Analysen und Methoden optimierender Compiler zur Steigerung der Effizienz von Speicherzugriffen in eingebetteten Systemen

Björn Franke

Dortmund, 30. August 1999

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Beim Einsatz von digitalen Signalprozessoren (DSP) in der Signalverarbeitung werden hohe Leistungsanforderungen gestellt. Zum einen muß eine große Geschwindigkeit in der Programmausführung erreicht werden, um den Durchsatzanforderungen zu genügen. Zum anderen gibt es strenge Grenzen des Speicherplatz- und Stromverbrauchs. Die Limitierung des Speicherplatzes ergibt sich aus den hohen Kosten für das Programm-ROM eines DSP, und aus den meist knappen Ressourcen des RAMs für den Datenspeicher. Der Stromverbrauch spielt vorwiegend in batteriebetriebenen, portablen Systemen, die einen DSP beinhalten, eine Rolle.

Ein möglicher Weg, den Anforderungen an DSP-Applikationen zu begegnen, liegt in der Optimierung von DSP-Programmen durch einen Compiler. Mit dem Ziel der Beschleunigung der Ausführung kann dieser versuchen, einen ähnlich guten Code zu generieren wie ihn ein erfahrener Assembler-Programmierer schreiben könnte. Das gewünschte Ziel wird jedoch vom Compiler nicht immer erreicht, es kann auch zu einer Verlangsamung kommen. Wenn eine Steigerung der Ausführungsgeschwindigkeit durch Programmoptimierungen erzielt wird, steht diese häufig in engem Zusammenhang mit dem Speicherplatzbedarf des Programms und dem Strombedarf des Prozessors. Viele Optimierungen beschleunigen nicht nur die Ausführung, sondern verkleinern gleichzeitig den Code-Umfang. Mit der Beschleunigung der Software können u.U. die Anforderungen an die Hardware zurückgenommen werden, z.B. durch Verminderung des Systemtakts oder durch den Einsatz weniger leistungsfähiger Prozessoren. Damit können der Stromverbrauch gesenkt oder Kosten gespart werden.

In der Vergangenheit sind viele Optimierungstechniken erarbeitet und vorgestellt worden ([1],[15],[3]). Diese wirken in sehr unterschiedlicher Weise und zielen in verschiedene Richtungen. Sie vermindern beispielsweise die Anzahl der Operationen zur Durchführung von Berechnungen, reorganisieren die Anordnung von Instruktionen zur Erzielung eines günstigeren Gesamtverhaltens, oder nutzen Informationen über die Zielarchitektur zur Anpassung des Ressourcenbedarfs von Programmen an das Ressourcenangebot des Prozessors. Unter den vielen Möglichkeiten zu Programmoptimierungen gibt es die Klasse der Speicherzugriffsoptimierungen, die ihr Bemühen darauf richten, Speicherzugriffe effizienter zu gestalten. Geschehen kann das durch eine Verringerung der Anzahl von Speicherzugriffen oder durch eine bessere Ausnutzung einer evtl. vorhandenen Speicherhierarchie. Da bei vielen DSP der Speicher einen *bottleneck* bildet, kann der Ersatz von Speicherzugriffen durch Registerzugriffe zu erheblichen Performance-Steigerungen beitragen. Neben der Beschleunigung kann es zu Einsparungen beim Stromverbrauch kommen, denn Zugriffe auf externen Speicher über ein Bus-System verbrauchen viel Strom.

1.1 Ziele der Diplomarbeit

Im Rahmen der Recherche zu dieser Diplomarbeit wurden die Bereiche der Abhängigkeitsanalysen und Speicherzugriffsoptimierungen untersucht. Die beiden Bereiche können nicht unabhängig voneinander betrachtet werden, da Optimierungen spezielle Informationen benötigen, die von geeigneten Analysen bereitgestellt werden. Daher ist eines der Recherche-Ziele dieser Diplomarbeit herauszufinden, welche für DSP geeigneten Speicherzugriffsoptimierungen existieren, und durch welche Analysen sie ermöglicht werden. Dazu wird ein Schwerpunkt auf die Analyse und Optimierung von Array-Zugriffen in Schleifen gelegt. Schleifen bieten einen guten Ansatzpunkt für Optimierungsverfahren, da der größte Zeitanteil der Programmausführung in Schleifen verbraucht wird. Schon kleine Gewinne in einer Iteration summieren sich über den ganzen Iterationsbereich zu beträchtlichen Verbesserungen auf. In Schleifen wird bei typischen DSP-Anwendungen häufig auf Array-Elemente zugegriffen, die sich bei einfachen, konventionellen Verfahren der Analyse und Optimierung entziehen, so daß an dieser Stelle noch Potential für weitere Verbesserungen besteht.

Beispiel 1.1.1 *Anwendung von Array-Redundanzeliminationen*

Original:

```
for(i = 0; i < 1000; i++)
{
   if (cond)
   {
      x = a[i];
      ...
   }
   a[i+1] = y;
}
```

Optimiert:

```
t = a[0];
for(i = 0; i < 1000; i++)
{
   if (cond)
   {
      x = t;
      ...
   }
   t = y;
   a[i+1] = t;
}
```

Beispiel 1.1.1 zeigt eine Programmschleife mit Zugriffen auf Array-Elemente und eine optimierte Schleifenvariante. In der Original-Version werden in jeder

Iteration zwei Zugriffe auf das Array `a` durchgeführt, davon kann der lesende Zugriff `a[i]` entfallen. Wenn eine Hilfsvariable `t` eingeführt wird, die dafür sorgt, daß der geschriebene Wert von `a[i+1]` in die nächste Iteration transportiert wird, kann der Zugriff auf das Array durch einen Zugriff auf die Hilfsvariable ersetzt werden. Wenn es zusätzlich noch gelingt, die Variable `t` in einem Register unterzubringen, kann beim gegebenen Beispiel die Anzahl der Speicherzugriffe für das Array `a` von 2000 auf 1001 reduziert werden. Positve Auswirkungen der Optimierung bestehen in der Beschleunigung der Programmausführung und der Entlastung des Speicherbusses. Als ungünstig könnten sich der Bedarf eines weiteren Registers im Schleifenkörper und der leicht angestiegene Code-Umfang erweisen.

Zur Durchführung der beschriebenen Optimierung ist die Information notwendig, daß `a[i+1]` und `a[i]` in aufeinanderfolgenden Iterationen das gleiche Daten-Element bezeichnen und `a[i+1]`, ohne zuvor überschrieben worden zu sein, auch beim Gebrauch von `a[i]` eintrifft. Mit Datenflußanalysen, die für skalare Variable entwickelt wurden, kann diese Information nicht erlangt werden, da sie nicht mit der Situation umgehen können, daß ein Datenobjekt zwei verschiedene Bezeichnungen zur Laufzeit haben kann (hier `a[i]` und `a[i+1]`).

Nicht nur die Elimination redundanter Speicherzugriffe trägt zur Steigerung der Effizienz von Speicherzugriffen bei. Auch die Nutzung von Hardware-Ressourcen zur Unterstützung von Speicherzugriffen können diese effizienter werden lassen. Wenn das in DSP vorhandene On-Chip-RAM und die AGU verstärkt genutzt werden, kann, ohne die Anzahl der Speicherzugriffe zu verringern, eine Performance-Steigerung erreicht werden.

Es muß geklärt werden, welche Analysen zur Durchführung geeigneter Optimierungen erforderlich sind, und wie sie realisiert werden. Die Verbindungen zwischen Analysen und Optimierungen müssen aber auch dahingehend untersucht werden, wie die Informationsqualität der Analysen die Optimierungsgüte beeinflußt. Darüberhinaus soll geklärt werden, welche weiteren Optimierungen von den Analysen profitieren können und welche Vor- oder Nachteile sie bei DSP-Architekturen haben. Die Wechselwirkungen von verschiedenen Optimierungen untereinander und mit der Zielarchitektur sind wichtig für die Auswahl und den Einsatz der Optimierungen.

Abbildung 1.1 verdeutlich die Abhängigkeitsverhältnissse zwischen Programmen, deren Analysen und Optimierungen sowie der Zielarchitektur. Einem gegebenen Programm entnehmen die Analysen Informationen, z.B. über Verwendungen von Variablen oder gemeinsame Teilausdrücke. Zur Informationsentnahme müssen die Analysen aktiv das Programm untersuchen, und gewinnen daraus zusammengefaßte Aussagen über bestimmte Eigenschaften. Die Optimierungen benötigen die Informationen aus geeigneten Analysen. Häufig geht die Güte der Analyse in die Güte einer Optimierung ein oder schafft gar erst die Voraussetzungen zu deren Anwendung. Zur anderen Seite unterstützen Optimierungen bestimmte Eigenschaften der Zielarchitektur bei der Verbesserung des Programms, damit eine möglichst hohe Leistung erzielt werden kann. Die

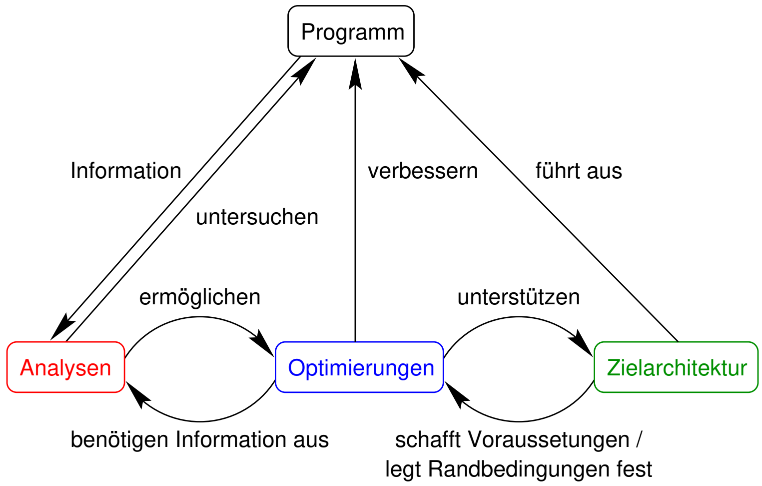


Abbildung 1.1: Abhängigkeiten zwischen Programmen, Analysen, Optimierungen und Zielarchitektur

Optimierungen müssen auf durch die Zielarchitektur geschaffene Voraussetzungen Rücksicht nehmen, denn nicht jede Optimierung führt automatisch und unabhängig von der Prozessorarchitektur zu den gewünschten Verbesserungen der Systemeigenschaften. Letztendlich kann das Programm vom Prozessor ausgeführt werden.

Konkretes Ziel dieser Diplomarbeit ist die Beantwortung folgender Fragen:

- Welche Datenabhängigkeitsanalysen sind bislang entwickelt worden und was leisten sie?
- Welche Anforderungen werden an Datenflußanalysen für Array-Elemente gestellt und wie können sie erfüllt werden?
- Welche Array-Datenflußanalysen stehen zur Verfügung und wodurch unterscheiden sie sich?
- Welche Datenabhängigkeitsanalysen sind zur Unterstützung von Speicherzugriffsoptimierungen für DSP geeignet?
- Welche Speicherzugriffsoptimierungen sind geeignet für digitale Signalprozessoren?
- Können die Besonderheiten der Prozessorarchitektur von DSP zur Unterstützung von Speicherzugriffsoptimierungen genutzt werden?
- Welche Vor- und Nachteile ergeben sich aus der Anwendung einer Speicherzugriffsoptimierung?
- Wie hängt die Ausprägung des Registersatzes eines DSP mit dem Erfolg bzw. der Anwendbarkeit einer Optimierung zusammen?

- Welche weiteren Optimierungen können von Array-Datenflußanalysen profitieren?

1.2 Überblick

Kapitel 2 dieser Diplomarbeit führt in die Grundlagen von DSP-Architekturen ein. Besondere Berücksichtigung erfahren deren Registersätze und Adressierungseinheiten sowie das On-Chip-RAM als Komponenten, die von Bedeutung für Speicherzugriffsoptimierungen sind. Kapitel 3 stellt die Grundlagen von Datenflußanalysen und Optimierungen für skalare Variable dar, auf die in den darauf folgenden Kapiteln zurückgegriffen wird. In Kapitel 4 werden verschiedene Verfahren für die Datenflußanalyse von Array-Elementen vorgestellt. Die Analysen verfolgen z.T. verschiedene Ansätze und haben unterschiedliche Komplexität. Die Qualität der Analyse-Ergebnisse variiert für ein eingeschränktes Datenflußproblem zwischen Näherungslösung und exakter Lösung. In Kapitel 5 werden die Grundversionen von Load/Store-Redundanzeliminationen gezeigt, die die Ergebnisse von Array-Datenflußanalysen nutzbringend verwerten. In Kapitel 6 werden Varianten der vorangegangen Optimierungen diskutiert, die den möglichen Einsatzbereich erweitern oder die Optimierungsqualität verbessern. Zudem wird eine Version der Optimierung *Register-Pipelining* erläutert, die mit der Hardware-Unterstützung der Adressierungseinheit und des On-Chip-RAM eines DSP weitere Effizienzsteigerungen bewirken kann. Weitere Optimierungen, die für die Verbesserung von DSP-Programmen nützlich sein können, kommen in Kapitel 7 hinzu. Diese Optimierungen werden entweder durch die vorgestellten Array-Datenflußanalysen gegenüber bekannten Standardversionen verbessert, sind aber selbst keine Speicherzugriffsoptimierungen, oder verwenden eigene, spezielle Analysen. In Kapitel 8 werden Versuche dokumentiert, die mit im Rahmen der Diplomarbeit implementierten Array-Datenflußanalysen und Load/Store-Redunanzoptimierungen durchgeführt wurden. Im letzten Kapitel werden schließlich die Ergebnisse dieser Diplomarbeit zusammengefaßt und es wird auf Ausblick auf weitere Aspekte des Themengebietes gegeben, die in dieser Diplomarbeit nicht behandelt werden. Im Anhang werden die Implementationen der Optimierungen dokumentiert, mit denen die Versuche aus Kapitel 8 durchgeführt wurden. Eine Literaturliste, die dazu genutzt werden kann, einen Einstieg in das Gebiet der Pointer-Analyse zu finden, ist zusätzlich im Anhang untergebracht.

Kapitel 2

DSP-Architekturen

Digitale Signalprozessoren (DSP) sind Prozessoren, die in ihren Architekturmerkmalen ihrem Haupteinsatzgebiet – der digitalen Signalverarbeitung – entgegenkommen. In vielen Bereichen der digitalen Signalverarbeitung kommt es darauf an, daß eine große Menge von Daten erfaßt und nahezu zeitgleich verarbeitet wird. Der tolerierbare Zeitverzug zwischen der Datenerfassung und der Ausgabe der verarbeiteten Daten ist oft sehr klein, so daß an DSP hohe Anforderungen bezüglich des Durchsatzes gestellt werden. Um diesen Anforderungen zu genügen, muß ein besonderer Augenmerk auf die Konzeption der Verarbeitungseinheiten wie auch des Speichersystems digitaler Signalprozessoren gelegt werden.

Zur Steigerung der Leistung der funktionalen Einheiten wird bei DSP oft auf Parallelverarbeitung, genauer auf instruktionsparallele Verarbeitung, zurückgegriffen. Nach [17] ist ein *Instruction-Level Parallel (ILP)* Prozessor ein paralleler Prozessor, dessen kleinste Berechnungseinheit, für die Scheduling- und Synchronisationsenscheidungen zu treffen sind, eine einzelne Operation ist. Dabei wird nicht unterschieden, ob entsprechende Entscheidungen zur Laufzeit eines Programms gefällt werden, oder schon während der Compilierung erfolgen. Unter diese Definition fallen VLIW-Prozessoren, da für sie Scheduling-Entscheidungen vom Compiler getroffen werden, ebenso wie superskalare Prozessoren, die z.B. durch dynamisches Instruction Scheduling erst zur Ausführung diese Entscheidungen treffen. Viele DSP besitzen VLIW-Architekturmerkmale (z.B. TI C60, siehe Abb. 2.2), so daß auch sie der Klasse der ILP-Prozessoren zuzuordnen sind.

In dieser Diplomarbeit stehen Belange der Datenverarbeitung – im Sinne von Berechnungen – durch DSP eher im Hintergrund, wichtiger in diesem Zusammenhang sind Besonderheiten digitaler Signalprozessoren im Bezug auf ihr Speichersystem. Dieses umfaßt Register, On-Chip- und Off-Chip-RAM sowie die sie verbindenden Bussysteme und zur Adressierung genutzte Adreßgenerierungseinheiten (Address Generation Unit, AGU). Die folgenden Abschnitte geben einen kurzen Überblick über die wichtigsten Unterschiede der Architekturmerkmale der Komponenten von Speichersystemen gängiger DSP gegenüber konventionellen General-purpose-Prozessoren.

2.1 Registersätze

Unter den DSP haben sich zwei verschiedene Klassen bzgl. ihrer Registersätze entwickelt. Zum einen sind *heterogene Registersätze* anzutreffen, bei denen die Register des Prozessors im Datenpfad verteilt und eng mit den funktionalen Einheiten verknüpft sind. Zum anderen gibt es DSP mit *homogenen Registersätzen*, die in Anlehnung an RISC-Prozessoren größere, allgemeine Registerbänke haben. Deren Register weisen i.a. keine zwingende Zuordnung zu funktionalen Einheiten – wie Addierern und Multiplizierern – auf und können frei verwendet werden.

2.1.1 Heterogene Registersätze

Registersätze, die mit funktionalen Einheiten eines DSP gekoppelt und evtl. sogar im Datenpfad verteilt sind, heißen *heterogene Registersätze*. Diese Art der Registersätze kommt i.a. den meist durchgeführten Operationen typischer DSP-Anwendungen entgegen. Eine Reihe von häufig benötigten Berechnungen werden effizient durch diese Prozessorarchitektur unterstützt. Für viele Anwendungen, z.B. Filter-Realisationen, werden fortlaufend mehr oder minder gleiche Berechnungen durchgeführt, die von einer allgemeineren Prozessorarchitektur nicht profitieren. So verfolgt die Anpassung der Architektur an die Anwendung nicht nur das Ziel der Effizienzsteigerung, sondern auch der Kostenminderung.

Abbildung 2.1 [2] zeigt den Prozessorkern der ADSP-21xx-Familie. Im unteren Bildteil befindet sich die arithmetische Einheit, die aus der ALU, einer MAC-Einheit und einem Shifter besteht. Jede der drei voneinander unabhängigen funktionalen Einheiten hat Eingabe- und Ausgabe-Register. In die Eingabe-Register werden Werte geschrieben, die als Operanden späterer arithmetischer Operationen dienen. Die Ausgabe-Register nehmen die von den Berchnungseinheiten gelieferten Resultate auf. Untereinander sind die Eingabe- und Ausgabe-Register mit einem Registerbus verbunden, um Werte aus den Ausgabe-Registern einer Einheit in Eingabe-Register der gleichen oder einer anderen Einheit zu transferieren. Zusätzlich sind die Eingabe- und Ausgabe-Register mit dem prozessor-internen Datenbus verbunden. Darüber können Werte geladen oder geschrieben werden, die von anderen internen oder externen Komponenten stammen oder dorthin gelangen sollen.

Werte, die in den Eingabe-Registern einer funktionalen Einheit gehalten werden, stehen den übrigen Einheiten nicht ohne weiteres zur Verfügung. Sie müssen erst durch Umkopieren in deren Eingabe-Register gebracht werden.

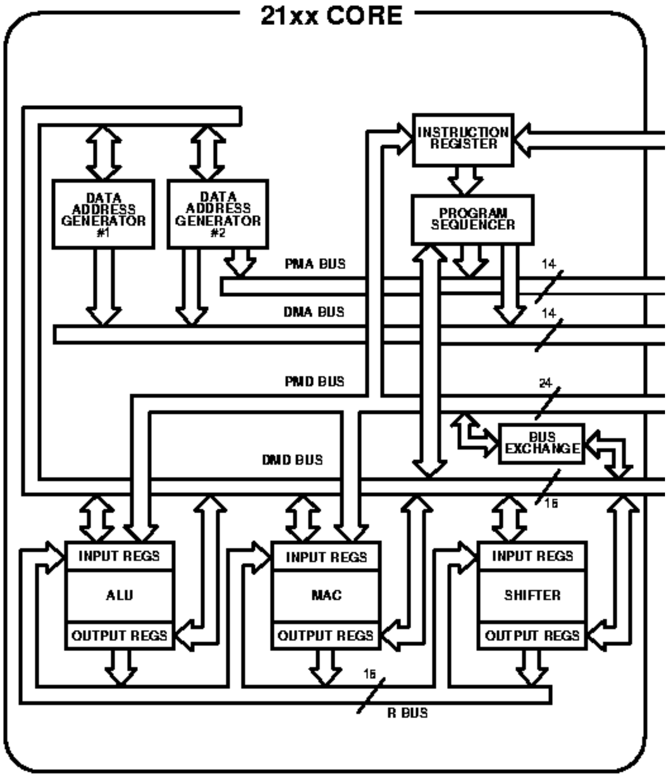


Abbildung 2.1: Prozessorkern der ADSP-21xx-Familie [2]

2.1.2 Homogene Registersätze

Mit immer komplexeren und immer vielfältigeren Aufgaben für DSP lassen sich verteilte Registersätze nicht mehr uneingeschränkt beibehalten. Zum einen ist die Spannweite möglicher Einsatzfelder von DSP sehr groß und die jeweils typischen Berechnungen variieren voneinander. Somit kann eine feste Auslegung der Architektur auf einen einzelnen Anwendungsfall unvorteilhaft sein. Zum anderen erschweren heterogene Registersätze Programmierern und/oder Compilern die Erzeugung von Code, der die vorhandenen Ressourcen optimal nutzt. Allgemein verwendbare Register, die nicht bestimmten funktionalen Einheiten zugeordnet sind, erleichtern den systematischen Umgang mit den Registern während der Code-Erzeugung. Dies hat zur Entwicklung einer Reihe von DSP mit *homogenen Registersätzen* geführt. Die meisten Register haben dabei keine spezifische Aufgaben- oder Ressourcen-Zuteilung, zudem ist häufig die Anzahl der verfügbaren Register höher als bei Architekturen mit heterogenen Registersätzen.

Abbildung 2.2 [20] zeigt ein Blockschaltbild der TI TMS320C62x-DSP-Familie. Grau unterlegt ist der Bereich des Prozessorkerns. In diesem befinden sich u.a. zwei Registerbänke A und B, die jeweils aus 16 32-Bit-Registern bestehen. Die beiden Registerbänke bilden zusammen mit acht funktionalen Einheiten zwei separate Datenpfade A und B. Jeweils vier der acht funktionalen Einheiten sind in einem Datenpfad eingebettet. Alle Register stehen für allgemeine Aufgaben zur Verfügung, sie können Daten, Adreßzeiger oder Bedingungen enthalten.

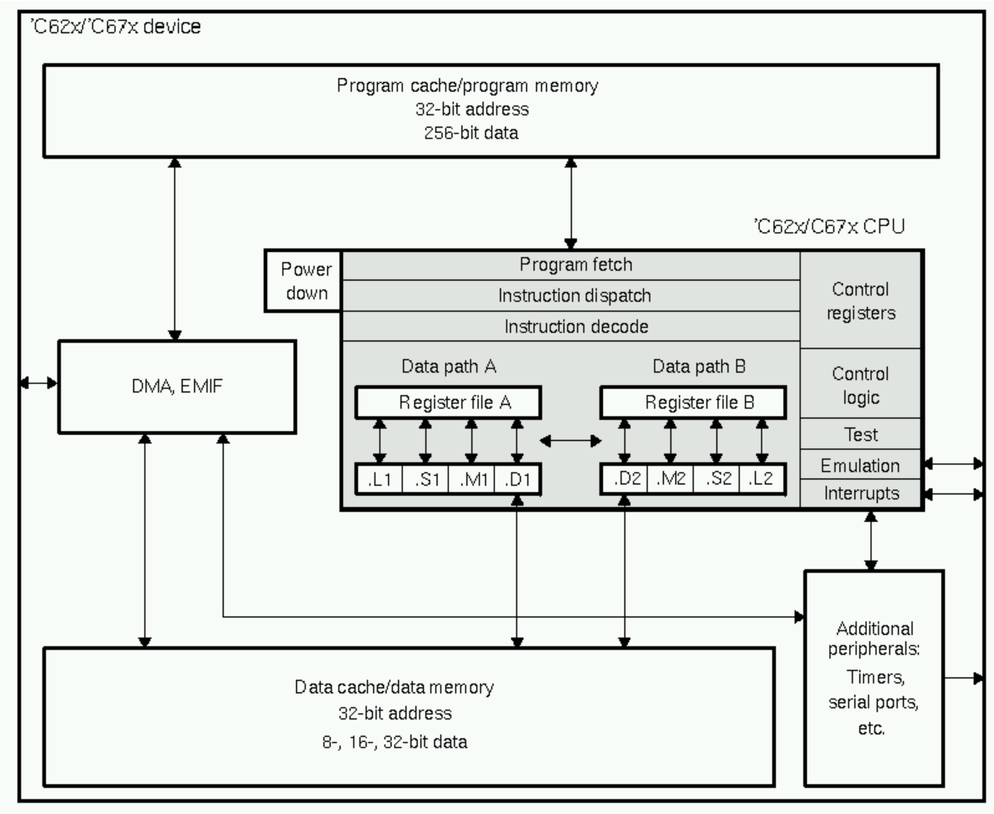


Abbildung 2.2: Prozessorarchitektur des TI C60 [20]

Die funktionalen Einheiten eines Datenpfades arbeiten i.a. mit der Registerbank ihres Datenpfades. Sie entnehmen ihr die Operanden und speichern die Resultate dorthin zurück. Zwischen den beiden Datenpfaden gibt es Querverbindungen. Die funktionalen Einheiten einer Seite können jeweils auf einen 32-Bit-Operanden aus der Registerbank der anderen Seite zugreifen.

Die Verbindung der Registerbänke zum Speicher erfolgt über vier je 32 Bit breite Busse. Pro Registerbank existiert ein Pfad zum Laden von Werten aus dem Speicher und ein Pfad zum Speichern.

2.2 Address Generation Unit

Typische DSP-Programme arbeiten bevorzugt mit Array-Datenstrukturen auf die mit regelmäßigen Mustern zugegriffen wird. Zur Entlastung der arithmetischen Einheiten des Prozessorkerns von Adreßberechnungsaufgaben enthalten viele DSP *Address Generation Units (AGU)*. Mit einer AGU kann parallel zur

Arbeit des Datenpfades die Adresse des nächsten Speicherzugriffs berechnet werden.

Die Verbindung einer AGU mit Post/Prä-Inkrement/Dekrement-Adressierungsarten erweist als besonders sinnvoll. Dadurch kann nach/vor einem Speicherzugriff die Adresse für den nächsten Zugriff um einen bestimmten Betrag erhöht/erniedrigt werden. Auf diese Weise lassen sich Array-Elemente effizient adressieren [4], denn der Speicherzugriff, die Adreßberechnung und u.U. Operationen der ALU finden gleichzeitig statt.

Abbildung[1] 2.3 zeigt den Aufbau einer einfachen AGU mit Adreß- und Modify-Registern. Anhand des Strukturbildes kann eine Post-Inkrement-Operation nachvollzogen werden. Aus dem Adreßregistersatz wird mit dem AR-Pointer ein einzelnes Register selektiert und dessen Inhalt wird als effektive Adresse auf den Adreßbus gelegt. Der Wert des selektierten Registers wird außerdem durch einen Addierer/Subtrahierer geführt, das Resultat wird zurück in das Adreßregister geschrieben. Der Summand bei dieser Operation kann eine konstante Eins, ein Direktoperand aus der Instruktion oder ein Wert aus der Modify-Registerbank sein. Damit sind unterschiedliche Schrittweiten der Zugriffe für verschieden große Array-Elemente zu realisieren. Für ILP-Prozessoren ergibt sich der Vorteil, daß Adreßarithmetik und Speicherzugriff zeitgleich mit den eigentlichen Berechnungen durchgeführt werden können. Die Zeiteinsparungen durch die exklusive Verwendung des Datenpfades für Programmdaten sind beträchtlich.

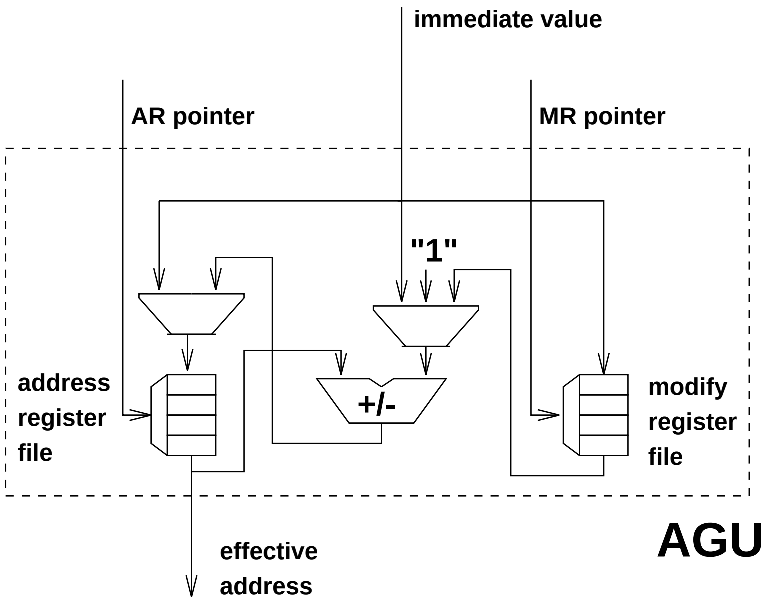


Abbildung 2.3: AGU mit Address- und Modify-Registern

Zum sequentiellen Adressieren eines `short int`-Arrays mit 16-Bit-Elementen

[1]Die Abbildungen 2.3 und 2.5 wurden von Rainer Leupers freundlicherweise zur Verfügung gestellt. Nochmals einen herzlichen Dank !

ab der Adresse 100 wird ein Adreßregister mit dem Wert 100 geladen. Ein Modify-Register wird mit dem Wert 2 geladen, da die 16-Bit-Werte einen Abstand von 2 untereinander haben. Nach dem ersten Zugriff mit einer Post-Inkrement-Operation wird der Wert im Adreßregister durch 102 ersetzt. Nachfolgende Adressen ergeben sich zu 104, 106, ... Soll neben neben dem `short int`-Array auch noch ein `long int`-Array mit 32-Bit-Elementen ab der Adresse 200 adressiert werden, so wird in einem anderen Adreßregister die Startadresse 200 gespeichert. In ein Modify-Register wird der Wert 4 geschrieben. Die Adressen einer Reihe von Zugriffen mit Post-Inkrement sind 200, 204, 208, ...

Bei der Realisation von FIR-Filtern [21] – einer häufigen und typischen DSP-Applikation – werden zwei Schieberegister benötigt. Diese werden üblicherweise mit Arrays implementiert, die als Ringpuffer organisiert sind. Dazu brauchen keine Inhalte verschoben und kopiert zu werden, sondern es reicht aus mit zwei Zeigern den Anfang und das Ende der Queue zu adressieren. Abbildung 2.4 zeigt die beiden Ringpuffer zusammen mit einem Multiplizierer-Akkumulator für einen FIR-Filter mit folgender Zeitfunktion:

$$y(t_N) = \sum_{k=0}^{N} \alpha_k x(t_{N-k}) \tag{2.1}$$

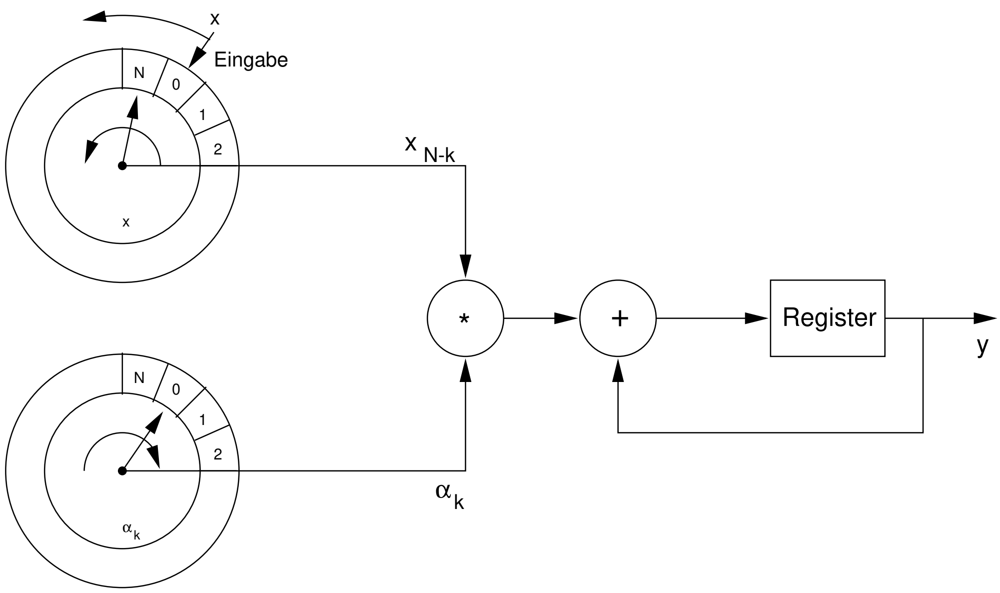


Abbildung 2.4: Prinzip der seriellen Berechnung eines FIR-Filters mit Ringpuffer [21]

Zur hardware-seitigen Unterstützung der für die Implementierung von Ringpuffern benötigten zirkularen Adressierung besitzen viele DSP Adressierungseinheiten mit einer Modulo-Logik. Diese bewirkt, daß bei einer Post/Prä-Inkrement/Dekrement-Operation der in das Adreßregister zurückgeschriebene Wert zuvor einer Modulo-Operation unterzogen wird. Abbildung 2.5 zeigt die Struktur einer solchen AGU. In einer Length-Register-Bank werden Werte gehalten, die der Modulo-Logik zugeführt werden. Die aus dem Addierer/Subtrahierer kommenden Werte werden diesem Modulo unterzogen.

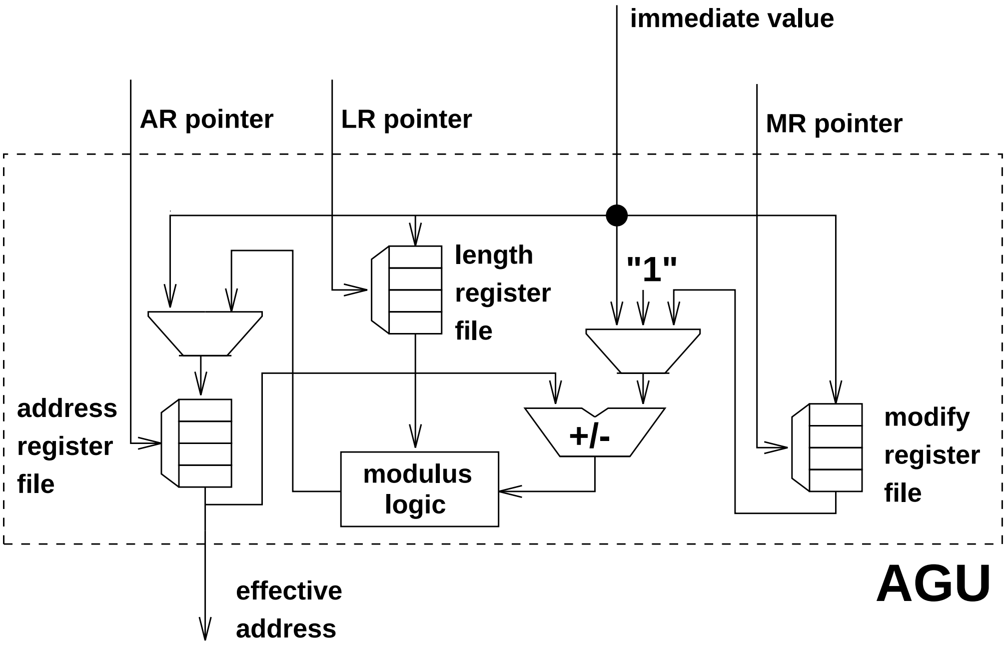


Abbildung 2.5: AGU mit Address-, Modify- und Length-Register

2.3 Speicherorganisation und On-Chip-RAM

DSP werden häufig als Harvard-Architekturen realisiert, die getrennte Daten- und Programmspeicher mit separaten Bussen vorsehen. Das ermöglicht den gleichzeitigen Transport von Instruktionen und Daten, was zu einer höheren Ausnutzung der Ressourcen und damit zur Steigerung der Effizienz beiträgt. Nach außen hin werden gelegentlich der Daten- und Programmbus zur Vereinfachung der äußeren Umgebung über einen externen Bus im Multiplex betrieben.

Viele DSP besitzen sowohl *internen Speicher (On-Chip-RAM)* als auch die Möglichkeit, über ein Bus-System externen Speicher zu adressieren. Der interne Speicher ist meist klein gegenüber dem gesamten Adreßraum, auf ihn kann ohne Verzögerungen zugegriffen werden. Der externe Speicher kann je nach Ausbau erheblich größer sein und ist abhängig von der Art des Speichers (SRAM,DRAM, ...) und der Leistungsfähigkeit des Bus-Systems langsamer. Interner und externer Speicher liegen meistens im gleichen Adreßraum, d.h. es müssen keine gesonderten Instruktionen zur Adressierung verwendet oder Segmentregister zuvor geladen werden.

Abbildung 2.6 zeigt das On-Chip-RAM bei einem TI TMS320C2x. Es besteht aus drei Blöcken, wovon einer (B0) als Daten- oder Programmspeicher konfiguriert werden kann. Die beiden übrigen (B1 und B2) sind immer Datenspeicher.

Das On-Chip-RAM kann ohne Wait-States angesprochen werden. Ein bis zu 64k großes externes RAM kann über das Bus-System adressiert werden. Über Multiplexer wird der Speicher vom Programmzähler oder vom Data-Page-Pointer adressiert (andere Quellen von Adressen sind möglich). Zum Datentransport sind die Blöcke B1 und B2 direkt mit dem Datenbus und B0 über einen Multiplexer mit dem Daten- und Programmbus verbunden.

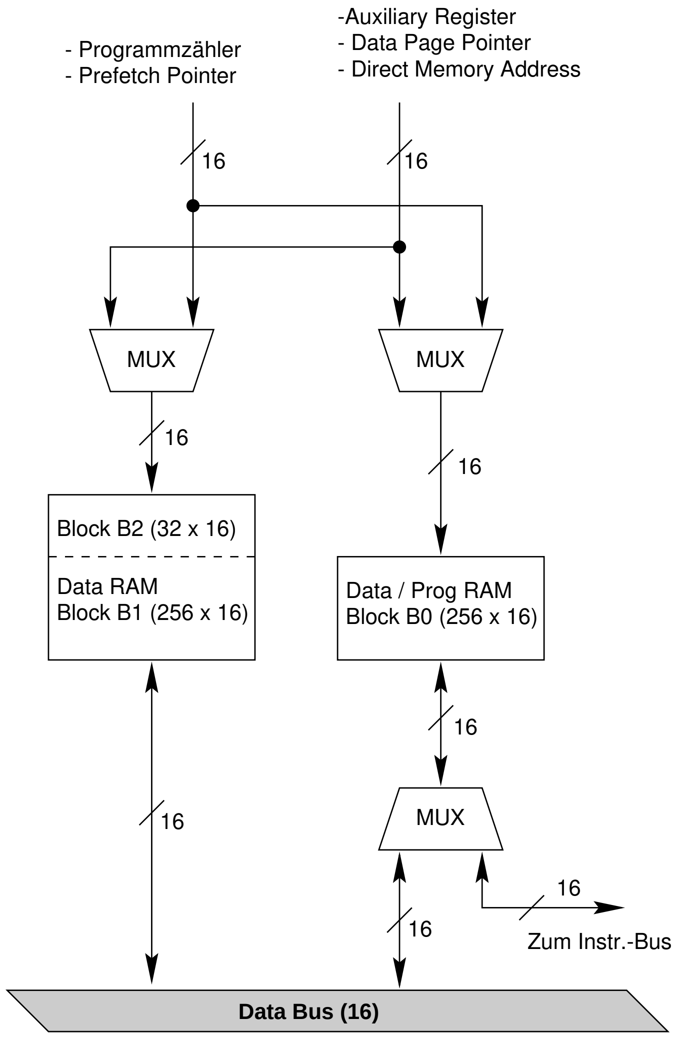


Abbildung 2.6: On-Chip-RAM beim TI TMS320C2x (nach [19])

Da die Größe des On-Chip-RAM mit 544 Speicherworten klein bemessen ist, sollten nur häufig frequentierte Daten dort abgelegt werden, um von der hohen Zugriffsgeschwindigkeit effizient Gebrauch zu machen. Verfahren zur Partitionierung von Daten auf On- und Off-Chip-RAM finden sich in [16] und [18].

Zum gleichzeitigen Zugriff auf zwei Operanden ermöglichen einige DSP (u.a. ADSP-21xx, TI C60) die *Dual Memory Execution*. Damit können zwei Speicherzugriffe parallel erfolgen, so daß z.B. zwei für eine Operation benötigte Operanden gleichzeitig aus dem Speicher in Register transferiert werden können.

2.4 Weitere Literatur

- Lapsley, P.
  DSP processor fundamentals : architectures and features
  IEEE Press, New York, 1997.

Kapitel 3

Datenabhängigkeit, Analysen, Optimierungen

Zur Durchführung von Programmoptimierungen sind einige Vorarbeiten notwendig, denn eine Optimierung muß wissen, an welchen Stellen sie Einsatz finden kann. Zudem darf sie nicht willkürlich das Programmverhalten ändern, denn eine Optimierung darf nicht die Korrektheit eines Programms riskieren. Die Ausführung der einzelnen Operationen unterliegt bestimmten Abhängigkeiten, seien es beispielsweise Abhängigkeiten von Verzweigungsbedingungen oder Abhängigkeiten von zuvor berechneten Werten, die für folgende Berechnungen benötigt werden. Auf diese Abhängigkeiten müssen Optimierungen Rücksicht nehmen. Die Ermittlung der relevanten Abhängigkeiten erfolgt durch Abhängigkeitsanalysen. Der Schwerpunkt liegt dabei auf *Datenabhängigkeiten*, die sich durch den wiederholten Zugriff verschiedener Instruktionen auf gemeinsame Daten ergeben.

3.1 Grundlegende Begriffe

Die Darstellung eines Programms durch seinen Quelltext oder durch eine Zwischendarstellung (IR, *intermediate representation*), die einer konventionellen Maschinensprache ähnlich ist, ist nicht für alle Analyseaufgaben eines Compilers geeignet. Wenn beispielsweise untersucht werden soll, ob ein bestimmtes Statement von einem anderen aus zu erreichen ist, so bietet sich eher eine Darstellung durch einen gerichteten Graphen an. Die Knoten des erzeugten Graphen repräsentieren einzelne Instruktionen und seine Kanten Kontrollflußübergänge zwischen Instruktionen.

Definition 3.1.1 *Ein* Kontrollflußgraph (CFG[1]) *ist ein Graph* $G = (N, E, s, e)$ *mit der Knotenmenge* N*, der Kantenmenge* $E \subseteq N \times N$*, einem Startknoten*

[1]Control Flow Graph

s und einem Endknoten e mit $s, e \in N$. Wenn n ein direkter Kontrollfluß-vorgänger von m ist, so gilt $(n, m) \in E$. n ist Vorgänger *von m, m ist ein* Nachfolger *von n. Der Startknoten s hat keinen Vorgänger, der Endknoten e keinen Nachfolger.*

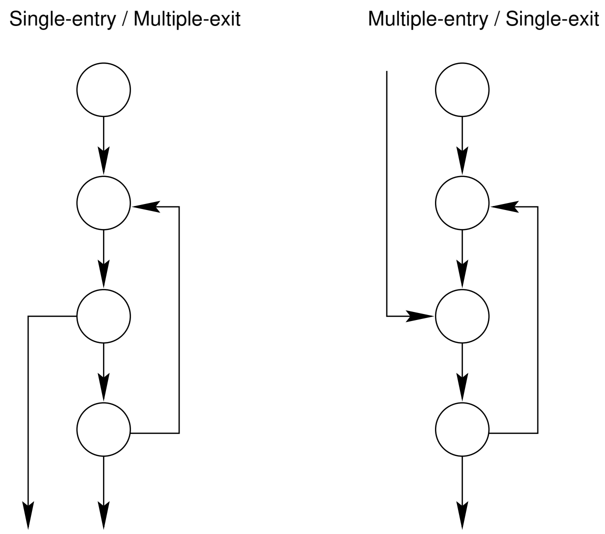
Definition 3.1.2 *Ein* Pfad *π in einem $CFG(N, E, s, e)$ besteht aus einer Folge von Kanten. π beginnt an einem Knoten n_1 und endet an einem Knoten n_k : $\pi = (n_1, n_2), (n_2, n_3), \ldots, (n_{k-1}, n_k)$ mit $(n_i, n_{i+1}) \in E, i < k$. Ein Pfad, der bei einem Knoten n_i beginnt und bei n_j endet, wird durch $n_i \rightsquigarrow n_j$ beschrieben.*

Kontrollflußgraphen bilden die Grundlage vieler Programmanalysen, da sie auf einfache und intuitive Weise mögliche Ausführungsreihenfolgen von Instruktionen modellieren.

Innerhalb von Kontrollflußgraphen können „zusammengehörige" Abschnitte zu einem *Basisblock* zusammengefaßt werden. In einem Programm sind Basisblöcke Programmfragmente, die immer „am Stück" ausgeführt werden. Wird die erste Instruktion eines Basisblocks ausgeführt, so wird auch unweigerlich dessen letzte Instruktion ausgeführt. In einem Basisblock gibt es keine Verzweigungen oder Sprungziele. Daher besteht keine Möglichkeit, einen Basisblock „in der Mitte" durch eine Verzweigung zu verlassen, oder durch einen Sprung dorthin zu gelangen.

Definition 3.1.3 *Eine* Basisblock *ist eine Folge von Knoten $(n_1, \ldots, n_k)$ eines CFG $FG = (N, E, s, e)$, so daß n_i der einzige Vorgänger von n_{i+1} ist, und n_{i+1} der einzige Nachfolger von n_i ist.*

Beispiel 3.1.1 *Schleifen mit mehreren Ein-/Ausgängen*



Die meisten Kontrollflußgraphen enthalten Zyklen mit mehreren beteiligten Knoten. Alle an einem Zyklus beteiligten Knoten bilden eine *Schleife (loop)*.

Wenn es einen Knoten gibt, über den alle Pfade von Startknoten s zu den Schleifenknoten führen müssen, d.h. dieser Knoten der einzige Knoten ist, der in die Schleife führt, dann besitzt die Schleife die *Single-entry*-Eigenschaft. Analog hat eine Schleife die *Single-exit*-Eigenschaft, wenn es einen Knoten gibt, über den alle Pfade aus der Schleife zum Endknoten e führen müssen. Schleifen, deren Pfade in die Schleife hinein oder aus der Schleife heraus über mehrere alternative Knoten verlaufen können, haben die *Multiple-entry-* bzw. *Multiple-exit*-Eigenschaft. Beispiel 3.1.1 zeigt verschiedene Schleifen mit den gerade genannten Eigenschaften.

Single-entry/Single-exit-Schleifen, die aus programmiersprachlichen Konstrukten wie `for i= ...` enstehen, also eine festgelegte Anzahl Iterationen[2] absolvieren, und auch nicht vorzeitig durch Sprünge beendet werden, heißen *strukturierte Schleifen* (Beispiel 3.1.2,a)).

Beispiel 3.1.2 *Verschiedene Schleifentypen*

a) Strukturierte Schleife

```
for(i = 0; i < 100; i++)
{
  a = x;
  if (a > y) {
     b = c;
  }
  else {
     x = y;
  }
}
```

b) Enge Schleifenschachtelung

```
for(i = 0; i < 100; i++)
{
   for(j = 0; j < 100; j++)
   {
      for(k = 0; k < 100; k++)
      {
         a[i][j][k] = 5;
         b[k] = f(k);

      }
   }
}
```

Strukturierte Schleifen Schleifen haben einen *Schleifenkopf* mit der Initialisierung und dem Test des Abbruchkriteriums, sowie einen *Schleifenkörper*, der aus der Sequenz von wiederholt ausgeführten Instruktionen besteht. Werden mehrere Schleifen ineinander geschachtelt, so liegt eine *Schleifenschachtelung (loop nest)* vor. Eine besondere Art eines loop nest ist dann gegeben, wenn mehrere strukturierte Schleifen direkt ineinander geschachtelt sind, d.h. alle Schleifen bis auf die innerste enthalten ausschließlich ein Statement, nämlich eine strukturierte Schleife. Eine solche Schleifenschachtelung wird *enge Schleifenschachtelung (tight loop nest)* (Beispiel 3.1.2,b)) genannt.

[2]Dazu gehört auch, daß die *Induktionsvariable* i im Schleifenkörper nicht redefiniert wird.

3.2 Abhängigkeitsrelationen

Die Abhängigkeiten der Instruktionen untereinander können durch Relationen beschrieben werden. Zum einen gibt es Abhängigkeiten zwischen Instruktionen aufgrund der Einschränkungen der Ausführungsreihenfolge durch den Kontrollfluß, zum anderen können Abhängigkeiten zwischen Instruktionen wegen gemeinsam benutzter Daten auftreten.

Definition 3.2.1 *Zwei Instruktionen S_1 und S_2 stehen genau dann zueinander in der Relation $S_1 \lhd S_2$, wenn die Instruktion S_1 der Instruktion S_2 in der Ausführungsreihenfolge vorangeht, also im CFG ein Pfad $S_1 \leadsto S_2$ existiert.*

Abhängigkeiten zwischen Instruktionen können deren Ausführungsreihenfolge einschränken. Je nach Ursache einer Abhängigkeit unterscheidet man zwischen *Kontrollabhängigkeiten*, die durch zwingend zu beachtende Kontrollstrukturen (IF ... THEN ... ELSE ...) in Programmen entstehen, und *Datenabbhängigkeiten*, bei denen eine Datum einer Instruktion in einer späteren (abhängigen) Instruktion erneut verwendet wird.

Definition 3.2.2 *Eine* Kontrollabhängigkeit (control dependence) *zwischen zwei Instruktionen erwächst aus dem Kontrollfluß eines Programms. Wenn eine Instruktion S_2 abhängig von der Auswertung der Instruktion S_1 ausgeführt wird, so ist S_2 von S_1 kontrollabhängig. Die formale Schreibweise dieser Kontrollabhängigkeit ist $S_1 \delta^c S_2$.*

Beispiel 3.2.1 *Kontrollabhängigkeit*

S_1 : `x = a;`
S_2 : `if (x > 3)`
S_3 : `    b = b + 1;`

In Beispiel 3.2.1 gibt es eine Kontrollabhängigkeit zwischen S_2 und S_3 ($S_2 \delta^c S_3$), da S_3 nur dann ausgeführt wird, wenn die Bedingung in S_2 wahr ist.

Bevor weitere Definitionen erfolgen können, muß eine feinere Differenzierung von Datenzugriffen getroffen werden. Ein lesender Zugriff auf eine Variable ist ein *Gebrauch*, während ein schreibender Zugriff eine *Definition* ist. Kommt es nicht auf die Art des Zugriffs an, d.h. es sind sowohl Gebräuche als auch Definitionen gemeint, kann von *Referenzen* gesprochen werden.

Definition 3.2.3 *Eine* Datenabhängigkeit (data dependence) *zwischen zwei Instruktionen ist eine mögliche Einschränkung der Ausführungsreihenfolge, die durch den Datenfluß zwischen diesen Instruktionen verursacht wird. Eine Instruktion S_2 ist* datenabhängig *von S_1 (formale Schreibweise $S_1 \delta^d S_2$), wenn eine in S_1 referenzierte Variable in S_2 erneut referenziert wird.*

Die Definition der Datenabhängigkeit ist in dieser Form sehr allgemein, da nichts über die Art der Wiederverwendung des Datums ausgesagt wird. Eine genauere Unterscheidung der Datenabhängigkeit ergibt sich durch Differenzierung nach der Art Zugriffs (Lesen/Schreiben) in den beteiligten Instruktionen.

Definition 3.2.4 *Wenn $S_1 \lhd S_2$ und S_1 eine Variable definiert, die in S_2 gebraucht und zwischen S_1 und S_2 nicht redefiniert wird, dann existiert zwischen S_1 und S_2 eine* Datenflußabhängigkeit (flow dependence, true dependence)*. S_1 und S_2 stehen in der Relation $S_1 \delta^f S_2$ zueinander.*

Definition 3.2.5 *Wenn $S_1 \lhd S_2$ und S_1 eine Variable gebraucht, die S_2 definiert ohne eine Redefinition der betreffenden Variablen zwischen S_1 und S_2, dann besteht zwischen S_1 und S_2 eine* Antiabhängigkeit (antidependence) *$S_1 \delta^a S_2$.*

Definition 3.2.6 *Wenn $S_1 \lhd S_2$ und S_1 und S_2 die gleiche Variable definieren und zwischenzeitlich keine weitere Definition dieser Variablen erfolgt, besteht eine* Ausgabeabhängigkeit (output dependence) *zwischen S_1 und S_2, bzw. $S_1 \delta^o S_2$.*

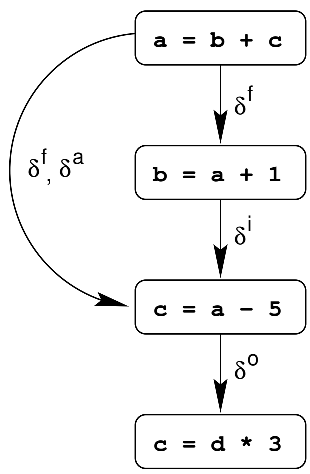
Definition 3.2.7 *Wenn $S_1 \lhd S_2$ und S_1 und S_2 eine gemeinsame Variable lesen, die zwischenzeitlich nicht neu definiert wird, so existiert zwischen S_1 und S_2 eine* Eingabeabhängigkeit (input dependence) *$S_1 \delta^i S_2$.*

Beispiel 3.2.2 *Abhängigkeiten und Abhängigkeitsgraph*

Codefragment

```
a = b + c;
b = a + 1;
c = a - 5;
c = d * 3;
```

Abhängigkeitsgraph



Von den vier Datenabhängigkeitsrelationen schränken die ersten drei die Ausführungsreihenfolge ein, nicht jedoch die vierte – die input dependence. Jedoch werden im Zusammenhang mit Optimierungen von Array-Zugriffen *Load-After-Load*-Situationen (siehe Kapitel 5.2) betrachtet, die dem Konzept der input dependence entsprechen.

Abhängigkeiten können durch gerichtete Graphen dargestellt werden, deren Knoten den Instruktionen entsprechen, und deren Kanten eine Abhängigkeit darstellen. Dabei wird eine Kante mit der Art der Abhängigkeit markiert. Für eine Abhängigkeit $S_1 \delta S_2$ wird die Kante von S_1 nach S_2 gerichtet. Beispiel 3.2.2 zeigt die Datenabhängigkeiten zu einem Code-Fragment in einem Abhängigkeitsgraphen.

3.3 Datenabhängigkeitsanalysen

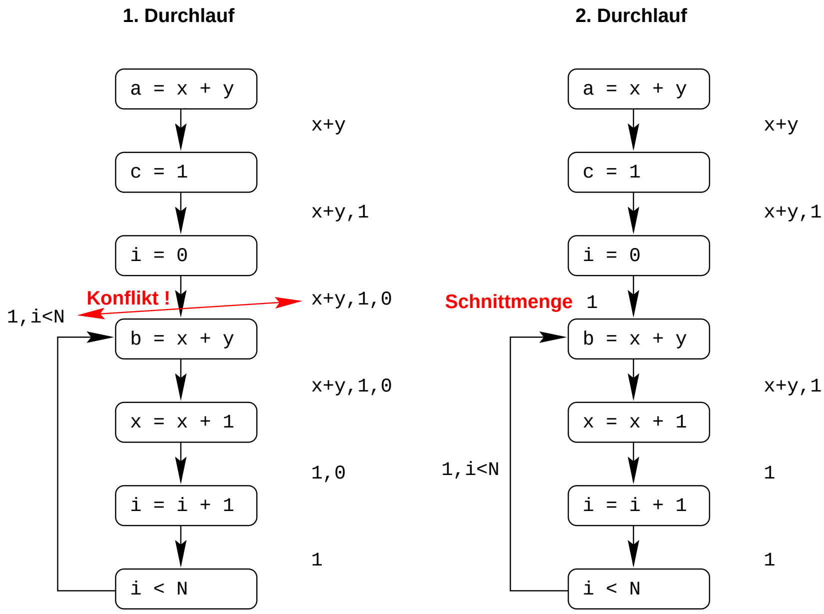
Der Schwerpunkt dieser Arbeit liegt bei der Untersuchung und Behandlung von Datenabhängigkeiten. Daher sollen zunächst die grundlegenden Techniken der Datenflußanalyse, die u.a. in [3], [15] und [11] nachzulesen sind, erläutert werden.

Beispiel 3.3.1 zeigt ein kleines Programmfragment, welches daraufhin überprüft werden soll, ob es gemeinsame Teilausdrücke enthält, die entfernt werden können (siehe Bsp. 3.4.1 zu näheren Detail dieser Optimierung). Ein Kandidat ist der Ausdruck `x+y`, der zu Beginn und in der Schleife vorkommt. Eine Analyse erfolgt in einer graphischen Darstellung der Schleife, dem *Kontrollflußgraphen*. In einem ersten Analysedurchlauf werden der Reihe nach alle Knoten besucht. Für jeden Knoten wird als dessen Eigenschaft notiert, welche Ausdrücke nach seiner Ausführung verfügbar sind. Ein Ausdruck ist verfügbar, wenn er einmal ausgewertet wurde und seine Operanden im nachhinein nicht verändert werden. Nach Bearbeitung der Instruktion `i<N` ergibt sich ein Konflikt. Entgegen der Feststellung beim ersten Betrachten des Knotens `b = x+y` erreichen diesen Knoten von dem erst später betrachteten Knoten `i<N` andere Ausdrücke als zuvor. Die Lösung dieses Konfliktes liegt in der Bildung der Schnittmenge der Ausdrücke und einem weiteren Durchlauf. Damit wird sichergestellt, daß nur mit Ausdrücken gearbeitet wird, die den Knoten `b = x+y` auf allen Pfaden erreichen – in diesem Fall 1. Beim zweiten Durchlauf müssen die Ergebnisse des vorherigen Durchlaufs durch diese Veränderung revidiert werden. Am Ende des Durchlaufs wird für `b = x+y` wieder die Schnittmenge der eintreffenden Ausdrücke gebildet, es ergibt sich wie zuvor die 1. Weitere Durchläufe können das Ergebnis nicht mehr verändern, die Lösung hat sich stabilisiert. Damit kann die Analyse beendet werden. Für die Optimierung muß das Ergebnis interpretiert werden. Die Frage war, ob `x+y` in `b=x+y` durch den Ausdruck aus `a=x+y` ersetzt werden kann. Da der Ausdruck `x+y` die Instruktion `b=x+y` aber nicht zwingend erreicht – dies schafft nur der Ausdruck 1 – kann die Optimierung nicht durchgeführt werden.

Beispiel 3.3.1 *Analyse eines Programmfragments*

Programmfragment mit Schleife

```
a = x + y;
c = 1;
for(i = 0; i < N; i++)
{
   b = x + y;
   x = x + 1;
}
```



Die folgenden Abschnitte führen in die grundlegenden Techniken Datenabhängigkeitsanalyse ein. Nach der Einführung geeigneter Abhängigkeitsbegriffe im vorherigen Abschnitt wird gezeigt, wie darauf basierend Schritt für Schritt ein formales Modell eines zu analysierenden Programms erstellt werden kann. Die Eigenschaften – im Beispiel die Eigenschaft, daß ein Ausdruck einen Knoten erreicht – können durch Verbände realisiert werden. Das Erzeugen und Vernichten von Eigenschaften – im Beispiel, ob ein Ausdruck nach einem Knoten verfügbar ist oder ob ein Ausdruck einen Knoten „passieren“ kann – wird durch Funktionen, die auf den Verbänden operieren, nachgebildet. Insgesamt bildet diese Modell die geeignete Voraussetzung zum Ansatz eines Verfahrens, daß die gewünschten Abhängigkeiten berechnen kann. Dazu wird wie im Beispiel iterativ vorgegangen bis sich die Knoteninformationen nicht weiter verändern. Mit der so gewonnen Information können nachfolgende Optimierungen ihre Arbeit aufnehmen.

Das Ziel der Datenflußanalyse ist es, bestimmte Eigenschaften bzgl. exisitierender Datenabhängigkeiten für alle Stellen eines gegebenen Programms zu berechnen. Diese Eigenschaften werden durch Werte einer Domäne D beschrieben.

Während und nach der Analyse werden alle Knoten n eines CFG mit Werten aus D versehen, welche die Gültigkeiten der betrachteten Eigenschaften vor und nach Ausführung der mit dem Knoten n verbundenen Instruktion[3] beschreiben. Die Auswirkungen einer Instruktion werden durch eine Transferfunktion $f : D \to D$ modelliert, d.h. eine mögliche Veränderung einer vor Eintritt in n geltenden Eigenschaft wird überführt in den Zustand nach Verlassen von n. Die Transferfunktion wird immer dann angewendet, wenn der Kontrollfluß den Knoten n passiert. Dementsprechend bildet der CFG eines Programms, dessen Knoten mit Transferfunktionen versehen sind, ein spezielles Gleichungssystem, das *Datenflußgleichungssystem*.

3.3.1 Datenflußverbände

Die Domäne D, durch die die zu analysierenden Eigenschaften abstrahiert werden, wird i.a. durch einen *Verband* L realisiert. Ein Verband bietet neben einer Menge von Werten, die den Eigenschaften, wie z.B. Gültigkeit eines zuvor definierten Ausdrucks, entsprechen, Operatoren für diese Werte. Dadurch können mehrere zusammenkommende Eigenschaften zu einer neuen kombiniert werden.

Im Eingangsbeispiel 3.3.1 besteht die Domäne D aus Aussagen über Ausdrücken, die in dem Programm vorkommen. Die Aussage, daß ein Ausdruck einen Knoten erreicht, kann wahr oder falsch sein. Die Schnittmengenbildung bei den zusammenlaufenden Kontrollflußpfaden ist ein Operator auf der Aussagenmenge. In praktischen Realisierungen werden statt der Mengenoperationen aussagenlogische Operatoren bevorzugt.

Definition 3.3.1 *Ein* Verband L *besteht aus einer Menge von Elementen, der* Trägermenge, *und zwei Operatoren* $\sqcap$ (meet) *und* $\sqcup$ (join) *mit folgenden Eigenschaften:*

1. *Abgeschlossenheit:*
   $\forall x, y \in L : \exists z, w \in L : x \sqcap y = z \quad \wedge \quad x \sqcup y = w$
2. *Kommutativität:*
   $\forall x, y \in L : x \sqcap y = y \sqcap x \quad \wedge \quad x \sqcup y = y \sqcup x$
3. *Assoziativität:*
   $\forall x, y, z \in L : (x \sqcap y) \sqcap z = x \sqcap (y \sqcap z) \quad \wedge \quad (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
4. *Supremum, Infimum:*
   - $\exists \bot \in L : \forall x \in L : x \sqcap \bot = \bot$
   - $\exists \top \in L : \forall x \in L : x \sqcup \top = \top$

   Das Element $\bot$ *heißt* bottom, $\top$ *heißt* top.

[3] Anstelle einer einzelnen Instruktion kann auch eine zu einem Basisblock zusammengefaßte Sequenz von Instruktionen durch einen Knoten repräsentiert werden.

Definition 3.3.2 *Ein Verband L heißt* distributiver Verband, *falls zusätzlich gilt:*
$\forall x, y, z \in L : (x \sqcap y) \sqcup z = (x \sqcup z) \sqcap (y \sqcup z) \quad \wedge \quad (x \sqcup y) \sqcap z = (x \sqcap z) \sqcup (y \sqcap z)$

Definition 3.3.3 *Die auf einem Verband L induzierte partielle Ordnung* $(L, \sqsubseteq)$ *ist durch* $\forall x, y \in L : x \sqsubseteq y \iff x \sqcap y = x$ *definiert. Die gleiche Definition ist auch mit dem* $\sqcup$*-Operator möglich.* $\sqsubset$, $\sqsupset$ *und* $\sqsupseteq$ *sind analog definiert.*

Definition 3.3.4 *Die* Höhe *eines Verbandes L ist definiert als:*
$height(L) = max\{n | \exists x_1, x_2, \ldots, x_n : \bot = x_1 \sqsubset x_2 \sqsubset \cdots \sqsubset x_n = \top\}$

Beispiel 3.3.2 *Der binäre Verband* $\mathbf{B} = (\{\bot, \top\}, \bot, \top, \bot < \top, min, max)$ *ist distributiv und hat die Höhe* $height(\mathbf{B}) = 2$. $\mathbf{B}$ *wird häufig in Datenflußanalysen benutzt. Ist eine Eigenschaft mit* $\top$ *an einem Knoten gekennzeichnet, so hat sie dort Gültigkeit. Ungültige Aussagen über Eigenschaften werden mit* $\bot$ *beschrieben.*

3.3.2 Transferfunktionen

Durch die Datenflußverbände werden Eigenschaften von Datenabhängigkeiten modelliert. Diese Eigenschaften verändern sich aber durch die Ausführung von Instruktionen. Daher wird ein Formalismus benötigt, der in der Lage ist, die durch die Programmausführung verursachten Veränderungen bzgl. der untersuchten Eigenschaften auf der Ebene der Datenflußverbände umzusetzen. Zu diesem Zweck werden *Transferfunktionen* definiert. Eigenschaften, die bei Erreichen eines Knoten gelten, müssen dies nicht auch bei Verlassen desselben tun, und umgekehrt. Damit modellieren Transferfunktionen die Semantik von Instruktionen in Bezug auf bestimmte Eigenschaften, wie z.B. die Veränderungen benutzter Variablen. Das Verhalten einer Sequenz von Instruktionen kann demnach durch fortgesetzte Anwendung der Transferfunktionen beschrieben werden.

Transferfunktionen sind Funktionen $f : L \rightarrow L$, die der Knotenmenge[4] des CFG FG=(N,E,s,e) zugeordnet sind: $tf : N \rightarrow (L \rightarrow L)$.

Die konkrete Definition einer Transferfunktion hängt von der zu beschreibenden Instruktion, den beteiligten Operanden und von der Art der zu untersuchenden Datenflußeigenschaften ab. Für Instruktionen ist es ausreichend zu beschreiben, wie sie auf ihre Operanden zugreifen. Es ist nicht wichtig, welche konkrete – z.B.

[4] Transferfunktionen können auch den Kanten des CFG zugeordnet sein. Damit kann das Verhalten von verschiedenen Kanten, die einen Knoten erreichen, differenziert beschrieben werden.

arithmetische – Operation sie ausdrücken. Wird eine Variable gelesen, so ist dies ein *Gebrauch*, wird sie geschrieben, handelt es sich dabei um eine *Definition*. Ist es egal, ob es sich um einen Gebrauch oder eine Definition handelt, so wird der allgemeinere Begriff *Referenz* verwendet. Je nachdem, ob eine Referenz dafür sorgt, ob fortan eine bestimmte Eigenschaft gilt, oder eine Eigenschaft nicht mehr gilt, handelt sich dabei um eine *erzeugende (generate)* oder *vernichtende (kill)* Referenz. Anhand der Eigenschaft einer Instruktion, ob sie erzeugend oder vernichtend ist, die nicht nur von der Instruktion selbst, sondern auch von der Art der zu bestimmenden Information abhängig ist, kann dann die Transferfunktion festgelegt werden.

Monotonie ist für Transferfunktionen eine wichtige Eigenschaft, sowie auch die *effektive Höhe* eines Verbandes bzgl. einer Funktion. Beide dienen dazu, später Aussagen zur Terminierung und Zeitkomplexität von Datenflußanalysen machen zu können.

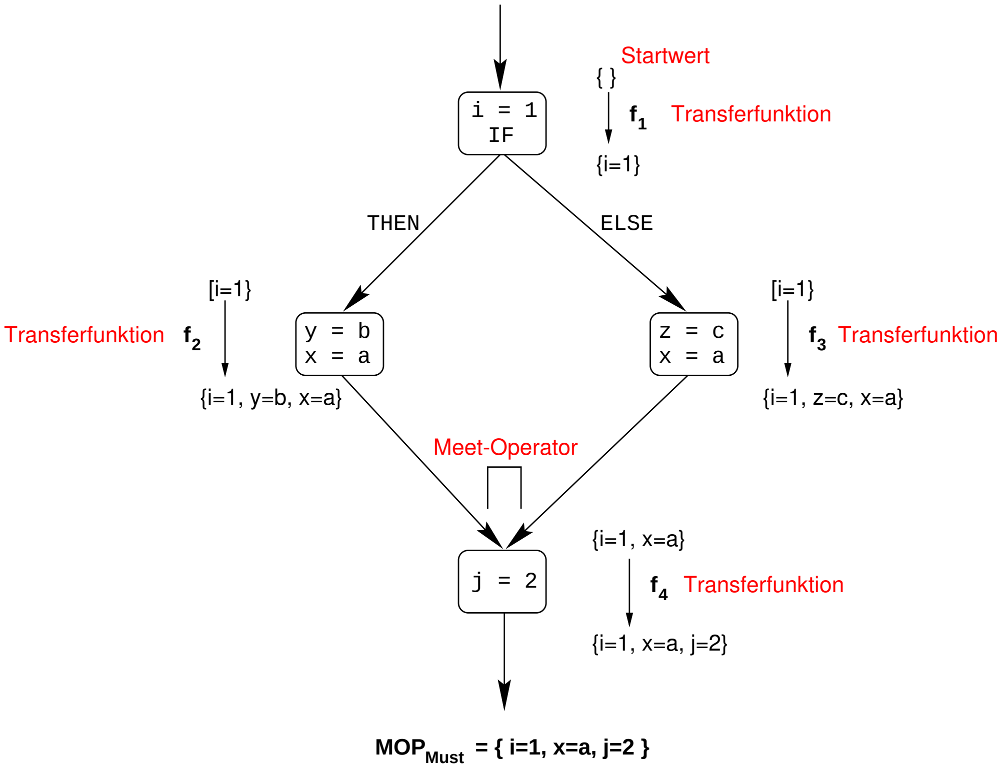
Definition 3.3.5 *Eine Funktion $f : L \rightarrow L$ heißt* monoton*, wenn gilt:*
$\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$.

