Diplomarbeit

Adreßzuweisung für den M3-DSP

David Kottmann

Diplomarbeit
des Fachbereichs Informatik
der Universität Dortmund

13. April 2000

Betreuer:

Dipl.-Inform. Markus Lorenz
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

Kapitel 1

Einleitung

Jeden Tag begegnen uns *eingebettete Systeme* in allen Bereichen des Lebens. Eingebettete Systeme sind Computersysteme, deren Hard- und Software speziell für die Lösung eines Problems entwickelt worden sind. Die Anzahl dieser Systeme nimmt immer mehr zu. So enthält jedes moderne Auto bereits Dutzende von eingebetteten Systemen, zum Beispiel für die Steuerung von ABS, Airbag oder der Zentralverrieglung. Die Anforderungen an diese Systeme sind sehr unterschiedlich: beim Airbag ist eine Echtzeitsteuerung wichtig, da der Airbag sofort im Falle eines Unfalls ausgelöst werden muß. In Mobiltelefonen, einem weiteren Einsatzgebiet eingebetteter Systeme, ist jedoch ein niedriger Stromverbrauch entscheidend, um die Laufzeit mit einer Akkuladung zu erhöhen.
Ein Thermostat stellt ein weiteres Beispiel für ein eingebettetes System dar. Die Umgebungstemperatur wird vom Thermostat gemessen, verarbeitet und an die Heizungssteuerung weitergegeben. Damit ist der Thermostat in das System Heizung eingebettet.
Im allgemeinen haben eingebettete Systeme keine interaktiven Ein- oder Ausgabeschnittstellen wie zum Beispiel eine Tastatur oder einen Bildschirm. Sie sind in ein größeres Gesamtsystem eingegliedert.
Es können unterschiedliche Komponenten für eingebettete Systeme verwendet werden: *ASICs*[1] sind speziell für die benötigte Anwendung entwickelt worden und erfüllen die Anforderungen effizient, die zum Beispiel in der Echtzeitumsetzung oder im minimalen Energieverbrauch liegen können.
Ein Nachteil der ASICs ist, daß Änderungen, die erst zum Ende der Entwicklung hin auf der Anforderungsseite auftreten, schlecht berücksichtigt werden können. ASICs werden für die zu Beginn spezifizierte Anwendung entwickelt, spätere Änderungen sind sehr zeitaufwendig und betreffen unter Umständen den kompletten Hardware-Entwurfsprozeß.
Neben ASICs existieren prozessorgesteuerte Lösungen, bei denen zum Beispiel *Mikrocontroller* oder *DSPs*[2] im eingebetteten System verwendet werden. Zwar wird auch dort eine besondere Architektur für die Anwendung ausgewählt, aber die Anpassung an die speziellen Anforderungen dieser Anwendung wird auf der Software-Seite durchgeführt.
Es findet eine Entwicklung hin zu einem prozessorgesteuerten eingebetteten System statt. Dies hat den Vorteil größerer Flexibilität, da bis zum Ende der Entwicklungszeit Änderungen an der Software schnell und einfach möglich sind. Die ausgewählte Hardware muß dazu

[1] Application Specific Integrated Circuit, anwendungsspezifischer Schaltkreis
[2] Digital Signal Processor, Digitaler Signalprozessor

nicht geändert werden. Weiterhin können solche eingebetteten Systeme wiederverwendet werden, da die Software neu programmiert werden kann. Damit können unterschiedliche Anwendungen auf demselben Prozessor ausgeführt werden, wenn sie sich bezüglich der Perfomance-Anforderungen nur geringfügig unterscheiden. Zur Anpassung softwaregesteuerter Prozessoren werden *DSP-Plattform-Lösungen* (wie zum Beispiel die M3-DSP Plattform) entwickelt, die den Entwurf von an die Anwendung angepaßten Prozessoren vereinfachen.

Es gibt mehrere Möglichkeiten, eine Anwendung auf den Zielprozessor abzubilden. Eine Möglichkeit besteht darin, ein Maschinenprogramm mittels Assembler-Programmierung manuell zu erzeugen. Dies führt in der Regel zu guten Ergebnissen, da der Programmierer die gegebene Hardware gut ausnutzen kann, ist aber insbesondere bei komplexeren Anwendungen sehr langwierig. Zudem ist die Fehleranfälligkeit des Programms sehr groß und eine Fehlersuche ist schwierig. Eine Portabilität auf andere Prozessoren ist nicht gegeben, da in einem solchen Fall das komplette Programm neu geschrieben werden müßte.
Eine andere Möglichkeit ist die Umsetzung einer Hochsprache (z. B. C oder C++) über einen Compiler in einen Maschinencode. Dies erfordert einen Compiler, der die speziellen Architekturmerkmale möglichst effektiv ausnutzen kann. Das Problem ist, daß heutige Compiler diese jedoch nur unzureichend ausnutzen. Deswegen besteht Bedarf an Compilern mit speziellen Optimierungs-Techniken.

Das Hauptaugenmerk der Arbeit liegt auf digitalen Signalprozessoren. Einige Besonderheiten zeichnen DSPs aus:
Der Trend geht heutzutage dahin, alle für den Prozessor notwendigen Einheiten auf einem Chip zu realisieren (System-on-chip, SOC). Gleichzeitig werden immer komplexere Anwendungen auf Prozessoren abgebildet, deren Programmspeicher nicht im gleichen Maße wie die Komplexität der Anforderungen wächst. Das erfordert eine möglichst optimale Ausnutzung des nur begrenzt vorhandenen Programmspeichers.
Viele DSPs besitzen eine *AGU*[3], um parallel zur Datenmanipulation eine Adreßberechnung durchführen zu können. Der Compiler sollte diese Parallelität ausnutzen können, um den vorhandenen Echtzeitanforderungen gerecht zu werden. Ein weiteres Merkmal sind spezielle Befehle, wie zum Beispiel der *MAC*[4], der in einem Taktzyklus eine Multiplikation und eine darauffolgende Addition durchführen kann. Viele DSPs bieten die Möglichkeit an, auf eine *saturating arithmetic* umschalten zu können. Da bei manchen Anwendungen bei DSPs Überschreitungen des gültigen Zahlenbereiches auftreten, ist es in diesen Fällen sinnvoll, einen Maximalwert auszugeben, zum Beispiel eine maximale Lautstärke oder maximale Helligkeit bei Monitoren. Oft haben DSPs zudem eine *heterogene* Registerstruktur, was bedeutet, daß es nicht möglich ist, jedes Register mit jedem anderen zu kombinieren. Dadurch sind die Möglichkeiten der Codegenerierung eingeschränkt, bei der Registerallokation stehen weniger mögliche Kombinationen zur Verfügung, Register zu kombinieren. Im Gegensatz dazu gibt es bei *orthogonalen* Registerstrukturen die Möglichkeit, alle Register miteinander zu kombinieren.
Der Compiler muß diese besonderen Hardware-Voraussetzungen kennen, um ausführbaren und effizienten Code zu generieren.

[3] Adress Generation Unit
[4] multiply-add-accumulate

Die Verwendung von *retargierbaren* Compilern führt bei speziellen Prozessorarchitekturen in der Regel zu ineffizientem Code. Dies beruht darauf, daß diese Compiler zwar an die jeweilige Architektur anpaßbar sind, aber keine architekturspezifischen Optimierungen enthalten. Deshalb werden eigene, an den Prozessor oder die Prozessorfamilie angepaßte Compiler entwickelt. Diese haben den Vorteil, die ihnen bekannte spezielle Architektur optimal ausnutzen zu können.

Damit eine DSP-Architektur die Anforderungen verschiedener Applikationen erfüllen kann, muß sie leicht anpaßbar sein. Dazu wurde in Dresden eine DSP-Plattform entwickelt, die *M3[5]-DSP Plattform*. Diese Plattform bietet eine Skalierbarkeit der Datenpfade sowie der Speicherbreite an. Außerdem kann ein an die Anwendung angepaßtes Verbindungsnetzwerk verwendet werden. Der *M3-DSP* ist ein Derivat dieser Plattform, ein eigens dafür entwickelter Compiler soll die software-seitigen Voraussetzungen für eine effiziente Umsetzung von Programmen für diesen Prozessor zur Verfügung stellen. Dieser Compiler soll die gegebenen Hardware-Voraussetzungen bei der Abbildung eines in C geschriebenen Programms in effizienten Maschinencode ausnutzen.
Wie aus den folgenden Kapiteln ersichtlich wird, stellt die *Adreßzuweisung* einen wichtigen Teilaspekt der Codegenerierung insbesondere für den M3-DSP dar und ist das Thema dieser Arbeit.

1.1 M3-DSP

Die M3-DSP Plattform wurde in Dresden entwickelt [32]. Um schnell einen anwendungsspezifischen Prozessor entwerfen zu können, muß eine Architektur leicht anpaßbar sein. Dazu gehört die Möglichkeit, eine größere Parallelität bei der Verarbeitung von Daten zu ermöglichen, zum Beispiel durch Erweiterung um neue Datenpfade oder Hinzufügen neuer Funktionseinheiten. Alle Prozessoren der M3-DSP Plattform besitzen ähnliche Architekturmerkmale, die Parametrisierbarkeit wichtiger Merkmale ermöglicht eine Anpassung an die auszuführenden Anwendungen.
Die M3-DSP Plattform folgt dem Prinzip der *feinkörnigen Parallelität* [32], also einer Parallelisierung innerhalb des Prozessors durch die Nutzung paralleler Datenpfade mit Hilfe einer SIMD-Architektur. Weiterhin besitzen alle Prozessoren dieser Plattform eine AGU und einen *VLIW-Cache*, in dem die von dieser Plattform benutzten *VLIWs*[6] gespeichert werden. Das VLIW dieser Plattform besteht unter anderem aus einem Datenmanipulationsteil, einem Datentransferteil und einem Adreßgenerierungsteil. Bei der M3-DSP Plattform werden die Änderungen des VLIWs dazu mit einem *TVLIW*[7] vorgenommen, nur das TVLIW wird im Programmcode vermerkt. Dadurch wird die Programmgröße reduziert.

Der M3-DSP ist ein Derivat dieser Plattform mit 16 Datenpfaden und einem 256 Bit breiten Speicher. Im VLIW-Cache werden vier Befehle abgespeichert.
Der M3-DSP soll in zellularen Telefonen (Handies) eingesetzt werden. Damit werden an

[5]Mobile Multimedia Modem
[6]Very Long Instruction Word
[7]Tagged Very Long Instrucion Word

den Prozessor Anforderungen bezüglich Low-Power und der Echtzeitausführung der Anwendung gestellt. Da der Programmspeicher begrenzt ist, spielt auch die Programmgröße eine wichtige Rolle.

Abbildung 1.1 zeigt die Architektur des M3-DSPs. Aus dem *Gruppenspeicher* kann der Inhalt einer ganzen über eine Adresse angesprochenen Speicherzeile in das Register M geladen werden. Jeder der 16 Werte, die mit nur einer Adresse im Speicher angesprochen werden, hat eine Breite von 16 Bit, so daß sich die Gesamtbreite von 256 Bit des Gruppenspeichers ergibt. Über ein dediziertes Verbindungsnetzwerk können Werte zwischen diesem Register M und den Eingangsregistern A, B, C und D der Datenpfade (Streifen) übertragen werden. Vorhandene Werte in Akkumulatoren können wieder in die Eingangsregister geschrieben werden.

Um den Echtzeitanforderungen zu genügen, besitzt der M3-DSP 16 Datenpfade, die parallel im **SIMD**[8]-Modus arbeiten können, d. h. alle 16 *MAC-ALUs*[9] führen dieselbe Operation aus. Da mit nur einem Befehlswort 16 *Datenpfade*[10] gesteuert werden, wird auch die Codegröße reduziert.
Kann diese Parallelität nicht ausreichend genutzt werden, besteht ferner die Möglichkeit, den Prozessor im **SISD**[11]-Modus zu betreiben. In diesem Fall erfolgt eine Abarbeitung nur im Datenpfad 0 (Abbildung 1.1). Zusätzlich zu den 16 Datenpfaden besitzt der M3-DSP einen Datenspeicher, der als Gruppenspeicher der Breite 256 Bit realisiert ist, wobei jede Zeile im Speicher aus 16 Werten der Länge 16 Bit besteht. Über eine Adresse wird eine Gruppe von 16 Werten angesprochen. Ein Lade- oder Speichervorgang betrifft also immer eine ganze Gruppe. Diese Gruppen werden in das Zwischenregister M geladen und können von dort aus an die Eingangsregister A, B, C, D der Datenpfade über ein Verbindungsnetzwerk weitergegeben werden. Das Verbindungsnetzwerk besitzt unter anderem zwei Betriebsarten[12]:

- *Vektordatentransfer, VDT*: Alle 16 Werte des Zwischenregisters werden parallel in eins der vier Eingangsregister der Datenpfade geladen, also beispielsweise alle Werte in die jeweiligen Eingangsregister A der 16 Datenpfade.

- *Element Data Transfer, ELDT*: Ein beliebiger Wert des Zwischenregisters kann in ein beliebiges Eingangsregister eines Datenpfades geladen werden.

Wenn der Prozessor im SIMD-Modus arbeitet, können so alle Werte, die nötig sind, um alle 16 Datenpfade simultan zu betreiben, parallel geladen werden, unter der Voraussetzung, daß diese Werte auch in der richtigen Gruppe stehen. Der M3-DSP besitzt eine AGU, mit der parallel zur Datenmanipulation auch eine Adreßberechnung durchgeführt werden kann.

[8]Single Instruction Multiple Data
[9]Arithmetical Logical Unit
[10]auch Funktionseinheiten
[11]Single Instruction Single Data
[12]Weitere Datentransfermodi sind zum Beispiel SWDT (Sliding Window Data Transfer) und SDT (Shuffle Data Transfer), auf die an dieser Stelle aber nicht weiter eingegangen werden soll.

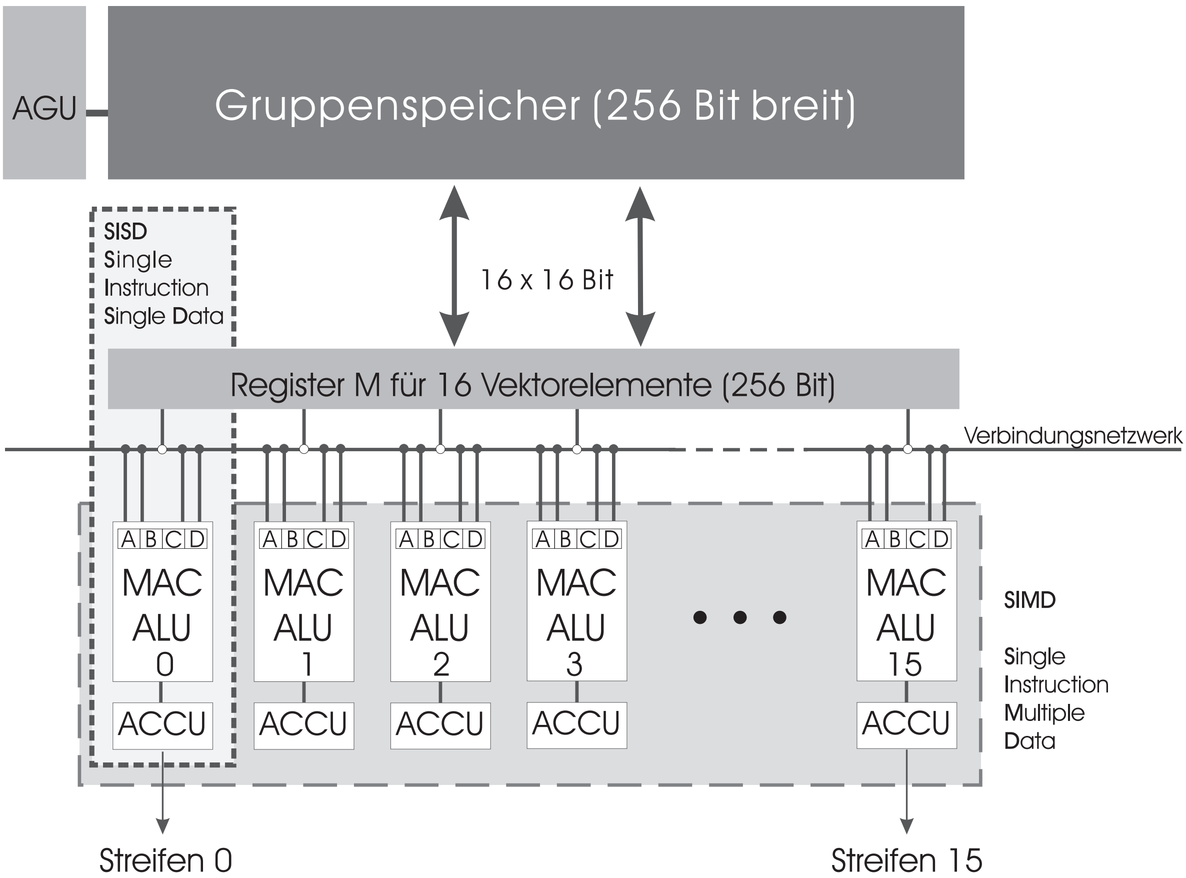

Abbildung 1.1: Architektur des M3-DSP

1.2 Problembeschreibung und Zielsetzung der Arbeit

Die Adreßzuweisung ist im Compilierprozeß ein Teil der Codegenerierung.
Der in Abbildung 1.2 vorgestellte Compilierprozeß startet mit dem Einlesen des *Sourceprogramms* durch das *Frontend*. Im Frontend wird dann überprüft, ob das Programm sowohl lexikalisch als auch semantisch korrekt ist. Die Ausgabe ist eine maschinenunabhängige *Zwischenrepräsentation* (ZR), auf der maschinenunabhängige Standard-Optimierungen ausgeführt werden können. Die *Codegenerierung* ist der erste vom Prozessor abhängige Teil des Compilierprozesses. Anhand des gegebenen Befehlssatzes werden die drei Phasen *Instruktionsauswahl*, *Registerallokation* und *Instruktionsscheduling* durchgeführt. Das Ergebnis ist Maschinencode. Die Aufgaben der Codegenerierung stellen sich im einzelnen wie folgt dar:
In der Instruktionsauswahl werden die Operationen, Ausdrücke und Anweisungen der Zwischenrepräsentation auf die spezifischen Befehle des Befehlssatzes (zum Beispiel den MAC-Befehl) abgebildet. In der Registerallokation werden den benötigten Variablen konkrete Lokationen (Speicher, Register) zugewiesen. Das Instruktionsscheduling ermittelt schließlich die Reihenfolge der ausgewählten Instruktionen unter Berücksichtigung der Lokationen der Variablen. Als Ausgabe entsteht in unserem Fall eine Variablenzugriffssequenz, in der (Speicher-)Zugriffe auf Variablen stehen. Diese Variablenzugriffssequenz ist die Grundlage der Adreßzuweisung, anhand derer ein geeignetes Speicherlayout erstellt wird. Das Speicherlayout wird dann wiederum als Grundlage für einen erneuten Durchlauf der ersten drei Phasen der Codegenerierung genommen. Dieser Vorgang kann bei Bedarf mehrmals durchlaufen werden.

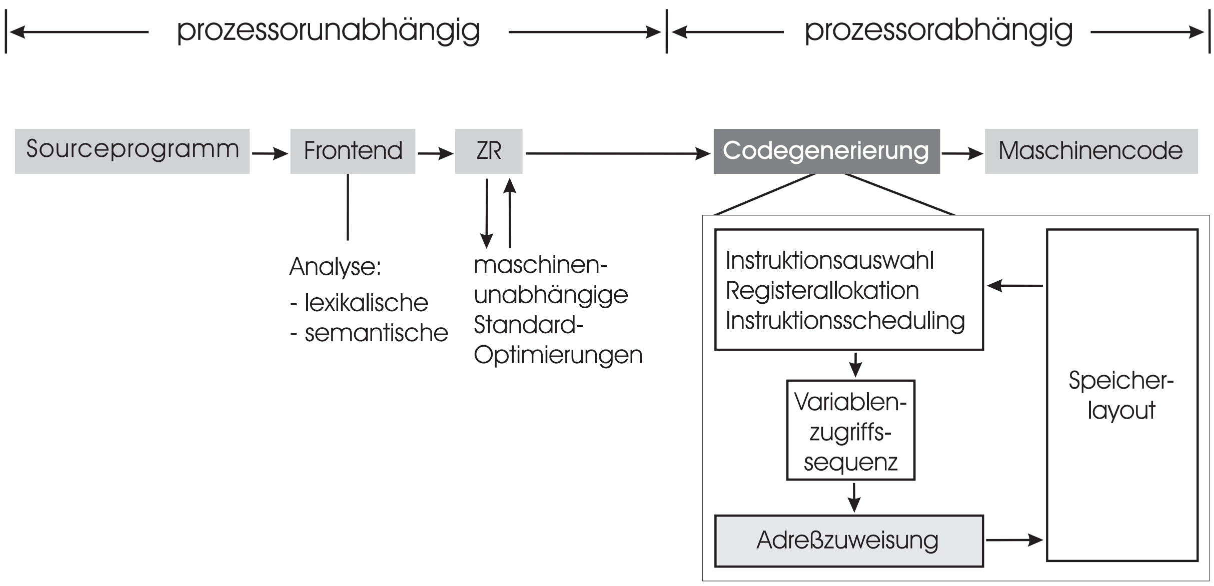

Abbildung 1.2: Einordnung der Adreßzuweisung in den Compilierprozeß

Bei den entstehenden Variablenzugriffen müssen lokale und globale Variablenzugriffssequenzen getrennt betrachtet werden. Dies ist erforderlich, da globale und lokale Variablen auch getrennt im Speicher abgelegt werden.

Abbildung 1.3 macht deutlich, wie diese unterschiedlichen Variablenzugriffssequenzen entstehen können: Ein Programm besteht aus mehreren Funktionen. Diese Funktionen wie-

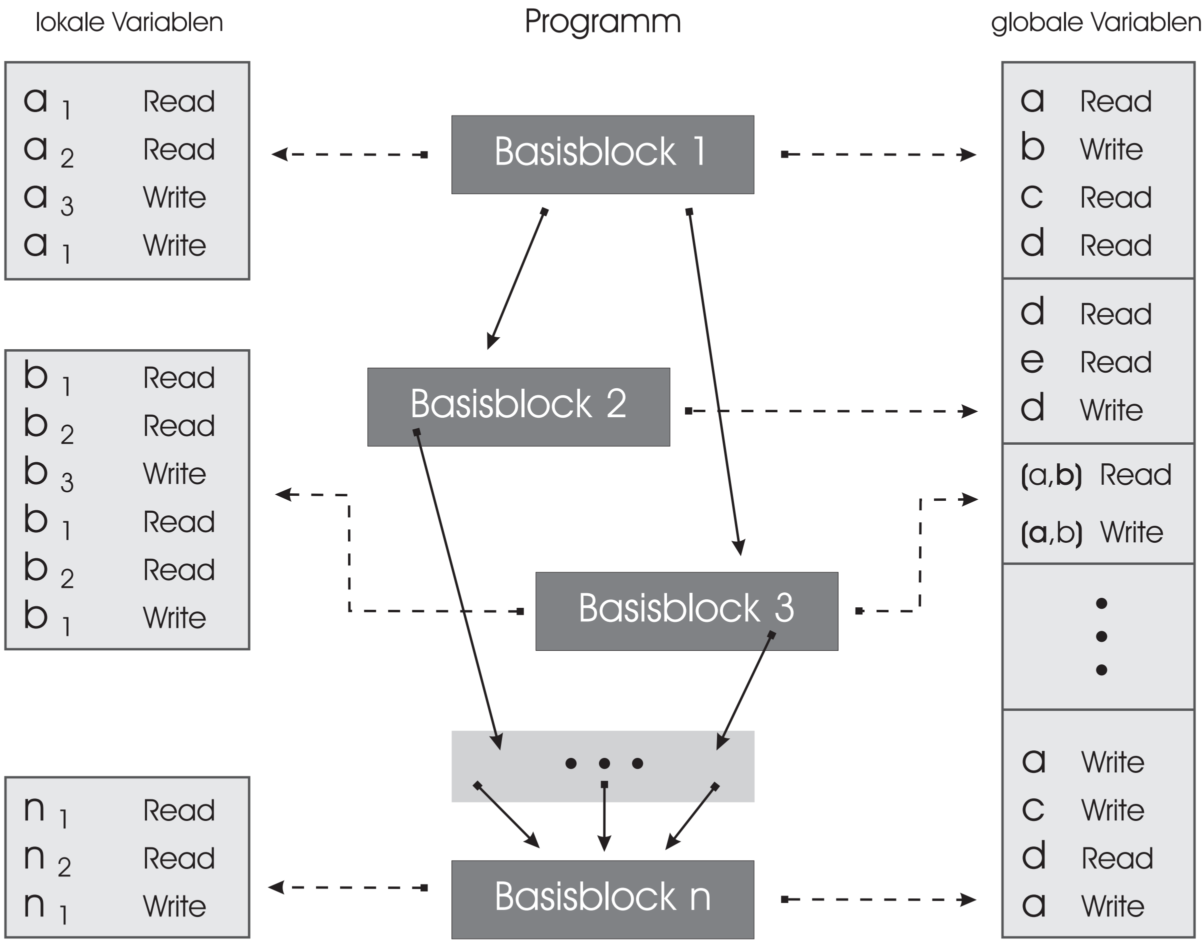

Abbildung 1.3: Globale und lokale Variablen eines in Basisblöcke unterteilten Programms

derum bestehen aus mehreren Basisblöcken. Eine einfache If-Then-Else Bedingung teilt den weiteren Ablauf einer Funktion schon in mehrere Basisblöcke, die später aber durchaus wieder zusammenfließen können. Für jeden Basisblock können eine lokale und eine globale Variablenzugriffsequenz ermittelt werden. Die **lokale Variablenzugriffssequenz** besteht ausschließlich aus (Speicher-)Zugriffen auf lokale Variablen, zum Beispiel aus *spills*[13]. Für diese ausgegebenen Variablen wird angenommen, daß der von den lokalen Variablen verwendete Speicher nach der letzten Verwendung der Variable wieder freigegeben werden kann. Anhand dieser Information können sich also mehrere lokale Variablen eine Speicherstelle teilen, wenn sie zu keiner Zeit gleichzeitig vorhanden sein müssen. Mit der gegebenen Sequenz kann ein Speicherlayout der lokalen Variablen ermittelt werden.
Die **globale Variablenzugriffssequenz** besteht ausschließlich aus Speicherzugriffen auf globale Variablen. Anders als bei lokalen Variablen müssen globale Variablen während des gesamten Programmablaufs im Speicher verfügbar sein, eine Mehrfachzuordnung dieser

[13]Dies sind Datentransporte von Variablen, die in eine andere Lokation, häufig den Speicher, gerettet werden müssen, um ein Überschreiben mit einem anderen Wert zu verhindern.

Variablen zur selben Speicherstelle wie bei lokalen Variablen ist nicht möglich.

Als Bewertungsgrundlage für die später vorgestellten Algorithmen werden die *Nachbarschaftsbeziehungen* der Variablen in einer Sequenz herangezogen. Zwei Variablen haben eine Nachbarschaftsbeziehung, wenn sie in einer Zugriffssequenz hintereinander vorkommen.

Das entwickelte Tool zur Adreßzuweisung kann zum Beispiel als Eingabe eine lokale oder eine globale Variablenzugriffssequenz erhalten. Da die lokalen Variablenzugriffssequenzen der Basisblöcke im allgemeinen keine direkten Beziehungen zueinander besitzen, kann das Tool für sie sequentiell gestartet werden. Soll das Tool auch bei globalen Variablen für jeden Basisblock in sequentiellen Schritten benutzt werden, muß beachtet werden, daß eine Zusammenfassung mehrerer Variablen zu einer Gruppe aus einem vorherigen Schritt bestehen bleiben muß. Dies muß allerdings von der Codegenerierung sichergestellt werden. Die Codegenerierung hat dazu die Möglichkeit, Constraints bezüglich des zu erstellenden Speicherlayouts in der eingegebenen Variablenzugriffssequenz anzugeben. Dazu können alle Zugriffe, die einzelne Variablen aus einer schon festgelegten Variablengruppe betreffen, als Zugriffe auf diese Variablengruppen angegeben werden. Ergibt sich zum Beispiel in Abbildung 1.3 nach der durchgeführten Adreßzuweisung für den ersten Basisblock, daß a und b in einer Gruppe stehen, sind auch die Zugriffe in allen anderen Basisblöcken Zugriffe auf beide Variablen. Im Beispiel werden im dritten Basisblock aus den Zugriffen $(a; Read)$ und $(b; Write)$ Zugriffe auf die ganze Gruppe $(a, b; Zugriff)$. Diese Zuordnung kann nicht mehr geändert werden, da jede Variable aus Konsistenzgründen nur einmal im Speicher abgelegt werden darf. Deswegen könnte es sinnvoll sein, bei einem solchen Einsatz des Tools zunächst den Basisblock zu wählen, der das größte Optimierungspotential verspricht oder im Lauf des Programms am Wichtigsten ist (dies betrifft zum Beispiel Anweisungen innerhalb von Schleifen).

Die Variablenzugriffssequenz stellt die Grundlage für die Berechnung eines geeigneten Speicherlayouts dar. Neben einer Reduzierung der Speicherzugriffe zur Energieeinsparung soll die Ausführungsgeschwindigkeit eines mit dem ermittelten Speicherlayouts compilierten Programms möglichst hoch sein. Zu diesem Zweck besitzt der M3-DSP die Möglichkeit, parallel zur Datenmanipulation Adreßberechnungen mit autoincrement und autodecrement durchzuführen, also eine parallele Veränderung einer beispielsweise in einem Adreßregister abgelegten Adresse um einen Offset, ohne zusätzliche Befehlsschritte zu verursachen. Ziel ist es, durch eine günstige Zuweisung der Variablen zu Adressen diese Parallelität möglichst gut auszunutzen.

Wie sich aus den bislang erwähnten speziellen Eigenschaften des M3-DSP ergibt, werden an die Codegenerierung besondere Anforderungen gestellt. Einen wichtigen Punkt stellt das Speicherlayout dar. Das Speicherlayout hat, wie im folgenden Beispiel (Abbildung 1.4) ersichtlich wird, einen wesentlichen Einfluß auf die Möglichkeit, effizienten Code zu generieren.

In diesem Beispiel ist eine Variablenzugriffssequenz gegeben, die Gruppengröße ist 2. Jeder Speicherzugriff auf eine Variable verursacht Kosten von 1, allerdings wird mit jedem Speicherzugriff eine ganze Gruppe geladen und ist bis zum nächsten nötigen Speicherzugriff in Registern vorhanden. Ein Speicherzugriff ist demnach nur nötig, wenn auf eine Variable in der Sequenz zugegriffen wird, die nicht in einem Register vorliegt. Wird nur die erste

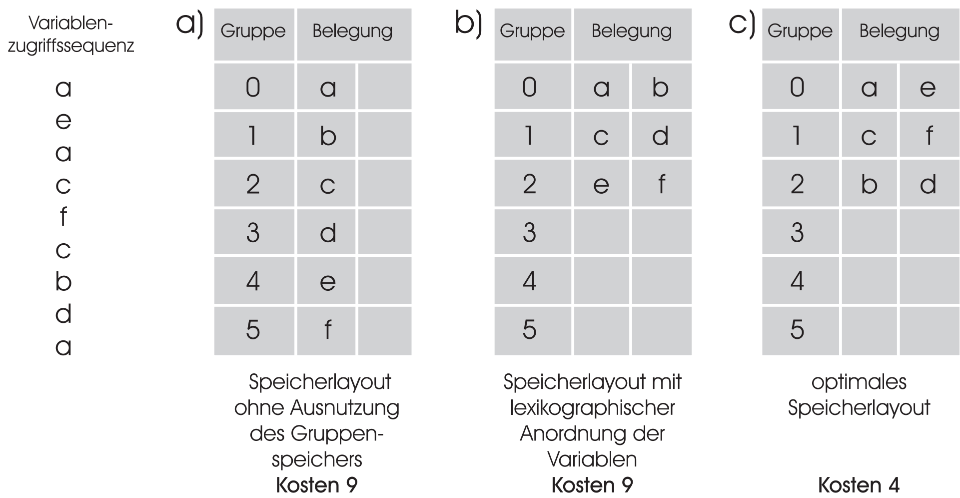

Abbildung 1.4: Einsparmöglichkeiten beim M3-DSP

Spalte jeder Speicherzeile benutzt, ergibt sich das Speicherlayout Abbildung 1.4a) mit den Kosten 9, da bei jeder Verwendung einer Variable erneut auf den Speicher zugegriffen werden muß, um die entsprechende Gruppe zu laden.
Eine naive Zusammenfassung zu Speichergruppen, beispielsweise über eine lexikographische Variablenanordnung, muß aber keine Reduzierung der Speicherzugriffe ergeben, wie Abbildung 1.4b) deutlich macht. Trotz einer Zusammenfassung zu drei Speichergruppen muß bei jedem Zugriff in der Variablenzugriffssequenz eine neue Gruppe geladen werden. Erst bei einer geschickten Anordnung wie in Abbildung 1.4c) sind erhebliche Einsparungen, in diesem Fall von fünf Speicherzugriffen, zu erzielen, so daß sich nur noch Kosten von vier Speicherzugriffen ergeben.
Eine weitere Möglichkeit Speicherzugriffe einzusparen besteht darin, Variablen mehrmals im Speicher abzulegen, da dann häufiger Variablen, die nacheinander in der Variablenzugriffssequenz auftauchen, zusammen geladen werden können. Allerdings würde dieser Gewinn durch die zusätzliche Konsistenzhaltung bei einem Speicherzugriff dieser Variablen wieder ausgeglichen. In diesem Fall müßten alle Speicherstellen, in denen die Variable abgelegt ist, geändert werden, was zu zusätzlichen Speicherzugriffen führen würde. Im vorliegenden Beispiel (Abbildung 1.5b) kommt a im Speicher doppelt vor. Dadurch wird ein Speicherzugriff im Verlauf der Variablenzugriffssequenz im Vergleich zu Abbildung 1.5a) eingespart, da die Nachbarschaftsbeziehungen von a und d ausgenutzt werden können. Allerdings müssen bei jeder Änderung von a beide Gruppen, in denen a vorkommt, geändert werden. Auch wenn a nur gelesen würde, müßte zu Beginn a zweimal im Speicher initialisiert werden. Das würde den späteren Gewinn in diesem Fall mit höheren Initialisierungskosten erkaufen.

Da die Zahl der Speicherzugriffe sinkt, werden auch die Energiekosten nachhaltig beeinflußt. Aufgrund des Gruppenspeichers verursacht ein Speicherzugriff beim M3-DSP erheblich mehr Energiekosten als andere Operationen, wie die Power-Estimation für den M3-DSP gezeigt hat (bis zu 30% mehr Energieverbrauch durch Speicherzugriffe im Vergleich zu anderen Operationen). Weil weniger Speicherzugriffe benötigt werden, um die

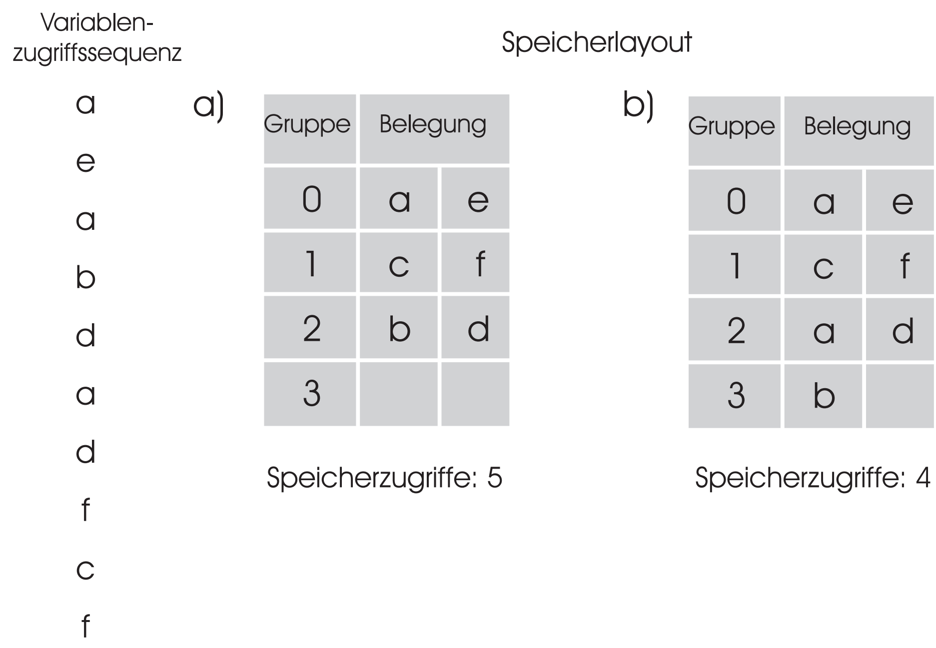

Abbildung 1.5: Mehrfaches Vorkommen von Variablen im Speicher

Variablensequenz abzuarbeiten, sinkt ebenfalls die Anzahl der Befehlsteile, die nötig sind, um Adressen für Speicherzugriffe zu berechnen. Damit sinkt die Programmgröße.
Das Problem der Adreßzuweisung kann in zwei Teilprobleme unterteilt werden, für die es im Rahmen dieser Diplomarbeit gilt, Lösungen zu finden (Abbildung 1.6):
Im ersten Schritt wird aus einer Menge von Variablen, die nur in *Einzelzugriffen*[14] in der Variablenzugriffssequenz[15] auftauchen, eine Menge von Gruppen gebildet, deren Größe der Breite des Gruppenspeichers entspricht - beim hier betrachteten M3-DSP also 16. Dies ist die **Gruppenbildung**. Das Ziel der Gruppenbildung besteht in der Reduzierung der Speicherzugriffe.
Die in der Variablenzugriffssequenz ebenfalls vorkommenden Zugriffe auf Variablen, die schon vom Compiler zusammengefaßt worden sind, oder Zugriffe auf Variablen, die in vorherigen Durchläufen des Tools schon zu Gruppen zusammengefaßt worden sind und als Constraints betrachtet werden, werden in der Gruppenbildung nicht weiter betrachtet, da diese Gruppen nicht verändert werden können. Diese *Variablengruppenzugriffe*[16] in der Sequenz werden nicht gelöscht. Würden diese Zugriffe nicht betrachtet, bekämen Zugriffe auf einzelne Variablen, zwischen denen ein Variablengruppenzugriff stattfindet, eine Nachbarschaftsbeziehung, obwohl sie nicht hintereinander aufgerufen werden. Die Gruppenbildung kann in Analogie zu einem Partitionierungsproblem betrachtet werden, da die Variablenmenge in Gruppen aufgeteilt werden muß. Die Lösung des Partitionierungsproblems ist NP-hart [13].

