Vermeidung dieses Problems ist die versetzte Ausrichtung der Felder um 32 Byte, was der Länge einer Cache-Speicherzeile entspricht. In diesem Fall können dann jeweils vier Elemente der beiden Felder gleichzeitig im Cache-Speicher gehalten werden. Die beste Ausrichtung erhält man, wenn die Felder A(I) und B(I) direkt hintereinander oder auf verschiedenen Hauptspeicherseiten mit einem Abstand von $n \cdot 4$ KByte (n ungerade) liegen.

Bei der Benutzung der Optimierungsstufe **-O4** werden die Felder `A`, `C` und `B` (wie in Abbildung 4.6 dargestellt) ausgerichtet. Im Gegensatz zur Option **-O4** werden die Datenfelder bei allen anderen Optimierungsstufen sequentiell im Hauptspeicher abgelegt. Bei mehrfachem Durchlaufen der Schleife aus Beispiel 4.18 ergab sich gegenüber der nichtoptimierten Version eine Leistungssteigerung um den Faktor 7.3, welches die Wichtigkeit einer effektiven Ausrichtung von Datenfeldern im Hauptspeicher hervorhebt.

Werden COMMON-Blöcke verwendet und der Parameter **-align dcommons** angegeben, so wird das erste Feld eines Blockes auf eine 32-Byte-Grenze ausgerichtet und die folgenden Felder dieses COMMON-Blockes sequentiell im Hauptspeicher abgelegt. Durch die Anwendung dieser FORTRAN-Befehle kann auch durch den Programmierer ein effizientes Ausrichten der Felder vorgenommen werden. Dies ist von Interesse, weil keine Informationen über die Strategie des Compilers beim Ausrichten der Felder vorliegen und Informationen über die vorgenommenen Ausrichtungen nur mühsam mit Hilfe des Debuggers ermittelt werden können. Das obige Programm würde bei Benutzung von COMMON-Blöcken wie folgt modifiziert werden:

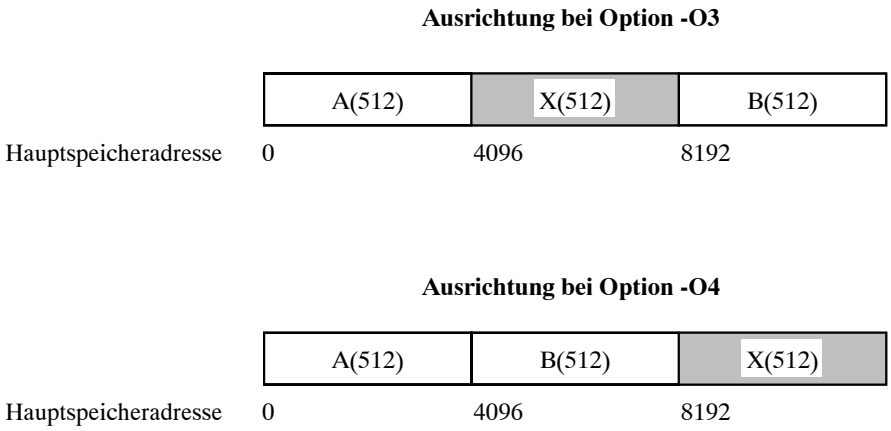


Abbildung 4.6: Beispiel 4.19: Verminderung von Lesefehlern im Cache-Speicher durch Änderung der Feldausrichtung im Hauptspeicher

Beispiel 4.20

```
REAL*8   A(512), C(512), B(512)
COMMON   /COM1/ A(512), B(512), C(512)

      DO I = 1, 512
         A(I) = A(I) + B(I)
      ENDDO
```

In diesem Fall werden bei der Übersetzung mit der Option **-O3** bzw. **-O4** dieselben Werte gemessen. Die Angabe der Option **-O2**, gekoppelt mit dem manuellen vierfachen Entfalten der inneren Schleife, führt ebenfalls zu diesen guten Performance-Werten. Bei dem gegebenen Beispiel ist jedoch zu beachten, daß nun A gegenüber `C` so ausgerichtet ist wie vorher A und B. Existiert eine entsprechende Schleife mit der Referenzierung von A und `C`, so geht der erreichte Performance-Gewinn wieder verloren. Solche Seiteneffekte sind bei der Einflußnahme durch den Programmierer zu berücksichtigen.

Eine zweite häufige Ursache für das massive Auftreten von Cache-Speicherlesefehlern sind Zugriffe auf ein Feld mit einem Vektorinkrement, welches einen ganzzahligen Teiler mit der Größe des Cache-Speichers bildet. Dies tritt beispielsweise bei der zeilenweisen Referenzierung eines zweidimensionalen Feldes auf, bei der das Vektorinkrement der Länge der Spalte entspricht. Wenn der zeilenweise Zugriff nicht vermeidbar ist, können die Felder vergrößert werden *(padding)*, so daß die Felder nicht vollständig mit Daten besetzt sind. Das folgende Beispiel verdeutlicht die Vorgehensweise.

Beispiel 4.21

```
REAL*8 A(16, 64, 64)

DO K = 1, N
   DO I = 1, 64
      A(1,1,I)
   ENDDO
ENDDO
```

Das dreidimensionale Feld `A(1,64,I)` wird innerhalb der angegebenen Schleife mit dem Vektorinkrement $16 \cdot 64 \cdot 8$ Byte = 8192 Byte referenziert, was genau der Speicherkapazität des internen Cache-Speichers entspricht. Dieses Zugriffsmuster hat zur Folge, daß nach dem ersten Durchlauf der inneren Schleife nur eine Zeile des Cache-Speichers mit den Daten von `A(1,64,I)` gefüllt ist und somit die Wiederverwendung der referenzierten Daten von `A` in der äußeren `K`-Schleife zu keiner Lokalität führt. Eine Möglichkeit zur Performance-Steigerung besteht nun in der Vergrößerung des Feldes `A` in der zweiten Dimension: Anstelle von `A(16, 64, 64)` wird nun `A(16, 65, 64)` deklariert. Das Vektorinkrement beträgt nun 8320 Byte, und die referenzierten Feldelemente werden auf 64 verschiedene Cache-Speicherzeilen abgebildet. Nach dem ersten Durchlauf der inneren Schleife besteht daher Lokalität für alle Feldelemente, die durch (N-1)-fache Wiederverwendung in der äußeren Schleife ausgenutzt werden kann.

Die beschriebene Technik ist nicht im DEC FORTRAN-Compiler implementiert, was bei ungünstig dimensionierten Feldern zu einer erheblichen Reduktion der zur Verfügung stehenden Cache-Speicherkapazität führt.

Fazit

Die beschriebenen Optimierungen des DEC FORTRAN-Compilers beziehen sich ausschließlich auf die verbesserte Ausnutzung der internen Prozessorkomponenten. Für ein verbessertes Instruktions-Scheduling in Schleifenkonstrukten wird die innere Schleife vierfach entfaltet, solange keine Kontrollflußabhängigkeiten vorliegen oder der Basisblock bereits ausreichend lang ist. Insbesondere bei Schleifenkonstrukten, die nur einen kurzen Basisblock beinhalten, verspricht ein stärkeres Entfalten eine weitere Performance-Steigerung.

Wie aus den Abbildungen 4.4 und 4.5 hervorgeht, reicht ein effektives Instruktions-Scheduling zur Erreichung einer möglichst hohen Performance allein nicht aus. Lesefehler in den Cache-Speichern können die für kleine Problemgrößen erzielten Performance-Steigerungen erheblich einschränken, wenn die Größe der referenzierten Felder anwächst. Da der untersuchte Compiler keine Transformationen für ein effektives Cache-Speicherverhalten vornimmt, soll im nächsten Kapitel untersucht werden,

welche Schritte vom Programmierer durchgeführt werden können, um diesen Mangel auszugleichen. Desweiteren wird beschrieben, inwiefern ein Entfalten der äußeren Schleifen in Schleifenkonstrukten zu weiteren Leistungssteigerungen führen kann.

Kapitel 5

Optimierung auf Hochsprachenebene

Wie im vorigen Kapitel beschrieben wurde, berücksichtigt der DEC FORTRAN-Compiler bei der Programmoptimierung das Speichersystem nicht ausreichend. Dies führt insbesondere bei Programmen mit einem hohen Anteil an Wiederverwendung zu gravierenden Performance-Einbußen, da die Zugriffe auf gleiche Feldelemente zeitlich so weit voneinander entfernt ausgeführt werden, daß keine Ausnutzung der Datenlokalität möglich ist. In diesem Kapitel werden eine Reihe von Schleifentransformationen beschrieben, die eine Umgruppierung der Feldzugriffe ermöglichen, so daß Referenzen auf gleiche Feldelemente zeitlich in einem kürzeren Abstand aufeinanderfolgen und so die Datenlokalität ausgenutzt wird. Desweiteren wird gezeigt, wie mit Hilfe der Informationen aus der Datenabhängigkeitsanalyse die Kombination der durchzuführenden Transformation ermittelt werden kann. Für das folgende Kapitel gelten folgende Annahmen:

- In allen Beispielen werden Feldelemente vom Typ REAL*8 benutzt, so daß in jeder Zeile des betrachteten internen Cache-Speicher jeweils vier Elemente enthalten sind. In den Abbildungen ist dies vereinfacht dargestellt. Der externe Cache-Speicher wird nicht berücksichtigt, da seine Existenz keinen Einfluß auf die im folgenden eingeführten Begriffe hat.

- Es werden nur solche Feldreferenzen analysiert, die einheitlich gebildet worden sind (*uniformly generated sets*): Sei d die Dimension des Feldindex, k die Anzahl der geschachtelten Schleifen und $\vec{i}$ der Iterationsvektor der in Kapitel 4 eingeführt wurde. Die durch die Index-Funktionen $\vec{f}$ und $\vec{g}$, $Z^k \rightarrow Z^d$, gebildeten Feldreferenzen $A(\vec{f}(\vec{i}))$ und $B(\vec{g}(\vec{i}))$ sind einheitlich gebildet, wenn gilt:

$$\vec{f}(\vec{i}) = H\vec{i} + \vec{C}_f \quad \text{und} \quad \vec{g}(\vec{i}) = H\vec{i} + \vec{C}_g$$

wobei H eine lineare Transformation ist und $\vec{C}_f$ und $\vec{C}_g$ konstante Vektoren sind.

Die Feldreferenzen `A(I)` und `A(I+1)` aus dem Beispiel 5.1 werden dann wie folgt gebildet:

$$H = [\,1 \quad 0\,]$$

$$\vec{f}(\vec{i}) = H\,\vec{i} + \vec{C}_f = [\,1 \quad 0\,] \begin{bmatrix} \mathtt{I} \\ \mathtt{J} \end{bmatrix} + [\,0\,] = \mathtt{I}$$

$$\vec{g}(\vec{i}) = H\,\vec{i} + \vec{C}_g = [\,1 \quad 0\,] \begin{bmatrix} \mathtt{I} \\ \mathtt{J} \end{bmatrix} + [\,1\,] = \mathtt{I} + 1$$

Beispiel 5.1

```
DO I = 1, N
   DO J = 1, N
      ...A(I)...
      ...A(I+1)...
   ENDDO
ENDDO
```

Bei existierenden vektorisierenden Compiler werden diesen Feldreferenzen wesentliche Bedeutung zugemessen, weil anhand dieser Ausdrücke die Vektorisierbarkeit eines Programm-Code identifiziert wird. In der vorliegenden Arbeit erhalten diese Referenzen besondere Beachtung, da sie auf ein hohes Maß an ausnutzbarer Datenlokalität hinweisen. Für anders generierte Feldreferenzen kann die ausnutzbare Datenlokalität durch Schleifentransformationen nur im geringen Maß verbessert werden [13, 36]. Sie werden aus diesem Grund nicht betrachtet.

5.1 Datenlokalität und Wiederverwendung

Unter der Datenlokalität in Rechnern mit verteiltem Speicher versteht man die Präsenz von Daten in den lokalen Speichern der einzelnen Prozessoren. Im folgenden wird als Datenlokalität die Anwesenheit der Daten in der betrachteten Speicherhierarchieebene (Register und interner Cache-Speicher) definiert.

Die Datenlokalität im Cache-Speicher führt an sich noch nicht zur Reduktion von Hauptspeicherzugriffen. Sind beispielsweise mit einem Ladebefehl vier 64-Bit-Daten in den Cache-Speicher geladen worden, aber nur ein Datum wird davon benutzt, so

besteht für die anderen drei Daten zwar Lokalität, sie wird jedoch nicht ausgenutzt. Erst mit der Ausnutzung der Lokalität sind Performance-Gewinne zu erzielen, da Hauptspeicherzugriffe eingespart werden. Das Ausmaß der möglichen Datenlokalität ist abhängig von der verwendeten Speicherarchitektur bzw. von der Größe der betrachteten Speicherhierarchieebene: Je größer beispielsweise der Cache-Speicher ist, desto mehr Daten können gleichzeitig lokal sein.

Um die Datenlokalität eines Programms bezüglich des Cache-Speichers zu bestimmen, ist zu überlegen, auf welche Weise ein Datum in den Cache-Speicher gelangt. Anhand des Beispiels 5.2 werden die verschiedenen Fälle erläutert.

Beispiel 5.2

```
DO I = 1, N
   A(I) = B(I)
   C(I) = C(I) * 3
ENDDO
```

Folgende Feldelemente werden in der ersten Iteration in den Cache-Speicher geladen.

1. `B(1)`, direkt durch den Ladebefehl `B(I)`,
2. `B(2)`, `B(3)` und `B(4)` indirekt durch den Ladebefehl `B(I)`,
3. `C(1)`, direkt durch den Ladebefehl `C(I)` und
4. `C(2)`, `C(3)` und `C(4)` indirekt durch den Ladebefehl `C(I)`

Bezüglich `A(I)` ist anzumerken, daß solange das referenzierte Feldelement nicht bereits im Cache-Speicher vorhanden ist, der Schreibbefehl direkt im Schreibpuffer abgelegt und der Cache-Speicher umgangen wird (*read allocate*). Ist `A(1)` bereits im Cache-Speicher enthalten, so erfolgt eine Aktualisierung des Eintrags und der Schreibbefehl wird danach dem Schreibpuffer zugeführt.

Die notwendige Ausführungszeit der Schreibzugriffe hängt nicht davon ab, ob das jeweilige Feldelement im Cache-Speicher vorhanden ist oder nicht. Performance-Steigerungen sind immer dann zu erwarten, wenn ein lesender Zugriff auf den Cache-Speicher einen Lesetreffer zur Folge hat. In dem obigen Beispiel referenziert der Speicherbefehl `C(I)` die gleiche Speicherstelle wie der Ladebefehl und ändert das Datum. Für die Betrachtung der Datenlokalität ist diese Änderung jedoch nicht von Interesse, da der folgende Ladebefehl in einem Lesetreffer resultiert. Für die Vorhersage der Datenlokalität würde die Betrachtung der Eingabeabhängigkeit von `C(I)` genügen. Diese Vorgehensweise wird bei einem ähnlichem Problem auch bei Gannon gewählt [13]. Genauso muß ein Schreibbefehl nicht beachtet werden, der vor einem entsprechenden Ladebefehl ausgeführt wird, da der Cache-Speicher in diesem Fall umgangen wird.

Zyklische Ausgabeabhängigkeiten jedoch müssen beachtet werden, da sie eine der Iterationszahl entsprechende Anzahl von Schreibbefehlen zum externen Speichersystem zur Folge haben. Dies kann bei Überlauf des Schreibpuffers zur Blockierung der Adreßeinheit und damit zu erheblichen Verzögerungszeiten führen.

Das Ziel ist es nun, bereits zur Übersetzungszeit beurteilen zu können, welche Feldreferenzen auf dasselbe Feldelement zugreifen und durch welche Änderung der Iterationsfolge diese Referenzen zeitlich enger zusammengebracht werden können. Um zur Übersetzungszeit beurteilen zu können, welche Daten zu welcher Zeit lokal sind, kann die Datenabhängigkeitsanalyse zur Erkennung der *Wiederverwendung* von Feldelementen in einem Programm benutzt werden. Es existieren mehrere Arten der Wiederverwendung, die im folgenden beschrieben werden.

Definition 5.1

> Es besteht **selbst-zeitliche Wiederverwendung**, wenn eine Feldreferenz auf dasselbe Feldelement in verschiedenen Iterationen zugreift.

Beispiel 5.3

```
DO I = 1, N
   DO J = 1, N
      ...= A(I) + B(J)
   ENDDO
ENDDO
```

In Beispiel 5.3 ist der der Index von `A(I)` bezüglich der `J`-Schleife konstant, so daß in verschiedenen Iterationen (1,1), (1,2),...,(1,`N`) dasselbe Feldelement `A(1)` referenziert wird: `A(I)` besitzt selbst-zeitliche Wiederverwendung in der `J`-Schleife. Die Zugriffe finden in aufeinanderfolgenden Iterationen statt, so daß ein Verbleiben des Datums im Cache-Speicher während der Durchführung der Instruktionen im Basisblock der inneren Schleife sehr wahrscheinlich ist: Die nach dem Zugriff auf `A(1)` bestehende Lokalität des ersten Feldelementes von `A(I)` wird daher ausgenutzt, und `A(1)` muß nicht mehr in jeder Iteration (1,1), (1,2),...,(1,`N`) aus dem externen Speicher geladen werden. Damit vermindern sich die notwendigen Hauptspeicherzugriffe für `A(1)` pro Iteration von 1 auf 1/`N`.

Der Ausdruck `B(J)` referenziert in den Iterationen (1,1), (2,1)...,(`N`,1) dasselbe Feldelement `B(1)`: `B(J)` besitzt selbst-zeitliche Wiederverwendung in der äußeren Schleife. Die Wiederverwendung ist jedoch, im Gegensatz zu `A(I)`, durch `N`-1 Zugriffe auf andere Feldelemente `B(J)` getrennt. Wenn `N` groß genug ist, so wird das betreffende Feldelement aus dem Cache-Speicher verdrängt, bevor es zum zweiten Mal referenziert wird. Damit kann die Lokalität von `B(J)` nicht ausgenutzt werden, und `B(1)` muß in allen Iterationen (1,1), (2,1),...,(`N`,1) neu geladen werden. Dies gilt entsprechend für alle Feldelemente von `B(J)`.

Wie unter Beispiel 5.2 beschrieben, kann ein Feldelement nicht nur durch dediziertes Laden in den Cache-Speicher gelangen und damit lokal sein, sondern auch infolge eines Blocktransfers Lokalität erlangen. In der Literatur spricht man in diesem Fall von räumlicher Wiederverwendung. Dies ist zwar ein wenig irreführend, da das Datum nicht zum zweiten oder dritten Mal, sondern erstmalig referenziert wird; weil jedoch der Begriff verbreitet ist, soll er auch im weiteren Verlauf der Arbeit verwendet werden.

Definition 5.2

Es besteht **selbst-örtliche Wiederverwendung**, wenn eine Feldreferenz auf Feldelemente innerhalb einer Cache-Speicherzeile in verschiedenen Iterationen zugreift.

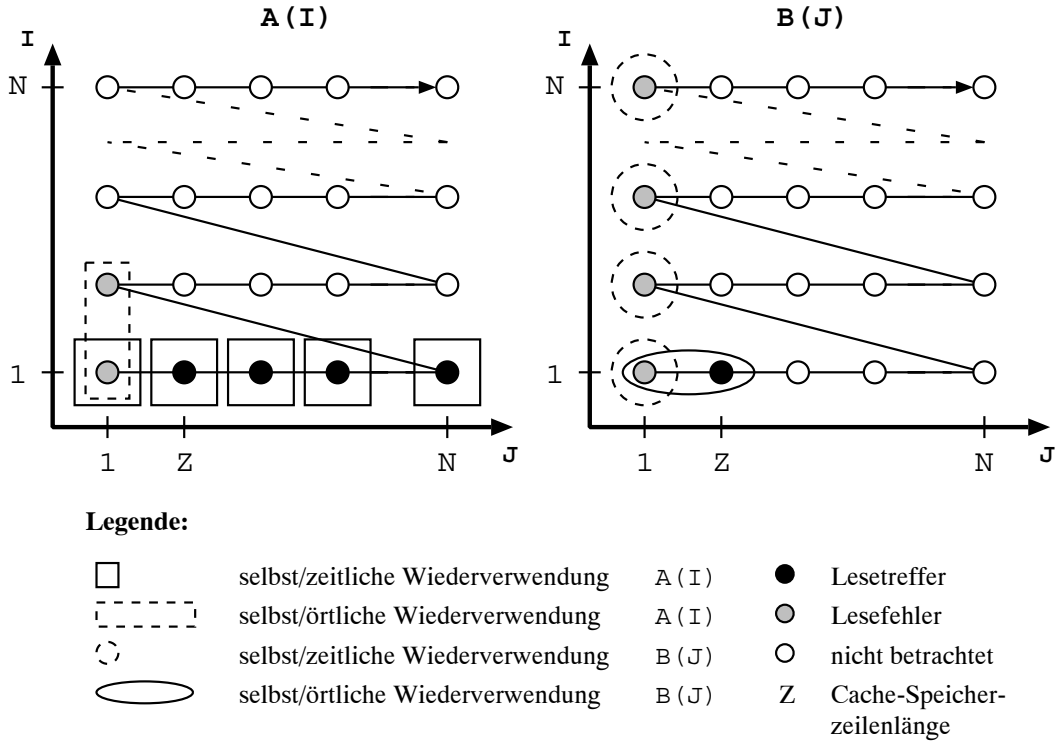


Abbildung 5.1: Beispiel 5.3: Selbst-zeitliche und selbst-örtliche Wiederverwendung

In Beispiel 5.3 wird mit dem Ladebefehl `A(1)` in der Iteration (1,1) auch die Elemente `A(2)`, `A(3)` und `A(4)` in dieselbe Cache-Speicherzeile geladen[1]. Da jedoch in der inneren Schleife sämtliche Elemente des Feldes `B(J)` in den Cache-Speicher geladen werden und `N` als so groß angenommen wird, daß nicht alle Daten von `B(J)` in den internen Cache-Speicher passen, wird `A(2)` überschrieben sein, bevor seine

[1]Unter der Voraussetzung, daß `A(1)` im Hauptspeicher auf einer 32-Byte-Grenze ausgerichtet ist.