Definition 3.3.6 *Die* effektive Höhe *eines Verbandes L bzgl. einer Funktion $f : L \rightarrow L$ ist :*
$height^f_{eff}(L) = max\{n | \exists x_1, x_2 = f(x_1), x_3 = f(x_2), \ldots, x_n = f(x_{n-1}) \in L : x_1 \sqsubset x_2 \sqsubset x_3 \sqsubset \cdots \sqsubset x_n \sqsubseteq \top$.
Mit der effektiven Höhe ist also die längste, streng monoton aufsteigende Kette iterierter Anwendungen der Funktion f gemeint.

Definition 3.3.7 *Sei π ein Pfad im CFG FG=(N,E,s,e) und seien $n_1, \ldots, n_k$ die an π beteiligten Knoten in ihrer Ausführungsreihenfolge. Dann bestimmt sich die* Transferfunktion des Pfades π *durch* $f_\pi = f_{n_1} \circ \ldots \circ f_{n_k}$.

Im allgemeinen ist ein Knoten n nicht nur über einen einzelnen Pfad zu erreichen, sondern es gibt mehrere Pfade, über die n von einem Startknoten aus zu erreichen ist. Daher ist eine Lösung anzustreben, die ein Erreichen eines Knoten n über alle Pfade ausdrückt. Je nach Zweck der späteren Verwendung liegt dabei das Interesse an der Erkundung aller Datenflußeigenschaften, die an Knoten n gelten *können (may)*, oder an n gelten *müssen (must)*. Der Unterschied entsteht dadurch, daß es potentielle und zwingende Gültigkeiten geben kann. Potentielle Gültigkeiten können aus der Lage von Instruktionen in nur einem von zwei möglichen Verzweigungsästen entstehen. Bei einer Programmausführung können wegen der Abarbeitung nur eines Verzweigungsastes Instruktionen unberücksichtigt bleiben. Sie könnten bei einer anderen Programmausführung aber durchaus durchgeführt werden. Alle potentiellen Gültigkeiten, die durch solche Situationen entstehen können, werden in der *May*-Lösung erfaßt. Werden alle zwingenden Gültigkeiten ermittelt, die unabhängig von einer konkreten Verzweigung Bestand haben, werden diese in der *Must*-Lösung zusammengebracht. Es werden diejenigen Möglichkeiten ausgespart, die auf einigen – aber nicht allen – Pfaden zu n Gültigkeit erlangen.

Beispiel 3.3.3 *Datenflußanalyse zur Gewinnung von* MOP_{Must}



Das Beispiel 3.3.3 zeigt für einen Programmausschnitt mit einer Verzweigung die zwingenden Gültigkeiten beim Problem *reaching definitions*. Vom Startwert mit der leeren Menge aus werden für jeden Knoten Transferfunktionen auf die sie erreichenden Definitionen angewendet. Beim Zusammenführen von Kontrollflußpfaden wird ein *Meet*-Operator angewendet, der aus mehreren Mengen eine gemeinsam zwingend gültige Lösung bestimmt.

Entsprechend *May* oder *Must* gibt es zwei unterschiedliche Darstellungen[5], die ausgehend von einem Startwert ι die Gültigkeit an einem Knoten n bezeichnen.

- Sei für einen Knoten $n \in N$ die Menge aller Pfade $s \rightsquigarrow n$ gegeben durch $\pi(n)$. Die *Merge-over-all-path (MOP)*-Lösung für das *May*-Problem ist dann
$$MOP_{May}(n) = \bigsqcup_{p \in \pi(n)} f_p(\iota) \tag{3.1}$$
für einen Initialwert $\iota \in L$.

[5]Gewöhnlich wird in der Literatur nur eine von beiden Lösungen dargestellt, ohne auf die andere hinzuweisen oder eine Unterscheidung anzugeben. Beim Zusammenführen mehrerer Quellen kann es daher Schwierigkeiten geben.

- Sei für einen Knoten $n \in N$ die Menge aller Pfade $s \rightsquigarrow n$ gegeben durch $\pi(n)$. Die *Meet-over-all-path (MOP)*-Lösung für das *Must*-Problem ist dann

$$MOP_{Must}(n) = \prod_{p \in \pi(n)} f_p(\iota) \tag{3.2}$$

für einen Initialwert $\iota \in L$.

Beispiel 3.3.4 *MOP_{May} und MOP_{Must}*

S_1: `a = x+y;`
S_2: `if (k > l)`
S_3: `    b = x+1;`
S_4: `else c = y+1;`
S_5: `d = c+d;`

- *Analyse:* Ausdrücke, die einen Knoten erreichen
- *Initialwert:* $\iota = \{\}$
- $MOP_{May}(S_5) = \{x + y, k > l, x + 1, y + 1\}$
- $MOP_{Must}(S_5) = \{x + y, k > l\}$

Das Beispiel 3.3.4 zeigt für ein kurzes Programm diejenigen Ausdrücke, die das Statement S_5 erreichen können oder müssen. Der Initialwert ist die leere Menge, da zu Beginn noch keine Ausdrücke ausgewertet vorliegen. Ausdrücke, die nur in einem Verzweigungsast vorkommen, erscheinen nur in der May-Lösung. Die Must-Lösung enthält nur die Ausdrücke, die unabhängig vom Verlauf der Verzweigung sind.

Die MOP-Lösungen enthalten die exakten Abhängigkeiten für alle Knoten n. Wenn die Transferfunktionen nicht monoton sind, oder es eine unendliche Anzahl verschiedener Pfade oder Elemente des Verbandes L gibt, sind die MOP-Lösungen nicht berechenbar. Das Problem der Bestimmung von Datenabhängigkeiten ist unentscheidbar. Wenn jedoch die Forderung nach *exakter* Bestimmung aller Datenabhängigkeiten fallengelassen wird, gibt es einen Ausweg. Wird einem Verfahren erlaubt, die Gültigkeit einer Aussage falsch zu bestimmen, d.h. fälschlicherweise Gültigkeit statt Ungültigkeit und umgekehrt zu berichten, so bestimmt es eine *Näherungslösung* oder *Approximation* der vollständig korrekten Lösung. Die Anzahl und die „Schwere" der Fehler eines *Approximationsverfahrens* lassen sich durch seine *Präzision* ausdrücken. Eine hohe Präzision ist erstrebenswert, denn damit wird die korrekte Lösung möglichst nahe erreicht.

Beispiel 3.3.5 *Approximation der MOP_{Must}-Lösung*

Zum Beispiel 3.3.4 könnte ein Verfahren folgende Approximationslösung für $MOP_{Must}(S_5)$ liefern: $MOP_{Must}(S_5) = \{x+y\}$. Der Fehler, der gemacht wurde, liegt im „Unterschlagen" des Lösungsanteils $k > l$.

Neben der Präzision gibt es aber noch ein weiteres sehr wichtiges Merkmal einer Approximation, nämlich die Art der möglichen Fehler. Es bestehen die Möglichkeiten, eine gültige Aussage fälschlich als ungültig zu klassifizieren, oder eine ungültige Aussage als gültig zu bezeichnen. Wird in einer der Analyse folgenden Optimierung ein falsches Ergebnis genutzt, so kann das je nach Fehler dazu führen, daß eine mögliche Gelegenheit zur Optimierung nicht erkannt wird, oder schlimmer eine Optimierung durchgeführt, die zu semantisch inkorrektem Programmverhalten führt. Während ein Fehler der ersten Art „nur" die Effizienz eines Programms beeinflußt, Fehler der zweiten Art jedoch die Korrektheit, sind letztere unbedingt zu vermeiden. Um die geforderte *Sicherheit* zu gewährleisten, dürfen nicht beide Fehlerarten *gleichzeitig* auftreten. Für ein Must-Problem ist es daher besser, weniger Eigenschaften als zwingend (must) einzustufen als tatsächlich vorliegen. Die irrtümliche Klassifikation einer Eigenschaft als gültig, ist „gefährlich". Bei May-Problemen ist es umgekehrt. Hier ist es besser, mehr potentielle Gültigkeiten zu berichten, als eine Möglichkeit zu unterschlagen. Auch Aussagen zur Gültigkeit einer bestimmten Eigenschaft wie „unbekannt" sind besser als falsche Aussagen.

Die Beispiele 3.3.6 und 3.3.7 verdeutlichen die Wichtigkeit der Sicherheit von Approximationen. In Beispiel 3.3.6 wird der Ausdruck $k > l$ fehlerhaft als nicht den Knoten S_5 erreichend klassifiziert. Dadurch kann eine mögliche Gelegenheit zur Common Subexpression Elimination nicht erkannt werden. Die Folge ist, daß eine Optimierungsmöglichkeit „verschenkt" wird, das Programm bleibt aber korrekt. In Beispiel 3.3.7 wird durch die fehlerhafte Klassifizierung von $y+1$ bewirkt, daß eine CSE durchgeführt wird, die die Korrektheit des Programms zerstört.

Beispiel 3.3.6 *Verhinderung einer CSE-Optimierung durch eine sichere Approximation $MOP_{Must}(S_5) = \{x+y\}$*

Ursprüngl. Programm	***Verpaßte Optimierung***
S_1: `a = x+y;`	*S_1:* `a = x+y;`
S_2: `if (k > l)`	*S_2:* `if (t = (k > l))`
S_3: `    b = x+1;`	*S_3:* `    b = x+1;`
S_4: `else c = y+1;`	*S_4:* `else c = y+1;`
S_5: `if (k > l)`	*S_5:* `if (t)`

Beispiel 3.3.7 *Fehlerhafte CSE-Optimierung durch eine unsichere Approximation* $MOP_{Must}(S_5) = \{x+y, k > l, y+1\}$

Ursprüngl. Programm	***Fehlerhafte Optimierung***
S_1: `a = x+y;`	S_1: `a = x+y;`
S_2: `if (k > l)`	S_2: `if (k > l)`
S_3: `    b = x+1;`	S_3: `    b = x+1;`
S_4: `else c = y+1;`	S_4: `else t = (c = y+1);`
S_5: `x = y+1`	S_5: `x = t`

Formal ergibt sich aus den Forderungen, daß eine *sichere Approximation* für ein Must-Problem eine Untermenge der MOP-Lösung sein muß, hingegen für ein May-Problem eine Obermenge der MOP-Lösung. Zum Beispiel 3.3.4 zeigt das folgende Beispiel 3.3.8 die Mengenbeziehungen zwischen der sicheren, der unsicheren und der exakten Lösung.

Beispiel 3.3.8 *Sichere und unsichere Approximationen*

- *Exakte Lösung:* $MOP_{Must}(S_5) = \{x+y, k > l\}$
- *Sichere Approximation:*
  $MOP_{Must}^{Approx}(S_5) = \{x+y\} \subset MOP_{Must}(S_5) = \{x+y, k > l\}$
- *Unsichere Approximation:*
  $MOP_{Must}^{Approx}(S_5) = \{x+y, k > l, y+1\} \not\subset MOP_{Must}(S_5) = \{x+y, k > l\}$

Sichere Approximationen lassen sich bestimmen, wenn die Höhe $height(L)$ des verwendeten Datenflußverbandes beschränkt ist und die Transferfunktionen monoton sind [11]. Dann nämlich können ausgehend von Initialwerten fortlaufend die Transferfunktionen angewendet werden, bis sich nach einigen Iterationen die den Knoten zugeordneten Eigenschaftswerte nicht mehr verändern. Es ist ein *Fixpunkt* erreicht.

Im Eingangsbeispiel 3.3.1 wurde bereits, ohne die formalen Voraussetzungen zu haben, ein Fixpunkt berechnet. Dort wurde die Datenflußinformation solange von Knoten zu Knoten propagiert bis sich ein Zustand einstellte, bei dem sich die Ausdrucksmengen durch weitere Durchläufe nicht mehr änderten.

Definition 3.3.8 *Ein* Fixpunkt *einer Funktion* $f : L \to L$ *ist ein Element* $x \in L$ *mit* $f(x) = x$.

Ein Fixpunkt ist eine Lösung eines Gleichungssystems aus Transferfunktionen. Jede weitere Anwendung einer Funktion verändert das Ergebnis nicht mehr. Das bedeutet, daß sich alle Abhängigkeiten (evtl. zyklisch) fortgepflanzt haben und sich das Gleichungssystem als ganzes stabilisiert hat. Damit verändern sich auch die Gültigkeiten der Datenflußeigenschaften nicht mehr, sie sind an allen Knoten bestimmt.

Zur Approximation der MOP-Lösung werden, ausgehend von einem Startwert, alle Transferfunktionen eines Pfades zu einem Knoten n solange angewendet, bis keine Veränderung der Lösungen mehr stattfindet. An Knoten zusammenlaufenden Kontrollflusses werden die Teillösungen wie folgt zusammengefaßt:

- Ein *Minimaler Fixpunkt (MFP)* für das *May*-Problem ist

$$MFP_{May}(n) = \begin{cases} \iota & \text{, falls } n = s \\ \bigsqcup\limits_{m \in pred(n)} f_n(n)(MFP(m)) & \text{, sonst} \end{cases} \tag{3.3}$$

  für einen Startwert $\iota \in L$.

- Ein *Maximaler Fixpunkt (MFP)* für das *Must*-Problem ist

$$MFP_{Must}(n) = \begin{cases} \iota & \text{, falls } n = s \\ \bigsqcap\limits_{m \in pred(n)} f_n(n)(MFP(m)) & \text{, sonst} \end{cases} \tag{3.4}$$

  für einen Startwert $\iota \in L$.

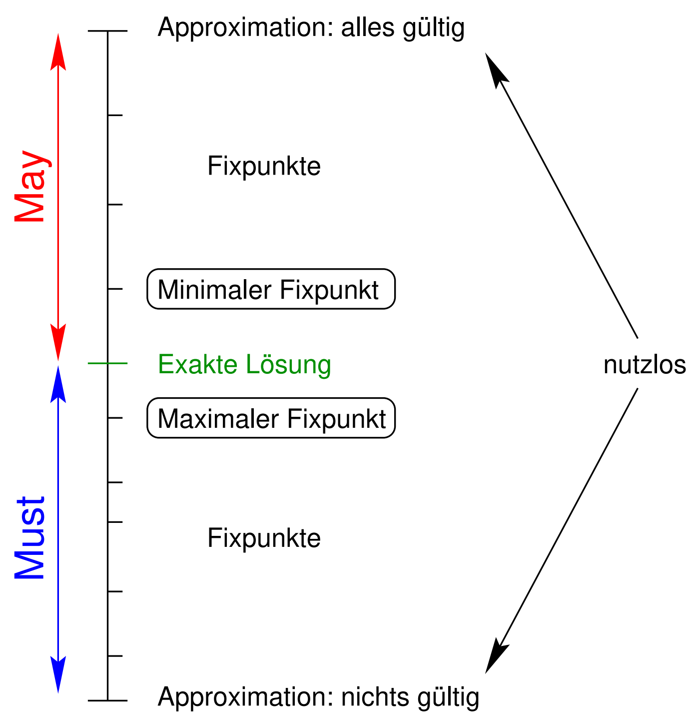


Abbildung 3.1: Exakte Lösung und Annäherungen durch Fixpunkte

Abbildung 3.3.2 zeigt die Bedeutung von Fixpunkten bei der Annäherung an die exakte Datenflußlösung. Um die exakte Lösung herum gibt es zwei Bereiche – den Bereich der Must-Lösungen und den Bereich der May-Lösungen. Obere Grenze der May-Lösungen ist der Punkt, an dem alle Aussagen als gültig erklärt werden. Der untere Grenze aller Must-Lösungen ist die leere Menge, es gelten keine Aussagen mehr. Sowohl alle Aussagen als gültig zu klassifizieren als auch keine Aussage zuzulassen, führt zu keinem Nutzen. Zwischen der oberen Grenzen (bzw. der unteren Grenze) unter der exakten Lösung liegen interessantere Bereiche für Approximationslösungen. Einzelne Lösungen sind Fixpunkte, bei denen alle Datenflußgleichungen erfüllt sind. Die Gesamtlösung ist stabil. Von mehreren möglichen Fixpunkten sind diejenigen die besten, die der exakten Lösungen so nahe wie möglich kommen. Für die May-Lösungen ist das der minimale Fixpunkt und für Must-Lösungen ist es der maximale Fixpunkt.

Zunächst ist jeder Fixpunkt eine Approximation der Lösung eines Datenflußproblems, denn für die o.g. Einschränkungen gilt: $\forall n \in N : MOP_{Must}(n) \supseteq MFP_{Must}(n)$ bzw. $\forall n \in N : MOP_{May}(n) \subseteq MFP_{May}(n)$. Da aber auch die Güte der Approximation – die Präzision – für die weitere Verwendung wichtig ist, werden maximale bzw. minimale Fixpunkte angestrebt, die von allen Fixpunkten die größte Menge an Information zusammentragen. Wenn über die bisherigen Voraussetzungen hinaus, alle Transferfunktionen noch distributiv sind, dann ist die MOP-Lösung in diesem speziellen Fall berechenbar und es gilt: $\forall n \in N : MFP(n) = MOP(n)$.

Im folgenden beschäftigt sich diese Arbeit nur noch mit MFP-Lösungen. Dazu soll zunächst für den Fall der Datenabhängigkeiten zwischen skalaren Variablen gezeigt werden, wie zu einer solchen Lösung gelangt werden kann.

3.3.3 Iterative Datenflußanalyse

Ein wichtiges und häufig verwendetes Verfahren ist die *iterative Datenflußanalyse*. Es stellt einen Algorithmus dar, der das Konzept der MFP-Lösungen für einen gegebenen CFG umsetzt.

Sei nun dieser CFG $FG = (N, E, s, e)$. Jedem Knoten n aus N werden zwei Werte $IN[n], OUT[n] \in L$ zugeordnet. Diese bezeichnen die Datenflußinformation, die den Knoten n erreicht bzw. verläßt. Die Datenflußgleichungen für einen Knoten n zur Bestimmung des *Must*-Information sehen dann so aus:

$$IN[n] = \begin{cases} \iota & \text{, falls } n = s \\ \bigsqcap_{m \in pred(n)} OUT[m] & \text{, sonst} \end{cases} \tag{3.5}$$

$$OUT[n] = f_n(IN[n]) \tag{3.6}$$

ι ist dabei wiederum ein geeigneter Startwert der Datenflußinformation. $IN[n]$ ergibt sich aus der größten gemeinsamen Menge (greatest lower bound) der Datenflußinformation vorhergehender Knoten. Im Eingangsbeispiel 3.4.1 konnte an dieser Stelle der Schnittmengenoperator zum Einsatz kommen. Die Anwendung der Transferfunktion auf Werte, die den Knoten erreichen, führt zu Werten $OUT[n]$, die den Knoten wieder verlassen.

Eine andere Darstellung ergibt sich, wenn nicht direkt auf die Transferfunktionen zur Berechnung von $OUT[n]$ zurückgegriffen wird, sondern wenn Mengen G und K definiert sind, die zu jedem Knoten n die erzeugten und vernichteten Eigenschaften enthalten:

$$OUT[n] = G[n] \cup (IN[n] - K[n]) \tag{3.7}$$

Der Worklist-Algorithmus 3.3.1 wählt einen Knoten aus der Worklist aus und berechnet $\sqcap$ über alle Vorgängerknoten. Anschließend wird die Transferfunktion angewendet, um die den Knoten verlassende Information zu bestimmen. Hat sich diese nicht verändert, so wird die Information auch nicht an Nachfolger propagiert und der Knoten braucht nicht wieder in die Worklist eingefügt zu werden. Ansonsten werden alle Nachfolger des veränderten Knotens in die Worklist gestellt. Anschließend kann eine weitere Auswahl eines Knotens zur Bearbeitung stattfinden. Diese Schleife wird solange durchlaufen bis die Worklist geleert ist. Dann ändern sich auch keine Werte mehr durch weitere Iterationen, es ist ein Fixpunkt erreicht.

Algorithmus 3.3.1 *Worklist-Algorithmus zur iterativen Datenflußanalyse*

procedure worklist_iterate($FG = (N, E, s, e), L, f, \iota$)

  für alle $n \in N$
    $IN[n] = \top$

  $IN[s] = \iota$
  $workset = \{n | (s, n) \in E\}$

  while $workset \neq \{\}$ *do*
    let $n \in workset$
      begin
        $workset = workset \setminus \{n\}$
        $IN[n] = \sqcap\{OUT[m] | \forall m \in pred(n)\}$
        $X = f_n(IN[n])$
        if $X \neq OUT[n]$ *then*
          $OUT[n] = X$
          $workset = workset \cup \{m | (n, m) \in E\}$
        fi
      end
  od

Die Eigenschaft, daß es sich bei der Lösung um einen *maximalen* Fixpunkt handelt, kann durch eine Überschätzung der Lösung während der Initialisierung sichergestellt werden. Die zu große Lösung wird durch die monotonen Transferfunktionen stetig verkleinert bis ein Fixpunkt erreicht wird. Dieser muß dann maximal sein[6].

Die Reihenfolge der Auswahl der Knoten n aus der Worklist ist bislang noch offen. Unter den vielen Möglichkeiten zur Anordnung der Knoten der Worklist, ist besonders *Reverse Postorder* zu nennen. Diese Reihenfolge gewährleistet, daß jeweils Knoten betrachtet werden, deren Vorgänger schon bearbeitet wurden. Somit sind die OUT-Werte der Vorgänger auch schon festgelegt. Die Anzahl der Iterationen wird gering gehalten, da keine unnötigen Veränderungen im nachhinein mehr vorkommen.

Bislang sind nur Datenflußprobleme betrachtet worden, deren Flußrichtung gleich der Richtung der Kontrollflusses ist. Wenn beispielsweise die Menge der an einem bestimmten Knoten lebendigen Variablen (siehe 3.3.4) bestimmt werden sollen, so ist eine Arbeitsrichtung entgegen der Kontrollflußrichtung angebracht. Die lebendigen Variablen sind solche, deren Definition erfolgte und die später auch noch benutzt werden. Ausgehend von einem (späteren) Gebrauch wird hin zu einer (früheren) Definition gearbeitet. *Rückwärtsprobleme (backward)* verlangen kleine Änderungen des Modells gegenüber den bisher betrachteten *Vorwärtsproblemen (forward)*. Zur Wahrung der Dualität fließt die Information vom Endnoten e zum Startknoten s und von $OUT[n]$ nach $IN[n]$. Die Datenflußgleichung aus 3.5 sieht dann entsprechend so aus:

$$OUT[n] = \begin{cases} \iota & \text{, falls } n = s \\ \displaystyle\bigsqcap_{m \in succ(n)} IN[m] & \text{, sonst} \end{cases} \tag{3.8}$$

$$IN[n] = f_n(OUT[n]) \tag{3.9}$$

3.3.4 Konkrete Datenflußprobleme

Nachdem der allgemeinen Beschreibung von Datenflußanalysen sollen nun konkrete Probleme und konkrete Analysen betrachtet werden.

Reaching Definitions: Eine Definition einer Variablen *erreicht* eine Knoten, wenn sie zwischenzeitlich nicht durch eine andere Definition vernichtet wurde. Die Analyse, die bestimmt, welche Definitionen einen Knoten erreichen können, heißt *Reaching Definitions*. Es handelt sich um ein May-Vorwärtsproblem, dessen Verband als Trägermenge einen Bitvektor verwendet, in dem für jede Definition ein Bit vorgesehen ist. Die Menge G

[6] Analog läßt sich der Algorithmus auch für *May-Information* formulieren. Ein *minimaler* Fixpunkt ergibt sich nach gleicher Argumentation.

der erzeugenden Referenzen umfaßt alle Definitionen, ebenso enthält die Menge K der vernichtenden Referenzen die Definitionen.

Available Expressions: Ein Ausdruck ist an einer Stelle in einem Block *verfügbar*, wenn entlang jedes Ausführungspfades vom Blockanfang dieser Ausdruck ausgewertet wird und danach keine an dem Ausdruck beteiligten Variablen redefiniert werden. *Available Expressions* sind die Menge der verfügbaren Ausdrücke an allen Knoten eines Blocks. Es handelt sich um ein Must-Vorwärtsproblem. Der Verband enthält für jeden Ausdruck ein Bit in einem Bitvektor. G und K sind jetzt Mengen von Ausdrücken, dementsprechend kommen als Operatoren der durch die Bitvektoren repräsentierten Mengen nun Mengenvereinigung und -durchschnitt in Frage.

Live Variables: Eine Variable heißt *lebendig* an einer Stelle in einem Programm, wenn auf einem Pfad von dieser Stelle zum Endknoten ein weiterer Gebrauch der Variablen liegt. *Live Variables* bestimmt zu allen Knoten die dort lebendigen Variablen. Dabei handelt es sich um ein May-Rückwärtsproblem, dessen Verband ein Bitvektor ist. Für jeden Gebrauch gibt es ein Bit im entsprechenden Vektor. G enthält bei diesem Problem alle Gebräuche von Variable, K alle Definitionen.

Je nach Größe des analysierten Programmabschnitts unterscheidet man zwischen

- *„Whole-program“-Analyse*, bei der das *gesamte* Programm inklusive aller globalen Datenstrukturen und aller Funktionen mit ihren evtl. wechselseitigen Aufrufen betrachtet werden, oder
- *interprozeduraler Analyse*, bei der Auswirkungen von Funktionsaufrufen auf die aufrufende Funktion berücksichtigt werden, oder
- *intraprozeduraler Analyse*, bei der das Verhalten einzelner Funktionen nur isoliert von anderen Funktionen analysiert werden, oder
- *Schleifenanalyse*, bei denen die analysierten Programmfragmente Schleifen sind, und
- *Basisblock-Analyse*, die innerhalb von Basisblöcken nach Abhängigkeiten sucht.

3.4 Skalare Optimierungen

Ziel der bislang diskutierten Bemühungen, das Programmverhalten zu analysieren, ist es, Verbesserungsmöglichkeiten für das analysierte Programm zu finden und auch anzubringen. Diese Verbesserungen können in unterschiedliche Richtungen abzielen, so z.B. die Ausführungsgeschwindigkeit erhöhen oder den Speicherbedarf vermindern. Je nach verfolgtem Ziel kommen unterschiedliche

Verbesserungen zum Einsatz, die auch unterschiedliche Informationen über das zu verbessernde Programm benötigen. Diese Informationen werden durch die Ergebnisse der vorherigen Analysen bereitgestellt, so daß es noch ihrer Interpretation bedarf, um sie auch nutzbringend in einer Codeverbesserung umzusetzen.

Beispiel 3.4.1 zeigt, wie gemeinsame Ausdrücke in verschiedenen Instruktionen erkannt und entfernt werden können.

Beispiel 3.4.1 *Common Subexpression Elimination (CSE)*

Zur Vermeidung einer redundanten Berechnung kann überprüft werden, ob bei einer Instruktion $t = x \otimes y$ *der Ausdruck* $x \otimes y$ *bereits* verfügbar *ist. Ist dies der Fall, kann die erneute Berechnung eliminiert und durch das zwischengespeicherte Resultat der vorherigen Berechnung ersetzt werden.*

Zur Durchführung der CSE sind Informationen über erreichende Ausdrücke (reaching expressions) *notwendig, die durch eine entsprechende Analyse bestimmt werden können. Ein Ausdruck wird erzeugt, wenn er ausgewertet wird, z.B. bei einer Zuweisung seines Resultats an eine Variable. Die Eigenschaft* reaching *bleibt so lange bestehen, wie keine an dem Ausdruck beteiligte Variable redefiniert wird, d.h. bei einer Redefinition wird die Eigenschaft* reaching *vernichtet. Mittels einer iterativen Datenflußanalyse kann für alle Knoten bestimmt werden, welche Ausdrücke dort die Eigenschaft* reaching *haben.*

Wird für einen Knoten festgestellt, daß ein in ihm berechneter Ausdruck dort auch die Eigenschaft reaching *hat, so kann die Elimination und Ersetzung vollzogen werden.*

Vorher:	***Nachher:***
$t = x \otimes y$	$h = x \otimes y$
$u = x \otimes y$	$t = h$
	$u = h$

Den meisten Codeverbesserungen liegt kein einheitliches formales Modell zugrunde, denn es gibt im Gegensatz zu den Datenflußanalysen keine gemeinsame theoretische Basis. Daher sollen im folgenden auch nur eine grobe Klassifikation möglicher Codeverbesserungen und ein Überblick möglicher wechselseitiger Beziehungen zwischen Analysen, Optimierungen und Zielarchitekturen gegeben werden.

3.4.1 Klassifikation der Codeverbesserungen

Rau [17] differenziert bei Verbesserungen zwischen *Optimierungen* und *Transformationen*. Optimierungen sind danach Compilertechniken, die redundante Operationen entfernen oder die Gesamtmenge der Berechnungen vermindern.

Transformationen hingegen sind Veränderungen der Berechnungsreihenfolge, die zum Ziel haben, zur Ausführung durch den Zielprozessor geeignetere Eigenschaften zu verstärken. Als beispielhafte Transformationen werden Verfahren genannt, die die Lokalität von Datenreferenzen vergrößern oder die Parallelität in äußeren[7] Schleifen erhöhen.

Nach dieser Definition beschäftigt sich die vorliegende Arbeit im wesentlichen mit Optimierungen. Viele der bekannten Optimierungen [1] befassen sich ausschließlich mit skalaren Variablen (*skalare Optimierungen)*, dabei werden Array-Elemente nicht berücksichtigt und der Zugriff auf sie nicht optimiert. In den folgenden Kapiteln soll daher gezeigt werden, wie auch für Arrays erfolgreich Optimierungen einzusetzen sind (*Array-Optimierungen*).

Wie schon bei der Unterscheidung von Optimierung und Transformation ist auch für die verschiedenen Optimierungen die Architektur des Systems, auf dem das optimierte Programm ausgeführt werden soll (Zielarchitektur), enorm wichtig. Ein häufig verwendetes Merkmal zur Unterscheidung von Optimierungen ist deren Maschinenabhängigkeit. Eine Optimierung ist *maschinenabhängig*, wenn die Effizienz ihrer Auswirkungen nur durch detaillierte Kenntnisse der Zielarchitektur abzuschätzen ist. Nach [5] ist die Unterscheidung zwischen *maschinenabhängig* und *maschinenunabhängig* jedoch nicht sinnvoll, da neben den eindeutig maschinenabhängigen Optimierungen (wie Registerallokation) auch viele scheinbar maschinenunabhängige Optimierungen (wie Constant Propagation) häufig stark von der Zielarchitektur in ihrer Leistungsfähigkeit beeinflußt werden. Stattdessen erscheint eine Unterteilung in *High-Level-* und *Low Level-*Optimierungen sinnvoller. High-Level-Optimierungen arbeiten auf einer nahezu hochsprachlichen Ebene, also mit einer großen Abstraktion der zugrundeliegenden Prozessorarchitektur, während Low-Level-Optimierungen zunehmend Details eines realen Instruktionssatzes (und der Zielarchitektur) berücksichtigen. Statt einer klaren Trennung dieser Klassen gehen sie fließend ineinander über.

Optimierungen, deren Ziel es ist, die Anzahl der Speicherzugriffe zu reduzieren, heißen *Speicherzugriffsoptimierungen*. Transformationen, die dazu dienen, Speicherzugriffe effizienter zu gestalten, z.B. durch eine bessere Nutzung einer Speicherhierarchie, sollen zur sprachlichen Vereinfachung mit unter diesen Begriff gefaßt werden.

3.4.2 Wechselbeziehungen

Nicht nur zwischen Optimierungen und Zielarchitektur gibt es starke Wechselbeziehungen, sondern auch zwischen Analysen und Optimierungen, sowie zwischen verschiedenen Optimierungen untereinander. Einzelne Optimierungsziele können miteinander einhergehen oder widersprechen.

[7] In Rau's Arbeit stehen Verfahren aus dem Bereich der Vektor- und Parallelrechner im Vordergrund. Parallelität in äußeren Schleifen ist für Signalprozessoren von untergeordneter Bedeutung.

Ein Beispiel für eine Abhängigkeit einer Optimierung von der Zielarchitektur ist die *Common Subexpression Elimination* aus Beispiel 3.4.1. Grundsätzlich erscheint es vorteilhaft, einen gemeinsamen Teilausdruck nur einmal zu berechnen und anschließend das zwischengespeicherten Resultat wiederzuverwenden. Wenn jedoch kein Register mehr frei ist, um das Zwischenergebnis dort zu speichern, muß der Wert in den Speicher geschrieben und von dort später wieder ausgelesen werden. Es kommt zum *Spilling*, was den gewünschten Effekt einer Beschleunigung der Programmausführung verringern oder ins Gegenteil verkehren kann. Ob es im konkreten Fall zum Spilling kommt, hängt vom optimierten Programm und der Anzahl der zur Verfügung stehenden Register des Prozessors ab. Weiterhin kann es vorkommen, daß der gemeinsame Teilausdruck durch die vorhandenen funktionalen Einheiten im Zielprozessor effizient ausgewertet werden kann, so daß die Wiederverwendung keinen Vorteil gegenüber der erneuten Auswertung erbringt.

Darüberhinaus hängt die Qualität einer Optimierung von der Präzision der vorherigen Analyse ab. Wenn die Analyse, deren Ergebnisse von der Optimierung interpretiert werden, nur eine grobe Approximation liefert, können unter Umständen viele Optimierungsmöglichkeiten nicht erkannt und ausgenutzt werden. Andersherum bedingt eine Erhöhung der Präzision einer Analyse nicht zwangsläufig eine höhere Qualität der Optimierung. Wenn z.B. anstelle von total redundanten Datenzugriffen durch eine in ihrer Präzision verbesserte Analyse auch partiell redundante Datenzugriffe erkannt werden, so kann eine Optimierung, die nur total redundante Zugriffe behandelt, nicht besser arbeiten. In diesem Falle wird eine Optimierung benötigt, die auch die Ergebnisse höherer Präzision in ihre Arbeit einbezieht, indem sie in der Lage ist, partiell redundante Zugriffe zu eliminieren.

Auch verschiedene Optimierungen beeinflussen sich untereinander, indem sie sich gegenseitig Möglichkeiten zur Anwendung schaffen oder nehmen. Nach einer *Copy Propagation* beispielsweise können sich Möglichkeiten zur *Dead Variable Elimination* ergeben, da nach der Copy Propagation Kopieroperationen verbleiben, deren Zielvariable im weiteren Verlauf nicht mehr benutzt wird. Diese Operationen können entfernt werden. Aber auch die Verhinderung von weiteren Optimierungen ist möglich. Durch die Copy Propagation können die Datenabhängigkeitsrelationen verändert werden, so daß sich evtl. Nachteile beim Instruction Scheduling ergeben.

Häufige Optimierungsziele sind Steigerung der Ausführungsgeschwindigkeit und Verringerung des Speicherplatzbedarfs. Durch verschiedene Optimierungen kann der Codeumfang verringert werden, was sich auch in einer erhöhten Ausführungsgeschwindigkeit bemerkbar machen kann. Andererseits kann Speicherplatz gespart werden, indem Zwischenergebnisse nicht gespeichert werden, sondern jedesmal von neuem berechnet werden. Das spart Platz, aber kostet Zeit.

Insgesamt werden in [5] folgende Beobachtungen beschrieben:

- Eine Optimierung beeinflußt nur einige Programme, auf die sie angewandt wird.

- Die Auswirkungen einer Optimierung auf verschiedene Programme sind verschieden groß.
- Die Summe der Auswirkungen einer Menge von Optimierungen ist häufig größer als die Summe der Einzeleffekte.

Das Resultat ist, daß keine allgemeinen Aussagen über die Qualität von Optimierungen getroffen werden können. Neben der Zielarchitektur sind andere Optimierungen und deren Ziele zu berücksichtigen. Selbst dann ergeben sich noch von Programm zu Programm andere mögliche Auswirkungen. Jedoch lassen sich Klassen bilden, die Vorteile und Nachteile in bestimmten Umgebungen zusammenfassen, von denen es zwar Ausnahmen gibt, die aber die häufigsten Erscheinungen widerspiegeln.

3.5 Static Single-Assignment (SSA) Form

Die bisherigen Abschnitte gehen von einer Zwischendarstellung (IR) aus, die eine feste Einheit von Variablen des Programms und Speicherzellen eines Maschinenmodells bilden. Jede Variable hat eine Adresse in einem Speicher und kann durch verschiedene Operationen dort adressiert werden.

Die *Static Single-Assignment Form* trennt die in einem Programm verwendeten Werte von den Speicherplätzen und ermöglicht dadurch teilweise effizientere Analysen und Optimierungen des Programmverhaltens. Die Darstellung in *Static Single-Assignment Form* verlangt, daß jede Variable im Quelltext des Programms maximal einmal definiert wird[8].Dadurch, daß nur eine Definition einer Variablen vorkommen kann, können auch unzusammenhängende Verwendungen der gleichen Variablen aufgelöst werden, da sie zu verschiedenen Variablen werden (z.B. gleiche Induktionsvariablen in zwei verschiedenen Schleifen).

Zur Transformation in SSA ist das Konzept der ϕ-Funktionen erforderlich, die an Stellen zusammenlaufenden Kontrollflusses *(Join Points)*, entsprechend des genommenen Pfades aus einer Menge von Variablen diejenige „passende" auswählt. Siehe dazu Beispiel 3.5.1.

Die SSA-Form mit ihren ϕ-Funktionen dient zur Abstrahierung von Programmen, sie ist *nicht* als Grundlage einer möglichen Ausführung des Programms gedacht. Daher besteht auch nicht die Notwendigkeit, die Implementierung einer ϕ-Funktion anzugeben. In vielen Fällen reicht die Kenntnis einer Verbindung von Definitionen und Gebräuchen aus, ohne genau wissen zu müssen, welcher Wert benutzt wird.

[8]Beachte: Die einmalige Definition einer Variablen im (statischen) Quelltext bedingt nicht, daß während der (dynamischen) Ausführung auch nur eine Zuweisung erfolgt. Befindet sich die (statische) Zuweisung beispielsweise in einer Schleife, so wird diese Zuweisung (dynamisch) durchaus mehrfach ausgeführt.

Beispiel 3.5.1 *Gegenüberstellung „konventionelle“ IR und SSA-Form*

„konventionelle“ IR	***SSA-Form***

`z = a;`
`if (z > 1)`
`   x = 1;`
`else`
`   x = 2;`
`z = x - 3;`
	$z_1 = a_1$;
	`if` $(z_1 > 1)$
	   $x_1 = 1$;
	`else`
	   $x_2 = 2$;
	$x_3 = \phi(x_1, x_2)$;
	$z_2 = x_3 - 3$;

Eine einfache SSA-Form ist recht einfach zu konstruieren, schwieriger ist die Erzeugung der minimalen SSA-Form eines Programms. Eine SSA-Form ist minimal, wenn die Anzahl der verwendeten ϕ-Funktionen minimal ist.

Weitergehende Darstellungen der SSA-Form, ihre effiziente Erzeugung und auf SSA bassierende Optimierungen finden sich in vielen Lehrbüchern, u.a. in [3] und [15].

Kapitel 4

Datenflußanalysen für Arrays

Während für skalare Variablen leistungsfähige Datenflußanalysen entwickelt wurden [1] und bei vielen Compilern als Grundlage vieler Optimierungen Verwendung finden, werden Array-Zugriffe in dieser Hinsicht häufig vernachlässigt. Das hat zur Folge, daß der so produzierte Code für Array-Referenzen eine recht „naive“ Qualität und eine geringe Effizienz hat. Möglichkeiten zur Redundanzelimination werden nicht genutzt. Ein unnötig hohes Aufkommen an Speicherbusverkehr und geringe Ausführungsgeschwindigkeiten der durch konventionelle Compiler erzeugten Executables verleiten oft den Entwickler von DSP-Applikationen zu einem recht „unsauberen“ Umgang mit Arrays. Solche Programme sind dann nicht mehr leicht zu verstehen und zu warten. Die in den folgenden Abschnitten vorgestellten Array-Datenflußanalysen sollen aufzeigen, welche Möglichkeiten zur Ermittlung von Datenabhängigkeiten zwischen Array-Elementen in Schleifen bestehen und welche Optimierungen aus deren Kenntnis ermöglicht werden.

Nachfolgend werden Unterschiede zu skalaren Datenflußanalysen aufgezeigt und die Grundlagen für Array-Datenflußanalysen geschaffen. Im Anschluß daran werden vier Analyse-Verfahren vorgestellt, die sich hinsichtlich ihres Anwendungsbereichs, ihrer Komplexität und Präzision unterscheiden. Von der δ-Technik, die die Methodik der skalaren Datenflußanalysen auf Arrays verallgemeinert, wird der Weg zum *Stretched-Loop*-Verfahren gezeigt. Dieses zeichnet sich durch eine erhöhte Präzision aus. Anschließend kommt mit der *Lazy*-Analyse ein Verfahren zur Diskussion, welches den Anwendungsbereich der Array-Datenflußanalyse auf nicht-affine Programmfragmente erweitert. Letztendlich wird ein leistungsfähiges, alternatives Verfahren beschrieben, daß auf einer *Dynamic-Single-Assignment*-IR arbeitet. Das Kapitel schließt mit einer Gegenüberstellung und Bewertung der vorgestellten Array-Datenflußanalysen.

4.1 Grundlagen zur Array-Datenflußanalyse

Skalare Datenflußanalysen lassen sich nicht auf Arrays anwenden. Eine Array-Referenz kann im Gegensatz zu einer skalaren Referenz eine Reihe verschiedener Speicheradressen bezeichnen. In Beispiel 4.1.1 stehen sich zwei Schleifen gegenüber, die sich dadurch unterscheiden, daß die erste eine Berechnung mit skalaren Variablen durchführt, und die zweite Schleife eine andere Berechnung mit Array-Elementen. In der ersten Schleife stehen die Variablen `a`, `b` und `c` für das gleiche Objekt. Es erfolgen mehrere Definitionen von `a` und `b`. Jedes Vorkommen einer skalaren Variable steht in eindeutiger Beziehung zu einer dem Datenobjekt zugeordneten Speicherzelle.

Beispiel 4.1.1 *Abhängigkeiten bei skalaren und indizierten Variablen*

Skalare Variablen

```
for(i = 0; i < N; i++)
{
   a = b + c;
   b = a * 3;
}
```

Indizierte Variablen

```
for(i = 0; i < N; i++)
{
   a[i] = b[i] + c[i];
   b[i] = a[i] * 3;
}
```

Die zweite Schleife unterscheidet sich auf den ersten Blick von der ersten dadurch, daß statt der skalaren Variablen nun Arrays mit dem Index `i` indiziert werden. Die eindeutige Zuordnung zwischen den textuellen Bezeichnung `a[i]`, `b[i]` und `c[i]` und entsprechen Speicherzellen geht verloren. Dagegen steht z.B. `a[i]` für eine Menge von Speicherzellen `a[1], ...,a[N]`. Welche Speicherzelle genau gemeint ist, hängt vom aktuellen Wert der *Induktionsvariablen* i ab.

Beispiel 4.1.2 *„Schweres“ Array-Datenabhängigkeitsproblem*

$$
\begin{array}{l}
\mathit{if}\ (n > 2)\{\\
\quad \mathit{if}\ (a > 0)\{\\
\quad\quad \mathit{if}\ (b > 0)\{\\
\quad\quad\quad \mathit{if}\ (c > 0)\{\\
\quad\quad\quad\quad x[a^n] = x[b^n + c^n] + 3\\
\quad\quad\quad \}\\
\quad\quad \}\\
\quad \}\\
\}
\end{array}
$$

Datenflußanalysen, die mit Arrays umgehen sollen, müssen diese Mehrdeutigkeiten von Array-Referenzen berücksichtigen. Dadurch wird die Datenflußanalyse gegenüber dem skalaren Fall komplizierter. Nicht immer können bei

Array-Referenzen die Abhängigkeitsrelationen bestimmt werden. Das allgemeine Problem der Bestimmung von Datenabhängigkeiten von Array-Referenzen ist unentscheidbar (siehe [14]). Dabei verhindern nicht nur statische Abhängigkeiten, wie sie z.B. durch Unbestimmtheiten aus Benutzereingaben auftreten, sondern auch dynamische Abhängigkeiten die algorithmische Behandlung im allgemeinen Fall. Das aus [14] entliehene Beispiel 4.1.2 verdeutlicht dies an einem schwierigen algebraischen Problem. Könnte für das Beispiel ein Verfahren angegeben werden, welches für alle Fälle die Datenabhängigkeiten bestimmt, so wäre Fermat's Vermutung bewiesen.

4.1.1 Problemeinschränkung vs. Approximationslösung

Um mit der Unentscheidbarkeit der Array-Datenflußanalyse umzugehen, bleiben zwei Möglichkeiten:

1. Einschränkung des Problems auf spezielle (leichtere) Problemklassen
2. Approximation statt exakter Lösung (siehe Kapitel 3.3.2)

Problemeinschränkung

Eine häufig verwendete Problemeinschränkung ist die Bestimmung *speicherbasierter* Abhängigkeiten statt *wertebasierter* Abhängigkeiten. Die bisher betrachteten Abhängigkeiten sind *wertebasierte* Abhängigkeiten bei denen es auf einen *Datenfluß* zwischen zwei abhängigen Instruktionen ankommt. Eine *speicherbasierte* Abhängigkeit liegt dann vor, wenn in Folge der gleiche Speicherplatz referenziert wird, dabei mindestens einmal schreibend, ohne daß eine Datenflußabhängigkeit vorliegen muß.

Beispiel 4.1.3 *Wertebasierte vs. speicherbasierte Abhängigkeit*

Wertebasierte Abhängigkeit	***Speicherbasierte Abhängigkeit***

`a = x + y;`
`...`
`z = a * 3;`
	`a = x + y;`
	`...`
	`x = b + 1;`
	`...`
	`x = c * 3;`

In Beispiel 4.1.3 wird der Unterschied zwischen beiden Typen der Abhängigkeit verdeutlicht. Im ersten Programmfragment besteht eine wertebasierte Abhängigkeit und eine speicherbasierte Abhängigkeit zwischen den beiden Instruktionen. Die Variable `a` wird in beiden Operationen referenziert und in einer der beiden geschrieben, damit ist die speicherbasierte Abhängigkeit gegeben. Da zwischen den beiden Instruktionen bzgl. `a` auch eine Datenflußabhängigkeit vorliegt, besteht darüberhinaus eine wertebasierte Abhängigkeit. Im zweiten Programmfragment wird der Wert von `x` erst gelesen, und dann zweimal neu geschrieben. Zwischen `a = x + y` und `x = b + 1` bzw. `x = c * 3` liegen speicherbasierte Antiabhängigkeiten vor, und zwischen `x = b + 1` und `x = c * 3` besteht eine Ausgabeabhängigkeit. Es bestehen keine wertebasierten Abhängigkeiten, da es keinen Datenfluß zwischen den drei Instruktionen gibt. Da aber die Variable `x` in Folge referenziert und dabei mind. einmal geschrieben wird, sind die Instruktionen `a = x + y` und `x = c * 3` speicherbasiert abhängig.

Maslov [12] definiert wertebasierte Abhängigkeiten mit Hilfe von speicherbasierten Abhängigkeiten:

> „Intuitively, a value-based dependence exists between two statement instances if there exists memory-based dependence between them and value written in the first statement instance is actually used in the second instance, that is, the memory cell written in the first statement instance is not overwritten before the second instance occurs."

Speicherbasierte und wertebasierte Abhängigkeiten lassen sich auch formal unterscheiden. Darauf soll aber an dieser Stelle verzichtet werden, weil für den Rest dieser Arbeit das Konzept zum Verständnis ausreicht. Ein formale Darstellung findet sich z.B. in [13]. Die Bestimmung von speicherbasierten Abhängigkeiten wird in der Literatur als *Memory Disambiguation* bezeichnet.

Die durch Memory Disambiguation eingeführte Einschränkung des behandelten Problems ist an dieser Stelle unzureichend. Zum einen ist Memory Disambiguation nur mit weiteren Einschränkungen berechenbar, und zum anderen ist die verwendete Form der speicherbasierten Abhängigkeit zur weiteren Verwendung bei Optimierungen für ILP-Architekturen wegen ihrer zu unpräzisen Information ungeeignet. Die Information, daß z.B. zwei Referenzen den gleichen Speicherplatz adressieren, genügt nicht zur Entscheidung, ob es sinnvoll ist, den Wert der ersten Referenz zur späteren Wiederverwendung in einem Register zu halten.

Eine anderer Möglichkeit zur Einschränkung des Array-Datenflußproblems besteht in der Reduktion der „Schwierigkeit" der erlaubten Ausdrücke. Eine Array-Referenz $X[f(i)]$ setzt sich aus dem Array X und der *Indexfunktion* $f(i)$ zusammen. Wenn sowohl die zulässigen Indexfunktionen wie auch Ausdrücke in Verzweigungsbedingungen etc. eingeschränkt werden, kann dadurch die Komplexität des Datenflußproblems verringert werden. Eine berechenbare und oft verwendete Klasse von Ausdrücken umfaßt die Menge der *affinen* Funktionen.

Definition 4.1.1 *Eine lineare Funktion* $f : \mathbf{N} \to \mathbf{N}$, $f(i) \mapsto a \times i + b$ *mit einer Basisinduktionsvariablen* i *und (symbolischen) Konstanten* a, b *heißt* affine Funktion.

Die ausschließliche Verwendung affiner Ausdrücke ermöglicht den Einsatz effizienter Verfahren zur exakten Lösung des so eingeschränkten Array-Datenflußproblems. Für Verfahren, die über den Bereich der affinen Ausdrücke hinausgehen, steigt deren Komplexität sehr schnell an.

Approximationslösung

Ein anderer Ansatz neben der Problemeinschränkung besteht in der Bestimmung von Approximationslösungen anstatt der exakten Datenflußrelationen. Die in Kapitel 3.3.2 getroffenen Aussagen zur Sicherheit von Approximationen gelten in analoger Form auch für Datenabhängigkeitsanalysen für Arrays. Wie auch bei den skalaren Abhängigkeitsanalysen ist es wünschenswert, eine möglichst hohe Präzision der Approximation zu erzielen, und andererseits keine Fehler zu erlauben, die als „unsicher" anzusehen sind.

Neben der Klassifikation zweier Referenzen als abhängig oder unabhängig kann noch eine dritte Möglichkeit hinzugenommen werden: „keine Lösung". Das bedeutet, daß ein angewendetes Verfahren nicht entscheiden kann, wie die Lösung aussieht. Es liegt dann an der folgenden Optimierung mit diesem Ergebnis so umzugehen, daß die semantische Korrektheit erhalten bleibt.

Problemeinschränkung und Approximationslösung

Die beiden Wege im Umgang mit dem unentscheidbaren Array-Datenflußproblem brauchen nicht isoliert voneinander gegangen zu werden. Problemeinschränkung und Erzeugung einer Approximationslösung sind auch zusammen sinnvoll. Möglich ist beispielsweise ein Verfahren, welches auf affinen Ausdrücken exakte Abhängigkeiten bestimmt und bei Referenzen mit nicht-affinen Indexfunktionen alle weiteren Referenzen auf dasselbe Array als davon abhängig abschätzt.

In [14] wird neben der Schwäche der Memory Disambiguation gezeigt, welche Arten von Fehlern bei solchen Arten der Approximation entstehen und wie deren Verteilung für eine Reihe von Benchmarks aussieht.

4.1.2 Begriffe

Datenabhängigkeiten von Array-Referenzen in Schleifen lassen sich sowohl nach der Richtung der Abhängigkeit als auch nach den beteiligten Iterationen einteilen.

In Abbildung 4.1,a) sind zunächst Array-Datenabhängigkeiten nach ihrer Richtung unterschieden. Dabei gibt es die gleichen Möglichkeiten zwischen true/anti und output dependence wie bei den skalaren Datentypen in Kapitel 3.2.

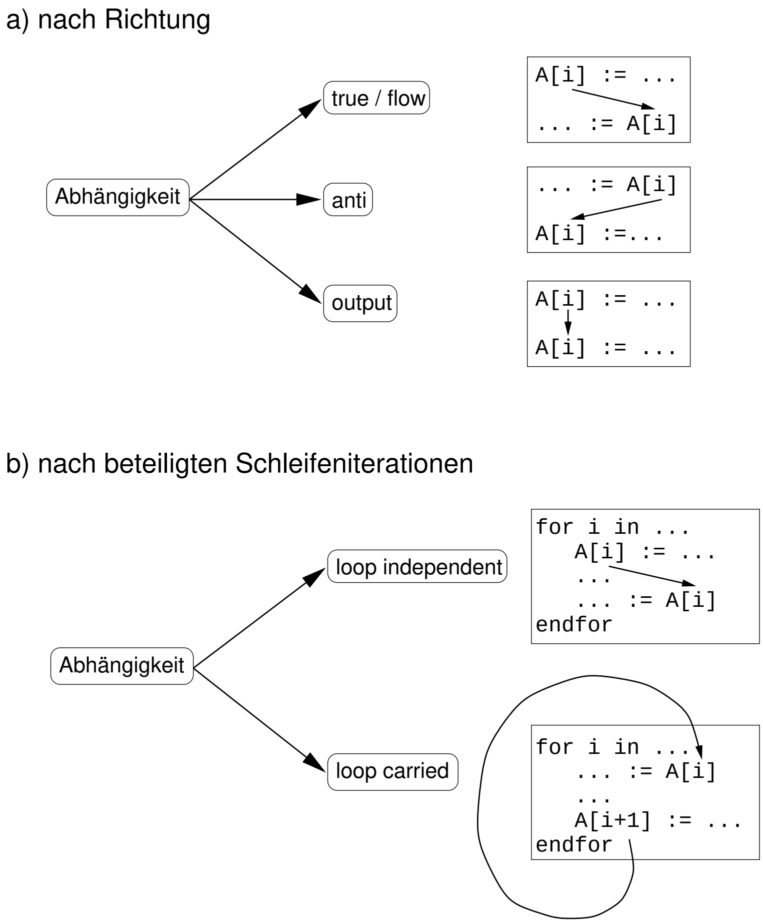


Abbildung 4.1: Typen von Datenabhängigkeiten bei Array-Referenzen in Schleifen

Für Schleifen kann man Datenabhängigkeiten im Schleifenkörper als *schleifenunabhängig (loop independent)* oder als *schleifenabhängig (loop carried)* unterscheiden. Schleifenunabhängige Abhängigkeiten treten ohne Zusammenhang mit einer Veränderung der Induktionsvariablen auf. In Abbildung 4.1,b) bedeutet dies, daß die Abhängigkeit zwischen den zwei Instruktionen im Schleifenkörper bzgl. `a[i]` auch dann existieren würde, wenn die Instruktionen nicht von einer Schleife umgeben wären und in einer linearen Ausführungsreihenfolge abgearbeitet würden. Neu hinzu kommt für Schleifen die Möglichkeit zur Ausbildung einer Datenabhängigkeit im Schleifenkörper, die von der umgebenden Schleife abhängig ist. Diese Abhängigkeiten würden nicht bestehen, wenn der Schleifenkörper nicht in einer Schleife ausgeführt würde. Im Beispiel besteht eine Datenflußabhängigkeit zwischen `a[i+1]` und `a[i]`, die nur deshalb zustande kommt, weil die Induktionsvariable i beim Übergang in die nächste Iteration inkrementiert wird.

Es sind einige Abstraktionen zur Darstellung von Array-Datenabhängigkeiten entwickelt worden. Einige wichtige Abhängigkeitsdarstellungen[1] sind *Distanzvektoren*, *Richtungsvektoren* und *exakte Datenflußabhängigkeitspaare*.

Distanzvektoren erfassen Datenabhängigkeiten durch ihre Distanzen im Iterationsraum der umgebenden Schleife, d.h. es werden die Differenzen der Induktionsvariablen zu den Zeitpunkten der Definition und des Gebrauchs gebildet. *Richtungsvektoren* sind eine Vereinfachung von Distanzvektoren, die nur noch das Vorzeichen der Distanz der Datenabhängigkeit widerspiegeln. Sie geben an, ob die Induktionsvariablen zwischen Definition und Gebrauch vergrößert oder verkleinert wurden. Die *exakten Datenflußabhängigkeitspaare* bieten die größte Präzision der Darstellung, denn sie fassen für eine Schleife die Induktionsvariablen einer Definition und eines abhängigen Gebrauchs zusammen. Für das Beispiel 4.1.4 (aus [13]) werden die verschiedenen Darstellungen gegenübergestellt.

Beispiel 4.1.4 *Programmschleife mit Datenabhängigkeit*

```
for(i = 0; i <= L; i++)
   for(j = 0; j <= M; j++)
      for(k = 0; k <= N; k++)
         a[j] = a[j] + ...
```

Datenfluß-Richtungsvektoren:

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} : \left\{ \begin{bmatrix} 0 \\ 0 \\ + \end{bmatrix} , \begin{bmatrix} + \\ 0 \\ - \end{bmatrix} \right\} \tag{4.1}$$

[1] Eine ausführlichere Liste befindet sich in [13].

Datenfluß-Distanzvektoren:

$$\begin{bmatrix} i \\ j \\ k \end{bmatrix} : \left\{ \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ -N \end{bmatrix} \right\} \tag{4.2}$$

Exakte Datenflußabhängigkeitspaare:

$$\begin{aligned} &\left\{ \left(\begin{bmatrix} i \\ j \\ k-1 \end{bmatrix}, \begin{bmatrix} i \\ j \\ k \end{bmatrix} \right) \middle| \begin{array}{l} 0 \leq i \leq L \\ 0 \leq j \leq M \\ 1 \leq k \leq N \end{array} \right\} \\ \bigcup &\left\{ \left(\begin{bmatrix} i-1 \\ j \\ N \end{bmatrix}, \begin{bmatrix} i \\ j \\ 0 \end{bmatrix} \right) \middle| \begin{array}{l} 1 \leq i \leq L \\ 0 \leq j \leq M \end{array} \right\} \end{aligned} \tag{4.3}$$

Im Beispiel ist eine Schleifenschachtelung bestehend aus drei Schleifen zu sehen. Es bestehen zwei Datenflußabhängigkeiten, die durch Datenfluß-Richtungsvektoren, Datenfluß-Distanzvektoren und durch die exakten Datenflußabhängigkeitspaare dargestellt werden. Der Schleifenkörper der inneren Schleife enthält zwei Array-Referenzen `a[j]`. Beim Übergang in eine neue Iteration der inneren Schleife wird der Wert von `a[j]` gelesen, der in der vorherigen Iteration geschrieben wurde. Somit ist die Datenflußabhängigkeit bzgl. `k` von niedrigen zu höheren Iterationen gerichtet und hat die Distanz 1, da zwischen Definition und Gebrauch eine Iterationsgrenze überschritten wird. Durch den Richtungsvektor $(0, 0, +)$ bzw. durch den Distanzvektor $(0, 0, 1)$ kann die Abhängigkeit bzgl. der inneren Induktionsvariablen `k` ausgedrückt werden. In der exakten Darstellung werden Tupel angegeben, deren erste Komponente die Induktionsvariablen bei der Definition umfassen, und deren zweite Komponente den Gebrauch anzeigt. Die zweite Datenflußabhängigkeit des Beispiels ergibt sich durch die innere und äußere Schleife. Wenn die innere Schleife ihr Schleifenende erreicht und `k` von N auf 0 „zurückspringt", wird `j` der mittleren Schleife erhöht. Wenn später `i` erhöht wird, wird für jedes `j` ein Wert gelesen, der am Ende des Iterationsbereichs von `k` der vorherigen Iteration von `i` geschrieben wurde. Für `i` ist die Abhängigkeit vorwärtsgerichtet, denn `i` wurde zwischenzeitlich erhöht. `k` aber hat beim Gebrauch einen kleineren Wert als bei der Definition, daher ist bzgl. `k` die Abhängigkeit rückwärts gerichtet. Die Distanzen der Iterationsvariablen betragen zwischen Definition und Gebrauch $(1, 0, -N)$. Die exakte Darstellung gibt wiederum in allgemeiner Form die Beziehung zwischen den Induktionsvariablen zum Zeitpunkt der Definition und des Gebrauchs an.

4.2 δ-Verfahren zur Array-Datenflußanalyse

Die von Duesterwald et al. [8] vorgestellte Methode zur datenflußbasierten Analyse von Abhängigkeiten zwischen Array-Elementen ist in der Lage, verschiedene Optimierungen zu unterstützen. Dazu gehören z.B. Load/Store-Optimierungen, Loop Unrolling oder die Registerallokation bei sequentiellen oder feinkörnig-parallelen Prozessorarchitekturen.

Das Verfahren zur Analyse setzt strukturierte Schleifen (siehe Kapitel 3) und affine Indexfunktionen (siehe Kapitel 4.1) voraus. Schleifen, die diesen Voraussetzungen nicht genügen, können nicht analysiert werden. Die Einschränkung auf die Affinität ist im Bereich der DSP-Applikationen häufig nicht allzu einengend, da nach [4] und auch nach eigenen Erfahrungen (siehe Kapitel 8) sehr oft lineare Indexfunktionen Verwendung finden.

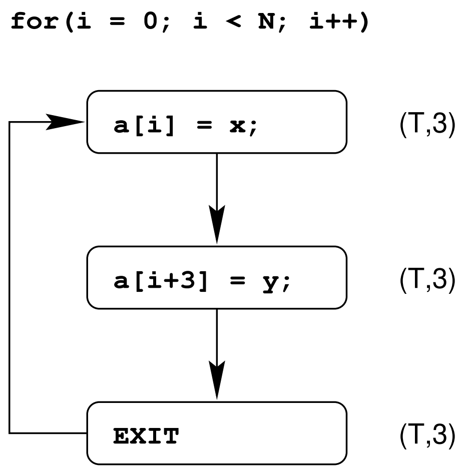
Ein Vorteil des Verfahrens ist seine Parametrisierbarkeit. Es ist möglich, sowohl Vorwärts- als auch Rückwärts-Analysen durchzuführen. Zu den Vorwärtsanalysen gehört z.B. die *Reaching-definitions*-Analyse, eine Rückwärtsanalyse ist die *Live-variable*-Analyse (siehe Kapitel 3.3.4). Weiterhin besteht die Auswahl zwischen der Berechnung von *Must*- und *May*-Information. Die Festlegung der Klasse (Vorwärts, Rückwärts, Must, May) der tatsächlich durchzuführenden Analyse erfolgt vorab durch die Definition einiger Funktionen und des Aussehens des verwendeten Verbandes, während hingegen der Typ der Analyse innerhalb einer Klasse (Reaching-definitions, δ-Redundanz, etc.) durch Parametrisierung mit geeigneten Transferfunktionen geschieht. Die Anwendbarkeit der Analysen ist nicht auf eindimensionale Felder beschränkt, sondern läßt sich durch Betrachtung äußerer Induktionsvariablen als symbolische Konstanten in inneren Schleifen auch auf mehrdimensionale Felder erweitern.

Ein weiterer Vorteil ist die geringe Komplexität des Verfahrens. Die Implementierung ist nicht allzu schwierig und die benötigten Ressourcen zur Laufzeit halten sich in geringen Grenzen. Nachteilig gegenüber aufwendigeren Verfahren ist die etwas geringere Präzision mit der Datenabhängigkeiten erkannt und dargestellt werden.

Kern des Verfahrens sind Berechnungen von maximalen *Iterationsdistanzwerten* zu allen Knoten eines *Schleifenkontrollflußgraphen*. Eine Iterationsdistanz bezeichnet die Anzahl an Iterationen über deren Dauer die Lösung an einem Knoten Gültigkeit behält. Damit entspricht sie etwa dem Distanzvektor aus Kapitel 4.1.2. Der Schleifenkontrollflußgraph entsteht durch Extraktion des Bereichs der Schleife aus dem Kontrollflußgraphen des gesamten Programms und Einfügen eines Knoten zur expliziten Darstellung von Iterationsübergängen.

Beispiel 4.2.1 zeigt den Schleifenkontrollflußgraphen einer kleinen Programmschleife. Die Knoten sind mit Iterationsdistanzen markiert, die anzeigen, welche Definitionen mit welcher Iterationsdistanz den Knoten erreichen. Der erste Wert steht für die erste Definition `a[i]`, und entsprechend der zweite Wert für die zweite Definition `a[i+3]`. Die zweite Definition erreicht die übrigen Knoten mit der Distanz 3, da die erste Definition in diesem Abstand Werte überschreibt. Die Werte der ersten Definition dagegen bleiben immer erhalten, was durch $\top$ gekennzeichnet wird.

Beispiel 4.2.1 *Schleifenkontrollflußgraph und Iterationsdistanzen*



Schleifenkontrollflußgraph

Zur Berechnungen von Iterationsdistanzen wird bei der δ-Datenflußanalyse der binäre Verband der skalaren Datenflußanalyse zu einem mehrwertigen, linearen Verband verallgemeinert. Die hinzugenommenen Werte repräsentieren die Gültigkeitsdauer von Datenflußeigenschaften. $\perp$ steht weiterhin für Ungültigkeit einer Eigenschaft und $\top$ kennzeichnet deren Gültigkeit über alle Iterationen.

Das δ-Verfahren geht bei der Bestimmung des Datenflusses in ähnlicher Weise vor wie das iterative Datenflußverfahren aus Kapitel 3. Einzelne Knoten des Schleifenkontrollflußgraphen enthalten Instruktionen der Schleife. Die Beeinflussung von Daten durch einzelne Instruktionen wird mit Transferfunktionen modelliert. Die Transferfunktionen müssen nun so ausgelegt sein, daß sie mit verschiedenen Bezeichnungen für ein Array-Element oder gleichen Bezeichnungen für verschiedene Array-Elemente klarkommen. Beispielsweise können `a[i]` in der Iteration $i = 7$ und `a[i+3]` in der Iteration $i = 4$ das gleiche Element `a[7]` bezeichnen. Andererseits kann `a[i]` in den Iterationen $i = 7$ und $i = 4$ verschiedene Elemente `a[7]` und `a[4]` meinen. Wenn die Transferfunktionen bestimmt sind, kann ein iteratives Verfahren eingesetzt werden, welches einen Fixpunkt bestimmt. Dieser Fixpunkt ist eine Lösung des Datenflußgleichungssystems, aus dem sich die gesuchten Iterationsdistanzen ablesen lassen.

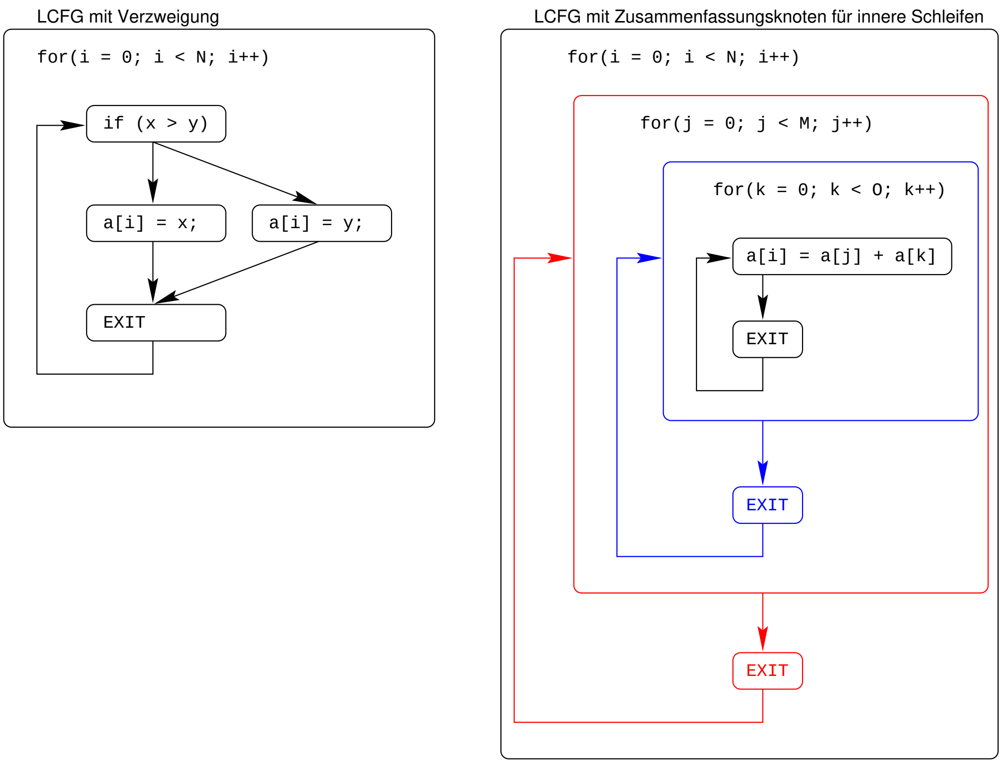
Im einzelnen werden in den folgenden Abschnitte die genannten Komponenten des δ-Verfahrens genauer vorgestellt. Ausgehend von dem Schleifenkontrollflußgraphen und dem Datenflußverband wird die Konstruktion der Transferfunktionen erläutert. Anschließend wird das Datenflußgleichungssystem und seine Lösung vorgestellt. Letztlich wird die oben erwähnte Parametrisierung für andere Analysen gezeigt, bevor erweiterte Möglichkeiten wie die Behandlung mehrdimensionaler Arrays und Loop Nests vorgestellt werden.

4.2.1 Schleifenkontrollflußgraph

Die Array-Datenflußanalyse arbeitet auf einem *Schleifenkontrollflußgraphen*, der gegenüber einem gewöhnlichem Kontrollflußgraphen einige Erweiterungen und Besonderheiten besitzt:

Definition 4.2.1 *Ein* Schleifenkontrollflußgraph (Loop Control Flow Graph, LCFG) *$FG = (N, E)$ mit der Knotenmenge N und der Kantenmenge E repräsentiert den Kontrollfluß innerhalb eines Schleifenkörpers. Die Knoten in N bezeichnen dabei Instruktionen im Schleifenkörper, die Kanten in E sind mögliche Kontrollflußübergänge. Im Unterschied zu einem normalen Kontrollflußgraphen gibt es außerdem einen ausgezeichneter Knoten* Exit*, der explizit den Übergang zur nächsten Iteration modelliert. Es führen von allen Knoten, die den Schleifenkörper beenden, Kanten zu* Exit*; von* Exit *gehen Kanten zu allen Knoten am Anfang des Schleifenkörpers. Neben gewöhnlichen Knoten gibt es* Zusammenfassungsknoten*. Innere Schleifen eines Loop Nests werden bei der Konstruktion des LCFG einer umgebenden Schleife zu einem Zusammenfassungsknoten kontrahiert. Deren Verhalten durch ihnen zugeordnete Informationen beschreiben.*

Beispiel 4.2.2 *Verschiedene Schleifenkontrollflußgraphen*



Die Definition hat zur Folge, daß es keine LCFG gibt, die verschachtelte Zyklen enthalten, denn innere Schleifen werden bei der Behandlung umgebender

Schleifen durch Zusammenfassungsknoten repräsentiert.