[14] Diese Variablen werden nur einzeln gelesen oder geschrieben, beim M3-DSP muß aber trotzdem die ganze Speicherzeile, in der diese Variablen stehen, geladen werden.

[15] Die in dieser und allen folgenden Abbildungen verwendeten Abkürzungen R und W stehen für Lese- (englisch: Read) und Speicherzugriffe (englisch: Write)

[16] Damit sind sowohl schon zusammengefaßte Variablen als auch Variablen, die schon vom Compiler zusammengefaßt worden sind, gemeint.

Im zweiten Schritt, der **Gruppenanordnung**, wird die Menge der vorhandenen Gruppen, bestehend aus den Variablengruppenzugriffen der Sequenz und den erzeugten Gruppen aus der Gruppenbildung, so im Speicher angeordnet, daß ein Speicherlayout entsteht, welches möglichst wenig *Adreßgenerierungswechsel* verursacht und damit wenig Programmcode benötigt. Ein Adreßgenerierungswechsel liegt vor, wenn zwei aufeinanderfolgende Befehle im Programmcode Adressen ansprechen, für die ein neues Teilbefehlswort ermittelt werden muß. Da der M3-DSP ein VLIW verwendet, werden Adreßgenerierungen parallel zum Rest des Befehls durchgeführt. Änderungen zu einem der vorherigen Befehle werden im Programmcode abgelegt. Muß also ein neues Teilbefehlswort für die Adreßgenerierung ermittelt werden, erhöht das die Codegröße. Die für die Gruppenanordnung zu lösenden Teilprobleme wie das SOA[17]-Problem und das GOA[18]-Problem sind schon NP-hart, so daß dies auch das Problem der Gruppenanordnung ist.

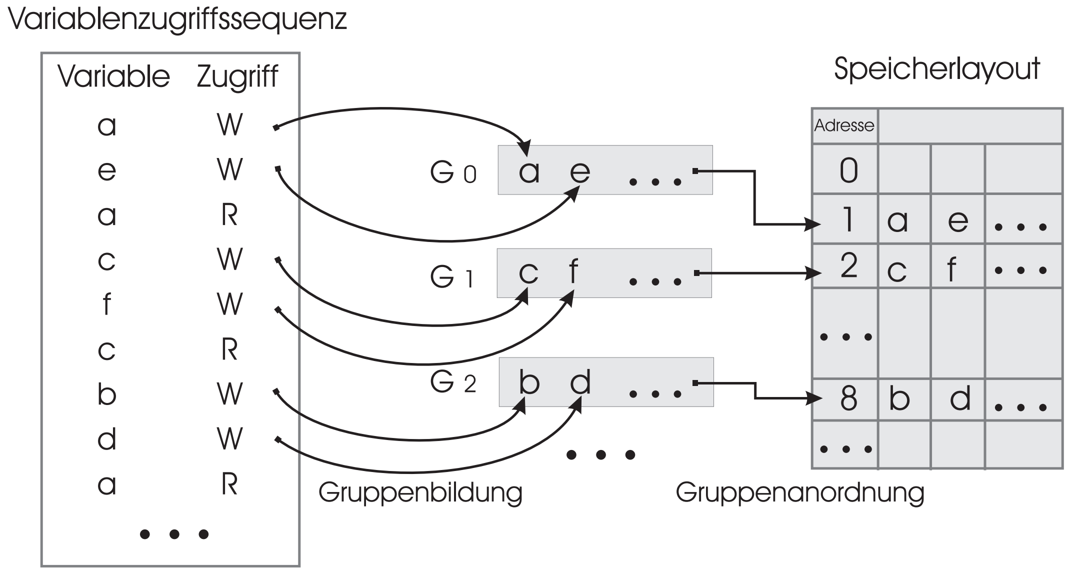

Abbildung 1.6: Adreßzuweisung beim M3-DSP

Diese beiden Teilprobleme können getrennt betrachtet werden, da nur geringe Abhängigkeiten zwischen den Problemen der Gruppenbildung und der Gruppenanordnung bestehen. Dies macht auch aufgrund der Komplexität beider Einzelprobleme Sinn. Eine Abhängigkeit beider Probleme besteht beispielsweise darin, daß nach der Gruppenbildung Variablengruppen eventuell aus zwei Teilgruppen bestehen, die voneinander völlig unabhängig sind, also keine Variable einer Teilgruppe eine Nachbarschaftsbeziehung zu einer Variable der anderen Teilgruppe hat. Diese Teilgruppen sind aber einer gemeinsamen Gruppe zugeteilt worden, weil sie zusammen die Gruppengröße nicht überschreiten. Dadurch wird eine völlig freie Anordnung unabhängiger Teilgruppen unterbunden, eine solche Zusammenfassung der Teilgruppen kann nicht mehr aufgehoben werden. Abbildung 1.7 verdeutlicht dies, die maximale Gruppengröße ist 5: Alle Variablen der vier vorkommenden Teilgruppen (a, b), (e, f, g), (c, d) und (h, i, j) haben in diesem Beispiel keine Nachbarschaftsbeziehungen zu einer Variable einer anderen Teilgruppe. Dadurch ist es in

[17] Simple Offset Assignemnt [2]
[18] General Offset Assignment [21]

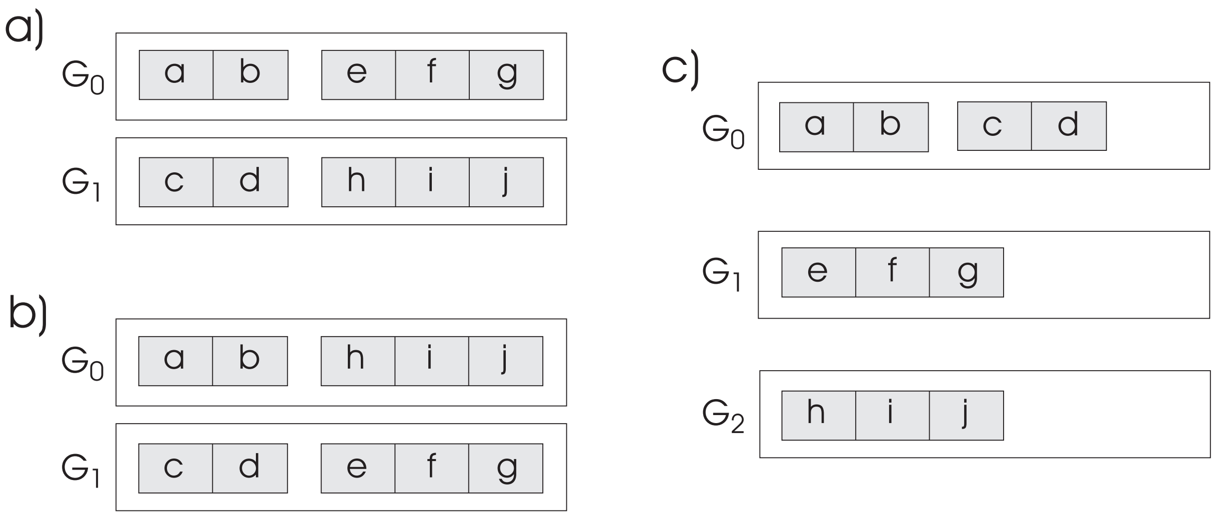

Abbildung 1.7: Möglichkeiten der Gruppenbildung

der Gruppenbildung unerheblich, ob sie wie in Fall a), b) oder c) Gruppen zugeordnet werden. Sogar eine Verteilung auf vier Gruppen wäre in der Gruppenbildung denkbar, ist aber aus Speicherplatzgründen nicht sinnvoll. In der Gruppenanordnung können diese verschiedenen Fälle jedoch Auswirkungen haben, da es dann wichtig ist, welchen Adressen Gruppen zugeordnet werden. Im Fall c) sind andere Adressen als in den Fällen a) oder b) nötig, damit kommt es später eventuell zu einer anderen Anzahl von Adreßgenerierungswechseln. Die Auswirkungen einer guten Zusammenfassung aller Variablen zu Gruppen in der Gruppenbildung auf die Gruppenanordnung sind im Vergleich zu den Auswirkungen einer anderen Anordnung von Teilgruppen zu Gruppen aber bedeutender.

Aufgrund der Komplexität werden beide Teilprobleme mit einem Genetischen Algorithmus in Verbindung mit heuristischen Verfahren gelöst. Heuristische Verfahren alleine finden häufig nur suboptimale Lösungen, wobei ihre Laufzeit in den meisten Fällen kurz ist. Die Laufzeit Genetischer Algorithmen ist polynomiell, aber in der Regel länger als die Laufzeit heuristischer Verfahren. Durch die parallele Suche im Lösungsraum wird oft nur eine geringfügig vom Optimum abweichende Lösung gefunden. Deswegen verspricht eine Kombination beider Ansätze eine gute Lösung. Eine optimale Lösung zu errechnen erfordert (zum Beispiel mit CLP, Constraint Logic Programming) sehr lange Laufzeiten und ist deswegen für größere Problemstellungen nicht praktikabel.

Um das Tool für alle DSPs der M3-DSP Plattform einsetzen zu können, ist es parametrisierbar. Im folgenden wird beispielhaft von einer Speicherbreite von 16 ausgegangen, was der Speicherbreite des M3-DSP entspricht.
Das Ziel dieser Diplomarbeit besteht in der Bereitstellung eines Tools, das auf der Basis einer gegebenen Variablenzugriffssequenz ein Speicherlayout erzeugt, das als Eingabe für die Phasen Instruktionsauswahl, Registerallokation und Instruktionsscheduling der Codegenerierung die Compilierung eines Programms ermöglicht, das mit diesem Speicherlayout energieeffizient arbeiten kann. Das Tool sollte für die wichtigsten Eigenschaften des M3-DSP parametrisierbar sein.

Darüberhinaus sollte das compilierte Programm möglichst wenig Programmspeicher benötigen. In dieser Arbeit liegt das Hauptaugenmerk bei der Ermittlung des Speicherlayouts auf skalaren Variablen, Arrays werden nicht betrachtet.

Diese Arbeit gliedert sich folgendermaßen:
Da zur Lösung dieser Probleme unter anderem ein Lösungsverfahren auf der Basis eines Genetischen Algorithmus vorgestellt wird, wird in Kapitel 2 eine allgemeine Einführung in Optimierungstechniken mit Genetischen Algorithmen gegeben. Daraufhin beschreibt Kapitel 3 das Problem der Gruppenbildung und stellt Lösungen mit Hilfe Genetischer Algorithmen und anderer Algorithmen vor. Kapitel 4 beschäftigt sich dann mit der Gruppenanordnung, ebenfalls unter Anwendung Genetischer Algorithmen.

Kapitel 2

Genetische Algorithmen

Viele technische Probleme werden heute in Analogie zur Natur gelöst. Es entwickelte sich der Gedanke, auch den Lösungsweg der Natur nachzuempfinden. Das Grundprinzip der Evolution, *survival of the fittest*, wird bei den *Evolutionären Algorithmen* nachgebildet. Diese können in zwei Bereiche aufgeteilt werden: *Evolutionsstrategien (ES)* und *Genetische Algorithmen (GAs)*.
Evolutionsstrategien werden hauptsächlich bei Problemen eingesetzt, bei denen es um die Optimierung reellwertiger Parameter geht. Einsatzgebiete von GAs sind Probleme mit booleschen oder ganzzahligen Parametern. Da es sich bei den Parametern der gegebenen Probleme um ganzzahlige Werte handelt, werden in dieser Arbeit GAs eingesetzt und im folgenden vorgestellt:
Genetische Algorithmen wurden von John Holland [11] entwickelt und seitdem in vielen Bereichen eingesetzt. GAs haben sich bei NP-harten Problemen bewährt. Inzwischen gibt es eine umfangreiche Literatur zu Genetischen Algorithmen.
Durch die probabilistische Suche kann eine optimale Lösung meist nicht garantiert werden. Ein Genetischer Algorithmus wird häufig bei sehr großen Lösungsräumen eingesetzt, für die andere Algorithmen keine befriedigende Lösung in einer vertretbaren Zeit ermitteln. Genetische Algorithmen führen im allgemeinen keine erschöpfende Suche im kompletten Lösungsraum durch. Damit ist nicht bekannt, ob es sich bei der ermittelten besten Lösung um das globale Optimum handelt. Daher werden Genetische Algorithmen bei Problemen eingesetzt, bei denen das Ergebnis nicht optimal sein muß. Die Rechenzeit Genetischer Algorithmen ist mit der für ein Problem entwickelten heuristischer Suchverfahren im allgemeinen nicht zu vergleichen, GAs sind langsamer, da sie in mehreren Generationen (Iterationen) parallel den gesamten Lösungsraum durchsuchen. Heuristische Verfahren suchen gezielter nach einem globalen Optimum, finden aber aufgrund unzureichender Fähigkeiten, lokale Optima zu überwinden, häufig nur lokale Optima. Durch die gezielte Suche benötigen sie eine geringere Rechenzeit. Bei großen Lösungsräumen, für die es keinen Algorithmus gibt, der in vertretbarer Rechenzeit ein für die Anforderungen ausreichendes Ergebnis erzielt, sind GAs jedoch inzwischen ein Standardverfahren.

Für einen Genetischen Algorithmus, der zur Lösung eines Problems eingesetzt werden soll, muß zunächst eine geeignete Kodierung des zu lösenden Problems gefunden werden. Das Problem wird damit in eine Form gebracht, mit der der GA arbeiten kann.

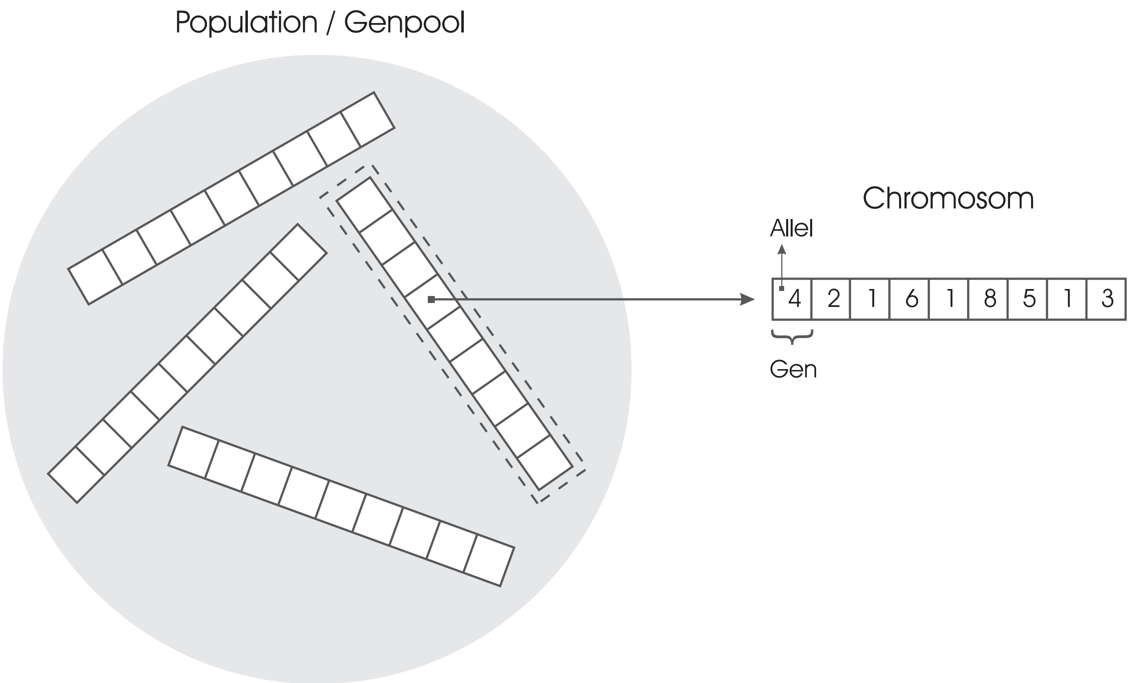

Abbildung 2.1: Aufbau der Population beim Genetischen Algorithmus

Abbildung 2.1 zeigt den Aufbau einer *Population*[1] des Genetischen Algorithmus. Die Population besteht aus einzelnen *Chromosomen*[2], die aus einzelnen *Genen* zusammengesetzt sind. Jedes dieser Chromosomen stellt eine Lösung des Problems dar, dabei haben die Gene eine genau definierte Bedeutung für das gegebene Problem. Der Wert eines Gens wird auch als *Allel* bezeichnet. Die Länge des Chromosoms ergibt sich aus der Zahl der benötigten Gene zur Darstellung einer Lösung. Im allgemeinen haben alle Chromosomen die gleiche Länge, da dies die genetischen Operatoren *Crossover* und *Mutation* vereinfacht.

Abbildung 2.2 zeigt den allgemeinen Ablauf eines Genetischen Algorithmus. Zu Beginn des Algorithmus wird eine Initialpopulation ermittelt, meist über eine Zufallswahl der Allele innerhalb der erlaubten Grenzen. Jedes Chromosom stellt eine Lösung dar, die anhand einer Bewertungsfunktion nach der Initialisierung bewertet wird. Es wird nun überprüft, ob die Abbruchbedingung erfüllt ist. Meist ist eine bestimmte Menge von Iterationen vorgegeben, bevor der Algorithmus stoppt. Es kann aber auch beispielsweise eine bestimmte Mindestgüte des besten Individuums oder der durchschnittlichen Güte aller Individuen der Population als Abbruchbedingung definiert werden. Die Güte eines Individuums ergibt sich aus der Bewertung. Ebenfalls dort wird die Fitneß eines Individuums bestimmt, die sich aus der Güte eines Individuums im Vergleich zur Güte aller anderen Individuen dieser Population ergibt.
Sollte die Abbruchbedingung noch nicht erfüllt worden sein, muß eine neue Generation, die Nachfolgegeneration, ermittelt werden. Diese besteht teilweise aus Chromosomen der Vorgängergeneration, die in der Selektionsphase ausgewählt werden. Eine bestimmte Menge von Chromosomen der Vorgängergeneration wird nicht in die Nachfolgegenerati-

[1] oder Genpool
[2] oder Individuen

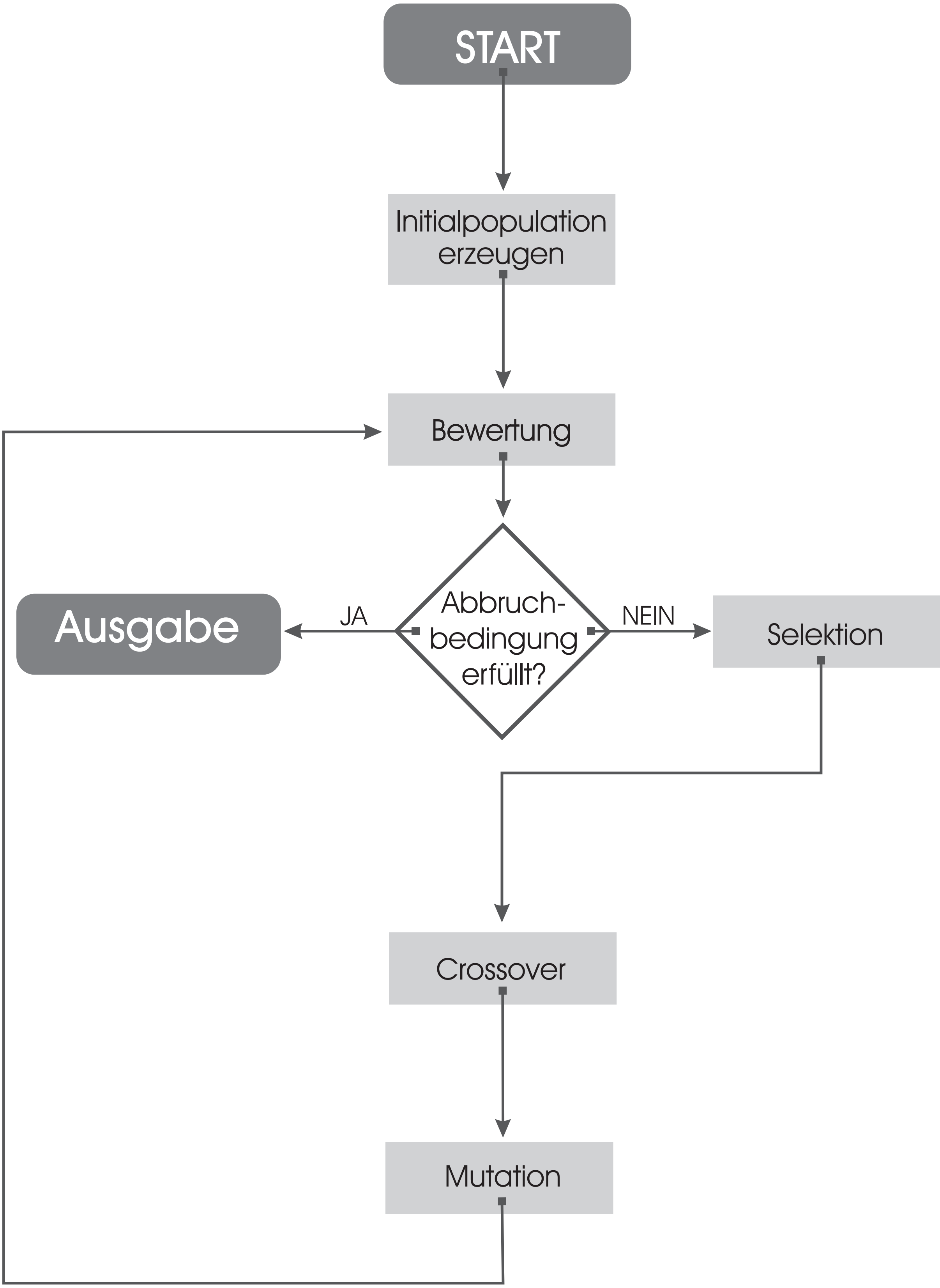

Abbildung 2.2: Allgemeiner Ablauf eines Genetischen Algorithmus

on übernommen - meist aufgrund ihrer schlechteren Fitneß. Eine entsprechende Anzahl von Chromosomen der Nachfolgegeneration wird über das an die Selektionsphase anschließende Crossover gebildet. Dazu werden im Crossover jeweils zwei Chromosomen der Vorgängergeneration, die ihre Gene in die nächste Generation vererben sollen, anhand ihrer Fitneß selektiert. Diese Chromosomen werden *rekombiniert*, es werden mit Crossover also Nachkommen gebildet. Zuletzt wird auf der neuen Generation die Mutation durchgeführt. Die neue Iteration muß mit einer Bewertung der neugebildeten und veränderten Chromosomen beginnen.
In den folgenden Teilabschnitten wird von der Kodierung des Genetischen Algorithmus ausgehend auf die Bewertungsfunktion und Selektion sowie die genetischen Operatoren Crossover und Mutation näher eingegangen.

2.1 Kodierung

Die Kodierung bringt eine Lösung des gegebenen Problems in eine Form, mit der der Genetische Algorithmus arbeiten kann. In der Kodierung muß entschieden werden, wie lang die Chromosomen sind und womit die Gene dieser Chromosomen belegt werden.
Ein Chromosom besteht aus mehreren unteilbaren Genen. Jedes dieser Gene enthält eine kodierungsabhängige Information, das Allel. Die gebräuchlichste Kodierung ist die **Binärkodierung**, also eine erlaubte Belegung jedes Gens nur mit den Allelen 1 oder 0. Dies ist historisch bedingt, da der erste GA binär kodiert war [11].
Um Probleme direkt umzusetzen sind allerdings **Zahlenkodierungen** besser geeignet. Lösungen für Probleme sind selten direkt mit einer Folge aus den Ziffern 0 und 1 darstellbar. In diesen Fällen müssen einzelne Werte, die in einer Lösung auftauchen, auf mehrere Gene verteilt werden. Dies erschwert die folgenden Operationen Crossover und Mutation, da die Gene nicht mehr voneinander unabhängig sind.
Um dem zu begegnen, können sogenannte *Hot Spots* an bestimmten Genen angebracht werden, um sie für eine mögliche Rekombination als Sollbruchstellen zu markieren. Damit werden aber die Rekombinationsmöglichkeiten des GAs stark beschnitten.

Weiterhin besitzen Zahlen- oder Zeichenkodierungen gegenüber der Binärkodierung den Vorteil, daß Chromosomen bei gleicher Länge mehr Informationen speichern können.
Es ergibt sich mit der Wahl der Kodierung auch die Zahl der möglichen Lösungen: Ein Chromosom der Länge 6 kann binär 64 verschiedene Lösungen beinhalten, mit der Kodierung aller Ziffern 0 bis 9 jedoch 1.000.000 verschiedene Lösungen. Soll die gleiche Anzahl von möglichen Lösungen auch binär vorhanden sein, muß das binärkodierte Chromosom erheblich länger sein. Ein Genetischer Algorithmus mit einer großen Chromosomenlänge hat jedoch einen größeren Speicherbedarf und in der Regel eine größere Laufzeit als ein Genetischer Algorithmus mit kurzen Chromosomen. Deswegen kann es sinnvoll sein, Zahlenkodierungen zu verwenden.

2.2 Bewertungsfunktion

Die Bewertungsfunktion ist in jeder neuen Generation der erste Schritt. In der Bewertungsfunktion werden zwei Werte jedes Chromosoms ermittelt: Die *Güte*[3] und die *Fitneß* dieses Chromosoms. Die Güte eines Chromosoms ist ausschließlich von den Genen dieses Chromosoms abhängig.
Die Fitneß ist wiederum von der Güte abhängig. Die Fitneß eines Individuums gibt seine Güte im Vergleich zu allen anderen Chromosomen dieser Population an, stellt also diese zum Rest der Population ins Verhältnis. Die Fitneß kann unter Umständen auch durch Simulationen oder praktische Versuche ermittelt werden [22].

2.3 Selektion

Beim Genetischen Algorithmus müssen mehrmals Individuen anhand ihrer Fitneß ausgewählt werden. Dies erledigt eine Selektion. Die Selektion von Individuen tritt in zwei Fällen auf:
Bei der Übernahme von Individuen in die nächste Generation wird eine Selektion durchgeführt. Dazu werden *Ersetzungsschemata* angewandt.
Neue Individuen, die den Platz der nicht übernommenen Individuen einnehmen sollen, werden mittels Crossover aus den Vorgängerchromosomen gebildet. Diese Chromosomen müssen ebenfalls selektiert werden, dafür werden die *Heiratsschemata* verwendet.
Es gibt verschiedene Selektionsverfahren für beide Aufgaben. Als Grundlage für die Selektion eines Individuums können sein Fitneßwert und seine Position in einer Fitneßtabelle (Rang) aller Individuen dieser Generation verwendet werden. Die meisten Verfahren nutzen nur einen dieser Werte als Grundlage für eine Selektion aus.

Der *Rang* eines Individuums ist seine Position in einer Tabelle absteigender Fitneß aller Individuen. Die Unterschiede des Fitneßwertes zwischen zwei benachbarten Rängen werden dabei nicht betrachtet und können über die gesamte Tabelle gesehen stark differieren.

Der *Selektionsdruck* gibt an, wie hoch die Wahrscheinlichkeit ist, daß nur gute[4] Individuen ausgewählt werden und ihre Gene in die Nachfolgegeneration vererben dürfen. Je höher der Selektionsdruck ist, desto schneller konvergiert der Genetische Algorithmus, da nur die besten Individuen in der Population verbleiben und sich fortpflanzen können. Die Konvergenz kann aber auch vorzeitig in einem lokalen Optimum enden, da nicht der komplette Lösungsraum betrachtet wird.

Es gibt viele unterschiedliche Selektionsmöglichkeiten, einige der bekannteren werden hier vorgestellt. Die Elite-Selektion ist ein Ersetzungsschema, Roulette-Wheel-Selektion und die Wettkampf-Selektion können sowohl als Ersetzungsschema als auch als Heiratsschema verwendet werden.

---

[3] oder Bewertung
[4] über Fitneß oder Rang ermittelt

2.3.1 Elite-Selektion

Bei den meisten Genetischen Algorithmen ist es üblich, das beste oder die n besten Individuen unverändert in die Nachfolgegeneration zu übernehmen. Das wird als Elite-Selektion oder Elitismus bezeichnet und geht auf einen Vorschlag von De Jong [12] zurück.
Damit ist gewährleistet, daß die beste bislang gefundene Lösung auf jeden Fall erhalten bleibt. Dies kann allerdings zu einer verfrühten Konvergenz führen, da alle Chromosomen der Population schnell ähnlich sind. Um diesem Problem zu begegnen, wurde der *Schwache Elitismus* [24] entwickelt, der das oder die n besten Individuen nur in mutierter Form in die Nachfolgegeneration übergibt. Hierbei taucht wiederum das Problem auf, eine gute Lösung im Laufe des Genetischen Algorithmus wieder zu verlieren. Dieses Problem stellt sich vor allem, wenn manche Gene eine überproportionale Bedeutung für die Fitneß besitzen. Werden diese Gene mutiert, kann die gute Lösung schnell verloren gehen.

2.3.2 Roulette-Wheel-Selektion

Diese Selektion ist eine *fitneßproportionale Selektion*. Sie kann mit einem Glücksrad verglichen werden: Jedes Individuum erhält auf dem Rad einen Abschnitt. Je größer die Fitneß eines Individuums ist, desto größer ist der zugehörige Abschnitt auf dem Glücksrad. Das Glücksrad wird zur Selektion jedes benötigten Individuums zufällig gedreht. Dabei ergibt sich das Problem, daß es theoretisch möglich ist, immer dasselbe Individuum zu ermitteln. Da sich auch schlechte Individuen fortpflanzen können, weil auch kleine Bereiche des Glücksrades ausgewählt werden können, hat dieses Verfahren einen niedrigen Selektionsdruck.

Um zu verhindern, daß immer gleiche Individuen ermittelt werden, stellte Baker [1] das Stochastic Universal Sampling (SUS) vor.
Das Glücksrad ist genauso aufgebaut wie oben, besitzt aber Zeiger, die im gleichen Abstand angebracht sind. Die Anzahl der Zeiger ergibt sich aus der Anzahl der zu selektierenden Individuen. Das Rad wird hier genau einmal gedreht. Die Individuen, auf die der entsprechende Zeiger zeigt, werden dabei selektiert. Damit kann genau die vorher bestimmte Anzahl an Individuen selektiert werden. Es wird der Nachteil des Standardverfahrens vermieden, da es nicht möglich ist, ein vergleichsweise schlechtes Individuum übermäßig häufig zu ermitteln, der Selektionsdruck wird erhöht.

2.3.3 Wettkampf-Selektion

Dieses Verfahren geht auf unpublizierte Überlegungen von Wetzel zurück. Aus einer Population werden zwei oder mehr Individuen zufällig gezogen und das beste Individuum selektiert. Bei der binären Wettkampf-Selektion, also jeweils einer Auswahl von zwei Elementen, ist der Selektionsdruck niedriger, da auch schlechtere Individuen eine höhere Selektionswahrscheinlichkeit haben. Die Wahrscheinlichkeit, ein schlechtes Individuum zu selektieren, sinkt, je mehr Individuen pro Wettkampf betrachtet werden. Damit steigt der Selektionsdruck.

2.4 Crossover

Ein Teil der Nachfolgegeneration wird durch Rekombination bestimmt. Der verwendete genetische Operator ist das Crossover. Dazu werden jeweils zwei Elternchromosomen selektiert, die über Crossover rekombiniert werden sollen. Dies geschieht, indem Gene zwischen diesen ausgetauscht werden und in die Nachfolgegeneration übertragen werden. Die Anwendung von Crossover ist abhängig von der angegebenen Crossoverwahrscheinlichkeit. Es gibt eine Vielzahl verschiedener Crossover-Varianten. Am häufigsten in der Literatur tauchen Uniform-Crossover, N-Punkt-Crossover und Shuffle-Crossover auf, es gibt jedoch noch zahlreiche weitere Varianten [22].

2.4.1 Uniform-Crossover

Für jedes Gen der Elternchromosomen wird beim Uniform-Crossover mittels einer vorgegebenen Wahrscheinlichkeit entschieden, ob ein Tausch dieses Gens mit dem Gen an der gleichen Position des anderen Elternchromosoms stattfindet. Wenn ja, werden die beiden Gene getauscht, wenn nicht, bleiben sie in ihrem jeweiligen Chromosom. Abbildung 2.3 zeigt einen solchen Vorgang. Die Chromosomen $1, 2, 4$ und 6 werden im ersten Schritt zum Tausch ausgewählt, dieser Tausch wird dann im zweiten Schritt vollzogen. Der Vorteil dieser Variante besteht darin, daß jedes Gen mit der gleichen Wahrscheinlichkeit ausgetauscht werden kann.

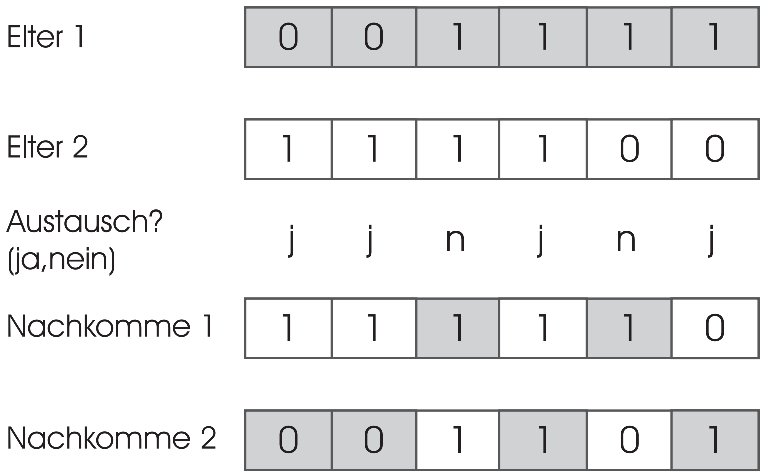

Abbildung 2.3: Uniform-Crossover

2.4.2 N-Punkt-Crossover

Zur Durchführung des N-Punkt-Crossovers werden N Kreuzungspunkte bestimmt. An diesen wird dann das Crossover durchgeführt. Abbildung 2.4 zeigt den Vorgang am Beispiel des 4-Punkt-Crossovers. Im ersten Schritt werden vier Punkte markiert und das Chromosom somit in fünf Abschnitte unterteilt. Am ersten Punkt werden alle dahinterliegenden Abschnitte des Chromosoms, hier die Abschnitte 2 bis 5, getauscht. Am zweiten

Punkt werden nun die Abschnitte 3 bis 5 getauscht, bei den folgenden analog. Nach dem Abschluß des vierten Tausches ergeben sich zwei Nachkommen wie in Abbildung 2.4 dargestellt. Häufig werden 1-Punkt-Crossover und 2-Punkt-Crossover verwendet. Je höher die Anzahl der Punkte ist, desto weniger Gene werden im Durchschnitt ausgetauscht [22]. Allerdings steigt mit der Zahl der Punkte wieder die Durchmischung der Gene, so daß wie beim Uniform-Crossover keine großen Gengruppen aus den Elternchromosomen an die Nachkommen übergeben werden. Zudem ist die Wahrscheinlichkeit für jedes Gen, ausgetauscht zu werden, unterschiedlich groß. Bei einem 1-Punkt-Crossover wird das letzte Gen beispielsweise auf jeden Fall getauscht, das erste jedoch gar nicht.

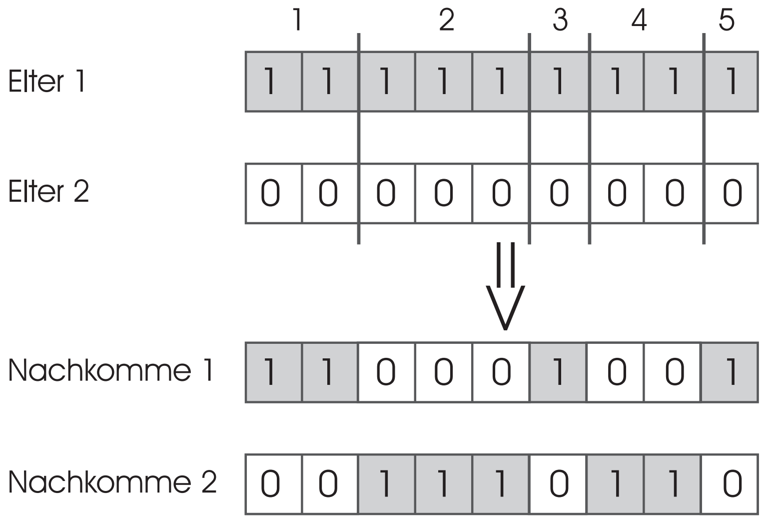

Abbildung 2.4: 4-Punkt Crossover

2.4.3 Shuffle-Crossover

Das Shuffle-Crossover kann nur in Verbindung mit einem anderen gewählten Crossover-Verfahren verwendet werden. Zunächst wird eine Numerierung der Gene benötigt. In Abbildung 2.5 ist das Verfahren dargestellt: Nachdem die Gene im ersten Schritt numeriert worden sind, werden sie im zweiten Schritt innerhalb des Chromosoms gemischt (shuffled). Beide Chromosomen werden dabei gleich gemischt. Im dritten Schritt muß nun das gewählte Crossover durchgeführt werden, in diesem Beispiel ein 1-Punkt Crossover. Anschließend werden die Gene im vierten Schritt wieder entmischt (unshuffled). Die Gene werden auch bei dieser Crossover-Variante unter Umständen so durchmischt, daß keine Gengruppen der Elternchromosomen erhalten bleiben. Die Menge der ausgetauschten Gene hängt von der verwendeten Variante im dritten Schritt ab.