Lokalität ausgenutzt werden kann[2]. Im Gegensatz dazu kann die Lokalität der Daten `B(2)`, `B(3)` und `B(4)` in den Iterationen (1,2),(1,3) bzw. (1,4) ausgenutzt und die Anzahl der notwendigen Ladezugriffe für `B(J)` um den Faktor 1/Z reduziert werden, wobei Z die Cache-Speicherzeilenlänge ist. Wird auf `B(J)` nicht mit der Zugriffsspanne 1 zugegriffen, sondern beispielsweise mit dem Inkrement 2 und nimmt man eine Cache-Speicherzeilenlänge von vier Feldelementen an, so wird nur die Lokalität der Feldelemente `B(3)`, `B(7)` usw. ausgenutzt. Die Reduktion der Ladezugriffe pro Iteration ist damit auch abhängig von der Zugriffsspanne n und ergibt sich allgemein zu:

$$\frac{n}{Z}, \quad \text{wenn } n \leq Z;$$

$$1, \quad \text{wenn } n > Z.$$

Die bestehende Wiederverwendung der Feldreferenzen `A(I)` und `B(J)` ist in Abbildung 5.1 dargestellt. Wiederverwendung, die zu ausnutzbarer Lokalität führt, ist mit durchgezogenen Linien gekennzeichnet, Wiederverwendung von Feldelementen, die zu keiner ausnutzbaren Lokalität führt, ist gestrichelt eingetragen. In den nächsten Abschnitten werden Schleifentransformationen vorgestellt, die die Iterationsfolge ändern und damit die ausgenutzte Lokalität der Feldelemente erhöht.

Zusätzlich zu den vorigen Arten der Wiederverwendung können noch folgende Fälle unterschieden werden:

Definition 5.3

Es besteht **gruppen-zeitliche Wiederverwendung**, wenn mehrere Feldreferenzen auf dasselbe Feldelement in verschiedenen Iterationen zugreifen.

Definition 5.4

Es besteht **gruppen-örtliche Wiederverwendung**, wenn mehrere Feldreferenzen auf Feldelemente innerhalb einer Cache-Speicherzeile zugreifen.

Anhand des Beispiels 5.4 sollen diese Definitionen veranschaulicht werden.

[2]Dabei wird angenommen, daß A(1) während der Iterationen (1,1),...,(1,N) in einem Register gehalten wird und somit kein Cache-Speicherzugriff bezüglich A(I) erfolgt. Ohne diese Annahme, würde auch die Lokalität von A(2) ausgenutzt werden, da A(1) in Iteration (1,N) im Cache-Speicher resident ist und damit auch A(2) in der folgenden Iteration (2,1).

Beispiel 5.4

```
DO I = 1, N
   DO J = 1, N
      ...= A(I) + A(I+1)
      ...= B(1,I) + B(2,I)
   ENDDO
ENDDO
```

Zusätzlich zu der selbst-zeitlichen Wiederverwendung von `A(I+1)`, kann nun auch `A(I)` von der Lokalität des Datums profitieren, welches von `A(I+1)` in der vorhergehenden Iteration referenziert wurde: Für die Feldreferenzen `A(I)` und `A(I+1)` besteht gruppen-zeitliche Wiederverwendung. Die notwendigen Hauptspeicherzugriffe verringern sich für `A(I)` damit von 2 auf $1/(2N)$. `B(1,I)` und `B(2,I)` referenzieren in keiner Iteration dasselbe Feldelement. Dennoch resultiert der Zugriff `B(2,I)` immer in einem Lesetreffer, wenn `B(1,I)` im Basisblock zuerst geladen wird. Das liegt daran, daß mit dem Zugriff auf `B(1,I)` auch die Elemente `B(2,I)`, `B(3,I)` und `B(4,I)` in dieselbe Cache-Speicherzeile geladen werden. In diesem Fall besteht gruppen-örtliche Wiederverwendung, und die notwendigen Zugriffe auf das externe Speichersystem reduzieren sich für `B(1,I)` und `B(2,I)` ebenfalls von 2 auf $1/(2N)$. Bei der Quantifizierung der Datenlokalität muß darauf geachtet werden, daß die Wiederverwendung der Feldreferenzen nicht doppelt gezählt wird. Eine mathematische Formulierung der Quantifizierung ist in [36] angegeben.

In diesem Abschnitt wurde gezeigt, wie aus der Betrachtung der einzelnen Feldreferenzen zur Übersetzungszeit Aussagen über die Lokalität von Daten gewonnen werden können. Im nächsten Abschnitt soll nun untersucht werden, inwiefern Transformationen von Schleifen die Ausnutzung der Lokalität verbessern. Das Ziel der Transformationen besteht darin, Feldreferenzen, die dasselbe Element bezeichnen, zeitlich näher zusammenzubringen und so die Wahrscheinlichkeit zu erhöhen, daß ein Feldelement bei der zweiten Referenz noch im Cache-Speicher vorhanden ist und ein Hauptspeicherzugriff vermieden werden kann.

5.2 Transformation von Schleifen

Für die Verminderung von Hauptspeicherzugriffen ist in der Regel eine einzige Transformation, wie beispielsweise die Schleifenvertauschung, nicht ausreichend. Um jedoch die Wirkungsweise der hier beschriebenen Transformationen

- Schleifenvertauschung,
- Schleifenblockung und
- Schleifenentfaltung

zu erläutern, werden im folgenden einfache Beispiele beschrieben, für die die Anwendung einer einzelnen Transformation zu einer Erhöhung der ausnutzbaren Lokalität führt. Dabei soll im einzelnen auf

- die Funktionsweise,
- die Voraussetzungen zur Anwendung und
- auf die Umsetzung in Programm-Code

eingegangen werden.

5.2.1 Schleifenvertauschung

Eine Schleifenvertauschung führt dann zu einer Performance-Steigerung, wenn für den Großteil der Feldreferenzen die ausnutzbare Lokalität erhöht wird. Diese Forderung macht das Problem klar: Die bessere Ausnutzung der Lokalität eines Datums durch Schleifenvertauschung kann gleichzeitig eine Erhöhung der Hauptspeicherzugriffe für andere Feldreferenzen bedeuten. Desweiteren beinhalten örtliche und zeitliche Wiederverwendung unterschiedliches Optimierungspotential: Während die zeitliche Wiederverwendung eine Verminderung der notwendigen Hauptspeicherzugriffe um die Anzahl der Iterationen der jeweiligen Schleife bewirken kann, ist die Einsparung von Haupstspeicherzugriffen bei der örtlichen Wiederverwendung proportional zur Breite der Cache-Speicherzeile. Da die Iterationszahl in der Regel die Anzahl der Daten einer Cache-Speicherzeile weit übersteigt, ist die Ausnutzung von Lokalität für Daten mit zeitlicher Wiederverwendung zu bevorzugen.

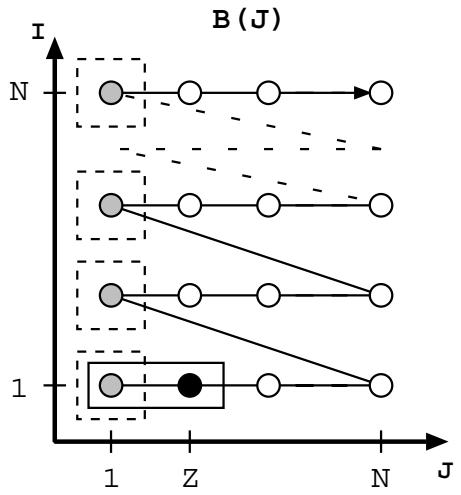
Selbst-zeitliche Wiederverwendung kann während der Datenabhängigkeitsanalyse durch sogenannte *self-cycles* im Abhängigkeitsgraph entdeckt werden. Anhand von Beispiel 5.5 soll die Ermittlung solcher Wiederverwendung und die Auswirkung der Schleifenvertauschung auf den Grad der ausnutzbaren Lokalität erläutert werden.

Beispiel 5.5

```
      DO I = 1, N
         DO J = 1, N
            ...
   S        ... = B(J) ...
            ...
         ENDDO
      ENDDO
```

Für `B(J)` besteht in Beispiel 5.5 folgende Eingabeabhängigkeit:

$$\delta^{I}_{\mathrm{B}}(1,0)^{+}\colon \mathrm{S} \rightarrow \mathrm{S}.$$



Ausdruck	Lokalität: J Wiederverwendung zeitl.	Wiederverwendung örtl.	Zugriffe / Iteration
B(J)	I	I, J	1/Z

Abbildung 5.2: Beispiel 5.5: Datenlokalität vor dem Vertauschen der Schleifen

Das gleiche Element von B(J) wird also in der äußeren I-Schleife (N-1)-mal wiederverwendet. Zur Berechnung der Wiederverwendung wird die ursprünglich notwendige Anzahl an Ladeoperationen durch die Iterationszahl dividiert und mit der Schrittweite multipliziert. Darüberhinaus ist wichtig, daß die Wiederverwendung in der äußeren Schleife besteht. Überschreitet nun das Produkt von Iterationszahl der inneren Schleife und Anzahl der Feldreferenzen, in deren Index die innere Schleifenvariable vorkommt, die im Cache-Speicher maximal speicherbare Feldelementanzahl, so wird B(J) verdrängt sein, bevor es wiederverwendet wird. Wie in Abbildung 5.2 dargestellt, kann lediglich die örtliche Wiederverwendung ausgenutzt werden. Dies hat eine Reduktion der Lesefehler von 1 auf 1/Z pro Iteration zur Folge hat.

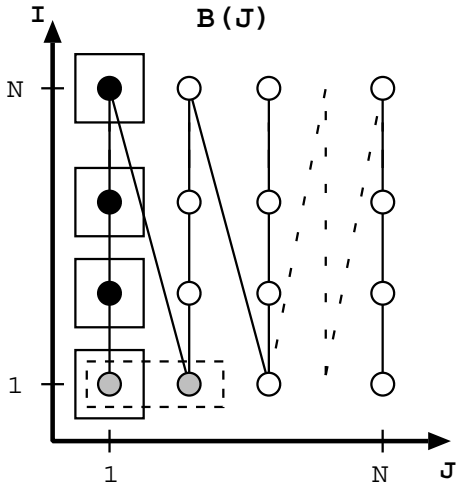
In Beispiel 5.5 ist eine Schleifenvertauschung sinnvoll, die die Wiederverwendung von B(J) in der inneren Schleife ermöglicht. Eine Vertauschung der Schleifen führt zu folgender Eingabeabhängigkeit für B(J):

$$\delta^I_{\mathrm{B}}(0,1)^+\colon \mathrm{S} \rightarrow \mathrm{S}.$$

In Abbildung 5.3 ist dargestellt, wie die Änderung der Iterationsfolge zu einer besseren Lokalität der Feldreferenz B(J) führt. Die resultierenden Lesefehler werden auf 1/(N) pro Iteration reduziert, da nun die selbst-zeitliche Wiederverwendung für B(J) besteht.

Korrektheit der Transformation

Eine Schleifenvertauschung darf nur dann vorgenommen werden, wenn die Semantik des transformierten Programmstückes beibehalten wird. Das Vertauschen der



Lokalität: I			
Ausdruck	Wiederverwendung zeitl.	Wiederverwendung örtl.	Zugriffe / Iteration
B(J)	I	I J	1/N

Abbildung 5.3: Beispiel 5.5: Verbesserte Ausnutzung der Lokalität nach dem Vertauschen der Schleifen

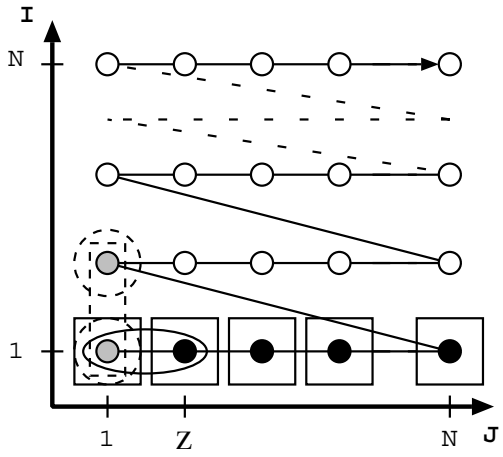
Schleifen bewirkt eine Änderung der Iterationsfolge was zu einer Vertauschung der Indizes im Distanzvektor führt. Wie im vorigen Kapitel beschrieben, darf das erste von Null verschiedene Element des Distanzvektors keine negative Zahl bzw. ein - enthalten, da sonst die bestehende Datenabhängigkeit verletzt wäre. Bei $\pm$ muß ein Hinweis vom Benutzer erfolgen, ob die Schleifenvertauschung zulässig ist.

5.2.2 Schleifenblockung

Das Blocken von Schleifen führt in der Regel zu einer teilweisen Änderung der Iterationsfolge. Die Methode des Blockens von Schleifen wird in der Literatur auch als *tiling* oder *strip mine and interchange* bezeichnet und kann bei entsprechender Wahl der Blockparameter auch eine reine Schleifenvertauschung wie im vorigen Kapitel bewirken.

Betrachtet man noch einmal das Beispiel 5.3, so besteht weiteres Optimierungspotential in der nicht ausgenutzten Lokalität der Feldelemente von `B(J)`. Die Ursache dafür ist der zu große zeitliche Abstand der aufeinanderfolgenden Referenzen des gleichen Feldelementes `B(J)`. Dies ist in Abbildung 5.4 schematisch dargestellt. Man erkennt, daß für `B(J)` lediglich die örtliche Wiederverwendung zu einer ausnutzbaren Lokalität führt, die die notwendigen Hauptspeicherzugriffe von 1 auf 1/Z vermindert. Obwohl die ausgenutzte zeitliche Wiederverwendung von `A(I)` zu einer erheblichen Reduktion der Zugriffe auf das externe Speichersystem führt, ist die Summe der Zugriffe bestimmt durch die Zugriffe auf Elemente von `B(J)`.

Der zeitliche Abstand der Zugriffe auf gleiche Elemente von `B(J)` kann mit Hilfe der



Lokalität: I			
Aus-druck	Wiederver-wendung zeitl.	Wiederver-wendung örtl.	Zugriffe / Iteration
A(I)	J	I J	1/Z
B(J)	I	I J	1/N
Z <<N		Gesamt: ≈	1/Z

Legende:

□	selbst/zeitliche Wiederverwendung von A(i)	●	Lesetreffer
▭	selbst/örtliche Wiederverwendung von A(i)	●	Lesefehler
◌	selbst/zeitliche Wiederverwendung von B(j)	○	nicht betrachtet
⬭	selbst/örtliche Wiederverwendung von B(j)	Z	Cache-Speicher-zeilenlänge

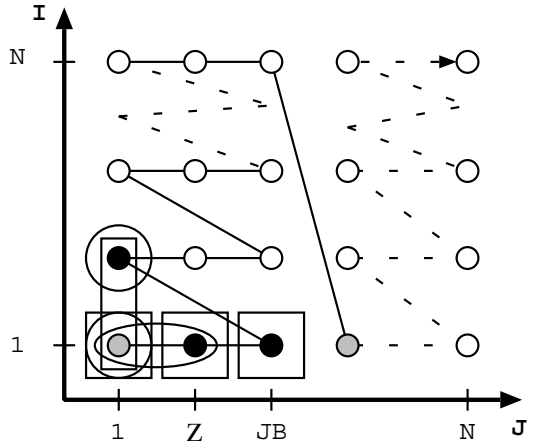
Abbildung 5.4: Beispiel 5.3: Ungenügende Ausnutzung der Datenlokalität im Cache-Speicher

Schleifenblockung verkürzt werden. Anstelle aller `N` Iterationen der inneren Schleife werden nur noch `JB` Iterationen durchlaufen, bevor die äußere Schleife inkrementiert wird. Die Größe `JB` wird Blockfaktor genannt und ist in diesem Beispiel so zu wählen, daß die Elemente von `B(J)` wiederverwendet werden, bevor sie aus dem Cache-Speicher verdrängt werden.

Die Iterationsfolge nach Durchführung der Schleifentransformation ist in Abbildung 5.5 dargestellt. Die Blockung der inneren Schleife hat dazu geführt, daß für `B(J)` neben der örtlichen auch die zeitliche Wiederverwendung zu einer ausnutzbaren Lokalität führt. Die notwendigen Hauptspeicherzugriffe pro Iteration vermindern sich um den Faktor $1/(ZJB)$, da zusätzlich für `JB` Elemente von `B(J)` die Lokalität im Cache-Speicher ausgenutzt wird. Darüberhinaus resultiert nun auch die örtliche Wiederverwendung von `A(I)` in einer ausnutzbaren Lokalität, so daß sich die für dieses Feld notwendigen Hauptspeicherzugriffe ebenfalls weiter reduzieren. Bei der Berechnung der Summe kann der Summand $1/(ZN)$ gegenüber $1/(ZJB)$ unter der Bedingung $N \gg JB$ vernachlässigt werden.

An diesem Beispiel kann auch erklärt werden, wann die örtliche Wiederverwendung bei gleichzeitig bestehender zeitlicher Wiederverwendung zu einer zusätzlichen Ausnutzung der Lokalität führt. Durch ihre Definition ist die zeitliche Wieder-

verwendung eine Untermenge der örtlichen Wiederverwendung, da die Benutzung des gleichen Feldelementes die Benutzung der gleichen Cache-Speicherzeile impliziert. Die Information der lokalen Wiederverwendung beinhaltet nur, daß die gleiche Cache-Speicherzeile referenziert wird. Erst wenn die selbst-zeitliche und selbst-örtliche Wiederverwendung für diese Referenz in unterschiedlichen Schleifen besteht, ist eine zusätzliche Verminderung der Hauptspeicherzugriffe zu erwarten. Bei einer Lokalität, die nur für die innere Schleife besteht, kann eine Referenz mit schleifeninvariantem Index (selbst-zeitliche Wiederverwendung) in der inneren Schleife nicht gleichzeitig ein anderes Element in derselben Cache-Speicherzeile referenzieren. Außerhalb der Schleife ist dies sehr wohl möglich.



Lokalität: I, J

Ausdruck	Wiederverwendung zeitl.	Wiederverwendung örtl.	Zugriffe / Iteration
A(I)	**J**	**I**, J	1/(ZN)
B(J)	**I**	I, **J**	1/(ZJB)
Z<<JB<<N		**Gesamt: ≈ 1/ZJB**	

Abbildung 5.5: Beispiel 5.3a: Verbesserte Ausnutzung der Datenlokalität im Cache-Speicher durch Blockung der inneren Schleife

Um die in Abbildung 5.5 dargestellte Iterationsfolge zu erzielen, muß das Programmbeispiel aus 5.3 in folgende Form transformiert werden:

Beispiel 5.3

```
DO I = 1, N
   DO J = 1, N
      ...= A(I) + B(J)
   ENDDO
ENDDO
```

Beispiel 5.3a

```
DO JJ = 1, N, JB
   DO I = 1, N
      DO J = JJ, MIN(JJ+JB-1, N)
         ...= A(I) + B(J)
      ENDDO
   ENDDO
ENDDO
```

Die innere J-Schleife wird nicht an einem Stück bis zur N-ten Iteration durchlaufen, sondern in Blöcken von JB. Die Korrektur um -1 wird hinzugefügt, damit die Anzahl

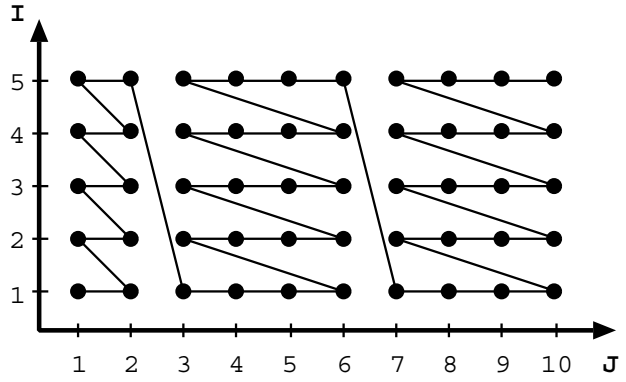
der durchlaufenen Iterationen in der inneren Schleife mit dem Blockfaktor übereinstimmt. Die restlichen Iterationen die sich aus MOD(N, JB) ergeben, werden zum Schluß durchlaufen, was durch die Minimum-Abfrage im Kopf der inneren Schleife ermöglicht wird. Die zusätzliche Ausführungszeit, die sich aufgrund des Hinzufügens einer weiteren Schleifenabfrage ergeben, kann durch die Betrachtung des generierten Assembler-Code geschätzt werden und ist bei einer automatischen Durchführung der Blockung durch den Compiler gegen die zu erwartenden Hauptspeicherzugriffszeiten abzuwägen. Eine weitere Verbesserung kann durch die frühzeitige Berechnung der Minimum-Funktion zwischen I- und J-Schleife erzielt werden.

In der gleichen Weise kann zusätzlich auch die I-Schleife durch Einfügen einer II-Schleife geblockt werden. Dabei ist zu beachten, daß die zugehörige Kontrollschleife ganz an den Anfang des Programm-Code gestellt wird, da ansonsten die I- und II-Schleife direkt aufeinanderfolgen und zusammengefaßt werden können. Die Wahl der Blockfaktoren bestimmt das resultierende Zugriffsmuster der einzelnen Feldreferenzen. Für JB in Beispiel 5.3a würde, bei einer Problemgröße von $N > 1024$ und einer Feldelementgröße von 8 Byte, ein Wert von 1024 benutzt werden, damit der 8 KByte große Cache-Speicher vollständig belegt ist. Bei der zusätzlichen Anwesenheit eines gleich dimensionierten Feldes D(J) würde sich der optimale Blockfaktor dementsprechend halbieren. Zusätzlich müssen die Felder im Hauptspeicher optimal ausgerichtet werden. Das Feldelement A(I) wird vor der inneren Schleife durch *scalar replacement* in einem Register gehalten. Die Berechnung des optimalen Blockfaktors ist schon bei relativ einfachen Algorithmen wie der Matrix-Multiplikation kompliziert und sollte bei hochoptimierten Programm-Codes durch Testläufe bestimmt werden.

Eine andere Form der Blockung ist in Abbildung 5.6 dargestellt. In diesem Programmstück wird der Rest der Iterationen (MOD(NJ-1,JB)+1) zuerst ausgeführt. Die Subtraktion und Addition mit 1 ist für den Sonderfall notwendig, daß die obere Grenze NJ ganzzahlig durch den Blockfaktor JB teilbar ist. Ohne Einfügen dieser Korrektur wäre das Ergebnis der Modulo-Berechnung gleich Null und damit auch die obere Grenze ZJ2, so daß keine Iteration durchgeführt würde. Mit der vorgesehenen Korrektur wird RESTJ schon zu Beginn mit dem Wert des Blockfaktors belegt, da kein Rest von Iterationen auszuführen ist. Diese Form vereinfacht die Formulierung von Schleifen, die sowohl geblockt als auch entfaltet werden sollen, was zum Ende dieses Kapitels beschrieben wird.