Das Beispiel 4.2.2 zeigt auf der linken Seite einen LCFG mit einer Verzweigung im Schleifenkörper. Von jedem Ast der Verzweigung führt eine Kante zum Exit-Knoten, da vom Ende jedes Verzweigungsastes die Iteration beendet werden kann. In der rechten Hälfte ist der LCFG eines Loop Nests zu sehen. Innere Schleifen werden zu bei der Analyse äußerer Schleifen zu einzelnen Zusammenfassungsknoten – hier blau und rot dargestellt – zusammengezogen.

4.2.2 Verwendeter Datenflußverband und Operatoren

Die betrachteten Eigenschaften bei der vorliegenden Analyse sind die maximalen Iterationsdistanzen. Diese Iterationsdistanzen – bezeichnet durch δ – werden durch den verwendeten Datenflußverband mathematisch umgesetzt. Anhand von *reaching definitions* kann die (maximale) Iterationsdistanz wie folgt definiert werden:

Definition 4.2.2 *Eine Definition d* erreicht *eine Stelle p mit der* Iterationsdistanz *δ, falls die letzten δ Instanzen von d p erreichen. Die Iterationsdistanz δ ist* maximal*, falls δ der größte Wert der Iterationsdistanz ist, für den die Definition d die Stelle p erreicht.*

Während der Analyse wird jeder Referenz ein Verbandselement zugeordnet, welches die maximale Iterationsdistanz, also die maximale Distanz der Gültigkeit der Lösung, angibt. Dazu wird der verwendete Verband dahingehend gegenüber dem bei der Behandlung skalarer Probleme eingesetzten (siehe Kapitel 3.3.1) erweitert, daß er in der Lage ist, auch Zwischenwerte zwischen $\perp$ und $\top$ zu repräsentieren. Daraus resultiert für den Verband die Wahl des Intervalls $[\perp, 0, 1, \ldots, \top = UB - 1]$ mit der oberen Iterationsgrenze UB als Trägermenge, sowie der linearen, streng monoton steigenden Ketten-Anordnung der Elemente dieser Menge als partielle Ordnung $\leq$, siehe dazu das Hasse-Diagramm in Abb. 4.2. Supremum und Infimum werden durch $\top$ bzw. $\perp$ eindeutig bestimmt. $\top = UB - 1$ entspricht der Gültigkeit der so bezeichneten Lösung über alle Iterationen hinweg, denn eine Eigenschaft kann frühestens in der ersten Iteration Gültigkeit erlangen und sie dann maximal über $UB - 1$ weitere Iterationen behalten. $\perp$ bezeichnet die Ungültigkeit, d.h. die Gültigkeit über *keine* Iteration.

Für die gewählte Menge fehlen noch die Operatoren. Die Operatoren müssen das Aufeinandertreffen verschiedener Kontrollflußpfade (meet) oder die Veränderung der Iterationsdistanz modellieren. Die Operatoren $\wedge$ und $\vee$ des Verbandes werden durch die Minimumfunktion *min* bzw. die Maximumfunktion *max* realisiert. Der Datenflußverband hat somit folgende Gestalt:

$$L = ([\perp, 0, \ldots, \top = UB - 1], \perp, \top, \sqsubseteq, \wedge, \vee) \tag{4.4}$$

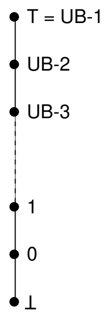


Abbildung 4.2: Hasse-Diagramm des Verbandes mit der zugehörigen partiellen Ordnung

mit

$$\sqsubseteq: \forall x_i \in [\bot, 0, \ldots, \top = UB-1] : x_i < x_{i+1} \tag{4.5}$$

und

$$\wedge(x,y) = \begin{cases} \bot & \text{, falls } x = \bot \text{ oder } y = \bot \\ x & \text{, falls } y = \top \\ y & \text{, falls } x = \top \\ min(x,y) & \text{, sonst} \end{cases} \tag{4.6}$$

$$\vee(x,y) = \begin{cases} \top & \text{, falls } x = \top \text{ oder } y = \top \\ x & \text{, falls } y = \bot \\ y & \text{, falls } x = \bot \\ max(x,y) & \text{, sonst} \end{cases} \tag{4.7}$$

Zu diesem Verband kommt noch eine weitere Operation hinzu, die später dazu dienen wird, den Übergang von einer Iteration in die nächste zu vollziehen. Es handelt sich dabei um einen speziellen Inkrement-Operator, der für alle Elemente aus L definiert ist:

$$x{+}{+} = \begin{cases} \top & \text{, falls } x = \top \\ \bot & \text{, falls } x = \bot \\ x+1 & \text{, sonst} \end{cases} \tag{4.8}$$

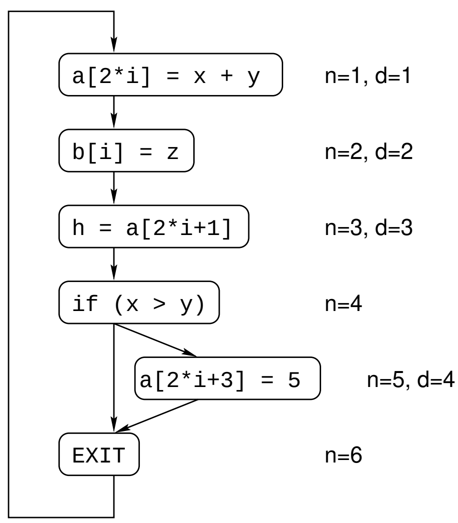
4.2.3 Transferfunktionen

Transferfunktionen abstrahieren das Verhalten von Instruktionen. Einzelne Instruktionen können die Gültigkeit einer Datenflußeigenschaft erzeugen oder vernichten, dementsprechend sind den Knoten des LCFG Mengen erzeugender und vernichtender Array-Referenzen zugeordnet. Mit diesen lassen sich die Transferfunktionen konstruieren. Nicht alle Transferfunktionen haben die gleiche Gestalt, sondern es gibt unterschiedliche Klassen. Die Klassen unterscheiden sich darin, ob eine Iterationsdistanz erstmalig erzeugt oder erhalten wird sowie nach dem Iterationsübergang. Bei der Erhaltung kommt es darauf an, den Iterationsdistanzwert zu überprüfen, denn es ist möglich, daß sich dieser durch die aktuelle Instruktion verändert.

Für jeden Knoten n eines LCFG gibt es eine Transferfunktion f_n, die das Verhalten einer Instruktion bezüglich eines speziellen Datenflußproblems modelliert. Für Knoten n des LCFG sind in der Menge $G[n]$ diejenigen Array-Referenzen enthalten, die als *erzeugende* Instanzen dienen, hingegen enthält die Menge $K[n]$ die *vernichtenden* Instanzen. Die Festlegung, welche Art von Referenz als erzeugend oder vernichtend anzusehen ist, hängt von der konkreten Datenflußanalyse ab. Diese Festlegung ist ein Parameter der Analyse. Beispiel 4.2.3 erläutert das Zustandekommen der Mengen $G[n]$ und $K[n]$ für das Datenflußproblem *Must-Reaching-Definitions*.

Beispiel 4.2.3 *Must-Reaching-Definitions*

Die Analyse Must-Reaching-Definitions *bestimmt, welche Definitionen einen Knoten entlang aller Kontrollflußpfade erreichen müssen. Für diese Eigenschaft werden Definitionen als erzeugend angesehen, denn erst nach einer Definition kann diese andere Knoten erreichen. Eine Definition kann nicht durch einen Gebrauch, sondern nur durch eine andere Definition vernichtet werden. Somit enthalten* $G[n]$ *und* $K[n]$ *jeweils alle Definitionen von Array-Variablen am Knoten* n*. Gibt es* m *erzeugende Array-Referenzen, so gibt es pro Knoten auch* m *eintreffende Verbandselemente, die von der Transferfunktion verarbeitet werden müssen. Für die Schleife*



ergeben sich G *bzw.* K *für* Must-Reaching-Definitions *zu* $G[1] = \{a[2 \times i]\}$, $K[1] = \{a[2 \times i]\}$, $G[2] = \{b[i]\}$, $K[2] = \{b[i]\}$, $G[3] = \{a[2 \times i + 1]\}$, $K[3] = \{a[2 \times i + 1]\}$, $G[4] = \{\}$, $K[4] = \{\}$, $G[5] = \{a[2 \times i + 3]\}$, $K[5] = \{a[2 \times i + 3]\}$, $G[6] = \{\}$, $K[6] = \{\}$. *Es gilt* $m = 4$.

Durch Vorgabe der $G[n]$ und $K[n]$, d.h. durch eine Programmschleife und die Information, wie bestimmte Typen von Referenzen den Datenfluß beeinflussen, können die Transferfunktionen eindeutig bestimmt werden. Wenn es m erzeugende Referenzen gibt, formal $|G| = m$, so hat f_n die Gestalt $f_n : L^m \to L^m$. Es werden also Tupel von Iterationsdistanzen durch die Transferfunktionen berechnet und entsprechend dem Kontrollfluß durch den LCFG FG propagiert. Die auf m Verbandselementen operierende Funktion f_n behandelt jede Komponente aus L separat und unabhängig voneinander, so daß sie sich aus m unären Funktionen f_n^d, $\forall d \in [1, \ldots, m]$ als kartesisches Produkt zusammensetzen läßt:

$$\forall (x_1, \ldots, x_m) \in L^m : f_n(x_1, \ldots, x_m) = (f_n^1(x_1), \ldots, f_n^m(x_m)) \tag{4.9}$$

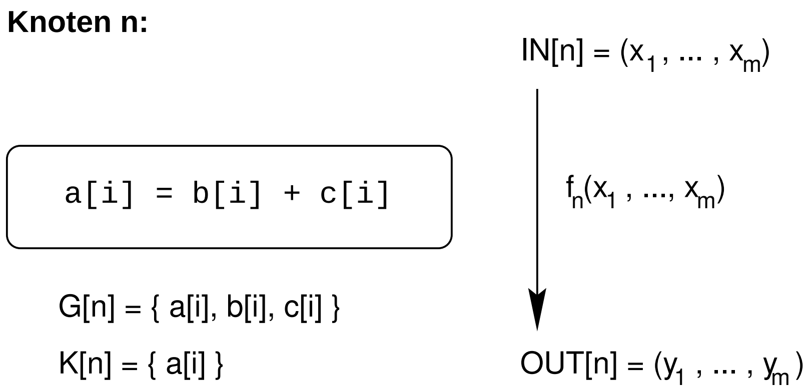


Abbildung 4.3: Einzelner Knoten eines LCFG mit zugehörigen Komponenten

In Abbildung 4.3 wird ein einzelner Knoten eines LCFG mit den zugehörigen Komponenten dargestellt. Die Mengen $G[n]$ und $K[n]$ enthalten die erzeugenden und vernichtenden Referenzen der Knoten. $IN[n]$ und $OUT[n]$ sind Datenflußinformationen, die den Knoten erreichen bzw. verlassen. Die Transferfunktion $f_n(x_1, \ldots, x_m)$ ist aus m einzelnen Funktionen zusammengesetzt. Jedes Argument der Transferfunktion wird durch eine separate Funktion behandelt.

Die einzelnen Funktionen f_n^d werden unter Berücksichtigung von $G[n]$ und $K[n]$ gebildet, wobei drei verschiedene Funktionstypen möglich sind:

- Erzeugungsfunktionen,
- Erhaltungsfunktionen, und
- Exit-Funktionen.

Erzeugungsfunktionen modellieren den Fall einer erzeugenden Referenz in einem Knoten. Eine Eigenschaft erhält ihre Gültigkeit mit der anfänglichen Iterationsdistanz null. Erhaltungsfunktionen dagegen können Iterationsdistanzen unverändert belassen, verringern oder vollständig zurücknehmen. Wenn eine Instruktion eine Referenz enthält, die Datenflußeigenschaften vorheriger Referenzen beeinflußt, so wird das durch diese Funktionenklasse ausgedrückt. Zur

Bestimmung der erhaltenen Iterationsdistanz muß eine Fallunterscheidung getroffen werden. Exit-Funktionen erhöhen die Iterationsdistanzen eintreffender, gültiger Eigenschaften um eins, denn sie beschreiben den Übergang in eine neue Iteration. Ungültige Eigenschaften bleiben unverändert.

Erzeugungsfunktionen

Für den Fall, daß in Knoten n die Referenz d mit der Distanz 0 erzeugt wird, d.h. $d \in G[n]$, gilt:

$$f_n^d(x) = max(x, 0) \tag{4.10}$$

f_n^d heißt *Erzeugungsfunktion* und beschreibt die Situation eines beginnenden Geltungsbereichs einer Referenz, also einer Erzeugung.

Erhaltungsfunktionen

Durchläuft der Kontrollfluß einen Knoten, der möglicherweise Instanzen vorheriger Definitionen vernichtet, so ist die maximale Iterationsdistanz erhaltener Instanzen zu bestimmen. Die dazu eingesetzten *Erhaltungsfunktionen* haben die Form

$$f_n^d(x) = min(x, p_n^d) \tag{4.11}$$

mit einer Konstanten p_n^d. Zur Bestimmung von p_n^d sind einige Fallunterscheidungen notwendig, darin bezeichne d eine Definition $d = X[idx_1(i)]$, die den Knoten n passiert und von der zu untersuchen ist, wie viele vorherige Instanzen erhalten bleiben. idx_1 ist eine von der Induktionsvariablen i abhängige, affine Indexfunktion.

1. Der Knoten n enthält keine Definition des Arrays X, d.h. entweder gilt $K[n] = \{\}$ oder n enthält eine Definition eines Arrays Y mit $X \neq Y$. Alle Instanzen von d bleiben erhalten, damit gilt $p_n^d = \top$.

2. Der Knoten n enthält eine Definition[2] $d' = X[idx_2(i)] \in K[n]$. Es wird also eine vorherige Instanz einer Definition d durch d' vernichtet, so daß die maximale Anzahl der erhaltenen Instanzen ermittelt werden muß. Diese Bestimmung ist nicht exakt, sondern nur eine Approximation. Deshalb ist es an dieser Stelle besonders wichtig, eine konservative Lösung – d.h. eine sichere Unterschätzung der erhaltenen Instanzen vorheriger Iterationen – zu liefern, da andernfalls die Korrektheit nicht gewährleistet wäre.

   In vorherigen Iterationen haben Instanzen von $d = X[idx_1(i)]$ aus Sicht des Knotens n die Form $d = X[idx_1(i - \delta)], \delta \geq 1$. Um den Fall zu

[2] An dieser Stelle und im folgenden weicht die Notation zur Vereinfachung des Lesbarkeit ein wenig von der Darstellung in [8] ab.

beschreiben, daß d im Schleifenkörper vor Knoten n erscheint, ohne dabei eine Iterationsgrenze zu passieren, wird die Distanz $\delta = 0$ mit zu den Bereich möglicher Iterationsdistanzen genommen. Um später diese beiden Fälle auseinanderhalten zu können, aber dennoch einheitlich im Modell zu behandeln, wird dafür ein Prädikat pr definiert:

$$pr(d,n) = \begin{cases} 0 & \text{, falls } d \text{ in einem Vorgänger von } n \text{ enthalten ist} \\ 1 & \text{, sonst} \end{cases} \tag{4.12}$$

Damit kann p_n^d formal beschrieben werden, abkürzend bezeichne I das Iterationsintervall $[1, \ldots, UB]$:

$$p_n^d = max\{\delta | \forall i \in I, \forall \delta, pr(d,n) \leq \delta < UB : idx_2(i) \neq idx_1(i - \delta)\} \tag{4.13}$$

Die Formel bringt zum Ausruck, daß also das maximale δ gesucht wird, für daß die Definition d' mit der Indexfunktion idx_2 und die „eintreffende“ Definition d mit der Indexfunktion idx_1 nicht miteinander „in Konflikt geraten“. Für die praktische Bestimmung von p_n^d ist diese Darstellung allerdings wenig geeignet, denn die Iterationsdistanz geht aus $idx_2(i) \neq idx_1(i - \delta)$ noch nicht ohne weiteres hervor. Dazu muß zunächst noch eine Umformung vorgenommen werden. Es werden idx_1 und idx_2 in ihre Bestandteile zerlegt ($idx_1(i) = a_1 \times i + b_1$ bzw. $idx_2(i) = a_2 \times i + b_2$), dann voneinander subtrahiert bzw. gleichgesetzt ($idx_1(i - \kappa) - idx_2(i) = 0$) und nach κ umgestellt, wobei das von i abhängige κ mit $k(i)$ bezeichnet wird ($k(i) = \kappa$):

$$k(i) = \frac{a_1 - a_2}{a_1} \times i + \frac{b_1 - b_2}{a_1} \tag{4.14}$$

Eingesetzt in (4.13) ergibt sich für p_n^d:

$$p_n^d = max\{\delta | \forall i \in I, \forall \delta, pr(d,n) \leq \delta < UB : \delta \neq k(i)\} \tag{4.15}$$

Durch eine weitere Fallunterscheidung[3] kann p_n^d dann in praktischen Anwendungen einfach ermittelt werden:

(a) $k(i)$ entspricht der Konstanten $pr(d,n)$, d.h. $\forall i \in I : k(i) = pr(d,n)$. Jede Definition d wird durch d' vernichtet, d bleibt über keine Iteration gültig. Somit gilt: $p_n^d = \perp$. Dieser Fall entspricht in einem Programm der Situation, daß einer erzeugenden Referenz ohne dazwischenliegende Iterationen eine vernichtende Referenz folgt, die das gleiche Array-Element betrifft.

[3] Entgegen der Darstellung in [8] sollte zwischen vier Fällen anstelle von drei Fällen unterschieden werden, da sonst Mehrdeutigkeiten bei der Zuordnung auftreten. Fall c) ist gegenüber der Original-Darstellung hinzugefügt.

Beispiel:

```
x = a[i];        n = 1, d = 1, G[1] = a[i], K[1] = {}
...
a[i] = y;        n = 2, d = 2, G[2] = K[2] = a[i]
```

$pr(d = 1, n = 2) = 0, k(i) = \frac{1-1}{1} \times i + \frac{0-0}{1} = 0$

(b) $k(i)$ ist immer kleiner als $pr(d, n)$, d.h. $\forall i \in I : k(i) < pr(d, n)$. Keine Instanz der Definition d wird durch d' vernichtet, daher bleiben alle vorherigen Instanzen erhalten: $p_n^d = \top$. Dieser Fall tritt auf, falls eine vernichtende Referenz nur Elemente adressiert, die von einer erzeugenden Referenz noch nicht adressiert wurden. Es besteht noch keine Gültigkeit, die vernichtet werden könnte.

Beispiel:

```
x = a[i];        n = 1, d = 1, G[1] = a[i], K[1] = {}
...
a[i+1] = y;      n = 2, d = 2, G[2] = K[2] = a[i + 1]
```

$pr(d = 1, n = 2) = 0, k(i) = \frac{1-1}{1} \times i + \frac{0-1}{1} = -1$

(c) Es existiert im Iterationsintervall eine Überschneidung zwischen den zwei Definitionen d und d', die jedoch nicht über das gesamte Iterationsintervall wiederkehrend ist, d.h. $\exists i \in I : k(i) = pr(d, n) \wedge \forall j \in I, j \neq i : k(j) < pr(d, n)$. Es gibt also zwischenzeitlich den Fall, daß eine Definition d von d' vernichtet wird. Um das Verhalten korrekt zu modellieren, wird die sichere Approximation $p_n^d = \perp$ verwendet.

Beispiel:

```
x = a[2*i+2];        n = 1, d = 1, G[1] = a[2 × i + 2], K[1] = {}
...
a[3*i+1] = y;        n = 2, d = 2, G[2] = K[2] = a[3 × i + 1]
```

$pr(d = 1, n = 2) = 0, k(i) = \frac{2-3}{2} \times i + \frac{2-1}{2} = -\frac{1}{2} \times i + \frac{1}{2}$
Für $i = 1$ gilt $k(i) = pr(d, n) = 0$, für $i > 1$ gilt $k(i) < pr(d, n)$.

(d) Andernfalls nimmt $k(i)$ Werte größer als $pr(d, n)$ an, d.h. $\exists i \in I : k(i) > pr(d, n)$. Instanzen von d' vernichten frühere Instanzen von d, so daß bloß einige erhalten bleiben. Deren Anzahl muß abermals durch eine konservative Approximation ermittelt werden:

$$p_n^d = \lceil min\{k(i) | \forall i \in I, k(i) > pr(d, n)\}\rceil - 1. \tag{4.16}$$

Von all denjenigen „Konflikten“, die zwischen den Definitionen auftreten, bestimmt diejenige mit der geringsten Iterationsdistanz die

Anzahl der erhaltenen Instanzen. Dieser Fall entsteht durch eine vernichtende Referenz, die regelmäßig in fester Iterationsdistanz von einer erzeugenden Referenz, das dort referenzierte Element erneut referenziert.

Beispiel:

```
x = a[i+3];
...
a[i] = y;
```

$n = 1, d = 1, G[1] = a[i+3], K[1] = \{\}$

$n = 2, d = 2, G[2] = K[2] = a[i]$

$pr(d = 1, n = 2) = 0, k(i) = \frac{1-1}{1} \times i + \frac{3-0}{1} = 3$

Exit-Funktionen

Bei Erreichen und Durchlaufen des *Exit*-Knotens geht der Kontrollfluß von einer Iteration zur nächsten über. Dabei wird die Induktionsvariable um eins inkrementiert. Die Datenflußinformation, die diesen Knoten passiert, muß darauf Rücksicht nehmen, denn die durch die Verbandselemente ausgedrückten Iterationsdistanzen erhöhen sich entsprechend. Nach Verlassen des *Exit*-Knotens hat die maximale Iterationsdistanz eine Gültigkeit von $\delta+1$, wenn sie zuvor bei Erreichen des *Exit*-Knotens über δ Iterationen gültig war. Dementsprechend wird die Transferfunktion eines *Exit*-Knotens durch

$$f^d_{exit}(x) = x++ \tag{4.17}$$

definiert, wobei der ++-Operator die oben definierte Inkrement-Operation auf L ist. Es wird beachtet, daß ungültige Lösungen auch in der folgenden Iteration ungültig bleiben, bzw. daß über alle Iterationen gültige Distanzen nicht weiter wachsen.

Beispiel 4.2.4 *Transferfunktionen zu Bsp. 4.2.3*

	d=1	*d=2*	*d=3*	*d=4*
n=1	*Erzeugung* $max(x,0)$	*Erhaltung,1)* $min(x,\top)$	*Erhaltung,2)* $min(x,\bot)$	*Erhaltung,2)* $min(x,1)$
n=2	*Erhaltung,2)* $min(x,\top)$	*Erzeugung* $max(x,0)$	*Erhaltung,1)* $min(x,\top)$	*Erhaltung,1)* $min(x,\top)$
n=3	*Erhaltung,2)* $min(x,\top)$	*Erhaltung,1)* $min(x,\top)$	*Erzeugung* $max(x,0)$	*Erhaltung,2)* $min(x,\bot)$
n=4	*Erhaltung,1)* $min(x,\top)$	*Erhaltung,1)* $min(x,\top)$	*Erhaltung,1)* $min(x,\top)$	*Erhaltung,1)* $min(x,\top)$
n=5	*Erhaltung,2)* $min(x,\top)$	*Erhaltung,2)* $min(x,\top)$	*Erhaltung,2)* $min(x,\top)$	*Erzeugung* $max(x,0)$
n=6	*Exit* $x++$	*Exit* $x++$	*Exit* $x++$	*Exit* $x++$

Das Beispiel 4.2.4 zeigt die Transferfunktionen der Knoten $n = 1, \ldots, 6$ für die vier erzeugenden Referenzen zur Schleife aus Beispiel 4.2.3. Für die Knoten $1 \ldots n$ sind jeweils der Typ der Funktion und die Funktion selbst angegeben. Für Erhaltungsfunktionen ist der Fall, nach dem sie konstruiert wurden, gekennzeichnet.

4.2.4 Datenfluß-Gleichungssystem und iterative Fixpunkt-Lösung

Nachdem für jeden einzelnen Knoten des LCFG die Transferfunktion bestimmt wurde, kann ein Datenfluß-Gleichungssystem aufgestellt werden, daß den Array-Datenfluß in der gesamten betrachteten Schleife modelliert. Zu diesem Gleichungssystem gehören für jeden Knoten n die Vektoren $IN[n] = (x_1, \ldots, x_m)$ und $OUT[n] = (y_1, \ldots, y_m)$, die die maximalen Iterationsdistanzen für jede der m Referenzen in der Schleife beim Erreichen und Verlassen des Knotens n beschreiben. Für eine bestimmte Referenz d und einen Knoten n kann die Information, mit welcher maximalen Iterationsdistanz der Knoten n von d erreicht wird bzw. verlassen wird, an $IN[n, d] = x_d$ bzw. $OUT[n, d] = y_d$ abgelesen werden.

Das Aufstellen eines Datenfluß-Gleichungssystems allein genügt noch nicht, es muß auch noch gelöst werden. Dies geschieht durch eine Fixpunkt-Iteration (siehe Kapitel 3.3.2). Dabei wird nach einer Initialisierung, die die Lösung überschätzt, nach und nach die Datenfluß-Information durch die Knoten des LCFG propagiert, d.h. es werden die entsprechenden Transferfunktionen auf die eintreffenden Werte angewendet. Während dieser fortgesetzten Iteration werden die Werte an den Knoten verkleinert, bis das Verfahren einen größten Fixpunkt erreicht, bei dem keine weitere Veränderung der Werte mehr auftritt. Damit terminiert die Iteration und die Lösung des Datenflußproblems kann an den Vektoren IN und OUT für jeden Knoten abgelesen werden.

Zur Initialisierung wird der LCFG in *Reverse Postorder* (siehe Kapitel 3.3.3) traversiert, dabei wird für jeden Knoten n und $\forall d \in [1, \ldots, m]$ folgende Zuweisung vorgenommen:

$$IN[n, d]^0 = \begin{cases} \bot & \text{, falls } n = 1 \text{ Schleifeneintritt} \\ \bigwedge\limits_{m \in pred(n)} OUT[m, d]^0 & \text{, sonst} \end{cases} \tag{4.18}$$

$$OUT[n, d]^0 = \begin{cases} \top & \text{, falls } d \in G[n] \\ \bigwedge\limits_{m \in pred(n)} IN[n, d]^0 & \text{, sonst} \end{cases} \tag{4.19}$$

Sobald eine Definition in einem Knoten erscheint, wird deren Gültigkeit in OUT durch $\top$ überschätzt, ansonsten passieren Werte einen Knoten unverändert. Den ersten Knoten des Schleifenkörpers erreichen natürlich zunächst noch keine Definitionen, so daß alle Werte in IN des ersten Knotens $\perp$ sind. Bei allen weiteren Knoten n wird mit der minimalen Iterationsdistanz aller Vorgängerknoten $pred(n)$ des Knotens n gerechnet.

Die nachfolgenden Iterationsschritte, die auch in *Reverse Postorder* erfolgen sollten[4], arbeiten wie folgt[5]:

$$IN[n,d]^{i+1} = \begin{cases} \bigwedge\limits_{m \in pred(n)} OUT[m,n]^i & \text{, falls } n = 1 \\ \bigwedge\limits_{m \in pred(n)} OUT[m,d]^{i+1} & \text{, sonst} \end{cases} \tag{4.20}$$

$$OUT[n,d]^i = f_n^d(IN[n,d]^i) \tag{4.21}$$

Wegen der geforderten *Single-entry/Single-exit*-Eigenschaft der Schleifen läßt sich 4.20 vereinfachen zu:

$$IN[n,d]^{i+1} = \begin{cases} OUT[Exit,n]^i & \text{, falls } n = 1 \\ \bigwedge\limits_{m \in pred(n)} OUT[m,d]^{i+1} & \text{, sonst} \end{cases} \tag{4.22}$$

Beispiel 4.2.5 *Lösung des Datenflußproblems aus Bsp. 4.2.3*

		Initialisierung	*1. Durchlauf*	*2. Durchlauf*
n=1	*IN[1]*	$\perp,\perp,\perp,\perp$	$\top,\top,\top,\perp$	$\top,\top,1,\perp$
	OUT[1]	$\top,\perp,\perp,\perp$	$\top,\top,\perp,\perp$	$\top,\top,\perp,\perp$
n=2	*IN[2]*	$\top,\perp,\perp,\perp$	$\top,\top,\perp,\perp$	$\top,\top,\perp,\perp$
	OUT[2]	$\top,\top,\perp,\perp$	$\top,\top,\perp,\perp$	$\top,\top,\perp,\perp$
n=3	*IN[3]*	$\top,\top,\perp,\perp$	$\top,\top,\perp,\perp$	$\top,\top,\perp,\perp$
	OUT[3]	$\top,\top,\top,\perp$	$\top,\top,0,\perp$	$\top,\top,0,\perp$
n=4	*IN[4]*	$\top,\top,\top,\perp$	$\top,\top,0,\perp$	$\top,\top,0,\perp$
	OUT[4]	$\top,\top,\top,\perp$	$\top,\top,0,\perp$	$\top,\top,0,\perp$
n=5	*IN[5]*	$\top,\top,\top,\perp$	$\top,\top,0,\perp$	$\top,\top,0,\perp$
	OUT[5]	$\top,\top,\top,\top$	$\top,\top,0,0$	$\top,\top,0,0$
n=6	*IN[6]*	$\top,\top,\top,\perp$	$\top,\top,0,\perp$	$\top,\top,0,\perp$
	OUT[6]	$\top,\top,\top,\perp$	$\top,\top,1,\perp$	$\top,\top,1,\perp$

[4] In [8] wird gezeigt, daß bei *Reverse Postorder*-Traversierung und Nichtvorhandensein von *Backward-Gotos* in der Schleife aufgrund der Monotonie und (schwachen) Idempotenz der verwendeten Operatoren zwei weitere Iterationen über den Schleifenkörper zum Erreichen der Konvergenz auf den Fixpunkt hinreichend sind. Die Platzkomplexität wird mit $O(N^2)$ bei N Instruktionen im Schleifenkörper angegeben.

[5] Die in der Original-Publikation [8] angegeben Formeln zur Datenfluß-Iteration enthalten einen Fehler, der in dieser Darstellung korrigiert ist.

Beispiel 4.2.5 zeigt zur Schleife aus Beispiel 4.2.3 die Belegungen der IN- und OUT-Vektoren während der Fixpunkt-Iteration. Die Lösung des *Must-Reaching-Definitions*-Datenflußproblems ist in der letzten Zeile abzulesen. Die erste und zweite Definition erreichen alle Knoten über alle Iterationen hinweg. Die dritte Definition erreicht die unmittelbar folgenden Knoten der gleichen Iteration bzw. den ersten Knoten der folgenden Iteration. Die vierte Iteration erreicht keinen Knoten zwingend, der Grund dafür ist die Lage in einem Verzweigungsast.

4.2.5 Möglichkeiten zur Parametrisierung

Das vorgestellte δ-Verfahren kann parametrisiert werden, damit unter Beibehaltung des Verfahrensablaufs eine Anpassung an verschiedene spezielle Datenflußprobleme erfolgen kann. So ist es möglich, mit dem gleichen Verfahren allein durch Wahl verschiedener externer Parameter sowohl Vorwärts- und Rückwärtsanalyse als auch Must- und May-Probleme zu behandeln.

Die Parameter, mit denen die Anpassung erfolgt, sind der verwendete Verband, die Mengen G und K, sowie die Funktion $k(i)$ und der LCFG. G und K spezifizieren dabei sowohl die konkrete Datenflußanalyse, aber auch die zu analysierende Schleife. Eine mögliche konkrete Parametrisierung ist die *Must-Reaching-Definitions*-Analyse, die in den vorangegangenen Beispielen verwendet wurde.

Die nachfolgenden Abschnitte zeigen wie die Parametrisierung für bestimmte Problemklassen erfolgen kann. Zu diesen Klassen gehören die Wahl der Datenflußrichtung (vorwärts/rückwärts), der Informationsqualität (must/may) sowie spezielle Eigenarten eines konkreten Datenflußproblems.

Vorwärts- und Rückwärtsprobleme

In den vorangegangenen Darstellungen war bislang von einem Vorwärtsproblem (z.B. *reaching definitions*) ausgegangen worden, bei dem der Datenfluß von den Kontrollflußvorgängern zu -nachfolgern, und von frühen Iterationen zu späteren Iterationen weitergegeben wird. Es gibt jedoch auch Probleme (z.B. *busy expressions*) deren Datenfluß die entgegengesetzte Richtung nimmt. Zur Behandlung eines solchen Rückwärtsproblems wird mit einem *umgekehrten LCFG* gearbeitet, d.h. alle Kanten des LCFG weisen in die entgegengesetzte Richtung als es bei einem LCFG eines Vorwärtsproblems der Fall ist. Der Datenfluß beginnt bei späteren Iterationen und fließt in Richtung früherer Iterationen. Entsprechend kehren sich positive und negative Iterationsdistanzen um, was durch eine Anpassung der Funktion $k(i)$ berücksichtigt wird:

$$k(i) = \frac{a_2 - a_1}{a_1} \times i + \frac{b_2 - b_1}{a_1} \qquad (4.23)$$

Must- und May-Probleme

Die bisherige Darstellung zeigt das Verfahren wie es für die Verwendung bei *Must*-Problemen einzusetzen ist. Zur Anpassung an *May*-Probleme sind Änderungen an der partiellen Ordnung des Verbandes und dessen Operatoren, an der Erhaltungsfunktion, und auch der Initialisierung der Fixpunkt-Iteration notwendig.

Die Ordnung des Verbandes muß für May-Probleme umgekehrt werden, d.h. aus 4.5 wird:

$$\sqsubseteq: \forall x_i \in [\bot = UB - 1, \ldots, 0, \top] : x_i < x_{i+1} \tag{4.24}$$

Fortan bezeichnet $\top$ Gültigkeit über „keine Instanz", $\bot$ hingegen Gültigkeit über „alle Instanzen". Der Meet-Operator $\wedge$ wird ersetzt durch die duale Operation, es gilt $\wedge = max$.

Die Erhaltungsfunktion, insbesondere die Bestimmung von p_n^d, muß ebenso verändert werden, denn bislang kam es bei einem potentiellen „Konflikt" zu einer Unterschätzung der Lösung. Bei einem *May*-Problem hingegen verliert eine Definition erst dann ihre Gültigkeit, wenn dies eindeutig nachgewiesen. Durch die Annahme, ein Definition könne solange gültig bleiben, wie diese von Elementen in $K[n]$ nicht mit Sicherheit vernichtet werden, beschreibt p_n^d eine Überschätzung der Lösung, die angibt, daß d bis zu p_n^d Iterationen erhalten werden *kann*. Es bleibt noch zu klären, wie eindeutig nachgewiesen werden kann, daß eine Definition wiederkehrend vernichtet wird. Das kann nur geschehen, wenn zwischen zwei Definition eine konstante Iterationsdifferenz[6] liegt, also gilt $d = X[idx(i)]$ mit $d \in K[n]$ und $d' = X[idx(i + c)]$. Es bleiben bis zu $c - 1$ Instanzen von d' durch d erhalten, während für die übrigen deren Vernichtung eindeutig nachgewiesen wurde. Es ergibt der obigen Darstellung analoge Fallunterscheidung für p_n^d, mit der Anpassung $p_n^d = c - 1$ für $k(I) = c, c > pr(d, n)$ bei ansonsten dualem Verhalten.

Durch die Umkehrung der Ordnung des Verbandes sowie der Notwendigkeit zur Überschätzung der Lösung, muß auch die Initialisierung den Verhältnissen angepaßt werden. Am einfachsten läßt sich dies durch den Wert $\top$ für alle Definitionen erledigen. Die Effizienz des Verfahrens sinken dadurch aber sinken oder im Extremfalls eines (theoretisch) unendlich großen UB ist die Terminierung nicht mehr sichergestellt.[7]

[6] In [8] wird an entsprechender Stelle in nicht ganz korrekter Weise $d' = X[idx(i) + c]$ verwendet. Das gilt jedoch nur, wenn für die Steigung a der affinen Indexfunktion idx gilt: $a = 1$. Allgemeiner ist die Version mit dem konstanten Summanden als *Argument* der Indexfunktion.

[7] Die Original-Publikation gibt Hinweise auf eine verbesserte Vorgehensweise, deren Terminierung gesichert ist und die *ohne* Initialisierung auskommt.

Konkretes Datenflußproblem

Ein konkretes Datenflußproblem läßt sich dadurch beschreiben, welche Referenzen als erzeugend oder vernichtend angesehen werden. Beispielsweise enthält $G[n]$ bei *δ-available-values* (siehe 5.2.1), die Definitionen und Gebräuche eines von Array-Elementen im Knoten n, während $K[n]$ ausschließlich die Definitionen umfaßt. Durch die Zuordnung der Referenzen zu G und K entsprechend einer gewünschten, speziellen Datenflußanalyse erfolgt also der letzte Schritt der Parametrisierung des allgemeinen Verfahrens.

4.2.6 Behandlung mehrdimensionaler Arrays und Loop Nests

Die δ-Technik wurde bislang nur für eindimensionale Arrays in einfachen Schleifen verwendet. Darüberhinaus sind durchaus auch mehrdimensionale Arrays in *Loop Nests* dem Verfahren zugänglich.

Zunächst soll dargestellt werden, wie Loop Nests verarbeitet werden. Bei einem Loop Nest wird die Analyse mit der innersten Schleife l_1 mit der Induktionsvariablen i_1 wie bisher gestartet, und dann mit der nächsten umgebenden Schleife l_2 mit der Induktionsvariablen i_2 fortgesetzt, bis schließlich die äußerste Schleife l_n mit der Induktionsvariablen i_n erreicht wird. Äußere Induktionsvariablen werden in Array-Referenzen der inneren Schleifen als symbolische Konstanten behandelt.[8] Nach Beendigung der Analyse von Schleife l_k kann diese im LCFG der umgebenden Schleife l_{k+1} vollständig durch einen *Zusammenfassungsknoten* ersetzt werden. Diesem werden ebenso wie gewöhnlichen Knoten zwei Mengen $G[l_k]$ und $K[l_k]$ zugewiesen, die eine Zusammenfassung der Referenzen des Schleifenkörpers von l_k modellieren. Zur sicheren Approximation enthält $G[l_k]$ dabei nur diejenigen Referenzen aus l_k, deren Indexfunktionen durch die Induktionsvariable l_{k+1} bestimmt werden, $K[l_k]$ hingegen alle Referenzen aus l_k. Damit wird der Möglichkeit Rechnung getragen, daß eine Referenz $X[a_1 \times i_k + b_1]$ alle Instanzen von $X[a_2 \times i_{k+1} + b_2]$ vernichten kann. Letztendlich kann bei der Analyse von l_{k+1} für den Zusammenfassungsknoten eine Transferfunktion f_k aus $G[l_k]$ und $K[l_k]$ gebildet werden. Somit kann die Analyse für l_{k+1} ohne Berücksichtigung von Ausnahmen durchgeführt werden.

Mehrdimensionale Array-Referenzen werden vor der weiteren Verarbeitung linearisiert, d.h. Referenzen der Form $X[idx_1(i), \ldots, idx_n(i)]$ mit n Dimensionen und je einer Indexfunktion $idx_k(i)$ für jede Dimension werden in die Form $X[idx(i)]$ gebracht. Dazu werden die Indexfunktionen jeder Dimension mit deren Größe multipliziert und anschließend summiert. Aus $X[i, i+2]$ mit der Größe

[8] Durch die Behandlung äußerer Induktionsvariablen als symbolische Konstanten muß die Berechnung der Iterationsdistanzen in den Erhaltungsfunktionen entsprechend angepaßt werden, d.h. es müssen Erweiterungen zum Umgang mit symbolischen Ausdrücken gefunden werden, die evtl. die äußeren Iterationsgrenzen mit in die Berechnung der maximalen Iterationsdistanzen einbeziehen.

N für die erste Dimension wird z.B. $X[i \times N + i + 2] = X[(N+1) \times i + 2]$. Anschließend kann die Analyse mit diesen linearisierten Ausdrücken unverändert weiterarbeiten.

Treten mehrdimensionale Array-Referenzen in Loop Nests auf, d.h. beide gerade genannten Erweiterungen gegenüber dem Grundmodell sind zugleich vorhanden, so können auch beide Erweiterungen nacheinander angewendet werden. Zunächst werden wieder die Array-Referenzen linearisiert.[9] Anschließend können die ineinandergeschachtelten Schleifen von innen nach außen analysiert werden. Wiederum gehen äußere Induktionsvariablen bei der Analyse innerer Schleifen als symbolische Konstanten ein. Dabei können wegen der sequentiellen Betrachtung der Schleifen auch nur Abhängigkeiten gefunden werden, die ausschließlich von einer Induktionsvariablen abhängen. Treten in einem Loop Nest wiederkehrende Zugriffe in Abhängigkeit von mehreren Induktionsvariablen auf, d.h. die Abhängigkeitsvektoren enthalten mehrere Richtungskomponenten, so können diese nicht ermittelt werden. Das aus [8] entliehene Beispiel 4.2.6 soll dies verdeutlichen:

Beispiel 4.2.6 *Abhängigkeiten in verschiedenen Richungen*

```
for(j = 1; j < UB1; j++)
   for(i = 1; i < UB2; i++)
   {
      X[i+1,j] = X[i,j];     /* 1) */
      Y[i,j+1] = Y[i,j-1];   /* 2) */
      Z[i+1,j] = Z[i,j-1];   /* 3) */
   }
```

Während die Instruktionen 1) und 2) nur in je einer Richtung einen wiederkehrenden Zugriff verursachen und somit bei der Analyse bzgl. der Induktionsvariablen `i` bei 1) bzw. `j` bei 2) erkannt werden, liegt bei 3) eine Regelmäßigkeit bzgl. `i` *und* `j` vor, die nicht ermittelt werden kann.

Eine Behandlung von mehrdimensionalen Arrays in *Tight Loop Nests* unter Berücksichtigung von Abhängigkeitsvektoren mit mehreren Richtungskomponenten scheint möglich zu sein durch eine simultane Betrachtung aller Induktionsvariablen. Dazu sollten die Indexfunktionen nicht linearisiert sein, sondern es sind getrennte Ausdrücke für jede Dimension erforderlich. Dem Vorschlag der Original-Publikation folgend, kann die Qualifizierung der Lösung durch einen Vektor von Verbandselementen zu einem Vektor von Vektoren von Verbandselementen verallgemeinert werden. Jeder Induktionsvariablen wird dabei ein eigener Vektor in der Lösung zugeordnet aus dem sich Abhängigkeiten bzgl. der entsprechenden Schleife ablesen lassen (siehe dazu Beispiel 4.2.7). Die Datenflußanalyse erfolgt nunmehr nicht in einem hierarchischem Vorgehen bei dem

[9] An dieser Stelle können Indexfunktionen mit mehreren verschiedenen Induktionsvariablen entstehen.

innere Schleifen durch Zusammenfassungsknoten ersetzt werden, sondern in einem Durchlauf, der für alle Induktionsvariablen gleichzeitig stattfindet. Zwingend ist die Voraussetzung, daß nicht über ein Zeilenende hinaus in die nächste Zeile eines Arrays zugegriffen werden darf. Die – in DSP-Applikationen durchaus zu findende – Annahme, daß $X[i, M+1] = X[i+1, 1]$ mit der Größe M der zweiten Dimension ist nicht gestattet. Ob diese Vorgehensweise jedoch auch korrekt unter allen möglichen Umständen ist, wurde in dieser Diplomarbeit jedoch nicht untersucht.

Beispiel 4.2.7 *Analyse von Loop Nests und mehrdimensionale Arrays mit zusammengesetzten Abhängigkeitsvektoren*

	Initialisierung		*1. Durchlauf*		*2. Durchlauf*	
	(i_1, i_2, i_3)	(j_1, j_2, j_3)	(i_1, i_2, i_3)	(j_1, j_2, j_3)	(i_1, i_2, i_3)	(j_1, j_2, j_3)
x=a[i][j]	$(\bot,\bot,\bot)$	$(\bot,\bot,\bot)$	$(\top,\top,\top)$	$(\top,\top,\top)$	$(2,\top,\top)$	$(1,\top,\top)$
	$(\top,\bot,\bot)$	$(\top,\bot,\bot)$	$(\top,\top,\top)$	$(\top,\top,\top)$	$(2,\top,\top)$	$(1,\top,\top)$
a[i-2][j-1]	$(\top,\bot,\bot)$	$(\top,\bot,\bot)$	$(\top,\top,\top)$	$(\top,\top,\top)$	$(2,\top,\top)$	$(1,\top,\top)$
	$(\top,\top,\bot)$	$(\top,\top,\bot)$	$(1,\top,\top)$	$(0,\top,\top)$	$(1,\top,\top)$	$(0,\top,\top)$
z=a[i-5][j-5]	$(\top,\top,\bot)$	$(\top,\top,\bot)$	$(1,\top,\top)$	$(0,\top,\top)$	$(1,\top,\top)$	$(0,\top,\top)$
	$(\top,\top,\top)$	$(\top,\top,\top)$	$(1,\top,\top)$	$(0,\top,\top)$	$(1,\top,\top)$	$(0,\top,\top)$
Exit	$(\top,\top,\top)$	$(\top,\top,\top)$	$(1,\top,\top)$	$(0,\top,\top)$	$(1,\top,\top)$	$(0,\top,\top)$
	$(\top,\top,\top)$	$(\top,\top,\top)$	$(2,\top,\top)$	$(1,\top,\top)$	$(2,\top,\top)$	$(1,\top,\top)$

4.2.7 Ermöglichte Optimierungen

Mit dem vorgestellten, allgemeinen Verfahren wird eine Verallgemeinerung bekannter skalarer Datenflußanalysen auf die bisher nicht zugängliche Domäne von Abhängigkeiten zwischen Array-Referenzen betrieben. Damit werden Array-Referenzen auch Optmierungen zugänglich, die bislang nur für skalare Variable anwendbar waren (z.B. Common Subexpression Elimination). Prinzipiell steht einer Übertragung skalarer, iterativer Datenflußanalysen auf die Anwendung bei Array-Referenzen nichts entgegen.

Darüberhinaus gestattet die Bestimmung von maximalen Iterationsdistanzen eine Reihe weiterer Optimierungen, die spezifisch für den Bereich der Array-Referenzen innerhalb von Schleifen sind. Durch passende Parametrisierungen können die Analysen *available-* und *busy-expression* zu *δ-available-values* bzw. *δ-busy-expressions* erweitert werden, so daß Array-Abhängigkeiten über Iterationsgrenzen hinweg betrachtet werden können. Dadurch werden die in Kapitel 5 diskutierten *Redundant Load/Store Eliminations* ebenso wie deren Verallgemeinerungen des *Register Pipelinings* ermöglicht.

Optimierungen, deren Ziel nicht die Redundanzverminderung ist, werden auch ermöglicht. So kann ein *Kontrolliertes Loop Unrolling* zur Steigerung der Parallelität einer Schleife oder eine *Registerallokation* für Array-Elemente und skalare Variable effizient durch Bereitstellung der benötigten Analysen unterstützt werden.

Weitere Analysen sind durch Parametrisierung möglich, so daß z.B. eine *May-δ-available values*-Analyse dazu dienen könnte, lokal antizipierbare Ausdrücke zur Elimination partiell redundanter Array-Zugriffe zu finden.

Einen wichtigen Beitrag leisten die maximalen Iterationsdistanzen auch beim *Stretched-Loop*-Verfahren zur Array-Datenflußanalyse (siehe 4.3). Dort werden Array-Datenabhängigkeiten präziser bestimmt als beim vorgestellten δ-Verfahren. Dazu werden aber u.a. die Werte maximaler Iterationsdistanzen gebraucht, die vorab bestimmt werden können.

4.2.8 Vor- und Nachteile

Ein wesentlicher Vorteil der δ-Datenflußanalyse ist die Möglichkeit zur Behandlung von Array-Referenzen. Damit wird die Ursache der bisherigen Vernachlässigung von Array-Referenzen bei späteren Optimierungsverfahren angegangen. Die Allgemeinheit des Verfahrens kommt einer Reihe verschiedener Optimierungen zugute, d.h. durch einfach zu realisierende Parametrisierungen können mit dem gleichen Verfahren sehr unterschiedliche Datenflußanalysen durchgeführt werden. Das Verfahren ist recht einfach zu implementieren und benötigt auch relativ wenig Ressourcen zur Laufzeit. Bei Anwendung von Reverse Postorder zur Traversierung eines LCFG mit N Knoten und Verbot von Bachward-Gotos im Schleifenkörper wird jeder Knoten maximal dreimal durchlaufen bis ein Fixpunkt erreicht wird. Der Platzbedarf für die IN- und OUT-Informationen an den Knoten beträgt $O(N^2)$.

Nachteilig ist die Beschränkung auf Programmfragmente, die ausschließlich affine Ausdrücke in Indexfunktionen enthalten. Bei komplexeren Ausdrücken können die Schleifen nicht mehr analysiert werden. Die Präzision der Lösung ist auf Darstellungen auf Iterationsebene beschränkt. Es werden keine Abhängigkeiten von einzelnen Instruktionen berichtet, sondern nur die Anzahl der Iterationen, über die die Lösung Gültigkeit besitzt. Optimierungen, die genauere Informationen verwerten können, werden somit nicht optimal unterstützt. Die Behandlung mehrdimensionaler Arrays und Loop Nests ist noch nicht ganz ausgereift und liefert nur grobe Approximationen der Datenabhängigkeiten.

4.3 *Stretched-Loop*-Array-Datenflußanalyse

In [6] wird ein Array-Datenflußanalyse-Verfahren vorgestellt, daß in seiner Approximationsgüte bessere Ergebnisse liefert als das im vorherigen Abschnitt besprochene δ-Verfahren. Dessen Nachteil, daß Abhängigkeiten nur auf Iterationsebene ermittelt werden, dient als Anlaß, die Analyse mit höherer Präzision auf Instruktionsebene durchzuführen. Durch die höhere Präzision werden Redundanzoptimierungen, die auch mit der δ-Technik möglich sind, zu höherer Optimierungsqualität geführt. Zusätzlich werden neue Optimierungen ermöglicht, die diese höhere Präzision zu ihrer Durchführung verlangen. Zwar steigt gegenüber dem δ-Verfahren der Aufwand zur Laufzeit an, er liegt aber immer noch in einem günstigen Rahmen.

Das Verfahren ermöglicht eine präzise Bestimmung des Verhaltens wiederkehrender Array-Referenzen zur Verwendung bei der Elimination partiell/total redundanter Loads/Stores. Eine effiziente Form des Register-Pipelinings wird ermöglicht und auch die Unterstützung zur Effizienzsteigerung von Software-Pipelining bietet sich an.

Ebenso wie bei der δ-Technik werden Vorwärts- und Rückwärtsanalysen unterstützt, genauso wie die Erzeugung von May- und Must-Information. Die Festlegung auf ein spezielles Datenflußproblem erfolgt durch Parametrisierung.

4.3.1 Voraussetzungen

Das *Stretched Loop*-Verfahren kommt mit weniger bzw. weniger strengen Voraussetzungen aus als das δ-Verfahren. Neben affinen Indexfunktionen wird gefordert, daß pro Instruktion der zu analysierenden Schleife maximal eine Array-Referenz vorkommt. Das ist keine allzu einschränkende Forderung, denn durch Einfügen von temporären Variablen lassen sich Instruktionen mit mehr Array-Referenzen in diese Form bringen. Die Lösung der Datenflußanalyse gilt wie bei 4.2 für einen *stabilisierten Zustand* der Schleife, bei dem sich Regelmäßigkeiten des Datenflusses ausgebildet haben. Der *stabilisierte Zustand* ergibt sich nach Durchlaufen einiger Iterationen im „Mittelabschnitt“ des Iterationsintervalls während noch hinreichend viele Iterationen abzuarbeiten sind[10]. Durch die Ausklammerung von Schleifenbeginn und -ende behält das Verfahren seine Gültigkeit für *multiple entry/multiple exit*-Schleifen bei.

4.3.2 Verfahrensüberblick

Anstelle einer sequentiellen Analyse einzelner Array-Referenzen werden alle das gleiche Element betreffenden Referenzen gemeinsam bearbeitet. Die gemeinsam betrachteten Referenzen werden zu *Kongruenzklassen* zusammengefaßt. Dadurch können Abhängigkeiten auf der Ebene einzelner Knoten betrachtet werden, so daß Aussagen darüber gemacht werden können , ob ein Knoten im Schleifenkörper innerhalb der *Live range* eines bestimmten Array-Elementes liegt. Desweiteren werden mehrere aufeinanderfolgende Iterationen, die die gesamte Lebensdauer eines Array-Elementes umfassen, zugleich analysiert. Referenzen außerhalb dieses Fensters können ignoriert werden, da sie für die Bestimmung der Abhängigkeiten keine Bedeutung haben. Für dieses Bearbeitungsfenster wird eine Form der Visualisierung in Gestalt *gestreckter Schleifen (Stretched Loop)*, die in etwa einem Kontrollflußgraphen einer teil-abgerollten Schleife entsprechen, vorgestellt.

[10]Bei der Anwendung in Optimierungen ist darauf zu achten, daß stets dieser stabilisierter Zustand vorliegt. Dazu kann es nötig sein, einen Schleifenprolog und -epilog zu erzeugen.

Bevor eine formale Darstellung des Datenflußanalyse-Verfahrens erfolgt, sind einige Definitionen zur Begriffsbildung erforderlich.

Definition 4.3.1 *Die Array-Referenzen r_1 und r_2 bzgl. eines Arrays A sind* kongruent, *falls sie die Form $r_1 = A[c \times i + k_1]$ und $r_2 = A[c \times i + k_2]$ haben, i die normalisierte Induktionsvariable*[11] *ist und es gilt $k_1 \mod c \equiv k_2 \mod c$.*

Definition 4.3.2 *Die größte Menge kongruenter Referenzen $A[c \times i + j]$ bzgl. eines Arrays A innerhalb einer Schleife bildet eine* Kongruenzklasse $[A, c \times i + k]$ *mit $j \mod c \equiv k \mod c$. Durch die Kongruenzrelation wird die Menge der Referenzen bzgl. eines Arrays A in einer Schleife in eine Menge disjunkter Kongruenzklassen partitioniert.*

Definition 4.3.3 *Sei d die größte Iterationsdistanz zwischen lexikographisch weitest voneinander entfernten Referenzen einer Kongruenzklasse. Eine* gestreckte Schleife (Stretched Loop) *umfaßt dann maximal $d + 1$ aufeinanderfolgende Schleifeniterationen, die fortlaufend von 0 bis d durchnumeriert sind. Die Abschnitte, die einzelnen Iterationen entsprechen, werden* Segmente *genannt. Ein Knoten n des ursprünglichen Schleifenkörpers, der in mehreren Segmenten der* Stretched Loop *liegt, wird zur Unterscheidung mit n_i für das Segment i bezeichnet.*

Definition 4.3.4 *Eine Programmschleife befindet sich im* stabilisierten Zustand, *wenn entsprechend viele Iterationen ausgeführt wurden, so daß sich ein kongruenter Wert entlang der längsten gestreckten Schleifen ausbreiten konnte und andererseits noch so viele Iterationen verbleiben, daß jeder erzeugte Wert auch entsprechend seiner Lebensdauer verbraucht werden kann.*

Im Prinzip heißt dies, daß sich ein *stabilisierter Zustand* einstellt, wenn für alle Werte über alle zugehörigen Iterationen ihrer Lebensdauer sich die Regelmäßigkeit ihrer Erzeugung und ihres Gebrauchs „eingependelt" hat, und keine Initial- oder Terminalwerte behandelt werden.

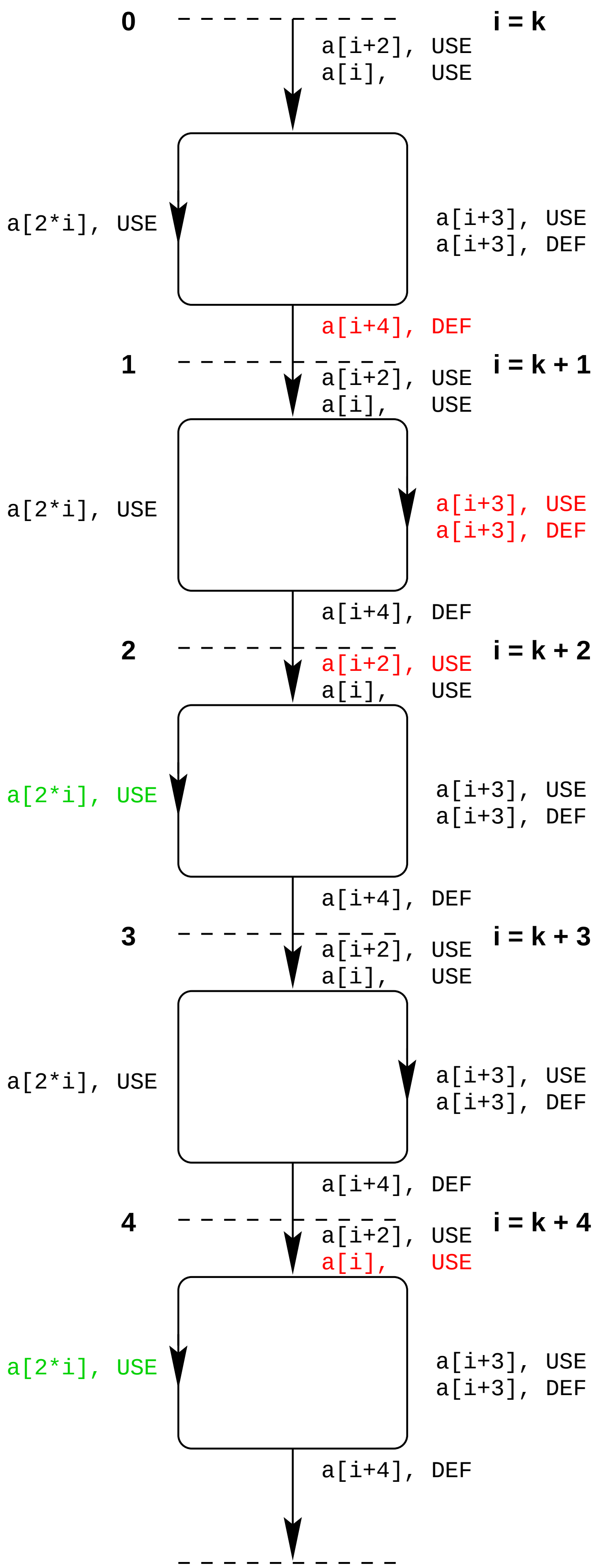
Die Kongruenzklassen werden einzeln analysiert, wobei auch Auswirkungen der Kongruenzklassen untereinander – sofern sie das gleiche Array betreffen – berücksichtigt werden. Vereinfacht gehören einer Kongruenzklasse diejenigen Referenzen an, die über mehrere Iterationen hinweg das gleiche Array-Element referenzieren können und damit in der Lage sind, ein regelmäßiges Zugriffsmuster zu bilden.

[11]Eine *normalisierte* Induktionsvariable ist eine Basisinduktionsvariable mit der Schrittweite 1 und nimmt Werte aus dem Bereich $1, \ldots, N$ an.

Beispiel 4.3.1 zeigt für eine Programmschleife eine Stretched Loop. Die Stretched Loop wird dadurch gewonnen, daß aus den vorkommenden Kongruenzklassen eine ausgewählt wird – hier $[a, 1 \times i]$. Für diese Kongruenzklasse wird das Array-Element `a[k+4]` symbolisch fixiert, da in Folge mehrmals darauf zugegriffen wird. Die Stretched Loop umfaßt fünf Segmente, da über eine Folge von fünf Iterationen wiederholt `a[k+4]` referenziert wird. Die benötigte Information, daß die Lebensdauer von `a[k+4]` fünf ist, kann z.B. mit der δ-Technik zuvor ermittelt werden. Für die Stretched Loop kann ein Ausführungspfad durch die fünf Segmente bestimmt werden. Dieser ist in der Zeichnung durch Pfeile markiert. Die Bedingung `odd(i)` läßt sich auswerten, wenn für das erste Segment der Stretched Loop die Annahme getroffen wird, daß `i` gerade ist. Die Zugriffe, die entlang des Ausführungspfades das fixierte `a[k+4]` referenzieren, sind in der Stretched Loop rot eingefärbt. Die grün eingefärbten Array-Referenzen sind Referenzen einer anderen Kongruenzklasse $[a, 2 \times i]$, die mit $[a, 1 \times i]$ interferieren. Zwischen beiden Kongruenzklassen gibt es wechselseitige Beeinflussungen, denn `a[2*i]` vernichtet den Wert von `a[i+4]` zwei bzw. vier Iterationen nach dessen Definition im ersten Segment.

Beispiel 4.3.1 *Schleife und Stretched Loop (in Anlehnung an [6])*

```
for(i = 0; i < N; i++)
{
   x = a[i+2];
   y = a[i];
   if (odd(i)) {
      a[i+3] = a[i+3] + 1;
   }
   else {
      a[2*i] = r+s;
   }
   a[i+4] = u;
}
```


0
i = k
a[i+2], USE
a[i], USE
a[2*i], USE
a[i+3], USE
a[i+3], DEF
a[i+4], DEF
1
i = k + 1
a[i+2], USE
a[i], USE
a[2*i], USE
a[i+3], USE
a[i+3], DEF
a[i+4], DEF
2
i = k + 2
a[i+2], USE
a[i], USE
a[2*i], USE
a[i+3], USE
a[i+3], DEF
a[i+4], DEF
3
i = k + 3
a[i+2], USE
a[i], USE
a[2*i], USE
a[i+3], USE
a[i+3], DEF
a[i+4], DEF
4
i = k + 4
a[i+2], USE
a[i], USE
a[2*i], USE
a[i+3], USE
a[i+3], DEF
a[i+4], DEF

Stretched Loops dienen zwei verschiedenen Zwecken. Zum einen werden sie – wie in Beispiel 4.3.1 – als graphisches Instrument zur Veranschaulichung des Datenflusses einer Kongruenzklasse eingesetzt. Zum anderen bilden sie die Grundlage des folgenden Analyseverfahrens.[12] Bei der Datenflußanalyse reicht es aus, das Verhalten der Referenzen einer Kongruenzklasse über die Dauer der Stretched Loop hinweg zu analysieren, da die Lebensdauer eines in einer Kongruenzklasse fixierten Array-Elementes auf die Stretched Loop begrenzt ist. Array-Referenzen außerhalb der Stretched Loop liegen auch außerhalb der Lebensdauer des Array-Elements und können dieses nicht mehr beeinflussen.

Der grobe Ablauf der gesamten Datenflußanalyse gliedert sich in folgende Schritte, die anschließend detailliert beschrieben werden:

1. Berechnung der Iterationsdistanzen (z.B. nach Kap. 4.2)
2. Bestimmung der Mengen G und K – der erzeugenden und vernichtenden Referenzen
3. Bestimmung der Transferfunktionen
4. Initialisierung und Propagierung der Datenflußwerte zur Lösung des Datenflußproblems

4.3.3 Verwendeter Datenflußverband und Operatoren

Bei der *Stretched Loop*-Datenflußanalyse kommen zwei Verbände zum Einsatz. Der erste wird zur Initialisierung und während der Lösung des Datenflußgleichungssystems verwendet, der zweite dient zur Darstellung der erzielten Lösung. Im einzelnen handelt es sich dabei um die wie folgt definierten Verbände:

Lösungsverband

Der Verband zur Darstellung der Datenfluß-Lösung ist der von skalaren Analysen bekannte binäre Verband mit der Trägermenge $(\bot, \top)$ und der Ordnung $\bot < \top$. Der *Meet*-Operator $\sqcap$ ist je nach Parametrisierung als min oder max definiert. Wie zuvor wird die Minimum-Funktion min bei *Must*-Problemen verwendet, die Maximum-Funktion max bei *May*-Problemen. Der Wert $\top$ bezeichnet – wie gewöhnlich – die Gültigkeit einer Eigenschaft bei einer bestimmten Datenflußanalyse, $\bot$ hingegen die Ungültigkeit.

$$L_l = ([\bot, \top], \bot, \top, \bot < \top, min, max) \tag{4.25}$$

[12] Auch wenn die Verwendung *Stretched Loops* in Implementierungen möglich ist, so wird doch in [6] ein Weg zur Implementierung gezeigt, der aus Gründen der Effizienz nicht explizit mit diesen teilweise abgerollten Schleifen arbeitet.

Hilfsverband

Als Datenfluß-Verband wird während der Analyse der Kettenverband mit der Menge $(\bot, Cond, \top)$ und der Ordnung $\bot < Cond < \top$ verwendet. Die Operatoren sind wie oben definiert, die Bedeutungen von $\bot$ und $\top$ auch. Dem zusätzlichen Element $Cond$ kommt eine besondere Bedeutung zu. Es drückt aus, daß an einem Knoten n_i die gleiche Lösung gültig ist wie am Startknoten $start_i$ der i-ten Iteration der *Stretched Loop*, ganz unabhängig davon wie der konkrete Wert bei $start_i$ auch ausgeprägt ist.

$$L_h = ([\bot, Cond, \top], \bot, \top, \bot < Cond < \top, min, max) \tag{4.26}$$

Beispiel 4.3.2 *Verwendung des Hilfsverbandes*

Ausdruck $x + y$ am Startknoten verfügbar:	$Cond$
`b = 2 * x;`	$Cond$
`x = 3;`	$Cond$
`c = y + 3;`	$\bot$

Im Beispiel 4.3.2 ist ein kurzes Code-Fragment dargestellt, für dessen Startknoten bedingt gilt, daß der Ausdruck $x + y$ verfügbar ist. Das Erreichen der Ausdrucks $x + y$ bei den Statements `b = 2 * x` und `x = 3` ist genauso bedingt wie das Erreichen des Ausdrucks am Startknoten. Erreicht $x + y$ den Startknoten nicht, so erreicht $x + y$ auch die beiden folgenden Knoten nicht. Der Ausdruck erreicht `c = y + 3` mit Sicherheit nicht, da zwischenzeitlich die beteiligte Variable `x` redefiniert wird.

Obwohl zwei verschiedene Verbände benutzt werden, ist keine explizite Umwandlung nach Beendigung der Datenfluß-Analyse erforderlich. Wie später gezeigt wird, entfällt das Element $Cond$ im letzten Schritt des Verfahren von selbst, so daß die Lösung durch den Lösungsverband repräsentiert wird.

4.3.4 Bestimmung von G und K

Die Mengen G und K der erzeugenden und vernichtenden Referenzen werden vorab für jede Kongruenzklasse C separat ermittelt, es ergeben sich für C G^C und K^C. Ob eine Referenz in G^C oder K^C enthalten ist, hängt von dem speziellen Datenflußproblem und damit der Parametrisierung der Analyse ab. Beispielsweise enthält G^C für ein *Must-availability*-Problem alle Gebräuche und Definitionen aus C, während K^C nur die Definitionen aus C umfaßt. Ob eine Referenz erzeugend oder vernichtend in der gestreckten Schleife ist, hängt aber auch von dem Segment der gestreckten Schleife ab, in der diese Referenz ausgeführt wird. Für jede Iteration i der gestreckten Schleife eigene Mengen G_i^C und K_i^C verwaltet. Diese werden durch *Spezialisierung* aus G^C und K^C durch Algorithmus 4.3.1 gewonnen.

Im Beispiel 4.3.1 bedeutet dies folgendes. Zwar ist jede Referenz in jedem Segment der Stretched Loop vertreten, ob eine Referenz aber erzeugend oder vernichtend ist, hängt vom Ausführungspfad ab. Die Referenz `a[i+4]` erscheint nur aktiv im ersten Segment, da sie das fixierte Element `a[k+4]` nur für $i = k$ referenziert. In den folgenden Segmenten $k = i + 1, \ldots$ referenziert `a[i+4]` andere Array-Elemente als `a[k+4]`. Somit gehört `a[i+4]` ausschließlich zu G_1^C. In den folgenden Segmenten ist `a[i+4]` weder erzeugend noch vernichtend. Ebenso wird für die übrigen Referenzen bestimmt, in welchem Segment sie Wirkung haben, d.h. das fixierte Element `a[k+4]` referenzieren. Gehört eine Referenz aus $[a, 1 \times i]$ zu den erzeugenden Referenzen eines Segments der Stretched Loop, so ist sie im Beispiel rot gefärbt.

Algorithmus 4.3.1 *Bestimmung von G_i^C und K_i^C aus G^C und K^C*

procedure Specialize

    für alle Referenzen $r \in G^C$
        für alle $i \in positions^C(r)$
            füge r zu G_i^C hinzu;

    für alle Referenzen $r \in K^C$
        für alle $i \in positions^C(r)$
            füge r zu K_i^C hinzu;

function $positions^C(r)$

    if $r \in G^C$ then $P \leftarrow \{it(r)\}$
    else für alle $s \in C$
            if $k_r^i \neq \infty$ then $P \leftarrow P \cup \{it(s) + k_r^i\}$
    return P;

Specialize geht alle Referenzen r durch, um r für alle Iterationen i, in denen r vorkommt, zur entsprechenden Menge der Iteration i hinzuzufügen. Diese Vorkommnisse von Referenzen r bestimmt die Funktion *positions*, denn mit Vorkommnisse sind nicht textuelle Vorkommnisse im Schleifenkörper gemeint, sondern die Nummern der Segmente in einer symbolischen Evaluation der Stretched Loop.

In *positions*[13] wird danach unterschieden, ob es sich um eine erzeugende oder vernichtende Referenz handelt. Erzeugende Referenzen werden dem Segment zugeordnet, in dem sie bei der symbolischen Auswertung auf das in der Stretched Loop wiederkehrend adressierte Element zugreifen. Die Iterationsnummer dieses Segments wird durch eine Funktion $it(r)$ berechnet.

[13] Die Darstellung in 4.3.1 gibt eine gegenüber der Original-Publikation [6] korrigierte Fassung der Funktion *positions* an.

Auf die Realisation der Funktion $it(r)$ wird in [6] nicht näher eingegangen. Sie könnte durch abstrakte Interpretation erfolgen. Für das Beispiel 4.3.1 würde das bedeuten, daß von der Annahme „$i = k$ ist gerade" ausgegangen wird. Damit kann die Verzweigungsbedingung `odd(i)` ausgewertet werden, der Ausführungspfad nimmt den Verlauf durch den `else`-Zweig. Nach dem Übergang in die nächste Iteration muß `i` ungerade sein, denn `i` wird um eins inkrementiert. So wird diesmal der `then`-Zweig gewählt. Wenn durch diese symbolische Auswertung der Ausführungspfad durch alle Segmente der Stretched Loop feststeht – oder sicher approximiert werden kann, dann können auch diejenigen Segmente bestimmt werden, in denen eine Referenz wiederkehrend auf ein Array-Element verweist. Im Beispiel sind alle Referenzen erzeugende Referenzen. Daher würde it für die Kongruenzklasse $[a, 1 \times i]$ die Segmentnummern der rot eingefärbten Referenzen zurückgeben.