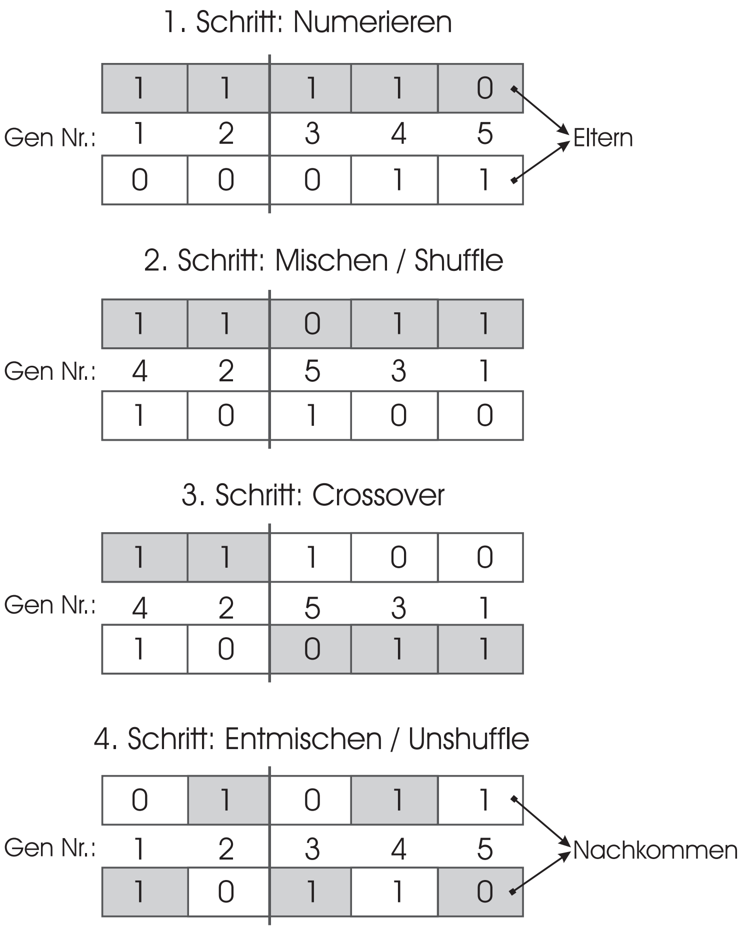

Abbildung 2.5: Shuffle-Crossover

2.5 Mutation

Bei der Mutation werden nur einzelne Gene von Chromosomen, abhängig von der Mutationswahrscheinlichkeit, verändert. Die Mutation wird zusätzlich zum Crossover durchgeführt, um eine vorzeitige Konvergenz zu verhindern: Durch eine ausschließliche Anwendung des Crossovers entwickelt sich die Population in eine bestimmte Richtung. Manche Allele an einer Genposition bei allen Chromosomen bekommen aber allmählich ein Übergewicht in der Population, obwohl sie vielleicht nur Teil einer suboptimalen Lösung sind und eine andere Belegung näher am Optimum liegen würde. Ein Gen, das in der ganzen Population nur mit genau einem Allel belegt ist, kann durch Crossover nicht mehr geändert werden. Da also durch eine reine Anwendung des Crossovers manche Bereiche des Lösungsraumes nicht erreicht werden können, gibt es eine weitere Operation bei GAs, die Mutation. Dadurch wird vermieden, daß der GA in einem lokalen Optimum vorzeitig konvergiert, da durch die Mutation Genmaterial, das im Optimierungsprozeß verlorengegangen ist, wiedergewonnen werden kann. Bei einer Binärkodierung bedeutet Mutation die einfache Negierung des Gens.
Die Mutationswahrscheinlichkeit wird im Vergleich zur Crossover-Wahrscheinlichkeit in der Regel sehr niedrig gewählt. Nur ein gutes Zusammenspiel von Crossover und Mutation liefern gute Ergebnisse, daher gibt es keine eindeutige Wahl einer bestimmten Crossover- oder Mutations-Variante.

Kapitel 3

Gruppenbildung

Der erste Schritt der Adreßzuweisung, die **Gruppenbildung**, besteht darin, aus einer gegebenen Variablenzugriffssequenz diejenigen Variablen, die nur als Einzelzugriffe in der Variablenzugriffssequenz vorkommen, zu Gruppen zusammenzufassen (Abbildung 3.1).

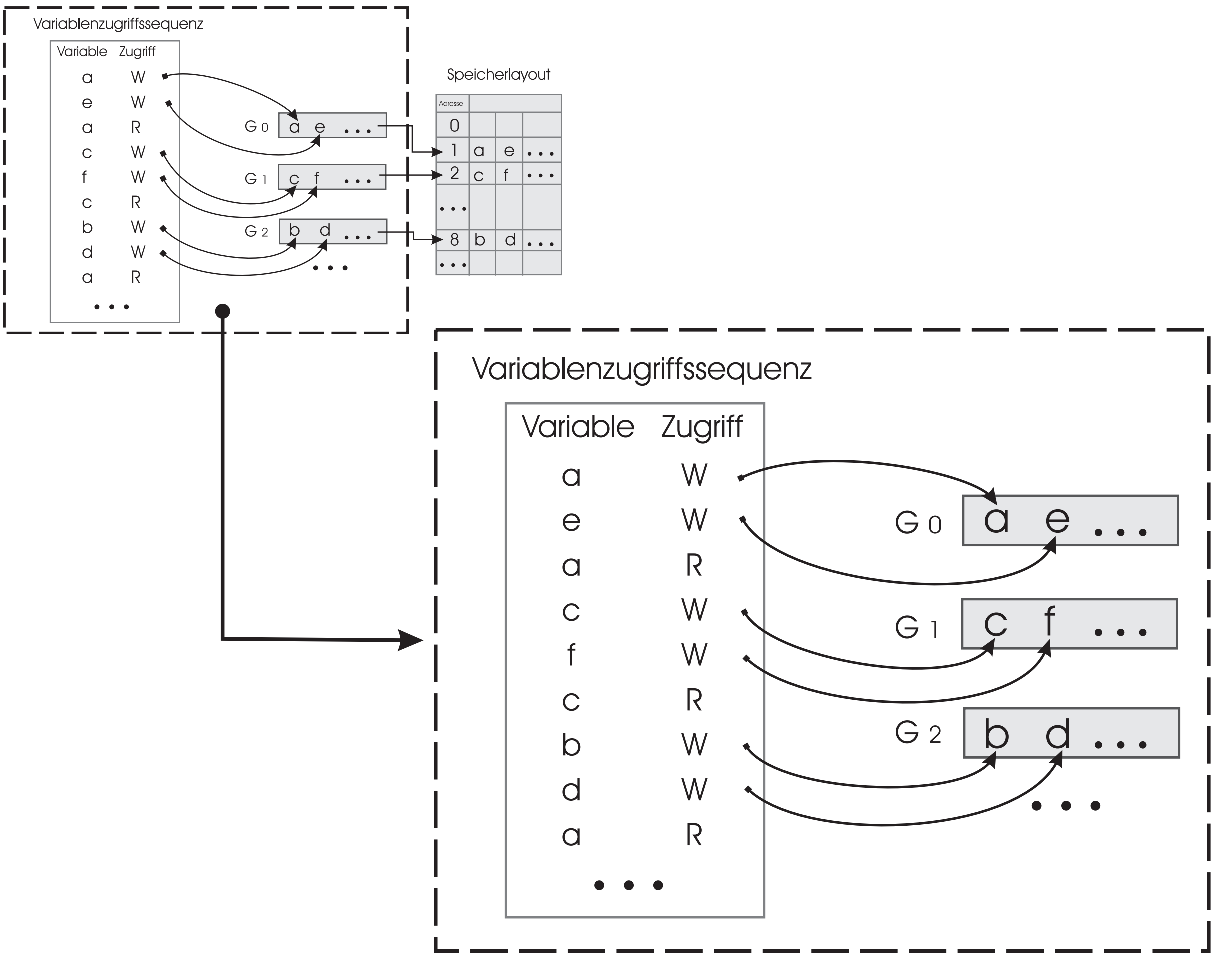

Abbildung 3.1: Gruppenbildung

Das Ziel der Gruppenbildung ist eine Reduzierung der Speicherzugriffe des zu compilierenden Programms.

Die Kriterien zur Zusammensetzung der Gruppen und zur Reduzierung der Speicherzugriffe ergeben sich aus den Nachbarschaftsbeziehungen der Variablen in der Variablenzugriffssequenz. In Abbildung 3.1 werden die Variablen a und e beispielsweise aufgrund ihrer zwei Nachbarschaftsbeziehungen der gleichen Gruppe zugeteilt (siehe gestrichelte Umrandung). Variablenzugriffe, die in der Variablenzugriffssequenz benachbart auftreten, verursachen bei einer Zuteilung zur selben Gruppe weniger Speicherzugriffe, da die zweite Variable durch den ersten Ladevorgang schon im Zwischenregister oder einem der Eingangsregister der Datenpfade vorliegt.

Die Gruppenbildung teilt die Menge von Variablen, die in der Variablenzugriffssequenz nur als Einzelzugriffe vorkommen, in Gruppen vorgegebener Größe (beim M3-DSP 16) auf. Die Gruppengröße wird durch die Breite des M3-DSP-Speichers bestimmt und ist im entwickelten Tool parametrisierbar. Offensichtlich besteht eine Äquivalenz des Problems der Gruppenbildung zu einem Partitionierungsproblem einer Menge in Teilmengen mit einer maximalen Anzahl von Elementen (beim M3-DSP 16).

Bei der Betrachtung des Gruppenbildungsproblems als Graphpartitionierung stellen die Knoten in diesem Graphen die Variablen dar. Kanten werden nur dann in den Graphen zwischen zwei Knoten eingefügt, wenn die entsprechenden Knoten mindestens eine Nachbarschaftsbeziehung besitzen. Die Kantengewichte entsprechen der Gesamtzahl an Nachbarschaftsbeziehungen zwischen den an der entsprechenden Kante liegenden Knoten. In dieser Analogie ist eine Lösung des Min-Cut-Problems auch eine Lösung des hier vorliegenden Problems einer Gruppenbildung zur Reduzierung der Speicherzugriffe, da das Gesamtgewicht der geschnittenen Kanten der Zahl der Speicherzugriffe entspricht, die es zu minimieren gilt.

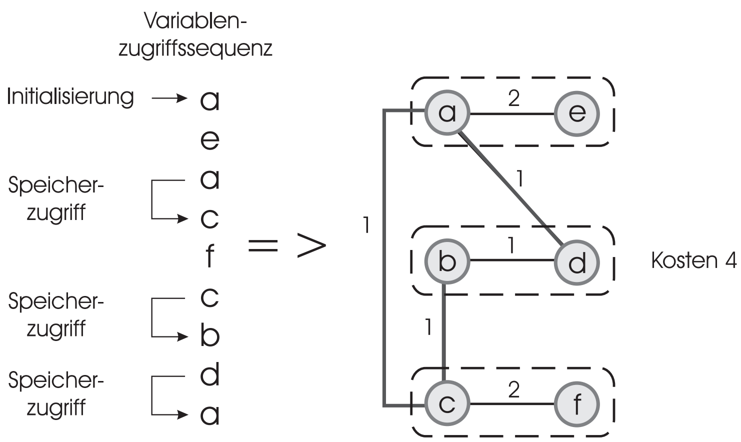

Abbildung 3.2: Variablenzuordnung als Partitionierungsproblem im Graphen

In Abbildung 3.2 ist ein Graph dargestellt, der sich aus der nebenstehenden Sequenz ergibt. In diesem Beispiel würde eine Gruppenzuteilung $a - e$, $b - d$ und $c - f$ bedeuten, daß die *externen*[1] Kanten eine der Summe ihrer Kantengewichte entsprechende Anzahl

[1]Dies sind Kanten, deren Knoten in verschiedenen Gruppen liegen.

an Speicherzugriffen verursachen; in diesem Fall also drei Speicherzugriffe. Hinzu kommen noch die Kosten für das Laden der Gruppe des ersten Knotens a, das auch einen Speicherzugriff verursacht. Damit ergeben sich Gesamtkosten von 4. Das Problem, eine optimale Gruppenaufteilung zu finden, ist NP-hart, da es auf ein Min-Cut Problem abgebildet werden kann [29]. Aus diesem Grund ist die Berechnung einer optimalen Lösung nur bedingt sinnvoll, da die Laufzeiten in Abhängigkeit von der Knotenzahl exponentiell steigen. Zur Lösung dieses Problems können heuristische Verfahren oder ein Genetischer Algorithmus herangezogen werden, da diese eine polynomielle Laufzeit besitzen, auch wenn die Lösungen beider Möglichkeiten nicht optimal sein müssen.

Grundlage der Gruppenbildung ist eine gegebene Variablenzugriffssequenz. Diese hat folgendes Aussehen:
Durch die Besonderheiten der M3-DSP-Architektur können in der Variablenzugriffssequenz sowohl Zugriffe auf einzelne Variablen als auch Variablengruppenzugriffe vorkommen. Die Menge der Variablen, auf die einzeln zugegriffen wird, muß ermittelt werden, denn dies sind die Variablen, die zu Gruppen zusammengefaßt werden sollen. In dieser Menge stehen aber unter Umständen Variablen, die auch in Variablengruppenzugriffen vorkommen. Diese Variablen müssen herausgefiltert werden, da ihre Speicherstellen schon feststehen, sie dürfen aus Konsistenzgründen nicht mehrmals im Speicher stehen. Die restliche Variablenmenge ist die Grundlage der Gruppenbildung.

Zur Durchführung der Gruppenbildung muß eine ermittelte Gruppenmenge bewertet werden. Je näher die in der Bewertung ermittelte Anzahl von Speicherzugriffen der in der Codegenerierung tatsächlich erforderlichen Anzahl kommt, desto besser ist diese Bewertung. Zu diesem Zweck ist es sinnvoll, noch zwei Besonderheiten der Variablenzugriffssequenz zu berücksichtigen. Dies sind schreibende Zugriffe und Schleifen innerhalb der Sequenz:

Zugriffe auf Variablen sind in der Zugriffssequenz als schreibend (W) oder lesend (R) markiert. In der Gruppenbildung werden aus Einzelzugriffen Zugriffe auf eine ganze Variablengruppe (Gruppenzugriffe), die auch in die Variablenzugriffssequenz anstelle der Einzelzugriffe geschrieben werden könnten. Der Zugriff auf diese Gruppen ergibt sich aus dem Zugriff auf die zugrunde liegenden Einzelzugriffe, auch Gruppenzugriffe sind schreibend oder lesend. Es treten zwei Besonderheiten auf:
War einer der Zugriffe in der Variablenzugriffssequenz vor einem Zugriff auf eine neue Gruppe schreibend, liegt die geschriebene Variable bislang nur in einem Register vor. Findet nun ein Zugriff auf eine neue Gruppe statt, die geladen werden soll, muß zunächst die aktuell in Registern befindliche Gruppe abgespeichert werden. Im vorliegenden Beispiel (Abbildung 3.3) werden im ersten Schritt mittels VDT die Variablen a und e aus Register M in die Eingangsregister A der Datenpfade geladen. Die Variable a wird in einem weiteren Schritt wieder im Eingangsregister A des Streifens 0 zwischengespeichert. Der folgende Zugriff auf c in der Variablenzugriffssequenz betrifft eine neue Gruppe, die gelesen werden muß. Zunächst jedoch muß die in den Eingangsregistern vorliegende Gruppe gespeichert werden, da a einen neuen Wert enthält und nochmals benutzt wird.
Ein weiterer Sonderfall liegt vor, wenn der erste und einzige Zugriff auf eine Gruppe schreibend ist. Auch hier muß ein zusätzlicher Zugriff erfolgen, da die Gruppe, in der die zu schreibende Variable liegt, erst geladen werden muß. Ansonsten werden die anderen Variablen dieser Gruppe mit falschen Werten aus den anderen Registern überschrieben. Diese besonderen Umstände müssen bei der Bewertung vom entwickelten Tool berück-

sichtigt werden, um die tatsächlich erforderliche Anzahl von Speicherzugriffen möglichst gut anzunähern.

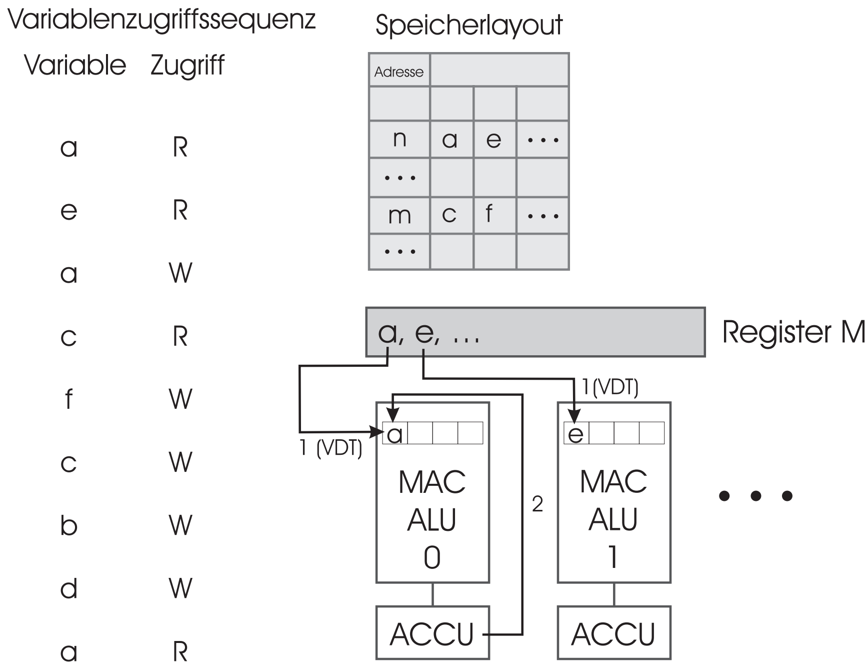

Abbildung 3.3: Zusätzliche Speicherzugriffe

Weiterhin soll das Tool **Schleifen** innerhalb der Variablenzugriffssequenz berücksichtigen können. Diese können innerhalb der Variablenzugriffssequenz markiert werden. Variablen innerhalb von Schleifen können dadurch bevorzugt einer Gruppe zugeteilt werden, da sie wahrscheinlich häufig gemeinsam benötigt werden.

Es kommen zwei Varianten von Schleifen vor:

Statische Schleifen:
Dies sind in diesem Zusammenhang Schleifen, deren Anzahl der Schleifendurchläufe s zur Compilierzeit schon feststeht. Die dadurch zusätzlich anfallenden Variablenzugriffe können exakt berechnet werden. Jede Variable dieser Schleife hat genau $2s$ Nachbarschaftsbeziehungen mehr. Mit Ausnahme der ersten und letzten Schleifenvariable haben alle Variablen s Nachbarschaftsbeziehungen zur jeweiligen Vorgängervariable und zum jeweiligen Nachfolger mehr.
Die erste Variable der Schleife hat zur letzten vor der Schleife liegenden Variable eine Nachbarschaftsbeziehung mehr, zur zweiten s Nachbarschaftsbeziehungen mehr.
Die letzte Variable innerhalb der Schleife hat zur ersten Variable zusätzlich $(s-1)$ Nachbarschaftsbeziehungen. In Abbildung 3.4 hat e zu a vier Nachbarschaftsbeziehungen durch den Rücksprung am Ende jedes Schleifendurchlaufs mehr. Die Schleife wird für die Berechnung der Speicherzugriffe betrachtet als wäre sie abgerollt.

Dynamische Schleifen:
Bei diesen Schleifen handelt es sich in diesem Zusammenhang um Schleifen, bei denen zur Compilierzeit nicht bekannt ist, wie oft sie durchlaufen werden. Mit einem für jede Schleife

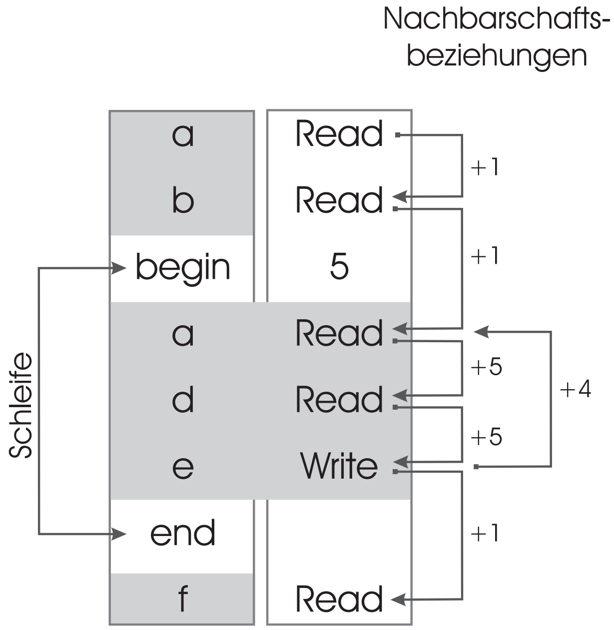

Abbildung 3.4: Nachbarschaftsbeziehungen bei Schleifen

in der Compilierung abgeschätzten Wert für die Anzahl der Durchläufe kann allerdings wie bei statischen Schleifen verfahren werden.

Bei einer Aufteilung der Variablen in Gruppen muß beachtet werden, daß es sinnvoll sein kann, Gruppen nicht vollständig zu füllen. Ein Beispiel dazu zeigt Abbildung 3.5: Das Speicherlayout a) ergibt sich aus einer vollständigen Füllung einer Speicherzeile, bevor die nächste Speicherzeile benutzt wird. Dabei werden aber die Knoten (d, e, f), die viele Nachbarschaftsbeziehungen haben, getrennt.
Ein anderes Speicherlayout ergibt sich in Fall b): Obwohl die Gruppen nicht vollständig gefüllt sind, nutzt dieses Layout mehr Nachbarschaftsbeziehungen aus und verursacht weniger Speicherzugriffe. Bei nur zwei Speicherzugriffen entstehen bei diesem Layout weniger als die Hälfte der bei Layout a) nötigen Speicherzugriffe.

Wie gezeigt, besteht ein direkter Zusammenhang zwischen den Nachbarschaftsbeziehungen der Variablen in der Variablenzugriffssequenz einerseits und den verursachten Speicherzugriffen für eine resultierende Partitionierung der Variablen andererseits. Die Ausgabe der Gruppenbildung ist eine Menge von Gruppen aus Variablen, die in der Variablenzugriffssequenz nur als Einzelzugriffe vorkommen. Da das der Gruppenbildung zugrunde liegende Partitionierungsproblem NP-hart ist [13], ist es sinnvoll, einige existierende heuristische Verfahren und Genetische Algorithmen zu betrachten, die für ähnliche Probleme entwickelt worden sind.

3.1 Partitionierungsverfahren

Partitionierungsverfahren werden dann eingesetzt, wenn es gilt, eine Menge in zwei oder mehr Teilmengen aufzuteilen. Die Aufteilung kann dann unter Vorgabe eines Optimie-

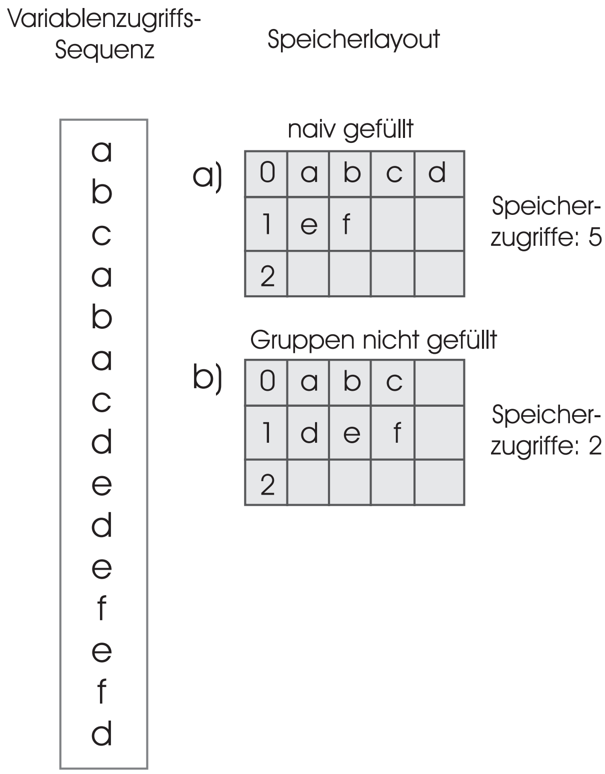

Abbildung 3.5: Vorteile unvollständiger Gruppenfüllung

rungskriteriums durchgeführt werden. Es gibt eine Reihe von Partitionierungsverfahren, die in *konstruktive* und *iterative* Verfahren [7] aufgeteilt werden können.

Konstruktive Verfahren erhalten als Eingabe eine Menge von Einzelelementen und eine maximale Mengengröße sowie die zu bildende Mengenzahl. Daraufhin werden entsprechend viele Teilmengen gebildet, die aus Einzelelementen bestehen und die solange vergrößert werden, bis alle Elemente zu genau einer Teilmenge gehören. Die Mengengröße kann auch genau festgelegt sein.

Iterative Verfahren starten mit der Gesamtmenge aller Elemente. Diese Menge soll in eine angegebene Zahl von Teilmengen aufgeteilt werden. Diese Partitionierung kann in einem Schritt in die gewünschte Anzahl von Teilmengen erfolgen oder in mehreren Schritten rekursiv erfolgen. Diese iterativen Verfahren können ebenfalls differenziert werden [5]:
2-Wege-Algorithmen werden häufig benutzt und teilen die Startmenge in zwei Teilmengen. Meist muß dieser Schritt so lange rekursiv wiederholt werden, bis die gewünschte Anzahl von Teilmengen ermittelt ist beziehungsweise eine bestimmte Anzahl von Elementen pro Menge nicht mehr überschritten wird.
Mehr-Wege-Algorithmen teilen die Startmenge direkt in eine bestimmte Anzahl von Teilmengen auf. Dabei kann die Anzahl der Elemente jeder Teilmenge exakt oder als obere Grenze angegeben werden. Diese Partitionierung kommt unter Umständen ohne Rekursion aus, wenn sofort im ersten Schritt die gewünschte Anzahl an Teilmengen erstellt wird.

Da das Gruppenbildungsproblem anhand eines Graphproblems dargestellt wird, werden in den folgenden Abschnitten einige allgemeine Graphpartitionierungsalgorithmen vorgestellt. Bei der Gruppenbildung wird zur Partitionierung ein Graph verwendet, in dem die Knoten die einzelnen Variablen repräsentieren und Kanten zwischen Variablen eingefügt werden, die eine Nachbarschaftsbeziehung in der Variablenzugriffssequenz besitzen. Jede Nachbarschaftsbeziehung wird mit +1 beim Kantengewicht bewertet, so daß vorhandene Kanten ein positives, ganzes Gewicht haben, das die Anzahl der Nachbarschaftsbeziehungen zwischen den Variablen, die diese Knoten repräsentieren, angibt.

3.1.1 Graphbasiertes Verfahren für duale Speicherbänke

Bei dem zum Gruppenbildungsproblem verwandten Problem einer Aufteilung von Variablen auf Speicherbänke sollen Variablen so auf zwei Speicherbänke aufgeteilt werden, daß ein Programm mit möglichst wenig Zyklen abgearbeitet werden kann. Ein laufendes Programm kann auf beide Speicherbänke parallel zugreifen. Durch eine Folge möglichst vieler Befehle, deren Variablen in verschiedenen Speicherbänken liegen und auf die dementsprechend parallel zugegriffen werden kann, wird Laufzeit eingespart[2]. Saghir, Chow und Lee [23] stellen ein graphbasiertes Verfahren zur Partitionierung einer Variablenmenge vor.
Das Ziel des Algorithmus besteht in der Partitionierung der Variablenmenge in zwei Teilmengen, so daß die Summe der *internen*[3] Kantengewichte beider Gruppen minimal und damit die Summe der externen Kanten maximal ist (Max-Cut).

Der Algorithmus initialisiert zwei Graphen A und B. Der Graph A ist zu Beginn der Gesamtgraph, Graph B ein leerer Graph. Die Knoten repräsentieren Variablen, die Kanten zwischen den Knoten zeigen an, daß diese Variablen gleichzeitig für einen Befehl als Eingangswerte benutzt werden. Das Kantengewicht zeigt dabei die Häufigkeit dieser Beziehungen an. Beziehungen innerhalb von Schleifen werden unabhängig von der Häufigkeit des Schleifendurchlaufs mit +2 beim Kantengewicht bewertet.
Es wird der Knoten aus dem Graphen A bestimmt, der die größte Summe aller Kantengewichte der über ihn laufenden internen Kanten besitzt. Dieser Knoten wird in den Graphen B bewegt. Daraufhin werden die Kosten überprüft, die sich aus der Summe der internen Kantengewichte beider Graphen A und B ergeben. Bei einer Reduktion der Summe der internen Kantengewichte wird die vorliegende Partitionierung gespeichert und das Verfahren fortgesetzt. Sollte eine weitere Verschiebung des Knotens mit der höchsten Summe der Kantengewichte aller über ihn fließenden internen Kanten aus dem Graphen A keine weitere Reduktion der internen Kantengewichte bringen, stoppt der Algorithmus und die letzte abgespeicherte Partitionierung wird ausgegeben. Von dieser Partitionierung aus gibt es keine Verschiebung, die die internen Kantengewichte minimiert. Ein Beispiel für eine solche Partitionierung zeigt Abbildung 3.6.

In Abbildung 3.6 wird mit dem Gesamtgraphen und einem leeren Graphen begonnen, die Kosten, also die Summe der internen Kantengewichte beider Graphen, belaufen sich auf 15. Der Knoten D hat den größten Summenwert aller Kantengewichte der Kanten, die über ihn fließen (Wert 8) und wird in den anderen Graphen bewegt. Daraufhin ergeben sich Kosten von 7. Nun hat der Knoten B den höchsten Wert (6) und wird bewegt. Der

[2] Eine genauere Betrachtung dualer Speicherbänke kann bei [23, 27] und den dortigen Referenzen nachgelesen werden.

[3] Die Knoten dieser Kanten gehören beide zu einer Teilmenge.

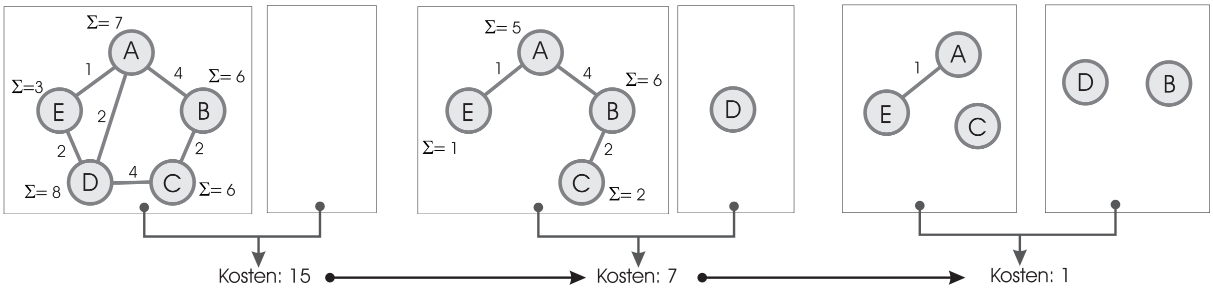

Abbildung 3.6: Algorithmus von Sahgir, Chow und Lee

resultierende Graph verursacht nur noch Kosten von 1, eine weitere Knotenbewegung bringt keine Kostenersparnis, der Algorithmus stoppt.

Als Ausgabe produziert der Algorithmus eine Partitionierung der Variablenmenge in zwei Teilmengen, die die Belegung der beiden Speicherbänke repräsentieren.
Der Algorithmus ist nach Aussage der Autoren schnell und liefert gute Ergebnisse [23]. Allerdings ist es möglich, daß die Ergebnisse nur suboptimal sind, da es sich um ein heuristisches Suchverfahren handelt.

3.1.2 Kernighan-Lin-Algorithmus

Der Kernighan-Lin-Algorithmus [13] aus dem Jahr 1969 ist der Vorläufer vieler heuristischer Graph-Partitionierungs-Algorithmen.
Ziel des Kernighan-Lin-Algorithmus ist es, aus einem gegebenen Graphen zwei Teilgraphen zu bilden, bei denen die Summe der Kantengewichte zwischen den beiden Teilgraphen minimal ist (Min-Cut). Da eine Aufteilung in zwei Teilgraphen stark unterschiedlicher Größe in der Regel eine kleine Menge geschnittener Kanten produziert, muß eine genaue oder maximale Teilgraphengröße beim Kernighan-Lin-Algorithmus vorgegeben werden. Im Extremfall wäre es sonst denkbar, daß als Partitionierung ein leerer Graph und der Gesamtgraph gebildet würden, was eine geschnittene Kantenmenge von 0 ergeben würde. Im allgemeinen soll ein Graph in zwei gleich große Teilgraphen aufgeteilt werden, bei einer ungeraden Elementzahl in zwei Teilgraphen, deren Größe sich um ein Element unterscheidet.

Initialisiert wird der Algorithmus mit einer (Zufalls-)Partitionierung des Startgraphen in zwei Teilgraphen. Abbildung 3.7 zeigt den Vorgang, der dem Algorithmus zugrunde liegt: Innerhalb der Teilgraphen A und B sollen die Knotengruppen A' und B' gebildet werden, die getauscht werden sollen. Die Bedingung für den späteren Tausch ist, daß die Summe der Kantengewichte zwischen den beiden Teilgraphen nach dem Tausch kleiner sein soll als vor dem Tausch. Dabei ist die Größe der beiden Gruppen A' und B' als λ vorgegeben, bei $\lambda = 1$ sind beispielsweise alle Knoten aus A mit allen Knoten aus B zu vergleichen.
Im vorliegenden Beispiel hat A' fünf externe Kanten, also Kanten, bei denen ein Knoten im anderen Teilgraphen B liegt, und zwei interne Kanten, bei denen beide Knoten zu A gehören. B' besitzt zwei interne und drei externe Kanten. Bei einem Tausch von A' und B' würde sich ein Gewinn von 2 ergeben. Dieser ergibt sich, da A' anschließend nur noch drei

externe Kanten besitzt und die Zahl der externen Kanten von B' sich nicht verändert hat. Dabei ist zu beachten, daß eine Kante zwischen A' und B' (in der Abbildung gestrichelt) immer eine externe Kante ist und bei einem Tausch beider Knotengruppen keinen Gewinn bringt. Aus Komplexitätsgründen ist wie in [13] beschrieben eine Wahl $\lambda = 1$ günstig.

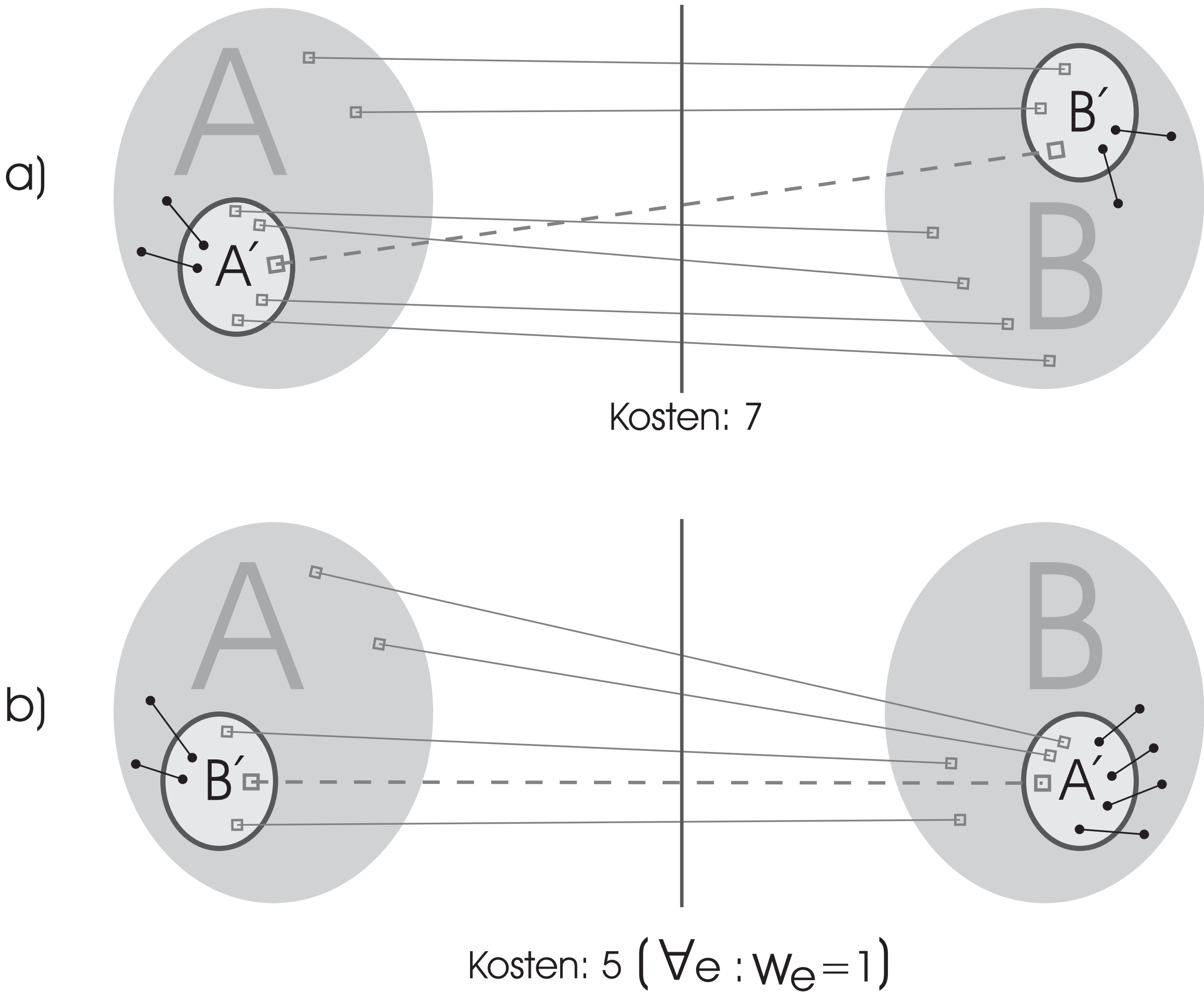

Abbildung 3.7: Kernighan-Lin-Algorithmus

Der Kernighan-Lin-Algorithmus läuft folgendermaßen ab:
Nach der Initialisierung werden alle möglichen Knotengruppen A', B' gebildet und anhand der Differenz zwischen externen und internen Kantengewichten sortiert. Die beiden Knotengruppen mit der höchsten Differenz werden getauscht. Anschließend werden die Knoten dieser Gruppen verankert und können nicht mehr bewegt werden. Die Kosten dieser Aufteilung, also die Summe der Kantengewichte der externen Kanten, werden gespeichert und alle Differenzen der verbleibenden Knotengruppen neu berechnet. Das wird solange durchgeführt, bis alle Knoten verankert sind. Nach Beendigung dieses Durchlaufs wird die Aufteilung mit den während dieses Durchlaufs geringsten Kosten wiederhergestellt.
Der Algorithmus kann mit der ermittelten Aufteilung erneut gestartet werden, solange eine Reduzierung der Summe der externen Kantengewichte in einem Durchlauf erzielt

wird. Die optimale Lösung dieses Partitionierungsproblems ist NP-hart [13].