Korrektheit der Transformation

Die Schleifenblockung stellt für die Erreichung einer besser ausgenutzten Lokalität ein wirkungsvolles Werkzeug dar. Aufgrund von Datenabhängigkeiten ist diese Methode jedoch nicht immer anwendbar. In [36] ist eine detaillierte Beschreibung der Zulässigkeit angegeben, in der zur Vereinfachung folgende Bedingung aufgestellt wird:




```
NI=5
NJ=10
JB=4
RESTJ=MOD(NJ-1,JB)+1
ZJ1=1
ZJ2=RESTJ
DO JJ=1,NJ,JB
   DO I=1,NI
      DO J=Z1,Z2
         A(I)=A(I)+B(J)
      ENDDO
   ENDDO
   ZJ1=ZJ1+RESTJ
   ZJ2=ZJ2+JB
   RESTJ=JB
ENDDO
```

Abbildung 5.6: Beispiel 5.3: Modifizierte Schleifenblockung

> Aufeinanderfolgende geschachtelte Schleifen können dann geblockt werden, wenn alle diese Schleifen vollständig permutierbar sind.

Zudem sind nach der Blockung alle Schleifen, die neu hinzugekommenen Schleifen einbezogen, permutierbar. Dies ermöglicht die weitere Blockung dieser Schleifen, beispielsweise für die bessere Ausnutzung der Datenlokalität im Hauptspeicher. Welche Schleifen geblockt werden müssen, geht aus der bereits beschriebenen Datenabhängigkeitsanalyse hervor. Im nächsten Kapitel wird dies exemplarisch am Beispiel der Matrixmultiplikation dargestellt.

5.2.3 Schleifenentfaltung

Auf die Schleifenentfaltung wurde bereits im vorigen Kapitel im Rahmen der globalen Optimierungen des DEC FORTRAN-Compilers eingegangen. Die vom Compiler durchgeführte Optimierung ist auf die innere Schleife beschränkt und benutzt zudem einen festen Entfaltungsfaktor. Das Ziel dabei ist die Vergrößerung des Basisblocks und dient einem verbesserten Instruktions-Scheduling. Die Entfaltung der inneren Schleife hat keine Änderung der Auswertungsreihenfolge zur Folge, was einerseits den Vorteil hat, daß keine Datenabhängigkeiten diese Transformation verhindern, andererseits jedoch keine Ausnutzung der Wiederverwendung von Feldvariablen in unterschiedlichen Iterationen ermöglicht wird.

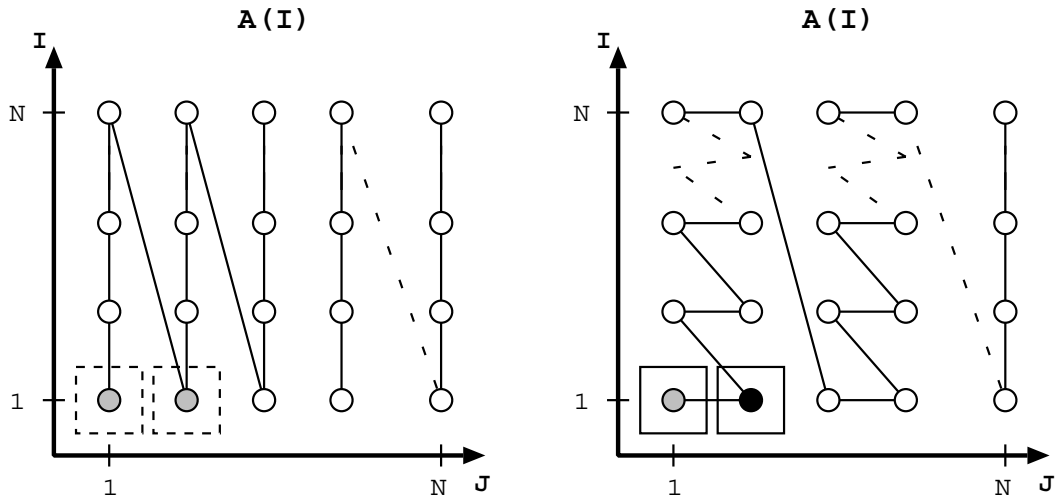


Abbildung 5.7: Beispiel 5.6: Wiederverwendung von Registerinhalten

Im folgenden Abschnitt soll die Entfaltung unter dem Gesichtspunkt der Wiederverwendung in den Registern untersucht werden. Dazu wird insbesondere die Wirkung der Entfaltung der äußeren Schleife auf eine effektivere Registerbelegung beschrieben. Es ist zu beachten, daß auf Registerebene im Fall des Alpha-Chip keine örtliche Wiederverwendung besteht, da kein Befehl vorgesehen ist, der mit dem Laden eines Registers zusätzlich auch benachbarte Register mit Daten belegt. Desweiteren wird die bestehende Datenlokalität immer ausgenutzt, da vom Compiler keine unnötigen Ladebefehle generiert werden.

Die vom Compiler benutzte Strategie zur Besetzung von Registern weist skalaren Variablen und schleifeninvarianten Feldreferenzen höchste Priorität zu. Bei der Entfaltung der inneren Schleife vervielfachen sich schleifeninvariante Feldreferenzen im Basisblock. Da diese vom Compiler in der inneren Schleife ohnehin in einem Register gehalten werden, ist dieser Effekt für die Performance nicht von Relevanz.

In Beispiel 5.6 ist noch einmal das Programmstück aus Beispiel 4.16 angeführt. Die zugehörige Auswertungsreihenfolge ist auf der linken Seite der Abbildung 5.7 dargestellt. Die gestrichelten Quadrate stellen die Feldreferenz `A(1)` dar, deren Wiederverwendung zeitlich zu weit von der ersten Referenz entfernt ist, als daß `A(1)` noch in einem Register enthalten wäre.

Beispiel 5.6

```
      DO J = 1, N
         DO I = 1, N
S           A(I) = A(I) + B(J)
         ENDDO
      ENDDO
```

Um die Zugriffe auf das Feld A(I) näher zusammenzubringen ist eine Änderung der Iterationsfolge notwendig, die in Abbildung 5.7 auf der rechten Seite dargestellt ist. Die Zugriffe auf A(I) sind nun zeitlich unmittelbar benachbart, so daß beim zweiten Zugriff der Inhalt von A(I) noch im Register enthalten ist. Diese Auswertungsreihenfolge kann durch zweifaches Entfalten der äußeren Schleife erzielt werden, was im Beispiel 5.7 angegeben ist. Diese Transformation ist zulässig, da die echte Datenabhängigkeit $\delta_A(1,0)$: S $\rightarrow$ S durch die Veränderung der Ausführungsreihenfolge nicht verletzt wird. Durch das Entfalten der äußeren Schleife kann die zeitliche Wiederverwendung von A(I) ausgenutzt werden, was auf der rechten Seite der Abbildung 5.7 dargestellt ist.

Bei der Betrachtung des resultierenden Programm-Codes wird dies ebenfalls deutlich:

Beispiel 5.7

```
      DO J = 1, N-1, 2
         DO I = 1, N
S1          A(I) = A(I) + B(J)
S2          A(I) = A(I) + B(J+1)
         ENDDO
      ENDDO
      DO J = J, N
         DO I = 1, N
            A(I) = A(I) + B(J)
         ENDDO
      ENDDO
```

Der Zugriff auf A(I) in Anweisung S2 resultiert nun nicht mehr in einem Cache- oder Hauptspeicherzugriff, da A(I) innerhalb des Basisblocks in einem Register gehalten wird. Der Inhalt der Feldelemente B(J) und B(J+1) wird in der inneren Schleife im Register gehalten, da die Referenzen schleifeninvariant sind. Damit hat sich die Anzahl der notwendigen Cache-Speicherzugriffe pro Iteration für A(I) um den Faktor 2 gesenkt. Die Wahl des Entfaltungswertes ist begrenzt durch die Anzahl der zur Verfügung stehenden Fließpunktregister. Zur Bestimmung eines geeigneten Entfaltungswertes müssen die Feldreferenzen (Lade- und Schreibbefehle) betrachtet werden, deren Index die Variable der entfalteten Schleife enthält. Ist der Entfaltungsgrad zu groß, so müssen Registerinhalte im Stack zwischengespeichert werden und die Anwesenheit einer zu großen Anzahl von Schreibbefehlen blockiert die Adreß-Funktionseinheit. Bei zu geringem Entfaltungsgrad wird die interne Pipeline-Struktur nicht optimal ausgenutzt.

Als Seiteneffekt bei der Entfaltung von Schleifen können Datenabhängigkeiten im Basisblock auftreten, die ein effektives Pipelining verhindern. Dies tritt auch in Beispiel 5.7 auf: Die Anweisungen S1 und S2 werden aufgrund der echten Datenabhängigkeit im Basisblock sequentiell ausgeführt. Vertauscht man jedoch die

Schleifen und entfaltet dann die äußere I-Schleife, so existiert keine echte Datenabhängigkeit mehr im Basisblock, und die Assembler-Befehle Laden, Addition und Abspeichern können überlappt ausgeführt werden, wie auch schon im vorigen Kapitel gezeigt wurde. Das bedeutet, daß im Gegensatz zum Blocken von Schleifen, für die bessere Ausnutzung der Wiederverwendung auf Cache- oder Hauptspeicherebene, bei der Ausprogrammierung von Schleifen sowohl die Wiederverwendung von Registerinhalten als auch das effektive Pipelining in Betracht gezogen werden müssen. Sobald die Möglichkeit des Pipelining eingeschränkt wird, haben die zuvor beschriebenen Methoden zur verbesserten Ausnutzung der Wiederverwendung keinen Performance-Gewinn mehr zur Folge. Dieser Sachverhalt verdeutlicht die Wichtigkeit, die richtige Kombination der Schleifentransformationen zu finden.

Korrektheit der Transformation

Die Entfaltung der äußeren Schleifen bewirkt, im Gegensatz zur Entfaltung der inneren Schleife, eine teilweise Änderung der Iterationsfolge. Dementsprechend sind die gleichen Voraussetzungen an die Durchführung dieser Transformation zu stellen wie bei der Schleifenblockung.

5.2.4 Kombination von Schleifenentfaltung und Schleifenblockung

Aus den bisherigen Betrachtungen geht hervor, daß in der Regel nur die gemeinsame Anwendung von Schleifenblockung und -entfaltung zu einer Performance-Steigerung führt. Daher soll zum Schluß dieses Kapitels die Kombination dieser zwei Transformationen beschrieben werden. In Beispiel 5.8 ist der Programm-Code enthalten, der nach Blockung beiden Schleifen und Entfaltung der äußeren Schleife aus Beispiel 5.3 hervorgegangen ist.

Beispiel 5.8

```
RESTI = MOD (NI-1, IB) + 1
ZI1   = 1
ZI2   = RESTI
DO II = 1, NI, IB
   RESTJ = MOD (NJ-1, JB) + 1
   ZJ1   = 1
   ZJ2   = RESTJ
   DO JJ = 1, NJ, JB
      DO I = ZI1, ZI2-1, 2
         DO J = ZJ1, ZJ2
            ...= A(I) + B(J)
            ...= A(I+1) + B(J)
         ENDDO
      ENDDO
```

```
      DO I = I, ZI2
         DO J = ZJ1, ZJ2
            ...= A(I) + B(J)
         ENDDO
      ENDDO
      ZJ1   = ZJ1 + RESTJ
      ZJ2   = ZJ2 + JB
      RESTJ = JB
   ENDDO
   ZI1   = ZI1 + RESTI
   ZI2   = ZI2 + IB
   RESTI = IB
ENDDO
```

Der Entfaltungsfaktor in der J-Schleife ist zu 2 gewählt, er kann jedoch alle Werte zwischen 1 und NJ annehmen. Dasselbe gilt für die Wahlmöglichkeit der Blockfaktoren. In Abbildung 5.8 ist die geänderte Iterationsfolge an einem konkreten Beispiel dargestellt. Die Iterationen, die sich aufgrund der Modulo-Berechnung ergeben, sind hervorgehoben.

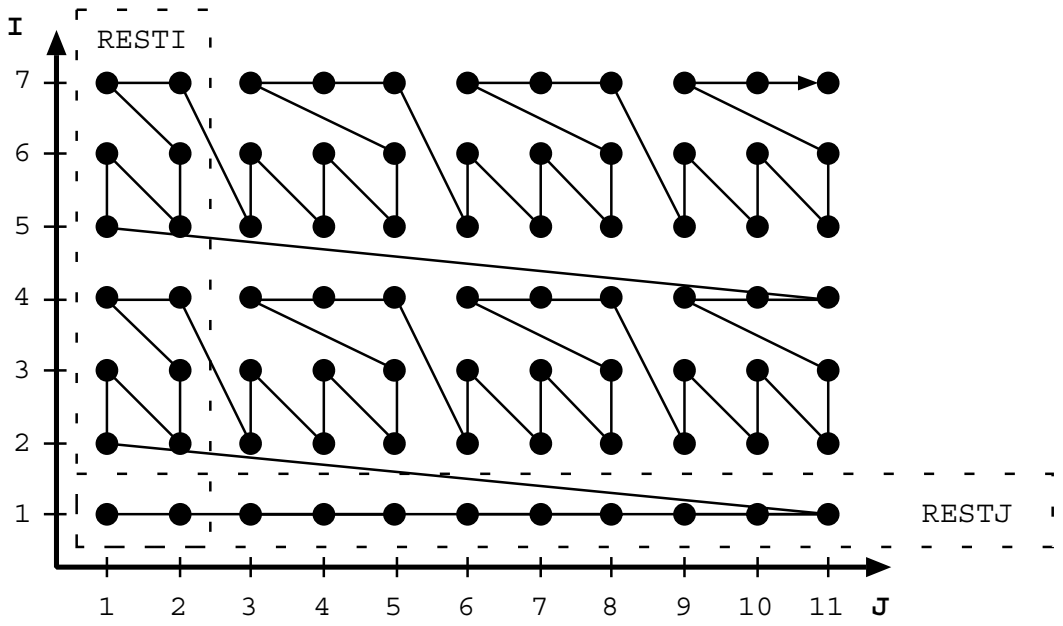


Abbildung 5.8: Beispiel 5.8: Gleichzeitige Anwendung von Schleifenblockung und -entfaltung

Mit Hilfe der vorgestellten Schleifentransformationen soll im folgenden Kapitel die Effektivität dieser Methoden zur Verbesserung der ausgenutzten Lokalität anhand von Beispielen belegt werden.

Kapitel 6

Anwendung der Schleifentransformationen

Die in Kapitel 5 vorgestellten Verfahren zur Optimierung von FORTRAN-Programmen auf Hochsprachenebene bedürfen der Validierung an einer Auswahl von Beispielen, die die Effektivität der Techniken in Bezug auf Performance-Steigerungen belegen. Dabei soll die Entscheidung, welche Schleifentransformation bzw. Kombination von Transformationen durchzuführen ist, möglich sein, ohne daß vorherige Messungen durchgeführt werden müssen. Folgenden Schritte werden dazu durchlaufen:

- Ermittlung der Datenabhängigkeit
  Zuerst werden die Datenabhängigkeiten ermittelt, die gegebenenfalls die Anzahl der möglichen Transformationen begrenzen.

- Bestimmung der Wiederverwendung und Lokalität
  Für alle Ausdrücke wird die Wiederverwendung quantifiziert und die erreichbare Lokalität durch die Anwendung der in Kapitel 5 vorgestellten Transformationen ermittelt. Dabei wird zuerst für den Cache-Speicher und danach für die Register optimiert.

- Ermittlung von Konflikten zwischen Wiederverwendung und Pipelining
  Die Anwendung der Schleifenentfaltung kann innerhalb des Schleifenrumpfes Datenabhängigkeiten hervorrufen, die das Instruktions-Scheduling behindern. Tritt dieser Fall auf, so ist eine andere Kombination von Transformationen anzuwenden.

- Wahl des optimalen Block- und Entfaltungsfaktors
  Maßgebend für die Wahl der optimalen Block- und Entfaltungsfaktoren sind die zur Verfügung stehende Cache-Speicherkapazität, die Zeilenlänge, der Abbildungsmechanismus und die Registeranzahl.

6.1 Effektive Pipeline- und Registerbelegung

Als erstes Beispiel wird der schon in im Kapitel 4 beschriebene einfache Programm-Code (Beispiel 6.1) hinsichtlich möglicher Optimierungen für eine effektivere Pipeline- und Registerbelegung untersucht.

Beispiel 6.1

```
DO J = 1, 100
   DO I = 1, 100
      A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

Im Vordergrund steht dabei die Wiederverwendung von Daten in den Registern und die Vergrößerung des Basisblockes für eine verbesserte Pipeline-Ausnutzung durch das Entfalten der inneren Schleife. Es werden nur solche Problemgrößen betrachtet, bei denen beide Felder `A(I)` und `B(J)` vollständig in den internen Cache-Speicher passen und eine Blockung der Schleifen nicht notwendig ist. Desweiteren wird nur ein kleiner Bereich der Felddimensionen betrachtet, um den Einfluß der durch die Entfaltung hervorgerufenen Korrekturschleifen zu untersuchen.

Der Programm-Code in Beispiel 6.1 kann durch eine andere Codierung wesentlich vereinfacht werden: Nach Ausführung des Programmstückes steht in allen Elementen von `A(I)` derselbe Wert, wenn `A(I)` zu Beginn mit gleichen Zahlen initialisiert wurde. Dies kann natürlich auch durch die Berechnung des Feldelementes `A(1)` und anschließender Speicherung in den anderen Feldelementen durchgeführt werden. Die im folgenden Abschnitt vorgenommenen Untersuchungen dienen daher in erster Linie zur Beurteilung, welche Entfaltung der Schleifen in welchem Maß zu Performance-Steigerungen führt. Die Ergebnisse sind insbesondere für die im nächsten Abschnitt untersuchte Matrixmultiplikation von Interesse, da dort ähnliche Betrachtungen durchzuführen sind. An diesem einfachen Beispiel lassen sich jedoch die Effekte der Entfaltung besser beobachten.

Datenabhängigkeit und Wiederverwendung

Aus der Datenabhängigkeitsanalyse gehen folgende schleifengetragene Abhängigkeiten hervor:

- $\delta^{O}_{\mathrm{A}}(1,0)^{+}$: $\mathrm{S} \rightarrow \mathrm{S}$
- $\delta^{I}_{\mathrm{A}}(1,0)^{+}$: $\mathrm{S} \rightarrow \mathrm{S}$
- $\delta_{\mathrm{A}}(1,0)$: $\mathrm{S} \rightarrow \mathrm{S}$
- $\delta^{I}_{\mathrm{B}}(0,1)^{+}$: $\mathrm{S} \rightarrow \mathrm{S}$

Weder die echte Datenabhängigkeit noch die Ausgabeabhängigkeit verhindern ein Vertauschen der Schleifen, so daß die Entfaltung der äußeren Schleife zulässig ist. Desweiteren ist ersichtlich, daß `A(I)` in der äußeren `J`-Schleife und `B(J)` in der inneren `I`-Schleife wiederverwendet wird und in beiden Fällen selbst-zeitliche Wiederverwendung besteht. Letzteres bedeutet, daß die Schleifenanordnung keinen Einfluß auf die notwendige Zahl der Hauptspeicherzugriffen hat. Demgegenüber besteht die Ausgabeabhängigkeit nur für `A(I)`, was im weiteren Verlauf für die Reduktion von Schreibbefehlen ausgenutzt wird. Die echte Datenabhängigkeit muß für ein effektives Pipelining beachtet werden, da sie nach Entfaltung der Schleifen zu Erzeuger/Benutzer- Konflikten im Basisblock führen kann.

Für die Minimierung der Speicherzugriffe für den Programm-Code in Beispiel 6.1 muß die äußere Schleife entfaltet werden (Beispiel 6.1a).

Beispiel 6.1a

```
DO J = 1, 97, 4
   DO I = 1, 100
      A(I) = A(I) + B(J)
      A(I) = A(I) + B(J+1)
      A(I) = A(I) + B(J+2)
      A(I) = A(I) + B(J+3)
   ENDDO
ENDDO
```

Dies hat den Vorteil, daß mehrere Zugriffe auf das Feldelement `A(I)` nicht mehr in jeder `N`-ten Iteration, sondern unmittelbar hintereinander durchgeführt werden. Dies reduziert sowohl die notwendigen Speicherzugriffe als auch die Schreibbefehle. Gleichzeitig werden jedoch zahlreiche echte Datenabhängigkeiten bezüglich `A(I)` im Basisblock generiert, die die überlappte Ausführung der einzelnen Additionen verhindern. Dies wird deutlich, wenn man den erzeugten Assembler-Code betrachtet.

```
L$45$
1a    ldt     f0, -8(r6)
1b    addl    J, 4, J
2a    ldt     f1, -8(r7)
2b    cmple   J, r3, r8
3a    ldt     f10, (r6)
3b    addt    f1, f0, f1      /*Registerkonflikt bzgl. f1 in 2a
4a    ldt     f0, 8(r6)
4b    addt    f1, f10, f10    /*Registerkonflikt bzgl. f1 in 3b
5a    ldt     f1, 16(r6)
5b    lda     r6, 32(r6)
6a    addt    f10, f0, f10    /*Registerkonflikt bzgl. f10 in 4b
```

```
6b    addt    f10, f1, f10    /*Registerkonflikt bzgl. f10 in 6a
7a    stt     f10, -8(r7)     /*Registerkonflikt bzgl. f10 in 6b
7b    bne     r8, L$45$
```

Die echten Datenabhängigkeiten der aufeinanderfolgenden Additions- und Speicherbefehle verursachen Erzeuger/Benutzer-Abhängigkeiten, die die Ausführungszeit erheblich verlängern.

Eine zuvor durchgeführte Vertauschung der Schleifen löst dieses Problem (Beispiel 6.1b). Nach Entfaltung der äußeren Schleife und *scalar replacement* für die Feldreferenzen `A(I)` sind innerhalb des Basisblocks keine Erzeuger/Benutzer-Abhängigkeiten vorhanden, und die Additionen können, wie schon in Kapitel 4 gezeigt, überlappt durchgeführt werden.

Beispiel 6.1b

```
DO I = 1, 97, 4
   A1 = A(I)
   A2 = A(I+1)
   A3 = A(I+2)
   A4 = A(I+3)
   DO J = 1, 100
      A1 = A1 + B(J)
      A2 = A2 + B(J)
      A3 = A3 + B(J)
      A4 = A4 + B(J)
   ENDDO
   A(I)   = A1
   A(I+1) = A2
   A(I+2) = A3
   A(I+3) = A4
ENDDO
```

Statt einem Lade und einem Schreibbefehl für `A(I)` in vier Iterationen der inneren Schleife (Beispiel 6.1a), wird nun nur noch ein Ladebefehl für `B(J)` ausgeführt. Die Feldelemente von `A(I)` werden außerhalb der inneren Schleife in die Register geladen und daher bei der Berechnung der Befehlsanzahl nicht mitgezählt. Desweiteren werden auch alle Speicherbefehle für `A(I)` außerhalb der inneren Schleife durchgeführt, was zu einem entsprechend kurzen Basisblock führt.