Vernichtende Referenzen können in mehreren Segmenten erscheinen, da sie den Fluß des kongruenten Wertes in verschiedenen Distanzen vom Ort der Erzeugung vernichten können. Somit werden vernichtende Referenzen in allen[14] Segmenten plaziert, in denen eine vernichtende Referenz die Lebensdauer einer erzeugenden Referenz aus C beendet. Die Segmente in denen eine vernichtende Referenz vorkommt, berechnen sich aus der Iterationsnummer $it(s)$ der Erzeugung addiert zu der Vernichtungsdistanz k_r^s. k_r^s ist die kürzeste Distanz in der die Referenz s von einer Referenz r vernichtet wird. Somit bleibt ein von s erzeugter Wert für mindestens $k_r^s - 1$ Iterationen erhalten und wird nicht von r vernichtet. Das ist aber genau der Wert der mit der δ-Technik in 4.2 bestimmten Iterationsdistanzen, die an dieser Stelle eingesetzt werden können.

4.3.5 Transferfunktionen

Jedem Knoten[15] des Kontrollflußgraphen der Stretched Loop wird eine Transferfunktion zugeordnet, die die Veränderung der Datenfluß-Information beim Passieren dieses Knotens beschreibt. Für den Knoten n_i des Stretched Loop-Segments i wird bzgl. der Kongruenzklasse C eine Transferfunktion $f_{n_i}^C$ gebildet.

Definition 4.3.5 *Transferfunktion* $f_{n_i}^C$

function $f_{n_i}^C(sol)$
    if n_i *enthält* $(r \in G_i^C)$
        return $\top$
    else if n_i *enthält* $(r \in K_i^C)$
        return $\bot$
    else return sol

[14] In [6] wird gezeigt, daß es für *Must*-Probleme ausreichend ist, vernichtende Referenzen r in dem der Erzeugung s nächstliegendem Ort in Datenflußrichtung zu plazieren.

[15] Ein Knoten kann eine Instruktion oder auch einen Basisblock enthalten. Das vorgestellte Modell setzt allerdings voraus, daß pro Knoten nur eine Array-Referenz erfolgt, somit soll davon ausgegangen werden, daß ein Knoten eine Instruktion umfaßt.

Nach Def. 4.3.5 gibt $f_{n_i}^C$ $\top$ zurück, falls in n_i eine erzeugende Referenz liegt, bzw. $\bot$, falls in n_i eine vernichtende Referenz liegt. Ansonsten durchläuft die eintreffende Datenflußinformation den Knoten unverändert.

4.3.6 Datenflußanalyse

Das Datenflußanalyse-Verfahren soll anhand der Stretched Loop erklärt werden, wobei es nicht notwendig ist, den mehrfach aneinandergereihten Schleifenkörper auch tatsächlich mehrfach zu durchlaufen. In der Original-Publikation [6] wird eine Implementation vorgestellt, die mittels des Einsatzes von Bitvektoren und dazugehörigen Operationen, nicht darauf angewiesen ist, die Schleife partiell abzurollen. Deshalb kann eine solche Implementation eine hohe Effizienz erzielen.

Sei $d + 1$ nach Def. 4.3.3 die Länge der *Stretched Loop*, die Schleife ist also $d + 1$-fach abgerollt. Jedem Knoten der gestreckten Schleife ist ein Element des „Hilfsverbandes“ zugeordnet. $start_i$ bezeichne den Startknoten der i-ten Iteration der gestreckten Schleife. Der Algorithmus durchläuft nacheinander folgende Schritte:

1. Zur *Initialisierung* wird der Wert aller $d+1$ Startknoten auf $Cond$ gesetzt. Unabhängig voneinander werden in allen $d + 1$ Iterationen der Stretched Loop die Startwerte zu allen davon abhängigen Knoten propagiert. Anschließend tragen alle Knoten n_i, deren Lösung vom Wert am Startknoten $start_i$ abhängig ist, den Wert $Cond$. Da alle Iterationen der Stretched Loop in diesem Schritt voneinander unabhängig sind, kann die Bearbeitung parallel erfolgen.

   Das Beispiel 4.3.3 zeigt eine mögliche Initialisierung und Propagierung für die Stretched Loop aus Bsp. 4.3.1. Bei anderen Datenflußanalysen können andere Ergebnisse entstehen. Es erfolgt kein Informationstransfer über die rot eingezeichneten Iterationsübergänge hinweg.

2. Im zweiten Schritt werden lediglich die $start_i$-Knoten betrachtet, um deren korrekte Datenfluß-Werte zu bestimmen. Dazu werden ausgehend von einem Initialwerte $\bot$ für $start_0$ der Wert $Cond$ durch den Wert ersetzt, der den Startknoten der vorherigen Iteration erreicht hat. Damit haben die Startknoten ihre endgültige (und korrekte) Lösung erreicht.

   Die schraffierten Flächen deuten in Beispiel 4.3.4, daß das „Innenleben“ der Schleifenkörper in der Stretched Loop keine Rolle spielt. Wichtig sind nur die Startknoten zu jedem Segment. Die Werte der Startknoten werden vom vorherigen Segment übernommen, falls der aktuelle Wert $Cond$ ist. In diesem Schritt werden Informationen über Iterationsgrenzen hinweg transportiert, jedoch werden die Schleifenkörper nicht einbezogen.

3. Im dritten Schritt werden wiederum alle $d + 1$ Iterationen unabhängig voneinander bearbeitet. In jeder Iteration i werden alle Knoten n_i untersucht, und falls auf ein Wert $Cond$ gestoßen wird, so wird dieser durch den

entsprechenden Wert von $start_i$ ersetzt. Letztendlich hat jeder Knoten, von dem im ersten Schritt ermittelt wurde, daß er den gleichen Lösungswert hat wie der Startknoten, auch den im zweiten Schritt ermittelten wahren Wert des Startknoten angenommen.

Ausgehend von den Startwerten aus Schritt 2 und dem Wissen aus Schritt 1, welche Knoten die gleichen Werte wie ihre Startknoten des Segments haben, können die korrekten Werte für alle Knoten in Beispiel 4.3.5 ermittelt werden. Parallel für alle Segmente läßt sich Schritt 3 durchführen, da zwischen den Segmenten keine Information ausgetauscht wird.

Bemerkenswert an diesem Algorithmus ist, daß lediglich zweimal (in Schritt 1 und 3) alle Knoten besucht werden, und dabei jeweils $d+1$ Iterationen voneinander unabhängig bearbeitet werden können. In Schritt 3 werden auch alle $Cond$-Werte ersetzt, so daß der Hilfsverband implizit in den Lösungsverband konvertiert wird. Dazu ist keine eigene Phase erforderlich. Algorithmus 4.3.2 zeigt den formalen Ablauf der Datenflußanalyse.

Algorithmus 4.3.2 *Stretched Loop-Datenfluß-Analyse*

Schritt 1: *Initialisierung und bedingte Lösung*

$IN^C[start][0\ldots d] \leftarrow (Cond,\ldots,Cond)$
Traversiere in Reverse Postorder, *für jeden Knoten n*
  for i = 0 to d do
    $IN^C[n][i] \leftarrow \bigwedge_{m\in pred(n)} OUT^C[m][i]$
    $OUT^C[n][i] \leftarrow f^C_{n_i}(IN^C[n][i])$

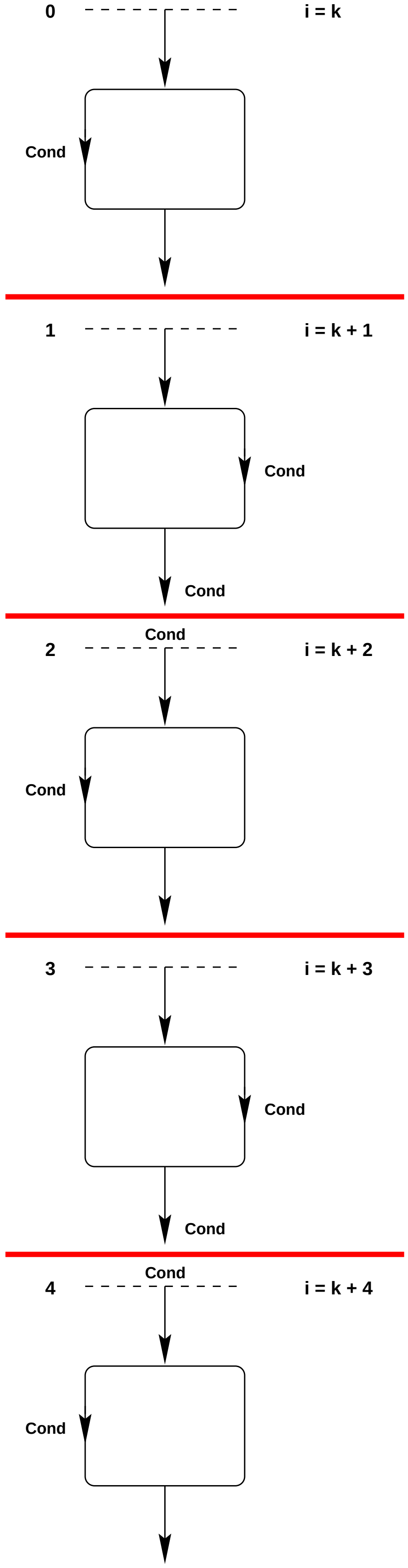
Schritt 2: *Lösung für die Startwerte*

$IN^C[start][0] \leftarrow \perp$
for i = 1 to d do
  if $IN^C[start][i] = Cond$ *then*
    $IN^C[start][i] \leftarrow IN^C[start][i-1]$

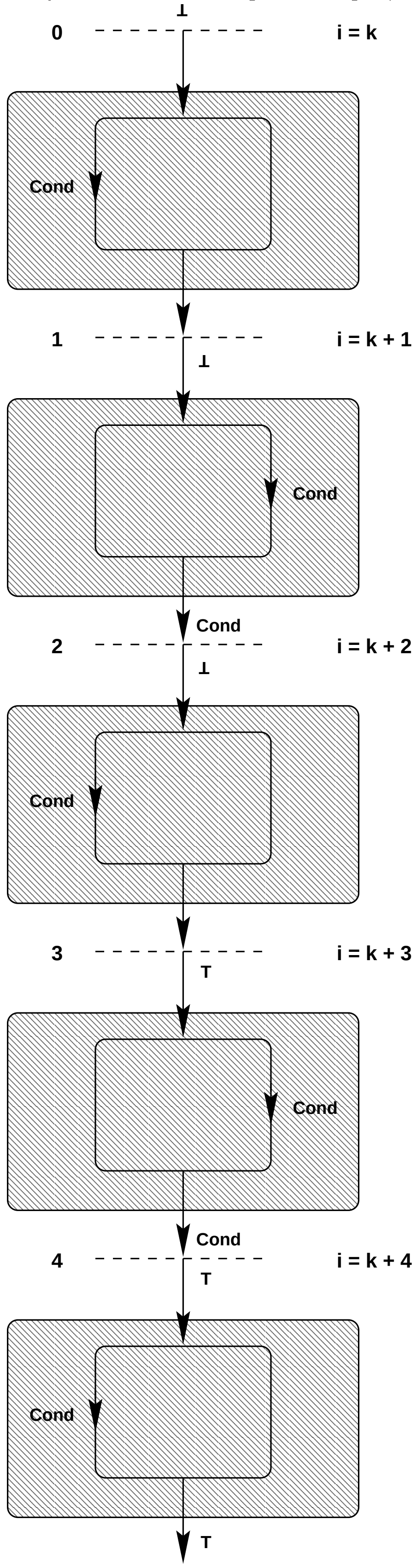
Schritt 3: *Auflösung der bedingten Lösung*

für alle Knoten n
  for i = 0 to d do
    if $IN^C[n][i] = Cond$ *then*
      $IN^C[n][i] \leftarrow IN^C[start][i]$
    if $OUT^C[n][i] = Cond$ *then*
      $OUT^C[n][i] \leftarrow IN^C[start][i]$

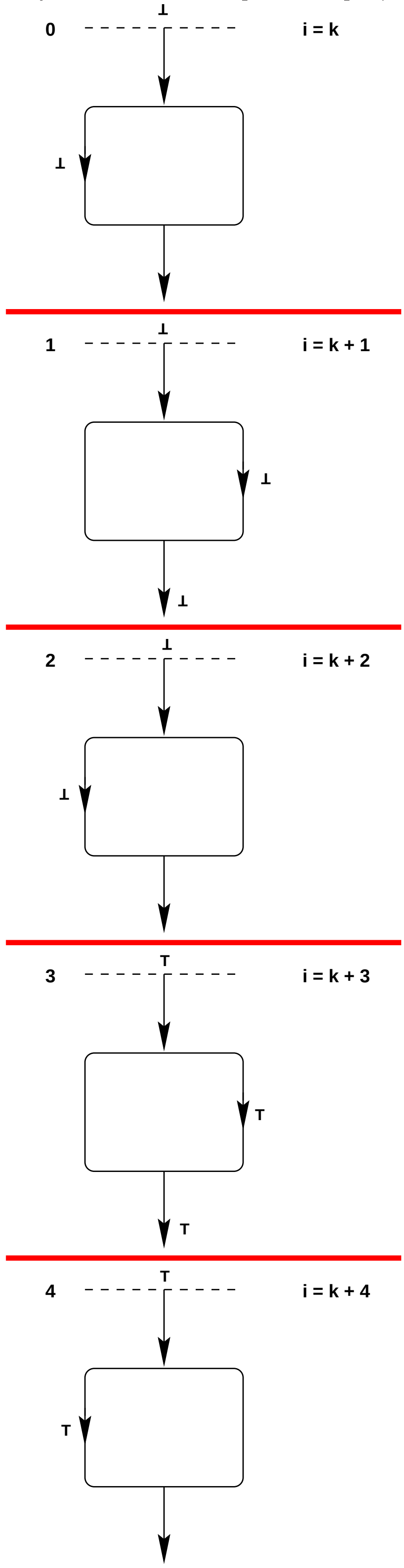
Beispiel 4.3.3 *Schritt 1 für Stretched Loop aus Bsp. 4.3.1*



Beispiel 4.3.4 *Schritt 2 für Stretched Loop aus Bsp. 4.3.1*



Beispiel 4.3.5 *Schritt 3 für Stretched Loop aus Bsp. 4.3.1*



Die Lösung des Datenflußproblems wird an den Knoten des ursprünglichen Schleifenkörpers dargestellt. Dabei bekommt jeder Knoten einen Vektor zugeordnet, der so viele Elemente enthält wie die Stretched Loop Segmente hat. Jedes Element des Lösungsvektors steht dabei für die Lösung im entsprechenden Segment.

4.3.7 Parametrisierung

Ebenso wie die δ-Technik ist auch das *Stretched Loop*-Verfahren parametrisierbar und damit an die Bearbeitung vieler verschiedener Datenflußprobleme anzupassen. Das allgemeine Verfahren ist in der Lage, zu Vorwärts- und Rückwärtsproblemen Must- und May-Information zu bestimmen. Die Anpassung an einzelne Problemklassen wird in den folgenden Abschnitten erläutert.

Vorwärts/Rückwärts

Die Richtung des Datenflusses kann durch die Richtung der Kanten des Kontrollflußgraphen vorgegeben werden. Bei einem Vorwärtsproblem arbeitet die Analyse in Richtung des Kontrollflusses auf der Stretched Loop, bei einem Rückwärtsproblem entgegengesetzt der Richtung des Kontrollflusses.

May/Must

Das bislang vorgestellte Modell ist für die Verwendung bei *Must*-Problemen gedacht. Für *May*-Information sind die dualen Verbände zu verwenden, d.h. die Bedeutungen von $\perp$ und $\top$ sind wie ihre Anordnungen vertauscht, für den $\sqcap$-Operator die duale Operation zu verwenden.

Spezielles Datenflußproblem

Die Festlegung auf ein spezielles Datenflußproblem erfolgt neben der Wahl der Datenflußrichtung (Vorwärts, Rückwärts) und der Informationsqualität (Must, May) durch die Definition der Mengen G und K. Dabei muß festgelegt werden, welche Referenzen einer Kongruenzklasse C für ein spezielles Problem als erzeugend oder vernichtend anzusehen sind. Entsprechend dieser Festlegung werden die Mengen G^C und K^C gebildet, die im Verlauf der Analyse zur Berechnung der Lösung herangezogen werden.

Beispiele verschiedener Parametrisierungen finden sich bei den im Kapitel 6.2 vorgestellten Optimierungen. Dabei wird auch geklärt, wie von einer Problemstellung zu der Festlegung der einzelnen Parameter zu gelangen ist.

4.3.8 Ermöglichte Optimierungen

Die Ergebnisse der *Stretched Loop*-Analyse ermöglichen eine ganze Reihe von Optimierungen. Zu nennen ist eine Variante des *Register Pipelinings*, deren Plazierung von Load-, Store- und Registerkopier-Operationen die Anzahl dieser Operationen minimiert. Auch werden redundante und tote Operationen durch *Partial Redundancy Elimination* und *Partial Dead Code Elimination* vermieden (siehe Kapitel 6.2). Für das *Oprimale Register Pipelining* müssen verschiedene spezielle Datenflußprobleme gelöst werden, die allesamt mit der vorgestellten Analyse behandelt werden können.

Zur Unterstützung des *Software Pipelinings* können falsche Abhängigkeiten erkannt und beseitigt werden, die Zyklen im zugehörigen Abhängigkeitsgraph bilden. Damit können die Schedules der Software Pipeline verkürzt werden, was zu gesteigerter Effizienz während der Ausführung einer so behandelten Schleife führt(siehe 7.2).

4.3.9 Vor- und Nachteile

Der größte Vorteil der *Stretched Loop*-Array-Datenflußanalyse gegenüber dem δ-Verfahren ist die gesteigerte Präzision. Die höhere Genauigkeit der Datenfluß-Lösung kann zu besseren Optimierungsgüten oder bislang nicht möglichen Optimierungen genutzt werden.

Der Aufwand zur Implementierung des Verfahrens liegt beträchtlich über dem der δ-Technik. Die δ-Datenflußanalyse wird zudem zusätzlich benutzt, da deren Ergebnisse in die Berechnungen des Stretched Loop-Verfahrens eingehen. Der Algorithmus selbst ist trotz des Aufwandes sehr effizient, denn er braucht nur drei Durchläufe durch die Stretched Loop. Mit N Knoten im Schleifenkörper und der Länge d_{max} der längsten Stretched Loop wird eine Laufzeitkomplexität von $O(N + d_{max})$ für die drei Schritte der Datenflußanalyse erreicht. Der Aufwand für die Funktion *it* ist dabei noch nicht enthalten. Bei einer vorgeschlagenen Implementierung der drei Schritte des Verfahrens durch Bitvektoren können sehr günstige Laufzeiten erzielt werden.

Nachteilig am Verfahren ist die weiterhin fehlende Möglichkeit zur Behandlung von Schleifen mit nicht-affinen Ausdrücken. Treten solche innerhalb einer Schleife auf, so kann die gesamte Schleife nicht analysiert werden. Trotz der gesteigerten Präzision handelt es sich immer noch um ein Approximationsverfahren, das keine exakte Bestimmung aller Array-Datenabhängigkeiten vornimmt. Während der Stretched Loop-Analyse wird auf eine Funktion *it* zurückgegriffen, deren Realisierung dem Anwender überlassen ist. Die Güte der Ergebnisse dieser Funktion beeinflußt die Gesamtgüte des Verfahrens, ebenso wie deren Laufzeit die Gesamtlaufzeit bestimmt.

4.4 *Lazy*-Verfahren zur Array-Datenflußanalyse

Mit dem *Lazy*-Verfahren soll eine Methode zur Array-Datenflußanalyse vorgestellt werden, die den bisherigen Einschränkungen auf affine Ausdrücke nicht unterworfen ist. Weiterhin ist das Verfahren ist in der Lage, für affine Ausdrücke, Datenflußbhängigkeiten *exakt* zu berechnen, und für nicht-affine Ausdrücke gute Approximationen zu liefern.

Idee des Algorithmus ist, von einer *Read*-Instruktion ausgehend, zunächst die im Iterationsraum naheliegenden *Write*-Instruktionen zu analysieren, und dann nach und nach die Suche auf größere Distanzen auszuweiten, so daß neben der hohen Präzision auch eine große Effizienz gewährleistet ist.

4.4.1 Voraussetzungen

Voraussetzungen zur Anwendung des exakten Lazy-Verfahrens sind affine Programmfragmente, d.h. ein Loop Nest mit affinen Indexfunktionen, Verzweigungsbedingungen und Schleifengrenzen. In affinen Ausdrücken sind neben Induktionsvariablen nur symbolische Konstanten zulässig.

Für das Approximationsverfahren, welches auch mit nicht-affinen Ausdrücken umgehen kann, werden strukturierte Programmfragmente *ohne* GOTO, BREAK und WHILE-Instruktionen gefordert. Zum einen soll damit die *Single entry/Single exit*-Eigenschaft gewahrt werden, und zum anderen muß im vorhinein der Iterationsbereich feststehen, was bei WHILE-Konstrukten nicht a priori gegeben ist.

4.4.2 Definitionen und Notation

An dieser Stelle sollen zunächst einige Definitionen und eine formale Notation eingeführt werden, auf die im Verlauf zurückgegriffen wird.

Symbolische Konstante: Eine symbolische Konstante ist eine Variable, der im *gesamten* betrachteten Programmfragment kein Wert zugewiesen wird. Das beinhaltet, daß es sich bei einer symbolischen Konstanten auch um keine Induktionsvariable handeln kann. Später wird zur Behandlung nicht-affiner Ausdrücke diese Definition etwas allgemeiner gefaßt, aber bis dahin reicht die hier angeführte Definition.

In Beispiel 4.4.1 ist `k` eine symbolische Konstante, wenn die Schleife der analysierte Programmausschnitt ist und im Schleifenkörper keine weitere Definition von `k` auftritt.

Beispiel 4.4.1 *Symbolische Konstanten*

```
k = c+d;
for(i = 0; i < N; i++)
{
   x = a[i+k];
   ...
}
```

Instruktion: Das Lazy-Verfahren arbeitet auf der Ebene der Instruktionen. Eine einzelne Instruktion wird durch W oder R gekennzeichnet – üblicherweise danach unterschieden, ob es sich um eine Definition oder einen Gebrauch handelt. Einzelne Array-Referenzen in Instruktionen, die mehrere Referenzen enthalten können, werden fortlaufend mit $A, B, \ldots$ bezeichnet. Um eine bestimmte Array-Referenz $A, B, \ldots$ in einer Instruktion R oder W darzustellen, wird diese durch $R.A$ oder $W.B$ gekennzeichnet. Instruktionen, die sich innerhalb eines Schleifenkörpers befinden, sind von den Induktionsvariablen der umgebenden Schleifen und von symbolischen Konstanten abhängig. Um eine bestimmte Instanz einer Instruktion W zu kennzeichnen, kann eine Instruktion mit einem Satz an Induktionsvariablen $\overrightarrow{w}$ und symbolischen Konstanten $\overrightarrow{s}$ versehen werden: $W[\overrightarrow{w}, \overrightarrow{s}]$.

Beispiel 4.4.2 *Instruktionen*

```
k = c+d;
for(i = 0; i < k; i++)
{
   x = a[i] + a[i+1];
   ...
   a[i] = y;
}
```

Die Beispiel 4.4.2 enthält zwei Instruktionen mit drei Array-Referenzen. Die erste Instruktion `x = a[i] + a[i+1]` wird mit R bezeichnet, die zweite Instruktion `a[i] = y` mit W. Die erste Array-Referenz `a[i]` der ersten Instruktion ist $R.A$, die zweite Array-Referenz `a[i+1]` ist $R.B$. R und W sind von `i` und `k` abhängig. Somit können einzelne Instanzen der Instruktionen mit $R[i, k]$ und $W[i, k]$ für feste i und k bezeichnet werden.

Ausführungsreihenfolge: Einzelne Instanzen von Instruktionen werden vor anderen Instanzen ausgeführt. Die Ausführungsreihenfolge ist abhängig von den Induktionsvariablen $\overrightarrow{w}$, $\overrightarrow{r}$ und von den symbolischen Konstanten $\overrightarrow{s}$. Die Instruktion $W[\overrightarrow{w}, \overrightarrow{s}]$ wird vor $R[\overrightarrow{r}, \overrightarrow{s}]$ ausgeführt, wenn $\overrightarrow{w}$ vor $\overrightarrow{r}$ im Iterationsraum liegt.

Für das Beispiel 4.4.2 gilt, daß $W[3, 10]$ vor $R[5, 10]$ ausgeführt wird, denn $i = 3$ liegt im Iterationsraum $I = [0, \ldots, k = 10]$ vor $i = 5$.

Formal

$$W[\overrightarrow{w}, \overrightarrow{s}] \lhd R[\overrightarrow{r}, \overrightarrow{s}] \Leftrightarrow$$
$$\overrightarrow{w}[1\ldots n] \ll \overrightarrow{r}[1\ldots n] \quad \vee \quad (\overrightarrow{w}[1\ldots n] = \overrightarrow{r}[1\ldots n] \quad \wedge \quad W \lhd R)$$

Darin bezeichnet $\overrightarrow{w}[1\ldots n]$ den Vektor, der sich durch die Elemente $1\ldots n$ des Vektors $\overrightarrow{w}$ ergibt (Projektion), und $W \lhd R$, daß W vor R im Programmtext erscheint. Die Operation $\overrightarrow{w} \ll \overrightarrow{r}$ bestimmt, ob der Vektor $\overrightarrow{w}$ lexikographisch kleiner ist als $\overrightarrow{r}$.

Als nächstes wird die Erstellung von *Abhängigkeitsrelationen* für affine Programmfragmente gezeigt. Erst danach werden Erweiterungen zur Approximation nicht-affiner Programmfragmente aufgezeigt.

4.4.3 Darstellung von Datenabhängigkeiten

Ziel ist es, ein formales Mittel zu erhalten, um Datenabhängigkeiten darzustellen. Vielfach verwendete funktionale Darstellungen sind an dieser Stelle nicht geeignet, so daß letztendlich Relationen zum Einsatz kommen.

Zur Repräsentation von Datenabhängigkeiten gibt es verschiedene Möglichkeiten (vgl. [13]). Es lassen sich *exakte* und *nicht-exakte* Darstellungen unterscheiden. Eine Möglichkeit, Gebräuche und Definitionen von Array-Elementen in Verbindung zu stellen, ist die Verwendung von *source functions*. Sie bilden eine bestimmte Instanz einer *Read*-Instruktion $R[\overrightarrow{r}]$ auf eine bestimmte *Write*-Operation $W[\overrightarrow{w}]$ ab. Auch zur Repräsentation von source functions gibt es verschiedene Alternativen, jedoch ist schon mit dem Konzept der Funktion ein entscheidender Nachteil verbunden. Bei nicht-affinen Approximationen treten Situationen auf, bei denen ein *Read* in Abhängigkeitsverbindung zu mehreren *Writes* gestellt wird. Der eindeutige, funktionale Zusammenhang zwischen Gebrauch und Definition geht verloren. Die Lösung dieses Problems liegt in der Verwendung von *Abhängigkeitsrelationen*, die in der Lage sind, mehrere Referenzen zueinander in Verbindung zu bringen.

Wenn ein Tupel bestehend aus einer Definition $W[\overrightarrow{w}]$ und einem Gebrauch $R[\overrightarrow{r}]$ Element der Abhängigkeitsrelation R sind, dann wird dadurch die Abhängigkeit von $R[\overrightarrow{r}]$ von $W[\overrightarrow{w}]$ ausgedrückt (symbolisch $W[\overrightarrow{w}] \rightarrow R[\overrightarrow{r}] \in R$).

Beispiel 4.4.3 zeigt ein Loop Nest mit mehreren Instruktionen und Array-Referenzen. Für die Instanzen von `xl` aus der `k`-Schleife gelten folgende Abhängigkeitsrelationen:

$$S_1[i,j] \rightarrow S_{16}[i,j,1] | 1 \leq i \leq NMOL1 \wedge i+1 \leq j \leq NMOL$$
$$\ldots$$
$$S_{14}[i,j] \rightarrow S_{16}[i,j,14] | 1 \leq i \leq NMOL1 \wedge i+1 \leq j \leq NMOL$$

Die Abhängigkeiten in der Schleife können durch die Vereinigung der gezeigten Relationen dargestellt werden. Wichtig ist, daß neben den zusammengehörigen Paaren von Statements (inkl. Induktionsvariablen) auch die Einschränkungen $(1 \leq i \leq NMOL1 \wedge \ldots)$ zu den Relationen gehören.

Beispiel 4.4.3 *Abhängigkeitsrelationen zu gegebener Schleife (Quelle [12])*

```
for(i = 1; i <= NMOL1; i++)
   for(j = i+1; j <= NMOL; j++)
   {
      xl[1] = xma - xmb;         /* S1  */
      xl[2] = xma - xb[1];       /* S2  */
      xl[3] = xma - xb[3];       /* S3  */
      ...
      xl[12] = xa[2] - xb[3];    /* S12 */
      xl[13] = xa[1] - xb[2];    /* S13 */
      xl[14] = xa[3] - ab[2];    /* S14 */
      for(k = 1; k <= 14; k++)
         xl[k] = xl[k] - ...     /* S16 */
   }
```

4.4.4 Abhängigkeitsrelationen

Aufgabe des Lazy-Verfahrens ist die Generierung einer Abhängigkeitsrelation $DepRel$, die die von einer lesenden Referenz $R.A$ stammenden Abhängigkeiten vereint. Wie in Beispiel 4.4.3 schon zu sehen war, kann die Abhängigkeitsrelation $DepRel$ aus der Vereinigung mehrerer einfacher Relationen bestehen.

Allgemein hat $DepRel$ folgendes Aussehen:

$$DepRel = \begin{cases} W_1[\vec{w}, \vec{s}] \to R.A[\vec{r}, \vec{s}] | DepRel_1(\vec{w}, \vec{r}, \vec{s}) \\ \ldots \\ W_m[\vec{w}, \vec{s}] \to R.A[\vec{r}, \vec{s}] | DepRel_m(\vec{w}, \vec{r}, \vec{s}) \end{cases} \tag{4.27}$$

$DepRel_i$ ist die Konjunktion der jeweiligen Einschränkungen. Für alle Einschränkungen zusammen muß gelten:

$$\bigcup_{i=1}^{m} \pi_{\vec{r}, \vec{s}}(DepRel_i(\vec{w}, \vec{r}, \vec{s})) \subseteq [R, \vec{s}] \tag{4.28}$$

In (4.28) ist $[R, \vec{s}]$ die Menge der Iterationsvektoren (Werte der umgebenden Induktionsvariablen), für die die Instruktion R ausgeführt wird, wenn die symbolischen Konstanten $\vec{s}$ vorgegeben werden. Alle Einschränkungen $DepRel_i$

zu einzelnen Teil-Abhängigkeitsrelationen zusammengenommen können keinen größeren Iterationsbereich umfassen. Daher ist die Vereinigung der Projektionen $\pi_{\vec{r},\vec{s}}$ aller Einschränkungen der Abhängigkeitsrelationen auf die Induktionsvariablen und symbolischen Konstanten eine Teilmenge aller überhaupt ausgeführten Iterationen.

Jede einzelne Abhängigkeitsrelation aus (4.27) steht für eine Abhängigkeit des $R.A$-Statements von einem W-Statement in einem durch $DepRel_i$ festgelegten Bereich des Iterationsraums. Alle zusammen bestimmen die Abhängigkeiten über den gesamten Iterationsraum.

Die Abhängigkeitsrelationen können durch folgendes Vorgehen konstruiert werden. Als Quelle der Abhängigkeit zu einer Referenz $R.A$ kommen alle diejenigen Definitionen $W.B$ in Frage, die das gleiche Array schreiben ($Arr(W.B) = Arr(R.A)$). Dann müssen die Indexfunktionen von $R.A$ und $W.B$ bei vorgegebenen Induktionsvariablen und symbolischen Konstanten gleiche Werte erzeugen ($W.B[\vec{w}, \vec{s}] = R.A[\vec{r}, \vec{s}]$). $\vec{w}$ bzw. $\vec{r}$ müssen dabei Werte haben, die auch bei vorgegebenen symbolischen Konstanten $\vec{s}$ angenommen werden können ($\vec{w} \in [W, \vec{s}]$ bzw. $\vec{r} \in [R, \vec{s}]$). Die bisherigen Bedingungen sind noch nicht ausreichend. Es fehlt noch, daß der Schreibzugriff *vor* dem Lesezugriff stattfindet ($W[\vec{w}, \vec{s}] \lhd R[\vec{r}, \vec{s}]$). Von allen gefundenen W-Instruktionen soll für die *DepRel*-Relation diejenige ausgewählt werden, die zur R-Instruktion die größte lexikalische Entfernung hat ($max_{\ll}$).

Alle Forderungen zusammen führen zu:

$$\begin{aligned}
&\forall \vec{r}, \vec{s} : (V[\vec{v}, \vec{s}] \to R.A[\vec{r}, \vec{s}]) \in DepRel(\vec{w}, \vec{r}, \vec{s}) \Leftrightarrow \\
&V[\vec{v}, \vec{s}] = max_{\ll}(W[\vec{w}, \vec{s}] | \vec{w} \in [W, \vec{s}] \wedge \vec{r} \in [R, \vec{s}] \wedge \\
&Arr(W.B) = Arr(R.A) \wedge W.B(\vec{w}, \vec{s}) = R.A(\vec{r}, \vec{s}) \wedge \\
&W[\vec{w}, \vec{s}] \lhd R[\vec{r}, \vec{s}])
\end{aligned} \tag{4.29}$$

Die Berechnung des lexikographischen Maximums[16] führt zur „zerteilten“ Darstellung der *DepRel*-Relation durch Teilrelationen aus (4.27).

Eine Methode zur *effizienten* Berechnung der Abhängigkeitsrelationen wird im nächsten Abschnitt durch einen *Lazy*-Algorithmus vorgestellt.

4.4.5 Algorithmus zur Lazy-Array-Datenflußanalyse

Zur Bestimmung der Abhängigkeitsrelationen kann der Iterationsraum in beliebiger Richtung bearbeitet werden, doch eine willkürliche Richtung der Suche ist oft ineffizient. Die Beobachtung, daß zusammengehörige *Read*- und *Write*-Operationen oft nicht weit auseinander liegen, d.h. Datenabhängigkeiten über

[16] Ein Algorithmus zur Berechnung von $max_{\ll}$ wird in der Original-Publikation [12] vorgestellt. An dieser Stelle reicht es aus zu wissen, daß ein Vektor $\vec{w}$ lexikographisch kleiner ist als $\vec{r}$ ($\vec{w} \ll \vec{r}$), wenn gilt $(w_1 < r_1) \vee (w_1 = r_1 \wedge w_2 < r_2) \vee (w_1 = r_1 \wedge w_2 = r_2 \wedge w_3 < r_3) \vee \ldots$.

eine geringe Anzahl an Iterationen wahrscheinlicher sind als über große Iterationsdistanzen, macht ein systematisches Vorgehen nach folgender Weise sinnvoll. Um ausgehend von einem *Read* dessen Abhängigkeiten zu analysieren, wird mit lexikographisch nahe liegenden *Writes* die Suche im Iterationsraum begonnen. Ist die Suche nicht schon nach kurzer Zeit erfolgreich, so wird sie nach und nach ausgeweitet. Dabei wird darüber Buch geführt, welche Gebräuche bereits „überdeckt“ sind. Die Suche kann abgebrochen werden, wenn alle *Reads* erfüllt wurden oder keine weiteren *Writes* mehr zur Untersuchung anstehen. Diese Art von Algorithmen, die endgültige Entscheidungen über noch offene Probleme solange hinauszögern bis sie unumgänglich sind, heißen auch *Lazy*-Verfahren, daher auch die Bezeichnung für die vorliegende Datenflußanalyse.

Der folgende Algorithmus[17] berechnet zu einem Gebrauch $R.A$, der von n Schleifen mit den Induktionsvariablen $\overrightarrow{r} = (r_1, \ldots, r_n)$ umgeben ist, und den symbolischen Konstanten $\overrightarrow{s}$ die Abhängigkeitsrelation *DepRel* nach (4.29).

Algorithmus 4.4.1 *Bestimmung der Abhängigkeitsrelationen DepRel*

Relation $DepRel = \{\}$;
DNF $NotCovered(\overrightarrow{r}, \overrightarrow{s}) = IsExecuted(R[\overrightarrow{r}, \overrightarrow{s}])$;
Integer $FixLoops = n$;
Statement $W = R$;

While (*NotCovered erfüllbar*) *do*
  $W =$ *Vorgänger-Instruktion von* W;
  if (*W ist Zuweisung && W definiert* $Arr(R.A)$) *then*
    DNF $SameCell(\overrightarrow{w}, \overrightarrow{r}, \overrightarrow{s}) =$ *„nicht-abgedeckte „Writes“, die ausgeführt werden und gleiche Werte der Indexfunktion berechnen;*
    Konjunktion $W_{sub}(\overrightarrow{w}, \overrightarrow{r}) =$ *Lexikographisch nahe liegende W-Kandidaten;*
    DNF $DepProb(\overrightarrow{w}, \overrightarrow{r}, \overrightarrow{s}) = SameCell(\overrightarrow{w}, \overrightarrow{r}, \overrightarrow{s}) \wedge W_{sub}(\overrightarrow{w}, \overrightarrow{r})$;
    Relation $C_{max} =$ *Lexikographisches Maximum von* $DepProb(\overrightarrow{w}, \overrightarrow{r}, \overrightarrow{s})$;
    $DepRel = DepRel \cup C_{max}$;
    $NotCovered = NotCovered$ *abzüglich durch* C_{max} *abgedeckter „Reads“;*
  Else If (W ist Schleifenende oder -anfang) then
    If (W ist Schleifenanfang) then
      $FixLoops = FixLoops - 1$;
      $W =$ *Schleifenende zu W;*
    EndIf;
  Else If (W ist Funktionsanfang) then
    $DepRel = DepRel \cup \{Entry \rightarrow R.A[\overrightarrow{r}, \overrightarrow{s}] | NotCovered(\overrightarrow{r}, \overrightarrow{s})\}$;
    While-Schleife verlassen;
  EndIf;
EndDo;

return(DepRel);

[17] An dieser Stelle wird „nur“ ein grober Abriß des Algorithmus dargestellt. Eine detaillierte Version befindet sich in der Original-Publikation.

DepRel wird mit der leeren Menge initialisiert. Die noch nicht „abgedeckten“ *Reads*, d.h. diejenigen ohne zugehöriges *Write*, umfassen alle Punkte des Iterationsraums, die sich durch die Indexfunktion des zu analysierenden *Reads* ergeben. Die nicht abgedeckten *Reads* werden in dieser Implementation als Disjunktive Normalform (DNF) ihrer Iterationsvektoren und symbolischen Konstanten realisiert[18]. Für Vektoren, die zur Menge $NotCovered$ gehören, berechnet sich der Wahrheitswert zu „wahr“, ansonsten zu „falsch“. Desweiteren werden zu Beginn alle n umgebenden Schleifen *fixiert.* Eine Schleife, die fixiert ist, ist nicht an *loop-carried dependences* beteiligt.

Anschließend werden solange Vorgänger von R untersucht, bis alle *Read*-Instanzen abgedeckt sind, oder der Anfang der zu untersuchenden Funktion erreicht ist, und somit keine weiteren Kandidaten mehr übrig bleiben.

Wird im Laufe der Untersuchung auf eine Array-Definition gestoßen, so wird überprüft, ob das gleiche Element wie in dem *Read* referenziert wird. Von den lexikographisch nahe liegenden Definitionen wird das Maximum bestimmt, d.h. diejenige Definition mit dem größten Abstand. Diese wird der Abhängigkeitsrelation hinzugefügt, und das *Read* ist abgedeckt.

Bei Erreichen eines Schleifenendes wird die Fixierung der nächsten umgebenden Schleife gelöst, um nach loop carried Abhängigkeiten aus dieser Schleife zu suchen. Wird dagegen der Beginn des untersuchten Programmfragments erreicht, so terminiert der Algorithmus.

Der Algorithmus ist aufgrund seiner Suchstrategie effizient, es soll aber nicht verschwiegen werden, daß er dennoch exponentielle Laufzeit erreichen kann. Das Teilproblem der Bestimmung des lexikographischen Maximums ist NP-vollständig. In der Praxis sollte aber ein günstiges Laufzeitverhalten zu erwarten sein.

4.4.6 Erweiterungen zur Behandlung nicht-affiner Programmfragmente

Nicht-affine Ausdrücke verhindern nicht per se ihre Analyse, sondern sie können im eingeschränkten Maße exakt und darüberhinaus approximativ behandelt werden. Ausschnitte nicht-affiner Programmfragmente, deren Abhängigkeitsanalyse „offensichtlich“ ist, finden sich in Beispiel 4.4.4.

Beispiel 4.4.4,a) mit der nicht-affinen Bedingung zeigt, daß das *Read* von den beiden Definitionen in beiden Zweigen der Verzweigung abgedeckt wird, unabhängig vom Typ der verwendeten Verzweigungsbedingung. Mit der Kenntnis, daß die entdeckte Abhängigkeit loop independent ist, können weitere Untersuchungen zu loop carried dependences ausgehend von diesem Gebrauch eingestellt werden. Mit der gewonnenen Information kann der lesende Zugriff auf

[18] Die Wahl der DNF zur Darstellung begründet sich durch die Verwendung des Omega-Tests zur Vereinfachung der Einschränkungen. Der Omega-Test verlangt dazu die DNF.

`a[j]` eliminiert werden, wenn die zuvor geschriebenen Werte in Registern bis zum Gebrauch transportiert werden.

Das Beispiel 4.4.4,b) demonstriert, daß ein nicht-affiner Ausdruck in einer Indexfunktion nicht zwangsläufig dazu führen muß, eine Referenz mit einer solchen Indexfunktion als abhängig von *allen* anderen Referenzen zu klassifizieren. Es liegt eine loop independent Abhängigkeit vor, denn das *Read* `a[x]` wird durch das vorherige *Write* `a[x]` vollständig abgedeckt. Wie auch im ersten Beispiel kann ein redundanter, lesender Speicherzugriff eliminiert werden.

Beispiel 4.4.4 *Programmfragmente mit nicht-affinen Ausdrücken (Quelle [12])*

a) Nicht-affine Bedingung

```
for(i = 1; i < n; i++)
   for(j = 1; j < m; j++)
   {
      x = f(i,j);
      if (x)
      {
         a[j] = ... /* S1 */
      }
      else
      {
         a[j] = ... /* S2 */
      }
      ... = a[j];   /* S3 */
   }
```

b) Nicht-affine Indexfunktion

```
for(i = 1; i < n; i++)
   for(j = 1; j < m; j++)
   {
      x = f(i,j);
      a[x] = ...;   /* S1 */
      ... = a[x];   /* S2 */
   }
```

Zur Erweiterung der Lazy-Datenflußanalyse zur Behandlung nicht-affiner Ausdrücke müssen einige Änderungen durchgeführt werden, nicht nur in der Implementation, sondern auch an Definitionen und verwendeten Konzepte. Als erstes werden für *symbolische Konstanten* und *Programmfragmente* modifizierte Definitionen benötigt.

Ein *Programmfragment* ist nun nicht mehr eine ganze Funktion oder ein Loop Nest als ganzes, sondern es wird *dynamisch* definiert. Zu einer Instruktion S wird eine *unfixierte Zone* $UnFixed(S, d)$ der Tiefe d betrachtet, zu der die d innersten Schleifen eines Loop Nest um S herum gehören. Zur *fixierten Zone* $Fixed(S, d)$ zählen dann alle Instruktionen, die nicht zu $UnFixed(S, d)$ gehören. Die unfixierte Zone ist für $d = 0$ leer, und alle Instruktionen sind in $Fixed(S, d)$ enthalten.

Basierend auf fixierten/unfixierten Zonen können auch *symbolische Konstanten* neu definiert werden. Wenn eine skalare Variable v in einer fixierten Zone $Fixed(S, d)$ definiert wird, so ist aus Sicht der Instruktion S die Variable v eine symbolische Konstante, $v \in SymConst(S, d)$. Zur Bestimmung der symbolischen Konstanten ist es notwendig, daß eindeutige Zuordnungen zwischen

Gebräuchen und Definitionen der Variablen geschaffen werden. Mittels der in Kapitel 3.5 vorgestellten SSA-Form ist diese Aufgabe zu bewältigen. Die „neuen“ symbolischen Konstanten gehen in Algorithmus 4.4.1 an der Stelle ein, wo ausgeführte, bislang nicht abgedeckte *Writes* ermittelt werden. Mit der modifizierten Definition können äußere Induktionsvariable nun auch als symbolische Konstanten angesehen werden, wenn ihre Definition außerhalb der unfixierten Zone – den inneren, bisher untersuchten Schleifen – liegt.

Beispiel 4.4.5 *Fixierte/unfixierte Zonen und symbolische Variable*

```
for(i = 0; i < N; i++)
{
   for(j = 0; j < M; j++)
   {
      x = y;
      for(k = 0; k < L; k++)
      {
         a = b; /* S */
      }
   }
}
```

Im Beispiel 4.4.5 besteht $UnFixed(S,1)$ aus der `k`-Schleife. Die `i`-Schleife und die tt j-Schleife mit der Instruktion `x = y` gehören zu $Fixed(S,1)$. Da die Variable `x` in der fixierten Zone $Fixed(S,1)$ zu S definiert wird, ist `x` aus der Sicht von S eine symbolische Konstante.

Das weitere Vorgehen bei nicht-affinen Ausdrücken gliedert sich wie folgt:

1. *Berechnung der oberen Schranke des Iterationsraumes.* Der Iterationsraum wird so erweitert, daß nicht-affine Einschränkungen bei den Relationen entfallen. Dazu werden Verzweigungsbedingungen mit konkreten Wahrheitswerten belegt. Es werden aber nicht die Bedingungen an sich, sondern die sich daraus ergebenden Einschränkungen für die Relationen betrachtet (z.B. $1 < i < x(i,j)$ für eine Bedingung `x(i,j)` in einer Verzweigung). Diese Einschränkungen sind Konjunktionen oder Disjunktionen verschiedener Einzelbedingungen, die zusammen einen Gültigkeitsbereich der Abhängigkeit im Iterationsraum beschreiben. Positive Literale in nicht-affinen Booleschen Ausdrücken werden durch „True“ ersetzt, negative Literale durch „False“. Es resultiert eine obere Schranke für den Iterationsraum.

2. *Berechnung der oberen und unteren Schranke der Abhängigkeiten.* Je mehr Schleifen aus der Fixierung gelöst werden, desto mehr nicht-affine Ausdrücke erscheinen in den Abhängigkeitsrelationen. Neu hinzugekommene nicht-affine Variablen werden durch „True“ oder „False“ abgeschätzt, so

daß sowohl eine obere als auch eine untere Schranke möglicher Abhängigkeiten entstehen. Zur Berechnung der unteren Schranke (Unterschätzung der Abhängigkeiten) werden positive Literale durch „False" ersetzt und negative Literale durch „True". Analog werden für die obere Schranke (Überschätzung der Abhängigkeiten) positive Literale durch „True" und negative Literale durch „False" ausgewertet.

Damit die Ausführungen nicht so abstrakt und unanschaulich bleiben, soll auf das Beispiel 4.4.4 zurückgegriffen werden.

Beispiel 4.4.6 *Behandlung nicht-affiner Ausdrücke*

Nicht-affine Bedingung zu Bsp. 4.4.4,a)

Für die Read-*Operation* `a[j]` *werden zunächst* `i` *und* `j` *fixiert.*
Damit ist $UnFixed(S,d) = \{\}$. Somit liegt auch die Definition von `x` *in einer fixierten Zone, und* `x` *ist eine symbolische Konstante.*

Folgende Abhängigkeiten können ermittelt werden:

$$S_1[i,j] \rightarrow S_3[i,j] | 1 \leq i,j \leq n \wedge x \tag{4.30}$$
$$S_2[i,j] \rightarrow S_3[i,j] | 1 \leq i,j \leq n \wedge \neg x \tag{4.31}$$

Die obere Schranke des Iterationsraums ist:

$$S_1[i,j] \rightarrow S_3[i,j] | 1 \leq i,j \leq n \tag{4.32}$$

Annahme: *Wäre* `x` keine *symbolische Konstante, so würde gelten:*

$$S_1[i,j] \rightarrow S_3[i,j] | 1 \leq i,j \leq n \wedge x(i,j) \tag{4.33}$$
$$S_2[i,j] \rightarrow S_3[i,j] | 1 \leq i,j \leq n \wedge \neg x(i,j) \tag{4.34}$$

Die obere Schranke der Abhängigkeiten würde sich ergeben als[19]*:*

$$\begin{cases} S_1[i,j] \rightarrow S_3[i,j] | 1 \leq i,j \leq n \\ S_2[i,j] \rightarrow S_3[i,j] | 1 \leq i,j \leq n \end{cases} \tag{4.35}$$

Die untere Abhängigkeitsschranke führt zur leeren Relation.

[19] An dieser Stelle kommen die Vorteile der Abhängigkeitsrelation gegenüber der source function zum Tragen. Die gezeigten Verbindungen sind nicht mehr eindeutig.

Beispiel 4.4.7 *Behandlung nicht-affiner Ausdrücke*

Nicht-affine Indexfunktion zu Bsp. 4.4.4,b)

Auch hier werden zuerst `i` *und* `j` *fixiert, so daß die Variable* `x` *zur symbolischen Konstante wird. Die Einschränkung der Relation sieht so aus:*

$$1 \leq i_w = i_r, j_w = j_r \leq n \wedge x = x \tag{4.36}$$

Daraus resultiert sofort die Abhängigkeitsrelation:

$$S_1[i,j] \rightarrow S_2[i,j] | 1 \leq i, j \leq n \tag{4.37}$$

In diesem Fall geht trotz nicht-affiner Ausdrücke (und Approximationsverfahren) keine Präzision verloren.

Anschließend können die oberen und unteren Schranken bei der approximativen Lösung folgender Situationen verwendet werden.

Nicht-affine Abhängigkeiten werden durch die obere Schranke der Abhängigkeiten abgeschätzt. Damit wird die Korrektheit im Sinne einer *sicheren* Approximation gewahrt. Auch wird nur eine *minimale* Anzahl an Abhängigkeiten (irrtümlich) hinzugefügt, und zwar diejenigen, die zur Erzeugung einer affinen Abhängigkeitsrelation dienen.

Die von einem *Write* abgedeckten Instruktionen werden durch die untere Abhängigkeitsschranke approximiert. Auch hier gilt es, sichere Abschätzungen zu liefern. Daher ist eine Unterschätzung der wahren Lösung der einzige sinnvolle Weg. Die Abhängigkeiten der durch die unteren Schranke bestimmten Menge existieren mit Sicherheit, alles darüberhinausgehende kann nicht garantiert werden.

Das lexikographische Maximum $max_{\ll}$ kann nicht für Relationen mit affinen Approximationen in deren Einschränkungen berechnet werden. Die darin enthaltene Information ist zu unpräzise, um zu bestimmen, welche Instanzen von Instruktionen tatsächlich ausgeführt werden. Sicherheitshalber werden *alle* Instanzen als abhängig klassifiziert.

4.4.7 Vor- und Nachteile

Die großen Vorteile der *Lazy*-Datenflußanalyse sind ihre Präzision und die Möglichkeit zur Behandlung nicht-affiner Ausdrücke. Für affine Programmfragmente werden die exakten Array-Datenflußabhängigkeitsrelationen bestimmt, und für nicht-affine Programmfragmente gute Approximationslösungen. Somit entziehen sich auch letztgenannte Programmabschnitte nicht mehr der Analyse und Optimierung.

Nachteilig wirkt sich die hohe Komplexität des Verfahrens aus. Nicht nur die Implementierung ist sehr aufwendig, sondern auch die Laufzeit kann möglicherweise viel Zeit beanspruchen. Zwar wird durch die *Lazy*-Strategie gesichert, daß häufig sehr effizient gearbeitet wird, doch in ungünstigen Fällen kann der Algorithmus exponentielle Laufzeit annehmen. Das *Lazy*-Verfahren ist nicht parametrisierbar. Somit können nur Optimierungen unterstützt werden, die genau die Datenflußabhängigkeiten benötigen. Das Verfahren ist nicht geeignet, Anti-, Input- oder Output dependences zu bestimmen. Die in späteren Kapiteln vorgestellten Redundanzeliminationen verlangen jedoch häufig gerade diese Informationen.

4.5 *DSA*-Verfahren zur Array-Datenflußanalyse

Die bisher vorgestellten Verfahren zur Array-Datenflußanalyse konnten entweder keine nicht-affinen Ausdrücke behandeln (δ-Technik, Stretched Loop) oder waren mit zu großem Aufwand verbunden (Lazy-Verfahren). Wünschenswert ist ein Verfahren, daß affine und nicht-affine Ausdrücke gestattet, dabei parametrisierbar und effizient ist, und auf beliebigen Kontrollflußgraphen arbeitet. Die Approximationsgüte für nicht-affine Programmabschnitte braucht nicht sehr gut zu sein, denn es reicht aus, Optimierung für die affinen Array-Referenzen zu ermöglichen.

Als ein Array-Datenflußanalyse-Verfahren, das diesen Forderungen nahekommt, soll das *Dynamic Single Assignment (DSA)*-Verfahren von Rau [17] vorgestellt werden. Es hebt sich schon dadurch von den anderen Verfahren ab, weil es auf einer anderen IR arbeitet. Der Grund dafür liegt darin, daß konventionelle IR zum einen nicht in der Lage sind, in einem Programm vorhandene Parallelität auf Instruktionsebene auszudrücken und zum anderen Array-Variablen vernachlässigen, so daß deren Analyse erschwert wird. Daher fordert Rau von einer für ILP geeigneten IR:

> „An intermediate representation for ILP must provide the ability
>
> - to explicitly and precisely represent the dependences between operations (including those between subscripted memory references) in the presence of arbitrary control flow graphs, especially cyclic ones, and
> - to express the program in a maximally parallel form (i.e., a minimum of antidependences and output dependences), whether or not the parallelism is explicit, while controlling the number of copy operations that are introduced as a result eliminating the anti- and output dependences.“

Diese Forderungen werden von der DSA-IR erfüllt, die im nächsten Abschnitt näher erklärt wird. Erst anschließend soll die eigentliche Datenflußanalyse besprochen werden.

Bei der Anwendung der DSA-Datenflußanalyse sind zwei Voraussetzungen zu beachten, die allerdings nicht zwingend[20] sind. Zum einen sollten die Array-Referenzen zur Erreichung hoher Präzision affin sein. Nicht-affine Indexfunktionen führen zu einer (groben) Klassifizierung der betreffenden Array-Referenz als abhängig, da deren Unabhängigkeit nicht nachweisbar ist. Zudem müssen alle Indexfunktionen von der gleichen Induktionsvariablen abhängig sein. Zum anderen darf die Induktionsvariable nur um einen konstanten Betrag verändert werden, das allerdings auch mehrfach innerhalb einer Schleife.

4.5.1 Dynamic Single Assignment

Zur Erfüllung der Forderung der maximalen Parallelität in der Darstellung eines Programms ist es notwendig, die Datenabhängigkeiten, die nicht die Semantik eines Programms betreffen, zu eliminieren. Von den im Kapitel 3.2 vorgestellten Datenabhängigkeiten, ist die *true dependence* die einzige, die semantisch wirklich bedeutend ist, während die übrigen nur bei der parallelen Ausführung von Instruktionen Einfluß haben. Diese weniger bedeutsamen Abhängigkeiten lassen sich zwar dadurch verhindern, daß einer Variablen nur einmal ein Wert zugewiesen wird, wie z.B. bei der Static Single Assignment-Form (siehe Kapitel 3.5), doch auch die SSA-IR kann nicht verhindern, daß z.B. in Schleifen *dynamisch*, d.h. während der Programmausführung, eine Variable mehrfach definiert wird.

In Beispiel 4.5.1 werden dynamisch auch in der SSA mehrfach Zuweisungen an `t06` und `t10 - t13` unternommen, jeweils eine pro Iteration. Es entstehen Anti-Abhängigkeiten zwischen einzelnen Operationen (z.B. s05, s07). Wegen der mehrfachen (dynamischen) Zuweisungen sind einzelne Definitionen nicht mehr zu unterscheiden. Zur Umgehung dieser Schwierigkeiten wird eine IR benötigt, die Single Assignment auch zur Laufzeit, also dynamisch, unterstützt.

Beispiel 4.5.1 *Schleife in C, konventioneller IR und SSA-Form*

Schleife in C:

```
k = 1;
for(i=1; i < 100; i++)
   k = k + 2;
```

[20] Wie diese Voraussetzungen umgangen werden können, wird in dieser Diplomarbeit nicht dargelegt. Bei Bedarf kann die Original-Publikation [17] daraufhin gelesen werden, die in diesem Punkt aber auch nicht sehr ausgiebig ist.

Schleife in konv. IR:

```
% t01=1
% t02=2
% t03=100

s01: t04=copy(t01)
s02: t05=copy(t01)

s03: t06=igeq(t05,t03)
s04:     brt(t06,s08)
s05: t04=iadd(t04,t02)
s06: t05=iadd(t05,t01)
s07: jmp s03
s08: ...
```

Schleife in SSA-Form:

```
% t01=1
% t02=2
% t03=100

s01: t04=copy(t01)
s02: t05=copy(t01)

s03: t06=igeq(t05,t03)
s04:     brt(t06,s10)
s05: t10=φ(t04,t12)
s06: t11=φ(t05,t13)
s07: t12=iadd(t10,t02)
s08: t13=iadd(t11,t01)
s09: jmp s05
s10: ...
```

Definition 4.5.1 *(nach [17]) Ein Programm erfüllt die* Dynamic Single Assignment (DSA)*-Eigenschaft, wenn allen virtuellen Registern (temporären Variablen) auf allen (dynamischen) Ausführungspfaden maximal einmal ein Wert zugewiesen wird.*

Dynamic Single Assignment hat zwar mit Static Single Assignment die Eigenschaft gemeinsam, daß eine Variable nur *einmal* definiert werden darf, aber es sind doch grundlegend verschiedene Eigenschaften. Ein Programm in SSA-Form kann aufgrund einer in Schleifeniterationen mehrfach ausgeführten Zuweisung die DSA-Eigenschaft verletzen. Umgekehrt ist es möglich, daß ein Programm mit der DSA-Eigenschaft mehrfache (statische) Definitionen einer Variablen enthält, wenn sichergestellt ist, daß (dynamisch) nur eine davon ausgeführt werden kann.

Einfache virtuelle Register (temporäre Variable) reichen im weiteren nicht mehr aus und müssen erweitert werden. Hinzu kommt eine neue Operation auf diesen erweiterten, virtuellen Registern:

Definition 4.5.2 *(nach [17]) Ein* erweitertes, virtuelles Register (EVR) *ist eine unendliche, linear geordnete Menge virtueller Register mit einer darauf arbeitenden Operation* remap(t). *Die Elemente eines EVR t können durch $t[n], n \in \mathbf{N}_0^+$ adressiert, gelesen oder geschrieben werden. Abkürzend wird $t[0]$ auch mit t bezeichnet. remap(t) bewirkt einen Shift der Inhalte von t, so daß sich der Inhalt von $t[n], \forall n$ anschließend in $t[n+1]$ befindet.*

Zur Transformation eines Programms aus einer gewöhnlichen IR in die DSA-Form werden einige Schritte benötigt. Diese werden im Anschluß an die Erklärung an der Schleife in Beispiel 4.5.2 exemplarisch angewendet.

1. Zunächst wird jeder Definition eines virtuellen Registers t eine $remap(t)$-Operation vorangestellt. Damit wird der bisherige Inhalt von $t[0]$ nach $t[1]$, usw. kopiert, so daß die folgende Zuweisung an t dynamisch ein anderes Element ist als es t zuvor war. Selbst bei wiederholter Ausführung bleibt die DSA-Eigenschaft erhalten, da jeweils ein anderes Element erzeugt wird. Wird in einer Instruktion t sowohl gelesen als auch geschrieben, so muß die Instruktion dahingehend geändert werden, daß $t[1]$ gelesen und t geschrieben wird.

2. Alle *remap*-Operationen in einem Schleifenkörper werden an den Beginn desselben vorgezogen. Wenn es durch das Vorziehen der *remap*-Operationen dazu kommt, daß ein Gebrauch eines virtuellen Registers t, der zuvor *vor* der Operation $remap(t)$ gestanden hat, nun *danach* steht, so ist der Gebrauch durch $t[1]$ zu ersetzen. Auch dieser Schritt ist im Grunde klar, da durch das vorgezogene *remap* der referenzierte Inhalt nun nicht mehr in $t[0]$ steht, sondern in $t[1]$. Somit ist der Gebrauch der veränderten Position im EVR anzupassen.

3. Letztendlich können *Copy Propagation* und *Dead Code Elimination* durchgeführt werden, um auszunutzen, daß verschiedene Elemente von EVRs nach Kopieroperationen gleiche Werte tragen. Dadurch kann es dazu kommen, daß auch *remap*-Operationen überflüssig werden und entfallen können.

Beispiel 4.5.2 *Schleife in C und konventioneller IR*

C-Code:

```
k = 0;
for(i = 0; i < 100; i++)
   k = k + 1;
```

Konventionelle IR:

```
%   t00 = 0, t01 = 1, t02 = 100

s00 t03 = copy(t00)
s01 t04 = copy(t00)

s10 t03 = iadd(t03,t04)
s11 t04 = iadd(t04,t01)
s12 t05 = ile(t04,t02)
          brt(t05,s10)
```

Beispiel 4.5.3 zeigt die Schleife nach dem Einfügen von *remap*-Operationen vor jeder Definition. Da in *s11* und *s13* gleiche EVR (*t03* und *t04*) geschrieben und gelesen werden, wird der lesende Zugriff ersetzt (*t03[1]* und *t04[1]*).

Beispiel 4.5.3 *Schleife nach aus Beispiel 4.5.2 nach Schritt 1*

```
%   t00 = 0, t01 = 1, t02 = 100

s00 t03 = copy(t00)
s01 t04 = copy(t00)

s10 remap(t03)
s11 t03 = iadd(t03[1],t04)
s12 remap(t04)
s13 t04 = iadd(t04[1],t01)
s14 remap(t05)
s15 t05 = ile(t04,t02)
        brt(t05,s10)
```

In Beispiel 4.5.4 wurde *remap*(t04) vor den Gebrauch in *s13* gezogen. Daher wurde *t04* durch *t04[1]* ersetzt. Alle *remap*-Operationen befinden sich nun am Schleifenanfang.

Beispiel 4.5.4 *Schleife nach aus Beispiel 4.5.2 nach Schritt 2*

```
%   t00 = 0, t01 = 1, t02 = 100

s00 t03 = copy(t00)
s01 t04 = copy(t00)

s10 remap(t03)
s11 remap(t04)
s12 remap(t05)
s13 t03 = iadd(t03[1],t04[1])
s14 t04 = iadd(t04[1],t01)
s15 t05 = ile(t04,t02)
        brt(t05,s10)
```

Das verwendete Beispiel 4.5.4 enthält keine Gelegenheit zur Anwendung von Copy Propagation und Dead Code Elimination. Es findet keine Veränderung statt.

4.5.2 DSA-Datenflußanalyse

Die Idee, die hinter der DSA-basierten Array-Datenflußanalyse steckt, sieht so aus: Wenn von einem Array-Element $a[i]$ bekannt ist, daß es an einer Stelle im Programm verfügbar ist, dann ist nach einer Instruktion $i = i + 1$ stattdessen das Element $a[i-1]$ verfügbar, denn es handelt sich um das gleiche Element.

Statt die Gültigkeit von $a[i]$ zu vernichten, kann das Element nun mit einer neuen Bezeichnung weiterexistieren.

Zur DSA-Array-Datenflußanalyse sind zwei wesentliche Schritte notwendig. Zuerst werden die speicherbasierten Abhängigkeiten bestimmt, danach unter Verwendung der gewonnenen Information die wertebasierten Datenabhängigkeiten. Die nächste Abschnitt beschreibt erst das Verfahren zur Ermittlung der speicherbasierten Abhängigkeiten, bevor dann die wertebasierte DSA-Datenflußanalyse vorgestellt wird.

Speicherbasierte Abhängigkeitsanalyse

Zur Untersuchung werden alle Array-Referenzen herangezogen, die innerhalb eines Abschnitts, in dem die Induktionsvariable nicht verändert wird, das gleiche Array – und damit möglicherweise das gleiche Array-Element – referenzieren. Sei also i die Induktionsvariable mit dem Iterationsintervall I und seien $f_1(i) = a_1 \times i + b_1$ und $f_2(i) = a_2 \times i + b_2$ die Indexfunktionen zweier zu vergleichender Array-Referenzen. Die Betrachtung der Referenzen führt zu fünf Kategorien bzgl. ihrer Abhängigkeit:

1. Falls $a_1 - a_2 = 0 \quad \wedge \quad b_1 - b_2 \neq 0$ oder $a_1 - a_2 \neq 0 \quad \wedge \quad (b_1 - b_2) \not\equiv 0 \quad mod \quad (a_1 - a_2)$ oder $a_1 - a_2 \neq 0 \quad \wedge \quad (b_1 - b_2) \equiv 0 \quad mod \quad (a_1 - a_2) \quad \wedge \quad \frac{b_1 - b_2}{a_1 - a_2} \notin I$, dann sind $a[a_1 \times i + b_1]$ und $a[a_2 \times i + b_2]$ *niemals gleich*. (Vgl. Bsp. 5.1.7 und 5.1.8).

2. Falls $a_1 - a_2 = 0 \quad \wedge \quad b_2 - b_1 = 0$, so handelt es sich um textuell gleiche Referenzen, die *immer gleich* sind.

3. Falls $a_1 - a_2 \neq 0 \quad \wedge \quad (b_1 - b_2) \equiv 0 \quad mod \quad (a_1 - a_2) \quad \wedge \quad \frac{b_1 - b_2}{a_1 - a_2} \in I$, dann sind die betrachteten Referenzen *wiederkehrend gleich*.

4. Falls $f_1(i) = f_2(i)$ nur für *ein* $i \in I$ gezeigt werden kann, dann sind die Referenzen *kurzzeitig gleich*.

5. Ansonsten ist die Unabhängigkeit der Referenzen *nicht nachweisbar*. Dies kann sowohl dann der Fall sein, wenn es sich um verschiedene Arrays handelt, die aber ein Alias untereinander bilden können, oder bei Array-Referenzen mit unterschiedlichen Index-Variablen, deren gegenseitige Beziehung nicht geklärt werden kann. Ebenso können nicht-affine Indexfunktionen zu dieser Einstufung führen.

Diese fünf Kategorien werden nun danach zusammengefaßt, ob sie die gleiche Speicherstellen adressieren. Daraus ergeben sich drei Klassen:

Unterschiedlich: Nur für den Fall, daß die Indexfunktionen zweier Referenzen *niemals gleich* sind, können die durch die Referenzen adressierten Speicherstellen als *unterschiedlich* klassifiziert werden.

Identisch: Zwei Referenzen, deren Indexfunktionen *immer gleich* sind, adressieren *identische* Speicherstellen.

Möglicherweise identisch: Die verbleibenden drei Fälle führen zu Klassifizierung *möglicherweise identisch.*

Referenzen *unterschiedlicher* Speicherstellen sind voneinander *unabhängig*, aber Referenzen *identischer* Speicherstellen sind immer voneinander *abhängig.* Falls *möglicherweise identische* Speicherstellen adressiert werden, so sind die zugehörigen Referenzen *möglicherweise abhängig.*

Wertebasierte Abhängigkeitsanalyse

Wie einleitend beschrieben, reicht die Kenntnis speicherbasierter Abhängigkeiten nicht aus, um Optimierungen für ILP-Prozessoren zu unterstützen. Aber die speicherbasierten Abhängigkeiten lassen sich verwenden, um in einem weiteren Schritt wertebasierte Abhängigkeiten zu bestimmen. Ähnlich wie bei der skalaren Analyse wird eine iterative Methode angewendet, die mit Transferfunktionen Datenflußinformation von Knoten zu Knoten weiterreicht. Nach einigen Durchläufen stabilisiert sich die Information und die Lösung kann abgelesen werden. Im Unterschied zur skalaren Analyse – und auch zum δ-Verfahren und der Stretched-Loop-Technik – wird beim DSA-Verfahren kein Datenflußverband verwendet, sondern es werden explizite Zuordnungen von Array-Referenzen zu Registern (EVRs) weitergereicht.

Bei der DSA-Datenflußanalyse muß für jede Stelle eines Programms eine extensionale Abbildung zwischen Array-Referenzen und EVRs geführt werden. In *Map-Tupeln* $(X[f(i)], t[k])$ wird für einen Knoten n gespeichert, daß die Array-Referenz $X[f(i)]$ – so wie sie auch im Programmtext erscheint – am Knoten n im EVR t unter $t[k]$ verfügbar ist. Das erste Element eines Map-Tupels ist der *M-name*, das zweite Element der *R-name*. Map-Tupel werden für alle Knoten eines Kontrollflußgraphen geführt. Map-Tupels gelten dabei ausschließlich an dem Knoten, für den sie definiert sind.

Für die im weiteren beschriebene *Available expressions*-Analyse wird für jeden Knoten eine Menge S von Map-Tupeln verwaltet, deren Elemente dafür stehen, daß der Wert von $X[f(i)]$ für den betrachteten Knoten im EVR-Element $t[k]$ verfügbar ist. Je nach Knoten und dessen Instruktion wird eine der folgenden Operationen ausgeführt, um die Menge S dieses Knoten zu verwalten. Die Transferfunktionen der Analysen aus den vorherigen Abschnitten (δ-Technik, Stretched Loop-Verfahren) finden hier ihre Entsprechung in Mengenoperationen.

- Falls die Menge S ein Tupel $(X[f(i)], t[k])$ enthält und der betrachtete Knoten eine Definition beinhaltet, die einem Element, welches *identisch* oder *möglicherweise identisch* zu $X[f(i)]$ ist, einen neuen Wert zuweist, kann das Tupel aus S entfernt werden. Wird durch die Definition der Inhalt von $t'[k']$ nach $X[f'(i)]$ geschrieben, so wird $(X[f'(i)], t'[k'])$ in S aufgenommen.

- Wird ein Gebrauch ausgeführt, der $X[f(i)]$ nach $t[k]$ lädt, so wird das Tupel $(X[f(i)], t[k])$ in S aufgenommen.

- Immer wenn die Induktionsvariable verändert wird, also Knoten mit Instruktionen der Form $i = g(i)$ erreicht werden, müssen alle Tupel der Form $(X[f(i)], t[k])$ aus S, deren M-name von i abhängt, gelöscht werden. Wenn g eine Umkehrfunktion g^{-1} besitzt, werden die Tupel $(X[f(g^{-1}(i))], t[k])$ der Menge S hinzugefügt.