Für das gegebene Problem der Gruppenbildung ist es aber erforderlich, die Knotenmenge in mehr als zwei Teilmengen aufzuteilen. Kernighan und Lin schlagen zwei Varianten vor, um eine Aufteilung in mehr als zwei Knotenmengen zu erreichen:

Variante 1 (hierarchisch):
Der Algorithmus wird mehrmals rekursiv angewendet. Um jeweils eine 2-Teilbarkeit zu erreichen, müssen von vornherein genug Platzhalterknoten hinzugefügt werden. Diese Knoten haben keine Kanten zu den anderen Knoten des Graphen und können nach Beendigung des Algorithmus wieder entfernt werden. Damit ist es auch möglich, ungleiche Gruppengrößen mit vorgegebenem Maximum zu errechnen.

Variante 2:
Zu Beginn wird eine Aufteilung in die k gewünschten Knotenmengen vorgenommen. Anschließend wird jede mögliche Kombination zweier Mengen einmal mit dem oben vorgestellten Algorithmus durchgeführt, bis dabei alle Knoten dieser beiden Mengen verankert worden sind. Sollte eine der dabei auftauchenden Zwischenstände eine Reduzierung der Summe der externen Kantengewichte ergeben, werden beide Mengen entsprechend verändert und als *frei* markiert. Sollte keine Reduzierung erreicht werden können, werden beide Mengen als *fest* markiert. Nach dem Durchlauf über alle Mengenkombinationen, in dem $O(k^2)$ Mengenvergleiche entstehen, wird überprüft, ob mindestens eine der Mengen frei markiert ist. Ist dies der Fall, wird ein neuer Durchlauf gestartet. Dies wird solange durchgeführt, bis alle Mengen nach einem Durchlauf mit fest markiert worden sind.

Bei beiden Varianten ist nicht genau klar, wie groß die Laufzeit ist, denn die Anzahl der Durchläufe hängt vom eingegebenen Graphen ab. Die Anzahl der Vergleiche pro Durchlauf ist allerdings quadratisch. In dieser Arbeit wurde zu Testzwecken ein Kernighan-Lin-Algorithmus mit $\lambda = 1$ implementiert. Die Aufteilung in Gruppen der Größe 16 wird schon in der Initialisierung durchgeführt, indem immer die nächsten 16 auftretenden Variablen einer Gruppe zugeordnet werden[4].
Platzhaltervariablen, deren Knoten im Kernighan-Lin-Algorithmus keine Kanten zu den anderen Knoten des Graphen haben, werden in ausreichender Zahl zu Beginn des Algorithmus hinzugefügt, so daß alle Gruppen die gleiche Größe haben. Kernighan und Lin hielten diesen Ansatz (Variante 2) in [13] für erfolgversprechender als Variante 1, da bei einer ungünstigen Teilung im ersten Schritt diese bis zum Ende fortbestehen würde und schon nach diesem ersten Schritt kein Optimum mehr erreicht werden könnte. Zudem wäre es nach ihrer Aussage nicht geschickt, zwei Knotenmengen zu bilden, die einen möglichst hohen Zusammenhangswert hätten, um sie dann in der nächsten Rekursionsstufe wieder zu trennen, wie das bei Variante 1 der Fall wäre. Wang, Lim, Cong und Sarrafzadeh stellten in [29] jedoch fest, daß ein Ansatz wie Variante 1 bessere Ergebnisse liefert (bei großen Graphen bis zu 77% besser) und bis zu 144 mal schneller ist.
Der Vorteil des Kernighan-Lin-Algorithmus gegenüber vielen heuristischen Verfahren besteht darin, daß lokale Optima unter Umständen überwunden werden können, da der Algorithmus für zwei Gruppen immer einen Tausch bis zur Verankerung aller Knoten durchführt und nicht sofort stoppt, sobald ein Tausch zweier Gruppen eine Verschlech-

[4] Das entspricht dem später bei der Initialisierung des GAG verwendeten *naiven Layout*.

terung der Kantengewichte der geschnittenen Kanten ergibt. Damit kann eine zeitweise Verschlechterung, deren Endergebnis aber einen Gewinn bringt, überwunden werden.

3.1.3 Kruskal-Algorithmus Varianten

Das Ziel des Kruskal-Algorithmus [14] ist es, zu einem gegebenen, ungerichteten Graphen einen minimalen *Spannbaum* zu ermitteln. Der Spannbaum eines Graphen ist ein freier Baum, der alle Knoten des Graphen enthält und dessen Kanten eine Teilmenge der Kanten des Graphen darstellen.

Bei dem gegebenen Problem ist das Ziel nicht ein minimaler Spannbaum des gesamten Graphen, sondern eine Partitionierung in Teilgraphen mit der maximalen Größe 16. Es erscheint sinnvoll, maximale Spannbäume der Größe 16 iterativ aus dem Graphen zu entfernen, die die zu ermittelnden Gruppen darstellen. Ein solcher Spannbaum enthält Kanten mit einem hohen Gewicht, die bei dem verwendeten Graphen die Nachbarschaftsbeziehungen anzeigen.

Der Standard-Kruskal-Algorithmus sortiert zu Beginn alle Kanten des zu bearbeitenden Graphen G nach ihrer Wertigkeit. Der Spannbaum des Graphen besteht zu Beginn nur aus allen Knoten des Graphen. Eine Komponente des Spannbaums besteht aus Knoten, die mit Kanten verbunden sind. Damit besteht der Spannbaum zu Beginn aus n Komponenten, wenn n die Zahl der Knoten des Graphen ist. Es werden nun alle Kanten mit der jeweils höchsten Wertigkeit von G dem Spannbaum hinzugefügt, wenn sie zwei Komponenten des Spannbaums verbinden. Verbinden sie keine zwei Komponenten, werden sie nicht betrachtet und die nächste Kante ausgewählt. Dies wird solange durchgeführt, bis der Spannbaum aus nur noch einer Komponente besteht. Die Laufzeit liegt bei $O(e \log e)$, wobei e die Anzahl der Kanten von G ist. Der Spannbaum ist minimal [14] .

In dieser Arbeit wurden zwei auf dem Kruskal-Algorithmus aufbauende Varianten zur Partitionierung implementiert:

Variante 1 (Kruskal A):
Bei dem Problem der Gruppenbildung anhand eines Graphen, in dem die Knoten die Variablen darstellen und Kanten nur zwischen Knoten eingefügt werden, deren Variablen eine Nachbarschaftsbeziehung besitzen, wobei die Kantengewichte die Anzahl dieser Nachbarschaftsbeziehungen darstellen, soll der Kruskal-Algorithmus eine Menge von maximalen Teilspannbäumen finden. Für eine Aufteilung des Graphen in k Gruppen wird dazu ein Kruskal-Algorithmus k mal gestartet. Sobald der in einem Durchlauf gebildete Spannbaum 16 Knoten enthält, werden diese als Gruppe ausgegeben. Daraufhin werden die Knoten und Kanten des Spannbaums im Graphen gelöscht und das Verfahren erneut gestartet. Dies wird solange wiederholt, bis alle Knoten bearbeitet worden sind. Der Vorteil eines solchen Verfahrens liegt in der Geschwindigkeit. Für eine Aufteilung eines Graphen in k Teilspannbäume muß der Kruskal-Algorithmus k mal gestartet werden. Als Nachteil erweist sich allerdings, daß Knotengruppen, die einen hohen Verbindungsgrad haben, aber vergleichsweise niedrige Kantengewichte, unter Umständen zugunsten einzelner, guter Kanten getrennt werden. Dies zeigt Abbildung 3.8 bei einer Gruppengröße von 4:

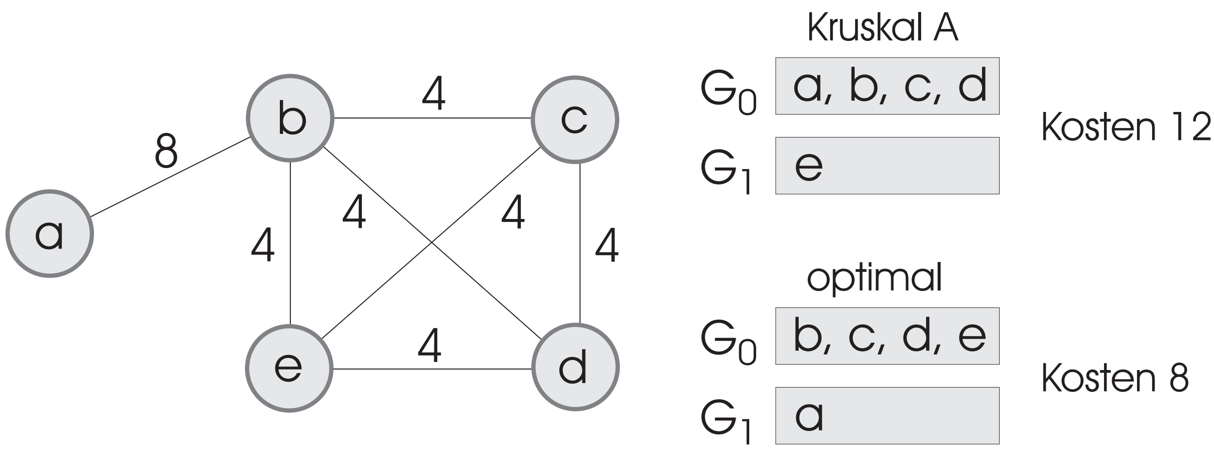

Abbildung 3.8: Vorgehensweise der Variante A des implementierten Kruskal-Algorithmus

Die Kante $A - B$ hat das höchste Gewicht (8) und wird ausgewählt. Unabhängig davon, welche beiden Knoten ebenfalls für die erste Gruppe ausgewählt werden, liegen die Kosten, die sich aus den Kantengewichten der externen Kanten ergeben, bei 12. Hier wird die Aufteilung der Gruppen (A, B, C, D) und (E) gewählt. Optimal wäre es allerdings, die Kante $A - B$ nicht zu betrachten und A in eine eigene Gruppe zu schreiben. Dann lägen die Kosten nur bei 8.

Variante 2 (Kruskal B):
Das obige Verfahren wird mit einer Tie-Break-Entscheidung [19] verfeinert. Der Tie-Break-Wert einer Kante ergibt sich, indem die Gewichte aller Kanten, die über die Endknoten dieser Kante laufen, zum Tie-Break-Wert dieses Knotens aufsummiert werden. Die Kanten des Graphen werden bei dieser Variante nach absteigendem Gewicht sortiert. Bei einer gleichen Kantengewichtung werden die Kanten in aufsteigender Reihenfolge anhand ihres Tie-Break-Wertes sortiert. Bei der wiederum gesuchten Menge von Teilspannbäumen wird dem ausgewählten Teilspannbaum zuerst die bestbewertete Kante hinzugefügt. Die entstehende Komponente K ist die einzige im Spannbaum, die mehr als einen Knoten enthalten darf. Danach werden nur noch Kanten in absteigender Reihenfolge ihres Gewichts betrachtet, die mindestens einen Knoten in K haben. Gewählt wird also die Kante mit der höchsten Bewertung. Besitzen mehrere Kanten die gleiche Bewertung, wird anhand des Tie-Break-Wertes entschieden, welche Kante gewählt wird. Sobald der Teilspannbaum 16 Knoten enhält, wird diese Knotengruppe ausgegeben und im Graphen die im Teilspannbaum enthaltenen Knoten markiert und Kanten gelöscht. Die Tie-Break-Werte werden neu berechnet und der nächste Teilspannbaum neu ausgewählt. Der Algorithmus stoppt, wenn alle Knoten markiert sind. Auch diese Variante würde in Abbildung 3.8 allerdings nicht das Optimum finden. Die Komplexität erhöht sich nur um die Berechnung des Tie-Break-Wertes, für die Berechnung jedes Tie-Break-Wertes einer Kante fällt eine Laufzeit von $O(n)$ an.

Beide Varianten sind heuristisch und finden eventuell nur ein lokales Optimum.

3.1.4 Genetische Partitionierungsalgorithmen

Vemuri [28] stellt einen genetischen Partitionierungsalgorithmus zur Lösung mehrerer Probleme beim *VLSI*[5]-Design vor. Eins der Probleme besteht beispielsweise darin, für eine Anwendung, bei der mehrere *FPGAs*[6] verwendet werden müssen, Größe, Speicherbedarf und die notwendige Verbindungsanzahl zwischen den FPGAs zu optimieren.
Vemuri setzt einen Standard-GA (siehe Seite 17) ein. Die Initialpopulation besteht aus probabilistisch ermittelten Chromosomen. Dazu werden einige von Vemuri nicht näher spezifizierte heuristische Partitionierungsergebnisse hinzugefügt, um die Konvergenz des Genetischen Algorithmus zu beschleunigen. Der Algorithmus verwendet Uniform-Crossover bei der Generierung einer neuen Population (zu 80%). Die restlichen Individuen werden in die nächste Generation übernommen und mutiert, wobei Elite-Selektion verwendet wird. Die Mutation besteht in einer einfachen Zufallsmutation der Allele.
Bewertet für die Güte der Individuen werden der Platzbedarf, Speicherbedarf und die nötigen Verbindungen. Je nach sekundärem Optimierungsziel können gleichbewertete Chromosomen sortiert werden. Genauere Details des Verfahrens von Vemuri wurden in [28] und den dortigen Referenzen nicht erwähnt und konnten nicht in Erfahrung gebracht werden.

3.2 Genetischer Algorithmus zur Gruppenbildung (GAG)

Der Lösungsraum wird beim vorliegenden Problem der Graphpartitionierung in Teilgraphen vorgegebener Größe schnell sehr groß [13]: $\frac{1}{k!} \cdot \binom{n}{p} \cdot \binom{n-p}{p} \cdot \ldots \cdot \binom{2p}{p} \cdot \binom{p}{p}$ verschiedene Lösungen bei einer Aufteilung von n Knoten in k Teilgraphen der Größe p.
Für einen Graphen mit 40 Knoten und einer Teilgraphengröße von 10 bedeutet dies schon mehr als 10^{20} Lösungen. Da Genetische Algorithmen und heuristische Suchverfahren auch in großen Lösungsräumen effizient arbeiten, werden in dieser Arbeit beim Genetischen Algorithmus zur Gruppenbildung (GAG) eine Kombination aus einem Genetischen Algorithmus und heuristischen Verfahren in dessen Initialisierung verwendet. Heuristische Suchverfahren allein liefern im allgemeinen sehr schnell Ergebnisse, diese sind aber oft suboptimal.

Die Eingabemenge des GAG ergibt sich aus der Variablenzugriffssequenz, in der nur Variablen betrachtet werden, die ausschließlich als Einzelzugriff vorkommen. Die Kodierung zeigt Abbildung 3.9:

Aus der Variablenzugriffssequenz ergibt sich eine Variablenmenge mit n unterschiedlichen Elementen. Die Elementzahl ergibt dabei die Chromosomenlänge. Die Position eines Elements in der Variablenmenge gibt an, welches Gen welche Variable repräsentiert. Allele kennzeichnen die Gruppenzugehörigkeit einer Variable, die durch das Gen, in dem das entsprechende Allel steht, kodiert wird. In Abbildung 3.9 entspricht eine Belegung mit dem Allel 0 einer Zuteilung der entsprechenden Variable zur Gruppe G_0, im Beispiel betrifft

[5]Very Large Scale Integration
[6]Field Programmable Gate Arrays

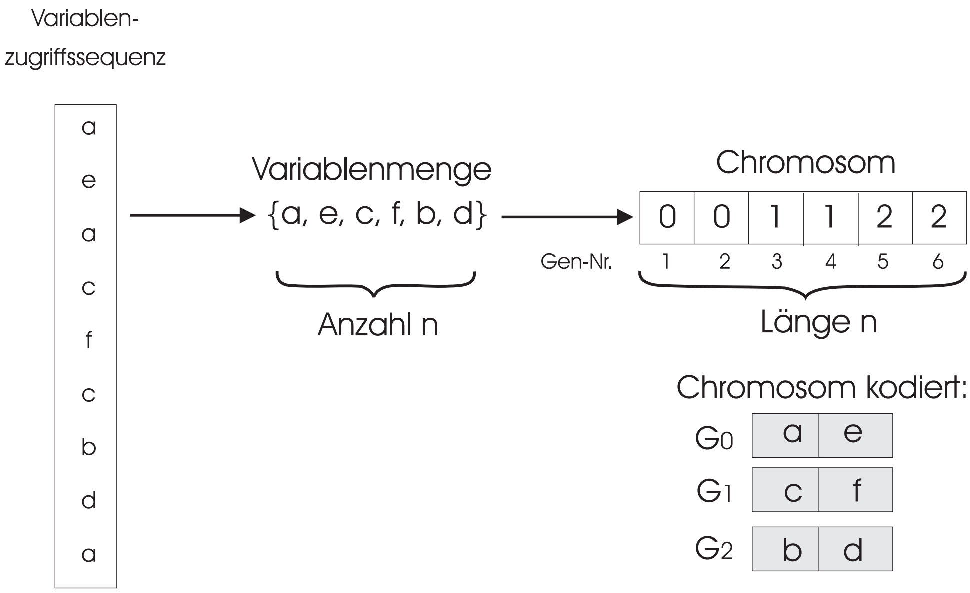

Abbildung 3.9: Kodierung beim GAG

das die Variablen a und e. Es werden nur gültige Lösungen in der Chromosomenmenge zugelassen, beispielsweise ist das Vorkommen eines Allels durch die Gruppengröße maximal begrenzt.

3.2.1 Initialisierung

Die Aufgabe der Initialisierung besteht darin, eine Population zu erschaffen, die aus möglichst unterschiedlichen und gültigen Lösungen besteht. Dazu werden die Gene der Chromosomen der Startpopulation probabilistisch belegt. Es müssen zwei Voraussetzungen erfüllt werden, um gültige Chromosomen zu erzeugen: Die Gruppen haben eine maximale Gruppengröße. Die verwendeten Gruppennummern dürfen keine Lücken aufweisen, es gibt also keine leeren Gruppen. Letzteres ist deshalb Bedingung, da die Mutation Variablen auf alle vorhandenen Gruppennummern bis zur letzten verteilen kann. Sollten Gruppen leer sein, würden die Gruppen zu Beginn sehr langsam gefüllt, weil immer wieder Variablen in leere Gruppen mutiert werden und beim Crossover häufiger Variablengruppen nur eine andere, im Chromosom vorher unbenutzte, Nummer zugewiesen werden würde, ohne Gruppen zusammenzufassen. Dadurch bekämen zu Beginn eines solchen GAG reine Zusammenfassungen kleinerer Gruppen zu einer größeren eine große Bedeutung für die Bewertung solcher Chromosomen, da eine solche Zusammenfassung meist eine bessere Ausnutzung der Nachbarschaftsbeziehungen bedeuten würde. Diese Chromosomen stellten aber meist keine qualitativ guten Gruppen dar, da sie nur durch eine Zusammenfassung wahrscheinlich vergleichsweise ungeeigneter Variablen eine gute Bewertung erzielten. Die Unterschiede bei der Bewertung zweier voller Gruppen, die wahrscheinlich die Qualität eines Chromosoms am Ende des GAG ausmachen, tauchen erst nach mehreren Iterationen auf, wenn die am Anfang existierenden Teilgruppen zusammengefaßt sind. Dadurch würde der GAG langsamer konvergieren.
Die maximale Gruppengröße wird sichergestellt, indem die Gene der zufällig erzeugten

Individuen sequentiell mit Werten belegt werden und die Zuteilung der Variablen zu Gruppen mitverwaltet wird. Dadurch kann bei der Auswahl jedes Allels überprüft werden, ob die entsprechende Gruppe schon voll ist und dementsprechend ein anderer Wert gewählt werden. Leere Gruppen werden verhindert, indem bei der sequentiellen Füllung als Wert eines Gens alle bisher bestehenden Gruppen und die erste leere Gruppe gewählt werden können. Dadurch ist es im Extremfall möglich, daß jede Variable einer eigenen Gruppe zugeordnet wird. Um dies zu vermeiden, wird eine neue Gruppe nur mit einer geringeren Wahrscheinlichkeit ausgewählt, wenn noch Plätze in den schon vorhandenen Gruppen frei sind.

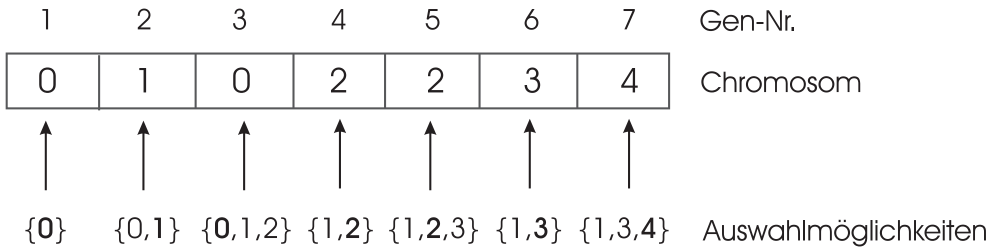

Abbildung 3.10: Initialisierung des GAG bei einer Gruppengröße von 2

Im vorliegenden Beispiel (Abbildung 3.10) wird ein Chromosom bei einer vorgegebenen Gruppengröße von 2 probabilistisch initialisiert. Das erste Gen kann nur mit dem Wert 0 belegt werden. Das zweite Gen kann mit den Werten 0 oder 1 belegt werden, in diesem Fall mit der 1. Da das dritte Gen als Wert eine 0 erhält, ist für die Gruppe 0 die maximale Größe erreicht. Das vierte Gen kann dann nur noch die Werte 1 oder 2 erhalten. Entsprechend werden die weiteren Gene belegt.

Zusätzlich kann die Startpopulation um Ergebnisse aus den implementierten heuristischen Suchverfahren erweitert werden, da der Lösungsraum sehr groß ist und der GAG durch eine Initialisierung mit diesen Ergebnissen die Möglichkeit hat, gezielter im Lösungsraum zu suchen.
Es werden die Lösungen von drei heuristischen Suchverfahren hinzugefügt:
Zum einen die Lösungen beider Kruskal-Algorithmus Varianten A und B, die wahrscheinlich suboptimal sind, aber bei deren Ermittlung die Laufzeit sehr kurz ist. Weiterhin kann die Lösung des Kernighan-Lin-Algorithmus ebenfalls hinzugefügt werden. Zusätzlich enthält die Initialpopulation eine naive Lösung. Dazu werden die Variablen in der Reihenfolge ihres ersten Auftretens an die nächste freie Position einer Gruppe geschrieben. Diese Lösung nutzt unter Umständen schon eine beachtliche Menge von Nachbarschaftsbeziehungen aus, da jede Variable mit bis zu zwei Nachbarn einer Gruppe zugeteilt wird. Auf eine Initialisierung einer naiven Lösung mit anderen Vorgaben, zum Beispiel einer alphabetischen Ordnung, wird verzichtet, da sie mit hoher Wahrscheinlichkeit keine Vorteile gegenüber einer zufälligen Lösung hätte.

3.2.2 Bewertung

Nach der Bildung der Initialpopulation müssen die Chromosomen bewertet werden. Auch im weiteren Verlauf des Genetischen Algorithmus ist die Bewertungsfunktion von ent-

scheidender Bedeutung für den Genetischen Algorithmus, da sie die Grundlage für die Selektion darstellt.
Die Bewertungsfunktion ermittelt die Güte eines Individuums aufgrund der Speicherzugriffe, die nötig sind, um die Variablenzugriffssequenz mit der im Chromosom kodierten Variablenzuordnung zu Gruppen abzuarbeiten. Dabei werden die besonderen Kosten bei Schreibzugriffen berücksichtigt, so daß Zugriffen auf Variablen, die schreibend sind, ein lesender Zugriff vorausgehen muß. Zudem muß eine Variable, die im Verlauf der Zugriffe auf eine Gruppe geschrieben wird, auch im Speicher gesichert werden, da die Zugriffe zunächst nur in Registern stattfinden. Das verursacht ebenfalls einen zusätzlichen Speicherzugriff.
Der für die Länge dynamischer Schleifen (siehe Seite 30) einzustellende Wert kann entweder vom Benutzer vorgegeben werden, oder der Mittelwert aller statischen Schleifen wird als Grundlage für dynamische Schleifen ermittelt.

3.2.3 Crossover

Die Nachfolgegeneration wird neben einer Übernahme der besten Individuen der Vorgängergeneration dadurch bestimmt, daß die fittesten Individuen der Vorgängergeneration rekombiniert werden. Dies geschieht mittels Crossover. Beim GAG wird 1-Punkt Crossover verwendet. Zwei Elternchromosomen werden ausgewählt und eine beliebige, in beiden Chromosomen gleiche Stelle markiert. Der hintere Teil der Chromosomen wird dann getauscht.
Dabei können ungültige Lösungen entstehen, da die Bedeutung gleicher Allele in jedem Chromosom unterschiedlich ist. In Abbildung 3.11 wurden zwei Elternchromosomen selektiert, die maximale Gruppengröße ist 4. Nun werden die letzten beiden Gene der Elternchromosomen getauscht und zwei Nachkommen erzeugt. Da aber der zweite Nachkomme im vorderen Teil schon drei Elemente der Gruppe 0 besitzt, erzeugen die beiden Gene mit dem Wert 0, die getauscht worden sind, eine ungültige Lösung, da die Gruppe 0 nun fünf Elemente enthält. Deswegen schließt sich für jeden so erzeugten Nachkommen eine Korrektur in der Mutationsphase an, die Nachkommen werden dort nicht unbedingt mutiert, aber auf jeden Fall korrigiert.

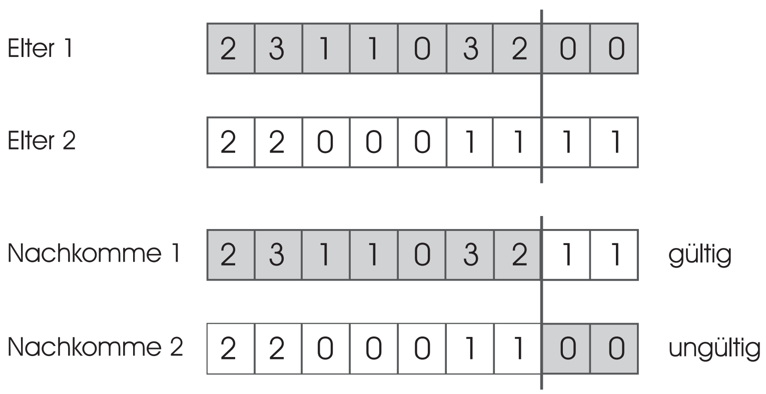

Abbildung 3.11: Crossoverfunktion des GAG bei einer Gruppengröße von 4

3.2.4 Mutation

Die Mutation hat beim GAG zwei Aufgaben:
Eine verfrühte Konvergenz des Genetischen Algorithmus durch eine zu einseitige Population wird verhindert. Zudem stellt die Mutation die Gültigkeit aller Chromosomen sicher, indem eine Korrektur auf allen Chromosomen durchgeführt wird, die zur Mutation ausgewählt wurden (die Korrektur betrifft auch die mit Crossover erschaffenen Chromosomen).
Die Mutation wird abhängig von der Mutationsrate sowohl auf Chromosomen, die ohne Rekombination in die Nachfolgepopulation übernommen werden sollen, als auch auf rekombinierte Chromosomen angewendet. Die Mutation läuft in zwei Schritten ab:
Zunächst werden die zu mutierenden Gene des zur Mutation ausgewählten Chromosoms bestimmt. Vor der Belegung dieser Gene mit neuen Werten wird die höchste vergebene Gruppennummer, die dieses Chromosom enthält, ermittelt. Gültige neue Werte für die ausgewählten Gene sind alle Gruppennummern bis zu dieser Gruppe und die darauffolgende leere Gruppe. Dadurch wird die Möglichkeit erhalten, den ganzen Lösungsraum zu betrachten, da wie zuvor gezeigt nichtvolle Gruppen von Vorteil sein können (siehe Seite 31). Es wird nicht überprüft, ob der gewählte Wert eine schon volle Gruppe darstellt. Damit wird ebenfalls die Möglichkeit gewahrt, den ganzen Lösungsraum zu betrachten, da ansonsten eine suboptimale Gruppenzusammensetzung unter Umständen nicht mehr getrennt wird. Durch die Zuteilung einer weiteren Variable muß die Gruppe wieder verkleinert werden. Ein beliebiges Gen der Gruppe wird verändert, womit bislang unbetrachtete Gruppenzusammenstellungen entstehen können.
Damit können nach diesem Schritt ungültige Chromosomen existieren, in denen Gruppen leer sind oder mehr Elemente als erlaubt enthalten.

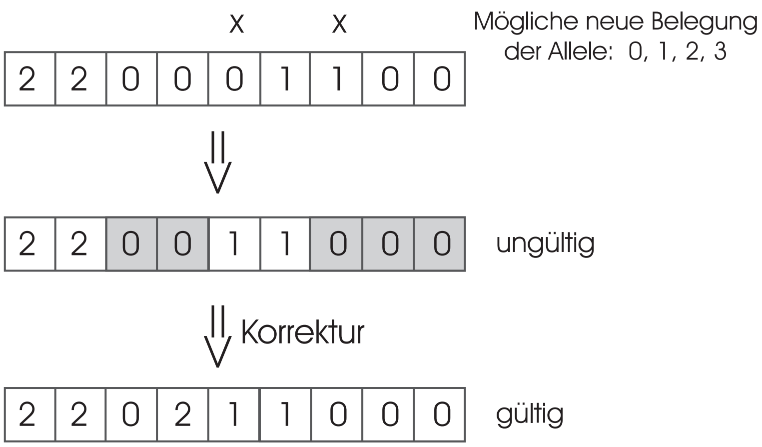

Abbildung 3.12: Mutationsfunktion des GAG bei einer Gruppengröße von 4

Der zweite Schritt der Mutation, die Korrektur, läuft in zwei Phasen ab und soll die Gültigkeit der Chromosomen wiederherstellen.
Zunächst werden sequentiell alle Gruppen, die mehr Elemente als erlaubt enthalten, auf die maximale Größe reduziert und die überzähligen Elemente nur auf solche Gruppen verteilt, in denen noch Platz ist. Dies kann ebenfalls die erste leere Gruppe nach der letzten nichtleeren sein, die Auswahl der Elemente und der Zielgruppen ist probabilistisch.

Danach werden alle unbenutzten Gruppennummern gelöscht und die restlichen Gruppennummern entsprechend angepaßt. Somit besitzen alle Gruppen wieder fortlaufende Nummern, es existieren somit nur noch gültige Chromosomen. Abbildung 3.12 zeigt die Mutation an einem Beispiel. Zunächst wird ein Chromosom zur Mutation bestimmt, hier der zweite Nachkomme aus Abbildung 3.11. Zwei Positionen werden in der Mutationsphase ausgewählt und dann mutiert, es ergibt sich das zweite Chromosom, das aber ebenfalls ungültig ist, da fünf Gene (im Beispiel grau markiert) das Allel 0 enthalten. Aus allen Genen, die das Allel 0 enthalten, wird eines für eine Veränderung ausgewählt, in diesem Fall das Gen 4. Als neue Werte für dieses Gen kommen $1, 2$ oder 3 in Frage, in diesem Beispiel wird die 2 als neues Allel gewählt. Damit ergibt sich ein gültiges Chromosom, das in die Nachfolgegeneration übernommen werden kann.

3.2.5 Betrachtung lokaler Variablen

Die bisher von dem GAG verarbeiteten Variablen erhalten einen Gruppenplatz und damit später (in der Gruppenanordnung) eine Speicherstelle, die sie für den ganzen Programmablauf belegen. Sie können damit wie globale Variablen betrachtet werden. Der Speicherplatz lokaler Variablen kann jedoch spätestens dann freigegeben werden, wenn diese Variablen im Verlauf einer Funktion nicht mehr verwendet werden. Um bewerten zu können, wann eine lokale Variable im Speicher vorliegen muß, wird der Begriff der *Lebensdauer* eingeführt [15]. Die Lebensdauer einer Variablen gibt die Zeitspanne an, in der die Variable im Speicher vorliegen muß. Außerhalb dieser Lebensdauer wird die Variable nicht benötigt und der Speicher kann anders belegt werden.
Bei einer Betrachtung lokaler Variablen können sich mehrere Variablen eine Speicherstelle teilen, wenn ihre Lebensdauer sich nicht überschneidet. Dadurch können sowohl Speicherplatz als auch Speicherzugriffe eingespart werden. Ein Beispiel dazu zeigt Abbildung 3.13: Da die Lebensdauer von a endet, bevor auf f das erste Mal zugegriffen wird, kann f an der gleichen Position gespeichert werden wie a vorher. Zusätzlich zu einer Reduzierung der insgesamt nötigen Speicherstellen um 1 werden Speicherzugriffe eingespart. Im vorliegenden Beispiel werden alle Nachbarschaftsbeziehungen von f zu den Variablen in der Gruppe 0 ebenfalls ausgenutzt, hier insbesondere zu der Variablen c.

Kurdahi und Parker [15] haben ein Verfahren vorgestellt, mit dem anhand einer Zugriffssequenz auf Variablen bestimmt werden kann, wieviele Register benötigt werden, um alle Variablen im Verlauf der gesamten Sequenz speichern zu können. Der *Left-Edge-Algorithmus (LEA)* ermittelt eine Zuweisung der Variablen zu bestimmten Registern.
Dies kann für das vorliegende Problem genutzt werden. Die Gruppen von lokalen Variablen können mehr als 16 Variablen enthalten und trotzdem die Gruppengröße nicht überschreiten, falls sich Variablen in ihrer Lebensdauer nicht überschneiden. Mit dem LEA kann für eine gegebene Menge von lokalen Variablen bestimmt werden, wieviele Speicherstellen benötigt werden (Abbildung 3.14). Ist die ermittelte Zahl nicht größer als die vorgegebene Gruppengröße, können diese Variablen später in einer Gruppe abgespeichert werden. In Abbildung 3.14 zeigt a) ein Layout ohne Anwendung eines LEA mit maximaler Gruppengröße 2. Wird der LEA auf die Variablen angewendet, werden jedoch nur zwei Gruppen benötigt (Abbildung 3.14b)). Somit werden Speicherplatz und Speicherzugriffe eingespart.

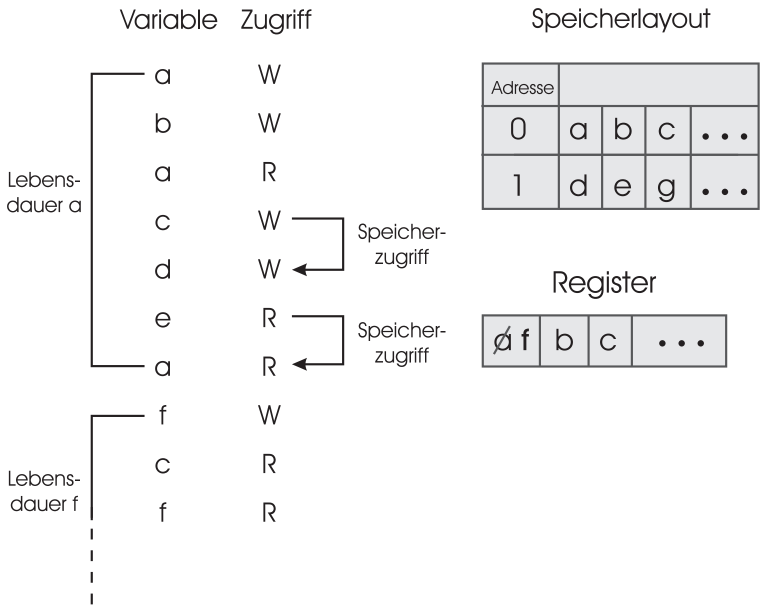

Abbildung 3.13: Einsparung von Speicherplatz und Speicherzugriffen

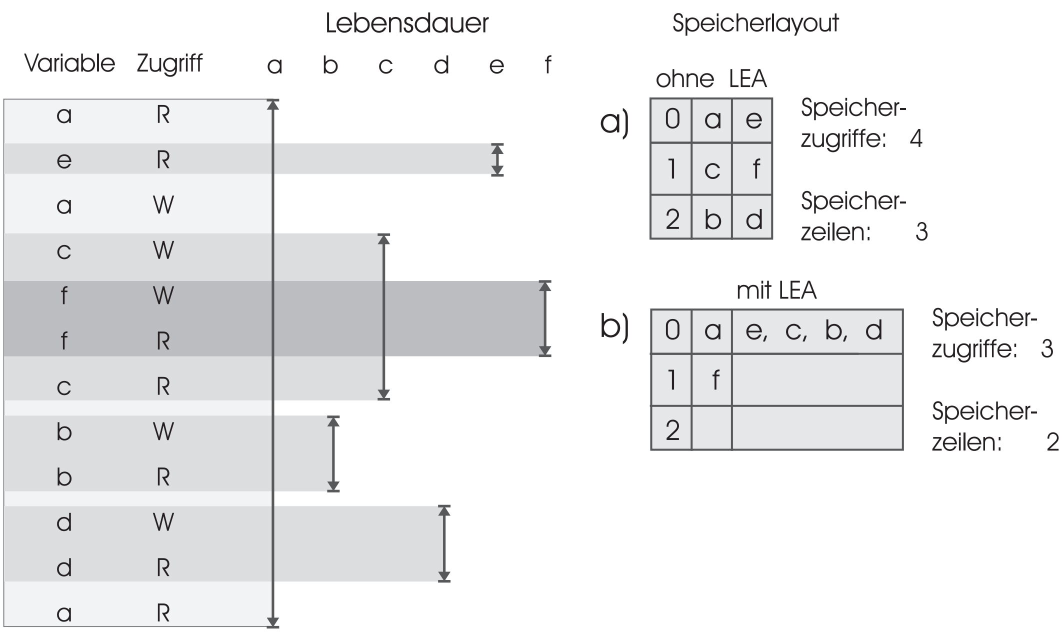

Abbildung 3.14: Left-Edge-Algorithmus bei lokalen Variablen beim GAG

Eingesetzt wird dieses Verfahren in der Initialisierung und der Mutation des GAG, da hier eine Prüfung auf Gültigkeit der Individuen stattfindet.
Der GAG wird mit dem LEA erweitert. Anstelle der Überprüfung, ob maximal 16 Variablen einer Gruppe zugeordnet sind, wird nun mit einem LEA überprüft, ob das Individuum gültig ist. Dies geschieht, indem für jede Gruppe von Variablen, die im Chromosom kodiert mehr als 16 Variablen enthält, überprüft wird, ob die Variablen dieser Gruppe mit 16 Speicherstellen auskommen. Ist das nicht der Fall, wird eine Variable einer anderen Gruppe zugeordnet und erneut überprüft, falls weiterhin mehr als 16 Variablen dieser Gruppe zugeordnet sind.
Das führt zu einer veränderten Ausgabe des GAG:
Bei globalen Variablen ist es nur wichtig, die Gruppenzugehörigkeit zu kennen. Bei lokalen Variablen muß zusätzlich noch die Spalte angegeben werden, in der die Variablen stehen, denn diese ist wichtig, um erkennen zu können, welche Variablen sich eine Speicherstelle teilen und damit Lebenszeiten haben, die sich nicht überschneiden.