```
L$45:$
1a   ldt     f12, (r4)
1b   addl    J, 1, J
2a   cmple   J, r16, r7
```

```
2b   lda     r4, 8(r4)
3a   addt    A1, f12, A1
3b   addt    A2, f12, A2
4a   addt    A3, f12, A3
4b   addt    A4, f12, A4
5a   bne     r7, L$45
```

Der Einfluß der Einsparungen von Lade- und Speicherinstruktionen und die Vermeidung von Pipeline-Blockierungen auf die Performance sind in Abbildung 6.1 erkennbar.

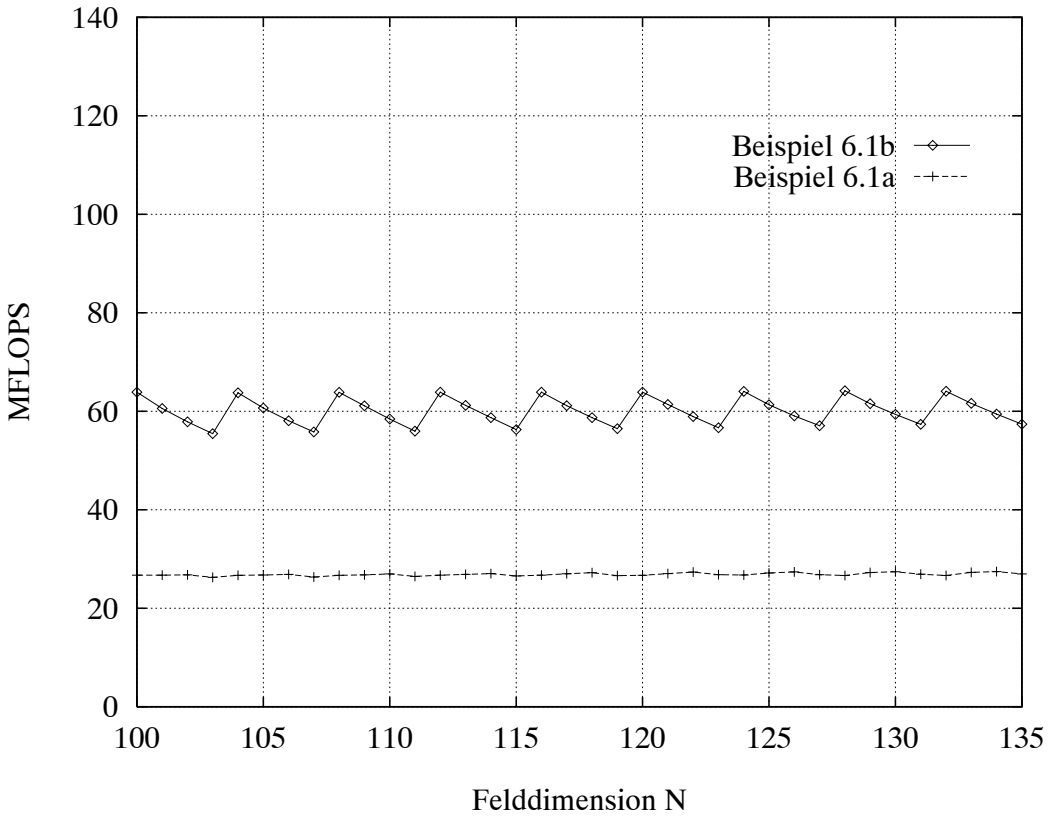


Abbildung 6.1: Performance-Messung, Beispiel 6.1a und 6.1b: Verschiedene Felddimensionen, Optimierungsoption **-O2**

Der sägezahnartige Verlauf der oberen Kurve resultiert aus der unterschiedlichen Anzahl an Iterationen in der Korrekturschleife. Für Problemgrößen, die durch vier ganzzahlig teilbar sind, wird dementsprechend die höchste Performance gemessen. Die untere Kurve zeigt einen glatten Verlauf, da die zusätzliche Ausführungszeit der Korrekturschleife gegenüber der Ausführungszeit des Basisblocks der inneren Schleife einen geringeren Anteil ausmachen als in Beispiel 6.1b.

Betrachtet man den Programm-Code in Beispiel 6.1a und bestimmt die notwendigen Fließpunktregister, so erkennt man, daß mit insgesamt fünf Registern längst nicht

alle ausgenutzt sind und zudem die Fließpunkt-Pipeline zu keiner Zeit voll belegt ist. Der nächste Schritt muß daher die stärkere Entfaltung der äußeren Schleife sein. Zu erwarten sind höhere Performance-Werte und eine ausgeprägtere Sägezahnstruktur der Performance-Kurven, da bei Problemgrößen die nach Division durch den Entfaltungsfaktor einen großen Rest aufweisen, die Korrekturschleife entsprechend häufiger durchlaufen werden muß.

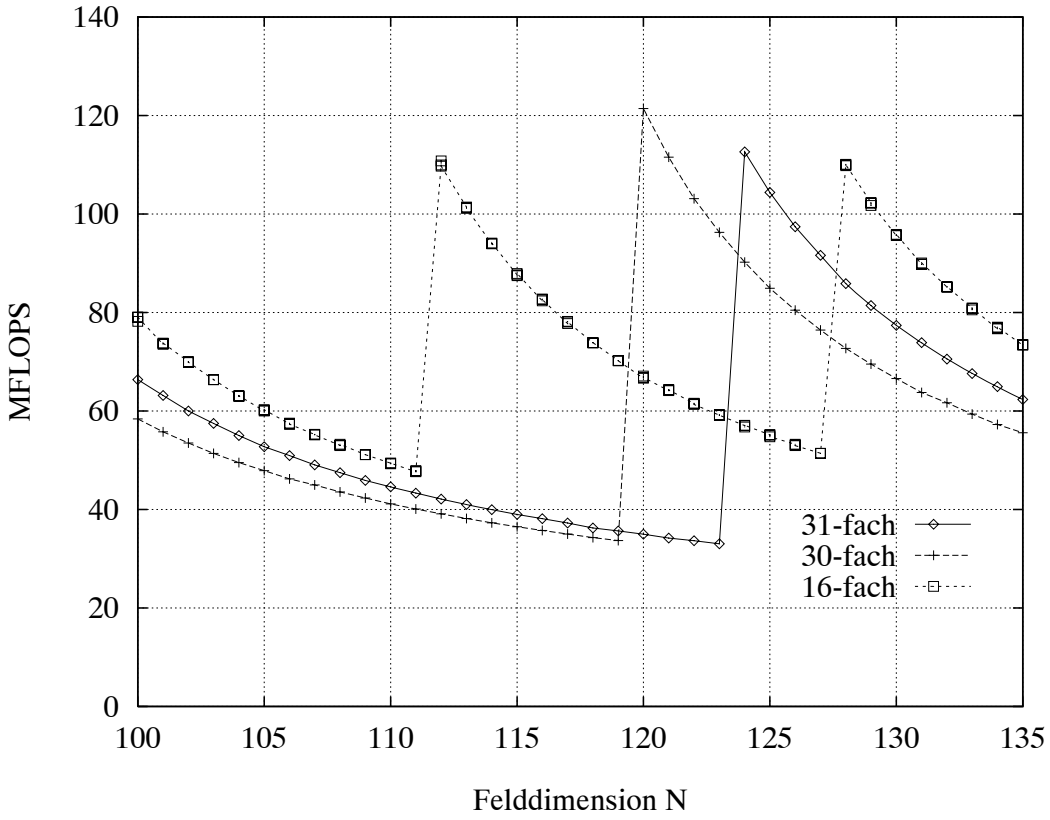


Abbildung 6.2: Performance-Messung, Beispiel 6.1: Unterschiedliche Entfaltung der äußeren Schleife, Optimierungsoption **-O2**

Abbildung 6.2 bestätigt diese Erwartungen. Der maximale Entfaltungsfaktor ist erreicht, wenn nicht mehr alle im Basisblock benötigten Feldelemente gleichzeitig in den Fließpunktregistern gehalten werden können und zur Zwischenspeicherung in den Speicher zurückgeschrieben werden. Die zur Zwischenspeicherung notwendigen Operationen, auch *spill code* genannt, haben dann eine Verringerung der Performance zur Folge. Dies erkennt man am Beispiel der 36-fach entfalteten Schleife. Desweiteren blockieren die durch das *scalar replacement* hervorgerufenen Speicherbefehle die Adreß-Funktionseinheit, da alle Speicherbefehle sowohl zum internen Cache-Speicher als auch zum externen Speichersystem gehen (*write through*).

Das alleinige Entfalten der äußeren Schleife ist gut anwendbar, wenn die Problemgröße zur Übersetzungszeit von dem Compiler erkannt werden kann. Ist dies nicht der Fall, so muß mit großen Performance-Einbußen gerechnet werden, wenn die

Korrekturschleife sehr oft durchlaufen werden muß. Dies kann durch die Wahl eines kleineren Entfaltungsfaktors vermieden werden. Eine andere Möglichkeit ist die zusätzliche Entfaltung der inneren Schleife, die automatisch durchgeführt wird, wenn die Optimierungsoption **-O3** oder **-O4** bei der Übersetzung angegeben wird. Die erzeugten Anweisungen im Basisblock ergeben sich dann aus dem Produkt der beiden Entfaltungsfaktoren, was zu sehr langen Basisblöcken führt. Übersetzt man beispielsweise den hier besprochenen Programm-Code mit einer achtfach entfalteten äußeren Schleife, so erhält man einen Basisblock, der $8 \cdot 4 = 32$ Anweisungen enthält. Dieser Effekt kann noch verstärkt werden, wenn man vor der Übersetzung zusätzlich die innere Schleife entfaltet hat. Bei achtfacher Entfaltung der äußeren und vierfacher Entfaltung der inneren Schleife im Quell-Code wird ein Basisblock mit $8 \cdot 4 \cdot 4 = 96$ Anweisungen erzeugt, was eine sehr gute Füllung der Fließpunkt-Pipeline ermöglicht.

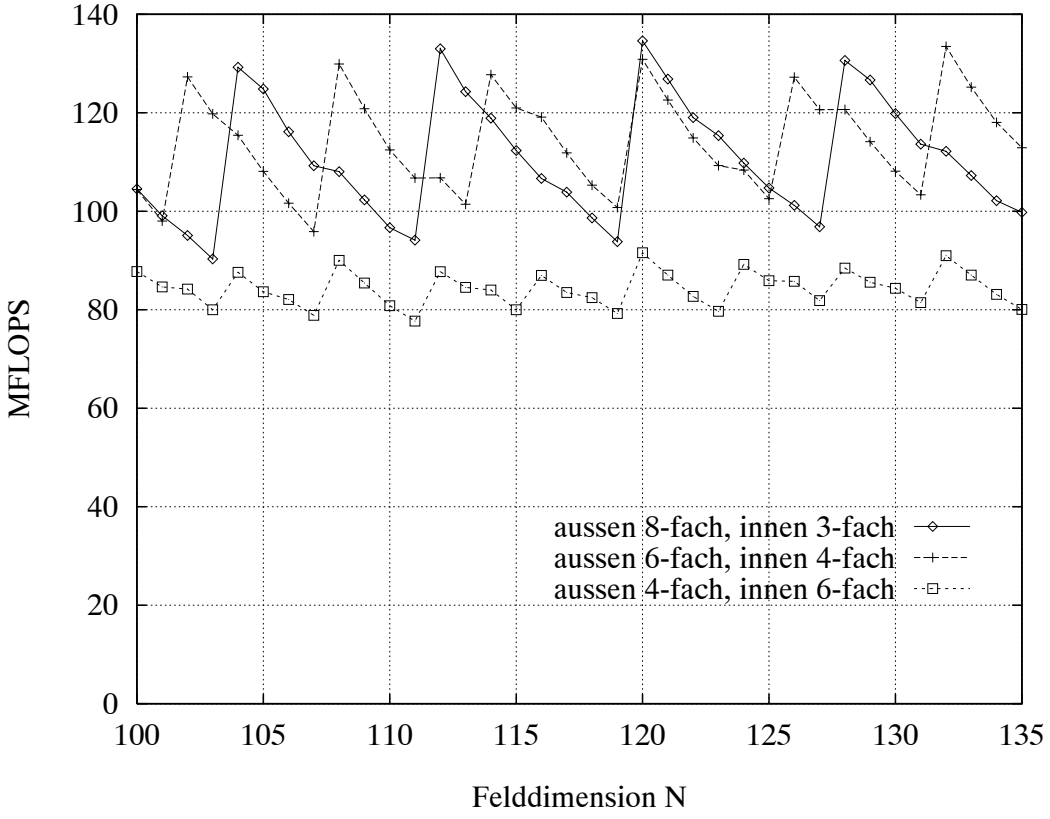


Abbildung 6.3: Performance-Messung, Beispiel 6.1: Innere und äußere Schleife entfaltet, Optimierungsoption **-O3**

Wie in Abbildung 6.3 liegen die gemessenen Performance-Werte zum größten Teil über den vorher gemessenen Werten bei ausschließlicher Entfaltung der äußeren Schleife. Interessant ist dabei, daß sich die Länge des 'Sägezahns' nach dem Entfaltungsfaktor der äußeren Schleife richtet. Dies liegt an der doppelt geschachtelten Korrekturschleife am Ende des Programmteils (Beispiel 6.1c) die für ungünstige Werte des äußeren Entfaltungsfaktoren sehr oft durchlaufen werden muß. Die ma-

ximale Performance wird erreicht, wenn keine Korrekturschleife durchlaufen werden muß, d.h. der Entfaltungsfaktor der inneren Schleife ganzzahlig durch die Iterationszahl der inneren Schleife teilbar ist und dasselbe entsprechend für die äußere Schleife gilt.

Bei diesem Beispiel wurde beobachtet, daß ab einer Anzahl von 50 Anweisungen im Basisblock des zu übersetzenden Programmteils, keine zusätzliche vierfache Entfaltung durch den Compiler bei Benutzung der Optimierungsstufen -O3 bzw. -O4 durchgeführt wurde. Dementsprechend weist die Version mit achtfacher Entfaltung der äußeren und vierfacher Entfaltung der inneren Schleife eine geringere Performance auf, da der Basisblock nur noch aus 32 anstelle von 96 Anweisungen besteht.

Fazit

Unter folgenden Bedingungen sind in Anwendungsprogrammen hohe Performance-Werte zu erwarten:

- Es bestehen keine echten Datenabhängigkeiten im Basisblock, die ein effektives Pipelining verhindern.
- Es existieren Feldelemente, für die eine zeitliche Wiederverwendung besteht, so daß durch *scalar replacement* eine effektivere Registernutzung erreicht wird.
- Der Basisblock ist möglichst lang, so daß ein effektives Instruktions-Scheduling durchgeführt werden kann.
- Es darf nur so stark entfaltet werden, daß keine Zwischenspeicherung von Registerinhalten im Cache-Speicher notwendig ist.

6.2 Matrixmultiplikation

Die Matrixmultiplikation stellt zur Anwendung der im vorigen Kapitel vorgestellten Transformationstechniken 'Schleifenblocken und -entfalten' ein geeignetes Beispiel dar, weil keine Datenabhängigkeiten die Permutation der Schleifen verhindern und eine hohe Wiederverwendung besteht, die wesentliche Performance-Verbesserungen gegenüber dem untransformierten Code erwarten läßt.