  Beispiel:

  1. $(a[i+5], t[1]) \in S$
  2. $i = i + 2$
  3. $(a[i+5], t[1])$ wird aus S entfernt
  4. $g(i) = i + 2$ umkehrbar $\Rightarrow g^{-1}(i) = i - 2$
  5. $f(i) = i+5$, $g^{-1}(i) = i-2 \Rightarrow f(g^{-1}(i)) = (i-2)+5 = i+3$
  6. $(a[i+3], t[1])$ wird in S eingefügt

- Immer wenn eine $remap(t)$-Instruktion verarbeitet wird, werden alle Tupel $(X[f(i)], t[k])$ mit dem R-name t durch Tupel $(X[f(i)], t[k+1])$ ersetzt.

- An Stellen zusammenlaufenden Kontrollflusses müssen auch die dort zusammenkommenden Mengen S_l mit $l = 1, \ldots, n$ zur neuen Menge S zusammengefaßt werden. Die *Meet*-Operation wird bei einer *Must*-Datenflußanalyse dadurch realisiert, daß in S nur diejenigen Tupel mit dem M-name $X[f(i)]$ aufgenommen werden, die entlang aller eintreffenden Pfade propagiert wurden, d.h. $\forall l \in (1, \ldots, n) : (X[f(i)], t_l) \in S_l$. Die *Meet*-Operation bildet also die Schnittmenge bzgl. der M-names der propagierten Datenflußinformation. Darüberhinaus sind aber auch die R-names zu beachten. Sind alle l R-names gleich, d.h. stimmen der Name des EVR und der Index überein, so kann der propagierte R-name beibehalten werden. Ansonsten wird ein neues EVR r benötigt, so daß $r[0]$ der R-name des neuen Tupels ist. Dem Knoten, an dem die Kontrollflußpfade zusammenlaufen, wird unmittelbar ein ϕ-Knoten angehängt, der die Werte zusammenführt.

Nachdem die „Transferfunktionen“ bestimmt sind, kann wie auch bei anderen Verfahren, eine Fixpunkt-Iteration durchgeführt werden. Dabei werden – wie üblich – fortlaufend die Mengen S aller Knoten verändert, bis sich die Lösung stabilisiert. Damit ist eine Fixpunkt-Lösung erreicht, anhand derer die Lösung konkreter Datenflußfragen geklärt werden kann. Die Tupel der Mengen S geben für alle Knoten an, welche Array-Elemente in EVRs gespeichert sind. Mittels der M-names kann dies schnell in einer bestimmten Menge überprüft werden. Fällt die Suche negativ aus, so ist der gesuchte Wert nicht verfügbar.

Zur Sicherstellung der Terminierung des Verfahren muß gewährleistet sein, daß die Mengen S nicht beliebig groß werden. Dies könnte bei monoton wachsenden oder sinkenden Induktionsvariablen durchaus der Fall sein, da fortlaufend neue Elemente zu den Mengen S hinzukommen, während die alten Werte – unter anderem M-name – erhalten bleiben. Um dem entgegenzuwirken, kann ein „Betrachtungsfenster" definiert werden. Dieses Fenster erlaubt nur M-names in einem bestimmten Wertebereich die Mitgliedschaft in einer Menge S. Nimmt in einem M-name eine Indexfunktion zu große oder zu kleine Werte an, so fällt das dazugehörige Tupel aus der Menge S. Damit ist die Größe der Menge S begrenzt, und auch gewährleistet, daß das Iterationsverfahren konvergiert, um zu einer Lösung zu kommen. Dabei bleibt auch die Korrektheit erhalten, denn Werte die aus S entfallen, werden (evtl. fälschlicherweise) als nicht verfügbar klassifiziert. Dieser mögliche Fehler ist aber ein „sicherer" Fehler, und somit erlaubt. Wenn das Fenster so groß wie die Distanz der Referenzen im zu analysierenden Schleifenkörper gewählt wird, dann fallen nur Elemente aus S, die ohnehin keine weitere Bedeutung erlangt hätten. Ist keine sinnvolle Fenstergröße im voraus zu bestimmen, kann auch das Iterationsverfahren dahingehend verändert werden, daß nur eine feste, vorgegebene Anzahl D an Iterationen stattfindet. Dadurch wird aber auch die Gültigkeit der so gewonnenen Information auf maximal D Iterationen begrenzt. Darüberhinaus sind alle Referenzen sicherheitshalber als voneinander abhängig anzusehen.

4.5.3 Anpassung

Die bisherigen Beschreibungen zeigen das Verfahren für einen Vorwärts-Must-Datenflußanalyse. Es sind aber auch sowohl Rückwärts- als auch May-Analysen möglich.

Für ein May-Problem braucht lediglich der *Meet*-Operator angepaßt zu werden. Anstatt einer Durchschnittsmenge muß dann die Vereinigungsmenge gebildet werden, wenn mehrere Kontrollflußpfade aufeinandertreffen. Es gibt mehrere Mengen S_l mit $l = 1, \ldots, n$, die zur neuen Menge S zusammengefaßt werden. In S befinden sich alle Tupel, die auch in irgendeiner Menge S_l sind.

Für die Behandlung von Rückwärts-Datenflußproblemen ist ein wenig mehr zu tun. Neben der Umkehrung der Arbeitsrichtung, sind die Schrite 3, 4 und 5 zu verändern.

- In 3. müssen alle Tupel $(X[f(i)], t[k])$, deren R-name ein Element aus t ist, durch Tupel $(X[f(g(i))], t[k])$ ersetzt werden, d.h. die Veränderung durch Anwendung von g muß rückgängig gemacht werden.
- In 4. müssen ebenso Veränderungen zurückgenommen werden, und zwar diejenigen, die durch $remap(t)$ entstehen. Dazu werden alle Tupel der

Form $(X[f(i)], t[k])$ in S, deren R-name durch t bestimmt wird, durch Tupel $(X[f(i)], t[k-1])$ ausgetauscht.

- In 5. sind die Änderungen auf den Kontrollflußgraphen bezogen. Statt eines ϕ-Knoten, muß ein sog. *Switch*-Knoten eingefügt werden, der das Aufteilen des Kontrollflusses entsprechend einer Verzweigung abbildet.

4.5.4 Ermöglichte Optimierungen

Das DSA-Verfahren ermöglicht die Array-Datenflußanalysen bei *beliebigen* Kontrollflußgraphen. Mit der durch Anwendung des Verfahrens gewonnenen Information werden verschiedene Load/Store-Redundanz-Eliminationen für Array-Elemente ermöglicht. Wenn z.B. festgestellt wurde, daß ein Array-Element in einem EVR verfügbar ist, kann ein weiterer lesender Zugriff – und damit ein Speicherzugriff – verhindert werden, indem statt auf den Speicher auf das EVR zurückgegriffen wird. Neben total redundanten Speicherzugriffen können auch partiell redundante durch entsprechend ausgelegte und interpretierte Analysen ermittelt werden.

In den Kapiteln 5 und 6 werden verschiedene Load/Store-Optimierungen vorgestellt, die auf dem δ- oder Stretched Loop-Verfahren basieren. Die gleichen Optimierungen werden auch durch das DSA-Verfahren unterstützt. Dazu müssen Parametrisierungen der DSA-Analyse eingesetzt werden, die den Parametrisierungen der im Zusammenhang mit den Optimierungen verwendeten Datenflußanalysen entsprechen. Die passende Parametrisierung auf das DSA-Verfahren zu übertragen ist dabei meist nicht allzu schwer. Durch die Verwendung der sehr anschaulichen Map-Tupel ist die Interpretation der Analyse-Ergebnisse ebenfalls keine große Hürde. In den Optimierungskapiteln wird daher nicht bei jeder Optimierung gesondert auf das DSA-Verfahren verwiesen.

Noch zu untersuchen ist, ob eine eventuelle praktische Realisierung von EVR Nutzen erbringt. Nicht nur als analytisches Werkzeug, sondern auch in einer Implementation könnten sie eingesetzt werden. Für wiederkehrende Zugriffe könnten die beteiligten EVR im Sinne einer Register-Pipeline verwendet werden (siehe 5.3).

Das DSA-Verfahren, aber insbesondere die DSA-IR bringt noch weitere interessante Möglichkeiten, die aber über den Rahmen dieser Diplomarbeit hinaus gehen. Die bislang kaum genutzte Eigenschaft der DSA-IR, Programme in ihrer maximal parallelen Form, d.h. mit einer minimalen Anzahl an Abhängigkeiten, darzustellen, ist besonders für die Phase des Instruction Scheduling von großer Bedeutung.

4.5.5 Vor- und Nachteile

Vorteilhaft am DSA-Verfahren sind seine Parametrisierbarkeit und die Möglichkeit zur Behandlung sowohl affiner als auch nicht-affiner Ausdrücke in Indexfunktionen. Wenn auch bei nicht-affinen Ausdrücken nur sehr grobe Klassifikationen erfolgen, so sind entsprechende Programmfragmente analysierbar. Wenn bessere Methoden zur Behandlung nicht-affiner Ausdrücke zur Verfügung stehen, können diese relativ leicht eingebracht werden. Dazu braucht bei der speicherbasierten Abhängigkeitsanalyse bei der Unterscheidung der Klassen in Schritt 5) der neue Algorithmus eingebunden zu werden. Durch das DSA-Verfahren werden speicherbasierte und wertebasierte Datenabhängigkeitsanalysen für beliebige Kontrollflußgraphen ermöglicht, dabei ist die Analyse einigermaßen effizient – wenn die Terminierung gesichert ist (s.u.). Es werden durch die DSA-Analyse viele Optimierungen unterstützt, nicht nur Speicherzugriffsoptimierungen, sondern auch spätere Compilerphasen können profitieren. Der Gültigkeitsbereich der Analyse-Ergebnisse ist nicht auf einen stabilisierten Mittelbereich einer Schleife beschränkt, es können auch Aussagen über den Beginn und das Ende des Iterationsbereich gemacht werden.

Nachteilig ist die geringe Verbreitung der DSA-IR, so daß wenige Standard-Algorithmen für darauf basierende Optimierungen existieren. Allerdings sind die notwendigen Veränderungen an Optimierungen gering, damit sie auf einer DSA-IR durchgeführt werden können. Schwerer wiegt die nicht immer gesicherte Terminierung des Analysealgorithmus. Eine solche Situation tritt aber nicht bei strukturierten Schleifen auf, sondern nur bei Schleifen, deren Iterationsbereich nicht im vorhinein feststeht (z.B. While-Schleifen). Darauf muß besonders Rücksicht genommen werden. Wird die Analyse auf N Iterationen beschränkt, können keine Zugriffsregelmäßigkeiten erkannt werden, die erst ab der $N+1$-ten Iteration auftreten.

Insgesamt erscheint das DSA-Datenflußanalyse-Verfahren als empfehlenswert. Es ist von mittlerer Komplexität und ermöglicht eine Reihe verschiedener, leistungsfähiger Optimierungen. Die Möglichkeit zur Bearbeitung beliebiger Kontrollflußgraphen ist gerade bei DSP-Anwendung von großem Nutzen, da dort oft recht „unsaubere“ Programme vorzufinden sind. Die grobe Approximation von Array-Referenzen mit nicht-affinen Ausdrücken ist nicht besonders schädlich, da sie zum einen nicht sehr häufig vorkommen und zum anderen mit diesem Verfahren im Gegensatz zu einigen anderen überhaupt behandelt werden können.

4.6 Vergleich und Bewertung der Array-Datenflußanalysen

In den vorangegangenen Abschnitten sind verschiedene Array-Datenflußanalysen vorgestellt worden, die allesamt ihre Stärken und Schwächen haben. Zur Unterstützung der Entscheidung, welches Verfahren zu wählen ist, wenn bestimmte Anforderungen erfüllt werden sollen, z.B. bzgl. Präzision, Umfang analysierbarer Ausdrücke, ermöglichter Optimierungen, Aufwand zur Implementierung etc., sollen hier ihre Merkmale nebeneinandergestellt und bewertet werden.

Das δ-Verfahren ist ein allgemeines, parametrisierbares Verfahren zur Array-

Datenflußanalyse. Es kann ausschließlich strukturierte Schleifen / Schleifenschachtelungen mit rein affinen Ausdrücken in Indexfunktionen und Verzweigungsbedingungen analysieren. Es kommt sowohl mit eindimensionalen als auch mit mehrdimensionalen Arrays zurecht, kann aber in seiner Grundversion nur Abhängigkeiten in einer Dimension eines mehrdimensionalen Arrays erkennen. Das Verfahren benötigt zur Anwendung einen Schleifenkontrollflußgraphen, der aus dem meist verfügbaren Kontrollflußgraphen leicht zu konstruieren ist. Der verwendete Datenflußverband ist inklusive seiner Operatoren recht einfach und kann effizient implementiert werden. Das Fixpunkt-Iterationsverfahren terminiert in den meisten Fällen bereits sehr früh, so daß wenige Durchläufe durch den Schleifenkörper erforderlich sind. Daher ist das Verfahren sehr schnell. Die durch die Analyse bereitgestellten Datenflußinformationen ermöglichen eine Reihe unterschiedlichster Optimierungen. Zwar ist die Präzision der Information nicht allzu groß – es werden nur die Anzahl der in Abhängigkeiten involvierten Iterationen bestimmt – dennoch werden Optimierungen wie die *Elimination redundanter Loads und Stores* ebenso wie deren Verallgemeinerung in Form eines *einfachen Register-Pipelinings* ermöglicht. Dabei ist die Übertragung bekannter Analysen aus dem Bereich skalarer Datenabhängigkeiten i.a. sehr leicht möglich. Die Parametrisierbarkeit erlaubt eine einfache Definition einer „neuen" Analyse durch die Festlegung einiger weniger Parameter. Dadurch können bekannte skalare Optimierungen durch Anpassung auch für Array-Referenzen zugänglich gemacht werden. Darüberhinaus kann die δ-Technik auch zur Ermöglichung Array-spezifischer Optimierungen eingesetzt werden. Verfahren zur Registerallokation für Array-Variablen werden erst durch Kenntnis des Lebensbereichs von Array-Elementen möglich. Zusammen mit einem *verbesserten Registerpipelining* können auch durch ein verallgemeinertes Graphenfärbungsverfahren zur Interferenzermittlung leistungsfähige Codeverbesserungen vorgenommen werden. Neben den Speicherzugriffsoptimierungen wird auch ein *kontrolliertes Loop Unrolling* unterstützt[21], welches die Parallelität eines Schleifenkörpers auf systematische Weise so weit erhöhen kann, bis eine „Sättigung" eintritt.

Das *Stretched-Loop*-Verfahren zeichnet sich gegenüber der δ-Technik durch eine größere Präzision aus. Anstatt Abhängigkeiten nur auf Iterationsebene zu bestimmen, wird der Schritt zur Instruktionsebene vollzogen. Das hat seinen Preis im zu betreibenden Aufwand – nicht nur zur Laufzeit der Analyse, sondern vor allem während der Implementierung. Die Voraussetzungen sind hier etwas aufgelockert, denn auf die Single-Entry/Single-Exit-Eigenschaft kann verzichtet werden. Zu analysierende Indexfunktionen müssen allerdings weiterhin affin sein. Auch können weiterhin die speziellen Datenflußanalysen durch Parametrisierung eines allgemeinen Verfahrens instanziiert werden. Die Laufzeiteffizienz des Verfahrens ist trotz der zuvor benötigten Ergebnisse einer δ-Analyse noch gut, zumal eine effiziente Implementation mit Bitvektoren möglich ist. Die Stretched-Loop-Datenflußanalyse ermöglicht Speicherzugriffsoptimierungen, insb. eine Variante des Register-Pipelinings von dem bestimmte Optimalitäten bewiesen werden können. Das ist u.a. die Folge der großen Präzi-

[21] Loop Unrolling wird aber auch durch die präziseren Verfahren der Stretched Loop- und DSA-Analyse unterstützt, nicht jedoch von der Lazy-Analyse.

sion des Verfahrens. Neben den Optimierungen, die auch schon mit dem δ-Verfahren ermöglicht wurden, können weitere interessante Optimierungen die Analyse-Ergebnisse nutzbringend verwerten.

Die *Lazy*-Datenflußanalyse hat seinen bestechenden Vorteil in der Möglichkeit zur Behandlung sowohl affiner als auch nicht-affiner Ausdrücke in Indexfunktionen und Verzweigungsbedingungen. Es arbeitet exakt für affine Ausdrücke und liefert für nicht-affine Ausdrücke gute Approximationen. Dafür ist der Aufwand zur Implementierung und zur Laufzeit recht hoch. Für Approximationslösungen wird die SSA-Form gebraucht, und während der Laufzeit müssen Teilprobleme, die als NP-vollständig nachgewiesen sind, gelöst werden. Dennoch dürfte die Laufzeit für die meisten praktischen Probleme nicht unermeßlich steigen, da aufgrund der gewählten Suchstrategie meistens „recht effizient" gearbeitet wird. Parametrisierung ist bei diesem Verfahren nicht vorgesehen, es liefert eine Abhängigkeitsrelation zusammengehöriger Definition-Gebrauchs-Paare. Daraus lassen sich die für die *Elimination redundanter Loads* notwendigen Informationen extrahieren.[22]

Das *Dynamic Single Assignment*-Verfahren ist wieder eine sehr allgemeine Array-Datenflußanalyse, die speziellen Problembedürfnissen angepaßt werden kann. Es arbeitet auf beliebigen Kontrollflußgraphen und ist somit nicht auf spezielle Schleifenkonstrukte eingeschränkt. Durch Parametrisierung werden Vorwärts- und Rückwärtsanalysen möglich, ebenso wie die Erzeugung von Must- und May-Information. Das Verfahren arbeitet gut bei nicht-affinen Indexfunktionen, liefert jedoch nur grobe Approximationen der Abhängigkeiten für nicht-affine Ausdrücke in Indexfunktionen. Durch Linearisierung der Indizes können neben eindimensionalen Arrays auch mehrdimensionale behandelt werden. Der Aufwand zur Vorbereitung der Datenflußanalyse ist mäßig groß. Die Programme müssen in die DSA-Zwischenform transformiert werden, damit sie analysiert werden können, und zudem muß vor einer wertebasierten Abhängigkeitsanalyse eine speicherbasierte durchgeführt werden. Anschließend erfolgt eine Fixpunkt-Iteration, deren Terminierung durch besonderes Augenmerk sichergestellt werden muß. Durch das DSA-Verfahren werden eine Reihe von Optimierungen ermöglicht. Neben den in dieser Arbeit vorrangig behandelten Speicherzugriffsoptimierungen wie die *Elimination redundanter Loads und Stores* und *Register-Pipelining* sind hier insbesondere Low-Level-Verfahren wie das *Instruction Scheduling* zu nennen. Durch die maximal parallele Darstellung in der DSA-IR werden Scheduling-Aufgaben unterstützt und stark vereinfacht.

Von allen Array-Datenflußanalysen wird Information geliefert die zur Nutzung durch ein Verfahren zur Unterstützung von Software-Pipelining geeignet ist. Die präziseren Verfahren *Stretched Loop* und z.T. auch *DSA* können die Optimierungen, die durch die δ-Technik ermöglicht werden, in ähnlicher Weise unterstützen, ohne daß dies ausdrücklich wird.

[22] Eine weitere Möglichkeit ist die Verwendung der Abhängigkeitsrelationen zur Entscheidung über die Parallelisierbarkeit von Schleifen, was allerdings für den Bereich gewöhnlicher DSP-Anwendungen von untergeordneter Bedeutung ist.

Die Tabelle 4.6 gibt einen Überblick über die vorgestellten Verfahren und über ihre Stärken und Schwächen. Positiv zu bewertende Eigenschaften sind grün gekennzeichnet, negative Eigenschaften rot.

Die Auswahl eines der vorgestellten Verfahren für eine konkrete Anwendung ist nicht leicht, denn es handelt sich dabei um eine komplexe Mehrzielentscheidung. Die folgenden Argumente sollen durch eine Bewertung einzelner Kriterien und Eigenschaften helfen, für den Bereich der zu analysierenden DSP-Applikationen sinnvolle Entscheidungen für oder gegen ein Verfahren zu fällen.

Das δ-Verfahren arbeitet nur auf affinen Ausdrücken und liefert nur eine mittlere Präzision bei den erkannten Abhängigkeiten. Anscheinend sprechen diese Argumente gegen das Verfahren. Bei typischen DSP-Applikationen treten jedoch häufig auch nur affine Ausdrücke auf, so daß die erste Einschränkung nicht allzu schwerwiegend ist. Die mäßige Präzision ist auch zu tolerieren, denn zu vielen Speicherzugriffsoptimierungen ist keine größere Präzision erforderlich, oder der Aufwand zur Erstellung und Verwendung einer besseren Analyse rechtfertigt deren geringfügig bessere Ergebnisse in darauf basierenden Optimierungen nicht. Viele der in DSP-Applikationen vorkommenden Schleifen sind recht einfach und genügen der single-entry/single-exit-Eigenschaft. Für das Verfahren sprechen der relativ geringe Aufwand zur Implementierung und zur Laufzeit. Weiterhin fallen die umfangreichen Möglichkeiten der auf den Analyse-Ergebnissen arbeitenden Optimierungen auf. Insgesamt erscheint das Verfahren durchaus empfehlenswert, da nicht zuletzt durch eine einfache Art der Parametrisierung schnell verschiedene spezielle Datenflußprobleme gelöst werden können. Sollte sich die Präzision als unzureichend für eine Anwendung herausstellen, kann immer noch das *Stretched-Loop*-Verfahren hinzugenommen werden, für das ohnehin die δ-Analyse Voraussetzung ist.

Die *Stretched-Loop*-Analyse kann auch keine nicht-affinen Ausdrücke bearbeiten und versagt bei solchen ebenso. Dafür ist ihre Präzision bei den affinen Ausdrücken einiges höher als beim vorherigen Verfahren. Das dürfte allerdings auch einer der Hauptgründe sein, warum das *Stretched-Loop*-Verfahren vorzuziehen sein könnte. Zwar können multiple-entry/multiple-exit-Schleifen verarbeitet werden, doch bei sauberer Programmierung – die leider bei DSP-Applikationen viel zu selten betrieben wird – sollten solche auch nicht zu häufig vorkommen. Es können alle Speicherzugriffsoptimierungen, die auch mit dem δ-Verfahren ermöglicht werden, auch durch dieses Verfahren unterstützt werden. Zusätzlich kommen im wesentlichen weitere Verfeinerungen hinzu. Das Verfahren erscheint mit leichten Einschränkungen empfehlenswert, insbesondere in Situationen in denen die Präzision des δ-Verfahrens nicht ausreicht, oder in denen es gerechtfertigt ist, einen relativ hohen Aufwand für einige kleinere Gewinnzuwächse bei den Optimierungen zu betreiben, bietet sich das *Stretched-Loop*-Verfahren an.

Besonders die Möglichkeit zur Behandlung nicht-affiner Ausdrücke und die erzielte hohe Präzision sprechen für die *Lazy*-Datenflußanalyse. Gelegentlich – und vermutlich mit der Komplexität der Applikationen zunehmend – finden sich auch in DSP-Programmen nicht-affine Ausdrücke. Um Schleifen mit solchen Ausdrücken nicht vollkommen ignorieren zu müssen, ist deren Handhabbarkeit durchaus sinnvoll. Allerdings gibt es auch gewichtige Gründe gegen das *Lazy*-Verfahren. Zum einen ist der Aufwand während der Implementierung sehr hoch. Wenn auch die Laufzeiteffizienz noch tolerierbar ist, obwohl nicht konkurrenzfähig zu den übrigen Verfahren, so kann durch fehlende Parametrisierbarkeit nur eine geringe Anzahl Optimierungen unterstützt werden. Es erscheint, als ob der große Aufwand den Einsatz i.a. nicht rechtfertigt, obwohl es sicherlich Situationen gibt, in denen auf das Verfahren zurückgegriffen werden sollte. Dabei wird es sich aber um eher seltene Sonderfälle handeln, bei denen eine hohe Anzahl nicht-affiner Ausdrücke zu bearbeiten sind und dadurch große Optimierungspotentiale ausgeschöpft werden können.

Wenn auch das *DSA*-Verfahren ein bislang selten verwendeter Exot ist, so ist es doch durchaus recht attraktiv für den Bereich der DSP-Anwendungen. Zum einen kann es Datenabhängigkeiten von Array-Referenzen mit nicht-affinen Indexfunktionen approximieren. Wenn auch diese Approximationen recht grob sind, so können doch immerhin Schleifen mit solchen Referenzen behandelt werden. Es ist nicht unbedingt notwendig, daß auch „gute“ Approximationen geliefert werden. In [14] wird gezeigt, daß die Fehler die durch grobe Approximationen der nicht-affinen Abhängigkeiten entstehen, bei den dort verwendeten Programmen insgesamt eine seltene Fehlerquelle waren. Daher gibt es nur eine geringe Rechtfertigung für einen großen Mehraufwand zur Erzielung „guter“ nicht-affiner Approximationen. Zum anderen kommt die *DSA*-Analyse mit beliebigen Kontrollflußgraphen zurecht, was insbesondere bei dem oft verworrenen Programmierstil einiger DSP-Anwendungsprogrammierer[23] nützlich ist. Neben einer Vielzahl an Speicherzugriffsoptimierungen kann nach der Transformation zur DSA-IR diese in folgenden Phasen ebenfalls noch sinnvoll und nutzbringend verwendet werden. Insgesamt erscheint das Verfahren empfehlenswert.

4.7 Weitere Literatur

- Collard, J.F., Barthou, D., Feautrier, P.
  Fuzzy Array Dataflow Analysis
  5th ACM SIGPLAN Conference on Principles and Practice of Parallel Programming, Santa Barbara, 1995.

- Creusillet, B., Irigoin, F.
  Interprocedural Array Region Analyses
  Rapports du Centre de Recherche en Informatique, Ecole des Mines de Paris, Jan. 1996.

- Duesterwald, E., Gupta, R., Soffa, M.L.
  Reducing the Cost of Data Flow Analysis By Congruence Partitioning
  International Conference on Compiler Construction, Edinburgh, UK, 1994.

[23] vermutlich Elektrotechniker

- Duesterwald, E., Gupta, R., Soffa, M.L.
  A Demand-Driven Analyzer for Data Flow Testing at the Integration Level
  ICSE' 96 Proceedings of the 18th international conference on Software engineering, p. 575-584, 1996.

- Forgacs, I.
  An Exact Array Reference Analysis for Data Flow Testing
  ICSE' 96 Proceedings of the 18th international conference on Software engineering, 1996.

- Hind, M., Burke, M., Carini, P., Midkiff, S.
  An Empirical Study of Precise Interprocedural Array Analysis
  Scientific Programming, 3(3), p. 255-271, 1994.

- Kallis, A., Klappholz, D.
  Extending Conventional Flow Analysis to Deal with Array References
  Lecture Notes in Computer Science, Vol. 589, p. 251-265, 1991.

- Maydan, D.E., Amarasinghe, S.P., Lam, M.S.
  Array Data Flow Analysis and its Use in Array Privatization
  Principles of Programming Languages, Jan. 1993.

- Pugh, W., Wonnacott, D.,
  Eliminating False Data Dependences using the Omega Test
  ACM SIGPLAN PLDI'92 conference, 1992.

- Pugh, W., Wonnacott, D.,
  An Exact Method for Analysis of Value-based Array Data Dependences
  Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing, 1993.

- Pugh, W., Wonnacott, D.
  Experiences with Constraint-based Array Dependence Analysis
  Technical report CS-TR-3371, Univ. of Maryland, November 1994.

- Pugh, W., Wonnacott, D.
  Nonlinear Array Dependence Analysis
  Technical report CS-TR-3372, Univ. of Maryland, November 1994.

		δ	**Stretched-Loop**	**Lazy**	**DSA**
Voraussetzungen	Ausdrücke	affin	affin	nicht-affin	nicht-affin
	Schleifen	single-entry/ single-exit	multiple-entry/ multiple-exit	single-entry/ single-exit	beliebig
Verfahren	Parametrisierbarkeit	ja	ja	nein	ja
	Gültigkeitsbereich	stab. Schleife	stab. Schleife	vollständig	vollständig
	Präzision (affin)	mittel	hoch	exakt	hoch
	Präzision (nicht-affin)			hoch	gering
	Aufwand zur Laufzeit	gering	mittel	hoch	mittel
	Implementierungsaufwand	gering	mittel	hoch	mittel
Ermöglichte Optimierungen	Speicherzugriff	RLE RSE einfaches+verbessertes RP	RLE RSE optimales RP PRE PDE	RLE	RLE RSE RP PRE
	Sonstige	kontroll. Loop Unrolling Registerallokation		Schleifen-parallelisierung	Instruction Scheduling

Tabelle 4.1: Vergleich der verschiendenen Array-Datenflußanalysen

Kapitel 5

Load/Store-Optimierungen

In vielen Compilern werden die Zugriffe auf Array-Elemente von der Optimierung ausgenommen, da ihre skalaren Analysen keine Datenabhängigkeiten zwischen einzelnen Array-Elementen feststellen können. Mit den in Kapitel 4 vorgestellten Array-Datenflußanalysen stehen jedoch leistungsfähige Instrumente zur Verfügung, um Array-Datenabhängigkeiten zu bestimmen. Speicherzugriffsoptimierungen für Arrays können dadurch effizient unterstützt werden. Das in diesem Kapitel verfolgte Ziel ist dabei die Erkennung und Entfernung redundanter Load- und Store-Operationen von Array-Elementen. Speicherzugriffsoperationen sind dabei redundant, wenn der von ihnen referenzierte Wert schon anderweitig, d.h. ohne diese Operation, verfügbar ist. Genauer beschreiben das die beiden folgenden Definitionen.

Definition 5.0.1 *Eine Lade-Operation* `load R,A[i]` *an einer Stelle p im Programm ist* partiell/total redundant*, wenn entlang einiger/aller Pfade zu p der Wert von* `A[i]` *schon in einem Register* `R'` *verfügbar ist.*

Definition 5.0.2 *Eine Schreib-Operation* `store R,A[i]` *an einer Stelle p im Programm ist* partiell/total redundant*, wenn entlang einiger/aller Pfade von p aus eine weitere Definition des gleichen Array-Elementes folgt, ohne daß zwischenzeitlich ein Gebrauch stattgefunden hat.*

Im folgenden werden total redundante Loads/Stores verkürzend als redundante Loads/Stores bezeichnet. Wenn hingegen ein partiell redundantes Load/Store gemeint ist, wird dieses auch als solches gekennzeichnet.

Mit der durch das δ-Verfahren gewonnenen Information werden die Erkennung und Entfernung redundanter Loads und Stores ermöglicht. In diesem Kapitel wird zunächst gezeigt, wie die δ-Analyse bei der Elimination redundanter Stores eingesetzt werden kann. Anschließend werden redundante Loads mit der gleichen Analyse-Technik behandelt. Die Verallgemeinerung der Entfernung redundanter Loads über mehrere Iterationen hinweg führt zum sog. *Register-Pipelining*, dessen Grundkonzept im Anschluß vorgestellt wird.

5.1 Elimination redundanter Stores

Die *Redundant Store Elimination (RSE)* nach [8] dient zur Beseitigung von redundanten Schreibzugriffen auf Array-Elementen innerhalb von Schleifen. Dabei ist ein Schreibzugriff redundant, wenn unabhängig vom aktuellen Kontrollflußpfad eine weiterer Schreibzugriff auf das gleiche Array-Element stattfindet, ohne daß zwischenzeitlich ein lesender Zugriff auf das Element erfolgte. Beispiel 5.1.1 zeigt ein Programmfragment mit einem redundantem Store.

Beispiel 5.1.1 *Einfache Situation mit redundantem Store*

```
for(i = 1; i < UB; i++)
{
   a[i] = x;  /* Redundantes Store */
   ...
   a[i] = y;
}
```

Der erste Schreibzugriff in Beispiel 5.1.1 ist redundant, weil zwischen den beiden Definitionen kein Gebrauch von `a[i]` stattfindet. Nicht mehr so ganz einfach sind Situationen, in denen sich die Schreibzugriffe über mehrere Iterationen einer Schleife verteilen und zwischenzeitlich auch Lesezugriffe – auf andere Array-Elemente – erfolgen. Siehe dazu Beispiel 5.1.2.

Beispiel 5.1.2 *Komplexere Situation mit redundantem Store*

```
for(i = 1; i < UB; i++)
{
   a[i] = x;
   ...
   if (k > l)
   {
      a[i+1] = y;   /* 1-redundantes Store */
      ...
   }
}
```

Hier wird das durch `a[i+1] = y` geschriebene Array-Element durch die erste Instruktion der folgenden Iteration überschrieben. Zwischen beiden Definitionen liegt eine Differenz der Induktionsvariable von 1, d.h. die *Iterationsdistanz* ist 1. Deshalb handelt es sich um ein *1-redundantes Store*. Ein redundantes Store bewirkt keine nutzbringenden Effekte während der Programmausführung, es kann also durchaus auch mit dem Ziel der Redundanzverringerung eliminiert werden. Dabei ist jedoch zu beachten, daß z.B. das 1-redundante Store aus Beispiel 5.1.2 in allen Iterationen bis auf der letzten redundant ist. Da der letzten Iteration keine weitere folgt – sonst wäre sie nicht die letzte – wird auch die den Wert überschreibende Definition der nächsten Iteration nicht ausgeführt. Also bleibt der Wert erhalten. Daraus ergibt sich auf einfache Weise ein Weg zur

Elimination des redundanten Stores und gleichzeitiger Behandlung des „Sonderfalls“ der letzten Iteration:

Beispiel 5.1.3 *Elimination eines 1-redundanten Stores*

```
for(i = 1; i < UB-1; i++)
{
   a[i] = x;
   ...
   if (k > l)
   {
      ...
   }
}
a[UB] = x;
...
if (k > l)
{
   a[UB+1] = y;
   ...
}
```

Nach der Elimination des 1-redundanten Stores aus dem Schleifenkörper muß die letzte Iteration unter Beibehaltung eben jenes Stores ausgeführt werden. Dazu wird der ursprüngliche Iterationsbereich um eins reduziert, hier von UB auf UB-1. Anschließend wird der ursprüngliche Schleifenkörper als Epilog der Schleife angefügt, wobei jedes Vorkommen der Induktionsvariablen mit dem Wert der oberen Iterationsgrenze substituiert wird, hier UB (siehe Beispiel 5.1.3).

Mit einer geeignet parametrisierten δ-Datenflußanalyse wird ermittelt, welche Definitionen bis zu welchem Knoten ohne anschließenden Gebrauch erfolgten. Die Definitionen ohne folgendem Gebrauch können mit einer δ-busy-Analyse ermittelt werden. Unter den Kandidaten für redundante Stores sind diejenigen überflüssig, die eine vorherige Definition ohne zwischenzeitlichen Gebrauch redefinieren. Die Entscheidung, ob eine Definition redundant ist, erfolgt durch Auswertung der Ergebnisse der δ-busy-Analyse. Die mit den Knoten verknüpften Verbandswerte müssen in geeigneter Weise interpretiert werden. Danach kann der eigentliche Optimierungsschritt mit der Elimination des redundanten Stores und der Erzeugung des Schleifenepilogs erfolgen.

Zur allgemeinen Behandlung von redundanten Stores werden zunächst einige Definitionen benötigt. Dabei handelt es sich um eine Verallgemeinerung der *Busy Expressions* und um *δ-redundante Stores.*

Definition 5.1.1 *Falls ein Store s in einem Knoten n entlang aller Kontrollflußpfade zu einem Knoten n' ausgeführt wird, ohne daß das gespeicherte Array-Element entlang dieser Pfade über δ Iterationen hinweg benutzt wird, so ist s bei n'* δ-busy *(Abb. 5.1).*

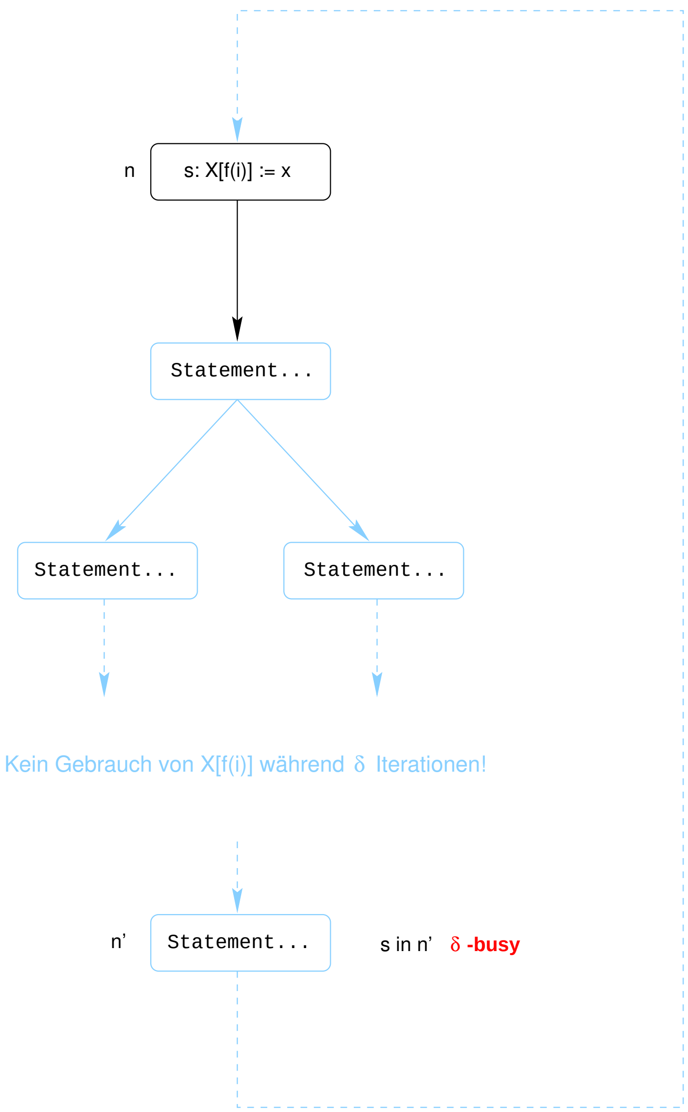


Abbildung 5.1: Store s in Knoten n' δ-busy

In Abb. 5.1 ist s δ-busy, wenn auf dem Pfad von n nach n' über δ Iterationen hinweg kein Gebrauch von $X[f(i)]$ stattfindet. Die Eigenschaft δ-busy ist wichtig bei der weiteren Suche nach redundanten Stores, denn δ-busy Stores sind Kandidaten für die Untersuchung auf δ-Redundanz. Nur Definitionen denen eine Redefinition ohne zwischenzeitlichen Gebrauch folgt, sind redundant. Bei einem δ-busy Store ist sichergestellt, daß das definierte Array-Element über δ Iterationen keinen Gebrauch erfährt. Wenn in diese δ Iterationen eine Redefinition des Elementes fällt, so ist das δ-busy Store redundant.

Definition 5.1.2 *Sei $s = (X[f(i)] = \ldots)$ ein Store in dem Knoten n des CFG. s im Knoten n ist genau dann* δ-redundant, *wenn es einen weiteren Store $s' = (X[f(i-\delta)] = \ldots)$ in einem Knoten n' im Schleifenkörper gibt, und s im Knoten n' δ-busy ist (Abb. 5.2).*

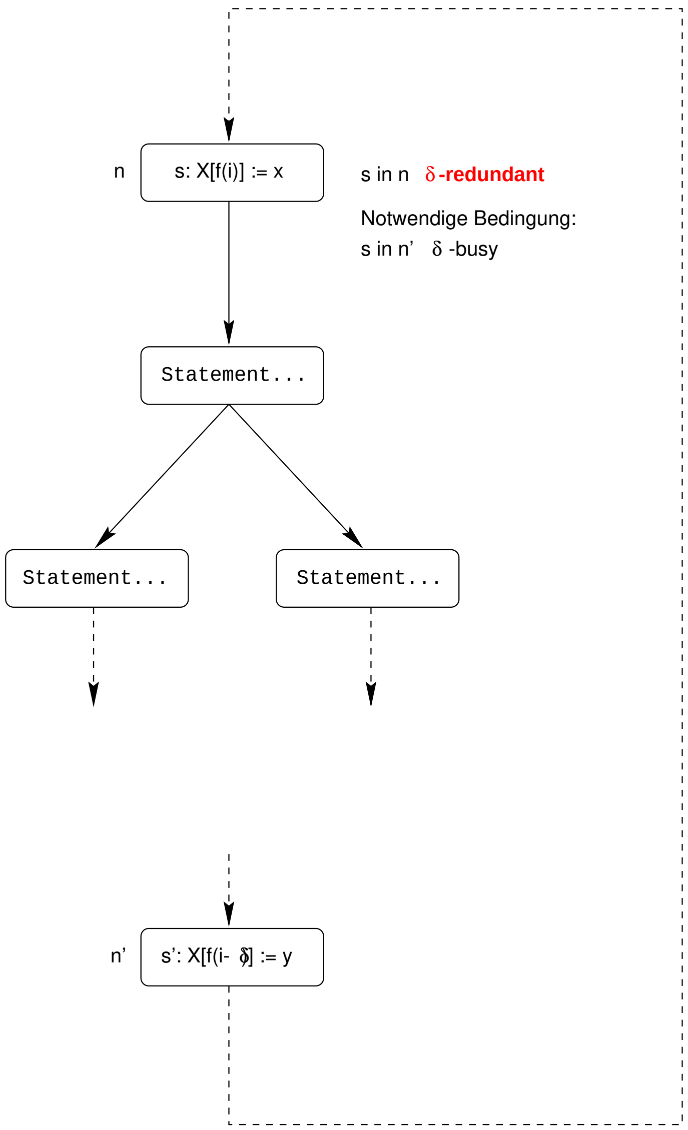


Abbildung 5.2: δ-redundantes Store s

In der Abbildung 5.2 ist gegenüber der Abbildung 5.1 im Knoten n' ein weiteres Store $X[f(i-\delta)] = y$ hinzugekommen. Wenn s weiterhin in n' δ-busy ist, dann wird das Element $X[f(i)]$ regelmäßig in n' überschrieben. Da zwischenzeitlich kein Gebrauch erfolgt – s ist in n' δ-busy – ist s δ-redundant. Damit kann das redundante Store s beseitigt werden.

Beispiel 5.1.4 *3-busy Stores*

```
for(i = 0; i < N; i++)
{
   a[i] = x; /* 1 */
   b[i] = a[i+3] + y; /* 2 */
   c[i] = k+1; /* 3 */
}
```

Im Beispiel 5.1.4 ist das Store aus Knoten 1 im Knoten 3 3-busy. Nach einer Definition von `a[i]` vergehen drei Iterationen bis zu einem Gebrauch des geschriebenen Wertes in Knoten 2. Das Store ist aber nicht redundant, da das Array-Element gebraucht und nicht überschrieben wird.

5.1.1 Analyse

Zur Analyse wird ausgehend von einer Definition eine vorherige Definition des gleichen Elementes gesucht, ohne daß ein zwischenzeitlicher Gebrauch des Elementes erfolgt. Die Suche von zeitlich späteren Stores hin zu früheren bestimmt die Rückwärts-Arbeitsrichtung. Dazu wird bei der Suche nach δ-redundanten Stores im umgekehrten Kontrollflußgraphen $\overline{CFG}$ gearbeitet, bei dem die Richtungen aller Kanten gegenüber dem „normalen“ CFG vertauscht sind. Die Definition von δ-Redundanz kann dabei direkt umgesetzt werden, wenn das δ der Knotenkennzeichnung bzw. dem Datenflußverband entspricht. Eine Definition erzeugt während der Analyse die Gültigkeit der untersuchten Eigenschaft δ-busy, während ein Gebrauch die Eigenschaft vernichtet.

Zur Bestimmung δ-redundanter Stores ist eine Analyse notwendig, die zuvor bestimmt, ob ein Store an einem Knoten δ-busy ist. Dies kann mittels der in Kap. 4.2 vorgestellten Array-Datenflußanalyse bewerkstelligt werden, wenn sie wie folgt parametrisiert wird. Es handelt sich um ein *Must*-Problem, so daß der *Must*-Verband eingesetzt werden muß. δ-busy Stores ist auch ein Rückwärts-Problem, so kommt der umgekehrte Schleifenkontrollflußgraph zum Einsatz. Die Menge $G[n]$ enthält zu einem Knoten n die verschiedenen dort in Definitionen auftretenden Array-Referenzen, während hingegen $K[n]$ die Gebräuche widerspiegelt.

5.1.2 Interpretation der Analyse

Die Eigenschaft δ-busy von Definitionen kann in den Knotenmarkierungen des LCFG abgelesen werden. Nicht jede Definition ist δ-redundant, deshalb muß zu jeder Definition nachgeschaut werden, ob bei dieser Definition eine andere Definition, die das gleiche Array-Element schreibt, δ-busy ist. Ist dies der Fall, so kann die Definition als δ-redundant klassifiziert werden.

Nach Lösung des Datenflußproblems kann an jedem Knoten die Lösung im Vektor IN abgelesen werden. IN deshalb, weil es sich wie oben erwähnt um eine Rückwärtsanalyse handelt, bei der die Datenflußrichtung umgekehrt ist. An einem Knoten n wird also für jeden Store $s \in G$ ein Verbandselement durch $IN[n, s]$ bezeichnet. Ein Wert $IN[n, s] = x$ bedeutet, daß das Store s bei Verlassen des Knoten n entsprechend obiger Definition δ-busy ist, für $pr(s, n) \leq \delta \leq x$, wobei pr(s,n) das in Kapitel 4.2 definierte Prädikat ist.

Um von einer Definition $s = (X[f(i)] = \ldots)$ in einem Knoten n zu bestimmen, ob sie δ-redundant ist, ist zu untersuchen, ob es im LCFG eine weitere Schreiboperation $s' = (X[f(i - \delta)] = \ldots)$ in einem Knoten n' gibt und gilt, daß s in n' δ-busy ist. Ist das der Fall, so ist s δ-redundant.

5.1.3 Optimierung

Nachdem feststeht, daß s δ-redundant ist, kann die eigentliche Optimierung vorgenommen werden. Dazu kann die Instruktion s aus dem Schleifenkörper eliminiert werden. Die obere Iterationsgrenze muß um δ auf nun $UB - \delta$ vermindert werden, und anschließend wird ein Epilog bestehend aus δ Aneinanderreihungen des ursprünglichen Schleifenkörpers (mit dem Store s) erstellt. In jedem Segment des Epilogs wird das Vorkommen der Induktionsvariablen durch den entsprechenden Wert ersetzt. Also ist bei δ Segmenten die Induktionsvariable beim ersten Segment durch $UB - \delta$ zu ersetzen, beim zweiten durch $UB - \delta + 1$, ... bis schließlich beim letzten Segment UB eingesetzt wird[1].

Eine Überprüfung darauf, ob die Weite δ der Datenabhängigkeit über die Anzahl der Iterationen UB der Schleife hinaus geht, $\delta > UB$, ist nicht erforderlich, da bei der Bestimmung der Eigenschaft δ-busy das Iterationsintervall der Schleife schon berücksichtigt wird (siehe dazu Kapitel 4.2).

Das Beispiel 5.1.5 zeigt die Optimierung einer Schleife mit einem 3-redundanten Store. Nach Entfernung des redundanten Stores aus dem Schleifenkörper werden vom oberen Iterationsende 10 drei Iterationen abgezogen. Der ursprüngliche Schleifenkörper wird dreifach an das Schleifenende angehängt. Dabei werden für die vorherige Induktionsvariable feste Werte eingesetzt.

[1] Bei einem größeren δ muß der Epilog nicht zwangsläufig durch eine textuelle Aneinanderreihung des Schleifenkörpers erzeugt werden, sondern kann auch durch eine Epilog-Schleife realisiert werden.

Beispiel 5.1.5 *3-redundantes Store*

Vorher:

```
for(i = 0; i < 10; i++)
{
   a[i] = x;
   b[i] = c[i] + y;
   a[i-3] = k+1;
}
```

Nachher:

```
for(i = 0; i < 7; i++)
{
   b[i] = c[i] + y;
   a[i-3] = k+1;
}
a[7] = x;
b[7] = c[7] + y;
a[7-3] = k+1;
a[8] = x;
b[8] = c[8] + y;
a[8-3] = k+1;
a[9] = x;
b[9] = c[9] + y;
a[9-3] = k+1;
```

Befinden sich in einer Schleife mehrere redundante Stores mit unterschiedlich großen Iterationsdistanzen, so wird das größte δ als Grundlage der Optimierung herangezogen. Beispiel 5.1.6 verdeutlicht dies. Es gibt ein 3-redundantes und ein 4-redundantes Store. Das 4-redundante Store verlangt, daß für die letzten vier Iterationen der ursprüngliche Schleifenkörper ausgeführt wird. Wenn auch für das 3-redundante Store eine weitere Iteration mit dem verkürzten Schleifenkörper ausgeführt werden könnte, würde das jedoch bei dem 4-redundanten Store zu einem Programmfehlverhalten führen. In Sinne der Sicherheit von Optimierungen ist also die größte Iterationsdistanz auszuwählen. Im Beispiel ist der Epilog als Schleife realisiert.

Beispiel 5.1.6 *Mehrere redundante Stores in einer Schleife*

Vorher:

```
for(i = 0; i < 10; i++)
{
   a[i] = x;
   b[i] = c[i] + y;
   a[i-3] = k+1;
   b[i-4] = d[i];
}
```

Nachher:

```
for(i = 0; i < 6; i++)
{
   a[i-3] = k+1;
   b[i-4] = d[i];
}
for(i = 6; i < 10; i++)
{
   a[i] = x;
   b[i] = c[i] + y;
   a[i-3] = k+1;
   b[i-4] = d[i];
}
```

Wichtig bei der Optimierung ist, daß zwischen den beiden Schreibzugriffen eine konstante Iterationsdistanz liegt. Seien die Indexfunktion der ersten Referenz $f_1(i) = a_1 \times i + b_1$ und die der zweiten Referenz $f_2(i) = a_2 \times i + b_2$. Bei Funktionen beschreiben also Geraden mit den Steigungen a_1 bzw. a_2 und einem Y-Versatz von b_1 bzw. b_2. Damit sich eine konstante Iterationsdistanz ergibt, müssen beide Geraden parallel verlaufen, d.h. es muß gelten $a_1 = a_2 = a$. Die Iterationsdistanz ist dann $d = \frac{b_2 - b_1}{a}$. Zur Veranschaulichung dient Beispiel 5.1.7.

Beispiel 5.1.7 *Konstante Iterationsdistanzen*

$f_1(i) = 3 \times i + 6 \longrightarrow a = a_1 = 3, b_1 = 6$
$f_2(i) = 3 \times i + 9 \longrightarrow a = a_2 = 3, b_2 = 9$
$d = \frac{b_2 - b_1}{a} = \frac{9-6}{3} = 1$

Somit erreicht $f_1(i)$ nach der Iterationsdistanz 1, also nach Inkrement von i um 1, den Wert von $f_2(i)$.

Wäre die Iterationsdistanz nicht konstant, so hieße das für die Optimierung, daß zwischen den Indizes der zwei betrachteten Schreib-Operationen ein variabler Abstand läge. In der Distanz variable Abhängigkeiten lassen sich aber nicht durch (statische) Programmtransformationen (wie der RSE) behandeln, sondern erfordern Fallunterscheidungen zur Laufzeit, was aber aus der Gründen der Effizienz überhaupt nicht zu vertreten ist. Wenn hingegen die Iterationsdistanz konstant aber nicht ganzzahlig ist, so bedeutet dies, daß es zu keiner Abhängigkeit kommen kann, denn die punktweise Berechnung der Indexfunktionen an den diskreten, ganzzahligen Werten der Induktionsvariablen i, sorgt dafür, daß die Funktionen zu mathematischen Folgen werden, die keine gemeinsamen Punkte haben (siehe dazu Beispiel 5.1.8).

Beispiel 5.1.8 *Nicht-ganzzahlige Iterationsdistanzen*

$f_1(i) = 3 \times i + 5 \longrightarrow a = a_1 = 3, b_1 = 5$
$f_2(i) = 3 \times i + 9 \longrightarrow a = a_2 = 3, b_2 = 9$
$d = \frac{b_2 - b_1}{a} = \frac{9-5}{3} = \frac{4}{3}$

$\forall i \in \mathbf{N} : f_1(i) = 8, 11, 14, 17, 20, 23, \ldots$
$\forall i \in \mathbf{N} : f_2(i) = 12, 15, 18, 21, 24, 27, \ldots$

5.1.4 Vor- und Nachteile von RSE

Die Anwendung der RSE ist mit Vor- und Nachteilen behaftet, die im folgenden gegenübergestellt werden.

- **Vorteile**
  - Elimination redundanter Array-Schreiboperationen
  - Verkleinerung des Schleifenkörpers
  - Verringerung des Laufbereichs der Schleife
- **Nachteile**
  - Schleifen mit einzelnen nicht-affinen Ausdrücken können *insgesamt nicht* behandelt werden
  - Vergrößerung des Code-Umfangs durch einen Epilog

Bei der RSE scheinen die Vorteile zu überwiegen, denn die Redundanzverminderung in der Schleife kann zu erheblich besserem Laufzeitverhalten führen. Zwar können Schleifen mit nicht-affinen Ausdrücken nicht behandelt werden, doch kommen diese in vielen DSP-Applikationen nur gelegentlich vor. Zudem können solche Schleifen leicht erkannt und bei der RSE übergangen werden. Die Vergrößerung des Code-Umfangs ist in vielen Fällen zu rechtfertigen, wenn dadurch eine höhere Ausführungsgeschwindigkeit zu erwarten ist. Im Einzelfall ist jedoch jeweils zwischen den hier konkurrierenden Zielen Geschwindigkeitssteigerung und Speicherplatzersparnis abzuwägen. Die Verkleinerung des Schleifenkörpers wird nur in den seltensten Fällen die Code-Vergrößerung durch den Epilog aufwiegen.

Weitere Effekte, die sich sowohl zum Vorteil als auch zum Nachteil auswirken können, kommen unter bestimmten Bedingungen hinzu. Die Ursachen weiterer Einflüsse auf RSE lassen sich grob in die Kategorien *weitere Optimierungen* und *Zielarchitektur* einteilen. Durch ihre Kenntnis kann im konkreten Anwendungsfall evtl. genauer über Nutzen der RSE entschieden werden.

- **Einfluß durch weitere Optimierungen:**

$\ominus$ Falls der redundante Speicherzugriff in der nicht-optimierten Version der Schleife in seiner Latenz durch andere Operationen versteckt wurde, so kann es dazu kommen, daß es trotz der Elimination zu keiner Beschleunigung kommt, gar aufgrund anderer in dieser Liste aufgeführter Einflüsse die Ausführung *verlangsamt* wird.

$\ominus$ Der Iterationsbereich der Schleife wird verkleinert, dafür wird ein der Anzahl der entfernten Iterationen entsprechender Epilog erzeugt. Dieser Vorgang kann durchaus als *partielles Loop Unrolling* angesehen werden – mit allen Konsequenzen, die Loop Unrolling für die Optimierung von Programmen nach sich zieht. Zum einen vergrößert sich der Codeumfang, da nun neben der eigentlichen Schleife der ursprüngliche Schleifenkörper evtl. mehrfach hintereinander auftaucht. Bei einem großen Schleifenkörper und einer großen Iterationsdistanz des eliminierten Stores, kann es dazu kommen, daß der Epilog nicht mehr annehmbar in seinem Umfang wird.[2] Bei Implementierungen ist also auf die Möglichkeit zur Begrenzung der zu bearbeitenden Iterationsdistanz zu achten. Wenn die Iterationsdistanz Werte in der Größenordnung der Gesamtanzahl der Iterationen erreicht, ist zu überlegen, ob die Schleife als ganzes einem Loop Unrolling unterzogen wird, um von der Unabhängigkeit von der Induktionsvariablen zu profitieren.

⊕ Im Epilog treten keine von der Induktionsvariablen abhängigen Indizes mehr auf, da ein Vorkommen der Induktionsvariablen durch einen entsprechenden Wert substituiert wurde. Somit werden die Indizes von Array-Referenzen konstante Ausdrücke, die einer einfachen Abhängigkeitsüberprüfung zugänglich sind. Dadurch ergeben sich im Epilog oft Anwendungsmöglichkeiten von einfachen Optimierungsverfahren, die in üblichen Compilern häufig integriert sind. Beispielsweise können skalare Optimierungen wie Copy Propagation und Constant Propagation oder Common Subexpression Elimination gemeinsame Adreßausdrücke vereinfachen, oder mehrfach vorkommende Array-Referenzen auf ein gleiches Element erkennen und durch einen Registerzugriff ersetzen. Desweiteren bietet der Epilog – vor allem, wenn einige Iterationen des ursprünglichen Schleifenkörpers in Folge auftreten – gute Möglichkeiten zum effizienten Scheduling der Instruktionen.

- **Einfluß durch Zielarchitektur:**

⊙ Falls der Prozessor, auf dem ein durch RSE behandeltes Programm zu Ausführung kommt, einen Instruktionscache besitzt, und die optimierte Schleife eine innere Schleife eines Loop Nests ist, so ist zu beachten, daß zwar die Elimination des redundanten Loads der inneren Schleife einer Beschleunigung der Programmausführung bewirken kann, das Anfügen des Epilogs aber den Schleifenkörper der umgebenden Schleife vergrößert, so daß dieser eventuell nicht mehr vollständig in den Instruktionscache paßt. Zur Beschleunigung kann in diesem Fall ein gegenläufiger Effekt hinzukommen, der trotz erfolgreicher Optimierung einer Schleife die Programmausführung verlangsamt, wenn die Gewinne in der inneren Schleife durch die Cache-Misses der äußeren Schleife verzehrt werden.

Wenn trotz der erfolgversprechenden Möglichkeiten der RSE selten Gelegenheiten zu deren Anwendung zu finden sind, liegt das vor allem daran, daß bei

[2] Natürlich ist es möglich, den Epilog auch als Schleife zu gestalten und damit seinen (statischen) Instruktionsumfang zu begrenzen. Nichtsdestotrotz bleibt seine Größe in der Größenordnung des ursprünglichen Schleifenkörpers bestehen.

wohlüberlegter Programmierung von Applikationen, insbesondere für zeitkritische Anwendungen, nur selten redundante Stores vom Programmierer „übersehen" werden. Bei kleineren Schleifenkörpern „sieht" der Programmierer die Redundanz, während größere, unübersichtliche Schleifenkörper bei DSP-Applikationen nicht so häufig sind. Dennoch hat dieses Verfahren seine Berechtigung, zum einen dadurch, daß auch im Bereich der DSP-Applikationen die Programme immer größer und komplexer werden, so daß sie nicht mehr so einfach „von Hand" zu optimieren sind, zum anderen wird die Möglichkeit gegeben, „naiv" – d.h. nahe an mathematischen Spezifikationen ohne Rücksicht auf ein bestimmtes Maschinenmodell – zu programmieren und dennoch effizientes Laufzeitverhalten zu erzielen.

5.2 Elimination redundanter Loads

Die *Redundant Load Elimination (RLE)* nach [8] und [9] dient zur Beseitigung von redundanten Array-Lesezugriffen innerhalb von Schleifen. Dabei ist ein Lesezugriff redundant, wenn unabhängig vom aktuellen Kontrollflußpfad ein weiterer Lese- oder Schreibzugriff auf das gleiche Array-Element vorangeht, ohne daß zwischenzeitlich eine neue Definition des Elementes erfolgte. Beispiel 5.2.1 zeigt ein Programmfragment mit einem redundanten Load.

Beispiel 5.2.1 *1-redundantes Load*

```
for(i = 1; i < UB; i++)
{
   if (k > l)
   {
      x = a[i];   /* 1-redundantes Load */
      ...
   }
   ...
   a[i+1] = y;
}
```

Nach der Abarbeitung des Schleifenkörpers mit der Instruktion `a[i+1] = y` wird in der nächsten Iteration auf `a[i]` lesend zugegriffen. Dabei handelt es sich um das gleiche Element, welches zuvor geschrieben wurde, denn mit dem Übergang in die nächste Iteration erhöht sich auch die Induktionsvariable von i auf $i + 1$. Statt eines erneuten Speicherzugriffs kann versucht werden, den Wert von der ersten Referenz zur zweiten Referenz in einem Register zu transportieren, und damit den (langsamen) Speicherzugriff durch einen (schnellen) Registerzugriff zu ersetzen. Zwischen den beiden Referenzen liegt *ein* Übergang in eine neue Iteration, somit ist `x = a[i]` *1-redundant*. Die Optimierung der Schleife aus Beispiel 5.2.1 zeigt Beispiel 5.2.2, dort wird bei der ersten Referenz der Wert in einer temporären Variablen zwischengespeichert, die zweite Referenz kann dann durch einen Zugriff auf diese temporäre Variable ersetzt werden.

Beispiel 5.2.2 *Elimination eines 1-redundantes Loads*

```
t = a[1];
for(i = 1; i < UB; i++)
{
   if (k > 1)
   {
      x = t;
      ...
   }
   ...
   t = y
   a[i+1] = t;
}
```

Der Schleife muß ein Prolog vorangestellt werden, der die temporäre Variable `t` initialisiert. Der erste Gebrauch von `t` liegt innerhalb des Schleifenkörpers vor der entsprechenden Definition, so daß im Prolog eine Zuweisung an `t` enthalten ist, die den Wert der im Schleifenkörper eliminierten Referenz an der Stelle $i = 1$ bereitstellt.

Zur Erkennung, welche Werte vorangegangener Referenzen nach einer Anzahl Iterationen noch verfügbar sind, dient die Analyse *δ-available values.*

Definition 5.2.1 *Eine Wert v eines Array-Elementes e dessen Definition an Knoten n erfolgt, ist bei einem Knoten n'* δ-available*, falls es entlang aller Pfade von n nach n' über δ Iterationen hinweg zu keiner Redefinition von e kommt.*

5.2.1 Analyse

Sobald eine Array-Element erstmalig referenziert wird, ist dessen Wert für die folgenden Knoten verfügbar. Bei einer Redefinition wird die Eigenschaft der Verfügbarkeit vernichtet. Entsprechend erscheint die Vorwärtsrichtung als Arbeitsrichtung der Datenflußanalyse angebracht. Die Distanzen, für die ein Wert über mehrere Iterationen verfügbar ist, entsprechen den Elementen des Datenflußverbandes. Wenn zu allen Knoten die verfügbaren Werte berechnet werden sollen, so kann dies iterativ mit dem δ-Verfahren aus Kapitel 4.2 bewerkstelligt werden.

Die geeignete Parametrisierung sie so aus. Wie bei RSE kommt der *must*-Verband zum Einsatz, da auch *δ-available values* ein Must-Problem ist. Unterschiedlich sind die Datenflußrichtung und die Mengen G und K. Es handelt sich nun um ein *Vorwärts*-Problem bei dem G alle Referenzen, d.h. Definitionen und Gebräuche, umfaßt, während K lediglich die Definitionen beinhaltet. Diese Festlegung ergibt sich daraus, daß beliebige Referenzen Array-Elemente verfügbar machen, jedoch nur Definitionen die Verfügbarkeit beenden können.

5.2.2 Interpretation der Analyse

Nach Terminierung der Fixpunkt-Iteration ist die Lösung des Datenflußgleichungssystems durch die Ausprägungen der Verbandselemente an den Knoten des LCFG abzulesen. Für *δ-available values* ist es wichtig zu wissen, welche Werte bei Erreichen eines Knoten n verfügbar sind. Um also festzustellen, ob eine Referenz r bei Knoten n *δ-available* ist, muß $IN[n,r]$ betrachtet werden. Dabei bedeutet $IN[n,r] = x$, daß die Referenz r am Knoten n *δ-available* bzgl. der Distanzen $pr(r,n) \leq \delta \leq x$ ist, wobei $pr(r,n)$ auch hier das in Kapitel 4.2 definierte Prädikat ist, welches ausdrückt, ob die Referenz r dem Knoten n im Schleifenkörper vorangeht oder folgt.

Zur Entscheidung, ob ein Load $l = X[f(i)]$ in Knoten n *δ-redundant* ist, muß es eine Referenz $r = X[f(i-\delta)]$ geben, die bei Eintritt in den Knoten n *δ-available* ist. Ist dies der Fall, kann erneute Gebrauch durch optimiert werden.

5.2.3 Optimierung

Die *RLE* bezeichnet den Fall der Elimination eines redundanten Loads mit einer Iterationsdistanz 1. Für ein $\delta > 1$ sind andere Techniken erforderlich, die in den nachfolgenden Abschnitten erläutert werden. Sei nun ein Load $l = X[f(i)]$ vorhanden, von dem feststeht, daß es redundant ist, da eine Referenz $r = X[f(i-\delta)]$ verfügbar ist. Zur Referenz r muß eine Operation hinzugefügt werden, die deren Wert in einer temporären Variablen zwischenspeichert. Die Ladeoperation l kann nun durch einen lesenden Zugriff auf eben diese temporäre Variable ersetzt werden. Dabei sind jedoch zwei Fälle zu unterscheiden:

1. l liegt im Schleifenkörper vor r, d.h. $pr(l, n_r) = 0$. Das skizzierte Verfahren kann ohne Änderung angewendet werden:

   Vorher:

   ```
   for(i = 1; i < UB; i++)
   {
      x = a[i];
      ...
      y = a[i-1];
   }
   ```

   Nachher:

   ```
   t = a[1];
   for(i = 1; i < UB; i++)
   {
      x = t;
      ...
      t = a[i-i];
      y = t;
   }
   ```

2. l liegt im Schleifenkörper nach r, d.h. $pr(l, n_r) = 1$. Da vor dem ersten Gebrauch durch l im Schleifenkörper eine in der ersten Iteration ungültige Zuweisung an die temporäre Variable durch r stattfindet, muß von außen durch eine zweite temporäre Variable der Initialwert in die Schleife hineingebracht werden und dort in jeder Iteration auf den aktuellen Wert gebracht werden:

Vorher:

```
for(i = 1; i < UB; i++)
{
   x = a[i-1];
   ...
   y = a[i];
}
```

Nachher:

```
t' = a[1];
for(i = 1; i < UB; i++)
{
   t = a[i-1];
   x = t;
   ...
   y = t';
   t' = t;
}
```

Hier ist darauf zu achten, daß ein nachfolgender Lauf einer *Copy Propagation* nicht auf einer IR-Ebene mit expliziten Array-Zugriffen arbeitet, sondern nur auf einer niedrigeren Ebene mit Registern, da ansonsten der eliminierte Speicherzugriff wieder eingefügt werden könnte.

5.2.4 Vor- und Nachteile der RLE

Die RLE entfernt nicht nur Operationen aus dem Schleifenkörper, sie fügt auch welche ein. Zusätzlich wird eine temporäre Variable eingeführt. Folgende Vor- und Nachteile stehen sich gegenüber:

- **Vorteile**
  - Geschwindigkeitssteigerung durch Ersatz einer redundanter Speicherleseoperationen durch eine Registerkopieroperation
- **Nachteile**
  - Erzeugung eines Schleifenprologs, damit Vergrößerung des Code-Umfangs
  - Einführen einer temporären Variablen

Ebenso wie bei der RSE überwiegen bei der RLE die Vorteile. Der erzeugte Schleifenprolog ist bei RLE sehr klein und umfaßt eine Speicherzugriffsoperation, die auch ohne Optimierung ausgeführt werden muß. Schwerer wiegt schon die Einführung einer temporären Variablen. Bei einem Prozessor mit einem homogenen Registersatz und hinreichend vielen Registern sollte aber i.a. noch ein Register für diese sehr effiziente Optimierung zur freien Verfügung stehen.

Durch andere Optimierungen oder Hardware-Einflüsse kann die Leistungsfähigkeit der RLE beeinflußt werden. Die folgende Auflistung zeigt weitere Möglichkeiten zur Interaktion von RLE mit diesen Einflüssen.

- **Einfluß durch weitere Optimierungen:**

⊕ Ein Schleifenprolog wird erzeugt, der die Initialisierung der temporären Variablen erledigt. In diesem Prolog treten nur konstante Indizes auf. Falls vor dem Eintritt in den Schleifenkörper genügend viele Instruktionen zur Verfügung stehen, kann die Latenz des Speicherzugriffs durch diese weiteren Operationen „versteckt" werden, so daß sich keine oder geringe Zusatzkosten für die Abarbeitung des Prologs ergeben.

⊕ Treten RSE und RLE gemeinsam auf und zwar in der Form, daß zunächst die Array-Zugriffe eine Schleife eine RSE ermöglichen und eine unmittelbar darauf folgende Schleife, die Anwendung der RLE auf Elemente des gleichen Arrays erlaubt, so reihen sich der Schleifenepilog der ersten Schleife und der Schleifenprolog der zweiten Schleife direkt aneinander. Falls nun Datenabhängigkeiten zwischen Array-Elementen der letzten Iterationen der ersten Schleife und den ersten Iterationen der zweiten Schleife bestehen, können diese in dem o.g. Bereich zwischen beiden Schleifen behandelt werden. Zwar dürfte dies in typischen Applikationen ein seltener Fall sein, sollte aber dennoch nicht übersehen werden.

- **Einfluß durch Zielarchitektur:**

⊖ Falls ein heterogener Registersatz vorliegt, kann es sein, daß kein allgemeines Register zum Halten des Wertes bis zum erneuten Gebrauch zur Verfügung steht. Das kann zum Aus- und späteren wieder Einlagern des Wertes führen *(Spilling)*.

⊖ Bei Architekturen mit heterogener Registersätzen kann auch ein Register zum Halten des Wertes belegt werden, daß eng mit einer funktionalen Einheit verbunden ist. Es ist möglich, daß durch die Belegung eines solchen Registers die Verwendung z.B. eines Addierers über eine längere Zeitdauer verhindert wird. Der Scheduler muß entscheiden, ob der Addierer blockiert oder das Register wieder geräumt wird. (Spilling). Besteht nicht der Zwang das Register zu räumen, wird zwar so die Anzahl der Speicherzugriffe vermindert, doch u.U. kommt es zu einem Rückgang der Gesamt-Performance, wenn der Addierer zwischenzeitlich sinnvoll genutzt werden könnte. Kommt es dagegen zum Spilling, so steigt die Anzahl der Speicherzugriffe wieder an.

⊕ Wenn die Kosten eines Speicherzugriffs groß sind, z.B. weil kein Datencache vorhanden ist, und der Speicherbus aufgrund seiner hohen Auslastung einen *Bottleneck* bei der Ausführung darstellt, kann RLE (aber auch RSE) eine besonders große Wirkung haben. Falls ein erheblicher Teil der Speicherzugriffe redundant ist, gelingt es, diesen Teil zu erkennen und zu eliminieren. Damit einher geht eine Steigerung der Effizienz der Speicherzugriffe, was wiederum zur Erhöhung der Performance des Systems beiträgt.