3.3 Resultate

In diesem Abschnitt werden die Ergebnisse des GAG vorgestellt. Dazu wird der GAG zunächst mit den optimalen Ergebnissen eines *CLP*[7]-basierten Verfahrens bei Sequenzen mit einer kleinen Variablenanzahl verglichen. Die Ergebnisse der verwendeten heuristischen Verfahren des Kernighan-Lin-Algorithmus und der beiden vorgestellten Kruskal-Algorithmus-Varianten werden ebenfalls vergleichsweise vorgestellt.
Bei größeren Sequenzen wird im späteren Verlauf als Vergleichswert eine naive Gruppenpartitionierung herangezogen. Dabei ist allerdings zu beachten, daß diese naive Partitionierung aufgrund des ersten Auftretens der Variablen in der Variablenzugriffssequenz gebildet wird, und nicht etwa völlig unabhängig von einer Bewertung, wie das bei einer alphabetischen Zuteilung der Fall wäre. Durch diese Zusammenfassung werden unter Umständen schon viele Nachbarschaftsbeziehungen ausgenutzt. Nach dem Vergleich des GAG mit einer naiven Lösung und den implementierten heuristischen Verfahren bei größeren Testsequenzen werden einige Variablenzugriffssequenzen realer Benchmarks bewertet. In beiden Fällen wird neben der Güte der Ergebnisse auch auf die Laufzeiten eingegangen. Daraufhin wird gezeigt, daß es sinnvoll ist, heuristische Verfahren zu verwenden, um den GAG zu initialisieren. Dies wird deutlich, wenn die Ergebnisse einiger vom GAG eingelesener Sequenzen mit und ohne Initialisierung durch heuristische Verfahren verglichen werden.
Die Verbesserungen beim Einsatz des Left-Edge-Algorithmus bei lokalen Variablen werden ebenfalls anhand einiger Testsequenzen aufgezeigt. Weiterhin wird gezeigt, welchen Einfluß die genetischen Parameter bei der Güte der Ergebnisse besitzen.

3.3.1 Vergleich des GAG mit einem CLP-basierten Verfahren

Die ermittelten Ergebnisse eines CLP-basierten Verfahrens können als Vergleichsbasis dienen, wie gut die Ergebnisse des GAG sind. Mit CLP können Probleme mit vielen

[7]Constraint Logic Programming

Randbedingungen (englisch: Constraints) relativ einfach umgesetzt werden. Bei der Betrachtung des Problems durch ein CLP-basiertes Verfahren wird unter Umständen der gesamte Lösungsraum durchsucht. Durch eine geschickte Wahl der Constraints kann aber ein Teil des Lösungsraums unbetrachtet bleiben, obwohl die optimale Lösung nachweisbar gefunden wird. Der größte Teil der Laufzeit wird bei dem verwendeten CLP-basierten Verfahren für die Verifikation einer zwischenzeitlich ausgegebenen angenommenen optimalen Lösung verwendet. Das Ergebnis dieses Verfahrens ist optimal. Allerdings ist die Laufzeit von CLP-basierten Verfahren exponentiell abhängig von der Variablenzahl, da die Lösung des Problems NP-hart ist. Deswegen werden nur kleine Variablenmengen verwendet. Bei einer Variablenmenge von 25 und einer Gruppengröße von 4 liegen die Laufzeiten des CLP schon bei über 8,5 Stunden (30907 Sekunden). Deshalb werden Zufallssequenzen der Länge 200 mit 5, 10, 15 und 20 verschiedenen Variablen als Eingabe für CLP und GAG verwendet. Um bewertbare Ergebnisse zu erhalten, wird die Gruppengröße auf 4 reduziert. Da die M3-DSP Plattform parametrisierbar ist, ergibt dies weiterhin sinnvolle Ergebnisse, erleichtert aber das Errechnen einer optimalen Lösung. Durch eine kleinere Gruppengröße entstehen auch bei wenigen Variablen viele Gruppen, was die Anordnungsmöglichkeiten erhöht, da die Reihenfolge der Variablen in der Gruppe unerheblich ist. Bei großen Gruppen entstehen mehr gleichwertige Lösungen (einfache Permutationen der Variablen). Der GAG wird für jede der Sequenzen 100 Mal gestartet. Die Berechnung dauert auf einer Sun Ultra-Sparc 10 pro Durchlauf für den GAG wenige Sekunden (4,5 Sekunden bei 20 Variablen). Abbildung 3.15 zeigt, wie oft sich von 100 Testläufen pro Testsequenz mit

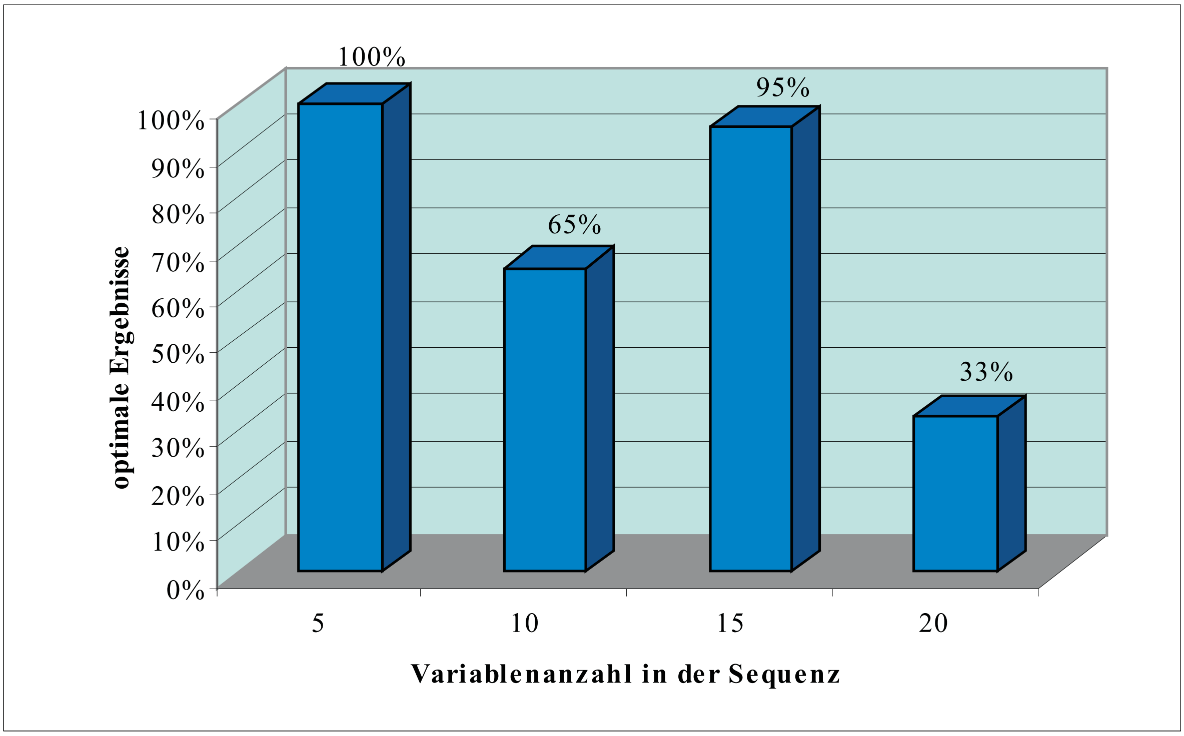

Abbildung 3.15: Anteil des optimalen Ergebnisses an der Gesamtzahl der Ergebnisse beim GAG

unterschiedlichen Zufallszahlen optimale Ergebnisse einstellten. Die Zahl der optimalen

Ergebnisse nimmt mit steigender Variablenzahl ab. Es wird aber auch deutlich, daß die Reihenfolge und die Häufigkeit der Variablen in der Sequenz unabhängig von der Variablenanzahl und der Sequenzlänge ebenfalls einen großen Einfluß auf das Ergebnis haben, da bei der dritten Testsequenz mit 15 Variablen häufiger das Optimum gefunden wird als bei der zweiten Testsequenz, in der nur 10 Variablen vorhanden sind.
Wichtiger als das optimale Ergebnis ist bei einem Genetischen Algorithmus die allgemeine Güte der Ergebnisse. Genetische Algorithmen können effizient in einem großen Suchraum nahezu optimale Ergebnisse finden. Abbildung 3.16 zeigt, wie nah der GAG dem optima-

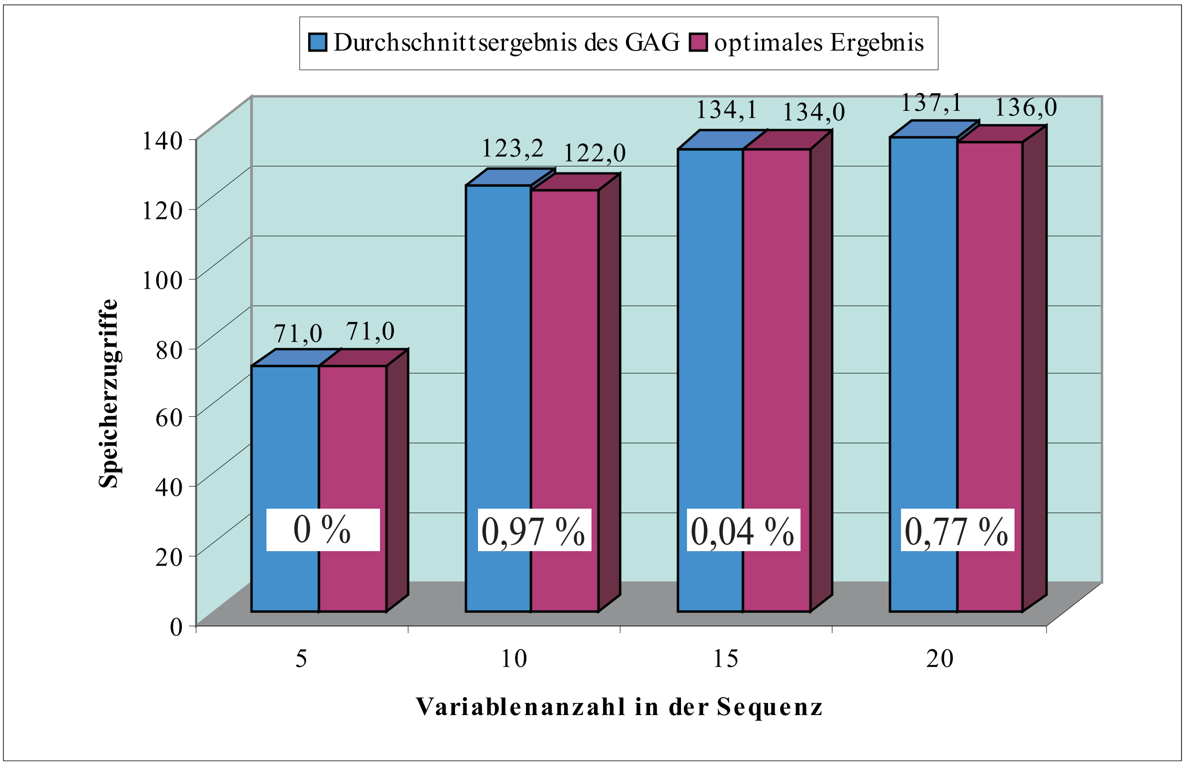

Abbildung 3.16: Durchschnittliche Abweichung der Ergebnisse des GAG vom Optimum

len Ergebnis kommt, auch wenn er es nicht immer findet. Die größte Abweichung aller Lösungen vom Optimum ergibt sich bei Sequenz 2 $(0,97\%)$. Selbst die schlechteste aller Einzellösungen hat nur eine Differenz von $2,22\%$ (bei 10 Variablen) zur optimalen Lösung.

Abbildung 3.17 zeigt die Abweichung der implementierten heuristischen Verfahren und des naiven Ergebnisses vom Optimum für diese vier Beispielsequenzen. Da der Kernighan-Lin-Algorithmus und beide Kruskal-Varianten bei einem Beispiel mit fünf Variablen schon ein globales Optimum finden, erklärt sich auch, daß der GAG immer eine optimale Lösung findet, da er mit den Ergebnissen dieser heuristischen Verfahren initialisiert wird. Es zeigt sich, daß die heuristischen Verfahren je nach Sequenz unterschiedlich gut abschneiden. Es ist deswegen sinnvoll, alle heuristischen Verfahren in die Startpopulation des GAG zu übernehmen, da durch die verschiedenen Lösungsansätze der Lösungsraum besser abgedeckt wird und auch die jeweils beste Lösung der heuristischen Verfahren vertreten ist. Der GAG kann in den meisten Fällen die Ergebnisse einzelner Heuristiken noch verbessern. Die Parameter des GAG wurden bei diesen Sequenzen auf eine Crossover-Rate von

0, 85, eine Mutationsrate von 0, 04, eine Populationsgröße von 30 und 1000 Iterationen eingestellt.

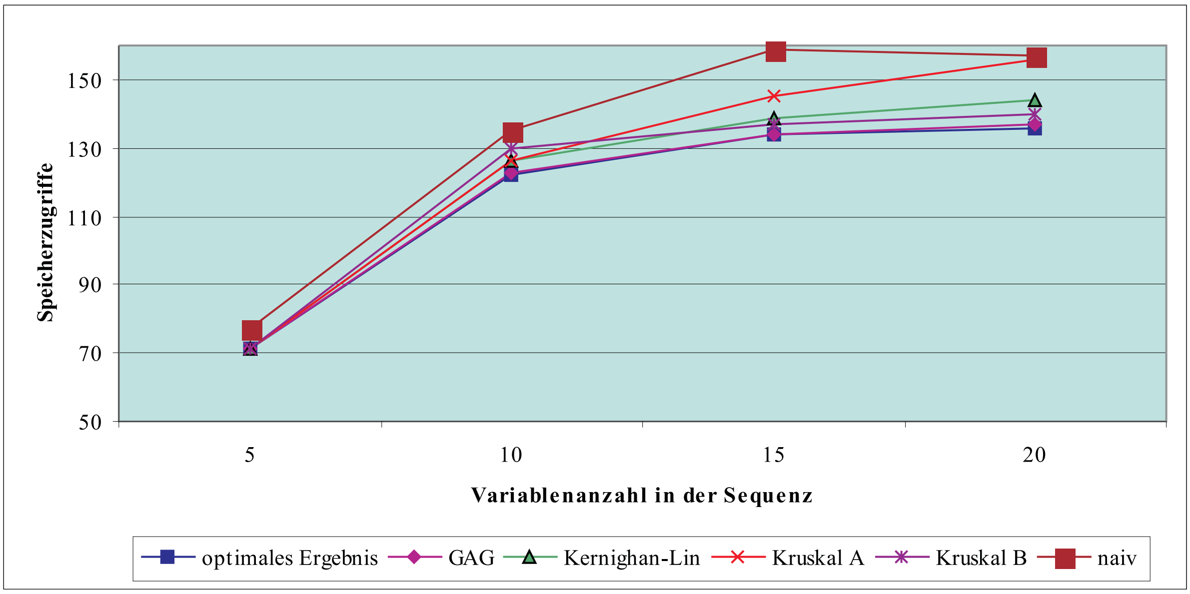

Abbildung 3.17: Vergleich der Lösungen der heuristischen Verfahren mit dem GAG und dem Optimum

3.3.2 Einsatz des GAG bei zufallsbasierten Testsequenzen

Die Sequenzlänge hat keinen großen Einfluß auf die Güte der Ergebnisse des GAG. Durch eine längere Sequenz werden nur die Nachbarschaftsbeziehungen zwischen Variablen erhöht. Eventuell werden die Bewertungsgrundlagen für die betrachteten Algorithmen deutlicher, da die Differenzen zwischen den Nachbarschaftsbeziehungen größer werden. Aber diese können auch bei Sequenzen gleicher Länge stark unterschiedlich sein. Was allerdings einen großen Einfluß auf die Einsparmöglichkeiten hat, ist die Zahl der unterschiedlichen Variablen. Durch eine größere Zahl von Variablen steigt die Zahl der möglichen Lösungen exponentiell [13]. Dadurch finden die verwendeten Algorithmen in der Regel schlechtere Lösungen. Außerdem steigt die Laufzeit.

Zunächst soll aber gezeigt werden, daß schon die Sequenz selbst einen großen Einfluß auf die Güte der Ergebnisse hat (Abbildung 3.18). Bei den verwendeten Sequenzen sind die Sequenzlänge mit 200 und die Variablenanzahl mit 25 konstant. Die Ergebnisse des GAG sind ermittelte Durchschnittswerte aus 20 Testläufen mit jeder Sequenz. Die Prozentwerte in der Abbildung geben bei jeder Sequenz die Einsparungen des GAG im Vergleich zur jeweils schlechtesten Heuristik an.
Der GAG hat den Vorteil, von allen heuristischen Verfahren ausgehend nach dem Optimum suchen zu können. Je nach Sequenz konvergieren die heuristischen Verfahren zu einem schlechten lokalen Optimum. Es fällt auf, daß der Kernighan-Lin-Algorithmus meist gute Ergebnisse findet. Teilweise ist aber die Variante B des Kruskal-Algorithmus besser

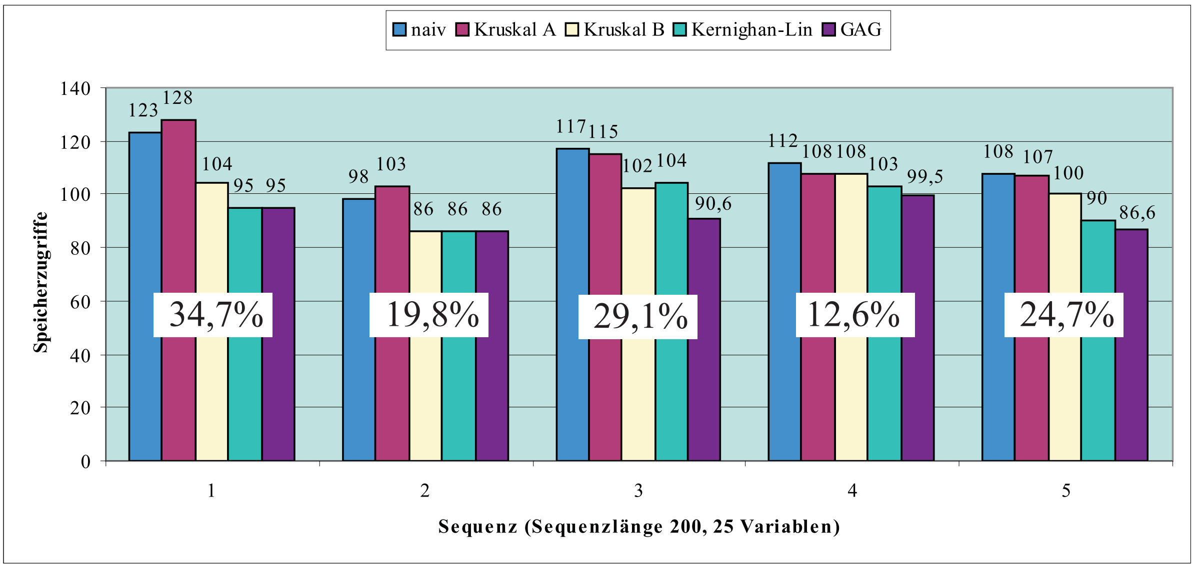

Abbildung 3.18: Einfluß der Variablenanordnung in der Sequenz auf die Güte der Ergebnisse

(siehe Sequenz 3). Der Kruskal-Algorithmus nach Variante A ist offensichtlich kein günstiger Partitionierungsalgorithmus, teilweise liefert er schlechtere Ergebnisse als eine naive Lösung. Auch sonst sind dessen Ergebnisse nur unwesentlich besser als die der naiven Lösung. Da jedoch die Laufzeit sehr kurz ist und ein weiteres Individuum in der Initialpopulation keine negativen Konsequenzen für den GAG hat, wird er zunächst als Initialisierung für den GAG belassen. Zudem besteht das Ziel der Initialisierung auch darin, möglichst unterschiedliche Lösungen zu erschaffen.

Abbildung 3.19 zeigt die prozentualen Einsparungen des GAG im Vergleich zur naiven Lösung bei Sequenzen unterschiedlicher Länge mit der Variablenzahl 25. Die prozentualen Verbesserungen im Vergleich zu einer naiven Lösung nehmen nicht zu, sondern sinken eher. Offensichtlich findet auch bei längeren Zufallssequenzen keine verwertbare Streuung der Nachbarschaftswerte statt. Es fällt allerdings auf, daß bei kleinen Zufallssequenzen das Einsparpotential hoch ist (10%), und ab einer Sequenzlänge von 2000 zwischen 5% und 8,5% liegt, ohne tendenziell zu sinken. Wahrscheinlich wirken sich bei kurzen Sequenzen zufällige Ansammlungen von Variablen mit hohen Nachbarschaftsbeziehungen in der Sequenz stärker aus.

Da die M3-DSP Plattform parametrisierbar ist, wird nun betrachtet, wie sich verschiedene Speicherbreiten auf die Ergebnisse auswirken (Abbildung 3.20). Das Einsparpotential im Vergleich zur naiven Lösung wird bei größeren Gruppen bei gleicher Sequenzlänge tendentiell größer. In Abbildung 3.20 wird dieser Effekt deutlich. Dazu wurden fünf verschiedene Testsequenzen pro Sequenzlänge erzeugt und an den GAG übergeben. Die Abbildung gibt die durchschnittliche prozentuale Einsparung bei unterschiedlichen Sequenzen abhängig von der Sequenzlänge an.
Abbildung 3.21 zeigt nochmals deutlich, daß die Gruppengröße (GG) einen Effekt auf die prozentualen Einsparungen hat. Dies kann dadurch erklärt werden, daß bei einer großen Gruppengröße ein höheres Optimierungspotential vorhanden ist. Bei einer kleinen Gruppe

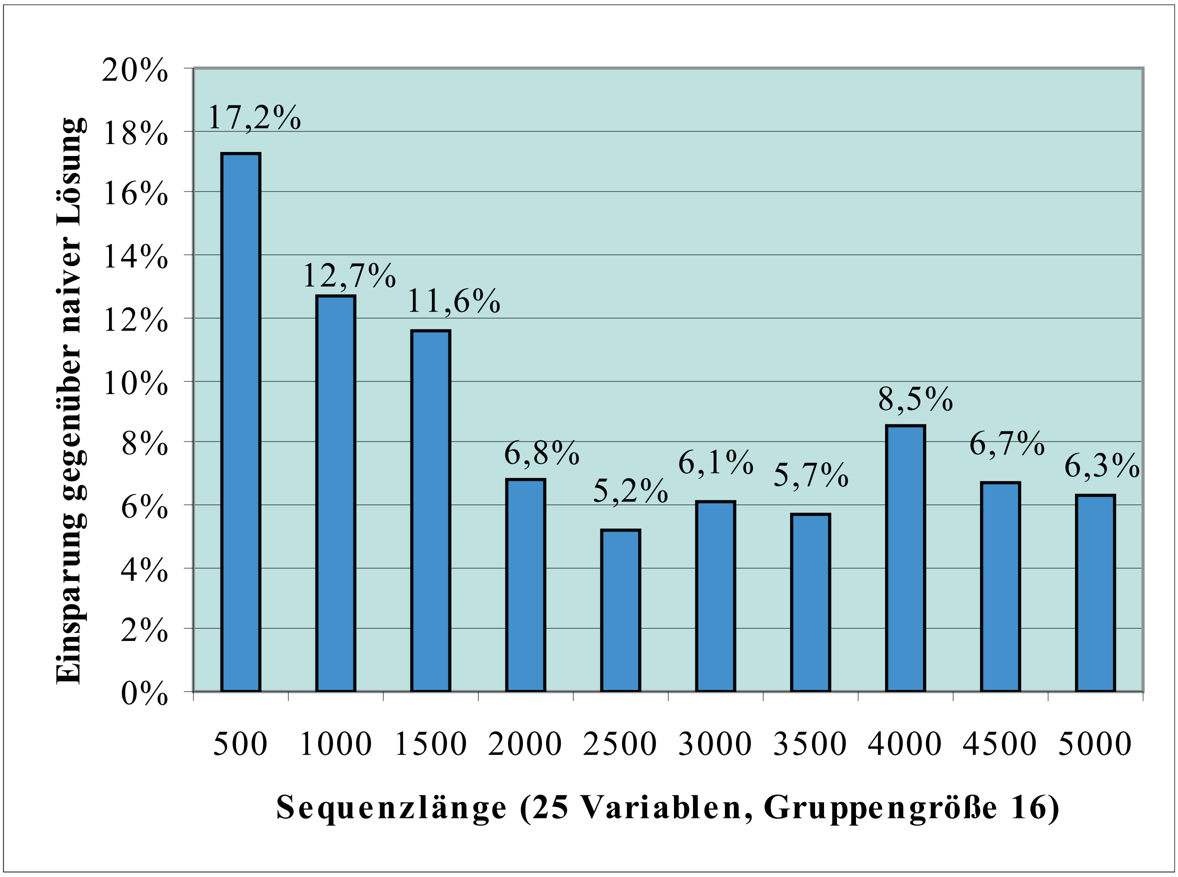

Abbildung 3.19: Prozentuale Einsparung des GAG bei Sequenzen unterschiedlicher Länge

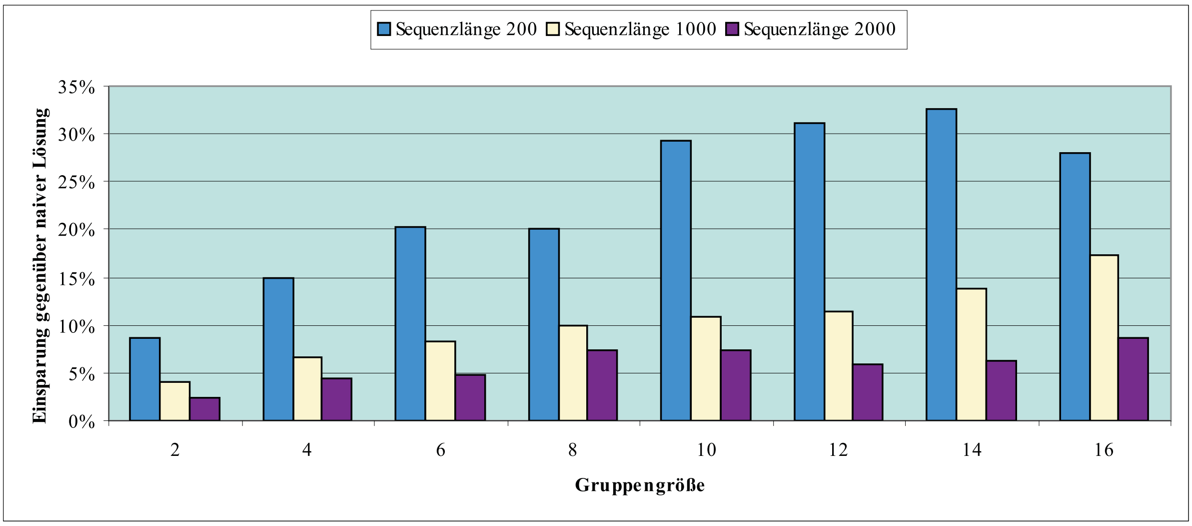

Abbildung 3.20: Verschiedene Gruppengrößen bei Sequenzen mit 25 Variablen

ist die Anzahl der nötigen Sprünge zwischen verschiedenen Gruppen auch mit einer Optimierung groß. Bei großen Gruppen können durch eine geschickte Bildung dieser Gruppen jedoch viele Speicherzugriffe eingespart werden. Abhängig von den Sequenzen kann dieser Effekt von anderen überlagert werden, wie die Sequenzen mit der Länge 200 bei der Gruppengröße 16 zeigen.

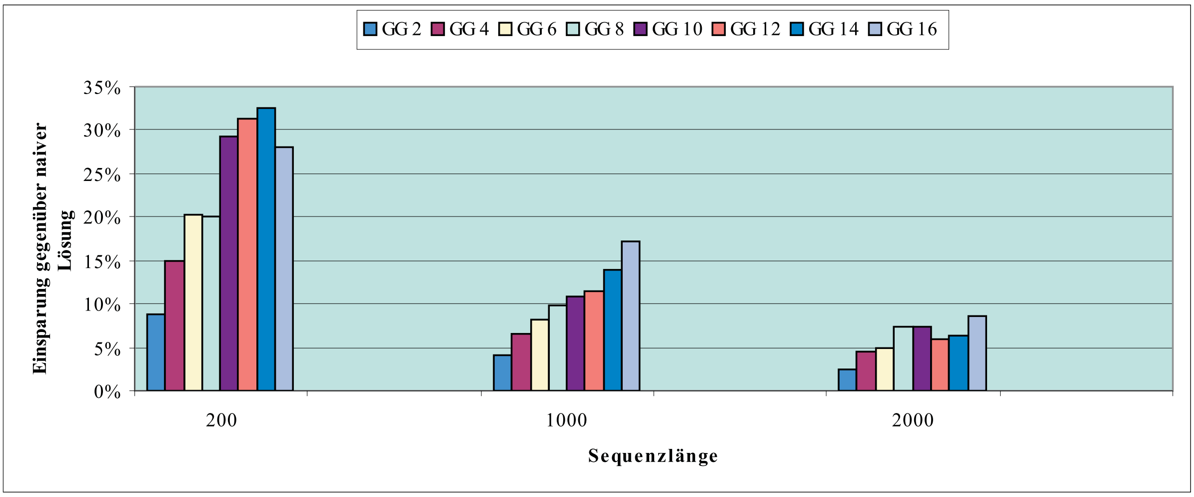

Abbildung 3.21: Abhängigkeit des Optimierungspotentials von der Gruppengröße

Es zeigt sich auch, daß die prozentualen Einsparungen mit zunehmender Sequenzlänge sinken. Dies kann aber an den zufallsbasierten Testsequenzen liegen. Die Unterschiede der Nachbarschaftsbeziehungen werden dadurch verringert, daß zwar die absoluten Abstände zwischen den Nachbarschaftswerten wahrscheinlich konstant bleiben, aber durch die insgesamte Vergrößerung der Nachbarschaftswerte die relativen Unterschiede geringer werden. Dadurch könnte das Optimierungspotential sinken.

Die Laufzeiten der betrachteten Sequenzen mit 25 Variablen sind linear abhängig von der Sequenzlänge. Da für eine Bewertung einer Gruppenaufteilung die Sequenz betrachtet werden muß und die Anzahl der Bewertungen nahezu konstant ist, war dieses Ergebnis zu erwarten (Abbildung 3.22).

Die Abhängigkeit der Laufzeiten des GAG von der Variablenanzahl stellt Abbildung 3.23 dar. Bei einer konstanten Sequenzlänge von 1000 und einer gewählten Gruppengröße von 4 wurden Sequenzen mit 8, 16, 32, 64 und 128 Variablen getestet. Die Laufzeit des GAG nimmt mit steigender Variablenzahl ohne heuristischen Verfahren nahezu linear zu. Dies erklärt sich dadurch, daß die Chromosomenlänge direkt von der Variablenanzahl abhängt und die zu leistende Arbeit der Schritte Mutation und Crossover direkt von der Länge der Chromosomen abhängig ist. Der Kernighan-Lin-Algorithmus hat jedoch keine lineare Laufzeit abhängig von der Variablenanzahl, was in Abbildung 3.23 ersichtlich wird. Bei 128 Variablen wird dieser Effekt sehr deutlich, davor sind die Laufzeiten aller heuristischer Verfahren deutlich kleiner als die des genetischen Teils des GAG. Der Kernighan-Lin-Algorithmus hat schon bei 128 Variablen eine beachtliche Laufzeit, die fast genauso groß ist wie die des GAG ohne eine Initialisierung mit heuristischen Verfahren. Aus diesem Grund kann der Kernighan-Lin-Algorithmus am Tool deaktiviert werden, sollten die

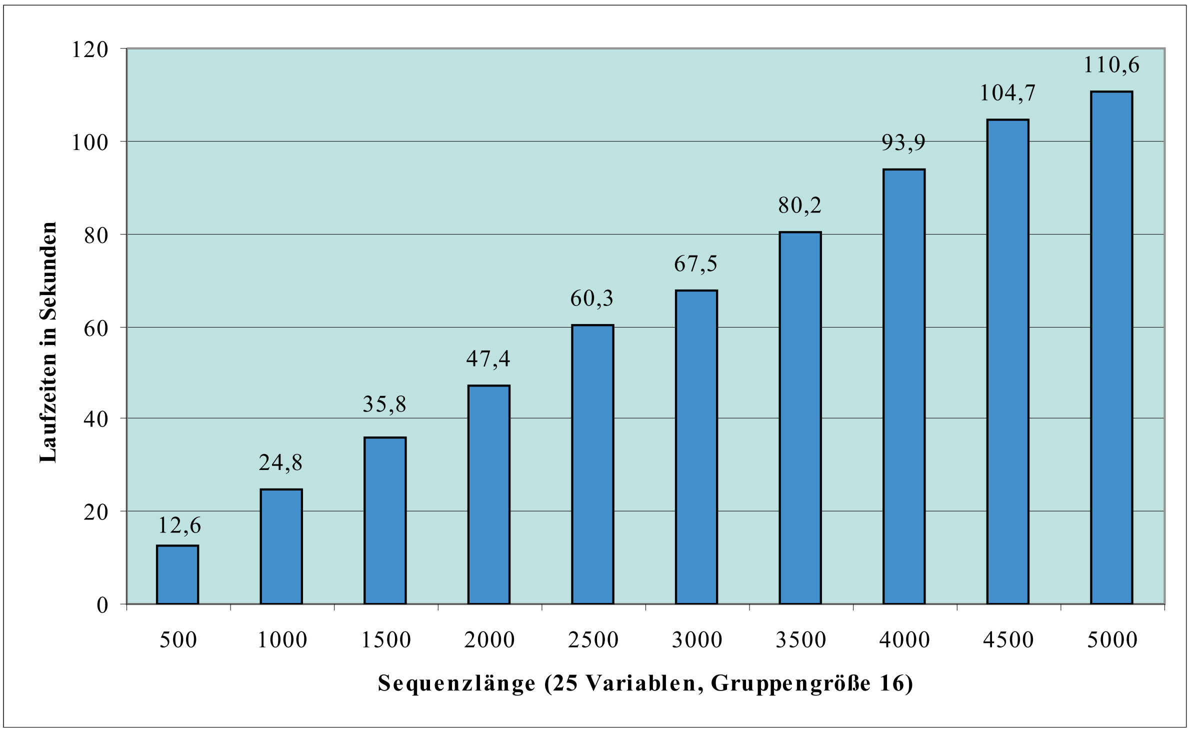

Abbildung 3.22: Laufzeitenvergleich des GAG bei verschiedenen Sequenzlängen

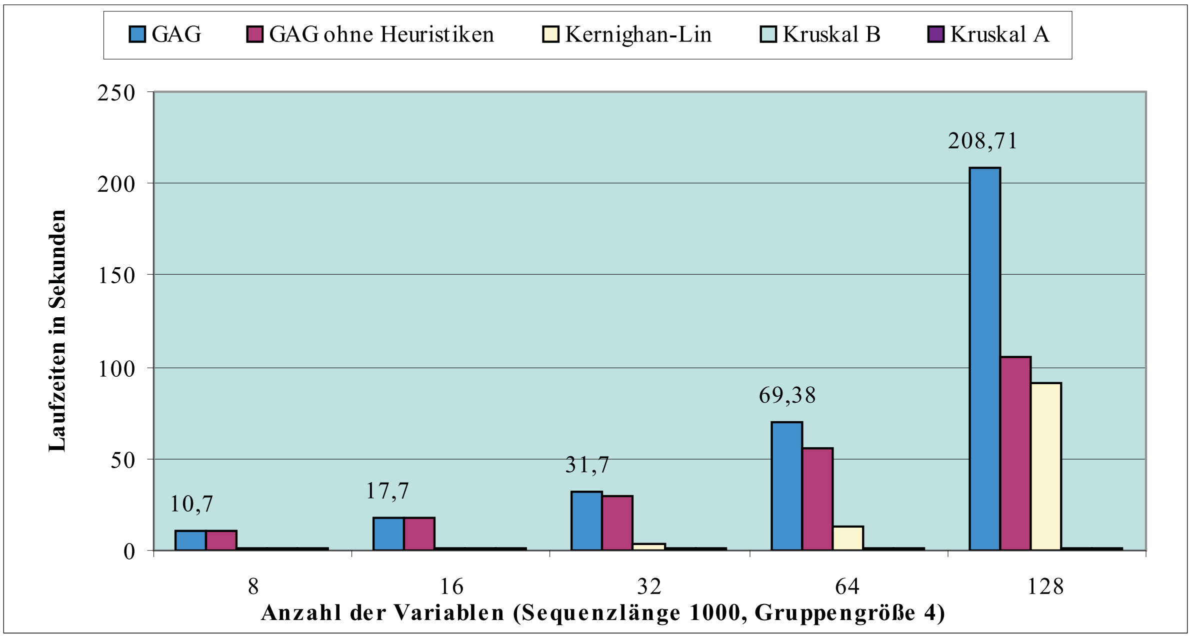

Abbildung 3.23: Laufzeitenvergleich des GAG mit und ohne heuristische Verfahren

Variablenzahlen und damit die Laufzeiten zu groß werden. Die beiden Kruskalvarianten haben eine im Vergleich zum restlichen GAG geringe Laufzeit und können auch bei großen Variablenzahlen zur Initialisierung dienen.