```
DO I = 1, N
  DO K = 1, N
    DO J = 1, N
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
    ENDDO
  ENDDO
ENDDO
```

Wie aus dem Code für die Matrixmultiplikation (*JKI*-Form) zu entnehmen ist, wird jedes Feldelement `N`-fach wiederverwendet:

- `A(I,K)` in `J`-Schleife,
- `C(I,J)` in `K`-Schleife,
- `B(K,J)` in `I`-Schleife.

Welche Performance-Steigerung bei der Matrixmultiplikation auf der untersuchten Alpha-Workstation erzielt werden kann, wird deutlich, wenn man den nichtoptimierten Code mit der DEC DGEMM-Routine[1] aus der Digital Extended Math Library für DEC OSF/1 AXP vergleicht.

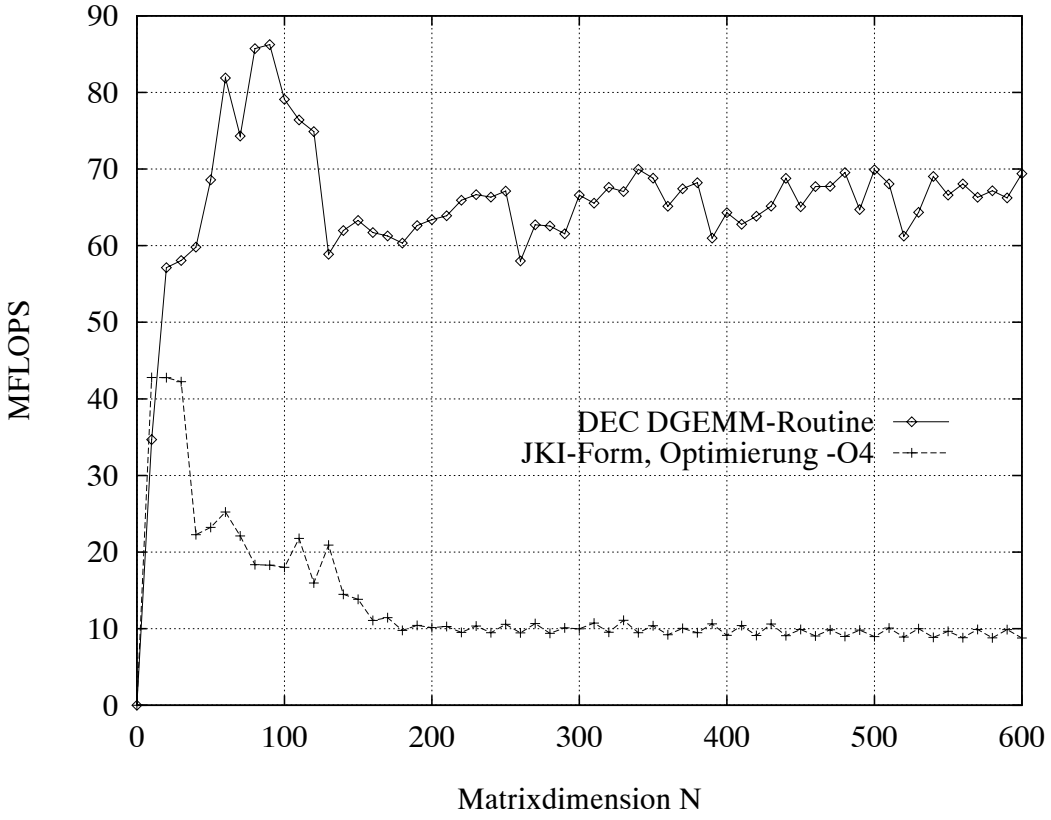


Abbildung 6.4: Performance-Messung Matrixmultiplikation: Nichttransformierte *JKI*-Form und DEC DGEMM-Routine

Aus Abbildung 6.4 geht hervor, daß der Mittelwert der erreichten Performance für quadratische Matrizen mit 64-Bit-Fließpunktwerten für $N > 130$ bei ca. 60 MFLOPS liegt, was eine Leistungssteigerung gegenüber der nichttransformierten Version um den Faktor 6 bedeutet. Der starke Abfall bei der Dimension $N = 130$ kann auf die schlechtere Ausnutzung des internen Cache-Speichers zurückgeführt werden. Eine eindeutige Aussage ist hier jedoch nicht möglich, weil der Source-Code der Routine

[1] DGEMM berechnet das Matrix-Matrix Produkt für 64-Bit-Werte (Double Precision).

nicht vorlag. Die Schwankungen der Performance resultieren aus dem Abbildungsmechanismus des direkt abbildenden Cache-Speichers: Da die drei Felder nacheinander im Hauptspeicher abgelegt werden, bewirkt eine Änderung der Dimension die Verschiebung der Startadresse des zweiten und dritten Feldes. Die Folge ist ein unterschiedliches Auslöschungsverhalten bei verschiedener Dimensionswahl aufgrund der anderen Abbildung auf den Cache-Speicher. Da die Alpha-AXP Architektur bisher keine Instruktionen vorsieht, die eine Kontrolle der Cache-Speicherbelegung ermöglichen, ist die ungewollte Aufnahme von Daten in den Cache-Speicher nicht zu vermeiden. Dies führt zu einer erheblichen Reduktion der effektiv ausnutzbaren Cache-Speichergröße und damit zur Verringerung der erreichbaren Performance. Wie man an den Schwankungen der DEC DGEMM-Performance-Kurve in Abbildung 6.4 erkennt, ist dies auch durch Anwendung von manuellen Optimierungen, wie z.B. der Veränderung des Zugriffsmusters und des Alignments der Felder im Hauptspeicher nicht ganz zu vermeiden.

Die nicht transformierte *JKI*-Version der Matrixmultiplikation zeigt für `N` > 200 einen konstanten Performance-Wert von 10 MFLOPS. Der Verlauf der Kurve ist deshalb so glatt, weil für keine Problemgröße die zeitliche Wiederverwendung von `A(I,K)` und `C(I,J)` ausgenutzt wird und somit Interferenzen in beiden Cache-Speichern keine sichtbaren Auswirkungen haben. Die Differenzen in der örtlichen Wiederverwendung bei verschiedenen Problemgrößen ist zu vernachlässigen. In dem Bereich bis `N` $= 200$ macht sich die unterschiedliche Ausnutzung der zeitlichen Wiederverwendung bemerkbar. Hierauf wird in Abschnitt 6.2.2 eingegangen.

Aus der Anleitung zur Benutzung der Bibliotheksroutinen geht hervor, daß die erzielte Performance der DEC DGEMM-Routine durch die Verbesserung der Datenlokalität in Registern, Cache-Speicher und Hauptspeicher erreicht wird. Dies bestätigt den im vorigen Kapitel vorgestellten Ansatz, durch Schleifentransformationen auf Hochsprachenebene die Performance von Schleifenkonstrukten mit hohem Wiederverwendungsgrad zu optimieren.

Die Matrixmultiplikation wird wegen der guten Optimierungsmöglichkeiten oft als Demonstrationsbeispiel für die effektive Ausnutzung von Rechnern mit Cache-Speicher verwendet [14, 23, 25]. Homberg und Hake untersuchen in [14] die Performance der Matrixmultiplikation auf dem Vektorrechner IBM 3090. Dieser Rechner verfügt über einen vierfach assoziativen 64 KByte großen Cache-Speicher mit einer Zeilenlänge von 128 Byte. Ähnlich wie in Abbildung 6.4 treten auch hier bei einigen Matrixdimensionen ausgeprägte Minima der Performance-Werte auf, die durch Lesefehler im Cache-Speicher und im Datenadreßpuffer (TLB) verursacht werden.

Liu und Strother behandeln in [25] die Optimierung von FORTRAN-Programmen auf der IBM 3090. Dabei wird insbesondere auf die Vektorisierungsoptionen des VS FORTRAN-Compilers und das manuelle Blocken von Schleifen zur Verbesserung des Speicherzugriffsverhaltens eingegangen. Der transformierte Code erreichte bei der

Wahl geeigneter Blockfaktoren nahezu die Performance der ESSL-Routine[2] DGEMUL. Eine manuelle Schleifenentfaltung mußte nicht vorgenommen werden, da die Vektorisierungsoption des Compilers diese Transformation automatisch vornimmt. Desweiteren wurde der optimale Blockfaktor nicht explizit hergeleitet, sondern durch Versuche bestimmt.

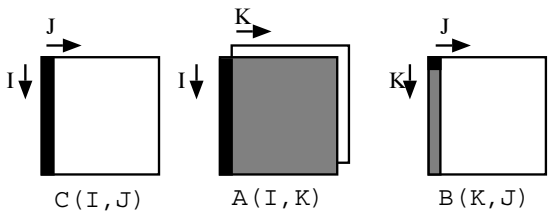
Die Berechnung des optimalen Blockfaktors bei unterschiedlichen Matrixdimensionen beschreiben Lam, Rothberg und Wolf in [23]. Sie untersuchten die Performance der Matrixmultiplikation auf einer DEC-Station 3100, die über einen direkt abbildenden internen Cache-Speicher mit einer Kapazität von 8 KByte und einer Zeilenlänge von 256 Bit verfügt. Dies entspricht, mit Ausnahme des externen Cache-Speichers, dem Aufbau des Speichersystems des in dieser Arbeit untersuchten DEC Alpha-Rechners. In Abschnitt 6.2.3 wird untersucht, inwiefern die gemachten Aussagen bestätigt werden können.

In keiner der angeführten Literaturstellen wird ausführlich auf das gemeinsame Anwenden von Cache-Speicher- und Registeroptimierung eingegangen. In [23] wird das Blocken für Register (Schleifenentfaltung) zwar erwähnt, es wird jedoch kein Hinweis gegeben, welche Schleifen zu entfalten sind und welche Permutation benutzt werden soll. Die gemeinsame Betrachtung von Register- und Cache-Speicheroptimierung ist jedoch notwendig, um das Potential der Wiederverwendung voll auszuschöpfen. Im folgenden soll gezeigt werden, wie ohne vorherige Messungen die notwendigen Transformationen ermittelt werden können und Performance-Werte erreicht werden, die zumindest in der Nähe der optimierten Bibliotheksroutine liegen. Um die Untersuchungen nicht zu umfangreich zu gestalten, werden folgende Einschränkungen festgelegt:

- Es werden keine Seitenfehler betrachtet, d.h. die untersuchten Datenfelder passen alle in den Hauptspeicher.
- Es werden nur quadratische Matrizen betrachtet, da ähnliche Aussagen auch für rechteckige Matrizen zutreffen.
- Optimierungen hinsichtlich des Datenadreßpuffers und des Hauptspeichers werden nicht vorgenommen.
- Gemessen wird nur die Durchführungszeit der Matrixmultiplikation. Die Initialisierung der Matrizen sowie das Nullsetzen der Ergebnismatrix wird nicht in die Zeitmessung einbezogen.
- Der externe Cache-Speicher wird nur bei der Berechnung der optimalen Blockfaktoren in Abschnitt 6.2.3 betrachtet.

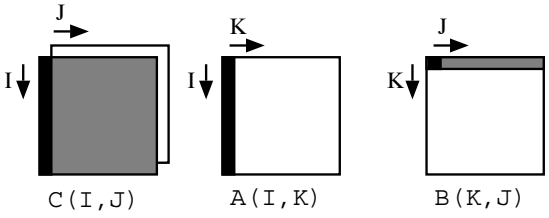
[2] IBM Engineering and Scientific Subroutine Library

JKI-Form



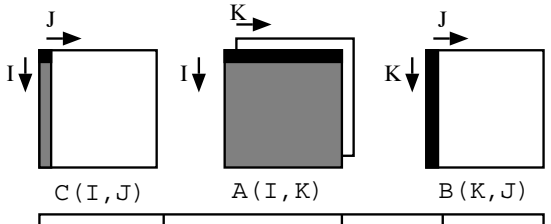
Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe Iteration
C(I,J)	K	I	I	1/4
A(I,K)	J	I	I	1/4
B(K,J)	I	K	I	1/N
1<<N			**Gesamt:**	**≈ 1/2**

KJI-Form



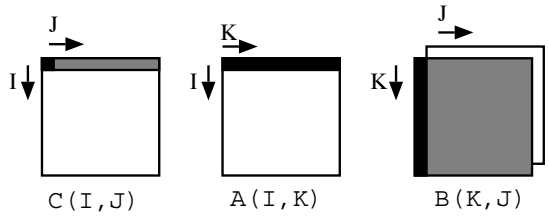
Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe Iteration
C(I,J)	K	I	I	1/4
A(I,K)	J	I	I	1/4
B(K,J)	I	K	I	1/N
1<<N			**Gesamt:**	**≈ 1/2**

JIK-Form



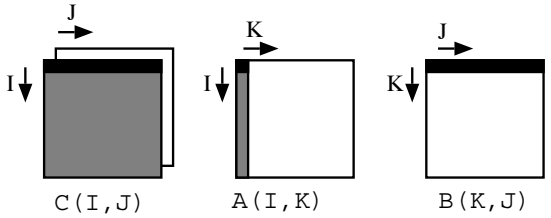
Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe Iteration
C(I,J)	K	I	K	1/N
A(I,K)	K	I	K	1
B(K,J)	I	K	K	1/4
1<<N			**Gesamt:**	**≈1+1/4**

IJK-Form



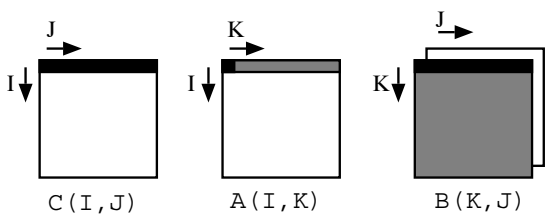
Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe Iteration
C(I,J)	K	I	K	1/N
A(I,K)	J	I	K	1
B(K,J)	I	K	K	1/4
1<<N			**Gesamt:**	**≈1+1/4**

KIJ-Form



Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe Iteration
C(I,J)	K	I	J	1
A(I,K)	J	I	J	1/N
B(K,J)	I	K	J	1
1<<N			**Gesamt:**	**≈ 2**

IKJ-Form



Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe Iteration
C(I,J)	K	I	J	1
A(I,K)	J	I	J	1/N
B(K,J)	I	K	J	1
1<<N			**Gesamt:**	**≈2**

Abbildung 6.5: Matrixmultiplikation: Mögliche Permutationen

6.2.1 Optimierung für Cache-Speicher

Als erste Optimierungsmaßnahme für den Cache-Speicher soll die Schleifenvertauschung untersucht werden. Insgesamt sind sechs verschiedene Permutationen des Matrixmultiplikations-Codes möglich, die die Wiederverwendung der Daten aus `A(I,K)`, `B(K,J)` und `C(I,J)` in unterschiedlicher Weise ausnutzen. In Abbildung 6.5 sind Zugriffsmuster, Wiederverwendung, Lokalität und die notwendigen Ladeoperationen für Daten im externen Speicher angegeben. Die Muster in den Abbildungen sind den verschiedenen Schleifenebenen zugeordnet:

- äußere Schleife weiß;
- mittlere Schleife grau;
- innere Schleife: schwarz.

Der doppelt ausgeführte Rahmen deutet an, daß dieses Feld komplett durchlaufen wird bevor es zu einer Wiederverwendung der Daten kommt. Am Beispiel der *JKI*-Form soll die graphische Darstellung und die Berechnung der ausgenutzten Wiederverwendung erläutert werden. Dabei wird angenommen, daß die Datenlokalität nur in der inneren Schleife besteht, d.h. $N \gg 1$.

Das Feld `C(I,J)` wird spaltenweise durchlaufen. Die Spaltenelemente werden in der mittleren Schleife `N`-fach wiederverwendet, was jedoch nicht ausgenutzt werden kann, da das erste Element `C(1,1)`, welches in der Iteration (1,1,1) geladen wurde, sich zum Zeitpunkt der potentiellen selbst-zeitlichen Wiederverwendung in (2,1,1) aufgrund der obigen Annahme nicht mehr im Cache-Speicher befindet. Dagegen kann die selbst-räumliche Wiederverwendung ausgenutzt werden: Mit `C(1,1)` werden gleichzeitig die nächsten Spaltenelemente `C(2,1)`, `C(3,1)` und `C(4,1)` in dieselbe Zeile geladen, so daß der externe Speicherzugriff für diese Elemente in den nächsten drei Iterationen entfällt. Damit reduziert sich die Anzahl der notwendigen Ladeoperationen für `C(I,J)` von eins auf 1/4 Ladeoperationen pro Iteration. Das Auftreten des Ausdrucks `C(I,J)` auf der rechten Seite der Anweisung hat keine Auswirkung auf die Berechnung der Hauptspeicherzugriffe, da die Lokalität von Daten im Cache-Speicher nur die Anzahl der Ladeoperationen reduziert. Auf Speichervorgänge hat die Datenlokalität keinen Einfluß. Das Feld `A(I,K)` wird ebenfalls spaltenweise durchlaufen, jedoch werden die Feldelemente erst in der äußeren Schleife `N`-fach wiederverwendet. Zur Ausnutzung der selbst-zeitlichen Wiederverwendung muß die gesamte Matrix im Cache-Speicher Platz finden, was laut Annahme nicht möglich ist. Die selbst-räumliche Wiederverwendung kann wie oben ausgenutzt werden.

Auch das Feld `B(K,J)` wird spaltenweise durchlaufen. Der Index von `B(K,J)` ändert sich in der inneren Schleife nicht, so daß der Wert in einem Register gehalten werden kann. Die Ausnutzung dieser selbst-zeitlichen Wiederverwendung reduziert die Hauptspeicherzugriffe pro Iteration für `B(K,J)` um den Faktor 1/`N`. Beim Addieren der Zugriffe kann der Faktor 1/`N` gegenüber 1/4 gemäß der Voraussetzung $N \gg 1$

vernachlässigt werden. Durchschnittlich sind also pro Iteration nur 0.5 Ladeoperationen aus dem externen Speicher notwendig.

Vergleicht man die Zugriffe der verschiedenen Permutationen, so ergibt sich eine Rangfolge, die durch Messung verifiziert werden kann. Bei der Erzeugung des Codes für alle folgenden Messungen wird die Optimierungsstufe **-O2** benutzt, da bei den Optionen **-O3** und **-O4** eine vierfache Schleifenentfaltung durchgeführt wird, die Einfluß auf die Registerbelegung und damit auf die Performance hat.

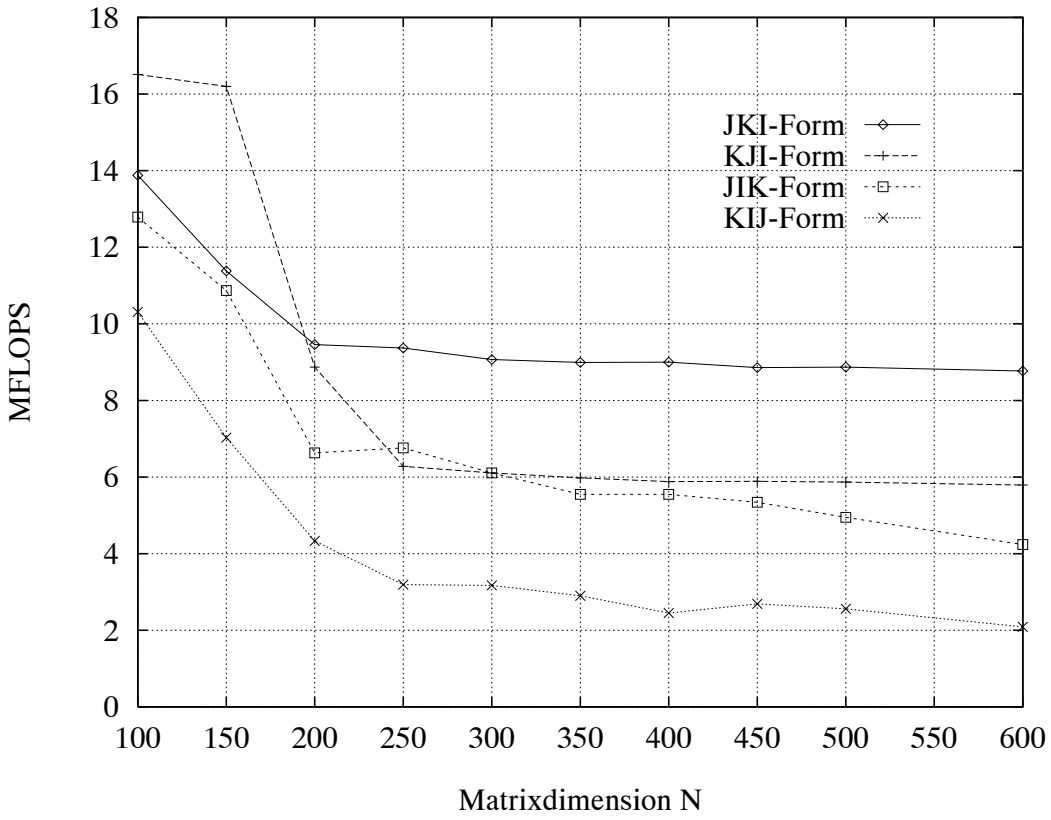
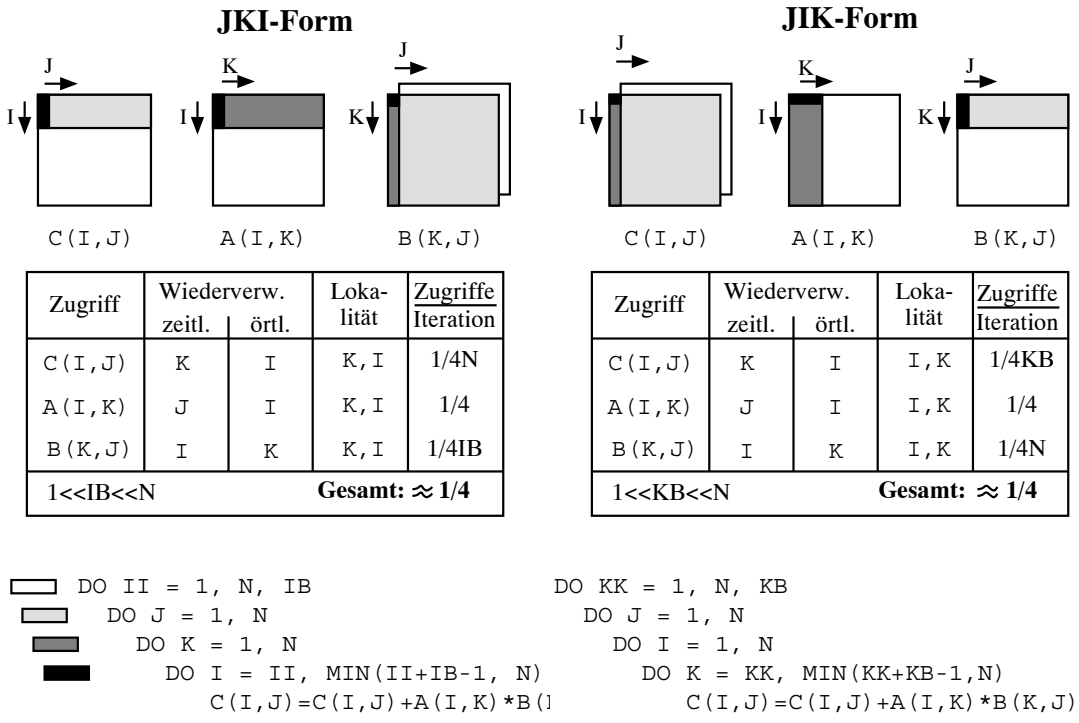


Abbildung 6.6: Performance-Messung Matrixmultiplikation: Verschiedene Permutationen, nicht optimiert

Wie aus Abbildung 6.6 hervorgeht, stimmen die gemessenen Werte mit den berechneten Werten in Abbildung 6.5 für Dimensionsgrößen von `N` > 200 überein. Die Werte für die Permutationen *IJK* und *IKJ* wurden aus Gründen der Übersicht weggelassen. Ihre Performance-Werte entsprechen denen ihrer linken Nachbarn in Abbildung 6.5. Für kleinere Problemgrößen besteht noch Lokalität einiger Daten im internen und externen Cache-Speicher. Die Quantifizierung der Wiederverwendung aus Abbildung 6.5 ermöglicht keine exakte Vorhersage dieser teilweise vorhandenen Lokalität. Die gewonnenen Werte sollen vielmehr als Richtwert dienen, inwiefern das Blocken von weiter außen liegenden Schleifen eine bessere Ausnutzung der Wiederverwendung verspricht. Die Wahl des optimalen Blockfaktors und die Vorhersage der Cache-Speicherbelegung setzt ein Modell des Zugriffsverhaltens der einzelnen

Feldausdrücke voraus. Hierauf wird in Abschnitt 6.2.3 eingegangen. Das schlechtere Abschneiden der *KJI*-Form gegenüber der *JKI*-Form bei großen Problemgrößen ist auf Lesefehler im Datenadreßpuffer zurückzuführen, die durch den zeilenweisen Zugriff von `B(K,J)` verursacht werden.



Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe / Iteration
C(I,J)	K	I	K,I	1/4N
A(I,K)	J	I	K,I	1/4
B(K,J)	I	K	K,I	1/4IB
1<<IB<<N			**Gesamt: ≈ 1/4**	

Zugriff	Wiederverw. zeitl.	Wiederverw. örtl.	Lokalität	Zugriffe / Iteration
C(I,J)	K	I	I,K	1/4KB
A(I,K)	J	I	I,K	1/4
B(K,J)	I	K	I,K	1/4N
1<<KB<<N			**Gesamt: ≈ 1/4**	