Die RLE betrachtet in dieser Form nur redundante Lesezugriffe innerhalb einer Iteration (loop-independent) oder Abhängigkeiten über eine Distanz von maximal einer Iteration (loop-carried, δ=1). Eine Verallgemeinerung dieses Konzepts ermöglicht die Elimination von redundanten Lesezugriffen auch über größere Iterationsdistanzen. Dazu dient das *Register-Pipelining*, dessen Grundform im folgenden Abschnitt ausführlich diskutiert wird.

5.3 Einfaches Register-Pipelining

Register-Pipelining verfolgt das gleiche Ziel wie die RLE, jedoch ohne die Beschränkung auf maximale Iterationsdistanzen von 1. Beim RLE werden Speicherzugriffe nach Einführung *einer* temporären Variablen durch Registerzugriffe ersetzt. Wenn nun größere Iterationsdistanzen bearbeitet werden, so müssen auch *mehrere* temporäre Variablen eingeführt und verwaltet werden, da in jeder Iteration ein neuer Wert hinzukommt und ein anderer Wert verbraucht wird. Die dazu benutzte queue-artig organisierte Datenstruktur wird *Register-Pipeline* genannt, worin zusätzlich noch zum Ausdruck kommt, daß eine Allokation dieser Pipeline in den Registern des Prozessors zur Ausnutzung deren höherer Zugriffsgeschwindigkeit erforderlich ist.

Beispiel 5.3.1 *Register-Pipeline mit der Tiefe 3*

Ursprüngliche Schleife

```
for(i = 1; i < UB; i++)
{
   x = a[i];
   ...
   y = a[i+3];
   ...
}
```

Schleife mit Register-Pipeline

```
t''' = a[1];
t''  = a[2];
t'   = a[3];
for(i = 1; i < UB; i++)
{
   x = t''';
   ...
   t = a[i+3];
   y = t;
   ...
   t''' = t'';
   t''  = t';
   t'   = t;
}
```

In Beispiel 5.3.1 wird die Array-Referenz `a[i]` durch einen Zugriff auf `t''` ersetzt und der in jeder Iteration hinzukommende Wert von `a[i+3]` in `t` gespeichert. Darüberhinaus werden Kopieroperationen unter den an der Register-Pipeline (t, t', t'') beteiligten Variablen eingefügt, so daß die Werte, die in `t` geschrieben werden nach zwei Iterationen in `t''` zum Gebrauch zur Verfügung stehen.

Ebenso wie bei RLE ist es erforderlich, die temporären Variablen bzw. die Register-Pipeline in einem Schleifen-Prolog zu initialisieren. Dazu dienen in dem Beispiel die drei der Schleife vorangestellten Zuweisungen, die die ersten drei referenzierten Array-Elemente in die an der Register-Pipeline beteiligten Variablen kopieren.

5.3.1 Analyse und deren Interpretation

Um eine Register-Pipeline zu konstruieren, ist es notwendig zu erfahren, welche Werte an einem Load *verfügbar* sind. Darin ist eine Register-Pipeline dem RLE-Verfahren sehr ähnlich. Sie geht über die Beschränkung von RLE auf Iterationsdistanzen von eins hinaus, und kann auch redundante Loads über größere Distanzen eliminieren. Dazu muß die Analyse der verfügbaren Werte auch die Information liefern, welche vorherigen Referenzen über wie viele Iterationen verfügbar sind. Die gewünschte Information wird bereits von der δ-available-values-Analyse des vorherigen Abschnitts geliefert, dort wurden aber nur die Referenzen mit $\delta = 1$ für Optimierungen verwendet.

Da RLE also ein Spezialfall des *Register-Pipelinings (RP)* ist und die gleiche Analyse verwendet, kann mit der Information aus δ-available-values die Entscheidung getroffen werden, ob ein bestimmtes Load δ-redundant ist. Zur folgenden Optimierung werden *alle* δ-redundanten Loads herangezogen, nicht nur diejenigen mit $\delta = 1$.

5.3.2 Optimierung

Nach der Feststellung, daß ein Load δ-redundant ist und eine Abhängigkeit von einer anderen Array-Referenz über mehrere Iterationen besteht, kann nun eine *einfache Register-Pipeline* aufgebaut werden. Dazu wird der Gebrauch durch einen Zugriff auf eine temporäre Variable ersetzt. Zwischen der ersten Referenz auf ein Array-Element und dem eliminierten Zugriff wird der Wert durch eine Reihe von temporären Variablen transportiert. Mit dem Übergang von einer Iteration in die nächste, werden die Inhalte aller an der RP beteiligten temporären Variablen einen Schritt weiter kopiert. Nach δ Iterationen schiebt sich der Wert durch die RP bis zur temporären Variable, die anstelle des redundanten Array-Elementes referenziert wird.

Die RP ist deshalb *einfach*, weil lediglich eine Menge von temporären Variablen verwendet wird, zwischen denen durch Kopieroperationen Werte verschoben werden. Die Zuweisung von Registern an diese Variablen wird einer folgenden

Registerallokationsphase des Compilers überlassen, in der Hoffnung, daß auch wirklich alle temporären Variablen in Registern Platz finden. Auch wird keine besondere Hardware-Unterstützung für die Verwaltung der RP in Anspruch genommen, insbesondere werden die Kopieroperationen nicht durch andere Operationen ersetzt, die weniger Aufwand erfordern. Strategien zur besseren Plazierung von Load-, Store- und Kopier-Operationen werden vorerst noch nicht verwendet.

Unterscheiden läßt sich zwischen den Fällen Load-After-Load und Load-After-Store, was bedeutet, daß eine Load-Operation den Wert einer Load- bzw. Store-Operation einer vorherigen Iteration erneut nutzt. Im folgenden wird nur der Fall Load-After-Store betrachtet, Load-After-Load ergibt sich analog.

Load *vor* Store im Schleifenkörper, aber Load-After-Store:

Vorher:

$$
\begin{array}{l}
\mathtt{for(i = 1; i < UB; i++)}\\
\{\\
\quad \mathtt{x} = \mathtt{X_1[a_1 \times i + b_1]};\\
\quad \vdots\\
\quad \mathtt{X_2[a_2 \times i + b_2]} = \mathtt{y};\\
\}
\end{array}
$$

mit:
$X_1 = X_2$,
$a_1 = a_2 = a$, $b_1 < b_2$,
$d := \frac{b_2 - b_1}{a} > 1$

Nachher:

$$
\begin{array}{l}
\mathtt{t^{(d)}} = \mathtt{X_1[a_1 \times 1 + b_1]}\\
\vdots\\
\mathtt{t^{(1)}} = \mathtt{X_1[a_1 \times d + b_1]}\\
\mathtt{for(i = 1; i < UB; i++)}\\
\{\\
\quad \mathtt{x} = \mathtt{t^{(d)}}\\
\quad \vdots\\
\quad \mathtt{t} = \mathtt{y};\\
\quad \mathtt{X_2[a_2 \times i + b_2]} = \mathtt{t};\\
\quad \vdots\\
\quad \mathtt{t^{(d)}} = \mathtt{t^{(d-1)}}\\
\quad \vdots\\
\quad \mathtt{t^{(1)}} = \mathtt{t}\\
\}
\end{array}
$$

Auch hier ist wiederum darauf zu achten, daß zwischen den Referenzen konstante Iterationsdistanzen liegen, damit die Register-Pipeline eine konstante Länge erhält.

Beispiel 5.3.2 *Auswahl zwischen verschiedenen Datenquellen*

```
for(i = 1; i < N; i++)
{
   a[i] = x;
   y = a[i-3];
   z = a[i-5];
}
```

Stehen zur Elimination eines redundanten Loads mehrere verschiedene Datenquellen bereit, so ist diejenige zu bevorzugen, die zur kürzesten Register-

Pipeline führt. In diesem Zusammenhang ist es Vorteilhaft, wenn mehrere redundante Loads entfernt und durch eine RP behandelt werden können. Beispiel 5.3.2 zeigt eine Schleife mit zwei redundanten Loads. Es soll mit der Elimination von `a[i-5]` angefangen werden. Als Datenquellen für eine RP stehen sowohl `a[i]` als auch `a[i-3]` bereit. Die erste Referenz führt zu einer längeren RP als die zweite, also sollte man eine RP von `a[i-3]` nach `a[i-5]` erzeugen. Da aber auch `a[i-3]` redundant ist, kann eine RP von `a[i-5]` nach `a[i]` konstruiert werden, die „unterwegs“ den Gebrauch `a[i-3]` ersetzt und mit Werten versorgt.

5.3.3 Vor- und Nachteile

Eine Register-Pipeline stellt ein komplexes Verfahren zur Elimination redundanter Speicherzugriffe dar, welches selbst neben dem gewünschten Effekt der Beschleunigung Operationen in den Schleifenkörper einfügt. Diese arbeiten der Beschleunigung durch ihre eigene Ausführung entgegen. Es bleibt aber nicht nur bei diesen zwei gegenläufigen Effekten, der Verbrauch an Registern und das Einfügen von Instruktionen stehen in dichter Wechselwirkung mit anderen Optimierungen und der Zielarchitektur, so daß sich die genauen Auswirkungen im Einzelfall oft nur schwer voraussagen lassen. Folgende Vor- und Nachteile lassen sich ausmachen:

- **Vorteile**
  - Entfernung eines redundanten Array-Lesezugriffs
- **Nachteile**
  - Einführung einer Reihe von temporären Variablen
  - Vergrößerung des Code-Umfangs durch Schleifenprolog
  - Vergrößerung des Schleifenkörpers durch Kopieroperationen zur Verwaltung der RP

Die Liste der Nachteile einer RP wirkt schwer. Es muß gründlich überlegt werden, ob und unter welchen Bedingungen der Einsatz einer RP Vorteile erbringt. Mit Sicherheit kann die vorgestellte einfache Variante des Register-Pipelinings nur bei Prozessoren mit homogenen Registersätzen und einer großen Anzahl an Registern für die allgemeine Verwendung Nutzen erbringen. Ansonsten stehen die für den Datentransport verwendeten Register nicht zur Verfügung. Ob die eingefügten Kopieroperationen den Gewinn durch die Elimination des redundanten Zugriffs auf- oder überwiegen, hängt von der Tiefe der Pipeline ab, weil damit die Anzahl der temporären Variablen und der Kopieroperationen zunimmt. Die Vergrößerung des Code-Umfangs durch den Schleifenprolog ist i.a. von untergeordneter Bedeutung, da sich dort nur eine Reihe von Lade-Operationen ansammeln. Schwerer wiegt der erhöhte Registerdruck im Schleifenkörper, der im ungünstigsten Fall zum Spilling führen kann, und damit auch die Anzahl der Speicherzugriffe und die Ausführungsdauer erhöht. Kapitel 8 dokumentiert die Ergebnisse, die mit einer praktischen Realisierung der RP-Optimierung gemacht wurden.

Auch eine Register-Pipeline steht in enger Beziehung zu anderen Optimierungen und zur verwendeten Zielarchitektur. Die folgende Liste zeigt mögliche Effekte, die damit zusammenhängen und die Effizienz der Optimierung beeinflussen können.

- **Einfluß durch weitere Optimierungen:**

⊖ Falls der redundante Speicherzugriff in der nicht-optimierten Version der Schleife in seiner Latenz durch andere Operationen versteckt wurde, so kann es dazu kommen, daß es trotz der Elimination zu keiner Beschleunigung kommt, gar aufgrund anderer in dieser Liste aufgeführter Einflüsse die Ausführung *verlangsamt* wird. Größere Effekte sind bei kurzen kritischen Pfaden im Schleifenkörper zu erwarten, da zum Verstecken der Latenzen nur wenige Operationen zur Verfügung stehen.

⊙ Es stehen Verfahren zur effizienten Nutzung der AGU bei Array-Zugriffen in Schleifen bereit, die darauf basieren, in einem *Indexing Graph* kleinste Zyklen und Pfade zu finden, denen jeweils ein Adreßregister der AGU zugewiesen wird [4]. Durch die Elimination eines redundanten Speicherzugriffs kann es zur Zerteilung eines solches Zyklus in einen Pfad, bzw. eines Pfades in zwei kleinere Pfade kommen. Ersteres hat zur Folge, daß am Ende einer Iteration Code zum Laden eines Adreßregisters eingefügt werden muß, während letzteres zur Verwendung eines weiteren Adreßregisters (inklusive Initialisierung und Update am Ende des Schleifenkörpers) führt. Beides verlangsamt die Ausführung, insbesondere wenn es über das Update hinaus noch zu einem Spilling der Adreßregister kommt. Die umgekehrte Wirkung ist jedoch auch möglich, denn durch die Elimination eines Speicherzugriffs kann das Schließen eines Pfades zu einem Zyklus ermöglichen. Damit kann der AR-Update-Code eingespart werden. Ebenso können möglicherweise zwei kleinere Pfade zu einem größeren zusammengesetzt werden, wodurch ein Adreßregister eingespart werden kann.

⊙ Die Elimination redundanter Speicherzugriffe kann bei nachfolgender Anwendung von Software-Pipelining einen Beitrag zur Angleichung der Dauern der einzelnen Stufen der Software-Pipeline leisten. Die Effizienz des Software-Pipelinings wird von der längsten Abhängigkeitskette beeinflußt. Kann diese Kette durch die Elimination eines redundanten Speicherzugriffs verkürzt werden, können vorhandene Ressourcen (z.B. ALU) gleichmäßiger ausgenutzt werden. Es kann aber auch zum gegenteiligen Effekt kommen. Falls die Dauern der einzelnen Stufen schon annähernd gleich sind, kann die Entfernung eines Speicherzugriffs eine Stufe verkürzen. Die Laufzeit bleibt nahezu unverändert, da sie durch die Dauer der längsten Stufe bestimmt wird.

$\oplus$ In Verbindung mit Software-Pipelining ergibt sich ein Vorteil durch die explizit in skalaren Variablen ausgedrückte Datenabhängigkeit. Dadurch kann das Software-Pipelining an einigen Stellen aggressiver arbeiten, wo zuvor konservative Annahmen über Array-Datenabhängigkeiten getroffen werden müssen.

$\odot$ Eine erhöhte Flexibilität beim Instruction Scheduling ergibt sich aus der höheren Mobilität der Instruktionen, die von der entfernten Speicheroperation abhängig sind und der Entlastung der Speicher-Ports[3]. In die Lücken der vorher vorhandenen Latenzzeiten lassen sich nun Operationen schieben. Allerdings sammeln sich am Ende des Schleifenkörpers Registerkopier-Operationen, die voneinander abhängig sind. Das erschwert das Instruction Scheduling für die Hardware-Pipeline des Prozessors.

$\odot$ Auf das Umkopieren von Daten am Schleifenende kann völlig verzichtet werden, wenn der Schleifeninhalt um den Faktor der Tiefe der längsten Register-Pipeline abgerollt wird. Dadurch werden loop carried dependences der ursprünglichen Schleife zu loop independent dependences der abgerollten Schleife, bei der keine Daten von einer Iteration in die nächste transportiert werden müssen. Allerdings wird durch Loop Unrolling der Schleifenkörper z.T. erheblich länger.

$\ominus$ Es wird ein Schleifenprolog erzeugt, in dem einige lesende Array-Zugriffe in Folge stehen. Aufgrund der Speicherzugriffslatenzen kann es vorkommen, daß die Hardware-Pipeline des Prozessors für die Dauer der Abarbeitung des Prologs große Lücken aufweist, was eine ineffiziente Bearbeitung bedeutet. Falls jedoch hinreichend viele weitere Instruktionen vor dem Eintritt in die Schleife zur Verfügung stehen, können diese während des Instruction Schedulings zum Verstecken der Latenzzeiten verwendet werden. Nicht nur die Latenzen erschweren die Bearbeitung des Prologs, sondern auch mögliche Speicherbankkonflikte bei Prozessoren die mehrere getrennte Speicherbänke unterstützen.

- **Einfluß durch Zielarchitektur:**

$\ominus$ Einige Signalprozessoren besitzen die Möglichkeit zur *Dual Load Execution*, d.h. die gleichzeitige Durchführung von zwei Speicherzugrif-

[3] vgl. [9]

fen. Falls während des Register-Pipelinings ein einzelner Zugriff einer solchen Operation eliminiert wird, so muß immer noch der verbleibende Zugriff ausgeführt werden. Wenn die weiteren Operationen von dem Wert dieses Zugriffs abhängig sind, so kann keine große Beschleunigung vom Register-Pipelining erwartet werden.

⊖ In jeder Iteration müssen Instruktionen zur Verwaltung der Register-Pipeline ausgeführt werden. Bei einer einfachen Register-Pipeline sind dies Registerkopieroperationen, deren Anzahl proportional mit der Tiefe der Pipeline wächst. Damit ist klar, daß eine Register-Pipeline nur dann effizient arbeiten kann, d.h. eine Beschleunigung gegenüber einem Speicherzugriff herbeiführen, wenn die Kosten des Verwaltungsoverheads pro Iteration geringer sind als die Kosten eines Speicherzugriffs. [4]

5.4 Erweiterte Möglichkeiten

Zu den bisherigen Grundversionen der Optimierungen RSE/RLE/RP gibt es zahlreiche Erweiterungsmöglichkeiten. Weitere Möglichkeiten beim Einsatz der RLE stehen im nächsten Abschnitt im Vordergrund, danach soll die Behandlung von Loop Nests und mehrdimensionalen Arrays beim Register-Pipelining vorgestellt werden.

5.4.1 Weitere Fälle zur RLE

Eine weitergehende Unterscheidung der Einsatzfälle der RLE mit weiteren Verbesserungen der Optimierung findet sich in [9]. Zum einen handelt sich dabei um schleifeninvariante Ausdrücke, die aus der Schleife herausgenommen werden können. Zum anderen soll durch Verschieben der redundanten Operation dann folgender Aufhebung des redundanten Zugriffs versucht werden, die temporäre Variable zum Transport des Datums einzusparen.

- **Load-After-Load Optimierung**
  Die Load-After-Load Optimierung entspricht der erläuterten Grundvariante, wenn sowohl l als auch r Lese-Operationen sind. Es kann gezeigt werden, daß die Load-After-Load Optimierung die Anzahl benötigter Register nicht vergrößert.

- **Load-After-Store Optimierung**
  Die Load-After-Store Optimierung versucht, die einem Gebrauch folgende

[4] Eine ähnliche Kostenbetrachtung läßt sich auch bezüglich des Stromverbrauchs durchführen. Eine Register-Pipeline ist dann als effizient zu betrachten, wenn der Stromverbrauch der Registerkopieroperationen geringer ist als der Stromverbrauch eines Speicherzugriffs.

Definition eines Array-Elementes durch eine Registerkopieroperation zu ersetzen. Dabei kann es zum *Read-Wrong-* oder *Write-Live-*Konflikt kommen. Ein Read-Wrong-Konflikt liegt vor, wenn zwischen Definition und Gebrauch eine Redefinition des transportieren Registerinhaltes erfolgt. Zu einem Write-Live-Konflikt kommt es, wenn zwischen Array-Definition und Array-Gebrauch zu einer Definition des Zielregisters der Load-Operation kommt. Die möglichen Konflikte verhindern nicht die Optimierung, sie verhindern lediglich das Verschieben der betroffenen Operationen zueinander hin. Ohne Konflikte kann einerseits der Store hin zum Load vorgezogen oder andererseits das Load zum Store verschoben werden. Damit könnte eine temporäre Variable eingespart werden. Im Konfliktfall ist diese unverzichtbar. Abbildung 5.4.1 zeigt einen Schleifenkörper vor und nach der Optimierung. Die Optimierung ist möglich, weil zwischen den Store und dem Load keine Redefinition von `a[i]` und `b` erfolgt. Ist dies der Fall, so liegt ein *Read-Wrong-*Konflikt vor.

Beispiel 5.4.1 *Load-After-Store-Optimierung und mögliche Konflikte*

Vorher:

```
a[i] = b;
...
c = a[i];
```

Nachher:

```
a[i] = b;
c = b;
...
```

Read-Wrong-Konflikt

```
a[i] = b;
b = d;
...
c = a[i];
```

Write-Live-Konflikt

```
a[i] = b;
c = d;
...
c = a[i];
```

- **Entfernung von Invarianten**
  Schleifeninvariante Array-Referenzen sind Lese- und Schreiboperationen, deren Indexfunktionen nicht von der Induktionsvariablen abhängig sind, und somit im ganzen Iterationsintervall konstant sind. Handelt es sich um einen lesenden Zugriff, so kann dieser in den Schleifenprolog gezogen werden; ein schreibender Zugriff wird in den Epilog geschoben:

Vorher:

```
for(j = 1; j < UB1; j++)
   for(i = 1; i < UB2; i++)
      a[j] = a[j] + b[i];
```

Nachher:

```
for(j = 1; j < UB1; j++)
{
   t = a[j];
   for(i = 1; i < UB2; i++)
   {
      t' = t + b[i];
   }
   a[j] = t';
}
```

Treten sowohl invariante Lese- als auch Schreibzugriffe auf, so ist diese Verfahren nur dann korrekt, wenn alle Invariante aus der Schleife entfernt werden können.

5.4.2 Behandlung mehrdimensionaler Arrays und Loop Nests

Bislang behandeln die RSE/RLE/RP-Optimierungen nur eindimensionale Arrays und einfache Schleifen. Nachdem im Kapitel 4.2.6 Verfahren zur Datenflußanalyse von mehrdimensionalen Arrays in *Tight Loop Nests* gezeigt wurden, sollen nun Verfahren zur Nutzung der Analyseergebnisse vorgestellt und bewertet werden. Die Anwendung wird in Zusammenhang mit dem Register-Pipelining gezeigt, die Anwendung für RLE/RSE ergibt sich daraus in analoger und vereinfachter Form.

Grundsätzlich ist zwischen drei Fällen zu differenzieren:

1. Ausschließliche Abhängigkeit von der inneren Induktionsvariablen,
2. ausschließliche Abhängigkeit von einer umgebenden Induktionsvariablen, und
3. Abhängigkeit von mehreren Induktionsvariablen.

Wenn ausschließlich Abhängigkeiten bzgl. der inneren Induktionsvariablen vorliegen, so kann das Verfahren des einfachen Register-Pipelinings unverändert übernommen werden, siehe dazu Beispiel 5.4.2

Beispiel 5.4.2 *Register-Pipeline bei mehrdimensionalem Array und Abhängigkeit von der* inneren *Induktionsvariablen*

Ursprüngliche Schleife

```
for{j = 1; j < M; j++}
  for(i = 1; i < N; i++)
  {
     x = a[i+1][j];
     ...
     y = a[i][j];
  }
```

Schleife mit Register-Pipeline

```
for(j = 1; j < M; j++)
{
   t' = a[1][j];
   for(i = 1; i < N; i++)
   {
      t = a[i+1][j];
      x = t;
      ...
      y = t';
      t' = t;
   }
}
```

Bei einer Abhängigkeit von einer äußeren Induktionsvariablen kann das Optimierungsverfahren nur nach einer Anpassung auf die Mehrdimensionalität verwendet werden. Dazu ist zu beachten, daß bei einem n-dimensionalem Array mit entsprechender Schachtelungstiefe des Loop Nests und einer Abhängigkeit zwischen Iterationen der k-ten Induktionsvariablen temporäre Variable der Dimension $k-1$ verwendet werden müssen. Anstatt skalarer temporärer Variablen kommen nun also $k-1$-dimensionale temporäre Arrays zum Einsatz. Zur Veranschaulichung des zweidimensionalen Falls dient Beispiel 5.4.3:

Beispiel 5.4.3 *Register-Pipeline bei mehrdimensionalem Array und Abhängigkeit von einer* umgebenden *Induktionsvariablen*

Ursprüngliche Schleife

```
for{j = 1; j < M; j++}
  for(i = 1; i < N; i++)
  {
     x = a[i][j+1];
     ...
     y = a[i][j];
  }
```

Schleife mit Register-Pipeline

```
t' = a[1];
for(j = 1; j < M; j++)
{
   for(i = 1; i < N; i++)
   {
      t[i] = a[i][j+1];
      x = t[i];
      ...
      y = t'[i];
   }
   t' = t;
}
```

`t` und `t'` sind in diesem Beispiel eindimensionale Arrays der Größe N. Der Schleife ist wie üblich ein Prolog vorangestellt, jedoch handelt es sich bei der Initialisierung `t' = a[1]` nun um die Zuweisung der ersten Zeile (j=1) des Arrays `a` an `t'`. Im inneren Schleifenkörper können dann einzelne Elemente des temporären Array referenziert werden, und so Zugriffe auf das ursprüngliche, höher-dimensionale Array vermeiden. Nach Beendigung der inneren Schleife findet das Verschieben der Register-Queue, wobei die Instruktion `t' = t` nun für das Kopieren des temporären Arrays `t` nach `t'` steht. Diese Kopieroperation sollte aus Gründen der Effizienz ebenso wie die Initialisierung nicht durch elementweises Kopieren erfolgen, sondern durch Umsetzen von Zeigern, so daß die Anzahl der dafür benötigten Instruktionen klein gehalten werden kann.

Erheblich aufwendiger wird die Optimierung bei der Abhängigkeit von mehreren Induktionsvariablen. Es ist auf den korrekten Transport der Werte zwischen äußeren und inneren Iterationen in mehrdimensionalen Arrays zu achten, wobei bei manueller Optimierung schon bei kleinen Dimensionen schnell der Überblick verloren geht. Daher wird anstelle einer allgemeinen Lösung auf ein zweidimensionales Beispiel zurückgegriffen (Beispiel 5.4.4).

Zwischen den äußeren Iterationen werden die Werte mit den Arrays `t` und `t'` transportiert. In der Schleife wird die temporären Arrays dann aber mit der inneren Iterationsvariablen indiziert, um den benötigten Wert an der passenden Stelle zu referenzieren. Die der inneren Schleife folgenden Iterationen dienen zum einen zum Transfer der Daten von `t` nach `t'`, zum anderen müssen aber nicht im Schleifenkörper gelesene Elemente aus dem ursprünglichen Array, dessen Zugriffe vermindert werden sollen, separat in `t'` gespeichert werden.

Beispiel 5.4.4 *Register-Pipeline bei mehrdimensionalem Array und Abhängigkeit von* innnerer und umgebender *Induktionsvariablen*

Ursprüngliche Schleife

```
for{j = 1; j < M; j++}
  for(i = 1; i < N; i++)
  {
     x = a[i+1][j+1];
     ...
     y = a[i][j];
  }
```

Schleife mit Register-Pipeline

```
t' = a[1];
for(j = 1; j < M; j++)
{
   for(i = 1; i < N; i++)
   {
      t[i+1] = a[i+1][j+1];
      x = t[i+1];
      ...
      y = t'[i];
   }
   t' = t;
   t'[1] = a[1][j];
}
```

Die Verwendung von RP bei mehrdimensionalen Arrays hat Vor- und Nachteile, daher ist vor der Verwendung anhand der folgenden Liste abzuschätzen, mit welchen Folgen ein Einsatz behaftet sein kann:

- **Vorteile**
  - Bei Abhängigkeiten von *einer* äußeren Variablen kommt es zur Einführung von temporären Arrays. Somit werden redundante Array-Zugriffe durch andere Array-Zugriffe ersetzt. Sofort kommt die Frage auf, ob ein solches Vorgehen sinnvoll ist. In Verbindung mit im nächsten Abschnitt vorgestellten Techniken zur Nutzung der AGU und des On-Chip-RAMs kann diese Frage bejaht werden, denn bei Ersetzung von redundanten Array-Lesezugriffen durch Zugriffe auf ein temporäres Array im On-Chip-RAM kann durch dessen Geschwindigkeitsvorteil durchaus eine Laufzeitverbesserung erzielt werden. Dabei ist aber darauf zu achten, daß die Anzahl der temporären Arrays sowie ihre Größe [5] klein bleibt, damit sie auch im On-Chip-RAM untergebracht werden können. Desweiteren empfiehlt sich die Unterstützung der AGU zu Adressierung der Arrays, da eine explizite Adreßberechnung zu zeitaufwendig ist.
- **Nachteile**
  - Die Verwendung von RP bei Abhängigkeiten bzgl. der inneren Induktionsvariablen gilt prinzipiell das zu eindimensionalen Arrays gesag-

[5] evtl. durch Begrenzung der Tiefe der behandelten Abhängigkeiten sowie durch Schleifenvertauschen oder andere Schleifentransformationen zu erreichen

te. Hinzu kommt der Effekt der Vergrößerung des Schleifenkörpers der umgebenden Schleife, was bei Vorhandensein eines Instruktions-Caches zu erhöhtem Auftreten von Cache-Misses führen kann.

– Bei Abhängigkeit von *mehreren* Induktionsvariablen verliert die Optimierung ihre Effizienz. anders. Grundsätzlich können zwar redundante Lesezugriffe vermieden werden, doch ist das Verfahren sowohl schwer zu implementieren (und verifizieren) als auch aufwendig zur Laufzeit. Selbst bei dem Vorhandensein einer großen Anzahl redundanter Zugriffe dürfte sich die Anwendung kaum lohnen, da der Overhead insbesondere bei größeren Iterationsdistanzen sehr beträchtlich wird, denn mit wachsender Iterationsdistanz werden in der Richtung dieses Wachstums mehr Update-Operationen benötigt, die auch wieder Array-Zugriffe verursachen.

Unter dem Aspekt fehlender effizienter Optimierungen von redundanten Zugriffen bei Abhängigkeiten von mehreren Induktionsvariablen erscheint eine weitere Bemühung um eine Verbesserung der Datenflußanalyse-Techniken für mehrdimensionale Arrays zunächst wenig attraktiv.

Möglicherweise ist das Abrollen (Loop Unrolling) der inneren Schleife(n) eine bessere Alternative zum RP bei mehrdimensionalen Abhängigkeiten. Anschließend könnten einfachere Datenflußanalysen auf den dann teilweise konstanten Array-Referenzen arbeiten und so diverse Optimierungen (Common Subexpression Elimination, Loop Invariant Code Motion, Copy und Constant Propagation etc.) ermöglichen. Schleifentransformationen wie z.B. Schleifenverwinden (Loop Skewing) könnten dazu benutzt werden, die diagonalen Distanzvektoren in die Richtung der Einheitsvektoren zu bringen, was einer Dimensionsreduktion der Datenabhängigkeit entspricht. Anschließend könnte das beschriebene Verfahren evtl. erfolgreicher anzuwenden sein.

5.5 Beurteilung der verschiedenen Optimierungen

RSE und RLE sind in ihren Grundformen interessante Optimierungen, die durch ihren Einsatz häufig zu einer Verbesserung des Laufzeitverhaltens eines Programms bezüglich Geschwindigkeit und Anzahl der Speicherzugriffe führen. Sowohl RSE und RLE sollten bei einer Zielarchitektur mit einem homogenen Registersatz weniger Schwierigkeiten bereiten als bei heterogenen Registersätzen. Grundsätzlich besteht eine Eignung, der Erfolg hängt aber von vielen – in jeweiligen Kapiteln erläuterten – Faktoren ab. Bei heterogenen Registersätzen kommt es sehr auf den Einzelfall an, es sollten auch dort erfolgversprechende Anwendungsfälle zu finden sein.

Register-Pipelining ist in der Grundvariante nur bei Zielarchitekturen mit großen homogenen Registersätzen anwendbar, dann aber auch nur für vergleichsweise kleine Iterationsdistanzen. Wird die Tiefe der Register-Pipeline zu groß, so nehmen die Kosten der Verwaltungsoperationen stark zu, so daß der Nutzen der Optimierung verloren geht. Die Behandlung von mehrdimensionalen Arrays durch RP ist nur bei Abhängigkeiten in einer Richtung sinnvoll. Wenn diese eine Richtung durch eine Induktionsvariable einer äußeren Schleife vorgegeben wird, ist schon zu überlegen, ob der Aufwand gegenüber dem erwarteten Nutzen gerechtfertigt ist. Für eine innere Induktionsvariable kann das Verfahren ebenso wie im eindimensionalen Fall bei kurzen Abhängigkeitsketten gut verwendet werden.

Kapitel 6

Erweiterte Load/Store-Optimierungen

In diesem Kapitel sollen weitere Load/Store-Optimierungen vorgestellt werden, die über die Möglichkeiten der vorangegangen Verfahren hinausgehen. Es handelt sich dabei um verschiedene Varianten des Register-Pipelinings, die eine höhere Effizienz als das Basisverfahren erzielen. Das wird beim *verbesserten Register-Pipelining* durch die Einbeziehung der Elemente der RP in die Registerallokation erzielt, während das *optimale Register-Pipelining* basierend auf einer genaueren Analyse auch partiell redundante Zugriffe eliminiert werden. Insgesamt kann bzgl. einiger Kriterien eine Optimalität erlangt werden. Anschließend soll eine Implementierungsmöglichkeit der RP vorgestellt werden, die von den besonderen Hardware-Eigenschaften von DSP Gebrauch macht. Durch die Verwendung der AGU und des On-Chip-RAMs kann eine Register-Pipeline effizient als Ringpuffer realisiert werden.

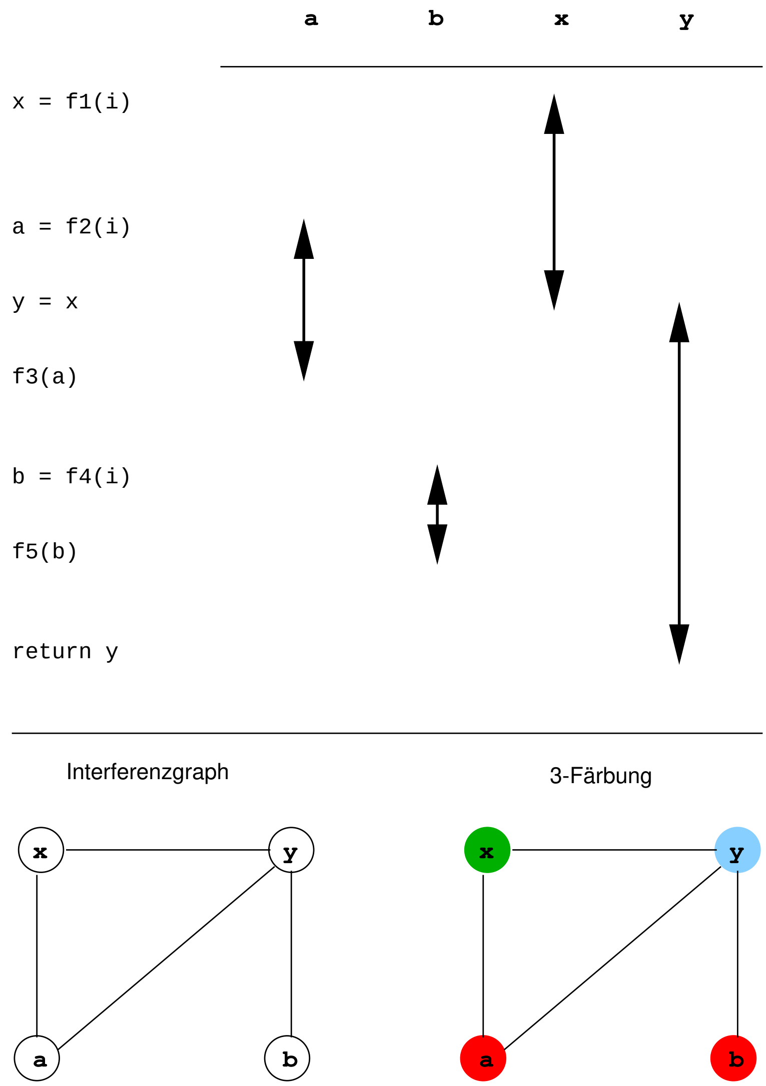
6.1 Verbessertes Register-Pipelining

Die Verwendung einer einfachen Register-Pipeline ist mit dem Nachteil behaftet, daß die RP mit einem Satz an temporären Variablen realisiert wird. Deren Registerzuweisung wird einer späteren Registerallokationsphase überlassen. Es stellt sich als äußerst ungünstig heraus, wenn es zum Spilling kommt und einzelne Elemente einer RP ein- und ausgelagert werden müssen. Vorteilhaft ist die gemeinsame Registerallokation von skalaren Variablen und Elementen einer RP. In [8] wird zur Bewältigung dieser Aufgabe ein Verfahren vorgestellt, über das ein kurzer Überblick gegeben werden soll.

Die Voraussetzungen des Verfahrens sind die gleichen wie auch bei einer einfachen Register-Pipeline. Die durch Analysen bereitgestellten Informationen werden an dieser Stelle besser ausgenutzt als zuvor.

Gewöhnliche Registerallokationsverfahren für skalare Variable bestimmen zunächst die Lebensdauer von Variablen für einen betrachteten Programmausschnitt. Aus den Lebensdauern wird ein *Interferenz-Graph* erzeugt, dessen Knoten den Variablen entsprechen. Zwischen zwei Knoten befindet sich eine Kante, falls sich die Lebensdauern der zugehörigen Variablen überschneiden, d.h. miteinander interferieren. Für eine feste Anzahl k an Registern wird dann versucht, eine k-Färbung des Graphen zu erzielen. Dabei entspricht jeder Farbe ein Register.

Beispiel 6.1.1 *Lebensdauern, Interferenzgraph und 3-Färbung*



Beispiel 6.1.1 zeigt für einen kurzen Programmausschnitt die Lebensdauern von vier darin enthaltenen Variablen `a`, `b`, `x` und `y`. Der Interferenzgraph verbindet `a`, `x` und `y` miteinander, denn ihre Lebensdauern überschneiden sich. Die Lebensdauer von `b` überschneidet sich nur mit der von `y`, so daß im Graphen auch

nur eine Kante zwischen `b` und `y` enthalten ist. Für $k = 3$ verfügbare Register wird eine 3-Färbung des Interferenzgraphen vorgenommen. Eine mögliche 3-Färbung ist unten rechts zu sehen. Danach können `a` und `b` demselben Register zugewiesen werden, ohne miteinander in Konflikt zu geraten.

Um eine Register-Pipeline mit in die Registerallokation einzubeziehen, müssen die Lebensdauern von Array-Elementen bekannt sein. Mit konventionellen, skalaren Datenflußanalyse-Verfahren können diese nicht bestimmt werden, dafür aber mit einer Array-Datenflußanalyse. Das δ-Verfahren aus Kapitel 4.2 kann in der gleichen Weise wie bei der Erzeugung der einfachen RP parametrisiert werden, um die Lebensdauern von Array-Elementen zu ermitteln. Die Information der *δ-available values*-Analyse wird in Kapitel 5.2benutzt, zu ermitteln ob ein Array-Element nach einer Anzahl Iterationen noch zur Verfügung steht und um daraus die Länge der RP zu bestimmen. Die gleiche Information kann genutzt werden, um die Lebensdauer eines Array-Elementes in Iterationen zu errechnen. Die maximale Distanz mit der ein Array-Element verfügbar ist, ist dessen Lebensdauer.

6.1.1 Integrierter Interferenzgraph und dessen Färbung

Mit der Kenntnis der Lebensdauern von Array-Elementen kann ein Interferenzgraph konstruiert werden, der sowohl skalare Variable als auch Array-Elemente berücksichtigt. Es entsteht ein *integrierter Register-Interferenzgraph (IRIG)*. Zu dessen Färbung wird ein Algorithmus genutzt, der zusätzlich Prioritäten verwendet. Die Prioritäten gegen an, mit welchem Kosten/Nutzen-Verhältnis die Zuweisung einer Variablen an ein Register verbunden ist. Bei dem Fall, daß weniger Register vorhanden sind als durch Variablen zu belegen sind, können dadurch diejenigen Variablen ausgewählt werden, deren Registerallokation den größten Nutzen verspricht.

Die Prioritätsfunktion $P(l)$ zu einer Variablen l verwendet die Anzahl der Wiederverwendungen $access(l)$ von l über die Lebensdauer, die Lebensdauer $|l|$ selbst, die Tiefe $depth(l)$ der zu l gehörenden Pipeline und die Kosten C_{LD} eines lesenden Speicherzugriffs. Die Priorität ergibt sich zu

$$P(l) = \frac{(access(l) - 1) \times C_{LD}}{|l| \times depth(l)} \tag{6.1}$$

Die Priorität entspricht dem Verhältnis der Gesamtkosten lesender Speicherzugriffe zum Aufwand der RP ausgedrückt durch deren Länge. l geht zur „Normalisierung" des Ausdrucks ein. Da für skalare Variable keine Register-Pipeline geführt wird, muß $depth(l)$ danach unterscheiden:

$$depth(l) = \begin{cases} 1 & \text{,falls } l \text{ eine skalare Variable ist} \\ \delta_0(l) + 1 & sonst \end{cases} \tag{6.2}$$

In dieser Formel gibt $\delta_0(l)$ für ein Array-Element die maximale Iterationsdistanz zwischen der Erzeugung und einem Gebrauch von l an. Das entspricht genau der Lebensdauer. Für skalare Variablen wird der Wert eins zurückgegeben, denn es wird ein Register zum Aufbewahren benötigt.

Zur Färbung des mit Prioritäten versehenen Interferenzgraphen gibt es Standardverfahren, die darauf angepaßt werden müssen, daß Variable in einem Knoten wegen einer Register-Pipeline mehr als ein Register verlangen können. Die Standardverfahren unterteilen die Knotenmenge in *erfüllbare* und *unerfüllbare* Knoten. Erfüllbare Knoten haben weniger durch Kanten verbundene Nachbarn als verfügbare Register vorhanden sind, und unerfüllbare Knoten haben mehr Nachbarn als Register. Die unerfüllbaren Knoten werden zerteilt mit dem Ziel, alle Knoten erfüllbar zu machen. Die Schwierigkeit bei IRIGs liegt in der Entscheidung, ob ein Knoten unerfüllbar ist, denn es reicht nicht mehr aus, die Anzahl seiner Kanten mit der Anzahl der Register zu vergleichen. Zur Entscheidung müssen die Anzahl der durch den Knoten verlangten Register ebenso wie die der Nachbarknoten berücksichtigt werden. Bei k Registern ist ein Knoten erfüllbar, wenn gilt

$$depth(n) + \sum_{\text{m ist Nachbar von n}} depth(m) \leq k \tag{6.3}$$

Erfüllbare Knoten können mit $depth(n)$ Farben gefärbt werden. Mit dem so geänderten Verfahren kann eine passende Vielfärbung des Graphen gefunden werden, die direkt zur Registerallokation genutzt werden kann.

Nach der Registerallokation kann eine neue Schleife mit der Register-Pipeline erzeugt werden. Die Schritte sind prinzipiell die gleichen wie auch bei einer einfachen RP. Ein Schleifenprolog zur Initialisierung wird erzeugt, und redundante Gebräuche von Array-Elementen werden durch Zugriffe auf die entsprechenden Elemente der RP ersetzt. An das Schleifenende werden zusätzliche Registerkopier-Operationen gestellt, die dafür sorgen, daß beim Übergang in die nächste Iteration alle Inhalte der RP einen Schritt weiterbewegt werden.

6.2 Optimales Register-Pipelining

Die Grundform des Register-Pipelinings aus Kapitel 5.3 und auch das verbesserte Register-Pipelining aus Kapitel 6.1 haben den Nachteil, ausschließlich total redundante Loads zu betrachten. Partiell redundante Loads werden nicht optimiert. Die Registerkopieroperationen der RP werden an das Ende des Schleifenkörpers gestellt, auch wenn durch eine andere Plazierung Aufwand gespart werden könnte. Die Nachteile des bisherigen Verfahren entstehen aus der unpräzisen Information, die durch die verwendete δ-Analyse geliefert wird. Eine Analyse, die in der Lage ist, Information auf feinerer Ebene bereitzustellen, sollte dazu genutzt werden, die Leistungsfähigkeit einer Register-Pipeline zu erhöhen.

Mit dem *Stretched Loop*-Verfahren aus Kapitel 4.3 steht ein in der Präzision verbessertes Datenflußanalyse-Verfahren zur Verfügung. Durch dessen Nutzung kann das Register-Pipelining mit entsprechendem Aufwand zur Analyse und Durchführung der Optimierung zu einer gewissen Art von Optimalität weiterentwickelt werden. In [6] wird dazu eine Methode vorgestellt, um eine optimale Plazierung von Load-, Store- und Registerkopier-Operationen zu erzielen. Insbesondere zeichnet sich das *optimale Register-Pipelining* durch folgende Eigenschaften aus:

1. Die Anzahl der Load- und Store-Operationen entlang aller Kontrollflußpfade durch die Schleife wird minimiert.
2. Die Lebensdauern von Array-Elementen in der Schleife sind minimal.
3. Die durchschnittliche Anzahl der Registerkopier-Operationen ist minimal.
4. Die Anzahl virtueller Register (bzw. temporärer Variable), die für eine bestimmte Lebensdauer eingeführt werden, ist minimal.

6.2.1 Algorithmus

Der Optimierungsalgorithmus vollzieht seine Aufgaben nicht allesamt aufeinmal, sondern führt sie sequentiell in fünf aufeinanderfolgenden Schritten aus:

1. Plazierung von Load-Operationen
2. Plazierung von Store-Operationen
3. Plazierung von Registerkopier-Operationen und Einführung virtueller Registernamen
4. Erzeugung von Schleifenprolog und -epilog
5. Registerallokation

Die ersten drei Schritte benötigen zu ihrer Ausführung verschiedene Array-Datenflußanalysen. Durch sie werden die Informationen geliefert, die zur Erzielung der Optimalität bei der Plazierung notwendig ist. Schritt 4 sorgt dafür, daß sich die optimierte Schleife immer im stabilisierten Zustand befindet. Der letzter Schritt sorgt für die Registerallokation, d.h. der Zuweisung von Variablen an verfügbare Register des Prozessors.

Im folgenden sollen diese Schritte im einzelnen erläutert werden, jedoch nur soweit wie es zum Verständnis der Arbeitsweise notwendig erscheint. Details können in der Original-Publikation [6] nachgelesen werden. Die Registerallokation bleibt in dieser Darstellung ein wenig vernachlässigt, da Registerallokationsstrategien nicht Teil dieser Arbeit sein sollten. Auch hier gibt die referenzierte Publikation genauere Darstellung und Hinweise auf weitere Literatur.

Plazierung von Load-Operationen

Ziel der Load-Plazierung ist es, Werte so früh wie möglich zu laden, damit sie für folgende Operationen verfügbar sind. Dabei muß darauf geachtet werden, daß keine zusätzlichen partiell redundanten Loads erzeugt werden, weil z.B. ein Load nur über einen Pfad im CFG ausgeführt wird. Im Sinne einer Verkürzung der Lebensdauer des geladenen Wertes sollte das Load aber auch nicht zu weit vorgezogen werden.

Zunächst wird bei diesem Ansatz die Einheit von Definition/Gebrauch und Speicherzugriff aufgelöst, z.B. eine Definition berechnet einen Wert, der fortan in einem *Register* verfügbar ist, aber er befindet sich noch nicht – auch nicht im Falle einer Array-Referenz – in einer adressierten Speicherzelle. Dazu ist eine Store-Operation notwendig, die explizit den Wert aus dem Register in eine Speicherzelle schreibt. Bei einer nicht-optimierten Programmversion sind Array-Definition/Gebrauch und Speicherzugriff immer miteinander verbunden, so daß eine Reihe von Speicherzugriffen redundant sind. Beispiel 6.2.1 zeigt zunächst ein Programmfragment mit vereinten Definitionen und Speicherzugriffen, die im zweiten Teil dann getrennt sind.

Beispiel 6.2.1 *Gebrauch/Definition und Speicherreferenz vereint und getrennt*

Vereint:

```
a[i+1] = x + y;
     // Definition, Referenz
...
z = a[i+1] + 3;
     // Gebrauch, Referenz
```

Getrennt:

```
R1 = x + y         // Definition
store R1,a[i+1]    // Speicherzugriff
...
load R2,a[i+1]     // Speicherzugriff
z = R2 + 3         // Gebrauch
```

Beispiel 6.2.2 *Partiell redundantes Load*

Vorher:

```
for(i = 1; i < UB; i++)
{
   load R1, a[i]
   use R1;
   if <condition> then
      load R2, a[i+1]
      use R2;
   else
      load R3, b[k];
      use R3;
}
```

Nachher:

```
load R2, a[1];
for(i = 1; i < UB; i++)
{
   use R2;
   if <condition> then
      load R2, a[i+1];
      use R2;
   else
      load R3, b[k];
      use R3;
      load R2, a[i+1];
}
```

Ein Lesezugriff ist z.B. dann partiell redundant, wenn in einer vorherigen Iteration in einem Ast einer Verzweigung ein später erneut benutzter Wert bereits gelesen und zwischenzeitlich nicht verändert wird, im anderen Ast jedoch nicht. Eine Lösung für diesen Fall ist das Plazieren eines zusätzlichen Loads in dem zweiten Verzweigungsast, damit der Wert in *jedem* Fall zur Verfügung steht (siehe dazu auch Beispiel 6.2.2).

Das Ziel bei der Plazierung von Loads sollte also sein, Stellen zu finden, an denen ein Load keine partielle Redundanz hervorruft. Die Strategie dazu sieht wie folgt aus: Loads müssen in der Stretched Loop so früh wie möglich plaziert werden, damit spätere Gebräuche den geladenen Wert wiederverwenden können. Der Ort der Initialisierung heißt *Initialisierungspunkt*. Dabei dürfen aber nur solche Stellen zur Plazierung ausgewählt werden, von denen aus sich ein zwingender Gebrauch des Wertes ergibt. Dieser Bedingung entspricht das Datenflußproblem *must-anticipability*. Wenn der betreffende Wert am *Initialisierungspunkt* immer verfügbar ist, entsprechend dem Datenflußproblem *must-availability*, dann kann auf das Load verzichtet werden. Solche Stellen heißen *frühste Initialisierungspunkte*. Von den *frühsten Initialisierungspunkten* sollte der späteste ausgewählt werden, entsprechend dem Datenflußproblem *delayability*, um die Lebensdauer des Wertes und damit die Belegung eines Registers zu verkürzen. Diese Stellen nennen sich schließlich *späteste Initialisierungspunkte*. Weiter kann ein Load nicht verzögert werden.

Im Algorithmus zur Load-Plazierung werden drei *Stretched-Loop*-Datenflußanalysen durchgeführt. Zwei davon (*Must-Availability-of-Congruent-Uses* und *Must-Availability-of-Congruent-Values*) sind für die Bestimmung der frühesten Initialisierungspunkte notwendig. Die dritte Array-Datenflußanalyse (*Delayability*) unterstützt die Entscheidung, wie weit ein Load „nach hinten“ geschoben werden kann. Damit wird zu den spätesten Initialisierungspunkten gelangt. Wenn diese bekannt sind, können die Definition und das Load plaziert werden.

Da die Initialisierungspunkte entweder Gebräuche oder Definitionen von Array-Elementen sind, muß danach unterschieden werden. Definitionen an spätesten Initialisierungspunkten benötigen kein Load und können den erzeugten Wert gleich in das passende virtuelle Register schreiben.

Der in seiner formalen Darstellung nicht sehr übersichtliche Algorithmus erzielt beweisbar die einleitend genannte optimale Plazierung der Loads und ist in der Original-Publikation [6] in detaillierter Form dokumentiert.

Plazierung von Store-Operationen

Die Plazierung von Store-Operationen erfolgt in sehr ähnlicher Weise wie die Load-Plazierung. Eine der partiell redundanten Loads analoge Form von Stores ist zu berücksichtigen. Stores können möglichweise *partiell/total tot* sein. Nach [6] ist die Eigenschaft partiell/total tot für Stretched Loop wie folgt definiert:

Definition 6.2.1 *Eine Schreib-Operation* `store R,A[i]` *an einer Stelle p im Programm ist* partiell/total tot, *wenn entlang einiger/aller Pfade von p aus jeder mögliche Gebrauch des zuvor geschriebenen Wertes von* `A[i]` *aus dem Register* `R` *gelesen wird.*

Ein Store kann durch ein weiteres Store, daß in einem Verzweigungsast liegt und den zuvor geschriebenen Wert überschreibt, zu einem partiell toten Store werden. In einer Stretched Loop sind partiell tote Stores daran zu erkennen, daß es einen Ausführungspfad zu einem weiteren Store gibt, der das erste Store redundant werden läßt.

Beispiel 6.2.3 *Partiell totes Store*

```
for(i = 0; i < N; i++)
{
   if(cond)
      store a[i],R2;
   else
      store b[i],R2;

   store a[i+1],R1;
}
```

Im Beispiel 6.2.3 ist das Store `store a[i+1],R1` partiell tot, denn der geschriebene Wert wird je nach Verzweigung in der nächsten Iteration durch `store a[i],R2` überschrieben. In der Zwischenzeit könnte ein Gebrauch des Wertes durch einen Registerzugriff auf `R1` erfolgen.

Wenn partiell tote Stores in der Ausführungsreihenfolge weiter nach hinten geschoben werden, kann auf sie vollständig verzichtet werden. Eine *Partial Dead Code Elimination* kann solche Speicheroperationen entfernen.

Das Problem der Erkennung und Behandlung der gesuchten Fälle hat große Ähnlichkeit zu dem zuvor beschriebenen Verfahren für Load-Operationen. Es können analoge Techniken und Datenflußanalysen – in ihrer Arbeitsweise der neuen Situation angepaßt – auch hier angewendet werden. Zu den Details, auch der Parametrisierung der verwendeten *Stretched-Loop*-Datenflußanalyse, sei auf die Original-Publikation [6] verwiesen.

Plazierung von Registerkopier-Operationen und Einführung virtueller Registernamen

Die in den virtuellen Registern gehaltenen Werte einer Register-Pipeline müssen beim Übergang in die nächste Iteration durch Registerkopier-Operationen verschoben werden, damit jede Referenz das gleiche Register in jeder Operation adressieren kann. Bislang wurden die Registerkopier-Operationen am Ende eines Schleifenkörpers plaziert, und das obwohl der Lebensbereich einzelner virtueller Register gar nicht am folgenden Startknoten des Schleifenkörpers beginnt. Eine Registerkopier-Operation außerhalb des Lebensbereichs ist redundant, denn es folgt eine Operation, die den Wert des Registers überschreiben wird.

Statt die Registerkopier-Operationen ans Ende des Schleifenkörpers zu stellen, wird die Idee verfolgt, diese Operationen so über den Schleifenkörper zu verteilen, daß zum einen keine Registerkopier-Operation in eine Lücke eines Lebensbereichs fällt und somit redundant wird, und zum anderen deren durchschnittliche Anzahl pro Iteration minimiert wird. Dazu sind die exakten Lebensdauern der Werte, die durch Array-Referenzen angesprochen werden, notwendig. Mit dem *Stretched-Loop*-Verfahren aus Kapitel 4.3.3 wird die benötigte Präzision erreicht, um die Lebensdauern hinreichend genau zu bestimmten. Die Minimierung der mittleren Anzahl an Kopieroperationen wird dann durch einen Flußalgorithmus auf einem mit Ausführungswahrscheinlichkeiten markierten Graphen erreicht.

Der Ablauf der Algorithmus sieht aus wie folgt:

1. Bestimmung des Lebensbereich LR zu einer gegeben Kongruenzklasse C. Falls der Lebensbereich in disjunkte Abschnitte zerfällt, wird jeder Abschnitt als eigener Lebensbereich behandelt.

2. Bestimmung der Anzahl benötigter virtueller Register $Regs(LR)$. Es werden maximal so viele virtuelle Register benötigt, wie ein Lebensbereich Iterationen umfaßt. Wenn die Knotenmengen der ersten und der letzten Iteration des Lebensbereichs disjunkt sind, wird ein Register weniger gebraucht.

3. Zu jedem Knoten n wird die Anzahl seiner (lexikalischen) Vorkommnisse im Lebensbereich LR zu $app(n, LR)$ bestimmt. Dementsprechend viele Kopier-Operationen werden benötigt.

4. Wenn die Anzahl der Vorkommnisse $app(n, LR)$ genauso groß ist wie die Anzahl der virtuellen Register, so wird ein zusätzliches Register zum Zwischenspeichern eines Wertes während des „Verschiebens" der Inhalte benötigt. Zur Abschätzung der zusätzlichen Kosten wird gegebenenfalls ein Kostenmaß $Cost_{spill}$ zu $app(n, LR)$ addiert. Es resultiert $app_{safe}(n, LR)$.

5. Die Kosten $app_{safe}(n, LR)$ an jedem Knoten werden gewichtet mit der Wahrscheinlichkeit zur Ausführung dieses Knotens p_n. Diese Knotenmarkierungen bilden *gewichtete Knotenvorkommnisse*.

6. Eine Menge S von Knoten bildet einen Kandidaten für die Plazierung der Kopier-Operationen, wenn *genau* ein Knoten aus S entlang eines jedes Pfades durch den Schleifenkörper vorkommt. Die Menge der Knoten

S_{best}, die die minimale Summe der *gewichteten Knotenvorkommnisse* aller Kandidaten S hat, ist die optimale Menge an Knoten zur Plazierung der Registerkopier-Operationen. S_{best} kann effizient durch einen Algorithmus zur Maximierung von Flüssen in Netzwerken bestimmt werden (siehe [22]), wenn die Knotenmarkierungen $app_{safe}(n, LR)$ als Flußkapazitäten angesetzt werden. Eine Partitionierung der saturierten Knoten nach der Flußmaximierung liefert das gewünschte Ergebnis.

7. Letztlich können die an der RP beteiligten virtuellen Register mit Namen versehen werden.

Erzeugung von Schleifenprolog und -epilog

Das *Stretched Loop*-Verfahren beschreibt den Datenfluß von Array-Elementen einer Schleife im *stabilisierten Zustand.* Dieser wird jedoch erst nach Abarbeitung einiger Iterationen erreicht und nur bleibt bis einige Iterationen vor dem Erreichen der oberen Endes des Iterationsintervalls bestehen. Da die bisherigen Optimierungen diese Analyseergebnisse für den stabilisierten Zustand nutzen, sind auch sie davon abhängig. Damit müssen ein Schleifenprolog und -epilog geschaffen werden, die zusammen sicherstellen, daß die Registerpipeline zu Beginn der Schleife initialisiert und am Ende geleert wird.

Der Prolog umfaßt eine Reihe von Load-Operationen, die Werte in Register laden, auf die im Schleifenkörper zugegriffen wird. Dabei brauchen nur diejenigen Register geladen werden, die bei Überschreiten einer Iterationsgrenze leben, denn die übrigen Register erhalten ihre Werte in der Schleife. Im Epilog müssen Werte aus der Registerpipeline zurück in den Speicher geschrieben werden, denn durch die Entfernung partiell redundanter Store-Operationen befinden sich Werte in der Pipeline, aber nicht im Speicher.

Wenn ein Store über l Iterationen durch die RP verzögert wird, so muß für die letzten l Iterationen ein Epilog *mit* den entsprechenden Store-Operationen erzeugt werden. Sind bei Beendigung der Schleifenbearbeitung alle Werte, die noch in den Speicher geschrieben werden müssen, in Registern vorhanden, so können sie durch eine Folge von Store-Operationen nach Beendigung der Schleife gespeichert werden. Andernfalls kann der Epilog in Form einer separaten Schleife erzeugt werden, der die l_max letzten Iterationen der um diesen Betrag gekürzten, optimierten Schleife ersetzt. Im Epilog werden dann die Berechnungen der letzten Iterationen genauso wie das Zurückschreiben der Ergebnisse erledigt. l_max ist dabei die größte Iterationsdistanz zwischen einer Definition und dem dazugehörendem Store.

Registerallokation

Die Registerallokation ist nicht Teil dieser Arbeit. Die Original-Publikation [6] verweist an entsprechender Stelle auf Registerallokatoren, die sich bei den wie oben beschriebenen verkürzten Lebensdauern von Array-Referenzen mitsamt der Ersetzung durch skalare Variable besonders eignen.

6.2.2 Vor- und Nachteile

Das Verfahren des optimalen Register-Pipelinings versucht die Schwächen der einfacheren Varianten zu überwinden, die in der Nichtbeachtung partiell redundanter Zugriffe und der simplen Plazierung der Kopier-Operationen liegen. Daraus ergeben sich folgende Vor- und Nachteile:

- **Vorteile**
  - Behandlung partiell redundanter Zugriffe
  - Effizienzsteigerung der RP durch optimale Plazierung von Load-, Store- und Kopier-Operationen
  - Sparsamerer Umgang mit Registern
- **Nachteile**
  - Hoher Aufwand zur Implementierung
  - Verwendung von schwer zu ermittelnden Ausführungswahrscheinlichkeiten

Während der *Plazierung von Registerkopier-Operationen* wird auf die Ausführungswahrscheinlichkeit aller Knoten des Schleifenkörpers zurückgegriffen. Diese Wahrscheinlichkeiten sind schwer oder überhaupt nicht exakt zu bestimmen. Es ist notwendig, durch *Profiling* diese Werte für bestimmte Eingaben zu ermitteln oder sich auf allgemeine Schätzungen zu verlassen. Bei grober Fehlschätzung verliert diese Stufe des Algorithmus ihre Optimalität.

Durch die Steigerung der Effizienz der RP ist gegenüber einer *einfachen* Variante ein Geschwindigkeitsgewinn ebenso wie die Verminderung der Anzahl der Speicherzugriffe mit der Folge der Stromersparnis zu erwarten, sofern der Stromverbrauch von externen Speicherzugriffen über dem der Registerkopier-Operationen liegt. Der sparsamere Umgang mit Registern durch die verkürzten Lebensdauern kann den Registerdruck gegenüber der *einfachen* Version verringern, und somit einem Spilling entgegenwirken. Diese Argumente sprechen für den Einsatz des Verfahrens, allerdings ist abzuschätzen, ob der Gewinn gegenüber einfacheren Register-Pipelines den beträchtlichen Aufwand zur Erzielung der Optimalität rechtfertigt. Im Hinblick auf Register-Pipelines mit Hardware-Unterstützung (Kapitel 6.3) scheint der Aufwand an dieser Stelle zu hoch.

Ein weitere Auswirkung in Verbindung mit dem Instruction Scheduling erscheint möglich. Die Registerkopier-Operationen sind verteilt im Schleifenkörper anstatt einer Ansammlung am Ende des Schleifenkörpers. Die voneinander abhängigen Kopieroperationen sind über einen größeren Bereich verstreut und mit anderen Operationen verflochten. Damit kann es die Gelegenheit zu einem besseren Instruction Scheduling geben, welches die Ressourcen des Prozessors gleichmäßiger auslastet und zu einer Beschleunigung der Ausführung beiträgt.

Auf Registerkopieroperationen kann völlig verzichtet werden, wenn die Schleife um den Faktor der Länge der größten Lebensdauer (in Iterationen) eines Array-Elemente abgerollt wird. Wenn der Faktor klein ist, kann Loop Unrolling also eine interessante Alternative sein.

6.3 RP mit Einsatz der AGU und Verwendung von On-Chip-RAM

Alle bislang vorgestellten Verfahren des Register-Pipelinings machen keinen Gebrauch von den besonderen Hardware-Eigenschaften eines DSP, insb. der AGU und des On-Chip-RAM. Daher soll nun eine Implementation einer Register-Pipeline diskutiert werden, die diesen Mangel behebt. Die RP wird als Ringbuffer im On-Chip-RAM realisiert. Zwei Zeiger dienen zur Adressierung von Anfang und Ende. Die Zeiger können effizient durch die AGU unter Ausnutzung von Post-Inkrement-Adressierungsarten verwaltet werden. Zwar werden keine Speicherzugriffsoperationen eliminiert, sondern vom externen Speicher in das On-Chip-RAM verlagert, doch es ergeben sich einige Vorteile gegenüber den anderen RP-Arten. Es entfällt das Kopieren der Elemente in der RP beim Übergang in die nächste Iteration, so daß größere RP, die bislang am zu hohen Aufwand für das Kopieren der Registerinhalte scheiterten, effizient arbeiten können. Wenn bei einer größeren RP die Registerkopier-Operationen in ihren Kosten die Speicherzugriffe übertreffen, ist es sinnvoller, auf die Registerkopier-Operationen zu verzichten und weiterhin Speicherzugriffe durchzuführen. Wenn diese beschleunigt werden, ist ein größerer Gewinn zu erwarten. Der Platz für eine RP steht im On-Chip-RAM bereit, ohne daß dadurch der Registerdruck steigt. Das On-Chip-RAM schnell genug, um gegenüber dem externen Speicher einen Geschwindigkeitsvorteil zu erzielen. Die Verwaltung der RP vereinfacht sich, da das Weitersetzen von Zeigern durch die AGU erledigt werden kann.

Voraussetzungen zur Anwendung dieser Technik sind das Vorhandensein einer AGU mit mind. einem freien Adreßregister und der Möglichkeit zur zirkularen Post-Inkrement-Adressierung.

In Tabelle 6.1 werden die Operationen nebeneinander gestellt, die bei Einfügen und Entnehmen eines Elementes der RP mit N Elementen anfallen, sowie die Operationen am Iterationsübergang. Die RP mit Hardware-Unterstützung ist effizienter als die Standardversion, wenn ihre Mehrkosten durch die Speicherzugriffe auf das On-Chip-RAM geringer sind als die Kosten von N-1 Registerkopieroperationen.

Abbildung 6.1 zeigt schematisch die Verwendung der AGU und des On-Chip-

Aktion	Standard-RP	RP+OCR+AGU
Einfügen in RP	1 Reg.kopieroperation	1 Speicherzugriff OCR
Entnehmen aus RP	1 Reg.kopieroperation	1 Speicherzugriff OCR
Iterationsübergang	N-1 Reg.kopieroperationen	keine

Tabelle 6.1: Operationen von Standard-RP und RP mit On-Chip-RAM + AGU

RAM bei einer Register-Pipeline. Die Werte der $N + 1$ Elemente `a[i-N]` bis `a[i]` werden in On-Chip-RAM gespeichert. Ein Schleifenprolog muß für den Transfer der Startwerte aus dem externen RAM in das On-Chip-RAM sorgen. Wenn die Schleife ihren stabilisierten Zustand erreicht hat, gelangen die Werte durch Store-Operationen in die Queue bzw. werden ihr durch Load-Operationen entnommen. Ein Register im Adreßregistersatz beinhaltet die Adresse *start*, ein weiteres die Adresse *end*. *start* zeigt auf den „ältesten“ Eintrag, der als nächstes ausgelesen werden kann. *end* zeigt auf den ersten freien Platz, in den das nächste Element, welches in die RP geschoben wird, eingefügt wird. Bei der Entnahme eines Wertes aus der RP kann dieser durch eine zirkulare Post-Inkrement-Operation ausgelesen werden. Diese bewirkt neben der Rückgabe von `a[i-N]` eine Addition von eins[1] auf den aktuellen Wert des *start*-Registers. Damit zeigt es auf den nächsten Wert in der Register-Pipeline. Da es sich bei der RP um einen Ringpuffer handelt, wird ein Modulo bzgl. der Anzahl der Elemente in der RP gebildet. Nach Überschreiten einer oberen Adresse kann im unteren Speicherbereich weitergearbeitet werden. Auf analoge Weise wird das AGU-Register für das andere Ende der RP verwaltet. Es braucht nicht darauf geachtet zu werden, ob die Anfangszeiger den Endzeiger einholt. Pro Iteration wird ein Wert in die RP geschrieben und ein Wert ausgelesen, die Distanz der beiden Zeiger untereinander bleibt konstant.

Die Hardware-Unterstützung läßt sich sowohl bei einer einfachen RP einsetzen als auch bei der optimalen Variante. Es entfallen jeweils alle Operationen zum Kopieren von Registerinhalten, da bei dieser Implementation keine Inhalte bewegt werden. Dafür müssen das Einfügen und Auslesen mit spezielle, zirkular adressierende Post-Inkrement-Operationen verwendet werden. Die Erzeugung des Schleifenprologs muß das Laden der Initialinhalte in das On-Chip-RAM bewirken, aber auch die Adreß- und Indexregister der AGU müssen Startwerten geladen werden.

Eine verbesserte Alternative zur Implementierung entsteht daraus, daß der Abstand zwischen dem Anfang und dem Ende der Queue schon während der Compilierung bekannt ist. Damit ist es nicht unbedingt notwendig, zwei Adreßregister der AGU zu belegen. Es kann mit einem Adreßregister und unterschiedlichen Modify-Werten auf Anfang bzw. Ende zugegriffen werden. Die Modifier können entweder in zwei Modify-Registern gehalten oder als Direktoperanden in den Code eingestreut werden.

[1] Je nach Konzeption der AGU muß für ein 32-Bit-Wort um eins oder vier inkrementiert werden.

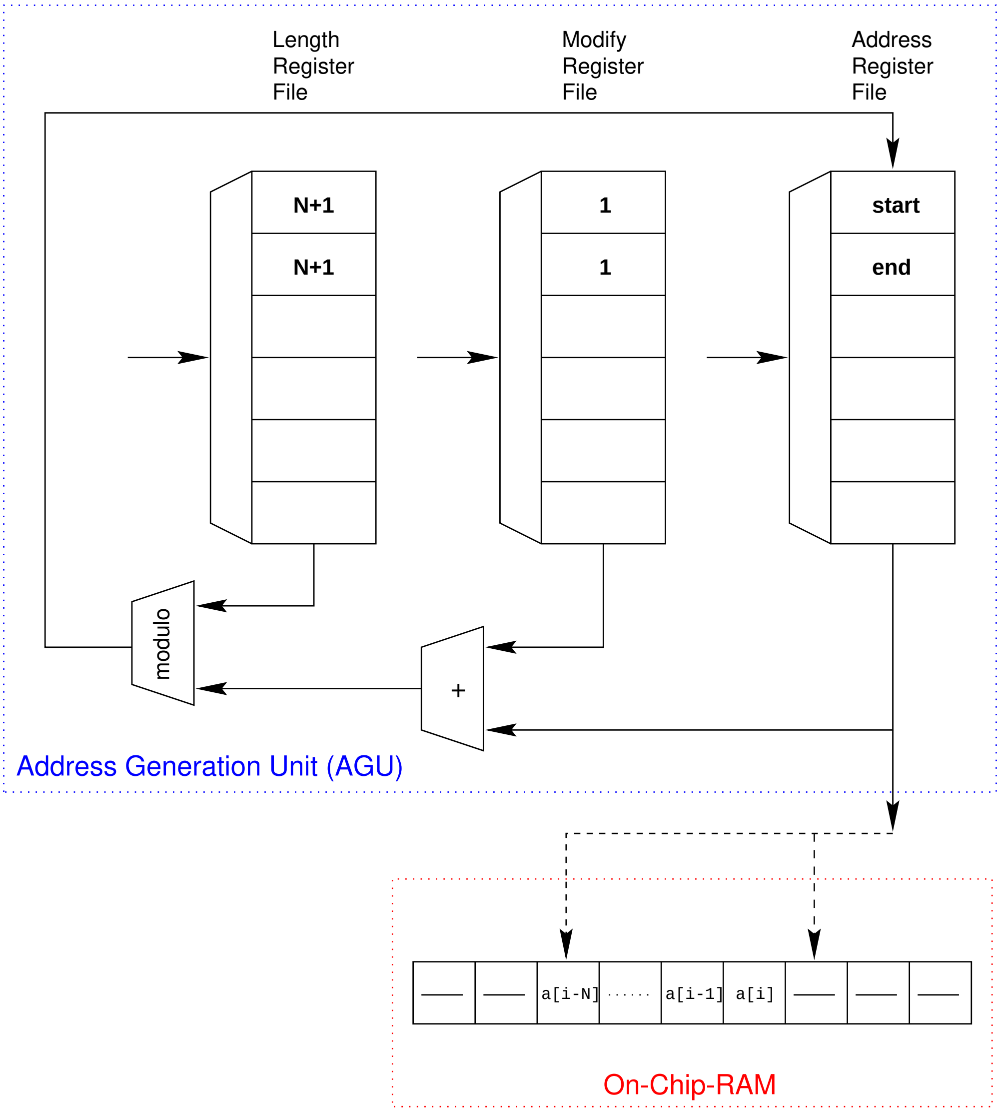


Abbildung 6.1: Register-Pipeline mit Hardware-Unterstützung

Es ist zwar nicht zwingend, die hier beschriebene RP im On-Chip-RAM unterzubringen – sie kann auch im externen Speicher liegen – doch sollte zur Erzielung einer beschleunigten Verarbeitung auf das On-Chip-RAM zurückgegriffen werden. Verschiedene Verfahren [16],[18] sind zur effiziente Plazierung von Daten im On-Chip-RAM entwickelt worden, die auch hier Einsatz finden können.