3.3.3 Einsatz des GAG bei Sequenzen realer Benchmarks

Die bisher verwendeten Testsequenzen können nur eine Tendenz aufzeigen, welches Einsparpotential ausgenutzt werden kann. Zufällig generierte Testsequenzen haben den Nachteil, daß Variablen über den ganzen Lauf der Sequenz auftauchen können. Dadurch werden durch die Sequenz auftretende Blöcke von Variablen mit hohen Nachbarschaftsbeziehungen, wie sie zum Beispiel durch Schleifen auftauchen können, weitestgehend vernachlässigt. Auch eine künstliche Schaffung solcher Blöcke kann reale Zugriffssequenzen nur schlecht simulieren. Der an der Universität Dortmund entwickelte Compiler für die M3-DSP Plattform gibt zwei Variablenzugriffssequenzen vor, die mit dem GAG optimiert werden. Eine erste Testreihe für Gruppengrößen von 2 bis 16 wurde mit einer Sequenz eines Basisblocks eines *IIR*[8]-Filters, die 19 Variablen bei einer Sequenzlänge von 22 enthielt, durchgeführt. Ergebnisse dieser Sequenz zeigt Abbildung 3.24. Der durchschnittliche Gewinn gegenüber der naiven Lösung liegt bei 27%.

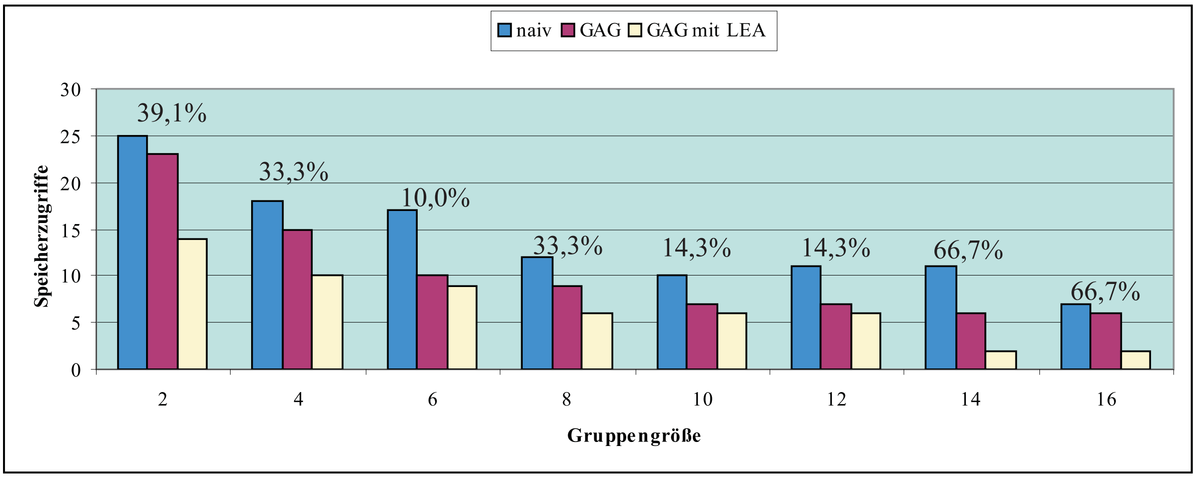

Abbildung 3.24: Ergebnisse einer Sequenz des IIR-Filters bei verschiedenen Gruppengrößen

Diese Ergebnisse werden an den Compiler zurückgegeben. Mit der ermittelten Gruppenaufteilung werden die Ergebnisse des Compilers bezüglich der bisher verwendeten Gruppenaufteilung zum Teil erheblich verbessert. Es wird auch sichtbar, daß das Ergebnis des GAG in einem Fall auch zu einer geringfügigen Verschlechterung beim M3-DSP-Compiler führt. Die muß aber nicht an einer falschen Optimierung des GAG liegen, sondern kann auch an einer Variablenzugriffssequenz liegen, die für das Gruppenbildungsproblem nicht aussagekräftig genug ist. Abbildung 3.25 zeigt die Ergebnisse bei verschiedenen Gruppengrößen. Die durchschnittliche Verbesserung gegenüber dem vorher verwendeten Spei-

[8] Infinite Impulse Response

cherlayout liegt bei ungefähr 16%. Wird zum Vergleich das naive Layout des GAG beim M3-DSP-Compiler verwendet, liegen die Einsparungen bei 3,1%.

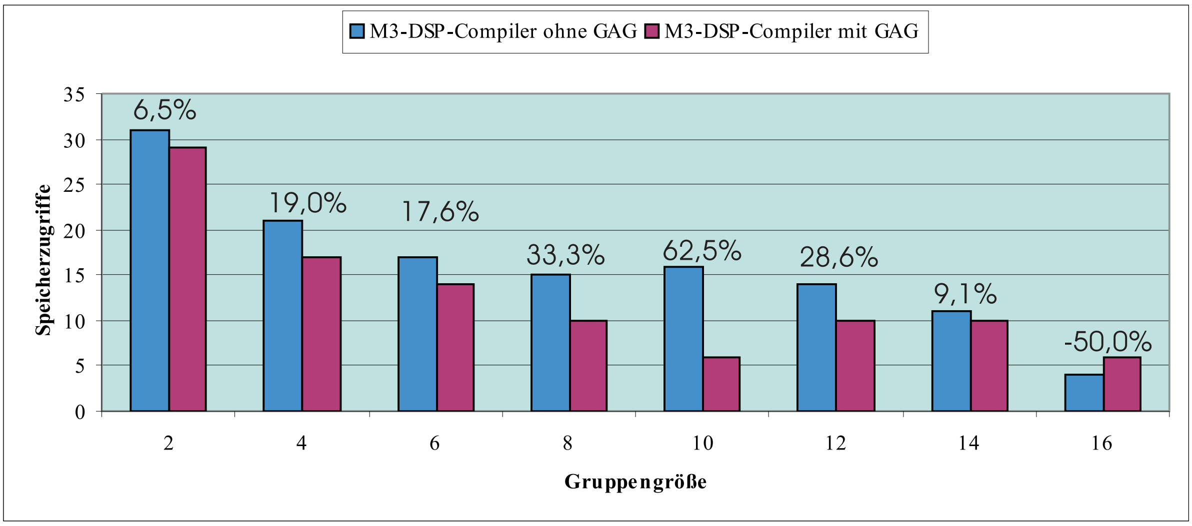

Abbildung 3.25: Ergebnisse einer Sequenz des IIR-Filters bei verschiedenen Gruppengrößen nach Einbindung in den M3-DSP-Compiler

Zusätzlich wurde eine Sequenz eines Basisblocks der *DCT*[9] vom GAG eingelesen und die Anzahl der Speicherzugriffe beim Ergebnis des Compilers für den M3-DSP vor und nach der Optimierung verglichen. Die Sequenz der DCT enthielt 437 Variablen bei einer Sequenzlänge von 524. Die Einsparungen für verschiedene Gruppengrößen zeigt Abbildung 3.26, die durchschnittliche Verbesserung liegt bei 10%. Die sich ergebende Gruppenpartitionierung wurde ebenfalls in den M3-DSP-Compiler eingelesen und ergab Einsparungen gegenüber der vorher verwendeten Gruppenpartitionierung, die in Abbildung 3.27 dargestellt sind. Die Einsparungen zum vorher verwendeten Gruppenpartitionierung sind mit durchschnittlich 61% beträchtlich. Bei einem Vergleich der vom GAG verwendeten naiven Gruppenpartitionierung mit der optimierten Gruppenpartitionierung ergeben sich immer noch durchschnittlich 8% Einsparung bei einer Verwendung beider Partitionierungen beim M3-DSP-Compiler.

3.3.4 GAG mit Initialisierung durch heuristische Verfahren

Die bisher vorgestellten Ergebnisse des GAG gründen auf einer Initialisierung mit einigen heuristischen Verfahren. Obwohl diese heuristischen Verfahren oft schlechtere Ergebnisse erzeugen als das Endresultat des GAG, wird die Konvergenz des GAG durch sie beschleunigt (Abbildung 3.28).

Bei einer Implementierung ohne heuristische Verfahren und ohne eine naive Lösung braucht der GAG länger, um sich dem Optimum von 158 zu nähern. Wird nur die naive Lösung hinzugefügt, konvergiert der Algorithmus schneller. Bei einer Initialisierung mit allen heuristischen Verfahren und der naiven Lösung findet der Algorithmus in 35% aller Fälle eine

[9] Discrete Cosine Transform

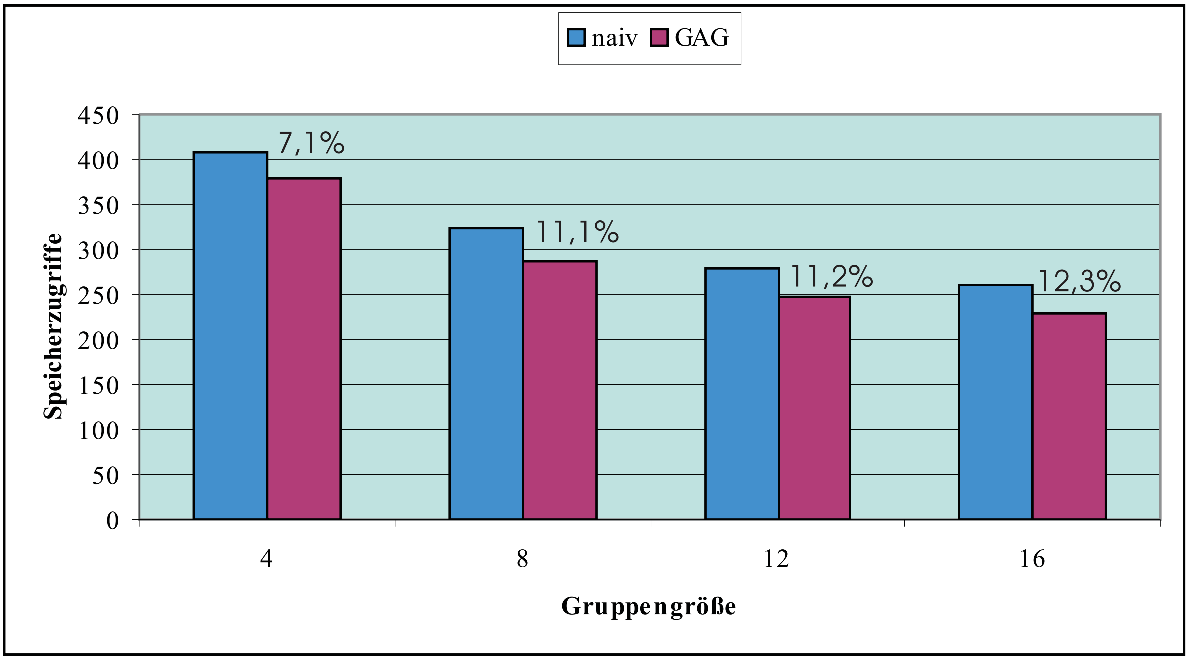

Abbildung 3.26: Ergebnisse der Sequenz der DCT bei verschiedenen Gruppengrößen

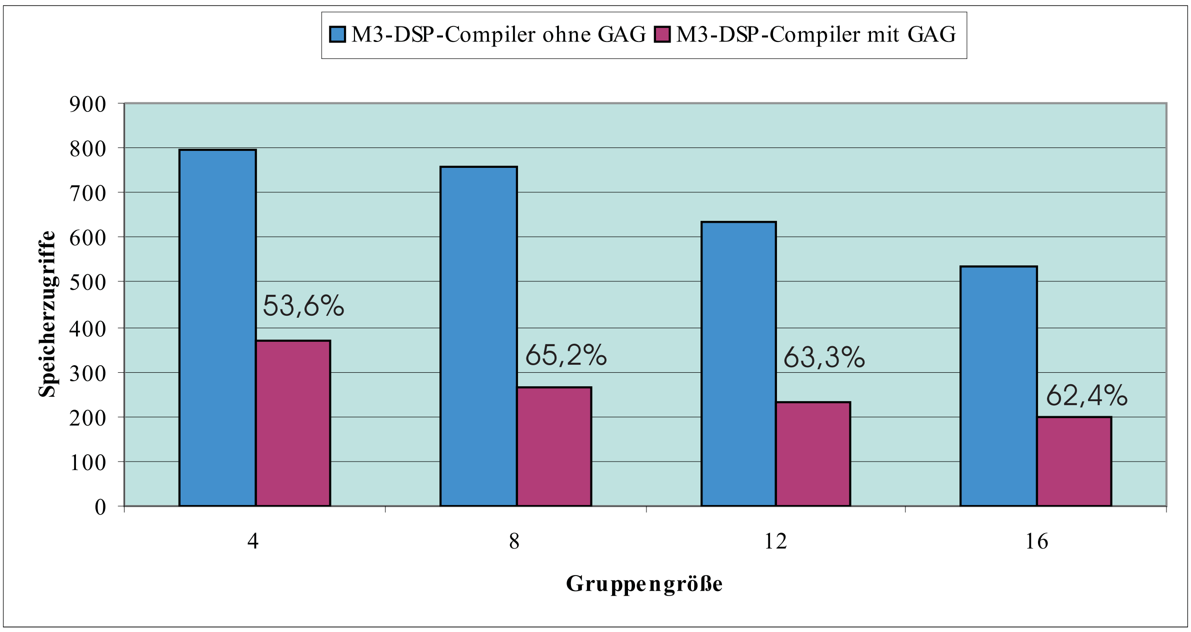

Abbildung 3.27: Ergebnisse der Sequenz der DCT bei verschiedenen Gruppengrößen nach Einbindung in den M3-DSP-Compiler

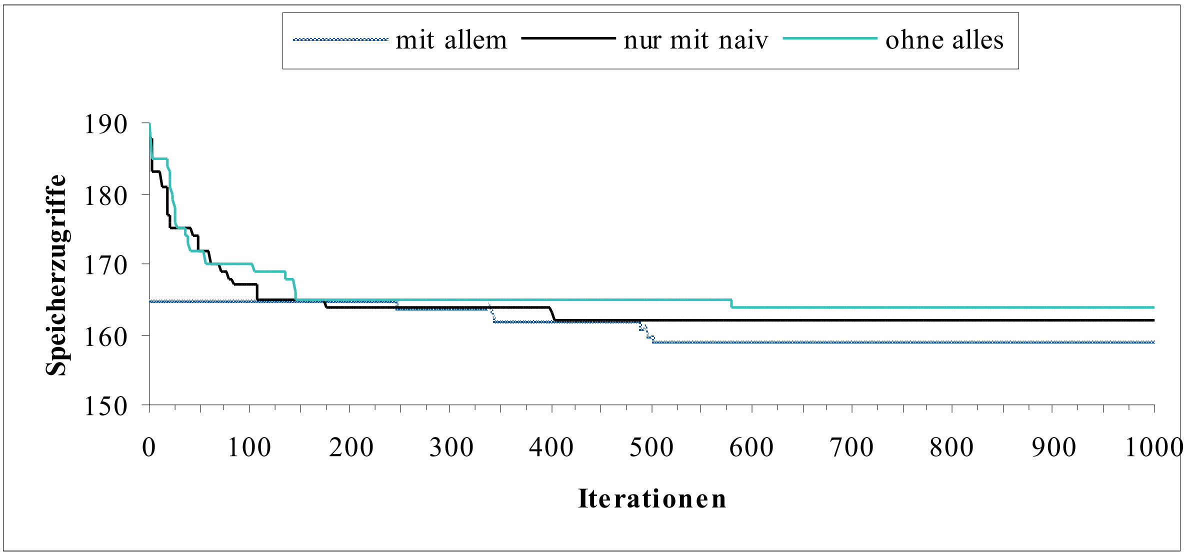

Abbildung 3.28: Konvergenz des GAG mit und ohne Initialisierung durch heuristische Verfahren

Lösung von unter 161 vor der Generation 500. Diesen Wert unterschreitet der GAG ohne eine Initialisierung mit heuristischen Verfahren nicht. Deswegen ist es sinnvoll, den GAG mit einigen heuristischen Verfahren zu initialisieren. Die Laufzeit der heuristischen Verfahren ist zudem im Vergleich zum genetischen Teil des GAG gering, die Laufzeit des Kernighan-Lin-Algorithmus trägt erst ab 100 Variablen entscheidend zur Gesamtlaufzeit bei.

3.3.5 Kombination des GAG mit einem Left-Edge-Algorithmus

Bei lokalen Variablen kann der belegte Speicherplatz freigegeben werden, wenn die entsprechende Variable zum letzten Mal benutzt worden ist. Dadurch wird nicht nur Speicherplatz eingespart, es wird auch die Zahl der Speicherzugriffe reduziert. Dies wird schon an einer kleinen Sequenz deutlich. Wird die in Abbildung 3.24 verwendete Sequenz mit Hilfe des LEA erneut bearbeitet, ergeben sich bei den betrachteten Speicherbreiten hohe Einsparungen. Die Ergebnisse zeigt Abbildung 3.29, die Prozentwerte geben dabei die Einsparungen des GAG mit LEA im Vergleich zum GAG ohne LEA an.

In diesem Fall wirkt sich der LEA besonders bei kleinen Gruppengrößen aus, da schon eine Zusammenfassung zweier Variablen viele Speicherzugriffe einspart. Da die Anzahl der Variablen mit disjunkten Lebensdauern konstant ist, kann ab einer gewissen Gruppengröße mit dem LEA keine Einsparung mehr erzielt werden. Mit dem LEA werden hier Einsparungen von durchschnittlich $34,7\%$ zum GAG ohne LEA erreicht.

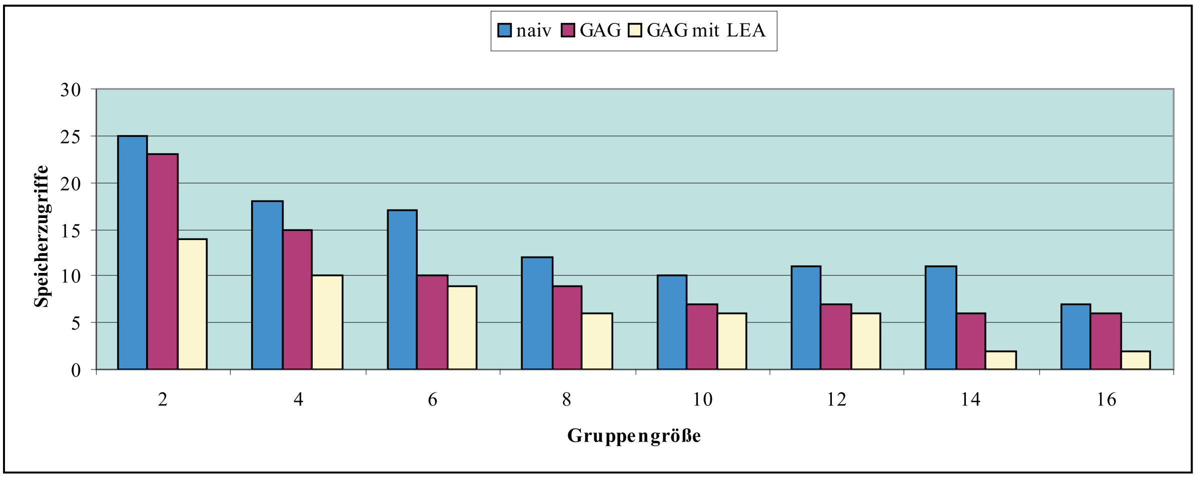

Abbildung 3.29: Vergleich des LEA mit GAG und naiver Lösung bei lokalen Variablen

3.3.6 Einfluß genetischer Parameter auf den GAG

Bei Genetischen Algorithmen können viele Parameter eingestellt werden, die Einfluß auf die Konvergenzgeschwindigkeit und damit auf die Optimierungsgüte haben. So ist es beispielsweise sinnvoll, die Mutationsrate abhängig von der Chromosomenlänge zu wählen. Bei sehr kurzen Chromosomen kann eine häufige Mutation dazu führen, daß die Ergebnisse zu stark probabilistisch werden. Beim GAG können die Mutationsrate, die Crossover-Rate und der Wert der in die nächste Generation übernommenen Individuen eingestellt werden. Auch die Populationsgröße ist variabel.

Abbildung 3.30 zeigt die Ergebnisse in Abhängigkeit von der Mutationsrate. Es zeigt sich, daß ein Wert von $1,875$ dividiert durch die Anzahl der Gene für die Mutationsrate der betrachteten Sequenzen am besten ist. Dieser Wert wird für alle Chromosomenlängen ausgewählt. An der Konvergenz in Abhängigkeit von der Mutationsrate können keine Auffälligkeiten festgestellt werden. Sie liegt für alle Mutationsraten im ungefähr gleichen Bereich von 400 bis 500 Generationen. Als bester Wert für die Crossover-Rate stellt sich $0,7$ heraus (Abbildung 3.31). Bei der Betrachtung der Konvergenz stellt sich heraus, daß bei einer Crossover-Rate von $0,7$ der GAG später als bei allen anderen Raten noch gute Ergebnisse findet. Da dies gewünscht ist und die Güte der Ergebnisse besser als bei den anderen Crossover-Raten ist, wird die Crossover-Rate auf $0,7$ festgesetzt.

Auch die Populationsgröße muß eingestellt werden. Die Ergebnisse zeigt Abbildung 3.32, es wird eine Populationsgröße von 50 gewählt. Es wird auch deutlich, daß der GAG in Abhängigkeit von der Populationsgröße deswegen bessere Ergebnisse erbringt, weil er später konvergiert. Das wird an den beiden Diagrammen in Abbildung 3.32 sehr deutlich.

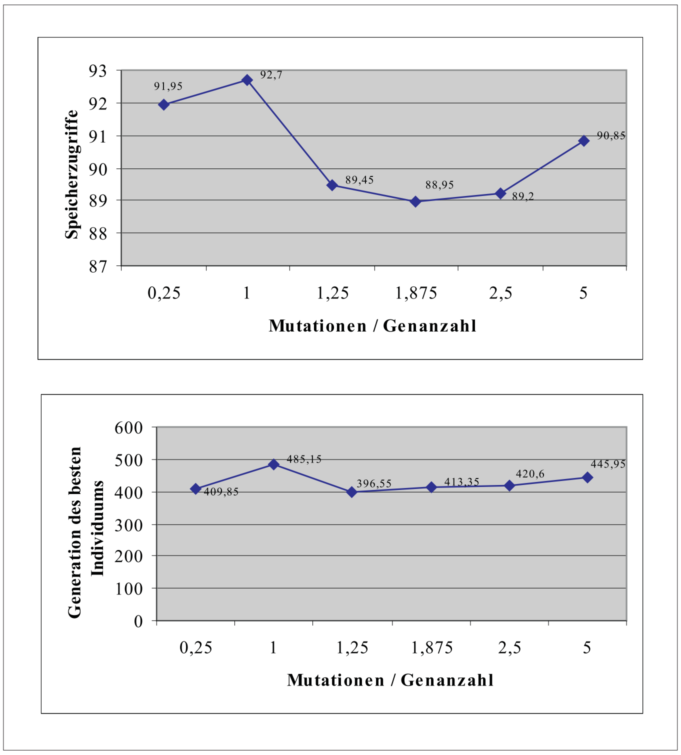

Abbildung 3.30: Einfluß der Mutationsrate auf die Güte

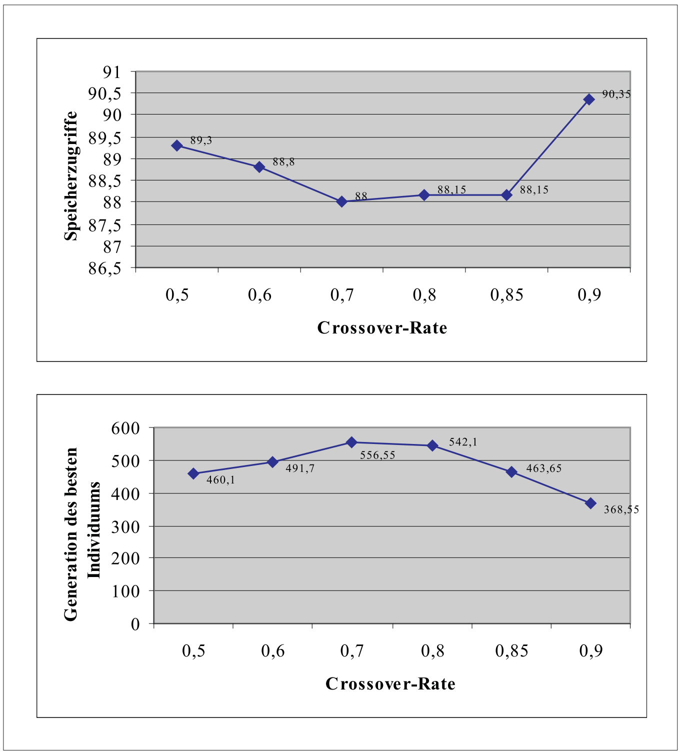

Abbildung 3.31: Einfluß der Crossover-Rate auf die Güte

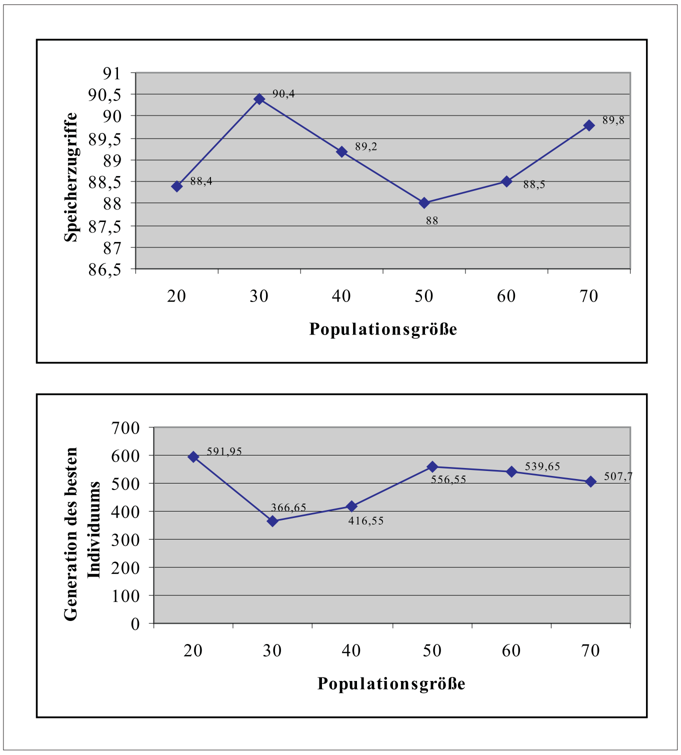

Abbildung 3.32: Einfluß der Populationsgröße auf die Güte

Die genetischen Parameter könnten auch die Laufzeit beeinflussen. Bei einer hohen Mutationsrate sind beispielsweise mehr Mutationsaufrufe vom GAG zu leisten. Es sind aber auch bei einem hundertfach größerem Mutationsfaktor keine Laufzeitsteigerungen meßbar. Nur die Populationsgröße hat einen linearen Einfluß auf die Laufzeit des GAG, weil bei mehr Individuen in Abhängigkeit von der Crossover-Rate und der Mutationsrate entsprechend mehr Operationen durchgeführt werden müssen, um eine neue Generation zu erzeugen.

Kapitel 4

Gruppenanordnung

Nachdem die Gruppenbildung abgeschlossen ist, besteht das Ziel der Gruppenanordnung (Abbildung 4.1) darin, die gebildeten Gruppen so im Speicher anzuordnen, daß möglichst wenig Adreßgenerierungswechsel[1] beim compilierten Programm vorkommen.

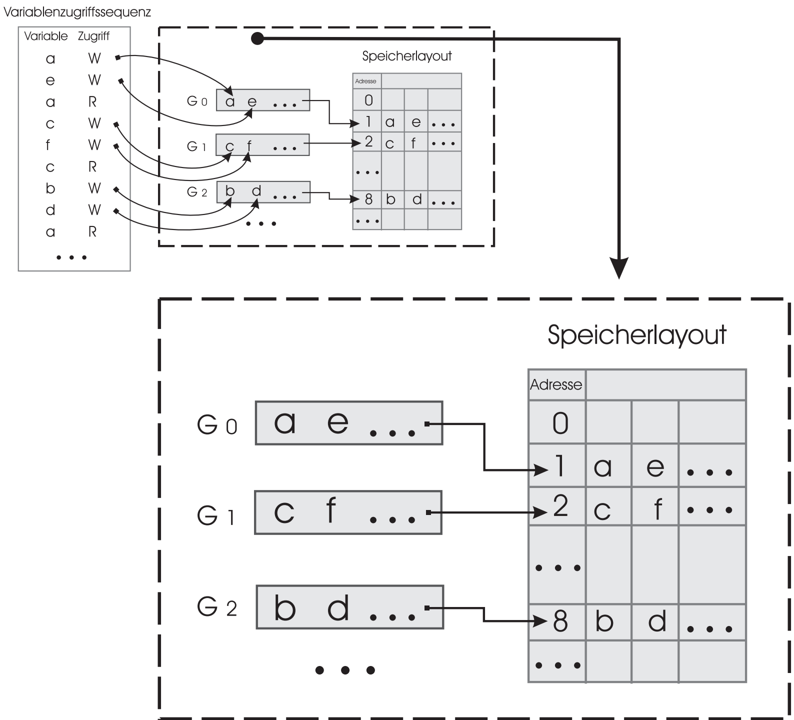

Abbildung 4.1: Gruppenanordnung

Alle Zugriffe in der Variablenzugriffssequenz auf einzelne Variablen können nach der abgeschlossenen Gruppenbildung als Gruppenzugriffe betrachtet werden, da jede Variable

[1] Ein Adreßgenerierungsbefehl besteht im weiteren Zusammenhang aus einem angesprochenen Adreßregister und dem zugehörigen Offset oder Modify-Register. Ein Wechsel ist dann die Folge zweier Adreßgenerierungsbefehle, bei denen entweder das angesprochene Adreßregister oder der verwendete Offset beziehungsweise das verwendete Modify-Register unterschiedlich sind.

zu genau einer Gruppe gehört. Abbildung 4.2 zeigt dies an einem Beispiel: Die ersten drei Zugriffe (a, R), (e, W) und (a, R) betreffen alle die Gruppe, die unter der Adresse 0 im Speicher abgelegt worden ist. Da der darauffolgende Zugriff (c, W) eine andere Gruppe betrifft, sind die Zugriffe auf die erste Gruppe zunächst beendet. Diese drei Zugriffe werden durch einen Gruppenzugriff (G_0, W) ersetzt. Das W bei einem Gruppenzugriff markiert, daß diese Gruppe sowohl gelesen als auch geschrieben werden muß, während das R nur einem Lesezugriff markiert. Der doppelte Zugriff beim Schreiben kommt im Beispiel dadurch zustande, daß der Zugriff auf e schreibend war und eine zu schreibende Gruppe unabhängig von der Existenz weiterer Zugriffe gelesen werden muß.
Entsprechend wird im weiteren Verlauf verfahren, so daß aus der Variablenzugriffssequenz eine *Gruppenzugriffssequenz* entsteht.

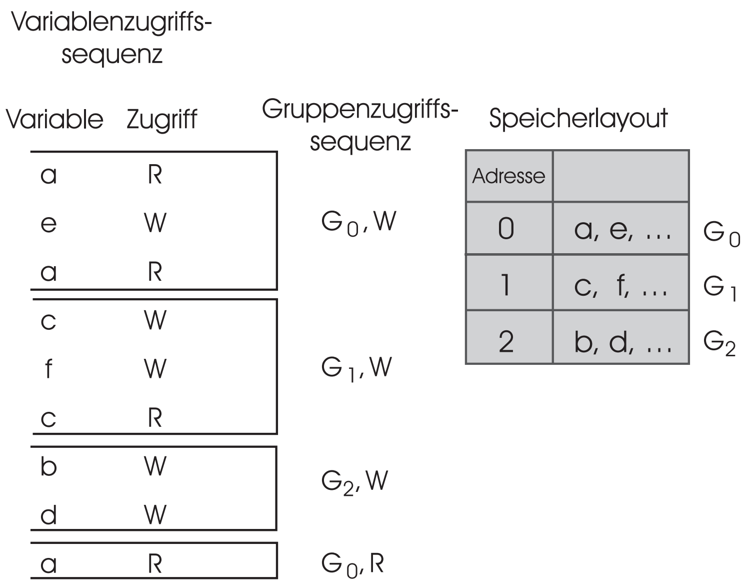

Abbildung 4.2: Entwicklung einer Gruppenzugriffssequenz

Das Optimierungskriterium der Gruppenanordnung besteht in der Minimierung der Programmgröße des compilierten Programms. Als Grundlage für diese Minimierung dient das Befehlswort des M3-DSP, das TVLIW [31]. Der M3-DSP hat eine auf dem Prinzip des VLIW aufbauende Befehlswortdefinition. Mit einem VLIW können mehrere unabhängige Funktionseinheiten parallel angesprochen werden. Dadurch können beispielsweise eine Datenmanipulation, ein Datentransfer und eine Adreßgenerierung parallel durchgeführt werden. Werden die Möglichkeiten der Parallelität, die das VLIW anbietet nur wenig genutzt, so muß bei Standard-VLIW-Architekturen trotzdem immer das ganze Befehlswort im Speicher abgelegt werden, obwohl nicht alle Befehlsteile benötigt werden. Dies erzeugt einen Codeoverhead des compilierten Programms.
Da bei den vielen Befehlen nur wenige Änderungen am VLIW vorgenommen werden müssen, besitzt der M3-DSP ein TVLIW (Abbildung 4.3). Im *TVLIW-Dekoder* werden mittels des TVLIWs Änderungen an einem im *VLIW-Cache* vorliegenden VLIW vorgenommen. Ein im VLIW-Cache bestehendes VLIW dient dabei immer als Grundlage für ein neues VLIW. Ein TVLIW enthält die Anzahl der zu verändernden Teilwörter des

zugrunde liegenden VLIWs im *Instruktionskontrollfeld*[2] *IWC* und bis zu zwei Teilwörter *FIW*[3], die mit einer Kennung (TAG) versehen sind, welche anzeigt, welches Teilwort des Ziel-VLIWs zu verändern ist. Als Ausgabe liefert der Dekoder ein vollständiges VLIW, wobei nichtbenötigte FIWs freigelassen werden[4].

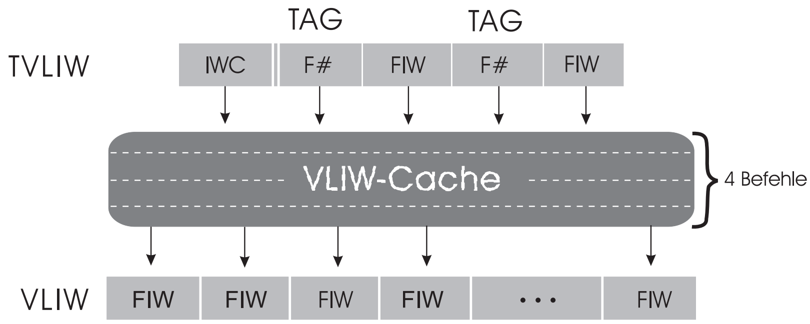

Abbildung 4.3: Tagged VLIW-Dekoder

Dadurch müssen nur die Änderungen am zugrunde liegenden Befehl gespeichert werden. Innerhalb von Schleifen, wo in der Regel eine große Parallelität von Befehlen ausgenutzt wird, müssen mit dem TVLIW-Dekoder VLIWs in mehreren Schritten verändert werden, da mit jedem Schritt maximal zwei Teilbefehle des VLIWs verändert werden können. Um diese relativ langsame Befehlsbildung auszugleichen, besitzt der M3-DSP einen VLIW-Cache, in dem vier Befehle gespeichert werden. Innerhalb von Schleifen können so Befehle wiederverwendet werden, die relativ aufwendige Bildung langer Befehle fällt nur zu Beginn der Schleife an.
Indem die Gruppen so im Speicher angeordnet werden, daß die Zahl der nötigen Wechsel von Adreßgenerierungsbefehlen minimiert wird, wird auch die Programmgröße verringert.

Ein neuer Adreßgenerierungsteil muß erst bestimmt werden, sobald mit den vorhandenen Adreßgenerierungsbefehlen aus dem VLIW-Cache die benötigte Adresse für die nächste Gruppe in der Gruppenzugriffssequenz nicht berechnet werden kann. Wenn eine solche Modifikation erfolgen muß, ist es für die Programmgröße unerheblich, welche Adreßberechnung verwendet wird, denn jeder Wechsel des Adreßgenerierungsteils verursacht den gleichen Codeoverhead. Die möglichen drei Adreßberechnungsbefehle sind im Hinblick auf das Optimierungsziel Programmgrößenreduktion aus diesem Grund gleich teuer. Allerdings unterliegen sie bestimmten Begrenzungen, die eine Auswahl des zu verwendenden Adreßberechnungsbefehls ermöglichen:

- Postmodify[5]-Adressierung mit einem Adreßregister P und einer Konstanten *const*: $\mathrm{AGU}(P_x, const)$, $x \in \{0, 1, 2, 3\}$
  Die Konstante *const* ist vier Bit breit, damit sind mit dieser Adressierungsart nur Adreßänderungen des P_x von $[-7, \cdots, 8]$ möglich.

[2]Instruction Word Control
[3]Functional Unit Instruction Word
[4]Dies geschieht durch Setzen von noops (no operations).
[5]Die Veränderung der zugrunde liegenden Adresse im Adreßregister erfolgt erst nach der Adressierung.

- Postmodify-Adressierung mit einem Adreßregister P und einem 16-Bit Modify-Register M:
  AGU(P_x, M_y), $x, y \in \{0, 1, 2, 3\}$
  Ein zu verwendendes Modify-Register muß eventuell noch mit dem gewünschten Wert initialisiert werden, was einen Extrabefehl erfordert. Es ist sinnvoll, Modify-Register erst dann einzusetzen, wenn eine Adreßänderung mit einer Konstanten nicht mehr möglich ist, weil der Offset zu der benötigten Adreßänderung zu groß ist.