```
DO II = 1, N, IB
  DO J = 1, N
    DO K = 1, N
      DO I = II, MIN(II+IB-1, N)
        C(I,J)=C(I,J)+A(I,K)*B(
```

```
DO KK = 1, N, KB
  DO J = 1, N
    DO I = 1, N
      DO K = KK, MIN(KK+KB-1,N)
        C(I,J)=C(I,J)+A(I,K)*B(K,J)
```

Abbildung 6.7: Matrixmultiplikation: *JKI*- und *JIK*-Form, innere Schleife geblockt

Somit ist bei alleiniger Anwendung der Schleifenvertauschung und der Betrachtung von Problemgrößen `N` > 200 die *JKI*-Formen die beste Wahl: Da alle Felder spaltenweise durchlaufen werden, besteht für diese Anordnungen die größte Lokalität. Die Vermutung liegt nahe, daß diese Schleifenanordnung auch geblockt die beste Wahl darstellt. Wie wir jedoch im nächsten Absatz sehen werden, kann dies aus den theoretischen Überlegungen zur Wiederverwendung nicht abgeleitet werden.

Blocken der inneren Schleife

Betrachtet man die Wiederverwendung der Felder in der *JKI*-Form, so fällt auf, daß bei der Referenzierung von `B(K,J)` die zeitliche Wiederverwendung optimal ausgenutzt wird[3]. Die starke Reduktion der notwendigen Speicherzugriffe um den Faktor $1/$`N` wird jedoch durch die geringe Lokalität der Feldelemente von `A(I,K)` und `C(I,J)` aufgehoben. In der Summe der Speicherzugriffe, die letztendlich die Durchlaufzeit bestimmt, ist $1/$`N` bei `N` $\gg 1$ gegenüber $1/4$ zu vernachlässigen. Das Ziel muß es

[3]Ab der Optimierungsstufe **-O2** wird `B(K,J)` in der inneren Schleife in einem Register gehalten.

nun sein, die Lokalität der Referenzen `A(I,K)` und `C(I,J)` durch Ausnutzung ihrer zeitlichen Wiederverwendung zu verbessern, so daß die Gesamtzugriffszeit minimal wird.

Durch das Blocken der inneren Schleife kann in allen Permutationen die zeitliche Wiederverwendung eines zusätzlichen Ausdruckes ausgenutzt werden. In Abbildung 6.7 ist dies für die *JKI*- und *JIK*-Form dargestellt. Im Fall der *JKI*-Form wird sowohl die zeitliche, als auch die örtliche Wiederverwendung von `C(I,J)` und `B(K,J)` ausgenutzt: Durch die Begrenzung der inneren Schleife wird die Datenlokalität auch auf die mittlere Schleife ausgedehnt, so daß zusätzlich zur örtlichen auch die zeitliche Wiederverwendung der `KB` Spaltenelemente von `C(I,J)` ausgenutzt werden kann. Bei der *JIK*-Form erhöhen sich zwar die Lesezugriffe für `C(I,J)` von `1/N` auf `1/4KB` pro Iteration, dagegen wird zusätzlich die zeitliche und örtliche Wiederverwendung der `IB` Spaltenelemente von `B(K,J)`, in Abbildung 6.7 schwarz markiert, in der `I`-Schleife ausgenutzt. Das Ziel, die Gesamtzahl der Referenzierungen zu minimieren, ist jedoch noch nicht erreicht, da der Summand 1/4 von `A(I,K)` in beiden Permutationen immer noch bestimmend ist. Im Vergleich mit Abbildung 6.5 muß zwar nicht mehr das ganze Feld durchlaufen werden, bevor es zur Wiederverwendung kommt, jedoch ist die einseitige Reduzierung in `I`- bzw. `K`-Richtung für die gewünschte Datenlokalität von `A(I,K)` noch nicht ausreichend.

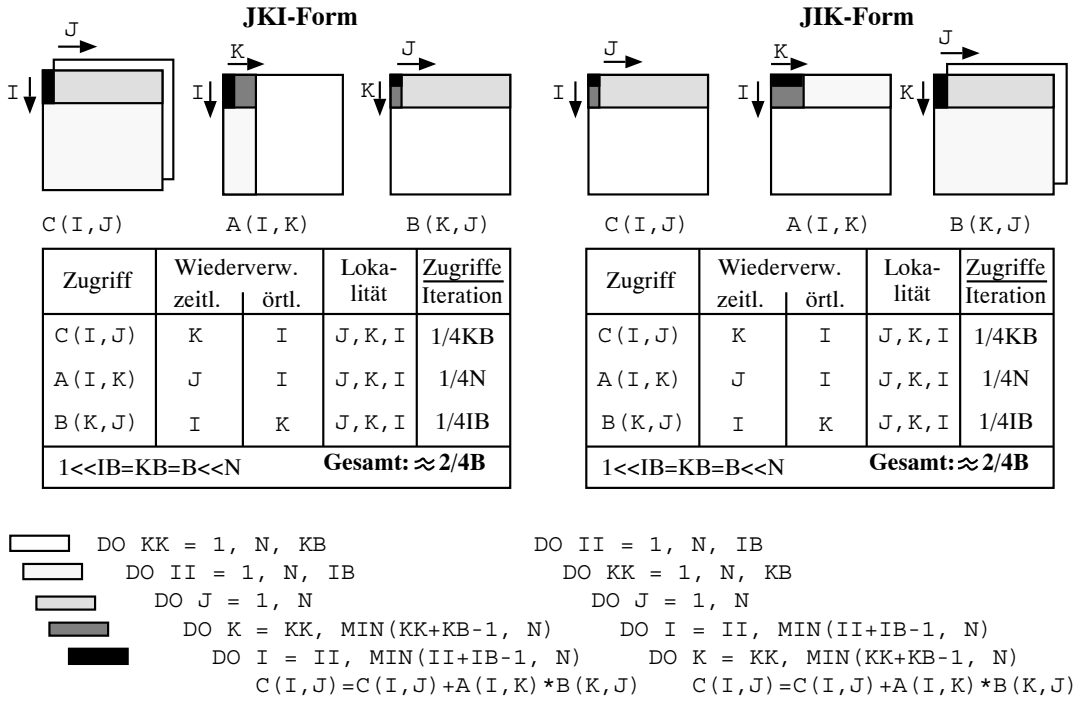


Abbildung 6.8: Matrixmultiplikation: *JKI*- und *JIK*-Form, innere und mittlere Schleife geblockt

Blocken der inneren und mittleren Schleife

Im nächsten Schritt wird nun auch die mittlere Schleife geblockt, was dazu führt, daß für alle Permutationen der maximale Grad an Wiederverwendung ausgenutzt wird. Wie aus der Abbildung 6.8 hervorgeht, besteht nun für einen quadratischen Block von `A(I,K)` Datenlokalität in der äußeren Schleife. Anstelle zweier identischer Blockfaktoren können auch unterschiedliche Werte benutzt werden, so daß anstelle des Quadrates ein Rechteck tritt. Dies hat den Vorteil, daß das Zugriffsmuster auf die Felder noch feiner variiert werden kann. Hierauf wird in Abschnitt 6.2.3 bei der Beschreibung des realen Cache-Speicherverhaltens eingegangen.

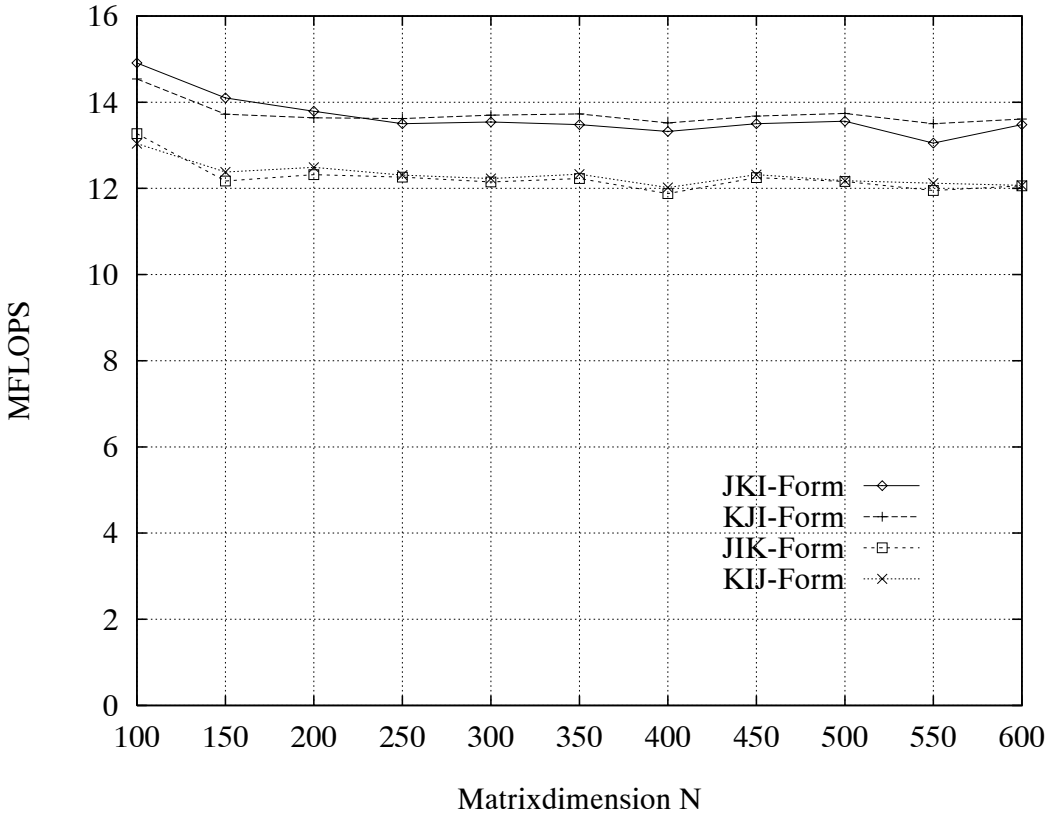


Abbildung 6.9: Performance-Messung Matrixmultiplikation: Verschiedene Permutationen, innere und mittlere Schleife geblockt

Wie schon bei der einfachen Blockung ist die berechnete Lokalität für alle doppelt geblockten Permutationen dieselbe, so daß keine Randbedingung besteht, welche Form für die Registeroptimierung zu bevorzugen ist. Zu diesem Ergebnis kommt auch Samukawa in [30] bei der Beschreibung von Programmierregeln zur besseren Ausnutzung des Speichersystems der IBM 3090. Er wählt diejenige Permutation aus, die für die Vektorisierung durch den IBM FORTRAN-Compiler am günstigsten ist. Da der untersuchte Alpha-Chip ein skalarer Prozessortyp ist, tritt an die Stelle der Vektorisierungsbefehle das Entrollen der Schleifen und Belegen der Register durch *scalar replacement*. Aus Abbildung 6.9 geht hervor, daß die alleinige Anwendung von Blocktransformationen für die angestrebte Performance-Steigerung

nicht ausreicht. Erst mit der im nächsten Abschnitt besprochenen zusätzlichen Registeroptimierung können Performance-Werte erreicht werden, die dem Ziel von 60 MFLOPS nahekommen.

6.2.2 Optimierung für Register

Ähnlich wie im letzten Abschnitt bietet es sich auch bei der Matrixmultiplikation an, die Schleifen so anzuordnen, daß die Indizes derjenigen Feldreferenz in der inneren Schleife konstant sind, die sowohl lesende als auch schreibende Zugriffe verursacht. In diesem Fall ist das `C(I,J)` und die in Frage kommenden Schleifenanordnungen sind dementsprechend die *JIK*- und die *IJK*-Form. Nach Entfaltung der äußeren `I`- und `J`-Schleife und *scalar replacement* für die Feldelemente von `C(I,J)` kann der Schreib- und Ladebefehl für `C(I,J)` außerhalb des Basisblocks durchgeführt werden.

Die günstigsten Entfaltungsfaktoren ergeben sich aus mehreren Randbedingungen.

- Um möglichst viele Speicher- und Schreibzugriffe zu vermeiden, sollten während der Ausführung der inneren `K`-Schleife möglichst viele Feldelemente von `C(I,J)` in Registern gehalten werden.
- Die einzelne Schleife sollte nicht zu stark entfaltet werden, da ansonsten eine starke Abhängigkeit der Performance von der Problemgröße zu erwarten ist. Dieser Effekt wurde bereits in Kapitel 6.1 anhand des sägezahnartigen Verlaufs der Performance-Kurve in Abbildung 6.2 beschrieben.
- Die `I`-Schleife sollte vierfach entfaltet werden, damit die örtliche Wiederverwendung der Feldelemente `C(I+1,J)`,...,`C(I+3,J)` und `A(I+1,K)`,...,`(I+3,K)` ausgenutzt werden kann. Da das Ergebnis einer Fließpunktaddition frühestens nach vier Takten verfügbar ist führt eine niedrigere Entfaltung der `I`-Schleife zu Erzeuger/Benutzer-Konflikten bezüglich der Feldelemente von `C(I,J)`.
- Die Anzahl der zur Verfügung stehenden Fließpunktregister bildet die obere Grenze für die Wahl der Entfaltungsfaktoren.

Diese Forderungen führen zu dem Ergebnis, die `J`-Schleife fünffach und die `I`-Schleife vierfach zu entfalten. Wie aus Beispiel 6.2 ersichtlich, werden die Register im Basisblock der `K`-Schleife wie folgt belegt:

- 4 Register mit Daten aus `A(I,K)`,
- 5 Register mit Daten aus `B(K,J)` und
- 20 Register mit Daten aus `C(I,J)`.

Eine stärkere Entfaltung der Schleifen würde eine Zwischenspeicherung der Registerinhalte im Speicher notwendig machen und die Performance entsprechend vermindern.

Beispiel 6.2