6.3.1 Vor- und Nachteile

Die Verwendung der AGU und des On-Chip-RAM bringt eine Reihe von Vorteilen mit sich, die neben einigen Nachteilen unten aufgeführt sind:

- **Vorteile**
  - Steigerung der Effizienz der RP bei größeren Tiefen
  - RP für DSP mit heterogenem Registersatz besser einsetzbar, wenn eine AGU und On-Chip-RAM statt vieler freier Register verfügbar ist
  - Keine Steigerung des Registerdrucks der allgemeinen Register
- **Nachteile**
  - Abhängigkeit von Vorhandensein von AGU mit mind. einem freien Adreßregister

Der Verwendungsbereich des RP vergrößert sich, denn die Kosten pro Iteration zur Verwaltung der RP verringern sich. Damit ist die Technik auch für größere Iterationsdistanzen verwendbar. Statt ungenutzt oder willkürlich genutzt zu werden, können die AGU und das On-Chip-RAM nutzbringend und systematisch zur Beschleunigung von Schleifen eingesetzt werden. Die gleichmäßige Nutzung mehrerer Ressourcen steigert die Effizienz der Ausführung. Der Registerdruck unter den allgemeinen Registern steigt nicht, die Register stehen weiterhin zu allgemeinen Zwecken zur Verfügung. RP wird in der beschriebenen Form auch für Prozessoren mit heterogenen Registersätzen nutzbar gemacht, sofern sie über eine AGU mit mind. einem freien Adreßregister verfügen. Insbesondere bei Verwendung mehrdimensionaler Arrays und Einsatz der RP bei Abhängigkeiten bzgl. einer äußeren Induktionsvariablen bietet das On-Chip-RAM genügend Platz zur Unterbringung der meist kleinen temporären Arrays, dabei ist die Zugriffsgeschwindigkeit deutlich höher als bei externem Speicher.

Auf die Elemente der RP wird (bei der Verwendung von zwei Adreßregistern) mit der Schrittweite eins zugegriffen. Es werden sequentiell aufeinanderfolgende Elemente ausgelesen bzw. in die Queue hineingeschrieben. Auch für Elemente (z.B. `a[3*i+2]`), die vorher nicht an aufeinanderfolgenden Speicheradressen abgelegt waren, kann mit der normierten Schrittweite eins in der RP zugegriffen werden, sofern sie im Zuge der Optimierung als redundant erkannt und durch eine RP ersetzt wurden. Bei einem eingeschränkten Bereich der Modify-Werte kann die AGU u.U. nicht für die Adressierung der Array-Elemente in der nichtoptimierten Schleifenversion genutzt werden. Durch die Normierung der Modify-Werte in der Queue auf eins kann eine solche AGU zum Register-Pipelining genutzt werden, was einen recht hohen Effizienzgewinn verspricht.

Der Nachteil in Form eines Mehrverbrauchs an AGU-Adreßregistern für die Zeiger auf den Anfang und das Ende des Ringpuffers wiegt gegenüber den Vorteilen nicht allzu schwer. Zwar sind i.a. nicht sehr viele Adreßregister in der AGU vorhanden, aber angesichts der möglichen Optimierung, die in der Regel auch nicht übermäßig häufig in Schleifen anzuwenden ist, liegen die zu erwartenden Geschwindigkeitsvorteile über möglichen Nachteilen. Wird der Druck unter den Adreßregistern zu hoch, so kann es zum unerwünschten Spilling der Adreßregister kommen. Daß das meist recht kleine On-Chip-RAM durch die RP belegt wird, kann nicht als echter Nachteil gelten. Schließlich ist es u.a. dazu da, häufig benutzte Daten, die nicht mehr in die Register passen, bei hoher Zugriffsbandbreite zwischenzuspeichern.

6.4 Beurteilung der verschiedenen Optimierungen

Das verbesserte Register-Pipelining schafft die Möglichkeit, die Elemente einer Register-Pipeline mit in die Registerallokation einzubeziehen. Der Aufwand dafür ist nicht allzu hoch, denn es können Informationen, die zur Erstellung der RP schon bereitgestellt wurden, für diesen Zweck erneut verwendet werden. Hinzu kommt ein modifiziertes Graphenfärbungsverfahren für den Register-Interferenzgraphen. Für das verbesserte Register-Pipelining gelten die gleichen Anwendungsbereiche wie für die Standardversion. Der Gewinn liegt im systematischen Vorgehen bei der Registerallokation, während die Basisversion von RP temporäre Variable schafft und deren Registerzuweisung einer späteren Compiler-Phase überläßt. Vorteile gegenüber dem einfachen Verfahren entstehen durch die Verbesserung genau dann, wenn es gelingt zu verhindern, daß es bei den Registern, die Elemente der RP beinhalten, zum Spilling kommt. Für Zielarchitekturen mit homogenen Registersätzen ist das verbesserte RP sicherlich nützlich anzuwenden, wenn die Tiefe der RP – so wie bei der Grundversion – nicht zu groß ist. Bei heterogenen Registersätzen bleibt die RP-Optimierung weiterhin wenig geeignet.

Die optimale Variante des Register-Pipelinings erweitert die Grundversion um mehr als nur die Registerallokation. Sie schafft es durch aufwendige Analyse- und Optimierungsschritte, eine minimale Anzahl an Load-, Store- und Registerkopier-Operationen zu erzeugen. Es werden gegenüber der Standardversion auch partiell redundante Referenzen behandelt. Damit ist der Einsatzbereich und der mögliche Gewinn der Optimierung stark erweitert. Es bleibt jedoch kritisch abzuwägen, ob der stark erhöhte Implementierungsaufwand gegenüber den verbesserten Optimierungserfolgen gerechtfertigt ist. In den meist recht einfachen Schleifenkörpern von DSP-Anwendungen sind Vorkommnisse partiell redundanter Array-Referenzen zwar möglich, doch nicht allzu häufig. Es bleibt zu prüfen, wie groß der Vorteil der Optimalität gegenüber nicht-optimalen Optimierungsergebnissen ist.

Interessanter erscheint die Verbindung von Register-Pipelining mit Hardware-Ressourcen von DSP. Unter Ausnutzung von AGU und On-Chip-RAM kann die Optimierung zur Behandlung von redundanten Array-Zugriffen über größere Iterationsdistanzen hinweg genutzt werden. Entgegen der verbesserten Version und dem optimalen RP kann die hardware-unterstützte RP-Variante auch bei Zielarchitekturen mit heterogenen Registersätzen erfolgversprechend eingesetzt werden.

6.5 Weitere Literatur

- Kolson, D.J., Nicolau, A., Dutt, N., Kennedy, K.
  Optimal Register Assignment to Loops for Embedded Code Generation
  ACM Transactions on Design Automation of Electronic Systems, Vol. 1., No. 2, April 1996.

- Miranda, M., Catthoor, F., Janssen, M., De Man, H.
  High-Level Address Optimisation and Synthesis Techniques for Data-Transfer Intensive Applications
  IEEE Transactions on VLSI Systems, 6(4), pp. 677-686, 1998.

- Panda, P.R., Dutt, N., Nicolau, A.
  Local Memory Exploration and Optimization in Embedded Systems
  IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, p.3, Vol. 18, No. 1, Jan. 1999.

- Sudarsanam, A., Malik, S., Tijang, S., Liao, S.
  Optimization of Embedded DSP Programs Using Post-Pass Data-Flow Analysis
  Proceedings of 1997 International Conference on Acoustics, Speech and Signal Processing, 1997.

- Sudarsanam, A., Liao, S., Devadas, S.
  Analysis and Evaluation of Address Arithmetic Capabilities in Custom DSP Architectures
  Proceedings of 1997 ACM/IEEE Design Automation Conference, 1997.

Kapitel 7

Spezielle Optimierungen

In diesem Kapitel sollen spezielle Optimierungen vorgestellt werden, die nicht in die Kategorien der vorherigen Kapitel passen. Sie sind entweder keine Redundanzeliminationen und erhöhen nicht die Effizienz von Speicherzugriffen, oder sie benötigen zu ihrer Durchführung keine Array-Datenflußanalysen. Der Grund, warum sie dennoch in dieser Arbeit vorgestellt werden, liegt darin, daß sie für die Optimierung von DSP-Applikationen von großer Bedeutung sein können und Beziehungen mit Teilaspekten der vorliegenden Arbeit aufweisen.

Als erstes Verfahren wird das *kontrollierte Loop Unrolling* diskutiert. Loop Unrolling ist eine weitverbreitete Optimierung, die durch Informationen aus Array-Datenflußanalysen profitieren kann. Es wird gezeigt, wie beim Loop Unrolling auf effiziente Weise ein günstiger trade-off zwischen Speicherplatzverbrauch und Steigerung der Parallelität auf Instruktionsebene erzielt werden kann.

Danach folgt ein Abschnitt über die Unterstützung von Software-Pipelining durch Array-Datenflußanalysen. Ohne das bekannte Optimierungsverfahren verändern zu müssen, werden die Möglichkeiten aufgezeigt, die durch Einbeziehung von Informationen aus Array-Datenflußanalysen entstehen.

Als letztes Verfahren wird die Behandlung von *Aggregate Array Computations (AACs)* vorgestellt. AACs sind eine besondere Form von Schleifen / Schleifenschachtelungen, die über Folgen von Array-Elementen Werte akkumulieren. AACs bieten große Potentiale zur Redundanzelimination, die über die bisherigen Ergebnisse weit hinaus gehen. Bislang wurde bei allen Optimierungen eine Verbesserung in der Größenordnung eines konstanten Faktors erzielt, während bei der Optimierung von AACs asymptotische Verbesserungen möglich sind. Da AACs in äußerst vielen DSP-Applikationen auftreten, hat ein Verfahren zur Verminderung ihrer Komplexität durch Verringerung der Speicherzugriffe in dieser Diplomarbeit seine Berechtigung.

7.1 Kontrolliertes Loop Unrolling

Loop Unrolling ist eine häufig verwendete Optimierung, die dazu dient, die Parallelität auf Instruktionsebene in einem Schleifenkörper zu vergrößern. Dies geschieht durch mehrfaches Aneinanderreihen des Schleifenkörpers und entsprechendem Vergrößern der Schrittweite. Nachteilig beim Loop Unrolling ist das Anwachsen des Code-Umfangs. Es wird eine Methode aus [8] vorgestellt, die mittels der Ergebnisse aus Array-Datenflußanalysen hilft, die Vorteile des Loop Unrolling auszunutzen, ohne daß dabei die Nachteile überwiegen. Das Ansteigen der Parallelität wird abgeschätzt und die Häufigkeit des Schleifenabrollens durch den erwarteten Parallelitätszuwachs und einen Qualitätsparameter kontrolliert.

Beispiel 7.1.1 *Loop Unrolling*

Original-Schleife: ***1-fach abgerollt:*** ***3-fach abgerollt:***

```
for(i=0; i<N; i++)  for(i=0; i<N; i+=2) for(i=0; i<N; i+=4)
{                   {                   {
   a[i] = b[i];        a[i]   = b[i];      a[i]   = b[i];
}                      a[i+1] = b[i+1];    a[i+1] = b[i+1];
                    }                      a[i+2] = b[i+2];
                                           a[i+3] = b[i+3];
                                        }
```

Beispiel 7.1.1 zeigt für eine Schleife mit einer einzelnen Instruktion im Schleifenkörper die durch 1-faches und 3-faches Loop Unrolling entstehenden Schleifen[1]. Mit steigendem *Grad* des Schleifenabrollens ergeben sich mehr voneinander unabhängige Operationen im Schleifenkörper, die bei entsprechenden Hardware-Ressourcen parallel ausgeführt werden können. Gleichzeitig steigt auch der Code-Umfang, der insbesondere bei eingebetteten Systemen mit deren häufig eng begrenzten Speichermöglichkeiten Schwierigkeiten bereiten kann.

Datenabhängigkeiten verhindern die parallele Ausführung von Instruktionen, so daß Loop Unrolling bei Vorliegen von loop carried dependences eine Sättigung bzgl. der möglichen Parallelität des Schleifenkörpers erfährt.

[1] Für das gewählte Beispiel wurde die Annahme getroffen, daß N durch zwei und vier teilbar ist. Wäre dies nicht der Fall, so müßte ein Schleifenepilog für die letzten Iterationen erstellt werden, deren Anzahl sich aus dem Divisionsrest aus N und zwei bzw. vier ergibt.

Beispiel 7.1.2 *"Sättigung" beim Loop Unrolling*

Original-Schleife:	***1-fach abgerollt:***	***3-fach abgerollt:***

```
for(i=0; i<N; i++)   for(i=0; i<N; i+=2)  for(i=0; i<N; i+=4)
{                    {                    {
   a[i] = a[i-2];       a[i]   = a[i-2];     a[i]   = a[i-2];
}                       a[i+1] = a[i-1];     a[i+1] = a[i-1];
                     }                       a[i+2] = a[i];
                                             a[i+3] = a[i+1];
                                          }
```

Das Beispiel 7.1.2 verdeutlicht den „Sättigungseffekt“ beim Loop Unrolling. Zwischen der Definition `a[i]` und dem Gebrauch `a[i-2]` der ursprünglichen Schleife gibt es eine loop carried dependence. Durch 1-faches Abrollen kann die Parallelität im Schleifenkörper gegenüber dem Original vergrößert werden, jedoch schon bei 3-fachen Abrollen steigt die Parallelität nicht mehr an. Die loop carried dependences sind zu loop independent dependences geworden, die die Parallelausführung aller Instruktionen verhindern. Die dritte Operation ist von der ersten abhängig, und die vierte von der zweiten. Weiteres Abrollen erbringt keine weitere Parallelitätssteigerung, sondern vergrößert nur unnötig den Schleifenkörper.

Ein Weg sicherzustellen, daß beim Loop Unrolling nicht über den Sättigungspunkt hinaus abgerollt wird, besteht darin, iterativ vorzugehen, dabei den Parallelitätszuwachs abzuschätzen und nur dann die Schleife abzurollen, wenn der Zuwachs ein bestimmtes unteres Maß überschreitet. Array-Datenflußanalysen ermöglichen die benötigte Abschätzung durch die von ihnen gelieferte Information.

Nicht die Parallelität des Schleifenkörpers selbst kann ermittelt werden, sondern es dient die Länge des *kritischen Pfades* als Maß der Parallelität. Der kritische Pfad ist der längste Pfad im Datenabhängigkeitsgraphen des Schleifenkörpers, und entspricht damit der längsten Folge von Datenabhängigkeiten, die parallele Ausführung verhindern. Die Länge des kritischen Pfades wird mit l bezeichnet. Wenn für den gegebenen Schleifenkörper die Länge des kritischen Pfades l ist, so muß kann die Länge l_{unroll} des kritischen Pfades der 1-fach abgerollten Schleife $2 \times l$ nicht überschreiten. Wenn keine loop carried dependences vorliegen, verlängert sich der kritische Pfad durch das Abrollen nicht. Ansonsten kann er sich maximal verdoppeln, wenn eine Abhängigkeit zwischen der ersten und der letzten Instruktion des abgerollten Schleifenkörpers besteht. Es muß gelten $l \leq l_{unroll} \leq 2 \times l$. Ein äußerer Parameter τ mit $l \leq \tau \leq 2 \times l$ dient als Schwellenwert, dessen Überschreiten das Schleifenabrollen beendet. Wird der kritische Pfad l_unroll durch das Abrollen länger als τ, so liegt der Parallelitätszuwachs unterhalb einer gewünschten Grenze und das Verfahren hat den Sättigungspunkt bis auf eine bestimmte Umgebung erreicht.

Zur Bestimmung der Länge des kritischen Pfades l_{unroll} müssen Array-Datenabhängigkeiten mit der Iterationsdistanz 1 bestimmt werden. Die Iterationsdistanz 1 reicht aus, da das Schleifenabrollen iterativ jeweils um einen Schritt erfolgt. Von den möglichen Datenabhängigkeiten (vgl. Kapitel 3.2) müssen Datenflußabhängigkeiten, Anti-Abhängigkeiten und Ausgabe-Abhängigkeiten berücksichtigt werden, denn sie verhindern die Parallelverarbeitung der abhängigen Instruktionen. Eingabe-Abhängigkeiten sind an dieser Stelle „unschädlich". Wenn für alle Knoten des Schleifenkörpers festgestellt wird, welche Referenzen sie mit einer Iterationsdistanz ≤ 1 erreichen (*δ-reaching-references*), können die gesuchten Abhängigkeiten festgestellt werden. Anschließend läßt durch Zurückverfolgen der Abhängigkeiten die längste Abhängigkeitskette und daraus die Länge des kritischen Pfades bestimmen.

Die Original-Publikation [8] verwendet die δ-Datenflußanalyse zur Bereitstellung der gesuchten Information. Ein entsprechende Parametrisierung für das spezielle Datenflußproblem *δ-reaching-references* kann dort nachgelesen werden. Auch die Stretched Loop-Analyse und das DSA-Verfahren liefern bei geeigneter Parametrisierung die benötigten Informationen. Das Lazy-Verfahren kann nur die Datenflußabhängigkeiten ermitteln, nicht aber die Ausgabe- und Anti-Abhängigkeiten und ist somit nicht geeignet.

7.1.1 Vor- und Nachteile

Der Vorteil des kontrollierten Loop Unrollings gegenüber dem nicht-kontrollierten Loop Unrolling liegt in der automatischen Bestimmung des Abrollfaktors, der zum günstigsten Verhältnis zwischen erzielter Parallelität und Schleifenlänge führt[2]. Ohne die Kontrolle muß entweder durch mehrfach wiederholte, manuelle Versuche die bestmögliche Optimierung bestimmt werden, oder es werden feste Faktoren verwendet ohne deren Qualität zu überprüfen. Mit der Möglichkeit zur Parametrisierung kann Einfluß auf die Code-Qualität bzgl. der Parallelität auf Instruktionsebene genommen werden.

Nachteilig am kontrollierten Loop Unrolling ist der zusätzliche Aufwand zur Datenflußanalyse. Da die Kontrollmöglichkeit aber bereits durch eine einfache Array-Datenflußanalyse wie dem δ-Verfahren ermöglicht wird, ist dieser zusätzliche Aufwand nicht allzu groß und kann häufig toleriert werden.

Da Loop Unrolling eine Optimierungstechnik ist, die bei zeitkritischen Schleifen häufig angewendet wird. Gerade dort lohnt sich hoher Aufwand zur Steigerung der Performance, so daß der Zusatzaufwand zur Datenflußanalyse nicht so sehr ins Gewicht fällt. Aber auch zeitkritischen Schleifen dürfen durch zu große Abrollfaktoren nicht zu viel Code-Umfang haben. Insbesondere wenn ein Instruktionscache vorhanden sein sollte, kann durch dadurch die Performance wiederum negativ beeinflußt werden. Insgesamt erscheint das kontrollierte Loop Unrolling sehr empfehlenswert, weil es die Möglichkeit bietet, eine Mehrzieloptimierung (Geschwindigkeit vs. Speicherplatz) effizient und durch einen Qualitätsparameter gesteuert durchzuführen.

[2]Die Original-Publikation [8] erwähnt eine weitere Möglichkeit, den mit dem Abrollfaktor steigenden Registerdruck auf ähnliche Weise abzuschätzen und zu kontrollieren

7.2 Unterstützung von Software-Pipelining

In diesem Abschnitt soll gezeigt werden, wie Array-Datenflußanalysen genutzt werden können, um Software-Pipelining effizienter zu gestalten. Das vielfach schon verwendete Optimierungsverfahren selbst wird nicht verändert, es soll lediglich gezeigt werden, wie es von der bereitgestellten Information profitiert.

Beim Software-Pipelining wird aus voneinander unabhängigen Teilen verschiedener aufeinanderfolgender Iterationen einer Schleife ein neuer Schleifenkörper geformt. Ziel dieser Transformation ist die Vergrößerung der Parallelität des Schleifenkörpers, so daß die Ressourcen von ILP-Architekturen gleichmäßiger genutzt werden. Loop carried Abhängigkeiten von Instruktionen begrenzen die Möglichkeiten des Software-Pipelining-Verfahrens. Daher ist es für die Erstellung guter Schedules wichtig, präzise Informationen über Datenflußabhängigkeiten zu haben. Von besonderer Bedeutung sind zyklische Abhängigkeitsketten, denn sie behindern die Optimierung in starkem Maße.

```
     for(i = 0; i < N; i++)
     {
A       x = a[i];

B       y = b[i];

C       z = x + y;

D       k = k + 1;
     }
```

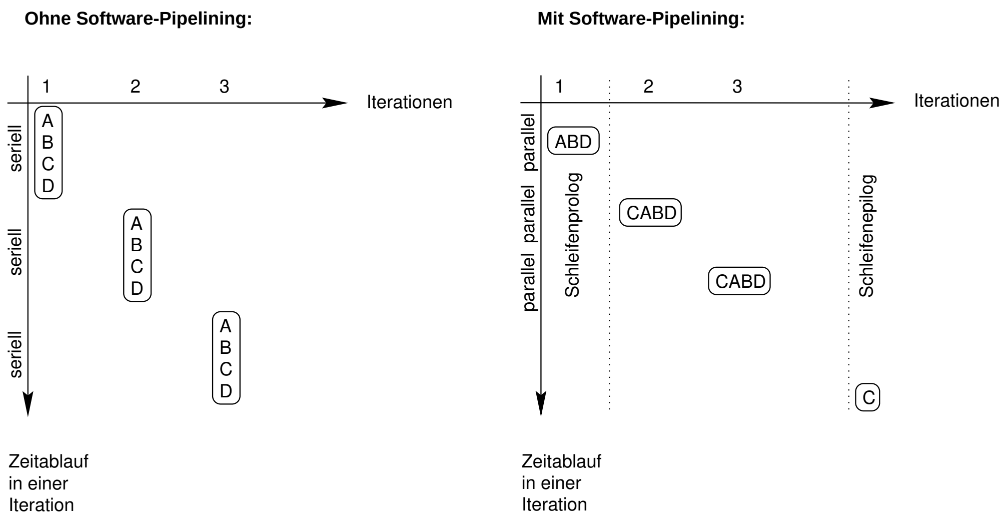


Abbildung 7.1: Anwendung des Software-Pipelinings

Abbildung 7.1 zeigt eine Schleife mit ihren Datenabhängigkeiten. Ohne Anwendung des Software-Pipelinings werden die Instruktionen $A \ldots D$ sequentiell in jeder Iteration ausgeführt bis das Schleifenende erreicht ist. Nach Anwendung von Software-Pipelining werden pro Iteration die Instruktionen $CABD$ parallel ausgeführt. Dabei stammt C aus der vorherigen Iteration. Es muß sichergestellt sein, daß C seine Operanden liest, bevor diese durch A oder B überschrieben werden. Zur Initialisierung ist ein Prolog aus ABD notwendig, während am Ende des Iterationsbereichs C allein im Schleifenepilog ausgeführt werden muß.

Treten in einer Schleife Array-Zugriffe auf, so müssen deren Abhängigkeiten beim Software-Pipelining berücksichtigt werden. Ein Weg dazu besteht in der Verwendung speicherbasierter Abhängigkeiten als Näherungslösung der exakten Abhängigkeiten. Das führt möglicherweise dazu, daß eine Reihe von Instruktionen fälschlich als abhängig klassifiziert werden und somit der Schedule unnötig verschlechtert wird. Mit den vorgestellten Array-Datenflußanalysen (Kapitel 4) wird eine höhere Präzision erreicht, die unmittelbar zu besseren Schedules führt.

```
for(i = 0; i < N; i++)
{
   a[i-1] = f(i);
      1
   b[i-1] = a[i-2];
      1
   c[i]   = 2 * b[i];
      2
   a[i]   = c[i-2];
}                     2
```

Abbildung 7.2: Schleife mit speicher- und wertebasierten Abhängigkeiten

Abbildung 7.2 zeigt eine Schleife mit Zugriffen auf zwei Arrays. Zwischen den einzelnen bestehen Datenabhängigkeiten, die sich über mehrere Iterationen erstrecken. Die wertebasierten Abhängigkeiten sind grün eingezeichnet, die speicherbasierten Datenabhängigkeiten umfassen zusätzlich die rot gefärbte Abhängigkeit. Während die speicherbasierten Abhängigkeiten einen Zyklus bilden, ist dieser bei wertebasierten Abhängigkeiten aufgelöst. Damit kann die Schleife dem Software-Pipelining unterzogen werden.

Die Ausführungen in [6] gehen von Stretched Loop-Analysen aus, um die wertebasierten Array-Datenabhängigkeiten zu bestimmen. Jedoch sind auch das δ-Verfahren, die Lazy-Datenflußanalyse und das DSA-Verfahren geeignet, die benötigte Array-Datenflußinformation bereitzustellen. Die Optimierung zum Software-Pipelining braucht nicht verändert zu werden. Es reicht aus, wenn statt der ungenauen Information einer speicherbasierten Abhängigkeitsanalyse oder einer anderen Approximation die genaueren Resultate verwendet werden.

Einige der unerwünschten Zyklen im Datenflußgraphen können auch durch vorherige Anwendung von RLE, RSE oder RP aufgebrochen werden. Diese Effekte wurden schon bei den betreffenden Optimierungen in Kapitel 5 diskutiert. Selbst wenn die genannten Optimierungen nicht durchgeführt werden, so kann doch mit den gleichen Analysen, die sie benötigen, das Software-Pipelining unterstützt werden. Es stellt eine Alternative dar für Fälle, in denen die Redundanzelimination mit zu vielen oder zu großen Nachteilen behaftet ist.

7.2.1 Vor- und Nachteile

Die Unterstützung des Software-Pipelinings durch Array-Datenflußanalysen erbringt den Vorteil, eine Effizienzsteigerung einer häufig verwendeten Optimierung allein durch bessere Analysen zu bewirken. Das Optimierungsverfahren selbst bleibt unangetastet. Die Analysen sind diegleichen wie für viele Redundanzeliminationen, so daß die Auswahl zwischen deren Anwendung oder dem Software-Pipelining besteht.

Der Nachteil eines gesteigerten Aufwands durch die Array-Datenflußanalysen kann nicht gewertet werden. Sie werden schon bei vorheriger Anwendung von RLE, RSE oder RP erforderlich.

Wenn es die Auswahl zwischen der Anwendung von Verfahren zur Redundanzeliminationen und Software-Pipelining gibt, sollte für die Fälle, die in Kapitel 5 als geeignet für Redundanzelimination charakterisiert wurden, diese auch angewendet werden. Anschließend kann Software-Pipelining angewendet werden. In denjenigen Fällen, in denen Redundanzeliminationen nicht anwendbar oder wenig geeignet sind, kann das Software-Pipelining alternativ eingesetzt werden.

7.3 Aggregate Array Computations

Es gibt eine spezielle Form von Programmschleifen, deren Gemeinsamkeit es ist, über eine Folge von Array-Elementen akkumulierte Werte zu berechnen. Diese Schleifen heißen *Aggregate Array Computations (AAC)*. Bei „naiver" Programmierung der AACs kann es zu erheblichen Redundanzen bei Speicherzugriffen kommen, die dadurch entstehen, daß bei der Berechnung eines akkumulierten Wertes viele Array-Elemente erneut referenziert werden, die auch schon in die Berechnungen vorheriger Werte einbezogen waren. Ziel des in [10] vorgestellten Verfahrens ist, solche mit starker Redundanz behafteten AACs zu entdecken und durch *Inkrementalisierung*, d.h. durch Speicherung und Gebrauch von Zwischenresultaten, zu größerer Effizienz zu verhelfen.

Beispiel 7.3.1 *AAC zur Berechnung von Partialsummen*

Vorher:

```
for(i = 0; i < n; i++)
{
   s[i] = 0;
   for(j = 0; j < i; j++)
      s[i] += a[j]
}
```

Nachher:

```
s[0] = a[0];
for(i = 1; i < n; i++)
   s[i] = s[i-1] + a[i];
```

Das Beispiel 7.3.1 demonstriert die Idee an der Berechnung partieller Summen. Für jedes i soll die Summe aller Elemente `a[0]` ... `a[i]` berechnet und in `s[i]` gespeichert werden. Bei diesem kleinen Beispiel ist es noch leicht zu sehen, daß in jeder inneren Schleife alle Elemente der vorherigen äußeren Iteration erneut referenziert werden. Die Inkrementalisierung besteht in der Verwendung des Zwischenergebnisses, hier des Ergebnisses der vorherigen äußeren Iteration, in der Berechnung des neuen Wertes. Damit können sehr viele redundante Zugriffe erspart werden. In diesem Beispiel wird sogar die Zeitkomplexität von $O(n^2)$ auf $O(n)$ reduziert. Während Compiler-Optimierungen häufig die asymptotische Komplexität der bearbeiteten Programme nicht verändern, sondern eine Verbesserung um konstante Faktoren erzielen, liegt hier ein Verfahren vor, welches einen (speziellen) Algorithmus in einen anderen (speziellen) Algorithmus transformiert, der schon grundsätzlich in Bezug auf die Laufzeit bessere Eigenschaften hat.

Voraussetzung für die Anwendung des Verfahrens auf AACs der Form

$$\text{for } i = e_1 \text{ to } e_2 \text{ do } v = f(v, g(a[h(i)], \ldots)) \tag{7.1}$$

ist, daß die *Akkumulationsfunktion* f eine Umkehrfunktion f^{-1} im folgenden Sinne besitzt:

$$f^{-1}(f(v,c),c) = v \tag{7.2}$$

Nicht unbedingt notwendig, aber durchaus wünschenswert sind Assoziativität und Kommutativität der Akkumulationsfunktion f. An die Indexfunktion $h(i)$ werden keine weiteren Forderungen gestellt, und auch die Wahl der *beitragenden Funktion* g, die verschiedene *beitragende Array-Referenzen* $a[h(i)]$ verknüpft, wird nicht eingeschränkt. v ist die *akkumulierende Variable*. Nicht zwingend, aber im weiteren Verlauf zur Vereinfachung der Darstellung angenommen wird eine Schleifenschrittweite von 1.

Zur automatisierten Behandlung von AACs sind folgende vier Teilprobleme zu behandeln:

1. Erkennung von AACs,

2. Transformation von AACs in inkrementalisierte Darstellungen, und

3. Erzeugung neuer Programmschleifen.

7.3.1 Erkennung von AACs

Bevor AACs optimiert werden können, müssen sie erkannt und ihre wesentlichen Eigenschaften analysiert werden. Infrage kommen loop nests, deren innerste Schleife Array-Elemente akkumuliert und deren äußere Schleifen die Indizes der Array-Referenzen stellen. Eine wichtige Eigenschaft einer AAC ist deren

Operation zum Update der in den beitragenden Array-Referenzen verwendeten Indexvariablen (*Subscript Update Operation (SUO)*).

Die Suche nach AACs in einem loop nest wird hierarchisch von innen nach außen betrieben, dazu kann 7.1 entsprechend verallgemeinert werden. Wird ein passender Kandidat A gefunden, so werden zunächst die Indizes der beitragenden Funktionen in der *beitragenden Menge* $S(A)$ zusammengefaßt. $S(A)$ ergibt sich zu

$$S(A) = \{h(i) | e_1 \leq i \leq e_2\} \tag{7.3}$$

Als nächstes wird dann versucht, die SUO zu bestimmen. Dafür sind erst eine Definition eines *Parameters* einer AAC notwendig, denn SUO manipulieren besondere Parameter einer AAC.

Definition 7.3.1 *Ein* Parameter *einer AAC A ist eine Variable, die außerhalb von A definiert, aber innerhalb von A gebraucht wird. Eine Redefinition eines Parameters ist genau dann eine* Subscript Update Operation (SUO) *für A, wenn der veränderte Parameter in A ausschließlich in Indexfunktionen der beitragenden Array-Referenzen verwendet wird. Für einen Parameter w wird die SUO mit $\oplus_w$ bezeichnet.*

Für das Beispiel 7.3.1 kann also festgestellt werden, daß

- die Akkumulationsfunktion $f(x, y) = x + y$ umkehrbar ist,
- die beitragende Funktion $g(x) = x$ ist,
- die Indexfunktion $h(j) = j$ ist,
- die akkumulierende Variable $s[i]$ ist,
- die beitragende Menge $S(A_i) = \{j | 0 \leq j < i\}$ ist,
- i ein Parameter ist, und
- $\oplus_i$ eine SUO ist.

Die Erkennung der AAC A zusammen mit der SUO $\oplus_w$ führen zur Probleminstanz $A^{\oplus_w}$ der Form:

$$\text{for } i = e_1^{\oplus_w} \text{ to } e_1^{\oplus_w} \text{ do } v^{\oplus_w} = f(v, g(a[h(i)], \ldots))^{\oplus_w} \tag{7.4}$$

Die Indizierung mit $\oplus_w$ kennzeichnet die Substitution in den Ausdrücken enthaltener w durch $w + 1$.

7.3.2 Transformation von AACs in inkrementalisierte Darstellungen

Bislang ist nichts weiter geschehen, als AACs zu erkennen und deren SUO zu bestimmen. Der nächste Schritt besteht darin, die bisherige AAC in eine inkrementalisierte AAC zu überführen. Dazu werden Datenzugriffe soweit wie möglich durch Zwischen- und Endergebnisse vorheriger Iterationen der Schleife ersetzt. Möglich wird das durch eine vorherige Bestimmung der Bereiche der beitragenden Arrays, die den Auswirkungen der SUO unterworfen sind. Anschließend kann eine inkrementalisierte AAC formuliert werden, die vermehrt auf bereits vorhandene Ergebnisse zugreift.

Zunächst werden Unterschiede der beitragenden Mengen zwischen A und $A^{\oplus w}$ berechnet, die Aufschluß über die Auswirkungen der SUO auf die Datenzugriffe geben. Dann kann abhängig von der Akkumulationsfunktion eine neue AAC konstruiert werden, die durch Nutzung vorhergehender Ergebnisse einen neuen Wert effizient berechnet.

Differenzberechnung

Die Unterschiede der beitragenden Mengen zwischen A und $A^{\oplus w}$ werden durch die zwei *Differenzen decS* und *incS* dargestellt. *incS* soll diejenigen Bereiche der beitragenden Menge von $A^{\oplus w}$ enthalten, die durch die SUO $w = w + 1$ zu $S(A)$ hinzugekommen sind. *decS* enthält die Bereiche, die durch $w = w + 1$ nun nicht mehr referenziert werden. Sie bestimmen sich zu [3]

$$decS(A, \oplus) = S(A) - S(A^{\oplus}) \tag{7.5}$$
$$incS(A, \oplus) = S(A^{\oplus}) - S(A) \tag{7.6}$$

Beispiel 7.3.2 $A_j^{\oplus_i}$ *und* $S(A_j^{\oplus_i})$

$A_j^{\oplus_i}$ *zu Beispiel 7.3.1:*

```
s[i+1] = 0;
for(j = 0; j < i+1; j++)
   s[i+1] += a[j]
```

mit
$S(A_j^{\oplus_i}) = \{j | 0 \leq j < i + 1\}$
und
$decS(A, \oplus_i) = \{\}$ *und* $incS(A, \oplus_i) = \{i + 1\}$.

[3] Die Berechnung und Darstellung der Differenz erfolgt nicht mit expliziten Mengen, d.h. Aufzählungen der enthaltenen Werte, sondern durch eine Formulierung einer Mengeneinschränkung. In der Original-Publikation [10] ist ein Algorithmus zur Berechnung dieser Differenzen angegeben, der auf dem Omega-Test basiert.

Beispiel 7.3.2 zeigt die AAC $A_j^{\oplus_i}$ aus dem Partialsummen-Beispiel von oben, ebenso wie die beitragende Menge $S(A_j^{\oplus_i})$. Entsprechend der Definition werden die Differenzen bestimmt, wobei hier $decS$ leer ist und $incS$ ein Element enthält.

Inkrementalisierung

Die Idee der Inkrementalisierung liegt in der vermehrten Verwendung von Elementen aus $incS(A, \oplus)$ und der Verminderung des Gebrauchs der Elemente aus $decS(A, \oplus)$. Dadurch werden mehr „neue" Elemente referenziert, und weniger auf „alte", d.h. bereits zuvor gelesene, Werte zurückgegriffen. Der Effekt ist, daß durch die Umstellung der Berechnung auf Verwendung erstmalig referenzierter Werte, weniger oft wiederholte und somit redundante Zugriffe erfolgen.

Das Vorgehen zur Inkrementalisierung besteht aus mehreren Schritten. Zunächst müssen die beitragenden Mengen in der Reihenfolge ihres Gebrauchs geordnet werden, damit die neue AAC in der richtigen Reihenfolge arbeiten kann. Dann müssen *zwei* neue Schleifen erzeugt werden, um unterschiedlichen algebraischen Eigenschaften der Akkumulationsfunktion f zu begegnen.

Im einzelnen sind mit dem Ordnen der Mengen folgende Aufgaben verbunden:

- Ordnen von $S(A)$, $S(A^{\oplus})$, $decS(A, \oplus)$ und $incS(A, \oplus)$ in der Gebrauchsreihenfolge in A.
- Einführung von Operatoren $first$ und $last$, die auf den geordneten Mengen das erste bzw. letzte Element zurückgeben.
- Einführung einer Operation, die überprüft ob eine Teilmenge S' *am Ende* einer Menge S liegt. Das ist der Fall, wenn die Elemente aus S' in der Reihenfolge in S denen von $S - S'$ folgen.

Mit diesen Vorarbeiten kann zum Kern der Transformation fortgeschritten werden. Es entstehen zwei Schleifen, von denen die erste Beiträge aus $decS$ entfernt, während die zweite Beiträge aus $incS$ hinzufügt. Beiträge aus $decS$ können nur dann entfernt werden, wenn $decS(A, \oplus)$ nicht leer ist. Zur Entfernung wird die Umkehrfunktion f^{-1} benutzt. Wenn f nicht assoziativ oder kommutativ ist, muß $decS(A, \oplus)$ am Ende von $S(A)$ liegen[4]. Dann können die Elemente von $decS(A, \oplus)$ in der umgekehrten Reihenfolge ihres ursprünglichen Hinzukommens entfernt werden. Somit ergibt sich die erste Schleife zu:

$v^{\oplus} = v;$
for $i = last(decS(A, \oplus))$ downto $first(decS(A, \oplus))$ do
    $v^{\oplus} = f^{-1}(v^{\oplus}, g(a[h(i)], \ldots));$

[4]Liegt $decS(A, \oplus)$ nicht am Ende von $S(A)$, so muß f assoziativ und kommutativ sein.

Die zweite Schleife fügt sukzessive Elemente aus $incS((A, \oplus)$ in ihrer ursprünglichen Reihenfolge hinzu. Dementsprechend sieht sie so aus:

for $i = first(incS(A, \oplus))$ to $last(incS(A, \oplus))$ do
  $v^{\oplus} = f(v^{\oplus}, g(a[h(i)], \ldots))$;

Beide Schleifen zusammen erledigen die gleiche Berechnung wie die Ausgangs-AAC. Falls $decS$ keine Elemente enthält, entfällt die erste Schleife. Ob sich die Transformation lohnt, kann durch den Vergleich von $|decS(A, \oplus)| + |incS(A, \oplus)|$ und $|S(A^{\oplus})|$ abgeschätzt werden, wenn die Umkehrfunktion f^{-1} nicht schwerer zu berechnen ist als f.

Beispiel 7.3.3 *Inkrementalisierte Schleife aus 7.3.2*

```
s[i+1] = s[i];
for(j = i+1; j <= i+1; j++)
   s[i+1] = s[i+1] + a[i+1];
```

bzw. durch Vereinfachung

```
s[i+1] = s[i] + a[i+1];
```

Beispiel 7.3.3 greift die AAC aus 7.3.1 und 7.3.2 wieder auf. Da $decS$ leer ist, kann die erste Schleife entfallen. Die zweite Schleife wird jedoch erzeugt. In $incS$ befindet sich lediglich ein Element, so daß die erzeugte Schleife stark vereinfacht werden kann und zu einer einzigen Instruktion zusammenfällt.

7.3.3 Erzeugung neuer Programmschleifen

Inkrementalisierte AAC greifen auf Ergebnisse vorheriger Iterationen zurück. Zu Beginn der Ausführung einer Schleife gibt es aber noch keine Ergebnisse vorangegangener Iterationen, so daß eine Initialisierung benötigt wird. Dazu können die ersten Iterationen der ursprünglichen Schleife abgerollt werden, so daß dann die inkrementalisierte AAC einsetzen kann. Weiterhin befindet sich die inkrementalisierte AAC zur Zeit noch in einem Zustand, in dem eine folgende Iteration auf die aktuelle Iteration zurückverweist. Es ist praktischer, wenn eine Form vorliegt, bei der die aktuelle Iteration auf einer vorherigen Iteration basiert. Dazu kann einfach w durch $w - 1$ ersetzt werden.

Beispiel 7.3.4 *Optimierte Programmschleife zu 7.3.1*

```
s[0] = a[0];
for(i = 1; i < n; i++)
   s[i] = s[i-1] + a[i];
```

In Beispiel 7.3.4 wird die optimierte Programmschleife des Partialsummen-Berechnung gezeigt. Eine Iteration ist zur Initialisierung abgerollt und vereinfacht. Die restliche Iterationen entsprechen der inkrementalisierten Version. Jedes Vorkommen von i wird durch $i - 1$ substituiert.

7.3.4 Weitere Möglichkeiten

Nicht alle Möglichkeiten des AAC-Verfahrens können in dieser Arbeit in aller Ausführlichkeit vorgestellt werden. Nicht behandelt werden die Optimierung des Zugriffs auf mehrere beteiligte Arrays, mehrdimensionale Arrays und komplexere Schleifenkörper, die selbst wieder Schleifen enthalten können, also loop nests. Insbesondere diese Möglichkeiten können jedoch bei Anwendungen in der Bildverarbeitung, einer wesentlichen Domäne der DSP-Applikationen, von großer Wichtigkeit sein. Bei komplexeren AACs geht schnell der Überblick verloren, so daß sich Redundanzen der gezeigten Art bei der Programmierung einschleichen, und die mit einem automatisierten Verfahren effizient beseitigt werden können. Weiterhin werden auch leistungsfähige, systematische Wege zur effizienten Verwaltung zusätzlich benötigten Speichers zur Verwahrung von Zwischenergebnissen hier ausgespart. Auch der sparsame Umgang mit Speicherplatzressourcen ist bei DSP wichtig, da Speicher oft knapp ist.

7.3.5 Vor- und Nachteile

Der große Vorteil der Inkrementalisierung von AACs liegt in dem großen möglichen Geschwindigkeitsgewinn. Die meisten übrigen Verfahren, insb. die Load- und Store-Redundanz-Eliminationen aus Kapitel 5, sind auf Geschwindigkeitssteigerungen um einen konstanten Faktor beschränkt, also $O(1)$. Die AAC-Inkrementalisierung ist in der Lage, die asymptotische Komplexität zu verringern, z.B. von $O(n^2)$ auf $O(n)$. Daß es nicht nur bei diesen theoretischen Resultaten bleibt, wo für ein konkretes Problem trotz einer asymptotischen Verbesserung trotzdem eine praktische Verschlechterung bewirkt werden kann, zeigen die Beispiele in [10].

Vorteilhaft sind auch die geringen Forderungen an die verschiedenen Funktionen, die an einer AAC beteiligt sind. Selbst die Forderung nach Umkehrbarkeit der Akkumulationsfunktion ist häufig nicht sehr einschränkend, da bei vielen DSP-Applikationen simple und einfach umkehrbare Funktionen zum Einsatz kommen. Diese sind auch oft assoziativ und kommutativ, so daß bei der Inkrementalisierung zu keinen Einschränkungen kommt.

Nachteilig wirkt sich aus, daß unter Umständen neue, d.h. in der Original-Schleife nicht vorhandene, Datenstrukturen eingeführt werden müssen. Dadurch wird mehr Speicherplatz verbraucht. Das kann auch durch effiziente Verfahren zur Speicherplatzverwaltung nicht verhindert werden.

Zu berücksichtigen ist auch, daß das Verfahren zu Inkrementalisierung recht aufwendig ist. Die durchzuführenden Teilschritte erfordern teilweise sehr viel Arbeit bei Implementierung und Ausführung. Allerdings lassen sich alle Schritte automatisieren, so daß kein Eingreifen „von Hand" mehr notwendig ist.

- **Vorteile**
  - Asymptotische Verbesserungen der Laufzeit
  - Geringe Voraussetzungen an verwendete Funktionen
- **Nachteile**
  - Evtl. größerer Speicherplatzbedarf
  - Aufwendiges Verfahren

Insgesamt ist das Verfahren durchaus empfehlenswert. Der hohe Aufwand erscheint insbesondere bei komplexen DSP-Anwendungen gerechtfertigt, denn loop nests mit mehrdimensionalen Arrays lassen sich nicht mehr so einfach vom Programmierer überblicken, so daß z.B. bei Anwendungen in der Bildverarbeitung entsprechende Einsatzmöglichkeiten zu erwarten sind. Bei kleineren AACs wird es wohl seltener Einsatzmöglichkeiten geben, denn durch deren Überschaubarkeit wird der Programmierer sicherlich selbst die Gelegenheit zur effizienteren Programmierung einer Aufgabe sehen. Der zusätzliche Speicherplatzbedarf wird oft dadurch kompensiert, daß bei einer Handoptimierung in ähnlicher Weise zusätzlicher Speicher verwendet würde.

7.4 Beurteilung der verschiedenen Optimierungen

In diesem Kapitel wurden drei Optimierungsverfahren vorgestellt, die allesamt ihre Daseinsberechtigung im Bereich der DSP-Applikationen besitzen. *Loop Unrolling* und *Software-Pipelining* sind etablierte Techniken, die auch häufig benutzt werden. Ihre Verbesserung bei vergleichsweise geringem Aufwand ist von großem Nutzen. Gerade beim *kontrollierten Loop Unrolling* wird dem bei DSP wichtigen Speicherplatzverbrauch Rechnung getragen, um den bei konventionellen Verfahren auftretenden Nachteil des schnell wachsendem Code-Umfangs auf sinnvolle Weise zu begrenzen. Die Unterstützung des Software-Pipelinings ist eine Alternative zu den Redundanzeliminationen der vorangegangenen Kapiteln, für Fälle, in denen die dort beschriebenen Verfahren nicht besonders geeignet sind oder nur unbefriedigende Ergebnisse erbringen. *Aggregate Array Computations* benötigen im Gegensatz zu den vorherigen beiden Optimierungen eine eigene Analyse, d.h. keine Array-Datenflußanalyse. Viele DSP-Programme bieten Gelegenheiten zur Anwendung der Optimierung, und gerade bei Anwendungen in der Bild- und Videoverarbeitung sollte das Verfahren wegen der dort häufiger auftretenden mehrdimensionalen Arrays und Loop Nests gute Resultate erbringen. Es ermöglicht als einziges Verfahren in dieser Diplomarbeit die Erzielung asymptotischer Verbesserungen.

7.5 Weitere Literatur

- Clock, C., Cooper, K.D.
  Combining Analyses, Combining Optimizations
  ACM Transactions on Programming Languages Systems, 17(2), pp. 181-196, March 1995.

- McKinley, K.S., Carr, S., Tseng, C.-W.
  Improving Data Locality with Loop Transformations
  ACM Transactions on Programming Languages and Systems, TOPLAS, 18(4), pp. 424-453, 1996.

- Whitfield, L., Soffa, M.L.
  An Approach for Exploring Code Improving Transformations
  ACM Transactions on Programming Languages Systems, 19(6), pp. 1053-1084, Nov. 1997.

Kapitel 8

Versuche und empirische Resultate

Im Rahmen dieser Diplomarbeit sind einige der in Kapitel 4 beschriebenen Array-Datenflußanalysen und der in Kapitel 5 dokumentierten Load/Store-Redundanz-Eliminationen implementiert worden. Deren Wirkung sollte nicht nur in der Theorie, sondern auch im praktischen Einsatz getestet werden, um die genannten Vor-/Nachteile an Fallbespielen zu quantifizieren. Dazu wurde der LANCE-Compiler [7] verwendet, der einen bequemen Zugang zur IR gewährleistet und einfach um neue Optimierungen (inkl. Analysen) zu erweitern ist.

8.1 Implementierte Datenflußanalysen und Optimierungen

Zur Implementierung ausgewählt wurde die δ-Array-Datenflußanalyse aus Kapitel 4.2 in verschiedenen Parametrisierungen, damit sowohl die Entfernung redundanter Stores (RSE) als auch die Entfernung redundanter Loads (RLE) erprobt werden konnten. Über die einfache RLE hinaus kommt auch die einfache Form des Register-Pipelinings aus Kapitel 5.3 zum Einsatz. Details zur Implementation sind im Anhang A dokumentiert.

8.2 Versuche und Versuchsziele

Bei den Versuchen sollten verschiedene Fragen quantitativ geklärt werden, deren Beantwortung aufgrund verschiedener gegenläufiger Effekte bei den Optimierungen nur eingeschränkt theoretisch möglich ist. Die Versuche zielten darauf ab, folgende Fragen zu beantworten:

- Welchen Vorteil erbringt die Anwendung von RLE/RSE/RP in Bezug auf Geschwindigkeit und Anzahl der Speicherzugriffe gegenüber einer nicht-optimierten Programmvariante?
- Wie verhalten sich RSE/RLE/RP bei unterschiedlichen Iterationsanzahlen der Schleifen und konstanter Tiefe der Datenabhängigkeit?
- Welchen Einfluß hat die Anzahl verschiedener Arrays bzw. Abhängigkeitsketten in der Schleife auf die Effizienz von RSE/RLE/RP?
- Bis zu welcher Iterationsdistanz (Tiefe) ist der Einsatz von RP bei der gegebenen Zielarchitektur (TI C60) sinnvoll?
- Ist die Adressierung von Arrays mittels Pointern der indizierten und optimierten Adressierung überlegen?
- Welchen Einfluß haben andere Optimierungen auf RLE/RSE/RP?
- Wie verhalten sich RLE/RSE/RP bei unterschiedlich komplexen Indexfunktionen?
- Gibt es Unterschiede bei der Anwendung von RLE/RSE/RP gegenüber anderen (oder gar keinen) Optimierungen, wenn im Schleifenkörper nur lesende, nur schreibende oder gemischte Referenzen auf ein Array vorkommen?
- Wie wirken RLE/RSE/RP einzeln und zusammen angewandt?

Als Testprogramme wurden Programme aus der *DSPStone*-Sammlung ebenso wie selbstentwickelte Benchmark-Programme herangezogen. Die DSPStone-Programme referenzieren Arrays nahezu ausschließlich über Pointer, so daß die Optimierungen darauf nicht direkt zum Einsatz kommen können. Deshalb wurden sie in eine Darstellung mit expliziten Array-Referenzen zurück-codiert und in dieser modifizierten Fassung eingesetzt. Nicht alle Fragen aus obiger Liste lassen sich mit den DSPStone-Benchmarks beantworten. Zur Untersuchung vieler Eigenschaften mußten spezielle Testprogramme selbstentwickelt werden. Diese sind über Präprozessor-Kommandos parametrisierbar, um z.B. die Iterationsdistanzen der Abhängigkeiten variieren zu können.

Der Versuchsablauf gliedert sich in mehrere Schritte. Zunächst wird ein gegebenes C-Programm in die IR-Darstellung des LANCE-Compilers überführt. Auf dieser IR können die Optimierungen RLE/RSE/RP arbeiten. Sie erzeugen ihre Ausgabe wiederum in LANCE-IR. Ein Konverter von der LANCE-IR zurück nach C kommt anschließend zum Einsatz. Danach werden Profiling-Kommandos in den generierten C-Code eingefügt, so daß einzelne optimierte Schleifen gezielt untersucht werden können. Mit dem optimierenden Texas Instruments TMS320C6x Compiler (`cl6x`) werden Objektdateien erzeugt, die auf einem Simulator (`load6x`) lauffähig sind. Dieser berichtet die Anzahl der Taktzyklen, die für einen untersuchten Programmausschnitt verbraucht werden. Der Stand-Alone Simulator berechnet für einen externen Speicherzugriff acht Zyklen und rechnet ansonsten einigermaßen taktgenau (siehe dazu dessen Online-Dokumentation).

Optimierungsstufen nach [20]:

-o0 Kontrollflußgraphvereinfachung, Registerallokation, Schleifenrotation, Entfernung nicht-gebrauchten Codes, Ausdrucks- und Instruktionsvereinfachung, Inline-Function-Expansion,

-o1 zusätzlich: lokale Copy/Constant Propagation, Entfernung nicht-gebrauchter Zuweisungen, lokale Common Subexpression Elimination,

-o2 zusätzlich: Software-Pipelining, Schleifenoptimierungen, globale Common Subexpression Elimination, globale Entfernung nicht-gebrauchter Zuweisungen, Konvertierung von expliziten Array-Referenzen in inkrementelle Pointer-Zugriffe, Loop Unrolling,

-o3 zusätzlich: Entfernung nicht-aufgerufener Funktionen, Vereinfachung von Funktionen deren Rückgabewert nie benutzt wird, Inline-Aufrufe kleiner Funktionen, Neuordnung der Funktionsdeklarationen, so daß Attribute der aufgerufenen Funktion dem Aufrufer bekannt werden, Propagierung konstanter Parameter in den Funktionskörper, Identifikation von Variablencharakteristiken auf Datei-Ebene.

8.3 Versuchsbeobachtungen und Resultate

8.3.1 Gemischte einfache Referenzen

Die folgende Test-Schleife enthält vier verschiedene Arrays und mehrere Array-Referenzen. Die Indexfunktionen sind recht einfach, d.h. sie bestehen aus der Induktionsvariable und einem konstanten Summanden. Multiplikative Faktoren enthalten sie nicht. Zwischen den Array-Referenzen im Schleifenkörper bestehen sowohl loop independent als auch loop dependent dependences. Alle beteiligten Arrays werden sowohl gelesen als auch geschrieben. Die Iterationsdistanzen sind der abhängigen Referenzen sind relativ kurz, sie betragen maximal zwei. Für eine Schleife dieser Länge, enthält sie recht viele redundante Array-Zugriffe. Es werden ausschließlich Kopieroperationen ausgeführt, arithmetische Operationen sind nicht vorhanden. Für den TI-Compiler wurde die Optimierungsstufe *-o2* gewählt.

```
for(i = 1; i < LENGTH; i++)
  {
    a[i] = b[i+3];
    d[i+4] = c[i-1];
    a[i+2] = a[i];
    b[i+5] = d[i+3];
    c[i+1] = 5;
  }
```

Die Schleife erlaubt den erfolgreichen Einsatz von RLE bzw. RP und RSE. Abbildung 8.3.1 zeigt die Anzahl der für die Ausführung der Schleife verbrauchten

Takte gegenüber der Anzahl der Iterationen für die nicht-optimierte Schleife, die mit RLE bzw. RSE optimierte Schleife und die mit RLE *und* RSE optimierte Schleife.

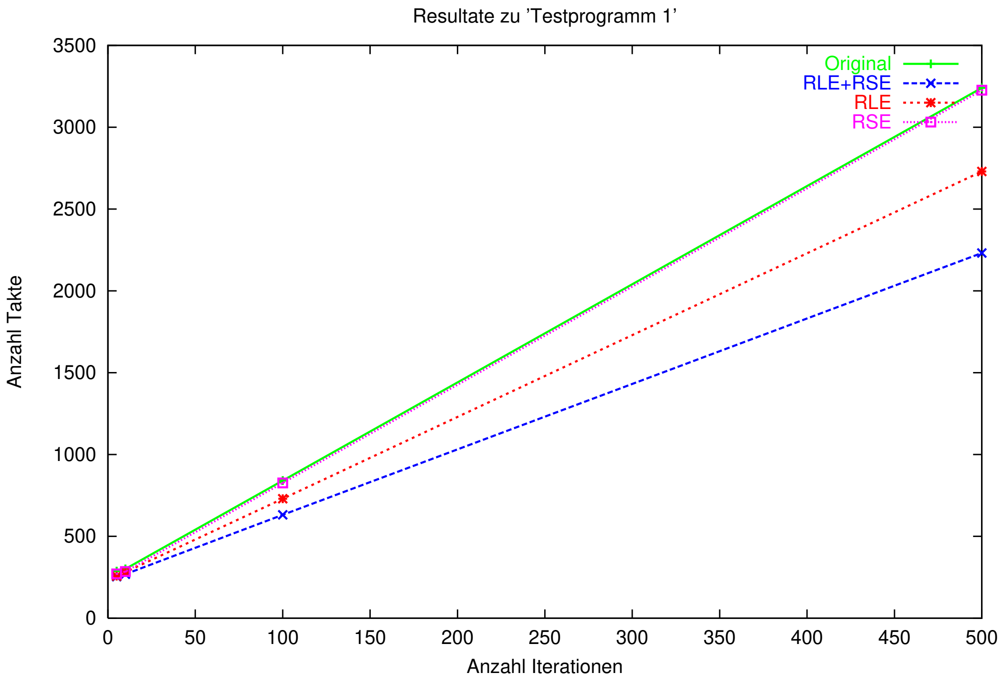


Abbildung 8.1: Ausführungszeiten der Schleife mit einfachen, gemischten Referenzen

Die RSE allein bringt gegenüber der Original-Version kaum eine Verbesserung ein. RLE und RSE+RLE hingegen bewirken eine von der Anzahl der Iterationen abhängige Beschleunigung. Die Hinzunahme von RSE zur RLE allein führt noch einmal zu dem gleichen Geschwindigkeitszuwachs wie die RLE allein. Bei 100 Iterationen ist eine RSE+RLE-optimierte Schleifenvariante 25% schneller als das Original, bei 500 Iterationen sind es schon 31%.

Die „konventionellen" Verfahren des TI-Compilers schaffen es nicht, Array-Datenabhängigkeiten über mehrere Iterationen hinweg zu erkennen und zu behandeln. Die einfachen Verfahren sind wegen der gemischten Zugriffe (lesen/schreiben) nicht in der Lage, die Redundanz der Zugriffe zu vermindern. Der Einsatz von RLE bzw. RLE+RSE führt zur Erkennung und Elimination der redundanten Zugriffe, was zu weniger Ausführungszeit (und weniger Speicherzugriffen) führt.

Der aus dem Rahmen fallenden Eigenschaft, daß RSE allein keine Verbesserung bringt, RLE und RSE zusammen jedoch eine recht hohe Beschleunigung hervorrufen, ist nicht analytisch auf den Grund gegangen worden, dürfte jedoch folgende Ursache haben. In der verwendeten Optimierungsstufe *-o2* werden Array-Referenzen in die inkrementelle Pointer-Darstellung konvertiert. Vermutlich[1] geht damit einher die Anwendung von Verfahren zur Optimierung von Adressierungscode, z.B. nach [4]. Damit wird bezweckt, daß die Post-Inkrement/Dekrement-Operationen der AGU des DSP effizient zur Adressierung des nächsten Array-Elementes genutzt werden. Diese Verfahren basieren auf Analysen, die in Indexierungsgraphen Pfade und Zyklen suchen. Zyklen können effizienter als Pfade zur Optimierung genutzt werden. Wenn die RSE redundante Referenzen entfernt, kann das dazu führen, daß bestehende Zyklen im Indexierungsgraphen zu Pfaden zerfallen. Die Folge ist, daß der Gewinn durch die Redundanzelimination durch den Verlust bei der Adressierung aufgewogen wird. Bei der Anwendung von RLE und RSE zusammen tritt nun dieser Effekt evtl. nicht auf, d.h. es zerfallen keine Zyklen. Damit verstärkt sich der Gewinn beider Verfahren. Die Abhängigkeit des Gewinns der Optimierung – insbesondere der RLE – von der Anzahl der Iterationen ergibt sich aus den fixen Kosten des Schleifenprologs. Bei wenigen Iterationen ist der Beitrag des Prologs im Verhältnis zu den Gesamtkosten höher als bei vielen Iterationen.

Das Beispiel zeigt, daß Schleifen mit lesenden und schreibenden Referenzen auf ein Array RLE- und RSE-Optimierungen sehr zugänglich sind. Wenn die Schleifen redundante Array-Zugriffe enthalten, die sich über mehrere Iterationen erstrecken, so können diese durch die vorgestellten Verfahren entdeckt und beseitigt werden. Schon bei recht einfachen Indexausdrücken kommen „konventionelle" Analysen und Optimierungen mit einer solchen Situation nicht mehr zurecht, so daß sie diese Art von Zugriffen nicht optimieren können.

8.3.2 Gemischte komplexe Referenzen

Die folgende Schleife unterscheidet sich von der vorherigen durch etwas komplexere Indexausdrücke mit additiven und multiplikativen Anteilen. Es sind neben Array-Referenzen auch arithmetische Operationen in der Schleifenkörper eingestreut.

```
for(i = 1; i < LOOP_LENGTH; i++)
  {
    a[3*i+1] = a[2*i+3] * c[i+3];
    c[i-2] = a[2*i-2] - b[2*i];
    b[3*i] += a[3*i+4];
    c[2*i] = b[2*i-2] - b[3*i-3];
  }
```

Um nicht dem schwer zu extrahierenden Einfluß weiterer Optimierungen des TI-Compilers ausgesetzt zu sein, dennoch aber einfache Standardoptimierungen zur Verfügung zu haben, wurde die Optimierungsstufe *-o1* gewählt. Für 100 Iterationen der unoptimierten und optimierten Schleife sind deren Ausführungszeiten in Abb. 8.3.2 dargestellt.

[1] Das TI-Compiler-Manual gibt über die Interna der durchgeführten Optimierungen nur sehr wenig her.

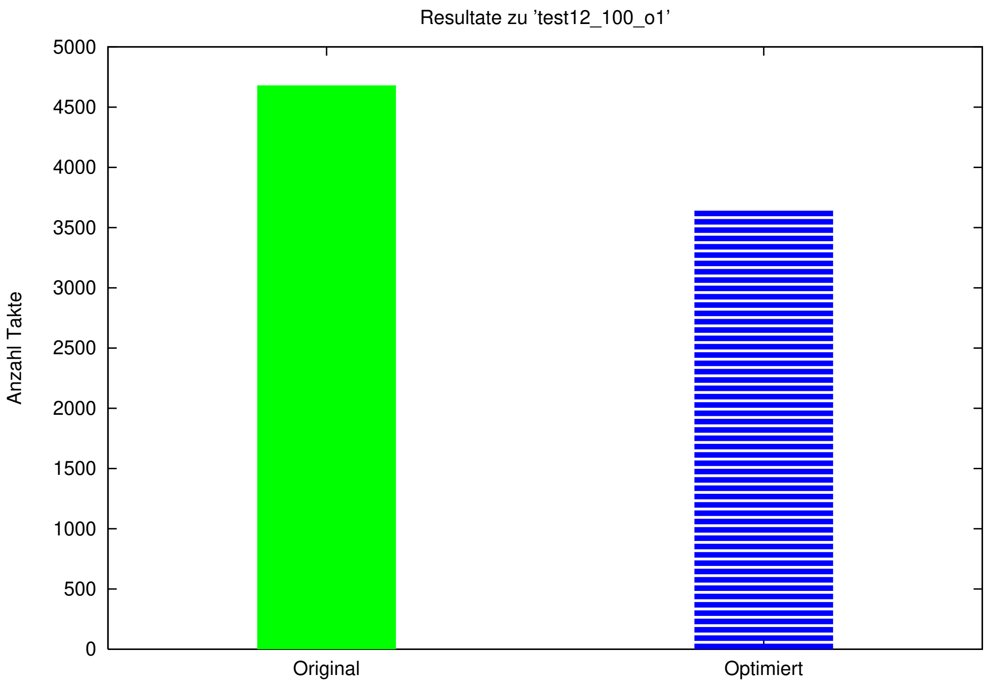


Abbildung 8.2: Ausführungszeiten der Schleife mit komplexeren, gemischten Referenzen

Die Schleife enthält eine Reihe redundanter Zugriffe, die von dem implementierten Verfahren erkannt und eliminiert werden. Gegenüber dem nicht-optimierten Programm ergbit sich eine Verringerung der Ausführungszeit um ca. 20%. Diese (untypisch) hohe Verbesserung ist u.a. auf die relativ kurzen Iterationsdistanzen abhängiger Referenzen zurückzuführen. Die Gesamtkosten der Registerkopieroperationen liegen unter denen von Speicherzugriffen. Die arithmetischen Operationen schaffen die Möglichkeit können gemeinsam mit den Kopieroperationen in einem Schedule zusammengefaßt werden.

Schleifen mit lesenden und schreibenden Zugriffen auf mehrere Arrays, deren Indexfunktionen affine Funktionen sind, kurzen Abhängigkeitsketten und eingestreuten arithmetischen Operationen sind ein bevorzugtes Anwendungsgebiet der RLE/RP-Optimierungen. Nicht immer ist die Anwendung erfolgreich, weil nicht immer ein so hohes Maß an Redundanz wie im gezeigten Beispiel vorliegt, doch eine Anwendung ist auch in Fällen mit geringerer Redudanz erfolgversprechend.

8.3.3 Variation der Abhängigkeitsdistanzen

Zur Klärung der Frage, bis zu welchen Abhängigkeitsdistanzen (in Iterationen) der Einsatz von RP die Ausführungszeit positiv beeinflußt, wurden mehrere Testprogramme entwickelt, bei über Präprozessor-Definitionen die Indexfunktionen verändert werden können. Damit läßt sich die Tiefe der erzeugten

Register-Pipeline beeinflussen und anschließend deren Effizienz mit einer nicht-optimierten Programmvariante vergleichen. Die folgende Schleife verdeutlicht das Prinzip.

```
for(i = 1; i < LOOP_LENGTH; i++)
  {
    b[i] += a[i] * a[i+DISTANCE2];
    a[i+DISTANCE1] = f(i);
  }
```

`DISTANCE1` und `DISTANCE2` sind zuvor über `#define`-Kommandos als Konstanten mit Werten belegt worden. Sie bestimmen die Iterationsdistanzen der Abhängigkeiten zwischen `a[i]`, `a[i+DISTANCE1]` und `a[i+DISTANCE2]`.

Abbildung 8.3 zeigt die Laufzeiten (in Takten) verschiedener Schleifen gegenüber der Tiefe der Register-Pipeline, die für den Fall `DISTANCE1=DISTANCE2` gemessen wurden. Die verwendeten Test-Schleifen unterscheiden sich in der Anzahl der verwendeten Arrays, den Indexfunktionen und dem Maß redundanter Zugriffe. Den optimierten Schleifen werden die Laufzeiten ihrer nicht-optimierten Pendants gegenübergestellt.

Auffällig ist, daß die nicht-optimierten Programmversionen von der Iterationsdistanz der Abhängigkeit unabhängig sind und gleichbleibende Ausführungszeiten haben, die optimierten Versionen jedoch stark von der Tiefe der Register-Pipeline abhängig sind. Die Ausführungszeit steht mit der Tiefe der RP in einem linearen Verhältnis aus der sich die steigende Gerade in den Diagrammen ergibt. Diese Beobachtung deckt sich mit der Theorie, daß mit steigender Länge der RP der Aufwand der Registerkopieroperationen zur Verwaltung der RP zunimmt. Für jede hinzukommende Stufe muß in der RP eine weitere Kopieroperation ausgeführt werden. Interessant ist der Bereich, bei dem das optimierte Programm der nicht-optimierten Schleife überlegen ist. Dieser variiert in den Beispielen zwischen Tiefen der Register-Pipeline unterhalb von sechs bis zwei. Der Schnittpunkt der beiden Geraden zeigt denjenigen Punkt an, bei dem die optimierte Version die gleiche Ausführungszeit benötigt wie die unoptimierte Version. Über den Schnittpunkt hinaus ist die optimierte Verion der nicht-optimierten Version unterlegen. An dieser Stelle wird das Verhältnis zwischen der Dauer eines Speicherzugriffs und eines Registerzugriffs wichtig. Der verwendete Simulator berechnet für einen Speicherzugriff acht Takte und für einen Registerzugriff einen Takt. Wenn die Registerkopieroperationen mit der Tiefe der RP zunehmen, so benötigen sie zusammen auch zunehmend mehr Zeit. Wenn sie die Dauer eines Speicherzugriffs übersteigen, so erzielt die Elimination eines redundanten Speicherzugriffs keinen Geschwindigkeitsvorteil. Grob abschätzen läßt sich die maximale Tiefe der RP, bei der die gleiche Geschwindigkeit zu einer unoptimierten Programmversion erreicht wird, mit $RP_{max} \leq \frac{t_{Speicher}}{t_{Register}}$.