- Adressierung mit dem Page-Pointer Register (PP):
  Die Adressierung wird mittels eines Offsets durchgeführt. Die letzten 6 Bit des PP-Registers werden durch den Offset bestimmt. Der Offset liegt dadurch im Bereich $[0, \ldots, 63]$ (Abbildung 4.4). Das PP neu zu setzen erfordert einen Extraschritt.

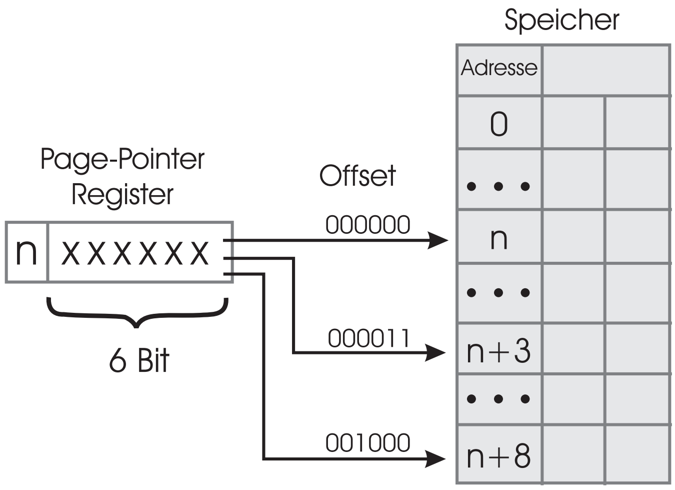

Abbildung 4.4: Page-Pointer Register

Die gegebenen Gruppen müssen im Speicher angeordnet werden. Ausgabe des Algorithmus ist eine Zuweisung der Gruppen zu Speicheradressen. Allerdings wird eine Bewertung eines vorhandenen Speicherlayouts dadurch erschwert, daß in der Adreßzuweisung sehr viele Variationsmöglichkeiten vorhanden sind:
Bis zu vier Befehlswörter werden im VLIW-Cache zwischengespeichert und können erneut benutzt werden. Programmbefehlswechsel werden eingespart, indem Befehlswörter, die im VLIW-Cache abgelegt worden sind, wiederverwendet werden. Das Problem besteht darin, den Befehl im VLIW-Cache auszuwählen, der verändert und ersetzt werden soll.

Abbildung 4.5 macht die Problematik deutlich:
Bei dem gegebenen Zustand der Adreßregister, Modify-Register, des Speicherlayouts und der vier Adreßgenerierungsbefehle im VLIW-Cache sind alle Gruppenzugriffe bis einschließlich $(n-1)$ abgearbeitet. Bei dem nun folgenden Zugriff auf die Variablen der Gruppe G_1 kann der Befehl 1 aus dem VLIW-Cache wiederverwendet werden, denn das verwendete Adreßregister P_1 enthält die richtige Adresse 0. Die ebenfalls im Befehl enthaltene Konstante 1 führt nach der Ausführung des Befehls zu einer Erhöhung der Adresse

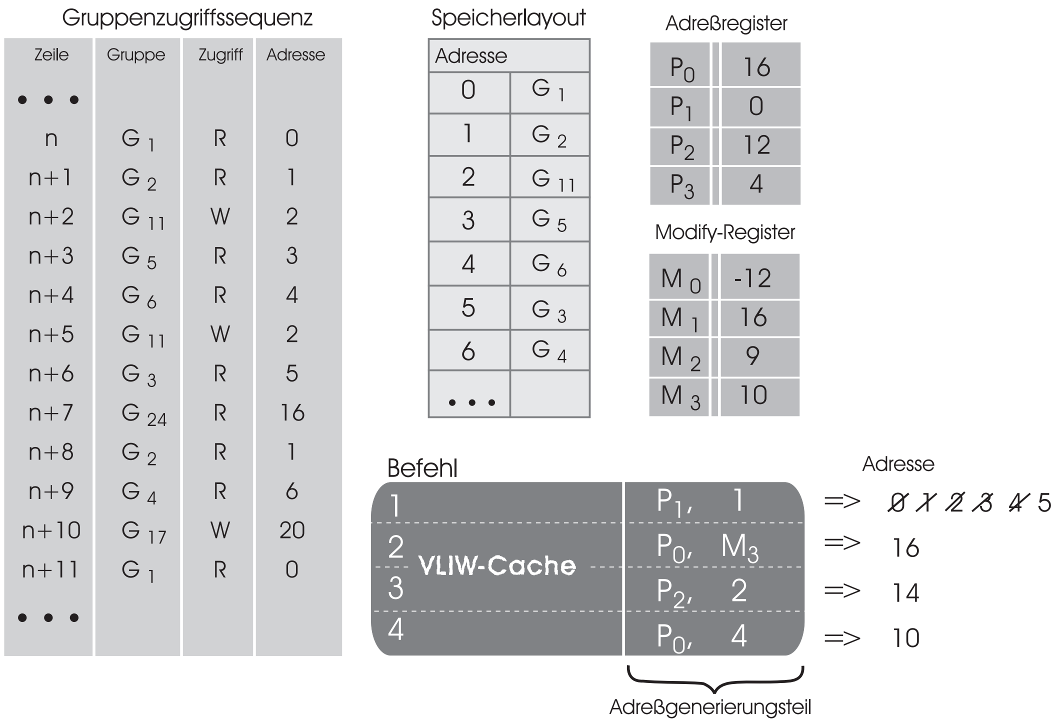

Abbildung 4.5: Zwischenzustand bei der Bewertung eines Speicherlayouts

in P_1 um 1. Damit kann auch der nächste Befehl mit diesem Adreßgenerierungsteil ausgeführt werden. Dieser Befehl AGU$(P_1, 1)$ kann noch viermal ausgeführt werden, bis der Zugriff $(n + 5)$ auf Adresse 2 ausgeführt werden muß. Keiner der nun im VLIW-Cache vorhandenen Befehle würde dazu führen, daß Adresse 2 erreicht wird. Es muß also ein neuer Adreßgenerierungsbefehl in einen VLIW geschrieben werden. Jeder der im VLIW-Cache vorhandenen Befehle könnte als Grundlage für den neuen Befehl gewählt werden, allerdings ist es beispielsweise nicht sinnvoll, den Adreßgenerierungsteil von Befehl 1 zu verändern, denn dieser kann in der vorliegenden Form noch zweimal wiederverwendet werden. Auch der Adreßgenerierungsteil von Befehl 2 kann noch mindestens einmal (bei Zugriff $(n + 7)$) verwendet werden. Somit ist es sinnvoll, die Wahl zwischen den Befehlen 3 und 4 zu treffen. Es sollte immer ein Befehl ersetzt werden, der möglichst selten wiederverwendet werden kann. Der neugenerierte Befehl sollte möglichst häufig benutzt werden können.
Müssen für die Befehlsgenerierung noch Modify-Register oder Adreßregister neu gesetzt werden, entstehen zusätzliche Befehle, da eine Änderung eines dieser Register ebenfalls mit dem Adreßgenerierungsteil vorgenommen wird. Damit wird dieser Befehl ebenfalls im VLIW-Cache abgelegt. Zudem muß vermieden werden, daß zu ähnliche Befehle im VLIW-Cache abgelegt sind, da dann nur ein kleiner Bereich des Adreßraumes ohne Wechsel der Modify-Register oder Adreßregister abgedeckt werden kann. Wenn alle Adreßgenerierungsteile beispielsweise positive Offsets haben, muß unter Umständen schneller ein Adreßregister neu gesetzt werden, um die ersten Adressen des benutzten Speichers zu erreichen.
Im vorliegenden Beispiel würde vielleicht der Befehl 4 durch AGU$(P_0, -1)$ ersetzt werden, nachdem P_0 auf 2 gesetzt worden ist.

Um das Problem eines guten Speicherlayouts, das wenig Wechsel der Adreßgenerierungsbefehle verursacht, zu lösen, wird von einigen Annahmen ausgegangen:

- Die betrachtete Variablenzugriffssequenz wird als ausgegebener Adreßgenerierungsteil der Befehlsfolge des Programms verstanden. Jeder Variablenzugriff wird also interpretiert, als würde er parallel zu genau einem Befehl des Programms ausgeführt und als würden keine Befehle des Programms ohne den Adreßgenerierungsteil noop vorkommen.
- Muß ein neuer Adreßgenerierungsteil gebildet werden, entscheidet die Adreßzuweisung, welcher Adreßgenerierungsteil im VLIW-Cache als Grundlage für diesen neuen Befehlsteil gewählt wird.

Diese Annahmen verursachen einige Ungenauigkeiten. In der Adreßzuweisung ist nicht bekannt, welche Befehle im VLIW-Cache vorhanden sind. Selbst wenn der Adreßberechnungsteil bekannt wäre, ist bei einer gegebenen Wiederverwendungsmöglichkeit nicht gesagt, daß dieser Befehl auch tatsächlich verwendet wird, da die restlichen Befehlsteile des VLIWs so an dieser Stelle des Programms nicht ausgeführt werden können. Da die Adreßberechnung aus dem restlichen Compilierprozeß herausgelöst betrachtet wird, ergibt sich ein weiteres Problem: Die Variablenzugriffssequenz enthält nur Zugriffe auf den Speicher. Die dazugehörigen Befehle machen aber unter Umständen nur einen Bruchteil der Gesamtbefehle des Maschinenprogramms aus. In Teilen des Programms werden die Variablen nur in Registern bearbeitet und auf den Speicher gar nicht zugegriffen. Abbildung 4.6 stellt dies dar:

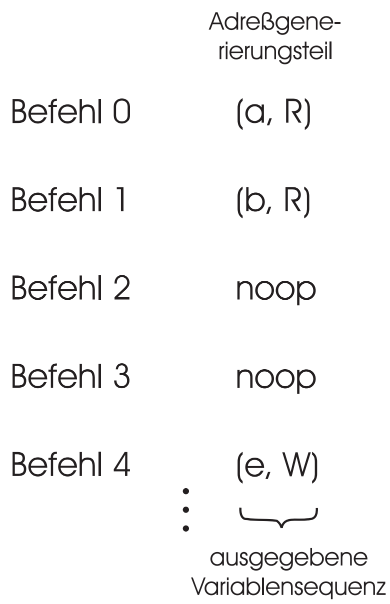

Abbildung 4.6: Ermittlung der Variablenzugriffssequenz in der Codegenerierung

Neben den Datenmanipulationen und sonstigen Befehlsteilen wird beim ersten Befehl auf eine Variable a zugegriffen, im zweiten Befehl auf eine Variable b. Da diese Variablen gelesen werden müssen, werden sie auch in der Variablenzugriffssequenz ausgegeben. In

den beiden folgenden Befehlen 2 und 3 wird mit bereits in Registern vorliegenden Variablen gearbeitet, der Adreßgenerierungsteil enthält ein noop. Da die Adreßgenerierung aber nur die Variablenzugriffssequenz erhält, scheint es so, als würden nur drei Befehle abgearbeitet, die Befehle 0, 1 und 4. Der Adreßgenerierungsteil besteht in dieser Abbildung nur aus Variablenzugriffen als Platzhalter für die Adressen, da die Adressen in der Adreßgenerierung erst ermittelt werden.
Es kann also vorkommen, daß zwischen zwei Zugriffen auf Variablen der komplette VLIW-Cache überschrieben wird und alle Adreßgenerierungsteile mit noops belegt sind.

Zu diesem Zeitpunkt der Codegenerierung ist nicht klar, welche Befehle während der Laufzeit im VLIW-Cache vorhanden sind, denn der Adreßgenerierungsteil ist nur ein Teilwort des Gesamtbefehlswortes. Die letztendliche Entscheidung, welcher Befehl jeweils als Grundlage dient, wird erst später getroffen. Dieses Verfahren macht allerdings dennoch Sinn, da es ebenfalls dort eingesetzt werden kann, wo diese Entscheidung getroffen wird. Eventuell sollte deswegen überlegt werden, dieses Verfahren auch erst später auszuführen. Der Genetische Algorithmus zur Gruppenanordnung baut auf schon bekannten Verfahren auf, die im folgenden vorgestellt werden.

4.1 Verfahren zur Adreßzuweisung

Die Optimierung von Speicherlayouts für DSPs ist ein noch relativ neues Forschungsgebiet. Infolgedessen existieren noch nicht besonders viele Veröffentlichungen auf diesem Gebiet. Die hier betrachteten Verfahren versuchen, anhand eines günstigen Speicherlayouts die Benutzung parallel ausführbarer automodify-Befehle zu maximieren. Da aber einige Verfahren ähnliche Problemstellungen wie das in dieser Diplomarbeit vorliegende Gruppenanordnungsproblem aufweisen, macht es Sinn, diese Verfahren zu betrachten.

4.1.1 Simple Offset Assignment-Problem

Das Simple Offset Assignment-Problem (SOA) besteht darin, aus einer gegebenen Variablenmenge und einer Variablenzugriffssequenz jeder Variable so eine Adresse zuzuweisen, daß die kostenfreien postmodify-Berechnungen (Offset $\in \{-1, +1\}$) von Adressen möglichst gut ausgenutzt werden. Es steht nur ein Adreßregister zur Verfügung. Das Ziel besteht darin, die zusätzlich erforderlichen Schritte für eine Berechnung der nächsten Adresse zu minimieren, also möglichst viele Adreßberechnungen parallel zu anderen Befehlen auszuführen.
Bartley [2] schlägt zur Lösung dieses Problems einen Graphalgorithmus vor. In dem Graphen stellen die Knoten die Variablen dar. Kanten werden nur zwischen Knoten eingefügt, die eine Nachbarschaftsbeziehung besitzen, die Kantengewichte geben dabei die Anzahl dieser Nachbarschaftsbeziehungen der Variablen in der Sequenz an (siehe auch Kapitel 3.1).
Gesucht ist nun der maximal gewichtete *Hamilton-Pfad*. Ein Hamilton-Pfad ist ein Pfad im Graphen, der jeden Knoten genau einmal enthält. Jeder Hamilton-Pfad stellt ein gültiges Speicherlayout dar. Das Finden eines optimalen Speicherlayouts entspricht also der

Suche nach dem maximal gewichteten Hamilton-Pfad, welcher die meisten automodify-Operationen erlaubt [2]. Da dieses Problem NP-hart ist, verwendet Bartley ein heuristisches Verfahren. Dieses Verfahren basiert auf dem Kruskal-Algorithmus (Kapitel 3.1.3) und läuft in zwei Schritten ab:
Im ersten Schritt werden alle Kanten anhand ihres Gewichtes in absteigender Reihenfolge sortiert. Danach wird der leere Pfad P gegründet. Es werden solange in absteigender Reihenfolge Kanten des Graphen dem Pfad P hinzugefügt, bis P alle Knoten enthält. Dabei werden nur solche Kanten hinzugefügt, die weder Bäume noch Zyklen in P produzieren.

4.1.2 General Offset Assignment-Problem

Die Beschränkung auf ein Adreßregister ist bei den meisten DSPs, die mit zwei oder mehr Adreßregistern arbeiten, nicht sinnvoll. Beim General Offset Assignment-Problem (GOA) wird die Anzahl der Adreßregister auf eine beliebige, aber feste Anzahl erweitert. Ansonsten liegt das gleiche Problem wie beim SOA vor: Den Variablen Adressen zuzuweisen.

Liao et al. [21] schlagen einen Algorithmus vor, für den sie von zwei Annahmen ausgehen:

- Jedes benutzte Adreßregister verursacht Kosten für die Initialisierung. Selbst wenn später mit diesem Adreßregister nur automodify-Operationen durchgeführt werden, muß der Startwert in einem Extraschritt geladen werden.
- Jede Variable wird genau einem Adreßregister zugeteilt und kann ausschließlich mit diesem Adreßregister adressiert werden.

Mit diesen Annahmen ist das GOA-Problem ein Partitionierungsproblem: Gesucht ist eine Partitionierung der vorhandenen Variablen auf die k Adreßregister. Für k Adreßregister werden genau k Optimierungs-Durchläufe zur Lösung des SOA-Problems benötigt; bei den SOA-Algorithmen besteht das Ziel darin, die Summe der Kosten für das einzelne Adreßregister mit der zu diesem Adreßregister gegebenen Variablenmenge zu minimieren. Liao et al. stellten fest, daß das Hauptproblem in einer guten Partitionierung der Variablen besteht. Sie geben an, daß eine geeignete Partitionierungsgröße von Problem zu Problem unterschiedlich ist und nur durch Tests ermittelt werden kann.

Leupers und Marwedel stellen in [19] eine Verfeinerung dieses Algorithmus vor, indem sie ein Verfahren zur Suche einer solchen Anfangspartitionierung angeben:
Nach der Bildung eines Graphen, in dem die Knoten die Variablen repräsentieren und die Kantengewichte die Nachbarschaftsbeziehungen in der Zugriffssequenz, werden die Kanten anhand ihres Gewichts sortiert. Für k Adreßregister werden maximal k disjunkte Paare von Variablen, zwischen denen die Kante das jeweils beste, positive Gewicht aufweist, auf gleiche Adreßregister verteilt und in diesem Schritt nicht mehr betrachtet. Sind weniger als k solcher Kanten im Graphen vorhanden, werden nur diese betrachtet. Die restlichen Variablen werden testweise allen Adreßregistern zugewiesen und jeweils die Zuteilung ermittelt, die die geringste Erhöhung der Kosten bei einem SOA-Algorithmus mit dieser Zuteilung verursacht. Für jede Kostenberechnung muß also ein SOA-Algorithmus herangezogen werden.

Da es sich um heuristische Verfahren handelt, erzeugen alle vorgestellten Verfahren unter Umständen nur suboptimale Lösungen für das SOA-Problem und das GOA-Problem.

4.1.3 Modify-Register-Belegung

Um ein Modify-Register einzusetzen, schlägt Leupers für DSPs allgemein in seiner Dissertation [16] vor, einen zweistufigen Algorithmus durchzuführen:
Im ersten Schritt wird ein gutes Layout ohne Berücksichtigung des Modify-Registers mit Hilfe der SOA- und GOA-Algorithmen erzeugt.
In zweiten Schritt wird durch eine gute Belegung des Modify-Registers versucht, die Kosten weiter zu senken. Dazu wird die Anzahl der vorkommenden Differenzen der Adressen, den Sprüngen im Speicher, zwischen zwei Zugriffen auf Variablen untersucht. Sprünge, die häufiger vorkommen, können in das Modify-Register geladen werden und erfordern keinen Extraschritt bei einer Adressierung, sondern können parallel zur Modifikation des Adreßregisters eingesetzt werden. Allerdings fallen Kosten für die Belegung des Modify-Registers an.

Der M3-DSP besitzt neben vier Adreßregistern auch vier Modify-Register, die orthogonal mit den Adreßregistern verwendet werden können. Deswegen muß der vorgestellte Algorithmus erweitert werden:
Das zu lösende Problem besteht darin, welches Modify-Register während des Laufs durch die Variablenzugriffssequenz mit einem neuen Wert belegt wird.

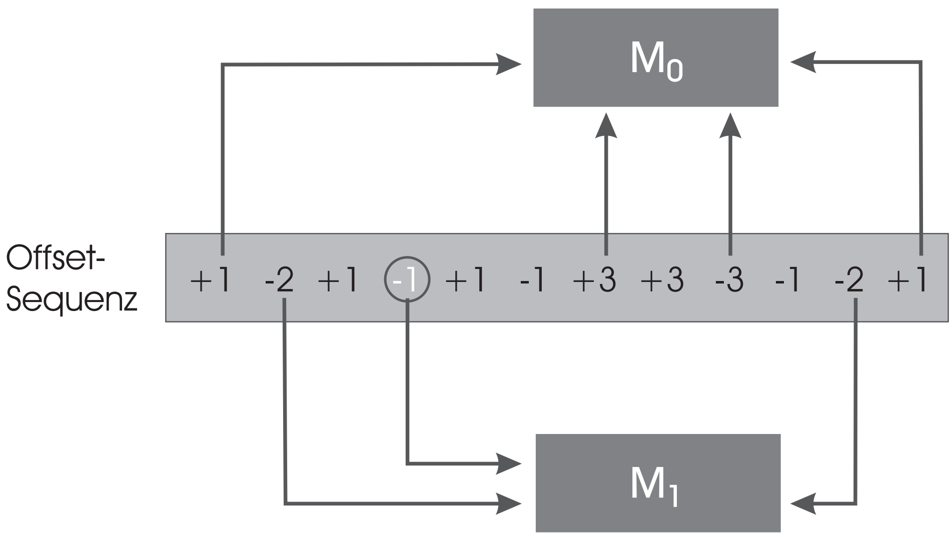

Abbildung 4.7: Neubelegung der Modify-Register mit Offsets

Die Lösung geht auf einen Algorithmus von Belady [3] zur optimalen Lösung der Ein- und Auslagerung von Speicherseiten in den Hauptspeicher zurück, wenn die Zugriffssequenz bekannt ist. Im Gegensatz zu Speicherseiten, bei denen die Zugriffssequenz nicht bekannt ist, ist sie das bei diesem Problem schon.
Wird ein Punkt in der Sequenz erreicht, an dem kein Modify-Register mehr frei ist, aber ein neuer Wert zugewiesen werden muß, wird derjenige Wert überschrieben, dessen nächste Nutzung am weitesten in der Zukunft liegt.

Abbildung 4.7 zeigt eine Offset-Sequenz, also eine Sequenz von Sprüngen im Speicher. Bei zwei vorhandenen Modify-Registern M_0, M_1 taucht das erste Mal ein Entscheidungsproblem auf, wenn der Offset -1 anliegt. Nun muß entschieden werden, welcher der Offsets

$+1$ und -2 am weitesten in der Zukunft wieder benötigt wird. Dies ist der Offset -2, der sich in M_1 befindet und daraufhin vom Offset -1 überschrieben wird.

4.1.4 Genetisches GOA

Die heuristischen Verfahren, die für die Lösung der Probleme SOA und GOA entwickelt wurden, ermitteln eventuell suboptimale Ergebnisse. Fabian David setzt in seiner Diplomarbeit [8] auf genetische Verfahren für die Lösung beider Probleme. Die entwickelten genetischen Verfahren ermitteln oft das optimale Ergebnis, wenn sie vorher mit einer der obigen heuristischen Verfahren initialisiert werden, und liegen im Durchschnitt sehr nah an einer optimalen Lösung. Das verwendete genetische Verfahren kann mit einer variablen Anzahl von Adreß- und Modify-Registern arbeiten.

Da das Gruppenanordnungsproblem unter anderem auch darin besteht, Gruppen in der Gruppenzugriffssequenz Adreßregister zuzuweisen, von denen vier vorhanden sind, enthält das Gruppenanordnungsproblem als Teilproblem das GOA und ist infolgedessen NP-hart.

4.2 Genetischer Algorithmus zur Gruppenanordnung (GAA)

Der Genetische Algorithmus zur Gruppenanordnung (GAA) soll eine Anordnung der Gruppen im Speicher ermitteln, die mit möglichst wenig Befehlswechseln auskommt.
Als Eingabe erhält der GAA eine Gruppenmenge, die sich aus der Menge der Variablengruppenzugriffe der Variablenzugriffssequenz und den Gruppen aus dem GAG ergibt (Abbildung 4.8). Dabei werden die aus dem GAG ermittelten Gruppen und die schon in der Variablenzugriffssequenz vorhandenen Gruppen mittels GAA im Speicher angeordnet und als Speicherlayout ausgegeben. Dazu wird jeder Variable genau eine *Speicherstelle*, die aus Adreßzeile und der Position innerhalb der zugehörigen Speicherzeile besteht, zugewiesen.

Der Algorithmus geht von folgenden Voraussetzungen aus:

- Die Zusammensetzung der vorhandenen Gruppen kann nicht mehr verändert werden. Damit können die verschiedenen Gruppen prinzipiell wie Variablen behandelt werden, die unter einer Adresse abgespeichert sind.
- Gleiche Adreßbefehlsteile im VLIW-Cache können wiederverwendet werden.
- Der Algorithmus hat die Kontrolle darüber, welcher Befehl im VLIW-Cache ausgewählt wird, um verändert zu werden.

Beim GAA handelt es sich um einen Genetischen Algorithmus, bei dem in der Bewertungsfunktion ein heuristisches Verfahren die Befehlswechsel ermittelt. Da vier Modify-Register, vier Adreßregister und ein VLIW-Cache mit vier Befehlen zur Verfügung stehen,

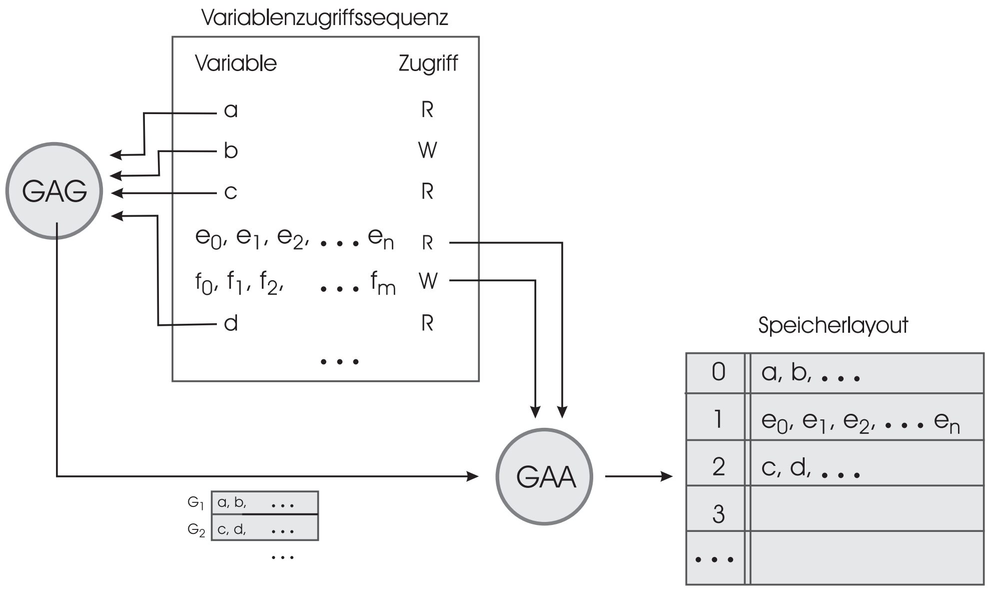

Abbildung 4.8: Eingabe und Ausgabe des Genetischen Algorithmus zur Gruppenanordnung, GAA

wobei jeder Adreßgenerierungsteil dieser Befehle aus drei verschiedenen Adressierungsarten gewählt werden kann, ist eine einfache, optimale Berechnung in polynomieller Zeit nicht möglich.
Das Problem kann auch als Graphproblem betrachtet werden. Die wie in Abbildung 4.9 ermittelte Gruppenzugriffssequenz gibt an, in welcher Reihenfolge auf die Gruppen zugegriffen wird.

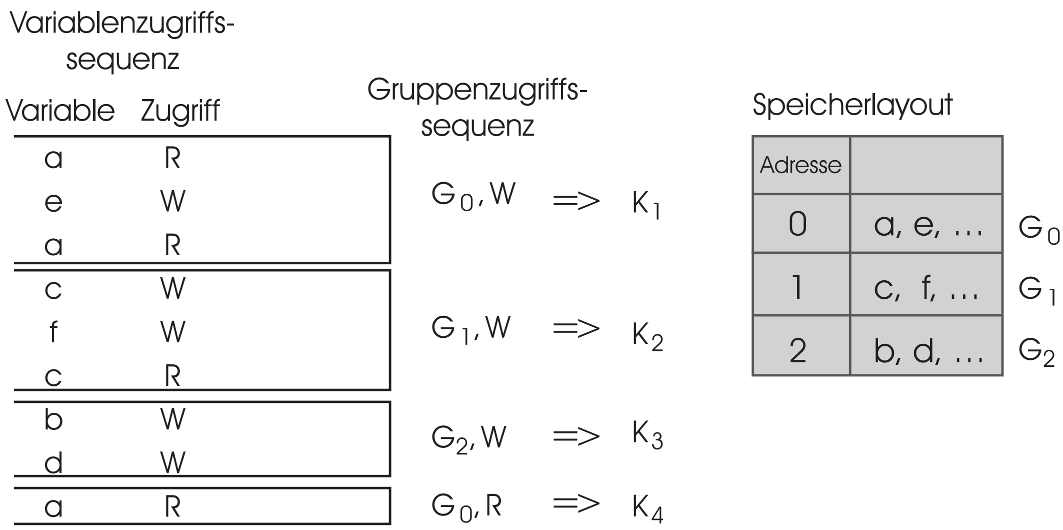

Abbildung 4.9: Bildung von Graphknoten für das bewertende heuristische Verfahren

Die einzelnen Elemente der Gruppenzugriffssequenz ergeben die Knoten K eines Graphen (Abbildung 4.9). Die Knoten sind in der Reihenfolge ihres Auftretens in der Sequenz numeriert. Nun werden von jedem Knoten K Kanten zu allen Knoten gezogen, die eine größere

Nummer haben. Das ganzzahlige Kantengewicht entspricht der nötigen Adreßänderung bei einem Adreßwechsel zu dieser in der Sequenz weiter hinten liegenden Gruppe eines Knotens. Damit ergibt sich ein Graph wie in Abbildung 4.10. Das Ziel besteht darin, jeden Knoten als bearbeitet zu markieren.

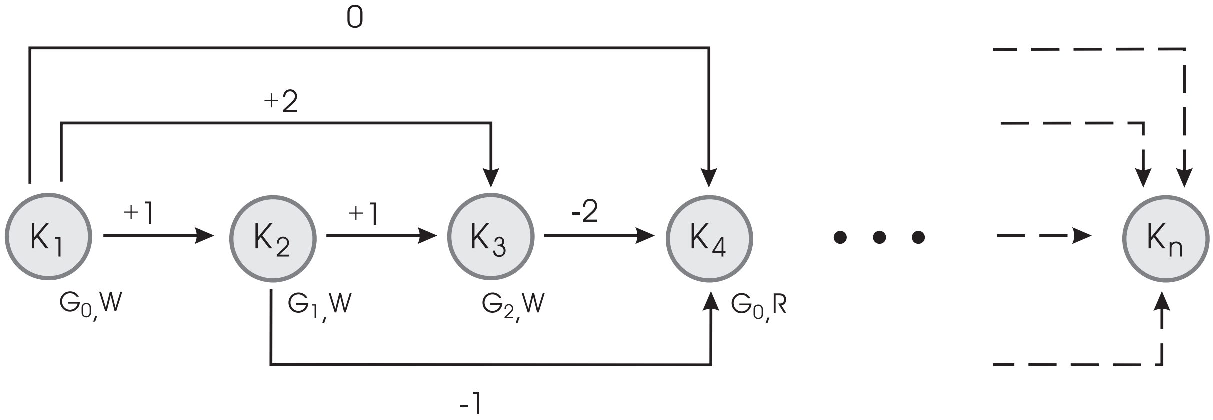

Abbildung 4.10: Beispielgraph für die Bewertung eines Speicherlayouts

Der Adreßgenerierungsteil eines Befehls besteht aus zwei Teilen: Einem verwendeten Adreßregister und einem Offset, mit dem das Adreßregister nach der Adressierung modifiziert wird. Im vorliegenden Graphen (Abbildung 4.10) adressiert ein Adreßgenerierungsbefehl genau dann die mit einem Knoten verbundene Gruppe, wenn die Adressen dieser Gruppe und des Adreßregisters übereinstimmen. Stimmt der Offset des Befehlsteils mit dem Kantengewicht einer von diesem Knoten ausgehenden Kante überein, kann auch die mit dieser Kante erreichte Gruppe im nächsten Schritt von diesem Befehl erreicht werden. Im Beispielgraphen würde zum Beispiel ein Befehlsteil von $\mathrm{AGU}(P_0, 2)$ mit $P_0 = 0$ die Gruppe des ersten Knotens adressieren, wenn G_0 an der Adresse 0 steht, und danach die Adresse von P_0 so verändern, daß beim übernächsten Zugriff auf die Gruppe G_2 im Knoten K_3 die Adresse von P_0 mit der benötigten Adresse von K_3 übereinstimmt. Diese beiden Knoten K_1, K_3 werden also mit einem Befehl bearbeitet. Das Ziel besteht nun darin, alle mit den Knoten des Graphen verbundenen Gruppen mit möglichst wenigen Befehlen zu adressieren. Werden nur vier verschiedene Befehle benötigt, um alle Knoten des Graphen zu bearbeiten, gäbe es nur vier Adreßwechsel, da jeder Befehl einmal im VLIW-Cache abgelegt werden müßte. Werden mehr als vier Befehle benötigt, besteht die Aufgabe des GAA darin, durch eine gute Anordnung der Gruppen im Speicher die Wechsel der Adreßgenerierungsteile im VLIW-Cache zu minimieren.

In der Bewertungsfunktion wird beim Genetischen Algorithmus mit Hilfe des oben beschriebenen Graphen ermittelt, wieviele Wechsel des Adreßgenerierungsteils nötig sind. Zunächst jedoch wird die Initialisierung des GAA vorgestellt.

4.2.1 Initialisierung

Nach Ausführung des GAG existiert eine feste Gruppenmenge (Abbildung 4.8). Die Gruppen dieser Menge werden nicht mehr verändert. Die Kodierung des GAA ist stark an die

Kodierung des GAG angelehnt. Dadurch kann eine spätere Kombination beider Verfahren GAG und GAA ermöglicht werden. Auch eine eventuelle erneute Aufteilung der Gruppen kann dadurch wieder ermöglicht werden, falls dies wünschenswert sein sollte (siehe Seite 14). Die Initialpopulation des GAA erhält zunächst die beste ermittelte Lösung des GAG, erweitert um die Gruppenzugriffe aus der Variablenzugriffssequenz. Die Allele der Gene entsprechen nun Adressen im Speicher.

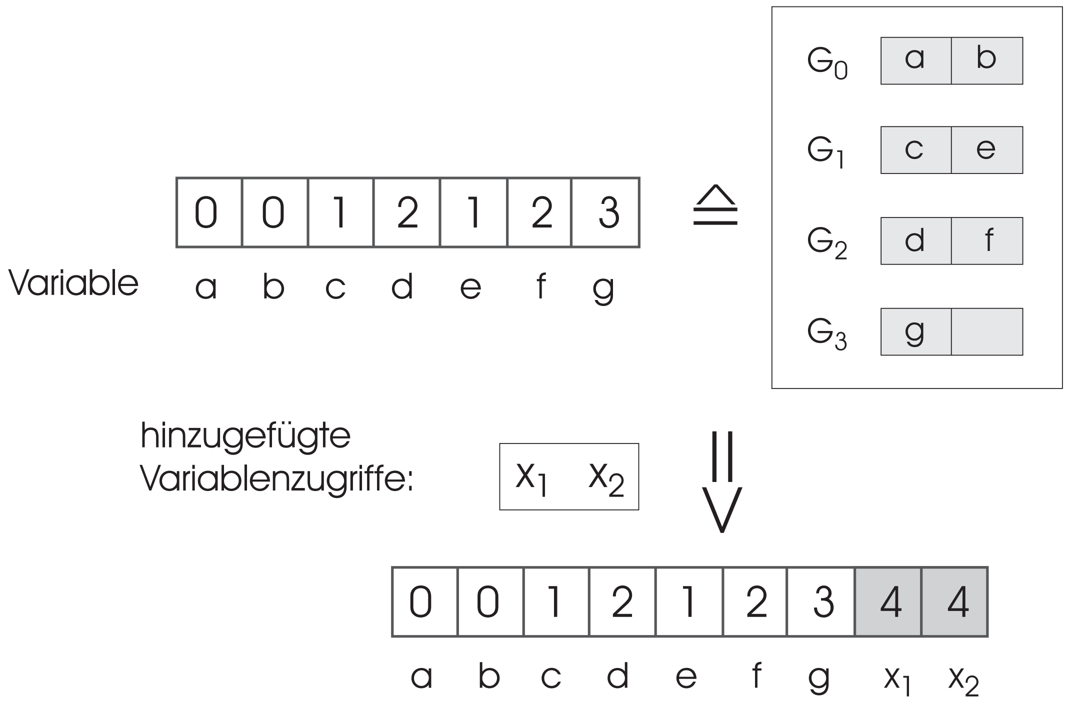

Abbildung 4.11: Die Kodierung beim GAA

Abbildung 4.11 zeigt die Kodierung an einem Beispiel: Das Chromosom hat wieder die Länge n, was der Anzahl der Variablen aus dem GAG entspricht. Die Belegung der Gene dieses Chromosoms kodiert eine bestimmte Gruppenaufteilung. Angehängt werden noch die vorhandenen Zugriffe von Variablen, die schon im Gruppenmodus in der Variablenzugriffssequenz vorliegen. Dadurch verlängert sich entsprechend das Chromosom. Diese Gruppen erhalten neue Gruppennummern, im Beispiel werden die Gene der Gruppe (x_1, x_2) mit dem Allel 4 belegt.
Das beste Chromosom des GAG entspricht dem Ausgangschromosom aus Abbildung 4.12. Anhand dieses Chromosoms werden nun die anderen Chromosomen der Initialpopulation gebildet. Die Gruppenzusammensetzung darf dabei nicht verändert werden. Wird einer Gruppe eine neue Adresse zugewiesen, so müssen alle Gene der Gruppe mit dieser Adresse neu belegt werden. Eine eventuell mit diesem Wert schon belegte Gengruppe wird auf einen neuen Wert gesetzt. Dabei sind leere Gruppen zwischen den gewählten Werten erlaubt. Dies kann zu einer besseren Ausnutzung der Adreßgenerierungs-Befehle führen. Im Tool kann angegeben werden, wie groß diese Lücken zwischen gefüllten Gruppen sein dürfen. Es kann aus Speicherplatzgründen auch ganz unterdrückt werden, daß leere Gruppen[6] im Verlauf des GAA entstehen.

[6] Leere Gruppen entsprechen freien Zeilen im Speicher.