```
...
DO J = 1, NJ-4, 5
   DO I = ZI1, ZI2-3, 4
      C1 =  C(I  ,J)
      C2 =  C(I+1,J)
      C3 =  C(I+2,J)
      C4 =  C(I+3,J)
      ...
      C17 = C(I  ,J+4)
      C18 = C(I+1,J+4)
      C19 = C(I+2,J+4)
      C20 = C(I+3,J+4)
      DO K = ZK1, ZK2
         C1  = C1   + A(I  ,K) * B(K,J)
         C2  = C2   + A(I+1,K) * B(K,J)
         C3  = C3   + A(I+2,K) * B(K,J)
         C4  = C4   + A(I+3,K) * B(K,J)
         ...
         C17 = C17  + A(I  ,K) * B(K,J+4)
         C18 = C18  + A(I+1,K) * B(K,J+4)
         C19 = C19  + A(I+2,K) * B(K,J+4)
         C20 = C20  + A(I+3,K) * B(K,J+4)
      ENDDO
      C(I  ,J)   = C1
      C(I+1,J)   = C2
      C(I+2,J)   = C3
      C(I+3,J)   = C4
      ...
      C(I  ,J+4) = C17
      C(I+1,J+4) = C18
      C(I+2,J+4) = C19
      C(I+3,J+4) = C20
   ENDDO
ENDDO
...
```

Die Entfaltung einer einzigen Schleife, beispielsweise die `I`-Schleife, verhindert zwar den zeilenweisen Zugriff auf Feldelemente von `B(K,J)`, was, wie in Abschnitt 4.2.4 gezeigt, für einige Operationen ein verbessertes Instruktions-Scheduling ermöglicht. Da jedoch die Elemente von `A(I,K)` in der inneren Schleife nicht wiederverwendet werden, ist maximal ein 15-faches Entfalten der `I`-Schleife möglich. Dadurch werden im Basisblock insgesamt 10 Fließpunktoperationen weniger ausgeführt als bei dem Programm-Code in Beispiel 6.2. Messungen haben ergeben, daß diese Entfaltung eine niedrigere Performance und eine stärkere Abhängigkeit von der Problemgröße zur Folge hat.

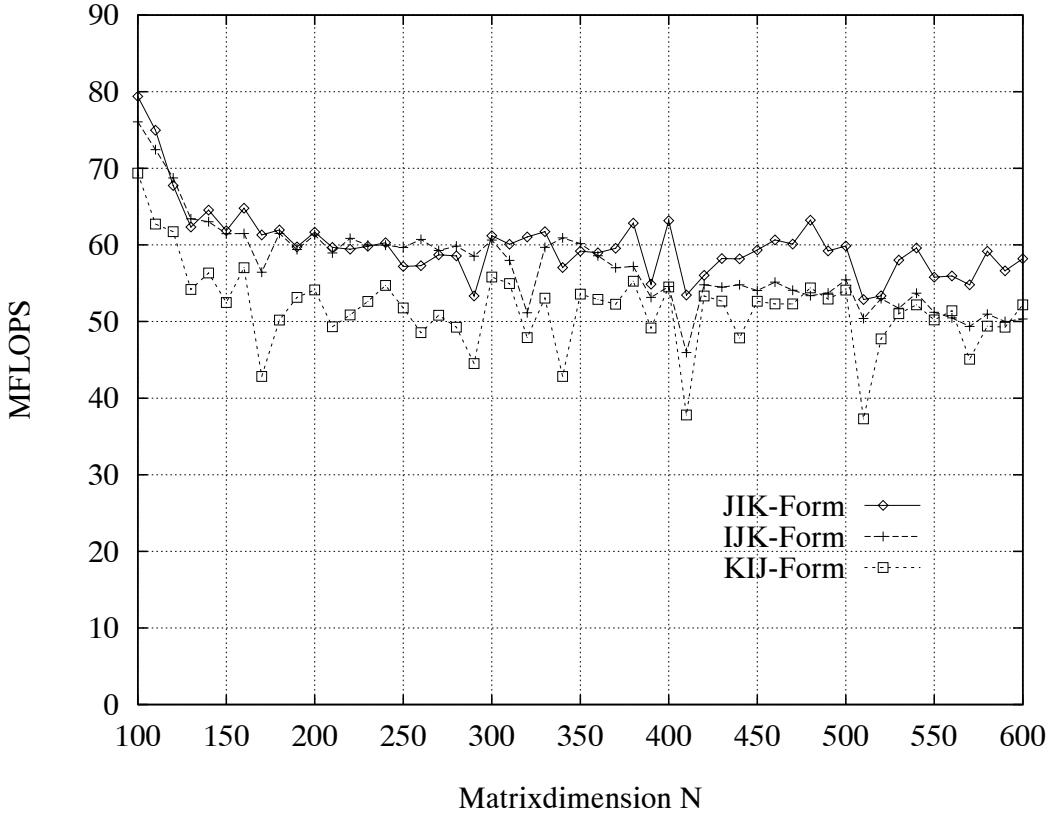


Abbildung 6.10: Performance-Messung Matrixmultiplikation: Verschiedene Permutationen, geblockt und entfaltet

Wie in Abbildung 6.10 dargestellt erfüllt sich die Voraussage, daß für die Schleifenformen, die ein *scalar replacement* für `C(I,J)` ermöglichen, die besten Performance-Werte erzielt werden. Dabei erreicht die *KIJ*-Form im Vergleich zu den anderen nicht aufgeführten Permutationen die beste Performance. Der Grund hierfür ist das effektive Pipelining, die Verminderung von Hauptspeicherzugriffen durch die örtliche Wiederverwendung der Feldelemente und die Reduktion von Cache-Speicherzugriffen durch die Wiederverwendung von Registerinhalten. Die Differenzen in den Performance-Werten der *JIK*- bzw. *IJK*-Form sind auf die unterschiedlichen Korrekturschleifen zurückzuführen.

Für die Messungen wurden die jeweils inneren zwei Schleifen mit identischen Faktoren geblockt und für die jeweilige Matrixdimension der maximale Performance-Wert durch Variation des Blockfaktors ermittelt. Alle Programme wurden mit der Optimierungsoption **-O4** übersetzt. Der Einfluß verschiedener Blockfaktoren auf die Performance wird im nächsten Abschnitt untersucht.

Aus Abbildung 6.11 geht hervor, da"die erreichbare Performance durch die Hauptspeicherzugriffszeit begrenzt ist. Auf dem niedriger getakteten, aber dafür mit einem breiteren Datenbus ausgestatteten Modell 400 werden für alle gemessenen Matrixdimensionen höhere Performance-Werte gemessen, als auf dem Modell 300.

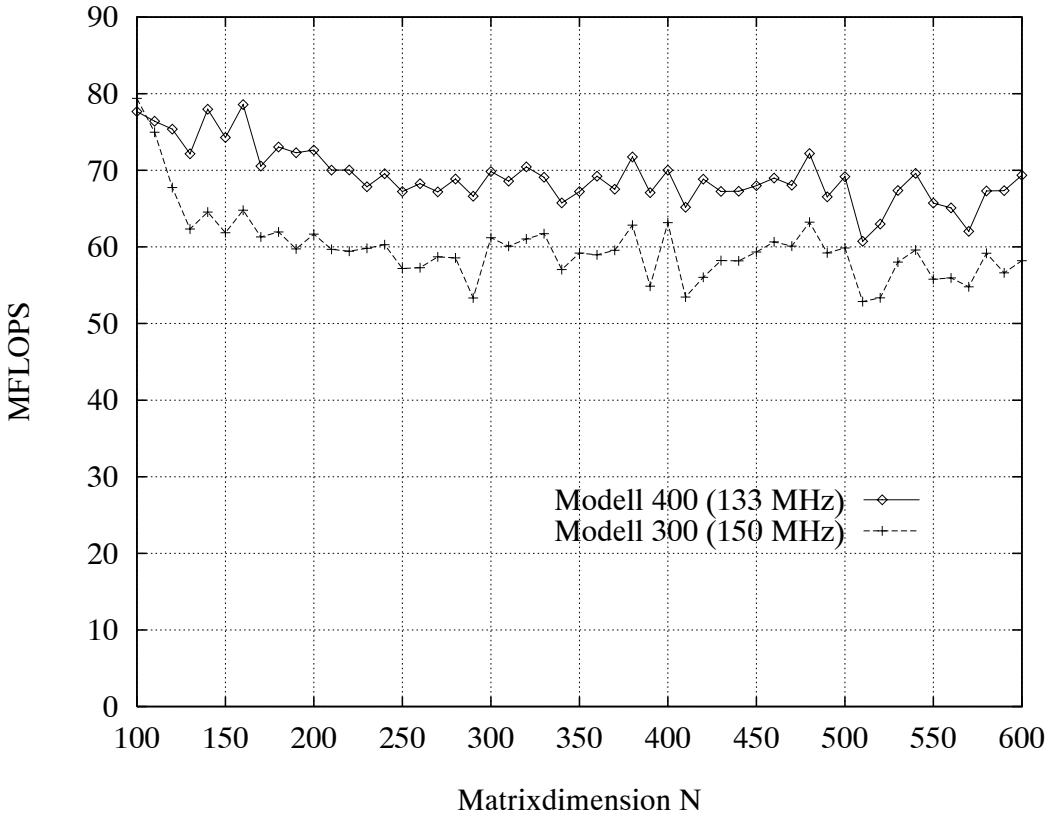


Abbildung 6.11: Performance-Messung Matrixmultiplikation: Modell 300 und Modell 400, *JIK*-Form

Die Schwankungen der Performance-Werte ist auf eine Auslöschung der Daten im Cache-Speicher zurückzuführen. Im nächsten Abschnitt wird der Einfluß der Matrixdimensionen und der Blockfaktoren auf die Performance beschrieben. Desweiteren wird eine Möglichkeit vorgestellt, wie mit der Benutzung eines festen Blockfaktors, konstant hohe Performance-Werte für unterschiedliche Matrixdimensionen erzielt werden können.

6.2.3 Reales Cache-Speicherverhalten

Bei der bisherigen Ermittlung der Datenlokalität im Cache-Speicher wurde das Auslöschen von Cache-Speicherdaten durch den Abbildungsmechanismus nicht berücksichtigt. Um den starken Einfluß der Matrixdimension auf die Performance zu verdeutlichen, sind in Abbildung 6.12 die Performance-Werte für einen kleineren Dimensionsbereich aufgezeichnet. Wiederum wurde der für die jeweilige Matrixdimension beste Blockfaktor benutzt.

Anhand des starken Einbruchs der Performance bei der Matrixdimension `N` $= 256$ lassen sich auch die übrigen Schwankungen erklären. Aus der Abbildung 6.8 (*JIK*-

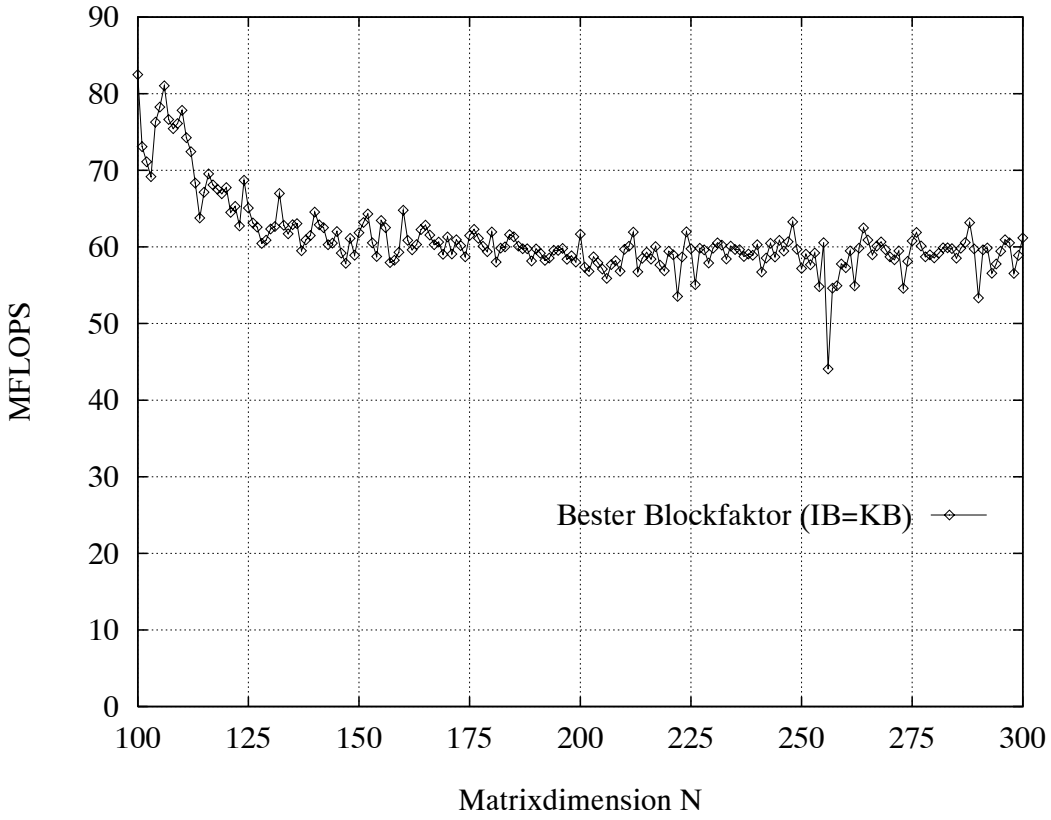


Abbildung 6.12: Performance-Messung Matrixmultiplikation: Einfluß des realen Cache-Speicherverhaltens

Form) geht hervor, daß auf das Feld `A(I,K)` zeilenweise zugegriffen wird. Bei der betrachteten Problemgröße liegen horizontal benachbarte Feldelemente jeweils 255 Maschinenworte (64 Bit) auseinander, weil in FORTRAN Matrizen spaltenweise abgespeichert werden. Da der interne Cache-Speicher maximal 1024 Maschinenworte aufnehmen kann, wird in jeder Zeile des quadratischen Blockes in Abbildung 6.8 das erste und das fünfte Element von `A(I,K)` auf denselben Cache-Speicherplatz abgebildet. Dasselbe gilt für das zweite und sechste Element der Zeile und ebenso für alle anderen Zeilen eines Blockes. Wird der Block erneut durchlaufen (in der `J`-Schleife), so ist keines der vorher referenzierten Elemente von `A(I,K)` mehr im Cache-Speicher und weder die örtliche noch die zeitliche Wiederverwendung führt zu einer ausnutzbaren Lokalität. Alle Elemente des Blockes müssen also aus dem externen Speicher geladen werden. Wandert der Block mit der Schrittweite des Blockfaktors nach rechts oder später nach unten, so wiederholt sich der beschriebene Effekt der Auslöschung. Das bedeutet, daß trotz Blockung alle Elemente von `A(I,K)` aus dem externen Speichersystem geladen werden müssen und die Performance dementsprechend absinkt. Diese Form der Auslöschung wird Selbstauslöschung genannt und hat einen starken Einfluß auf die ausnutzbare Datenlokalität. Für andere Matrixdimensionen tritt dieser Effekt unterschiedlich stark auf, was die Schwankungen der Performance-Kurve erklärt.

Der Blockfaktor, ab dem die Selbstauslöschung beginnt, wird in der Arbeit von Lam, Rothberg und Wolf [23] als optimaler Blockfaktor benutzt. In dem oben beschriebenen Fall hätte demnach der günstigste Blockfaktor die Größe 4, da es bereits bei einem Blockfaktor von 5 zur Auslöschung kommt. Messungen auf der DEC Alpha-Workstation Modell 300 ergaben für diesen Wert die geringe Performance von nur 10 MFLOPS, während der in Abbildung 6.12 angegebene Wert von 44 MFLOPS mit einer Blockgröße von 64 erzielt wurde. Desweiteren konnte auch die bei Lam, Rothberg und Wolf gemessene hohe Varianz der Performance bei variablen Blockfaktoren f"r eine feste Matrixdimension nicht bestätigt werden. Aus Abbildung 6.13 geht hervor, daß die Performance bei variablen Blockfaktoren zwischen 40 und 80 für die Matrixdimensionen `N` = 293 und `N` = 295 nahezu konstant sind. In [23] wurde bei der Matrixdimension `N` = 295 ein starke Häufung der Cache-Speicherlesefehler festgestellt, sobald der Blockfaktor größer als 24 gewählt wurde.

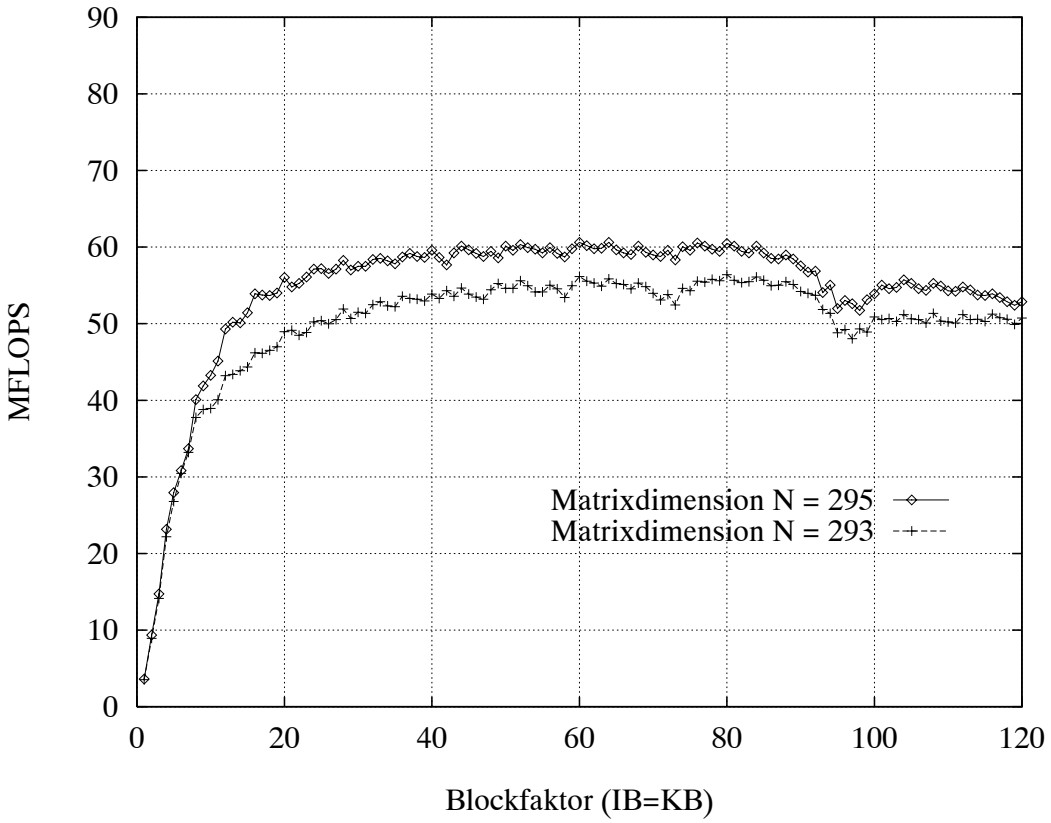


Abbildung 6.13: Performance-Messung Matrixmultiplikation: Unterschiedliche Blockfaktoren, konstante Matrixdimension N

Der in der zitierten Arbeit benutzte Rechner (DEC Station 3100) verfügt über keinen zusätzlichen externen Cache-Speicher, was die unterschiedlichen Beobachtungen erklären könnte. Da die Kapazität des externen Cache-Speichers bei beiden untersuchten Alpha-Workstations groß genug ist, alle in einem Block referenzierten Feldelemente gleichzeitig zu speichern, wird das Datum des Feldes `A(I,K)` bei einem Lesefehler im internen Cache-Speicher mit sehr großer Wahrscheinlichkeit nicht aus

dem Hauptspeicher, sondern aus dem externen Cache-Speicher geladen. Der Unterschied in der Zugriffszeit bei Lesefehler und Lesetreffer ist demnach vermutlich geringer als bei dem in [23] analysierten Rechner[4].

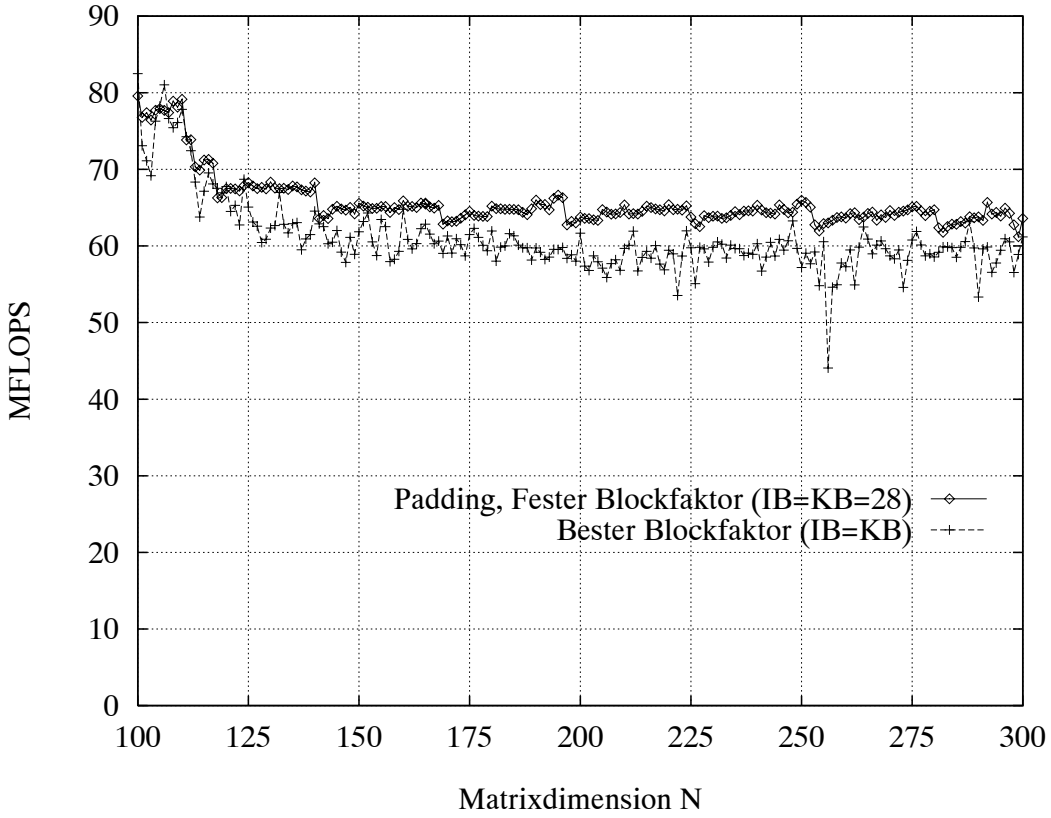


Abbildung 6.14: Performance-Messung Matrixmultiplikation: Konstante PerformanceWerte durch *padding*

Die im vorigen Abschnitt erwähnte Wahl ungleicher Blockfaktoren zur besseren Steuerung des Zugriffsmusters hatte keine nennenswerte Steigerung der Performance zur Folge (maximal 2 MFLOPS). Dagegen konnte durch die Verwendung der Compiler-Optimierungsoption **-O4** gegenüber den Optionen **-O3** und **-O2** für alle Problemgrößen eine in der Regel bessere oder zumindest gleich hohe Performance erzielt werden. Die Übersetzung mit **-O3** führte gegenüber der Optimierungsoption **-O2** zu keinem nachweisbarem Performance-Gewinn. Dies zeigt, daß die bei der Optimierungsoption **-O4** automatisch durchgeführte Lageänderung der Felder im Hauptspeicher eine geringere gegenseitige Auslöschung der Felder im internen Cache-Speicher zur Folge hat. Die Auslöschung durch andere Elemente desselben Feldes kann jedoch nicht vermindert werden. Dies wird erst durch das sogenannte *padding* der einzelnen Datenfelder ermöglicht, was im folgenden beschrieben wird.

Aus der Abbildung 6.12 geht hervor, daß die gemessenen Performance-Werte nicht kontinuierlich mit steigender Matrixdimension abfallen, sondern daß nach einem

[4]Konkrete Zugriffszeiten sind in der zitierten Arbeit nicht angegeben.

lokalen Minimum wieder höhere Werte gemessen werden. Diese Tatsache kann man ausnutzen, indem man die Felder bestimmt, für die eine hohe Performance gemessen wurde und diese Felddimensionen auch für die Berechnung kleinerer Felder benutzt. Diese Technik (*padding*) wurde bereits in Kapitel 4.2.5 beschrieben. An dieser Stelle kommt nun noch hinzu, daß für alle Felder der Blockfaktor benutzt werden muß, mit dem bei der ursprünglichen Messung des großen Feldes die beste Performance erzielt wurde[5].

In Abbildung 6.14 sind die beiden Kurven, mit bestem Blockfaktor und variabler Feldgrenze, bzw. mit konstantem Blockfaktor und konstanter Feldgrenze gegenübergestellt. Mit dieser Technik werden Performance-Werte erzielt, die zum Teil sogar die Performance-Werte der DEC DGEMM-Routine übertreffen. Weiterhin wurden auch die Performance-Schwankungen eliminiert, die in Abbildung 6.4 zu erkennen sind.

[5]Für alle Felder muß *padding* angewendet werden.

Kapitel 7

Zusammenfassung und Ausblick

Für die effektive Nutzung der hohen theoretischen Peak-Performance moderner superskalarer und superpipelined RISC-Prozessoren spielt die gemeinsame Betrachtung von Prozessor, Speicher und Compiler eine Schlüsselrolle. Zu dem gleichen Ergebnis kommt auch die vorliegende Arbeit, die in diesem Kontext, stellvertretend für einige neuartige RISC-Prozessor-Architekturen, den Alpha-Chip AA-21064 als Repräsentanten untersucht hat.

Mit der Alpha-Architektur hat Digital Equipment als erster Hersteller den Schritt zur vollständigen Unterstützung des 64-Bit-Datenformats vollzogen. Der damit geschaffene Adreßraum von 2^{64} Byte ist für eine absehbare Zeit ausreichend und beseitigt insbesondere die Einschränkung des Adreßraums von heutigen 32-Bit-Architekturen. Darüberhinaus führt die tiefe und zudem parallel angelegte Pipeline-Struktur, deren Kontrolle weitgehend von der Hardware vorgenommen wird, in einem hohen Maß zu internem Parallelismus.

Die Hardware-Kontrolle der Pipeline entbindet den Compiler von der Aufgabe, die einzelnen Stufen durch Erzeugung von entsprechenden Assembler-Befehlen weiterzuschalten, wie es etwa beim Prozessor i860 der Firma Intel der Fall ist. Dies ermöglicht eine starke Vereinfachung des Instruktions-Scheduling und damit eine effektivere Ausnutzung der parallelen Pipeline-Struktur.

Die Untersuchung des DEC FORTRAN-Compilers in Kapitel 4 ergibt, daß die wesentlichen Optimierungen auf Instruktionsebene vorgenommen werden, wie etwa *strength-reduction, dead code elimination, constant propagation* etc. Darüberhinaus fällt den Optimierungen, die eine effektive Ausnutzung des Parallelismus auf Instruktionsebene ermöglichen, ein besonderes Gewicht zu. Erst mit den Methoden des Software-Pipelining und der Vergrößerung von Basisblöcken zur besseren Durchführung des Instruktions-Scheduling können die Leistungspotentiale der parallelen Funktionseinheiten mit ihrer Pipeline-Struktur ausgenutzt werden. Für Schleifen mit kurzen Basisblöcken, wie sie beispielsweise bei der Matrixmultiplikation vor-

kommen, reicht die vom DEC FORTRAN-Compiler vorgenommene, konstant vierfache Entfaltung zur effektiven Belegung der Fließpunkt-Pipeline nicht aus. An dieser Stelle können massive Performance-Gewinne erst manuell durch den Programmierer erreicht werden, indem Schleifenkonstrukte entfaltet werden. Eine entsprechende Vorgehensweise wurde in dieser Arbeit entwickelt.

Der schon in der Einleitung erwähnte starke Einfluß der Speicherlatenzzeit auf die Performance hat sich im Verlauf der Arbeit bestätigt. Durch eine zu hohe Wartezeit bei dem Zugriff auf Daten aus dem Hauptspeicher wird der Parallelismus auf Instruktionsebene aufgehoben, und die erreichbaren Leistungen sind nicht größer als im skalaren Fall. Bei allen analysierten Programmen, deren Problemgröße eine häufige Referenzierung des externen Speichersystems notwendig machte, war die erreichbare Leistung durch die Bandbreite der Datenpfade begrenzt. Als logische Folge übertraf die niedriger getaktete Alpha-Workstation Modell 400 (133 MHz) die andere zur Verfügung stehende Workstation Modell 300 (150 MHz) hinsichtlich der Performance aufgrund des breiteren Datenpfades (256 Bit statt 64 Bit) und des größeren Cache-Speichers (512 KByte statt 256 KByte).

Eine Lösung zur Vermeidung dieses Datenengpasses besteht in der Reduktion der notwendigen Hauptspeicherzugriffe durch die Ausnutzung der Datenlokalität in den Cache-Speichern. Dies kann durch die Anwendung von Programmtransformationen auf Hochsprachenebene (Schleifenentfaltung und Schleifenblockung) erreicht werden. Dabei werden Ergebnisse erreicht, die denen der von Hand optimierten Assembler-Routinen aus der mathematischen Bibliothek des Herstellers sehr nahe kommen. Um die Beibehaltung der Programmsemantik zu garantieren, ist eine Datenabhängigkeitsanalyse erforderlich. Darüberhinaus liefert diese Analyse auch die notwendigen Informationen, welche Transformationen anzuwenden sind. Da beispielsweise das Vertauschen von Schleifen eine bessere Ausnutzung der Lokalität von Daten eines Feldes ermöglicht, gleichzeitig jedoch die Anzahl der notwendigen Hauptspeicherzugriffe für eine andere Feldreferenzierung erhöhen kann, ist eine Strategie zur Anwendung dieser Transformationen notwendig. Durch die dargestellte Vorgehensweise, Bestimmung der Datenabhängigkeiten und Wiederverwendung, Wahl der effektiven Block- und Entfaltungsfaktoren und Durchführung der in Kapitel 5 beschriebenen Schleifentransformationen, kann der Programmierer ohne intensives Probieren aller Möglichkeiten der Programmtransformationen in Programmen mit hoher Datenlokalität erhebliche Performance-Steigerungen erreichen.

Eine weitere Optimierungsmöglichkeit, die der DEC FORTRAN-Compiler nicht realisiert, ist das Einfügen von NOP-Befehlen zur effektiveren Instruktionsinitiierung. Durch die detaillierte Beschreibung des DEC Alpha-Chip in Kapitel 3 soll es dem Leser jedoch möglich sein, bei Bedarf selbst ein Instruktions-*Padding* im Assembler-Code vorzunehmen.

Für die Wahl des optimalen Blockfaktors ist die Auslöschung der Cache-Speicherdaten infolge von Feldreferenzen, deren Indizes die inneren Schleifenvariablen enthalten, bestimmend. Je kleiner der Blockfaktor ist, desto niedriger ist die ausgenutzte Da-

tenlokalität. Die starke Varianz der Performance in Abhängigkeit von der Matrixdimension, wie sie in [23] gemessen worden ist, konnte durch eine Vergrößerung der referenzierten Felder (*padding*) und die Benutzung eines festen Blockfaktors behoben werden.

Für die nächsten 25 Jahre erwartet der Hersteller Digital Equipment eine Leistungssteigerung der Alpha-Prozessoren um den Faktor 1000. Dieser Wert ergibt sich aus der Multiplikation der drei Faktoren Taktfrequenz, Instruktionsparallelität und Prozessoranzahl, die jeweils um den Faktor zehn gesteigert werden sollen [31]. Die Ergebnisse dieser Arbeit legen es nahe, vor allem auch die Zugriffszeit auf das Hauptspeichersystem durch den Einsatz von schnellen und größeren Cache-Speichern sowie durch die Implementation einer größeren Anzahl von Registern zu verkürzen. Darüberhinaus sind weitere Anstrengungen auf dem Gebiet der optimierenden Compiler notwendig, die insbesondere eine bessere Ausnutzung der Datenlokalität zum Ziel haben müssen.

Anhang A

Befehlssatz der DEC Alpha-AXP-Architektur

Fließpunkt Lade/Schreib		Fließpunkt Verzweigung	
LDF	Load F format (VAX single)	**FBEQ**	FP Branch if FPreg = 0
LDG	Load G format (VAX double)	**FBNE**	FP Branch if FPreg $\neq$ 0
LDS	Load S format (IEEE single)	**FBLT**	FP Branch if FPreg $<$ 0
LDT	Load T format (IEEE double)	**FBLE**	FP Branch if FPreg $\leq$ 0
STF	Store F format (VAX single)	**FBGT**	FP Branch if FPreg $>$ 0
STG	Store G format (VAX double)	**FBGE**	FP Branch if FPreg $\geq$ 0
STS	Store S format (IEEE single)		
STT	Store T format (IEEE double)		
Fließpunkt Operation			
CPYS	Copy sign	**CMPTUN**	Compare T format unordered
CPYSN	Copy sign, negate	**CVTGQ**	Convert G format to quadword
CPSYE	Copy sign and exponent	**CVTQF**	Convert quadword to F format
CVTQL	Convert quadword to longword	**CVTQG**	Convert quadword to G format
CVTLQ	Convert longword to quadword	**CVTDG**	Convert D to G format
FCMOVEQ	FP conditional move if FPreg = 0	**CVTGD**	Convert G to D format
FCMOVNE	FP conditional move if FPreg $\neq$ 0	**CVTGF**	Convert G to F format
FCMOVLT	FP conditional move if FPreg $<$ 0	**CVTTQ**	Convert T format to quadword
FCMOVLE	FP conditional move if FPreg $\leq$ 0	**CVTQS**	Convert quadword to S format
FCMOVGT	FP conditional move if FPreg $>$ 0	**CVTQT**	Convert quadword to T format
FCMOVGE	FP conditional move if FPreg $\geq$ 0	**CVTTS**	Convert T to S format
MF_FPCR	Move from FP control register	**CVTST**	Convert S to T format
MT_FPCR	Move to FP control register	**DIVF**	Divide F format (VAX single)
ADDF	Add F format (VAX single)	**DIVG**	Divide G format (VAX double)
ADDG	Add G format (VAX double)	**DIVS**	Divide S format (IEEE single)
ADDS	Add S format (IEEE single)	**DIVT**	Divide T format (IEEE double)
ADDT	Add T format (IEEE double)	**MULF**	Multiply F format (VAX single)
CMPGEQ	Compare G format = (VAX double)	**MULG**	Multiply G format (VAX double)
CMPGLT	Compare G format $<$ (VAX double)	**MULS**	Multiply S format (IEEE single)
CMPGLE	Compare G format $\leq$ (VAX double)	**MULT**	Multiply T format (IEEE double)
CMPTEQ	Compare T format = (IEEE double)	**SUBF**	Subtract F format (VAX single)
CMPTLT	Compare T format $<$ (IEEE double)	**SUBG**	Subtract G format (VAX double)
CMPTLE	Compare T format $\leq$ (IEEE double)	**SUBS**	Subtract S format (IEEE single)
		SUBT	Subtract T format (IEEE double)

Tabelle A1: Fließpunktinstruktionen

Integer Berechnung, Bedingte Verschiebung		Integer Lade/Speicher, Byte Manipulation	
LDA	Load address	**ADDL**	Add longword
LDAH	Load address high	**S4ADDL**	Add longword, scale by 4
LDL	Load sign-extended longword	**S8ADDL**	Add longword, scale by 8
LDQ	Load quadword	**ADDQ**	Add quadword
LDQ_U	Load unaligned quadword	**S4ADDQ**	Add quadword, scale by 4
LDL_L	Load sign-extended longword locked	**S8ADDQ**	Add quadword, scale by 8
LDQ_L	Load quadword locked	**CMPEQ**	Compare signed quadword =
STL_C	Store longword, conditional	**CMPLT**	Compare signed quadword <
STQ_C	Store quadword, conditional	**CMPLE**	Compare signed quadword ≤
STL	Store longword	**CMPULT**	Compare unsigned longword <
STQ	Store quadword	**CMPULE**	Compare unsigned longword ≤
STQ_U	Store unaligned quadword	**MULL**	Multiply longword
EXTBL	Extract byte low	**MULQ**	Multiply quadword
EXTWL	Extract word low	**UMULH**	Multiply quadword high, unsigned
EXTLL	Extract longword low	**SUBL**	Subtract longword
EXTQL	Extract quadword low	**S4SUBL**	Subtract longword, scale by 4
EXTWH	Extract word high	**S8SUBL**	Subtract longword, scale by 8
EXTLH	Extract longword high	**SUBQ**	Subtract quadword
EXTQH	Extract quadword high	**S4SUBQ**	Subtract quadword, scale by 4
INSBL	Insert byte low	**S8SUBQ**	Subtract quadword, scale by 8
INSWL	Insert word low	**AND**	AND logical
INSLL	Insert longword low	**BIS**	OR logical
INSQL	Insert quadword low	**XOR**	XOR logical
INSWH	Insert word high	**BIC**	AND-NOT logical
INSLH	Insert longword high	**ORNOT**	OR-NOT logical
INSQH	Insert quadword high	**EQV**	XOR-NOT logical
MSKBL	Mask byte low	**SLL**	Shift left, logical
MSKWL	Mask word low	**SRL**	Shift right, logical
MSKLL	Mask longword low	**SRA**	Shift right, arithmetic
MSKQL	Mask quadword low	**CMOVEQ**	Conditional move if reg = 0
MSKWH	Mask word high	**CMOVNE**	Conditional move if reg ≠ 0
MSKLH	Mask longword high	**CMOVLT**	Conditional move if reg < 0
MSKQH	Mask quadword high	**CMOVLE**	Conditional move if reg ≤ 0
		CMOVGT	Conditional move if reg > 0
		CMOVGE	Conditional move if reg ≥ 0
		CMOVLBC	Conditional move if reg low bit clear
		CMPVLBS	Conditional move if reg low bit set
		CMPBGE	Compare bytes, unsigned
		ZAP	Clear selected bytes
		ZAPNOT	Clear inselected bytes

Integer Verzweigung			
BEQ	Branch if reg = 0	**BLBS**	Branch if low bit set
BNE	Branch if reg ≠ 0	**BR**	Branch
BLT	Branch if reg < 0	**BSR**	Branch to subroutine
BLE	Branch if reg ≤ 0	**JMP**	Jump
BGT	Branch if reg > 0	**JSR**	Jump to subroutine
BGE	Branch if reg ≥ 0	**RET**	Return from subroutine
BLBC	Branch if low bit clear	**JSR_COR.**	Jump to subroutine, return

Tabelle A2: Integer-Instruktionen

System			
CALL_PAL	Call priveleged architecture library	**RC**	Read and Clear
TRAPB	Trap barrier (precise exception)	**RS**	Read and set
FETCH	Prefetch (cache) date hint	**PALRES0**	PALcode reserved opcode 0
FETCH_M	Prefetch (cache) data, modify hint	**PALRES1**	PALcode reserved opcode 1
MB	Memory barrier (serialize)	**PALRES2**	PALcode reserved opcode 2
WMB	Memory barrier (serialize) write	**PALRES3**	PALcode reserved opcode 3
RPCC	Read process cycle counter	**PALRES4**	PALcode reserved opcode 4

Tabelle A3: Systeminstruktionen

Anhang B

Programm-Code für Matrixmultiplikation (JIK-Form)