Die Anzahl der Speicherzugriffe ist neben der Geschwindigkeit ein anderes wichtiges Qualitätsmerkmal. Die Elimination redundanter Speicherzugriffe vermindert zunächst die Anzahl an Speicherzugriffen. Auch wenn die Optimierung in Bezug auf Geschwindigkeit nicht erfolgreich sein sollte, so kann sie doch

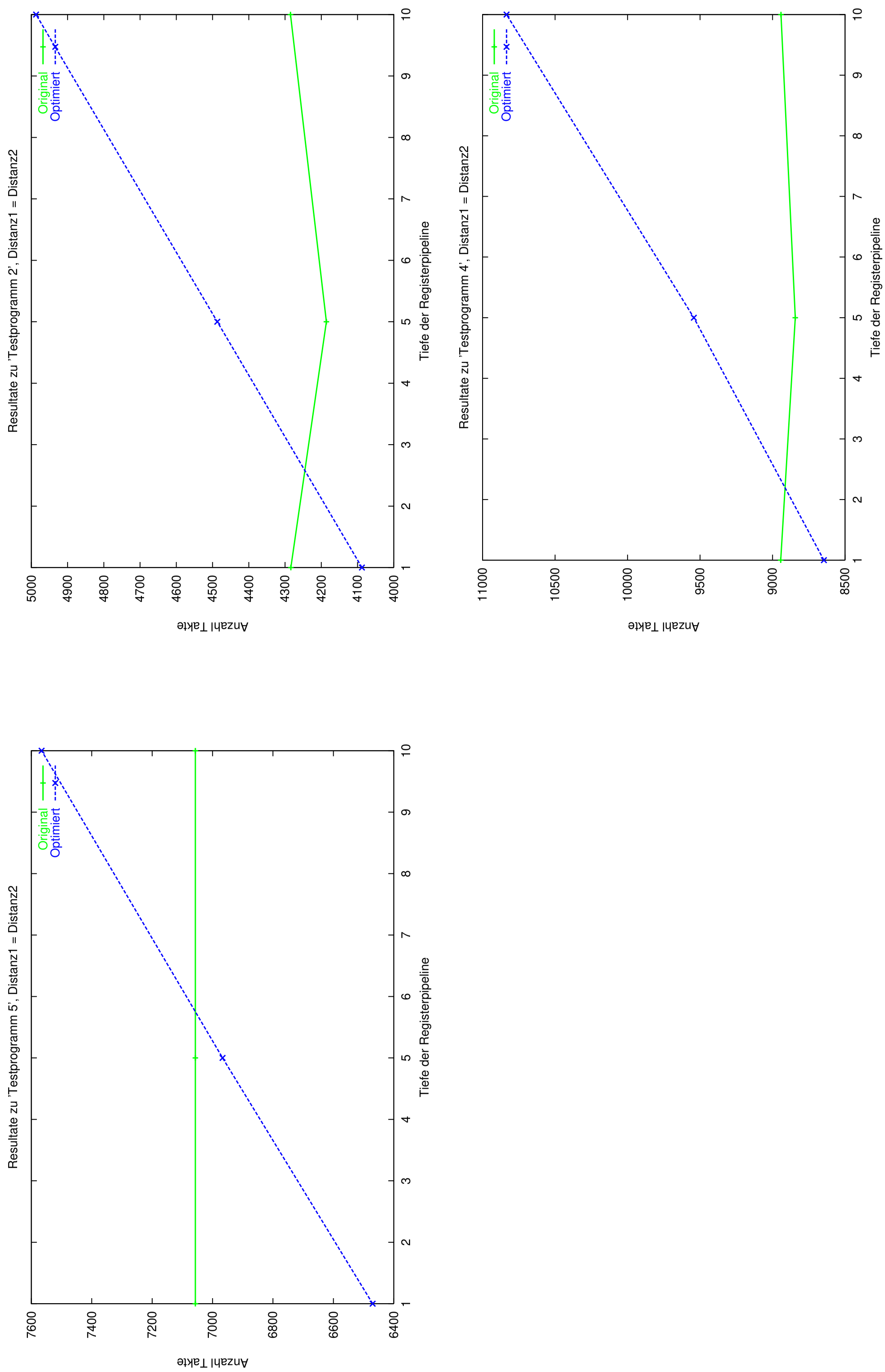


Abbildung 8.3: Ausführungszeiten bei verschiedenen Iterationsdistanzen der Abhängigkeiten

bezüglich der Speicherzugriffe eine erhebliche Verminderung erbringen. Für die oben dargestellte Beispielschleife ergibt durch (statisches) Auszählen der Speicherzugriffsinstruktionen im Schleifenkörper des erzeugten Assemblercodes, daß die Anzahl der lesenden Speicherzugriffe von drei auf eins reduziert werden konnten. Die Anzahl der schreibenden Zugriffe bleibt unverändert bei zwei.

Es gilt zu beachten, daß der zur Verwaltung der RP eingefügte Code die Gewinne aufzehren kann. Wenn die Anzahl der Registerkopieroperationen groß wird, kann es sein, daß zwar die Anzahl der Datenspeicherzugriffe gesenkt werden kann, doch die Anzahl der Instruktionsspeicherzugriffe kann sich umso mehr vergrößern.

Wird die Register-Pipeline viel zu groß, d.h. überschreitet die Tiefe der RP die Anzahl der für allgemeine Zwecke zur Verfügung stehende Register des Prozessors, so kommt es zum Spilling. Abbildung 8.4 zeigt diesen Effekt durch „Abknicken" der Geraden und Übergang zu einer größeren Steilheit. Die Kosten pro Kopieroperation steigen, da nun erneut Speicherzugriffe notwendig sind. Der Punkt des Beginns vom Spilling liegt bei etwa 20, und damit in der Größenordnung der Anzahl der Register eines Datenpfades (16) vom TI C60.

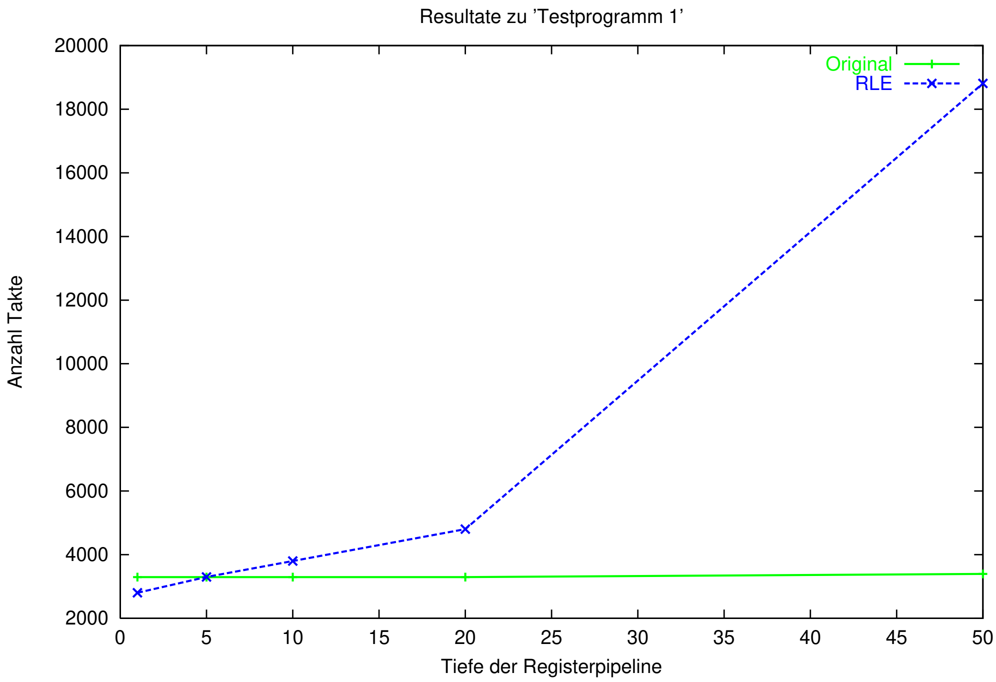


Abbildung 8.4: Ausführungszeiten bei sehr großen Iterationsdistanzen der Abhängigkeiten

Das Halten von Werten in Registern über eine große Anzahl Iterationen ist äußerst ungünstig. Für die vorliegende Architektur kann die Empfehlung gegeben werden, nicht über zehn Iterationen Werte zu transportieren.

8.3.4 Verschiedene Optimierungsstufen

Das Programm *biquad_N_sections* aus dem DSPStone-Paket enthält die folgende Schleife, bzw. deren für explizite Array-Referenzen modifizierte Fassung:

Zugriffe durch Pointer:

```
for (f = 0 ; f < NumberOfSections ; f++)
  {
    w  = y -  *ptr_coeff++ * *ptr_wi1 ;
    w -= *ptr_coeff++ * *ptr_wi2 ;

    y  = *ptr_coeff++ *  w ;
    y += *ptr_coeff++ * *ptr_wi1 ;
    y += *ptr_coeff++ * *ptr_wi2 ;

    *ptr_wi2++ = *ptr_wi1;
    *ptr_wi1++ = w ;

    ptr_wi2++ ;
    ptr_wi1++ ;
  }
```

Explizite Zugriffe:

```
for (f = 1 ; f < NumberOfSections_plus_1 ; f++)
  {
    w  = y - coefficients[5*f-5] * wi[2*f-2];
    w -= coefficients[5*f-4] * wi[2*f-1];

    y  = coefficients[5*f-3] * w;
    y += coefficients[5*f-2] * wi[2*f-2];
    y += coefficients[5*f-1] * wi[2*f-1];

    wi[2*f-1] = wi[2*f-2];
    wi[2*f-2] = w;
  }
```

Es finden wiederholt redundante Zugriffe auf Array-Elemente statt, z.B. wird `wi[2*f-1]` mehrfach referenziert. Die Datenabhängigkeiten sind allesamt loop independent, über die Iterationsgrenzen hinaus reicht keine Abhängigkeit. Bei den Abhängigkeiten handelt es sich um input dependences, so daß die RLE diese optimieren kann.

Im Versuch stehen sich drei Programmvarianten gegenüber. Zum einen die Version mit Pointer-Zugriffen auf Array-Elemente, und zum anderen die Version mit expliziten Array-Zugriffen – einmal unoptimiert und einmal mit RLE optimiert. Der Versuch wird mehrfach wiederholt mit unterschiedlichen Iterationsanzahlen und unterschiedlichen Optimierungsstufen durch den TI-Compiler.

Gemessen werden die Anzahl benötigter Takte gegenüber der Anzahl durchlaufener Iterationen. Die folgenden vier Abbildungen geben einen Überblick über die vier Optimierungsstufen (o0, o1, o2, o3) und zeigen jeweils die drei Schleifenvarianten im Vergleich.

Die Diagramme zeigen, daß die Versionen mit expliziten Array-Zugriffen in etwa gleich schnell sind wie die Pointer-Zugriffsvarianten. Bei höheren Optimierungsstufen sind explizite leicht Array-Zugriffe leicht im Vorteil, bei geringeren Stufen Pointer-Zugriffe. Auffällig ist, daß die RLE-optimierte Array-Variante in allen Fällen wesentlich mehr Takte zur Verabeitung braucht als die beiden übrigen Versionen. Die Anzahl der durchgeführten Iterationen ändert an diesen Verhältnissen nichts.

Die RLE kann in diesem Fall keine Optimierung durchführen, die nicht auch durch andere Optimierungsverfahren erreicht werden könnte. Der wiederholte Gebrauch eines Array-Elementes ohne eine zwischenzeitliche Definition kann durch eine Common Subexpression Elimination festgestellt werden. Eine dann folgende Copy Propagation kann einen erneuten Speicherzugriff verhindern (siehe dazu Beispiel 8.3.1).

Damit kann aber noch nicht erklärt werden, daß die optimierte Variante *schlechter* ist als die unoptimierte. Die RLE fügt explizite Kopieroperationen ein, um den Transport eines Datums vom ersten Gebrauch zu Orten eines erneuten Gebrauchs zu erzielen. Wenn der Wert aber bei dem erneuten Gebrauch noch in einem Register verfügbar ist, so keine Verwendung eines zusätzlichen Registers mit den damit verbundenen Kopieroperationen notwendig. Dieser zusätzliche Aufwand, der auch bei höheren Optimierungsstufen nicht wieder rückgängig zu machen ist, führt zu einer Verschlechterung der Laufzeit.

Beispiel 8.3.1 *Alternative Optimierung*

Vorher:	***CSE:***	***CP:***
`... = w[2*i+2];`	`h = 2*i+2;`	`h = 2*i+2;`
`... = w[2*i+2];`	`... = w[h];`	`h' = w[h];`
	`... = w[h];`	`... = h';`
		`... = h';`

Auch andere – hier nicht dargestellte – Beispiele belegen, daß die Anzahl der Iterationen keine Rolle bei der Anwendung der RLE spielt. Beim TI-Compiler zu bemerken, daß der Gebrauch von expliziten Array-Referenzen in etwa gleiche Ausführungsgeschwindungkeiten bewirkt wie Pointer-Zugriffe.

Loop independent dependences mit mehreren redundanten Lesezugriffen sind also Situationen, in denen die Anwendung von RLE zu ungünstigen Auswirkungen führen kann. Nicht nur die Laufzeit vergrößert sich, auch können mit anderen Verfahren (CSE,CP) gleiche Optimierungen (Entfernung redundanter Zugriffe) erzielt werden.

Ein anderes Beispiel zeigt, daß abhängig von den weiteren Optimierungen mal

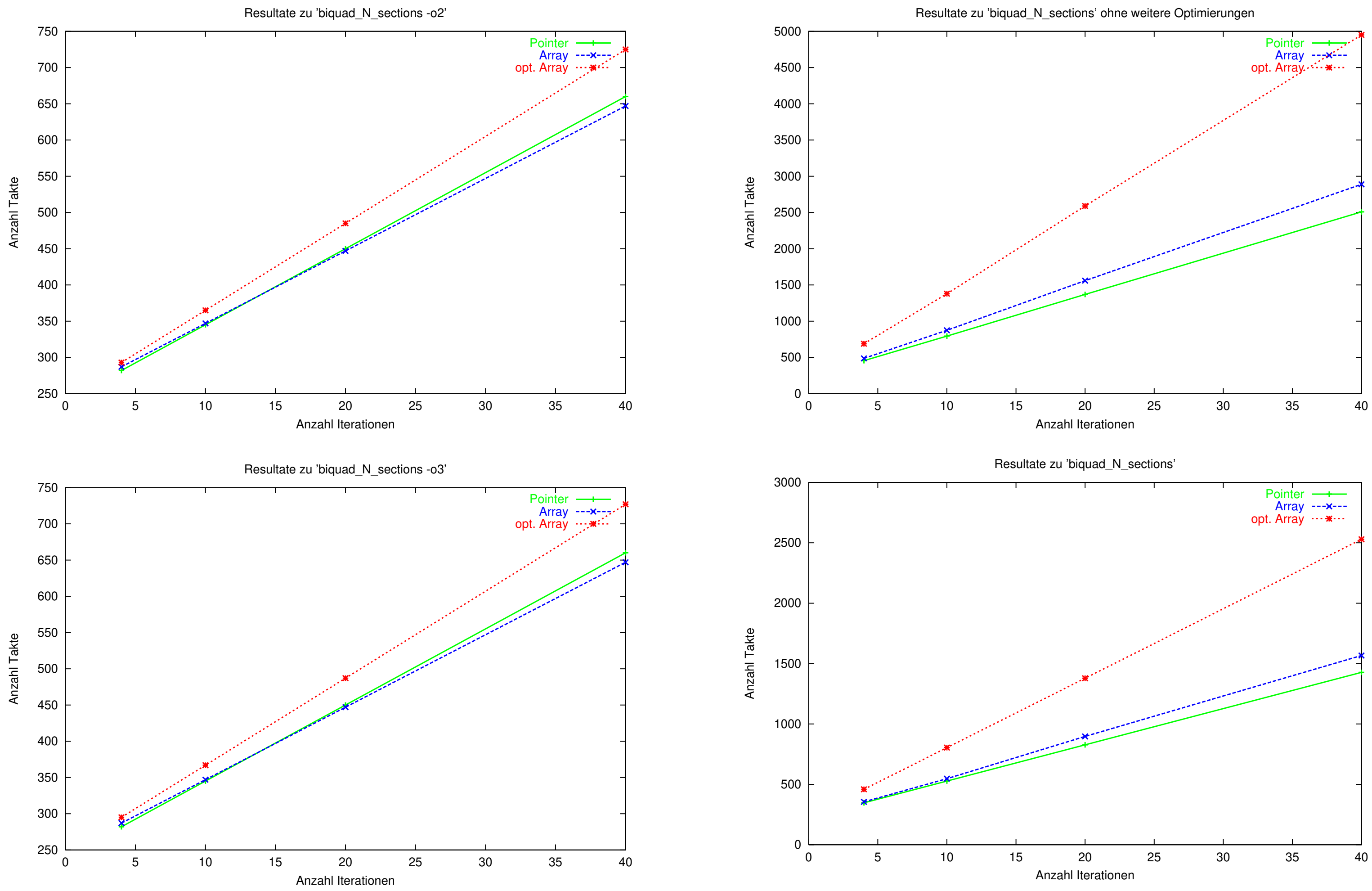


Abbildung 8.5: Ausführungszeiten bei verschiedenen Optimierungsstufen

Vorteile und mal Nachteile zu erwarten sind. Für ein Loop Nest aus der Implementation eines FIR-Filters sind die Original-Schleife und die (von Hand) optimierte Schleife dargestellt.

Nicht optimiert :

```
for (j = 0; j < (m >> 1); j++)
{
    y0 = y1 = round;

    for (i = 0; i < (n >> 1); i++)
    {
        y0 += _mpy  (x[i + j],     h[i]);
        y0 += _mpyh (x[i + j],     h[i]);
        y1 += _mpyhl(x[i + j],     h[i]);
        y1 += _mpylh(x[i + j + 1], h[i]);
    }

    *y++ = (int)(y0 >> s);
    *y++ = (int)(y1 >> s);
}
```

RLE-optimiert :

```
for (j = 0; j < (m >> 1); j++)
{
    y0 = y1 = round;

    t = x[j];
    for (i = 0; i < (n >> 1); i++)
    {
        y0 += _mpy  (t,     h[i]);
        y0 += _mpyh (t,     h[i]);
        y1 += _mpyhl(t,     h[i]);
        t = x[i + j + 1];
        y1 += _mpylh(t, h[i]);
    }

    *y++ = (int)(y0 >> s);
    *y++ = (int)(y1 >> s);
}
```

Die Ausführungszeiten bei den Optimierungsstufen *-o1* und *-o3* sind in Abb. 8.6 dargestellt.

Während die optimierte Version bei *-o1* deutlich langsamer ist als die nicht-optimierte Version, verzeichnet sie hingegen bei *-o3* einen Geschwindigkeitsvorteil. Der Grund liegt hier darin, daß bei der höheren Optimierungsstufe

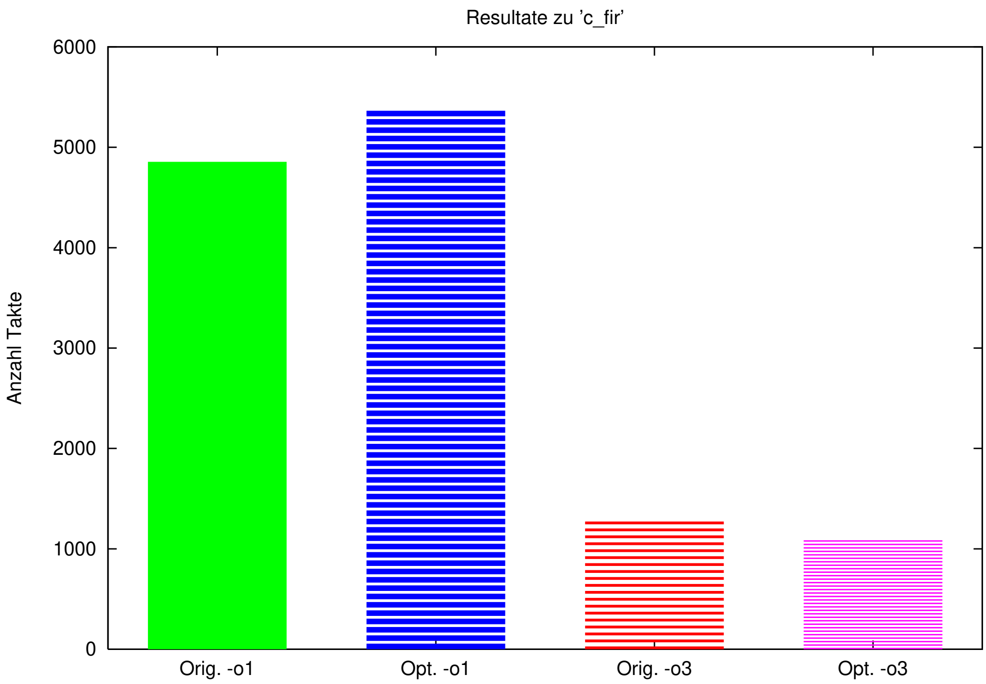


Abbildung 8.6: Ausführungszeiten bei verschiedenen Optimierungsstufen

die Array-Zugriffe durch Pointer-Zugriffe ersetzt werden. Dabei werden durch die explizite Abhängigkeitsdarstellung inkrementelle Pointer-Zugriffe effizient unterstützt, so daß die optimierte Variante hier die Voraussetzung zum erfolgreichen Einsatz einer anderen Optimierung geschaffen hat.

8.3.5 Gegenbeispiele

Wie schon zwischenzeitlich zu sehen war, gibt es Situationen in denen RSE, RLE und RP zu deutlichen Verschlechterungen der Ausführungsgeschwindigkeit können. Ein drastischer Fall ist Schleife aus Beispiel 8.3.2, die aus einer FIR-Filter-Implementation stammt.

Die beiden Felder `coefs` und `input` sind in der die Schleife umgebenden Funktion als `const` deklariert. Optimierungsmöglichkeiten ergeben sich bei `input`, denn nur jeweils ein Element (`input[i+14]`) muß zwingend aus dem Speicher gelesen werden. Die übrigen Elemente können jeweils von Iteration zu Iteration in Registern weitergereicht werden. Zudem können die schleifeninvarianten Zugriffe auf `coefs` aus der Schleife herausgezogen werden, doch das ist in diesem Falle nicht gemacht worden. Die Ausführungszeiten der optimierten und der nicht-optimierten Schleife werden in Abb. 8.7 gegenübergestellt.

Beispiel 8.3.2 ***Schleife mit Möglichkeit zu RP-Optimierung:***

```
for (i = 1; i < 41; i++)
{
    sum  = coefs[0]  * input[i + 14];
    sum += coefs[1]  * input[i + 13];
    sum += coefs[2]  * input[i + 12];
    sum += coefs[3]  * input[i + 11];
    sum += coefs[4]  * input[i + 10];
    sum += coefs[5]  * input[i + 9];
    sum += coefs[6]  * input[i + 8];
    sum += coefs[7]  * input[i + 7];
    sum += coefs[8]  * input[i + 6];
    sum += coefs[9]  * input[i + 5];
    sum += coefs[10] * input[i + 4];
    sum += coefs[11] * input[i + 3];
    sum += coefs[12] * input[i + 2];
    sum += coefs[13] * input[i + 1];
    sum += coefs[14] * input[i + 0];
    sum += coefs[15] * input[i - 1];

    out[i - 1] = (sum >> 15);
}
```

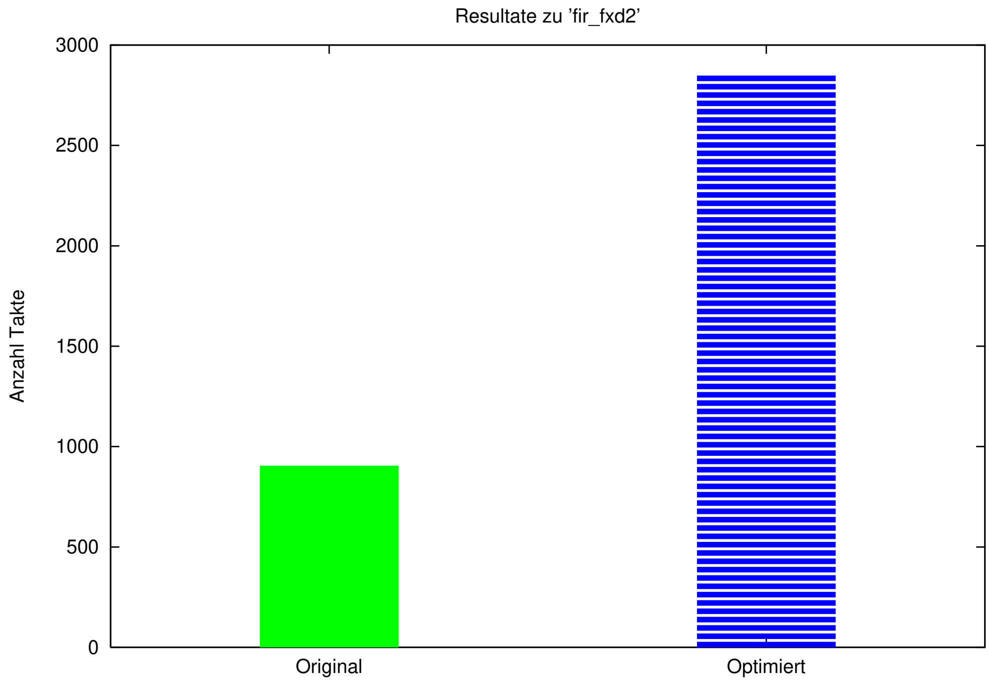


Abbildung 8.7: Geschwindigkeitsverlust durch Optimierung

Es ist ein deutlicher Geschwindigkeitsunterschied zwischen beiden Programmvarianten festzustellen. Die optimierte Version benötigt fast die dreifache Anzahl Takte zur Ausführung.

Die Speicherzugriffe – sowohl auf `coefs` als auch auf `input` – können effizient durch inkrementelle Pointer-Zugriffe erledigt werden, indem jeweils ein Pointer schrittweise durch das Array geführt wird. Bei `coefs` kann sogar die zirkulare Adressierung genutzt werden, um nach Ende der Iteration wieder auf das erste Element zu verweisen. Mittels Dual Load Execution können jeweils zwei Speicherzugriffe zugleich gestartet werden, so daß die beiden Operanden für die Multiplikation jeweils gleichzeitig zur Verfügung stehen. Dann folgen die arithmetischen Operationen, mit deren Ausführungszeit die Latenzen der folgenden Speicherzugriffe verdeckt werden können. Die Anwendung von RP vernichtet die Möglichkeit zur inkrementellen Pointer-Adressierung für `input`. Stattdessen werden pro Iteration Registerkopieroperationen für die Verwaltung der RP notwendig. Die hohe Anzahl an Register für die RP führt zu erhöhtem Registerdruck. Entweder kommt es zum Spilling, oder die Möglichkeit zum Verstecken von Speicherzugriffslatenzen steht nicht mehr zur Verfügung. Beide Alternativen verlangsamen die Programmausführung.

Die Zugriffsoptimierungen des verwendeten Compilers sind bei diesem Beispiel sehr wirksam und nutzen die Fähigkeiten der Zielarchitektur effizient aus. Das verwendete einfache Register-Pipelining kann zwar die Anzahl der Speicherzugriffe vermindern, aber nicht die Geschwindigkeit steigern. Falls überhaupt, könnte mit einer Variante des Register-Pipelinings, die das On-Chip-RAM und die AGU nutzt, eine Verbesserung erzielt werden.

8.4 Bewertungen

Bei den durchgeführten Versuchen konnten für die verwendete Zielarchitektur TI C60 Fälle erarbeitet werden, in denen die Optimierungen RSE/RLE/RP nützlich, bedingt nützlich oder schädlich sind. Bei Vorhandensein hinreichend großer Redundanz – Redundanzelimination bei von vornherein redundanzlosen Programmen ist nicht sinnvoll – können schon erhebliche Verbesserungen mit den einfacheren in dieser Diplomarbeit vorgestellten Verfahren erzielt werden.

Die Merkmale des bevorzugten Anwendungsfeldes sind:

- Lesende und schreibende Referenzen auf ein Array gemischt im Schleifenkörper,
- kurze Iterationsdistanzen zwischen abhängigen Referenzen,
- affine Indexfunktionen mit additiven und multiplikativen Anteilen.

Als bedingt nützlich haben sich die einfacheren Optimierungsvarianten im Zusammenspiel mit anderen Optimierungen herausgestellt. Häufig sind Effekte von Techniken zur effizienten Adressierung von Arrays unter Nutzung der AGU nicht vorhersagbar.

Ungünstige Anwendungsfälle sind diejenigen, in denen lange Abhängigkeitsketten bestehen, die entweder sehr viele beteiligte abhängige Referenzen besitzen, so daß es zum Spilling kommt, oder die Abhängigkeiten über eine große Anzahl an Iterationen haben. Das Halten von Werten über größere Zeitdauern in den Registern ist wegen der Blockade von Ressourcen nicht nützlich. Auch loop independent Abhängigkeiten sollten meistens mit anderen Verfahren als den hier vorgestellten behandelt werden, sofern es sich nicht um komplexe Schleifenkörper handelt.

Zur Beantwortung der Fragen, die am Kapitelanfang als Ziel der Untersuchungen gesetzt wurden, haben die Versuche und deren Analyse einen großen Beitrag leisten können. RLE/RSE und RP können die sowohl die Anzahl der Speicherzugriffe vermindern als auch die Ausführungsgeschwindigkeit vergrößern. Im bevorzugten Anwendungsbereich sind Geschwindigkeitsgewinne gegenüber nicht-optimierten Programmvarianten um bis zu ca. 30% zu erwarten. Solch hohe Gewinne sind aber eher untypisch, da sie eine hohe Redundanz im ursprünglichen Programm voraussetzen. Unter ungünstigen Bedingungen kann es aber auch zu beträchtlichen Einbußen bei Geschwindigkeit von Programmen geben. Wenn es darüberhinaus zum Spilling kommt, wird die Anzahl der Speicherzugriffe u.U. sogar noch vergrößert. Bei der Anwendung der Optimierungen sollte die empfohlene Anwendungsumgebung vorliegen. Für RLE finden sich in typischen DSP-Programmen Anwendungsgelegenheiten, für RSE hingegen allerdings seltener. Die Wirkung von RLE und RSE kann sich zusammen gegenüber der Einzelanwendung verstärken. Der Kosten des Schleifenprologs von RLE/RP amortisieren sich bereits nach wenigen Iterationen. Eine Anzahl verschiedener Arrays in einer Schleife stellen für die Optimierung kein Problem dar, solange die Anzahl von Register-Pipelines nicht zu groß wird. Dann kann es zur Knappheit an Register kommen. Allerdings ist bei üblichen Anwendungen nicht damit zu rechnen, daß derart viele redundante Zugriffe auftreten, die durch RP zu behandeln sind. Für einen TI C60 sollte eine Register-Pipeline (in der einfachen Version) nicht mehr als zehn Stufen haben. Empfehlenswert bis zu fünf Stufen. Die untersuchten Optimierungsverfahren beeinflussen andere Optimierungen wie z.B. Software-Pipelining und Zuweisungsverfahren für Adreßregister. Es gibt Wechselwirkungen, die es im Einzelfall schwer machen, den Erfolg einer Optimierung zu prognostizieren.

Die herausgearbeiteten Ergebnisse gelten zunächst nur für die verwendete Zielarchitektur und den verwendeten Compiler. Eine Übertragung der Ergebnisse auf andere Prozessoren bzw. Compiler ist nicht uneingeschränkt möglich, denn schon kleine Änderungen des Systems quantitative Änderungen hervorbringen. Die Wirksamkeit der Redundanzeliminationsverfahren beruht im wesentlichen darauf, daß Registerzugriffe schneller sind als Speicherzugriffe und daß hinreichend viele allgemeine Register zur Verfügung stehen. Ist das nicht der Fall, so kann sich das Anwendungsfeld der Optimierungen stark verschieben.

Kapitel 9

Konklusionen und Ausblick

Viele Probleme sind bei der Übersetzung hochsprachiger Programme in einen Maschinen-Code für DSP zu bewältigen. Erschwert wird diese Aufgabe durch die hohen Anforderungen an DSP-Anwendungen bzgl. Geschwindigkeit und Speicherplatzbedarf, und durch Eigenarten typischer DSP-Architekturen. Zur Bewältigung der auftretenden Probleme ist die Unterstützung durch optimierende Compiler unerläßlich, da die Optimierung „von Hand“ – z.B. durch Assemblerprogrammierung – aufwendig, teuer und fehleranfällig ist.

Der Beitrag dieser Diplomarbeit liegt in der Zusammenstellung und Eignungsüberprüfung leistungsfähiger Array-Datenflußanalysen und darauf basierender Optimierungstechniken zur Redundanzverminderung von Speicherzugriffen bei DSP als Zielarchitektur. Vorrangig handelt es sich um Techniken zur Erkennung und Vermeidung von redundanten Array-Load- und Store-Operationen, die sich über mehrere Iterationen einer Schleife erstrecken. Als Resümee dieser Diplomarbeit lassen sich die einleitend gestellten Fragen so beantworten:

Welche Datenabhängigkeitsanalysen sind bislang entwickelt worden und was leisten sie?
Für skalare Variable sind eine Reihe verschiedener Datenflußanalysen entwickelt worden, die allesamt zum etablierten Standard aktueller Compilerbau-Lehrbücher gehören. Am weitesten verbreitet – da grundlegend – sind iterative Fixpunkt-Verfahren, die ein Programm durch Transferfunktionen und Eigenschaften eines Programms durch Datenflußverbände modellieren. Auf diesem Modell berechnen sie i.a. eine Approximationslösung eines Datenflußproblems. Nach Einschränkung des Datenabhängigkeitsproblems können exakte Lösungen ermittelt werden. Keine der skalaren Datenabhängigkeitsanalysen kann mit Array-Elementen umgehen.

Welche Anforderungen werden an Datenflußanalysen für Array-Elemente gestellt und wie können sie erfüllt werden?
Datenflußanalysen für Array-Elemente stehen vor der Schwierigkeit, daß die eindeutige Zuordnung zwischen der textuellen Bezeichnung einer Variablen und einem Datenobjekt nicht mehr besteht. Zur Behandlung des Problems müssen

geeignete Datenflußanalysen die Indexfunktionen der Array-Referenzen in die Analyse einbeziehen. Werden zwei Array-Referenzen zusammen mit ihren Indexfunktionen betrachtet, um Datenabhängigkeiten aufzuspüren, so können die Indexfunktionen algebraisch nicht handhabbare Beziehungen untereinander entwickeln. Werden die zu untersuchenden Programmfragmente derart eingeschränkt, daß die darin vorkommenden Indexfunktionen ausschließlich affine Funktionen sind, so die Entscheidbarkeit gewährleistet. Ohne die Einschränkung auf den Bereich der affinen Funktionen können die Datenabhängigkeiten zwischen Array-Elementen nur approximativ bestimmt werden.

Welche Array-Datenflußanalysen stehen zur Verfügung und wodurch unterscheiden sie sich?
Array-Datenflußanalysen, die in der Lage sind, Datenabhängigkeiten zwischen einzelnen Array-Elementen zu bestimmen, lassen sich grob in zwei Kategorien einteilen: speicherbasierte und wertebasierte Analysen. Ihr Unterschied liegt in der Verwendung einer unterschiedlichen Definition von Datenabhängigkeit. Während es für das Vorliegen einer speicherbasierten Datenabhängigkeit ausreicht, daß ein Datenobjekt mehrfach referenziert wird, muß bei einer wertebasierten Abhängigkeit ein Datenfluß zwischen den abhängigen Instruktionen stattfinden.

Welche Datenabhängigkeitsanalysen sind zur Unterstützung von Speicherzugriffsoptimierungen für DSP geeignet?
Von den existierenden Array-Datenflußanalysen haben sich die wertebasierten Verfahren als geeignet für DSP herausgestellt. Speicherbasierte Datenabhängigkeitsanalysen sind zur Unterstützung von Optimierungen bei ILP-Prozessoren wenig hilfreich. Wertebasierte Array-Datenflußanalysen können Ergebnisse mit einer Präzision liefern, die für Load/Store-Optimierungen benötigt wird. Durch die vorgestellten Array-Datenflußanalysen lassen sich sowohl redundante Load- als auch Store-Operationen erkennen. Für affine Indexfunktionen der Array-Referenzen stehen Analysen zur Verfügung, die sehr effizient arbeiten und dabei mit unterschiedlicher Präzision regelmäßig wiederkehrende Zugriffsmuster entdecken können. Während ein weniger präzises Verfahren (δ-Technik) nur die Entdeckung total redundanter Zugriffe ermöglicht, können durch Verfahren (Stretched-Loop, DSA) mit höherer Präzision auch partiell redundante Array-Zugriffe erkannt werden. Der Aufwand, eine exakte Lösung (Lazy) für das eingeschränkte Problem zu erzeugen, hat sich als nicht gerechtfertigt herausgestellt. Viel wichtiger sind Array-Datenflußanalysen, die in der Lage sind, auch mit nicht-affinen Ausdrücken in Indexfunktionen und Verzweigungsbedingungen umzugehen. Zwei unterschiedliche Verfahren, die dies mit unterschiedlicher Präzision ermöglichen, werden vorgestellt. Es erweist sich, daß das ungenauere der beiden (DSA-Verfahren) aufgrund seiner wesentlich reichhaltigeren Möglichkeiten zur Parametrisierung mehr Informationen liefern kann, die bei verschiedenen Optimierungen gebraucht werden, als das genauere Lazy-Verfahren, welches nur einige wenige Optimierungen unterstützt.

Welche Speicherzugriffsoptimierungen sind geeignet für digitale Signalprozessoren?
Zur Beschleunigung der Programmausführung ist es sinnvoll, redundante Speicherzugriffe, insb. Array-Zugriffe, in Schleifen zu entfernen. Ein Array-Zugriff kann nicht nur deshalb redundant sein, weil ein weiterer Zugriff auf das Array-Element in der gleichen Iteration erfolgt, sondern es können durchaus mehrere Iterationen zwischen den beiden Zugriffen liegen. Geeignete Optimierungstechniken entfernen sowohl redundante Array-Load- als auch Store-Operationen. Redundante Store-Operationen können ersatzlos gestrichen werden, während die Entfernung redundanter Load-Operationen einen Ersatz durch Registerkopier-Operationen benötigt. Die Redundant-Load-Elimination und die Redundant-Store-Elimination vermindern die Anzahl der Speicherzugriffe. Gegenüber den Basisversion verbesserte Optimierungsverfahren kommen auch mit partiellen Redundanzen klar und erreichen eine Optimalität hinsichtlich einiger Kriterien. Liegen bei einem redundanten Load mehrere Iterationen zwischen den beiden betreffenden Array-Referenzen, so kann Register-Pipelining zum Einsatz kommen. Dabei werden Werte von der Stelle des ersten (nicht-redundanten) Array-Zugriffs bis zur (redundanten) Wiederverwendung in Registern transportiert, um so auf den erneuten Speicherzugriff zu verzichten. Zum Datentransport werden Registerkopier-Operationen eingefügt.

Können die Besonderheiten der Prozessorarchitektur von DSP zur Unterstützung von Speicherzugriffsoptimierungen genutzt werden?
Wird Register-Pipelining über eine größere Anzahl an Iterationen betrieben, so wird der Aufwand zum Umkopieren der Registerinhalte an den Iterationsübergängen sehr groß. Aus der Beobachtung, daß es sich bei einer Register-Pipeline prinzipiell um eine Queue handelt, an deren einem Ende Werte hineingeschoben und am anderen Ende entnommen werden, entsteht ein Ansatz zur Realisierung als Ringbuffer unter Verwendung DSP-typischer zirkularer Post-Inkrement-Adressierungsarten. Wird die AGU für die Adressierung einer Queue im On-Chip-RAM genutzt, so werden zwar beim Register-Pipelining keine Speicherzugriffe eingespart, aber es entfallen die Registerkopier-Operationen am Iterationsübergang. Für größere Tiefen einer Register-Pipeline können gegenüber der Version ohne Unterstützung durch AGU und On-Chip-RAM Geschwindigkeitsvorteile erzielt werden.

Welche Vor- und Nachteile ergeben sich aus der Anwendung einer Speicherzugriffsoptimierung?
Die Vorteile der vorgestellten Speicherzugriffsoptimierungen liegen in der Beschleunigung der Programmausführung durch die Verminderung der Anzahl der Speicherzugriffe. Weiterhin kann durch die geringere Nutzung der Speicherbusses und des externen Speichers Strom gespart werden. Nachteilig können sich Effekte auswirken, die durch Wechselwirkungen mit anderen Optimierungen entstehen, insb. Software-Pipelining und Verfahren zur Nutzung der AGU bei Array-Zugriffen in Schleifen. Die Nachteile bestehen in der Verhinderung oder Behinderung der genannten Optimierungen und äußern sich durch erhöhte Laufzeiten gegenüber nicht-optimierten Programmversionen. Die untersuchten

Speicherzugriffsoptimierungen behindern die genannten Optimierungen nicht grundsätzlich, sondern nur in einigen Fällen. Diese Fälle konnten z.T. in dieser Arbeit identifiziert werden, so daß beim Einsatz der Optimierungen darauf Rücksicht genommen werden kann.

Wie hängt die Ausprägung des Registersatzes eines DSP mit dem Erfolg bzw. der Anwendbarkeit einer Optimierung zusammen?
Einige der vorgestellten Optimierungen, vornehmlich die Basisversionen der Verfahren RLE und RP, eignen sich überwiegend für DSP mit großen, homogenen Registersätzen. Bei DSP mit heterogenen Registersätzen sind RLE und RP wegen ihres Registerbedarfs häufig nur in beschränktem Umfang einsetzbar, während RSE keine zusätzlichen Register für sich beansprucht. Damit kann RSE auch bei heterogenen Registersätzen erfolgreich eingesetzt werden. Die RP-Version mit Hardware-Unterstützung durch AGU und On-Chip-RAM ist dagegen von der Ausprägung des Registersatzes unabhängig.

Welche weiteren Optimierungen können von Array-Datenflußanalysen profitieren?
Über die Speicherzugriffsoptimierungen hinaus werden Bereiche der Registerallokation und allgemeine Optimierungen durch Array-Datenflußanalysen abgedeckt. Lösungsmöglichkeiten für Probleme bei der Registerallokation für Array-Elemente werden durch Array-Datenflußanalysen eröffnet, ebenso wie die Unterstützung etablierter Optimierungen wie Loop Unrolling und Software-Pipelining.

Eine Optimierung für eine spezielle Klasse von Schleifen, die in typischen DSP-Applikationen sehr häufig vorzufinden ist, ist die Inkrementalisierung von Aggregate Array Computations. Für die Optimierung, die asymptotische Verbesserungen erlaubt, wird eine eigene Analyse benötigt.

Insgesamt hat sich erwiesen, daß der erhöhte Aufwand zur Array-Datenflußanalyse und Optimierung bei DSP als Zielarchitektur gerechtfertigt ist. Den strengen Anforderungen an DSP-Applikationen kann durch Speicherzugriffsoptimierungen, die in besonderer Weise auf Hardware-Merkmale von DSP eingehen und den speziellen Eigenschaften von Signalverarbeitungsalgorithmen entgegenkommen, erfolgreich begegnet werden.

Ausblick

Die Lazy-Datenflußanalyse hat ihren Schwachpunkt in der Einschränkung auf Datenflußabhängigkeiten. Die Analyse wäre sehr viel nützlicher, wenn zusätzlich Anti-, Ausgabe- und Eingabe-Abhängigkeiten erkannt werden könnten. Aber auch die übrigen Analysen könnten verbessert werden. Von großem Nutzen wäre die Erweiterung der DSA-Analyse um ein besseres Verfahren zur Behandlung nicht-affiner Indexfunktionen. Bislang ist die Approximation äußerst grob, so daß an dieser Stelle noch ein großes Potential zu Verbesserungen besteht.

Es bleibt zu überprüfen, welche weiteren Optimierungen durch die in dieser Arbeit vorgestellten Array-Datenflußanalysen ermöglicht werden. Zum einen könnte es interessant sein, bekannte skalare Optimierungen um die Möglichkeit zur Behandlung von Array-Elementen zu verallgemeinern. Zum anderen besteht Bedarf an der Untersuchung, ob Gelegenheiten zu neuen Optimierungen durch die Array-Datenflußinformation geschaffen werden. Insbesondere für DSP mit heterogenen Registersätzen bietet sich ein großes Betätigungsfeld, denn einige der untersuchten Optimierungen führen zu nachteiligen Auswirkungen bei solchen Zielarchitekturen.

Weitere Arbeit an Optimierungen der Speicherzugriffe auf mehrdimensionale Arrays in Loop Nests könnte lohnenswert sein. Die bisherigen Optimierungsverfahren arbeiten bei Abhängigkeiten von mehreren Induktionsvariablen unbefriedigend, da sehr schnell eine hohe Komplexität erreicht wird, ohne dabei wirklich überzeugende Verbesserungen zu erzielen. Evtl. gibt es einfachere Verfahren, die dennoch leistungsfähig sind. Bislang ausgespart wurden Möglichkeiten zur Nutzung sehr großer Registersätze (z.B. mit mehr als 128 Registern). Unter Umständen eröffnet sich durch dieses Hardware-Merkmal eine Gelegenheit zu neuen Optimierungen. Für den Fall, daß DSP zukünftig mit Daten-Caches ausgestattet sein sollten, bekommen Schleifentransformationen eine hohe Bedeutung, die zum Ziel haben, eine größere Daten-Lokalität zu schaffen.

Literaturverzeichnis

[1] Aho, A.V., Sethi, R., Ullman, J.D.
Compiler: Principles, Techniques, and Tools
Addison-Wesley, Reading, Mass. 1985.

[2] Analog Devices Incorporated
ADSP-2183 Datasheet
`http://www.analog.com`, 1998.

[3] Appel, A.W., Ginsburg, M.
Modern Compiler Implementation in C
Cambridge University Press, Cambridge, United Kingdom, 1998.

[4] de Araujo, G.C.S.
Code Generation Algorithms for Digital Signal Processors
Dissertation, Princeton University, Department of Electrical Engineering, June 1997.

[5] Benitez, M.E., Davidson, J.W.
Target-specific Global Code Improvement: Principles and Applications
Technical Report CS-94-42, Department of Computer Science, Univesity of Virginia, Charlottesville, 1994.

[6] Bodik, R., Gupta, R.
Array Data Flow Analysis for Load-Store Optimizations in Fine-Grain Architectures
International Journal of Parallel Programming, 24(6):481-512, December 1996.

[7] Dingel, S., Leupers, R.
LANCE - LS12 ANSI C Compilation Environment - User's Guide
`http://ls12-www.informatik.uni-dortmund.de/~leupers/`
Universität Dortmund, March 29, 1999.

[8] Duesterwald, E., Gupta, R., Soffa, M.
A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations
Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation, 28(6):68-77, June 1993, Albuquerque, New Mexico.

[9] Kolson, D.J., Nicolau, A., Dutt, N.
Elimination of Redundant Memory Traffic in High-Level Synthesis
Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, No. 11, November 1996.

[10] Liu, Y.A., Stoller, S.D.
Loop optimization for aggregate array computations
Proceedings of the 1998 International Conference on Computer Languages, IEEE, 1998.

[11] Martin, F.
PAG - an efficient program analyzer generator
Universität des Saarlandes, FB14 Informatik, Saarbrücken, 1999.

[12] Maslov, V.
Lazy Array Data-Flow Dependence Analysis
Technical Report CS-TR-3110.1, University of Maryland, College Park CS, July 1993.

[13] Maydan, D.E., Amarasinghe, S.P., Lam, M.S.
Data Dependence and Data-Flow Analysis of Arrays
Conference Record of the 5th Workshop on Languages and Compilers for Parallel Computing, 1992.

[14] Maydan, D.E., Hennessy, J.L., Lam, M.S.
Effectiveness of Data Dependence Analysis
International Journal of Parallel Programming, 23(1):63-81, 1995.

[15] Muchnick, S.S.
Advanced Compiler Design and Implementation
Morgan Kaufmann Publishers, San Francisco, California, 1997.

[16] Panda, P.R., Dutt, N.D., Nicolau, A.
Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications
Proc. 1997 European Design and Test Conference (ED&TC,1997), March 1997.

[17] Rau, B.R.
Data Flow and Dependence Analysis for Instruction Level Parallelism
Lecture Notes in Computer Science, Vol. 589, p. 236-250, 1991.

[18] Sjödin, J., Fröderberg, B., Lindgren, T.
Allocation of Global Data Objects in On-Chip RAM
Whole Program Optimization for Embedded Systems Project, University of Uppsala, December 1998.

[19] Texas Instruments Incorporated
TMS320C2x User's Guide
January 1993.

[20] Texas Instruments Incorporated
TMS320C6x Optimizing C Compiler User's Guide
July 1997.

[21] Tietze, U., Schenk, Ch.
Halbleiterschaltungstechnik
10. Auflage, Springer-Verlag, 1993.

[22] Wegener, I.
Begleitunterlagen zur Stammvorlesung „Effiziente Algorithmen“
Universität Dortmund, SS 1995.

Anhang A

Dokumentation der Implementation

Im Rahmen dieser Diplomarbeit sind zur Durchführung empirischer Untersuchungen zur Qualität der recherchierten Analysen und Code-Optimierungen drei Tools entwickelt worden, deren Einsatz in Verbindung mit dem LANCE-Compiler möglich ist. Im einzelnen handelt es sich dabei um Implementationen der *Redundant Load Elimination* und der *Redundant Store Elimination* nach [8], die in Kapitel 5 ausführlich beschrieben werden. Beide Programme lesen Programme in LANCE-IR ein und schreiben sie in gleicher Darstellung wieder zurück. Weiterhin entstand ein Tool zur Konvertierung der LANCE High Level-IR zurück in C-Code. Damit wird Einsatz anderer Compiler als LANCE nach der Optimierung durch o.g. Tools ermöglicht.

Die beiden Optimierungstools sind sowohl von der Kommandozeile als auch aus der graphischen Benutzeroberfläche des LANCE zu bedienen. Sie passen sich dem in [7] beschriebenen Standard zur Erweiterung von LANCE an und weisen dabei keine Besonderheiten auf.

A.1 Redundant Load Elimination

Das Tool zur *Redundant Load Elimination* ermöglicht das Erkennen und Eliminieren redundanter Ladezugriffe auf Array-Elemente. Dabei werden die in 5.2 gezeigten Optimierungen durchgeführt, sowie eine Verallgemeinerung in Form einer sehr einfachen Registerpipeline. Die in 5.3 beschriebene Registerpipeline basiert auf einer der Bestimmung einer Priorität zur Zuweisung eines Registers an eine Variable mit einer gegeben Lebensdauer in einem integrierten Registerinterferenzgraphen. Da während der implementierten Optimierung, die architekturunabhängig ist, noch keine Registerallokation stattfindet, kann jenes Verfahren auch noch nicht zum Einsatz kommen. Daher wird eine Registerpipeline mit einen Satz skalarer, temporärer Variablen realisiert. Die Entscheidung darüber, welche Variablen in welchen Register zu halten sind, obliegt nachfolgenden Phasen der Registerallokation und Codegenerierung.

A.1.1 Voraussetzungen und Einschränkungen

Zum erfolgreichen Auffinden von optimierbaren Schleifen ist es notwendig, daß diese strukturiert sind und ein Iterationsintervall $[1 \ldots UB]$ mit einer festen oberen Grenze UB haben. Insbesondere ist die für C-Programme etwas unübliche untere Iterationsgrenze von 1 zu beachten. Ansonsten gelten die in Kapitel 5.2 erwähnten Voraussetzungen. Die Behandlung von geschachtelten Schleifen – und damit auch von mehrdimensionalen Arrays – ist nicht implementiert, jedoch ist die Implementation für eine zukünftige Erweiterung um diese Möglichkeit vorbereitet. Die Einschränkung auf einfache Schleifen und Arrays ist vor dem Hintergrund beschränkter zeitlicher Ressourcen bei der Implementierung der Optimierungen vertretbar, zumal die Leistungsfähigkeit des Verfahrens bei mehrdimensionalen Arrays auf die Erkennung von Abhängigkeiten in je einer Dimension beschränkt ist. Das Tools arbeitet auf der LANCE-High-Level-IR bei der `FOR`-Schleifen und `IF-THEN`-Konstrukte erhalten bleiben. Ebenso müssen Indexausdrücke vollständig erhalten sein. Somit sollte bei der Konfiguration des C-Frontends darauf geachtet werden, daß die Optionen `IR_split_conditionals`, `IR_split_forloop` und `IR_split_index` deaktiviert sind.

A.1.2 Bedienung

Obwohl eine Bedienung von der Kommandozeile aus möglich ist, empfiehlt sich die Verwendung der graphischen Benutzeroberfläche. Der Aufruf des Tools aus der Kommandozeile erfolgt mit:

```
rl-elim.$(LANCE_OS) (verbose=[0|1]) <filename.c>
```

Der optionale Kommandozeilenparameter `verbose` bestimmt dabei, ob Informationen zur gerade durchgeführten Verarbeitung ausgegeben werden sollen (1) oder nicht (0). Im wesentlichen handelt es bei diesen Ausgaben um Informationen zum Wert der jedem Knoten des Schleifenkontrollflußgraphen zugeordneten Verbandselemente, sowie um erkannte Kandidaten zur Elimination. Da es sich also im wesentlichen um Informationen zur Unterstützung der Wartung der Software handelt, ist es im „normalen“ Betrieb nicht ratsam, die Ausgabefunktion zu aktivieren. Für `<filename.c>` ist der Dateiname eines zuvor in die LANCE High-Level-IR transformierten Programmes einzusetzen.

Bei Verwendung der graphischen Oberfläche kann durch einfaches Anklicken des *rl-elim*-Buttons das zugehörige Tool gestartet werden. Wie üblich erscheinen im Ausgabefenster Statusmeldungen und ein kurzer Überblick über die Anzahl der durchgeführten Optimierungen, dagegen im IR-Fenster die neue (optimierte) Darstellung des zuvor geladenen Programms.

A.1.3 Konfiguration

Das RLE-Tool kann durch zwei Konfigurationsparameter in seiner Funktion beeinflußt werden. Es sind der Integer-Parameter `RLE_iteration_limit`, der die maximal zulässige Iterationstiefe von zu behandelnden Datenabhängigkeiten festlegt, und der Integer-Parameter `RLE_temporaries_limit`, der die maximale Anzahl temporärer Variablen zur Zwischenspeicherung von Arrayelementen zwischen den Punkten der Definition und des Gebrauchs bestimmt. Durch Variation von `RLE_iteration_limit` kann auf die Anzahl der Iterationen Einfluß genommen werden, über die Zwischenwerte temporär gespeichert werden. Werden die Abhängigkeitskette länger als durch diesen Wert vorgegeben, wird eine mögliche Optimierung unterlassen. Indirekt wird dadurch auch die Gesamtanzahl erzeugter temporärer Variablen in einem Schleifenkörper beeinflußt. Direkten Einfluß auf diese Anzahl läßt sich durch Verändern von `RLE_temporaries_limit` ausüben. Bei Erreichen dieser Grenze werden alle weiteren möglichen Optimierungen nicht mehr durchgeführt.

Beide Konfigurationsparameter befinden sich in der Datei `RLE.cfg` und können dort von Hand verändert werden. Bei Verwendung der GUI kann über die Menüleiste das Pull-Down-Menü *Config* geöffnet werden, dort befindet sich hinter dem Eintrag *Redundant Load Elimination* ein entsprechendes Fenster zur Voreinstellung der beiden Werte.

A.1.4 Aufbau und Struktur der Implementation

Das RLE-Tool ist modular aufgebaut, d.h. einzelne in ihrer Aufgabe zusammenhängende Teile sind in einzelnen Dateien, teilweise in eigenen Klassen, untergebracht. Die einzelnen Module werden von einer übergeordneten Datei `rl-elim` zusammengeführt, dort befinden sich auch die Definitionen global benötigter Variablen und die `main`-Funktion (siehe Abb. A.1).

Die einzelnen Module haben dabei folgende Aufgaben:

`rl-elim`	Hauptmodul, Laden und Speichern der Datei, globale Variablen definieren.
`prepass`	Herausziehen und Vereinzeln von Arrayzugriffen aus einzelnen Anweisungen, so daß pro Anweisung maximal ein Arrayzugriff stattfindet.
`findloop`	Durchsuchen einer Anweisungsfolge nach zur Optimierung geeigneten Schleifen, Erzeugen einer Datenstruktur zum Speichern der zur Schleife gefundenen Information.
`FLCFG`	Konstruktion des Vorwärts-Schleifenkontrollflußgraphen.

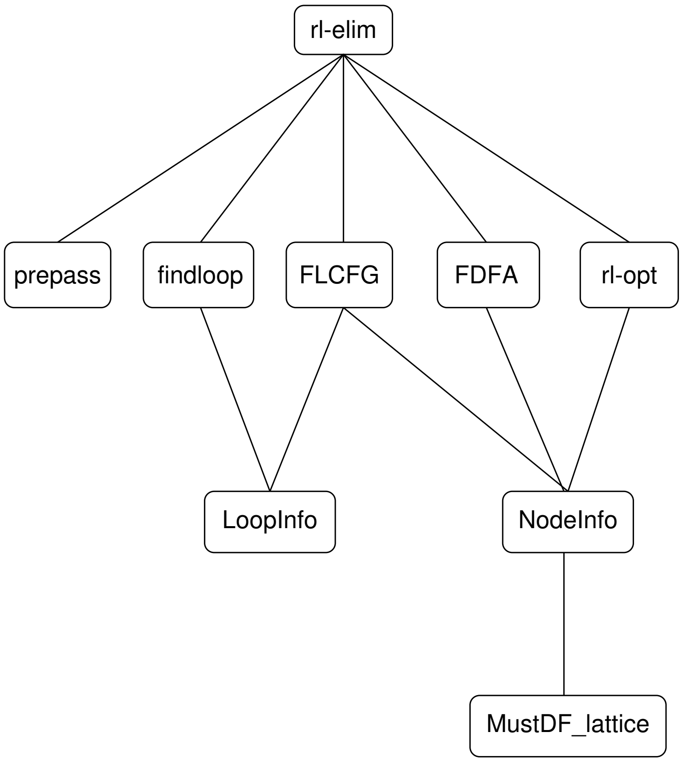


Abbildung A.1: Wesentliche Verbindungen einzelner Programm-Module der *Redundant Load Elimination*

`FDFA`	Initialisierung des Vorwärts-Datenflußgleichungssystems und iterative Fixpunkt-Lösung.
`rl-opt`	Auswertung der Analyseergebnisse und Elimination redundanter Array-Ladezugriffe.
`LoopInfo`	Klasse zum Zwischenspeichern von relevanten Schleifeninformationen.
`NodeInfo`	Klasse zum Speichern von an die Knoten des LCFG gebundenen Informationen.
`MustDF_lattice`	Datentyp zur Repräsentation des während der DFA verwendeten Verbandes mit seinen Operationen.

Das Modul `prepass` sorgt dafür, daß pro Anweisung maximal eine Array-Referenz erfolgt. Damit kann auf Listen-Datenstrukturen zur Verwaltung der Referenzen in den Knoten des LCFG verzichtet werden, ohne daß die Leistungsfähigkeit des Verfahrens eingeschränkt wird. Zusätzlich eingefügte Kopieroperationen und temporäre Variablen können leicht in nachfolgend durchzuführenden *Copy Propagation*- und *Dead Variable Elimination*-Läufen wieder entfernt werden. Die zeitliche Abfolge der einzelnen Aufgaben gibt Abb. A.2 an.

Während die meisten Module in „konventioneller“ Weise, d.h. imperativ, pro-

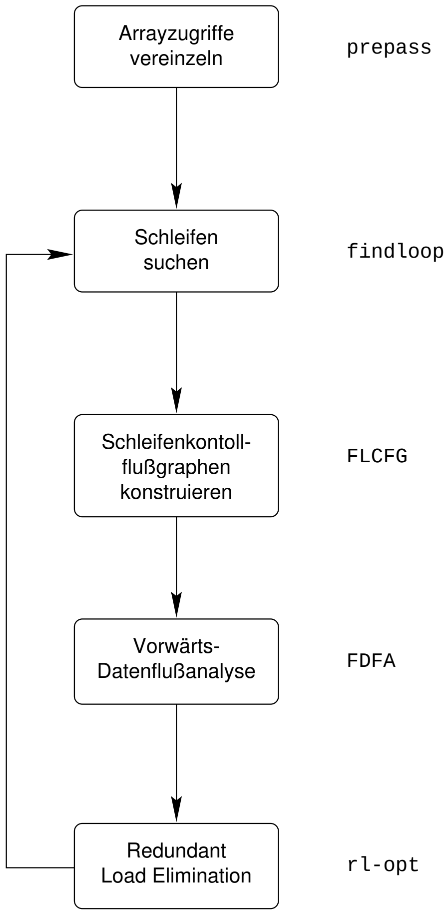


Abbildung A.2: Zeitliche Abfolge der Bearbeitung während der *Redundant Load Elimination* und Zugehörigkeit zu den Programm-Modulen

grammiert wurden, macht die insbesondere die Umsetzung der bei der Datenflußanalyse verwendeten Verbände Gebrauch von den Möglichkeiten der objektorientierten Programmierung. So wurde die Klasse `MustDF_lattice` wie folgt spezifiziert:

```
class MustDF_lattice
{
  friend ostream& operator<<(ostream&, MustDF_lattice);

  friend int operator==(MustDF_lattice, MustDF_lattice);

  friend int operator<(MustDF_lattice, MustDF_lattice);

protected:
  int range, value;

public:
```

```
  MustDF_lattice();

  MustDF_lattice(int);

  MustDF_lattice operator++(int);

  void set_range(int);

  void set_value(int);

  void set_top();

  void set_bottom();

  int is_top();

  int is_bottom();
};
```

Intern – und nach außen nicht sichtbar – gibt es die Attribute `range` und `value`, die die Größe des linearen Verbandes bzw. den Wert des aktuellen Elements repräsentieren. Zur Erzeugung eines Verbandsobjektes dienen die Konstruktoren `MustDF_lattice()` und `MustDF_lattice(int)`. Während ersterer einen Verband kreiert, dessen Größe später mit `set_range(int)` noch festgelegt werden muß, kann diese Information dem zweiten Konstruktor gleich mitgegeben werden. Die Festlegung auf einen aktuellen Wert erfolgt mit `set_value(int)`, oder allgemeiner für die Elemente $\top$ und $\bot$ mit `set_top()` und `set_bottom()`. Entsprechend kann auf Gleichheit mit $\top$ und $\bot$ mittels `is_top()` und `is_bottom()` getestet werden. Die in der Datenflußanalyse benötigten Operationen `min` und `max` auf Verbandselementen werden durch in Standardbibliotheken vorhandene Funktionen realisiert, die durch die Definition eines Vergleichs-Operators `<` auf `MustDF_lattice` zugänglich werden. Analog gibt es einen `==` Operator. Entsprechende Erweiterungen des `ostream` stehen zur typsicheren Ausgabe bereit.

A.1.5 Compilierung und Installation

Die Erzeugung des lauffähigen Programms aus den Quelltexten wird durch ein `Makefile` gesteuert. Nach Aufruf von `make` steht bei zuvor korrekt installiertem LANCE im `BIN`-Verzeichnis das `rl-elim.$(LANCE_OS)`-Executable zur Verfügung. Die Konfigurationsdatei `lance.cfg` ist um folgende Einträge zu erweitern:

```
#config RLE.cfg          Redundant_Load_Elimination
RLE_iteration_limit      int
RLE_temporaries_limit    int
```

Anschließend muß im `CONFIG`-Verzeichnis noch die Datei `RLE.cfg` mit dem Inhalt

```
RLE_iteration_limit = 5
RLE_temporaries_limit = 10
```

angelegt werden. Andere Werte als 5 oder 10 sind selbstverständlich möglich und können nach Belieben festgelegt werden.

A.2 Redundant Store Elimination

Das *Redundant Store Elimination*-Tool dient zur Erkennung und Entfernung von redundanten Array-Schreibzugriffen. Es werden die im Kapitel 5.1 vorgestellten Optimierungen umgesetzt.

A.2.1 Voraussetzungen und Einschränkungen

Es gelten die bei der *Redundant Load Elimination* getroffenen Aussagen auch hier.

A.2.2 Bedienung

Das RSE-Tool wird genauso wie das RLE-Tool bedient. Die Kommandozeilenversion wird mit dem Kommando `rs-elim.$(LANCE_OS)` gestartet.

A.2.3 Konfiguration

Für das RSE-Tool gibt es einen Konfigurationsparameter `RSE_iteration_limit`. Dieser begrenzt erwartungsgemäß die Anzahl der Iterationen bei der Behandlung eines δ-redundanten Stores. Entsprechend kann darüber Einfluß genommen werden auf die maximale Häufigkeit des Aneinanderfügens des Schleifenkörpers im Schleifenepilog. Bei großen Schleifenkörpern und großer Iterationsdistanz zwischen abhängigen Stores kann der Epilog durchaus einen beträchtlichen Umfang erreichen, der die Optimierungen im Schleifenkörper nicht rechtfertigt.

A.2.4 Aufbau und Struktur der Implementation

Auch hier gelten obige Ausführungen analog. Statt eines Vorwärts-Schleifenkontrollflußgraphen und einer Vorwärts-Datenflußanalyse gibt es an dieser Stelle einen Rückwärts-Schleifenkontrollflußgraphen `RLCFG` und eine Rückwärts-Datenflußanalyse `RDFA`. Das Optimierungsmodul heißt nun `rs-opt`.

A.2.5 Compilierung und Installation

Die Compilierung erfolgt ebenfalls mittels eines `Makefile`-Skripts. Die LANCE-Konfigurationsdatei `lance.cfg` muß um folgenden Abschnitt erweitert werden:

```
#config RSE.cfg          Redundant_Store_Elimination
RSE_iteration_limit      int
```

und im `CONFIG`-Verzeichnis die Datei `RSE.cfg` mit dem Inhalt

```
RSE_iteration_limit = 5
```

angelegt werden. Fertig!

A.3 IR-C-Konverter

Der IR-C-Konverter dient zur Re-Transformation von LANCE-IR-Code in C-Darstellung. Es handelt sich dabei um ein kleines Tool mit Prototypen-Charakter. Zweck dieser Transformation ist es, nach Durchführung von *Redundant Load / Store Eliminations* die Möglichkeit zu haben, das optimierte Programm mit einem anderen C-Compiler als LANCE für eine beliebige Zielarchitektur zu übersetzen. Unter Einsatz von Simulatoren oder durch manuelle Codeanalyse können die Auswirkungen der Optimierungen untersucht und bewertet werden.

A.3.1 Voraussetzungen und Einschränkungen

Der IR-C-Konverter arbeitet z.Z. ausschließlich auf der LANCE-IR mit High-Level-Statements. Weitergehende Fähigkeiten sind für die bisherigen Einsatzfälle nicht notwendig, einer entsprechenden Erweiterung gegenüber ist der Quellcode aber durchaus offen.

A.3.2 Bedienung

Die Bedienung erfolgt ausschließlich aus der Kommandozeile. Es gibt keine weiteren Parameter als den Namen des zur IR-Erzeugung genutzten C-Files. Ein Aufruf sieht somit wie folgt aus:

```
irc.$(LANCE_OS) <filename.c>
```

Es werden das im `OUTPUT`-Verzeichnis liegende IR-File und die Symboltabelle gelesen, und im aktuellen Arbeitsverzeichnis wird die Datei `<filename.c.ir.c>` geschrieben. Diese Datei kann anschließend beliebig weiterverarbeitet werden.

Anhang B

Literatur zu Alias- und Pointer-Analysen

In vielen bestehenden DSP-Programmen wird ein intensiver Gebrauch von Pointern betrieben. Vorrangig werden durch Pointer einzelne Array-Elemente adressiert. Durch den häufigen Einsatz von Pointer-Arithmetik sind in dieser Weise erstellte Programme nicht nur für Menschen schwer lesbar, sondern auch für optimierende Compiler schwer zu analysieren. Ein großes Problem bei der Datenflußanalyse von Pointern besteht darin, daß zwischen statischen Zeigervariablen und dynamischen Datenobjekten keine eindeutige Zuordnung bestehen muß. So kann ein Zeiger in C auf beliebige Datenobjekte zeigen, mit der Folge, daß nicht immer klar ist, ob ein Pointer auf ein Objekt zeigt, das eventuell auch über einen anderen Pointer zu erreichen ist. Aus dieser Unsicherheit heraus vermeiden viele Compiler die Optimierung Pointer-Zugriffen vollständig. Mit besseren Datenflußanalysen für Zeigervariablen könnten weitere bislang nicht genutzte Optimierungspotentiale erschlossen werden. Die folgende Liste enthält einige Ansatzpunkte für eine Recherche im Bereich der Pointer- und Alias-Analyse.

- Burke, M., Carini, P.R., Choi, J.-D., Hind, M.
  Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers
  In: Gelertner, D., Nicolau, A., Padua, D., *Lecture Notes in Computer Science*, 892, Springer-Verlag, 1995.

- Burke, M., Carini, P.R., Choi, J.-D.
  Interprocedural Pointer Alias Analysis
  IBM Research Report, RC 21055, T.J. Watson Research Center, New York, 1997.

- Cooper, K.D., Kennedy, K.
  Fast Interprocedural Alias Analysis
  Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, POPL 1989, Austin, Texas, Jan. 1989.

- Diwan, A., McKinley, K., Moss, J.E.B.
  Type-Based Alias Analysis
  ACM SIGPLAN Notices, 33(5):106-117, May 1998.

- Emami, M., Ghiya, R., Hendren, L.J.
  Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers
  PLDI, pp. 242-256, ACM, 1994.

- Ghiya, R.
  Practical Techniques for Interprocedural Heap Analysis
  Master Thesis, School of Computer Science, McGill University, Montreal, 1996.

- Hind, M., Pioli, A.
  An Empirical Comparison of Interprocedural Pointer Alias Analyses
  IBM Research Report, RC 21058, T.J. Watson Research Center, New York, 1997.

- Hind, M., Pioli, A.
  Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses
  IBM Research Report, RC 21251, T.J. Watson Research Center, New York, 1998.

- Horwitz, S., Pfeiffer, P., Reps, T.
  Dependence Analysis for Pointer Variables
  Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, pp. 28-40, ACM Press, 1989.

- Landi, W.
  Interprocedural Aliasing in the Presence of Pointers
  Technical report lcsr-tr-174, Rutgers University Laboratory for Computer Science Research, 1991.

- Lu, J.
  Interprocedural Pointer Analysis for C
  Ph.D. thesis, Department of Computer Science, Rice University, Houston, Texas, 1998.

- Mayer, H.G., Wolfe, M.
  InterProcedural Alias Analysis: Implementation and Empirical Results
  Software - Practice and Experience, Vol. 23(11), p. 1201-1233, Nov. 1993.

- Reinig, A.G.
  Alias Analysis in the DEC C and DIGITAL C++ Compilers
  Digital Technical Journal, Vol. 10(1), p. 48-57, 1998.

- Steensgard, B.
  Points-to Analysis in Almost Linear Time
  POPL'96, pp. 32-41, ACM Press, 1996.

- Wilson, R.P.
  Efficient Context-Sensitive Pointer Analysis for C Programs
  Ph.D. thesis, Stanford University, Computer Systems Laboratory, December 1997.

Index