```
      SUBROUTINE JIK_20(A, B, C, N, KB, IB )
      INTEGER*4  N
      INTEGER*4  I, II, NI, ZI1, ZI2, RESTI, IBLOCK
      INTEGER*4  K, KK, NK, ZK1, ZK2, RESTK, KBLOCK
      REAL*8     A(N,N), B(N,N), C(N,N)
      REAL*8     C1,C2,C3,C4,C5,C6,C7,C8,C9
      REAL*8     C10,C11,C12,C13,C14,C15,C16
      REAL*8     C17,C18,C19,C20
      REAL*8     CI1,CI2,CI3,CI4,CI5
      REAL*8     CII1,CII2,CII3,CII4,CII5,CII6
      REAL*8     CII7,CII8,CII9,CII10,CII11,CII12
      REAL*8     CIII1, CIII2, CIII3
      REAL*8     CIV1

      NJ    = N
      NK    = N
      NI    = N
      RESTI = MOD (NI-1, IBLOCK) + 1
      ZI1   = 1
      ZI2   = RESTI
c------Kontrollschleife der geblockten I-Schleife---------------------
      DO II = 1, NI, IBLOCK
         RESTK = MOD (NK-1, IBLOCK) + 1
         ZK1   = 1
         ZK2   = RESTK
c------Kontrollschleife der geblockten K-Schleife---------------------
         DO KK = 1, NK, KBLOCK
            DO J = 1, NJ-4, 5
               DO I = ZI1, ZI2-3, 4
c------------------scalar replacement fuer C(I,J)---------------------
                  C1 =  C(I  ,J)
```

```
                    C2 =  C(I+1,J)
                    C3 =  C(I+2,J)
                    C4 =  C(I+3,J)
                    C5 =  C(I  ,J+1)
                    C6 =  C(I+1,J+1)
                    C7 =  C(I+2,J+1)
                    C8 =  C(I+3,J+1)
                    C9 =  C(I  ,J+2)
                    C10 = C(I+1,J+2)
                    C11 = C(I+2,J+2)
                    C12 = C(I+3,J+2)
                    C13 = C(I  ,J+3)
                    C14 = C(I+1,J+3)
                    C15 = C(I+2,J+3)
                    C16 = C(I+3,J+3)
                    C17 = C(I  ,J+4)
                    C18 = C(I+1,J+4)
                    C19 = C(I+2,J+4)
                    C20 = C(I+3,J+4)
                    DO K = ZK1, ZK2
                       C1  = C1   + A(I  ,K) * B(K,J)
                       C2  = C2   + A(I+1,K) * B(K,J)
                       C3  = C3   + A(I+2,K) * B(K,J)
                       C4  = C4   + A(I+3,K) * B(K,J)
                       C5  = C5   + A(I  ,K) * B(K,J+1)
                       C6  = C6   + A(I+1,K) * B(K,J+1)
                       C7  = C7   + A(I+2,K) * B(K,J+1)
                       C8  = C8   + A(I+3,K) * B(K,J+1)
                       C9  = C9   + A(I  ,K) * B(K,J+2)
                       C10 = C10  + A(I+1,K) * B(K,J+2)
                       C11 = C11  + A(I+2,K) * B(K,J+2)
                       C12 = C12  + A(I+3,K) * B(K,J+2)
                       C13 = C13  + A(I  ,K) * B(K,J+3)
                       C14 = C14  + A(I+1,K) * B(K,J+3)
                       C15 = C15  + A(I+2,K) * B(K,J+3)
                       C16 = C16  + A(I+3,K) * B(K,J+3)
                       C17 = C17  + A(I  ,K) * B(K,J+4)
                       C18 = C18  + A(I+1,K) * B(K,J+4)
                       C19 = C19  + A(I+2,K) * B(K,J+4)
                       C20 = C20  + A(I+3,K) * B(K,J+4)
                    ENDDO
C-------------------Rueckspeicherung der Registerinhalte---------------
                    C(I  ,J)   = C1
                    C(I+1,J)   = C2
                    C(I+2,J)   = C3
                    C(I+3,J)   = C4
                    C(I  ,J+1) = C5
                    C(I+1,J+1) = C6
```

```
                  C(I+2,J+1) = C7
                  C(I+3,J+1) = C8
                  C(I  ,J+2) = C9
                  C(I+1,J+2) = C10
                  C(I+2,J+2) = C11
                  C(I+3,J+2) = C12
                  C(I  ,J+3) = C13
                  C(I+1,J+3) = C14
                  C(I+2,J+3) = C15
                  C(I+3,J+3) = C16
                  C(I  ,J+4) = C17
                  C(I+1,J+4) = C18
                  C(I+2,J+4) = C19
                  C(I+3,J+4) = C20
               ENDDO
C--------------Korrekturschleife fuer I-Schleife--------------------
C--------------Scalar replacement fuer C(I,J)-----------------------
               DO I = I, ZI2
                  CI1 = C(I,J)
                  CI2 = C(I,J+1)
                  CI3 = C(I,J+2)
                  CI4 = C(I,J+3)
                  CI5 = C(I,J+4)
C-----------------K-Schleife vierfach entfaltet----------------------
                  DO K = ZK1, ZK2-3, 4
                     CI1 = CI1 + A(I,K)   * B(K,J)
                     CI2 = CI2 + A(I,K)   * B(K,J+1)
                     CI3 = CI3 + A(I,K)   * B(K,J+2)
                     CI4 = CI4 + A(I,K)   * B(K,J+3)
                     CI5 = CI5 + A(I,K)   * B(K,J+4)
                     CI1 = CI1 + A(I,K+1) * B(K+1,J)
                     CI2 = CI2 + A(I,K+1) * B(K+1,J+1)
                     CI3 = CI3 + A(I,K+1) * B(K+1,J+2)
                     CI4 = CI4 + A(I,K+1) * B(K+1,J+3)
                     CI5 = CI5 + A(I,K+1) * B(K+1,J+4)
                     CI1 = CI1 + A(I,K+2) * B(K+2,J)
                     CI2 = CI2 + A(I,K+2) * B(K+2,J+1)
                     CI3 = CI3 + A(I,K+2) * B(K+2,J+2)
                     CI4 = CI4 + A(I,K+2) * B(K+2,J+3)
                     CI5 = CI5 + A(I,K+2) * B(K+2,J+4)
                     CI1 = CI1 + A(I,K+3) * B(K+3,J)
                     CI2 = CI2 + A(I,K+3) * B(K+3,J+1)
                     CI3 = CI3 + A(I,K+3) * B(K+3,J+2)
                     CI4 = CI4 + A(I,K+3) * B(K+3,J+3)
                     CI5 = CI5 + A(I,K+3) * B(K+3,J+4)
                  ENDDO
C-----------------Korrekturschleife fuer K-Schleife-----------------
                  DO K = K, ZK2
```

```
                  CI1 = CI1 + A(I,K) * B(K,J)
                  CI2 = CI2 + A(I,K) * B(K,J+1)
                  CI3 = CI3 + A(I,K) * B(K,J+2)
                  CI4 = CI4 + A(I,K) * B(K,J+3)
                  CI5 = CI5 + A(I,K) * B(K,J+4)
               ENDDO
C------------------Rueckspeicherung der Registerinhalte---------------
               C(I,J)   = CI1
               C(I,J+1) = CI2
               C(I,J+2) = CI3
               C(I,J+3) = CI4
               C(I,J+4) = CI5
            ENDDO
         ENDDO
C------------Korrekturschleife fuer J-Schleife-----------------------
         DO J = J, NJ
C---------------I-Schleife erneut entfaltet (12-fach)-----------------
            DO I = ZI1, ZI2-11, 12
C------------------Scalar repalcement fuer C(I,J)---------------------
               CII1 = C(I   ,J)
               CII2 = C(I+1 ,J)
               CII3 = C(I+2 ,J)
               CII4 = C(I+3 ,J)
               CII5 = C(I+4 ,J)
               CII6 = C(I+5 ,J)
               CII7 = C(I+6 ,J)
               CII8 = C(I+7 ,J)
               CII9 = C(I+8 ,J)
               CII10= C(I+9 ,J)
               CII11= C(I+10,J)
               CII12= C(I+11,J)
               DO K = ZK1, ZK2
                  CII1 = CII1 + A(I   ,K) * B(K,J)
                  CII2 = CII2 + A(I+1 ,K) * B(K,J)
                  CII3 = CII3 + A(I+2 ,K) * B(K,J)
                  CII4 = CII4 + A(I+3 ,K) * B(K,J)
                  CII5 = CII5 + A(I+4 ,K) * B(K,J)
                  CII6 = CII6 + A(I+5 ,K) * B(K,J)
                  CII7 = CII7 + A(I+6 ,K) * B(K,J)
                  CII8 = CII8 + A(I+7 ,K) * B(K,J)
                  CII9 = CII9 + A(I+8 ,K) * B(K,J)
                  CII10= CII10+ A(I+9 ,K) * B(K,J)
                  CII11= CII11+ A(I+10,K) * B(K,J)
                  CII12= CII12+ A(I+11,K) * B(K,J)
               ENDDO
C------------------Rueckspeicherung der Registerinhalte---------------
               C(I   ,J) = CII1
               C(I+1 ,J) = CII2
```

```
                  C(I+2 ,J) = CII3
                  C(I+3 ,J) = CII4
                  C(I+4 ,J) = CII5
                  C(I+5 ,J) = CII6
                  C(I+6 ,J) = CII7
                  C(I+7 ,J) = CII8
                  C(I+8 ,J) = CII9
                  C(I+9 ,J) = CII10
                  C(I+10,J) = CII11
                  C(I+11,J) = CII12
               ENDDO
C--------------Korrekturschleife fuer I-Schleife (dreifach entfaltet)
               DO I = I, ZI2 - 2, 3
C-----------------Scalar replacement fuer C(I,J)--------------------
                  CIII1 = C(I,J)
                  CIII2 = C(I+1,J)
                  CIII3 = C(I+2,J)
C--------------K-Schleife vierfach entfaltet-----------------------
                  DO K = ZK1, ZK2-3, 4
                     CIII1 = CIII1 + A(I,K)     * B(K,J)
                     CIII2 = CIII2 + A(I+1,K)   * B(K,J)
                     CIII3 = CIII3 + A(I+2,K)   * B(K,J)
                     CIII1 = CIII1 + A(I,K+1)   * B(K+1,J)
                     CIII2 = CIII2 + A(I+1,K+1) * B(K+1,J)
                     CIII3 = CIII3 + A(I+2,K+1) * B(K+1,J)
                     CIII1 = CIII1 + A(I,K+2)   * B(K+2,J)
                     CIII2 = CIII2 + A(I+1,K+2) * B(K+2,J)
                     CIII3 = CIII3 + A(I+2,K+2) * B(K+2,J)
                     CIII1 = CIII1 + A(I,K+3)   * B(K+3,J)
                     CIII2 = CIII2 + A(I+1,K+3) * B(K+3,J)
                     CIII3 = CIII3 + A(I+2,K+3) * B(K+3,J)
                  ENDDO
C-----------------Korrekturschleife fuer K-Schleife-----------------
                  DO K = K, ZK2
                     CIII1 = CIII1 + A(I,K)   * B(K,J)
                     CIII2 = CIII2 + A(I+1,K) * B(K,J)
                     CIII3 = CIII3 + A(I+2,K) * B(K,J)
                  ENDDO
C-----------------Rueckspeicherung der Registerinhalte-------------
                  C(I,J)  = CIII1
                  C(I+1,J)= CIII2
                  C(I+2,J)= CIII3
               ENDDO
C--------------Korrekturschleife fuer I-Schleife-------------------
               DO I=I, ZI2
C-----------------Scalar replacement fuer C(I,J)
                  CIV1 = C(I,J)
C-----------------K-Schleife vierfach entfaltet--------------------
```

```
                  DO K = ZK1, ZK2-3, 4
                     CIV1 = CIV1 + A(I,K)   * B(K,J)
                     CIV1 = CIV1 + A(I,K+1) * B(K+1,J)
                     CIV1 = CIV1 + A(I,K+2) * B(K+2,J)
                     CIV1 = CIV1 + A(I,K+3) * B(K+3,J)
                  ENDDO
C-----------------Korrekturschleife fuer K-Schleife-----------------
                  DO K = K, ZK2
                     CIV1 = CIV1 + A(I,K) * B(K,J)
                  ENDDO
C-----------------Rueckspeicherung fuer C(I,J)----------------------
                  C(I,J) = CIV1
               ENDDO
            ENDDO
            ZK1   = ZK1 + RESTK
            ZK2   = ZK2 + KBLOCK
            RESTK = KBLOCK
         ENDDO
         ZI1   = ZI1 + RESTI
         ZI2   = ZI2 + IBLOCK
         RESTI = IBLOCK
      ENDDO
      END
```

Literaturverzeichnis

[1] A.V. Aho, R. Sethi, J.D. Ullman. *Compilers, principles, techniques and tools.* Addison-Wesley Publishing Company, Reading, 1986.

[2] G.M. Amdahl, G. Blaauw, F.P. Brooks Jr. *Architecture of the IBM System/360.* IBM J. Res. Development,Vol. 8, No. 2, April 1967, pp. 87-101.

[3] U. Banerjee. *Dependence analysis for supercomputing.* Kluwer Academic Publishers, Dordrecht, 1988.

[4] D. Bhandarkar, D.W. Clark. *Performance from architecture: Comparing a RISC and a CISC with similar hardware organization.* ACM Sigplan Notices, Vol. 26, No. 4, April 1991, pp. 310-319 (Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, April 1991).

[5] D. Callahan, S. Carr, K. Kennedy. *Improving register allocation for subscripted variables.* ACM Sigplan Notices, Vol. 25, No. 6, June 1990, pp. 53-65.

[6] S. Carr. *Memory hierarchy management.* PhD thesis, Rice University, February 1993.

[7] *Alpha architecture handbook.* Digital Equipment Corporation, 1992.

[8] *DEC OSF/1, Assembly language programmer's guide.* Digital Equipment Corporation, 1992.

[9] *DEC Fortran for DEC OSF/1 AXP systems, user manual.* Digital Equipment Corporation, 1993.

[10] *DECchip 21064-AA microprocessor, hardware reference manual.* Digital Equipment Corporation, 1992.

[11] Deutsches Institut für Normung e.V. *DIN 44300, Informationsverarbeitung 1.* Beuth Verlag, Berlin, Köln, 1981.

[12] J.J. Dongarra, J. Du Croz, I. Duff, S. Hammerling. *A set of level 3 basic linear algebra subprograms.* ACM Trans. Math. Software, Vol. 16, pp. 1-17.

[13] D. Gannon, W. Jalby, K. Gallivan. *Strategies for cache and local memory management by global program transformation*. Journal of Parallel and Distributed Computing, No.5, 1988, pp. 587-616.

[14] J.F. Hake, W. Homberg. *The impact of memory organization on the performance of matrix multiplication*. Proceedings Supercomputing '90, New York, IEEE Computer Society Press, 1990, pp. 34-40.

[15] J.L. Hennessy, D.A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann Publishers Inc., 1990.

[16] J.L. Hennessy. *VLSI RISC processors*. VLSI Sytstems Design, Vol. 10, No. 10, 1985, pp. 22-32.

[17] F. Hoßfeld. *'Grand Challenges' - wie weit tragen die Antworten des Supercomputing?*. Informatik - Fachberichte Bd. 306, Springer-Verlag 1992, S. 241-251.

[18] F. Hoßfeld. *Wissenschaftliches Rechnen - Motor der Rechnerentwicklung*. FOKUS-Praxis, Information und Kommunikation, K. G. Saur Verlag, 1992.

[19] K. Hwang, F.A. Briggs. *Computer architecture and parallel processing*. McGraw-Hill, New York, 1988.

[20] N.P. Jouppi, D.W. Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*. SIGPLAN Notices,Vol. 24, Special Issue, May 1989, pp. 272-282. (Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, April 1989).

[21] N.P. Jouppi. *Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers*. In The 18th Annual Intl. Symposium on Computer Architecture, May 1991.

[22] G. Kane. *MIPS RISC architecture*. Prentice-Hall Inc., Englewood Cliffs, 1989.

[23] M. Lam, E.E. Rothberg, M.E. Wolf. *The cache performance and optimizations of blocked algorithms*. ACM Sigplan Notices, Vol. 26, No. 6, June 1991, pp. 63-74 (Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 1991).

[24] O. Lange, G. Stegemann. *Datenstrukturen und Speichertechniken*. Vieweg, Braunschweig, 1987.

[25] B. Liu, N. Strother. *Programming in VS FORTRAN on the IBM 3090 for maximum vector performance*. IEEE Computer, Vol. 21, No. 6, June 1988, pp. 65-76.

[26] E. McLellan. *The Alpha AXP architecture and 21064 processor*. IEEE Micro, June 1993, pp. 152-168.

[27] A. Mlynski-Wiese. *Leistungsuntersuchung des iPSC/860 RISC-Knoten-Prozessors: Architekturanalyse und Programmoptimierung*. Forschungszentrum Jülich, Jül-2766, Mai 1993.

[28] D.A. Patterson, D.R. Ditzel. *The case for the reduced instruction set computer*. Computer Architecture News, Vol. 8, No. 6, 1980, pp. 25-33.

[29] B.R. Rau, J.A. Fisher. *Instruction-level parallel processing: history, overview and perspective*. The Journal of Supercomputing, Vol. 7, No. 1/2, 1993, pp. 9-50.

[30] H. Samukawa. *Programming style on the IBM 3090 vector facility considering both performance and flexibility*. IBM Systems Journal, Vol. 27, No. 4, 1988, pp. 453-474.

[31] R.L. Sites. *Alpha AXP architecture*. CACM, Vol. 36, No. 2, Feb. 1993, pp. 33-44.

[32] D.P. Siewiorek, Ph.J. Koopman. *The architecture of supercomputers: Titan, a case study*. Academic Press Inc., San Diego, 1991.

[33] K. So, V. Zecca. *Program locality of vectorized applications running on the IBM 3090 with vector facility*. IBM Systems Journal, Vol. 27, No. 4, 1988, pp. 436-452.

[34] H.S. Stone. *High performance computer architecture*. Addison-Wesley Publishing Company, Reading, 1990.

[35] S.W.K. Tjiang, M.E. Wolf, M.S. Lam, K. Pieper, J.L. Hennessy. *Integrating scalar optimization and parallelization*. Computer Systems Laboratory, Stanford University, February 1992.

[36] M.E. Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford Unitversity, August 1992.

[37] M.J. Wolfe. *Optimizing supercompilers for supercomputers*. Pitman Publishing, London, 1989.

[38] H. Zima, B. Chapman. *Supercompilers for parallel and vector computers*. ACM Press: Addison-Wesley Publishing Company, Reading, 1990.

Danksagung

Die vorliegende Arbeit wurde als Diplomarbeit in Elektrotechnik am Lehrstuhl für Technische Informatik und Computerwissenschaften der Rheinisch-Westfälischen Technischen Hochschule (RWTH) Aachen erstellt.

Dem Lehrstuhlinhaber Herrn Prof. Dr. F. Hoßfeld danke ich für die Möglichkeit, die Arbeit am Zentralinstitut für Angewandte Mathematik (ZAM) des Forschungszentrums Jülich (KFA) anfertigen zu können. Mein besonderer Dank gilt Herrn Dr. W.E. Nagel und Herrn Dr. P. Weidner für die Betreuung der Arbeit und viele konstruktive Anregungen. Bei Herrn Dr. R. Vogelsang von der Firma CRAY Research bedanke ich mich für die zahlreichen Diskussionen, die mir einen guten Einblick in die internen Abläufe des Alpha-Chip ermöglichten.

Weiterhin möchte ich die gute und kameradschaftliche Atmosphäre unter den Studenten und Doktoranden am ZAM hervorheben, die zu einem guten Gelingen dieser Arbeit beigetragen hat. An dieser Stelle danke ich auch meiner Familie, die mich auf dem Weg durch mein Studium begleitet und unterstützt hat, dessen Abschluß diese Arbeit darstellt.