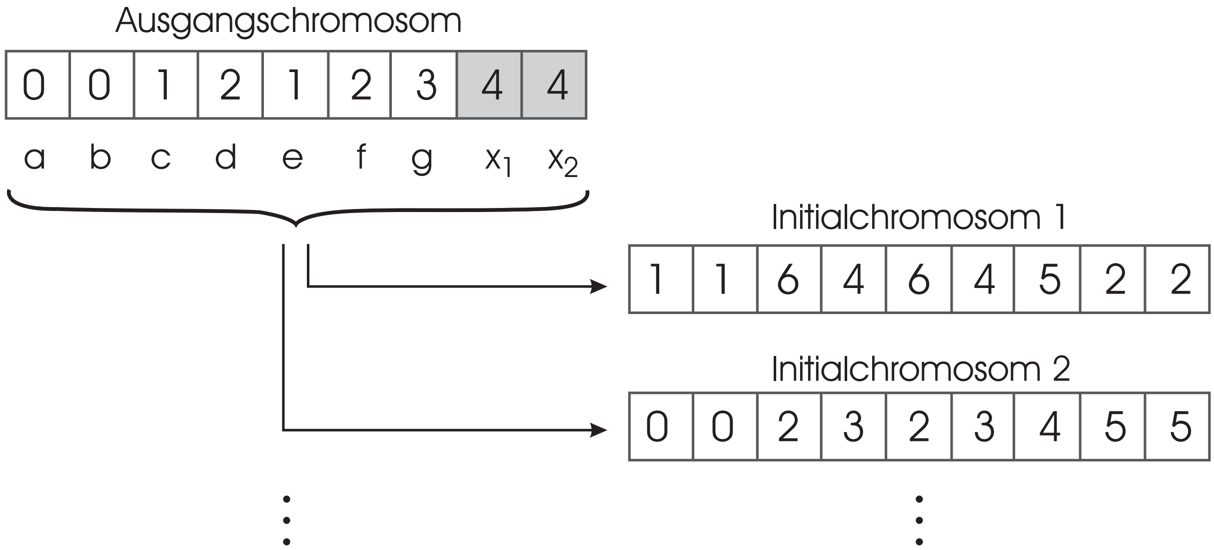

Abbildung 4.12: Die Initialisierung beim GAA

4.2.2 Bewertung

Durch das heuristische Verfahren, das in der Bewertungsfunktion die Zahl der Wechsel von Adreßgenerierungsteilen ermittelt, wird ein großer Teil der Arbeit der GAA hier erbracht. Zusätzlich dazu, daß der Genetische Algorithmus durch die Bewertungen der Individuen gesteuert wird, wird hier ermittelt, welche Adressierungsbefehle bei jedem Individuum vorliegen.

Als Grundlage der Bewertungsfunktion dienen die im Chromosom kodierte Zuteilung der Gruppen zu Speicheradressen und die Gruppenzugriffssequenz. Anhand der Sequenz wird ein wie oben beschriebener Graph (Abbildung 4.10) gebildet. Ziel ist es nun, die Gruppe jedes Knotens zu adressieren, was bedeutet, daß der Knoten von einem Adreßberechnungsbefehl erreicht worden ist. Das heuristische Verfahren berechnet auf dem Graphen die nötigen Adreßgenerierungswechsel: Gestartet wird vom ersten Knoten aus. Jede ausgehende Kante dieses Knotens wird betrachtet. Gesucht ist der längste Pfad im Graphen, dessen Kanten alle das gleiche Gewicht haben. Das Gewicht jeder Kante entspricht dem Sprung, der im Speicher gemacht werden muß, um die nächste Adresse zu erreichen, die die Gruppe des nächsten Knotens repräsentiert. Ein solcher Pfad im Graphen entspricht einer Folge von Speicherzugriffen, deren Offset identisch ist. Für diesen Pfad kann also immer der gleiche Befehl verwendet werden, dessen zugrunde liegendes Adreßregister durch postmodify-Operationen immer auf den nächsten Knoten des Pfades zeigt. Zwischen zwei Zugriffen darf dieses Adreßregister allerdings nicht anderweitig verwendet werden. Sollte der Graph in Abbildung 4.13 als längsten Pfad den markierten Pfad von Knoten K_m ausgehend mit dem Offset +2 enthalten, wäre der sich ergebende Befehl $(P_0, 2)$. Ist der Offset des ermittelten Befehles zu groß für eine Verwendung von Konstanten (Offset>8 oder Offset<-7), muß ein freies Modify-Register mit dem Offset belegt werden.

Anschließend wird mit der Adresse des ersten Knotens des ermittelten Pfades ein freies Adreßregister P_x ($x \in \{0,1,2,3\}$) belegt. Nachdem der Befehl ermittelt worden ist, werden alle Knoten des Pfades markiert und alle Kanten dieser Knoten gelöscht. Auf diese Weise werden zwei weitere Pfade ermittelt, deren Startknoten jeweils die ersten unmar-

kierten Knoten mit der kleinsten Nummer sind. Die Knoten dieser Pfade werden ebenfalls markiert. Für jeden Pfad können nur unmarkierte Knoten gewählt werden. Die drei durch die Pfade repräsentierten Befehle werden betrachtet, als würden sie die ersten im VLIW-Cache abgelegten Befehle bilden.
Die Knoten des Graphen repräsentieren eine zeitliche Sequenz von Gruppenzugriffen und werden in genau dieser Reihenfolge abgearbeitet. Der Befehl, dessen letzter Knoten die kleinste Nummer im Graphen hat, zeigt an, bis zu welchem Zeitpunkt der VLIW-Cache mit Befehlen belegt ist. Alle Knoten, die eine kleinere Nummer im Graphen besitzen und bislang nicht markiert wurden, können mit den Befehlen im VLIW-Cache nicht adressiert werden, da diese zu einem Zeitpunkt adressiert werden müssen, zu dem bis auf eine alle Befehlszeilen des VLIW-Caches mit anderen Befehlen belegt sind.
Aus diesem Grund wurde bislang der vierte Befehl des VLIW-Caches nicht benutzt. Alle Knoten die bis zu diesem Knoten nicht adressiert werden konnten, werden anschließend mit dem Page-Pointer Register adressiert. Dazu wird der Einfachheit halber angenommen, daß das Page-Pointer Register einen Offset besitzt, mit dem der ganze Speicher abgedeckt werden kann. Da jedoch trotzdem für jeden Knoten ein neuer Befehl generiert werden muß, entstehen Extrakosten für jeden so adressierten Knoten. Alle diese Knoten werden ebenfalls markiert. Im vorliegenden Beispiel (Abbildung 4.13) wird der Knoten K_{m+2} nicht markiert. Sollten die drei VLIW-Befehle schon feststehen und die kleinste adressierte Knotennummer aller Befehle größer $m + 2$ sein, kann dieser Knoten nur durch den vierten Befehl mittels des Page-Pointer Registers erreicht werden.

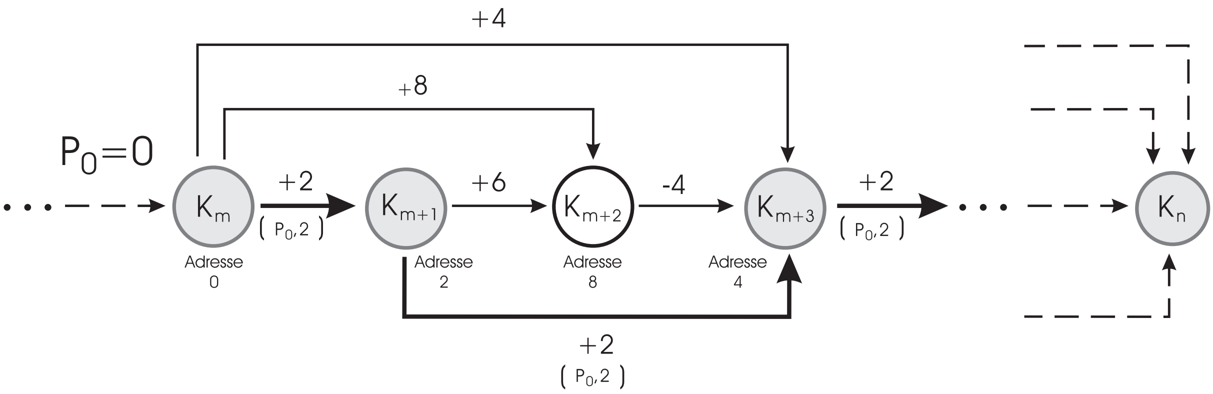

Abbildung 4.13: Pfadsuche des heuristischen Verfahrens des GAA

Nun sind alle Knoten bis zum Ende der *Lebensdauer*[7] eines Befehls im VLIW-Cache adressiert. Der Befehl mit dem frühesten Lebensdauerende kann gelöscht und ein neuer längster Pfad im Graphen gesucht werden. Der Startknoten dieses neuen Pfades ist der unmarkierte Knoten mit der kleinsten Nummer im Graphen. Nachdem so ein neuer Befehl ermittelt worden ist, der denjenigen Befehl ersetzt, dessen Lebensdauer abgelaufen ist, müssen wiederum alle Knoten bis zu dem Zeitpunkt, an dem die Lebensdauer eines nächsten Befehls endet, mit dem Page-Pointer Register markiert werden. Dieses Verfahren wird fortgesetzt, bis alle Knoten des Graphen markiert worden sind. Die verursachten Kosten der Zuteilung

[7]Damit ist die Zeitspanne von der Markierung des ersten Knotens mit diesem Befehl bis einschließlich des letzten Befehls gemeint.

der Gruppen zu Adressen ergeben sich aus den verursachten Adressierungen mittels des Page-Pointer Registers und der Zahl der verwendeten Durchläufe zur Markierung aller Knoten des Graphen.

4.2.3 Crossover

Beim GAA wird ein 1-Punkt-Crossover durchgeführt. Nach der Selektion zweier Elternchromosomen wird ein beliebiger Punkt als Überkreuzungspunkt ermittelt. Der hintere Teil beider Chromosomen wird getauscht. Die sich ergebenden Chromosomen sind eventuell keine gültigen Individuen und müssen korrigiert werden, da der Überkreuzungspunkt zufällig ermittelt wird und eventuell Gruppen getrennt werden. Die Gruppenzusammensetzung aller Chromosomen der Population ist gleich. Das bedeutet, daß eine bestimmte Gruppe immer eine feste Genmenge[8] belegt. Wird nun durch Crossover eine Gruppe getrennt, werden die ausgetauschten Elemente dieser Genmenge gezählt und mit der Gesamtzahl an Elementen dieser Gruppe verglichen. Werden mehr als die Hälfte der Elemente getauscht, werden auch alle anderen Gene dieser Gruppe im Zielchromosom mit dem Wert der getauschten Gene belegt. Ansonsten werden die getauschten Gene mit dem Wert der schon vorhandenen Gene belegt.
In Abbildung 4.14 werden die letzten drei Gene getauscht. Die Gengruppe mit dem Allel 3 von Elter 1 ist $\{4, 8, 9\}$. Da zwei der drei enthaltenen Gene getauscht werden, wird das dritte Gen (4) in Nachkomme 2 angeglichen. Da auf allen Chromosomen die gleichen Genmengen vorkommen, muß das Verfahren für Nachkomme 1 analog durchgeführt werden. Da das erste Gen des Tauschabschnittes nur ein Element aus einer Menge von drei Elementen dieser Gruppe ist, wird dieses Gen in beiden Nachkommen der restlichen Gruppe (aus zwei Genen im jeweiligen Chromosom) angeglichen. Anschließend sind beide Nachkommen gültige Individuen.

Ein weiterer Sonderfall tritt auf, wenn zwei Gruppen die gleichen Allele besitzen. In Abbildung 4.15 hat zum Beispiel das erste Gen des Tauschabschnittes von Elter 1 die gleiche Belegung wie eine schon existierende Genmenge im vorderen Teil von Elter 2. Nun werden die Genmengen verglichen, die größere Anzahl entscheidet, welche Genmenge unverändert bleibt und welche ein neues, in diesem Chromosom unbenutztes Allel erhält. Hier wird das Gen der Tauschmenge neu mit einer 0 belegt. Als gültiger Wert für eine Neubelegung können alle in diesem Individuum nicht benutzten Gruppennummern gewählt werden. Dies entspricht einer Verschiebung dieser Gruppe an eine andere Adresse. Am Tool kann eingestellt werden, wie groß die Zahl der unbenutzten Adressen sein darf. Bei einer 0 für diesen Wert gibt es keine unbelegten Zeilen, in diesem Fall muß bei jeder Neubelegung einer Zeile mit einem Wert ein Tausch durchgeführt werden. Im ersten Chromosom tauchen die Allele des Tauschabschnittes des zweiten Chromosoms zwar ebenfalls auf, werden aber gänzlich getauscht, so daß auf eine Angleichung verzichtet werden kann. Die Crossover-Funktion produziert nur gültige Individuen.

[8]Eine Gruppe belegt eine feste Menge von durchnummerierten Genen, zum Beispiel bezieht sich die Genmenge $\{1, 2, 5\}$ auf die Gene 1, 2 und 5. Alle Allele dieser Gene sind gleich.

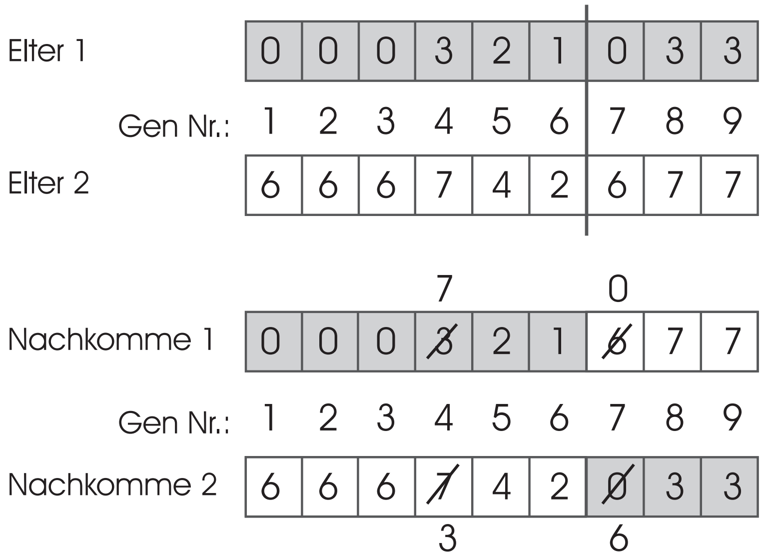

Abbildung 4.14: Crossoverfunktion des GAA

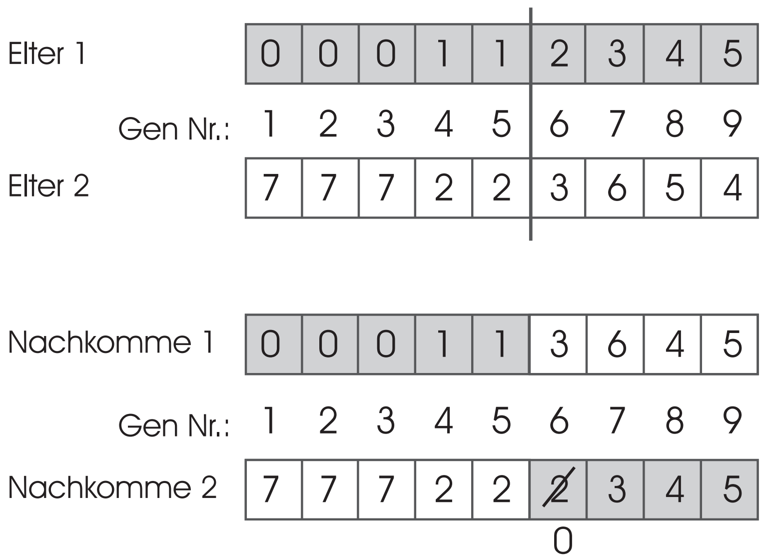

Abbildung 4.15: Gleiche Numerierung der Gengruppen

4.2.4 Mutation

Die Mutation betrifft nicht einzelne Gene, sondern Gengruppen, die sich aus der Gruppenzusammensetzung ergeben. Wird eine Gruppe zur Mutation ausgewählt, werden alle Gene mit einem neuen Wert belegt. Das entspricht im Speicher einer Verschiebung dieser Gruppe in eine andere Speicherzeile unter den gleichen Beschränkungen einer Verschiebung wie beim Crossover, sollten keine freien Speicherzeilen zugelassen sein (abhängig vom eingestellten Wert für freie Speicherzeilen). Werden keine freien Speicherzeilen zugelassen, wird in der Mutation nur getauscht.

4.3 Resultate

In diesem Abschnitt wird der GAA bei einigen zufallsbasierten Testsequenzen angewendet, wobei die Güte der Ergebnisse und die Laufzeiten vorgestellt werden. Zudem werden als reale Benchmarks die Sequenzen eines IIR-Filters und die Sequenz der DCT als Eingabe verwendet. Die hier als Vergleichsgrundlage verwendete naive Lösung geht davon aus, die drei am häufigsten vorkommenden Adressen in drei Adreßregister zu schreiben und dann die entsprechenden Befehle $AGU(P_x, 0)$ zu verwenden. Es ist klar, daß die naive Lösung bei wenigen anzuordnenden Gruppen effizienter ist als bei einer großen Gruppenmenge.

4.3.1 Einsatz des GAA bei zufallsbasierten Testsequenzen

Abbildung 4.16 zeigt Ergebnisse, die sich bei Zufallssequenzen mit 13 Gruppen ergeben (diese Testsequenzen wurden auch beim GAG verwendet). In diesem Fall wurden keine freien Speicherzeilen zugelassen. Die Verbesserungen im Vergleich zu einer naiven Lösung sind nur gering (durchschnittlich bei $0,9\%$).

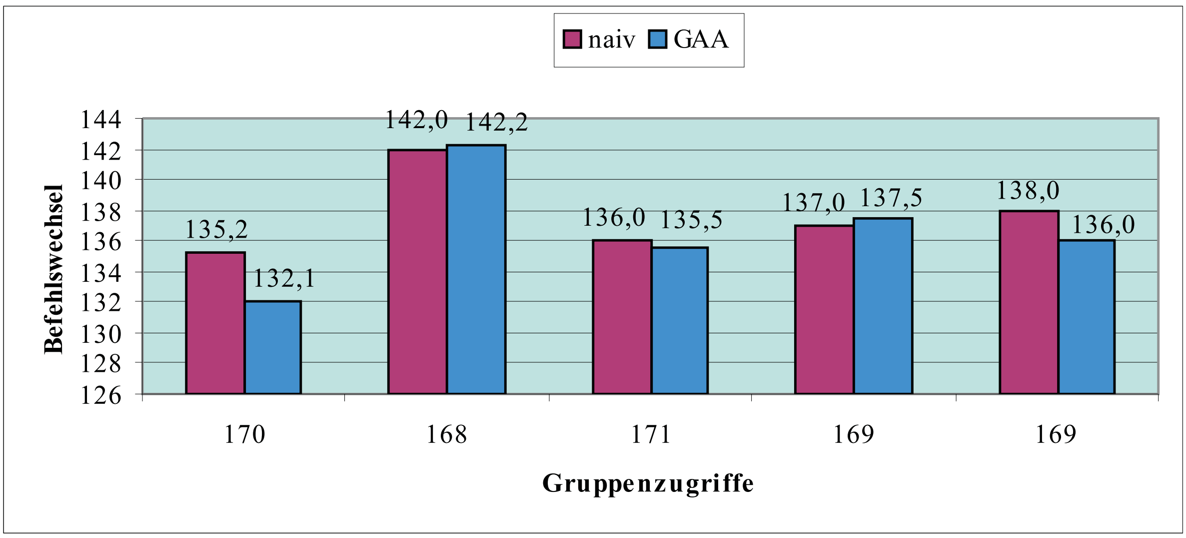

Abbildung 4.16: Ergebnisse des GAA bei Testsequenzen mit 13 Gruppen

Scheinbar ergibt sich erst bei einer größeren Anzahl von Gruppen ein höheres Optimierungspotential. Zu diesem Zweck wurde eine zweite Testreihe mit 5 bis 25 Gruppen bei einer Sequenzlänge von 100 Gruppenzugriffen durchgeführt. Die Ergebnisse stellt 4.17 dar. Es ist deutlich zu erkennen, daß erst ab einer Gruppenzahl von 15 ein Einsparpotential zur naiven Lösung bei diesen Testsequenzen vorhanden ist. Ab 15 Gruppen ist dieses Einsparpotential zwischen 15% und 16,5% relativ konstant.

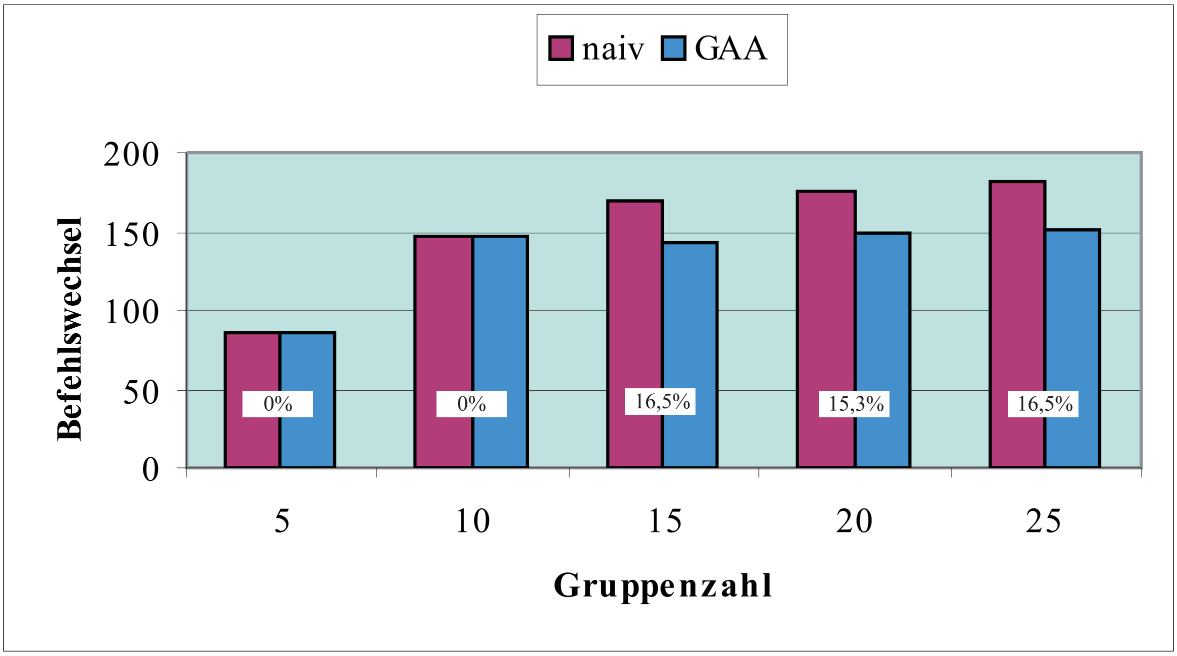

Abbildung 4.17: Vergleich der Ergebnisse des GAA bei verschiedenen Gruppenanzahlen

Die Laufzeiten des GAA hängen linear von der Gruppenzahl ab, Abbildung 4.18 macht das deutlich. Die Laufzeit des verwendeten Genetischen Algorithmus (die Phasen Crossover und Mutation) ändert sich durch längere Chromosomen nur linear. Auf die heuristische Bewertungsfunktion haben mehr Gruppen keinen Einfluß, da nur die Anzahl der Gruppenzugriffe gezählt werden, und diese liegen bei allen betrachteten Sequenzen bei 100 Gruppenzugriffen.

Die Abhängigkeit der Laufzeit von der Gruppenzugriffszahl ist allerdings nicht mehr linear, wie Abbildung 4.19 zeigt. In diesem Fall trägt nur die Bewertungsfunktion zur Laufzeit bei, die Laufzeiten von Crossover und Mutation des Genetischen Algorithmus ändern sich durch eine höhere Anzahl von Gruppenzugriffen nicht, da die Chromosomen dadurch nicht verändert werden.

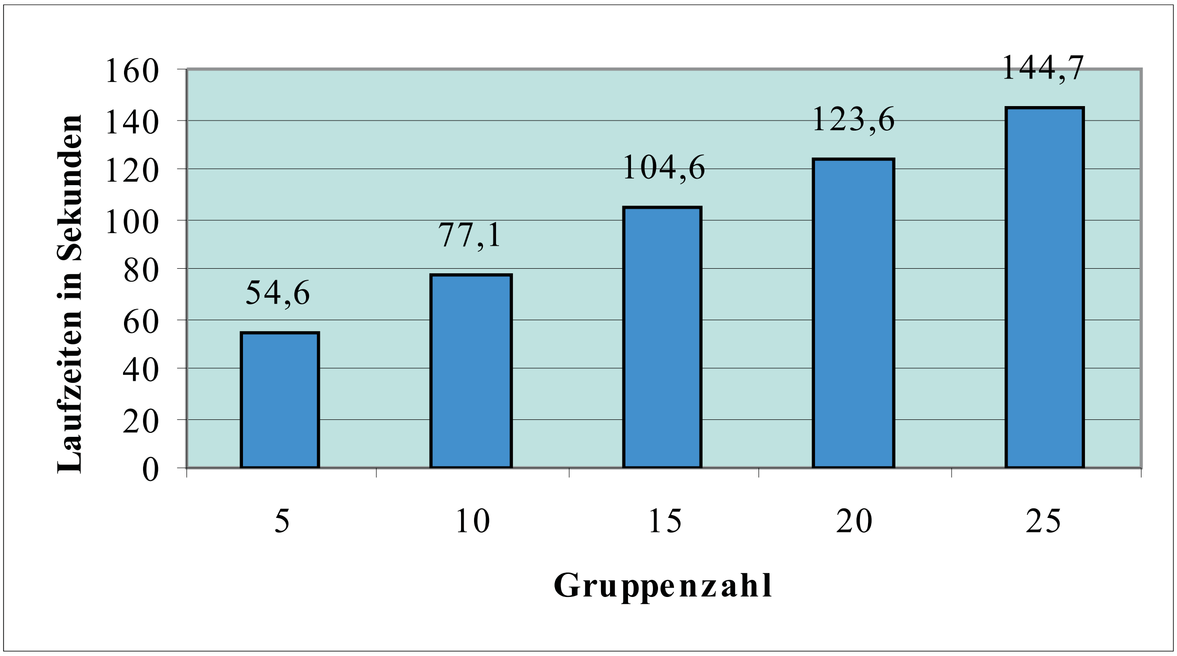

Abbildung 4.18: Laufzeitenvergleich des GAA bei verschiedenen Gruppenanzahlen

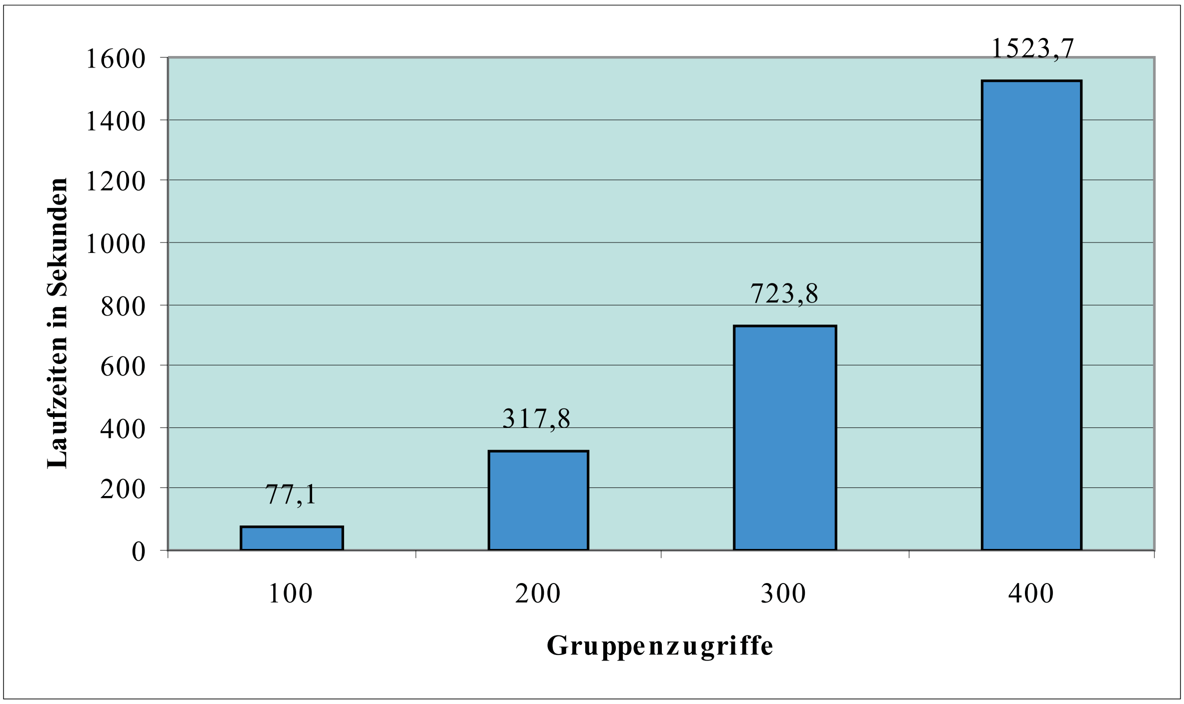

Abbildung 4.19: Laufzeitenvergleich des GAA bei unterschiedlicher Anzahl von Gruppenzugriffen

4.3.2 Einsatz des GAA bei Sequenzen realer Benchmarks

Als reale Benchmarks werden bei dieser Bewertung eine Sequenz eines IIR-Filters und eine Sequenz der DCT verwendet.
Bei dem Einsatz des GAA mit einer Sequenz des IIR-Filters sind nur Gruppengrößen von 2, 4 und 6 sinnvoll, da ansonsten jede der entstehenden Gruppen mit einem Befehl adressiert werden kann (die Sequenz enthält 19 Variablen). Die Ergebnisse mit den sich ergebenden 10, 5 und 4 Gruppen zeigt Abbildung 4.20. Es wird deutlich, daß ein Einsparpotential vorhanden ist. Allerdings sind die Ergebnisse der Sequenz nicht auf längere Sequenzen direkt übertragbar, da bei der verwendeten Sequenz jede Variablengruppe durchschnittlich seltener als zweimal (zwischen $1,6$ und $1,4$) aufgerufen wird. Dadurch ist die zu Beginn erläuterte naive Lösung nicht sehr aussagekräftig. Dies gilt dann auch für einen Vergleich mit dieser Lösung.

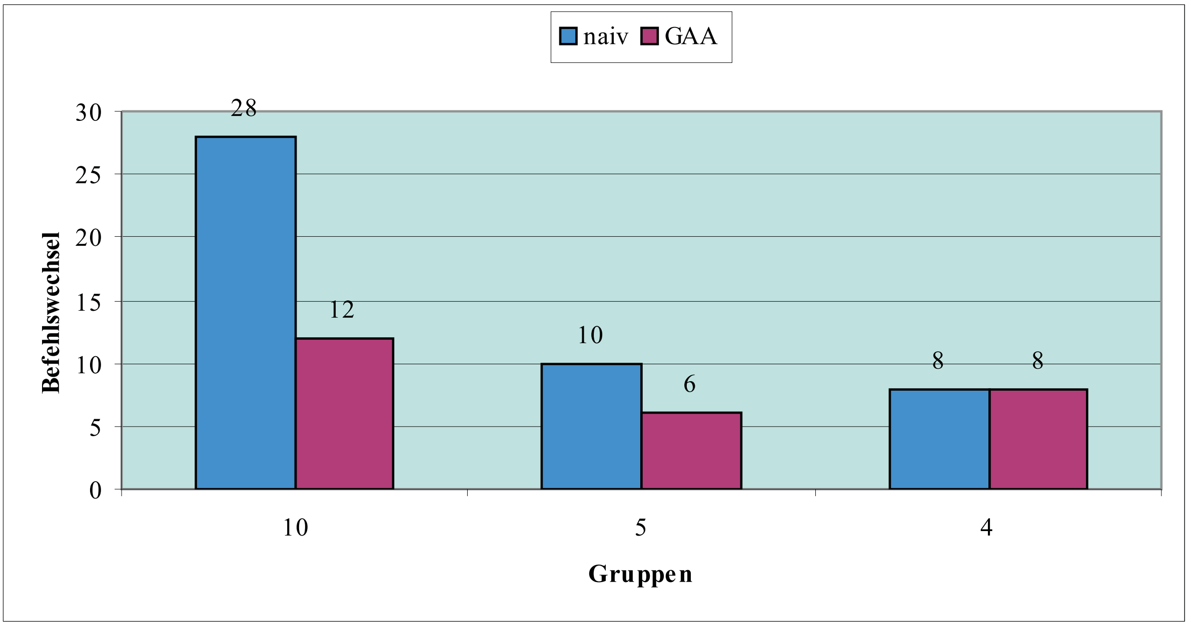

Abbildung 4.20: Adreßgenerierungs-Wechsel bei einer Sequenz des IIR-Filters

Abbildung 4.21 zeigt Ergebnisse, die sich ergeben, wenn im Speicher unterschiedlich viele freie Speicherzeilen zugelassen werden. Die Prozentwerte geben die Einsparungen des GAA gegenüber der naiven Lösung an. Es ist offensichtlich von Vorteil, Speicherzeilen freizulassen, da bei 0 zulässigen freien Speicherzeilen das Einsparpotential kleiner ist als bei 10 oder 20 freien Speicherzeilen. Der GAA kann die zusätzlichen Freiheiten bei der Gruppenanordnung ausnutzen. Es wird aber auch klar, daß ab einer gewissen Zahl von Speicherzeilen der Lösungsraum so groß werden kann, daß die Ergebnisse wieder schlechter werden (bei 30 und mehr freien Speicherzeilen). Die besten Ergebnisse des GAA sind $15,2\%$ besser als die der naiven Lösung.

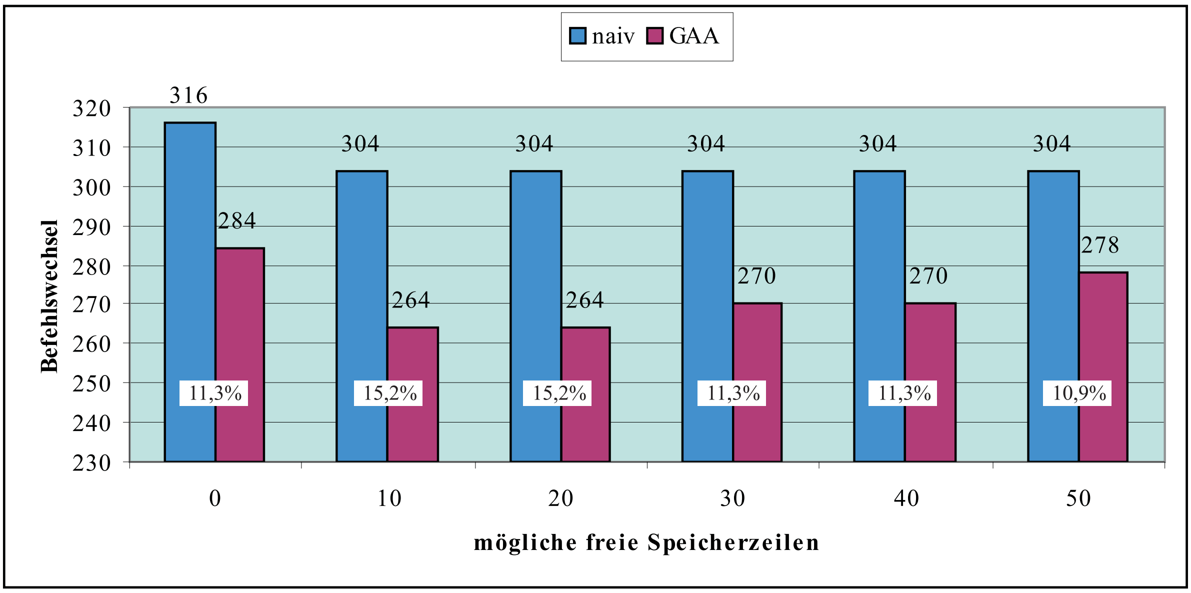

Abbildung 4.21: Adreßgenerierungs-Wechsel bei der Sequenz der DCT

Kapitel 5

Zusammenfassung & Ausblick

5.1 Zusammenfassung

Die Durchführung der Adreßzuweisung für den M3-DSP erfordert aufgrund des Gruppenspeichers besondere Optimierungs-Techniken. So gilt es einerseits die in einem Programm verwendeten Variablen zu Gruppen anzuordnen (*Gruppenbildung*) und andererseits diesen Gruppen Adressen zuzuweisen (*Gruppenanordnung*).
Das Problem der Gruppenbildung hat das Ziel, die Speicherzugriffe des compilierten Programms zu minimieren und dadurch indirekt auch die Ausführungszeit zu reduzieren. In dieser Arbeit wurde das Problem der Gruppenbildung auf ein Partitionierungsproblem abgebildet. Das zugrunde liegende Partitionierungsproblem ist NP-hart. Zur Lösung dieses Problems wurden einige heuristische Verfahren vorgestellt, die für eine Partitionierung eingesetzt werden können. Es wurden Varianten eines Kruskal-Algorithmus implementiert, die in polynomieller Laufzeit eine Partitionierung durchführen, deren Ergebnisse sich jedoch als unbefriedigend erwiesen. Ein ebenfalls implementierter Kernighan-Lin-Algorithmus führte schon zu recht guten Ergebnissen. Allerdings kann keine polynomielle Laufzeit garantiert werden.
Der in dieser Arbeit implementierte Genetische Algorithmus zur Gruppenbildung (GAG) lieferte in polynomieller Zeit vor allem für größere Variablenmengen bessere Ergebnisse als die Heuristiken. Die Einsparungen gegenüber den Kruskal-Algorithmus-Varianten lagen zwischen 10% und 35%, gegenüber dem Kernighan-Lin-Algorithmus bei 3% bis zu 10%. Die Ergebnisse des GAG brachten Einsparungen von Speicherzugriffen von durchschnittlich 15% gegenüber einer ebenfalls implementierten naiven Lösung. Bei einem Einsatz des GAG für eine Sequenz der DCT brachte die Partitionierung im Vergleich zur ermittelten naiven Lösung beim M3-DSP-Compiler Einsparungen von durchschnittlich 8%. Im Vergleich zu der vorher vom M3-DSP-Compiler verwendeten Gruppenpartitionierung lagen die Ergebnisse sogar bei über 60% Einsparung von Speicherzugriffen.
Beim Einsatz des M3-DSP-Compilers könnte zum Beispiel zu Testzwecken die beim GAG verwendete naive Gruppenpartitionierung verwendet werden, da sie schnell zu ermitteln ist und bereits zu recht guten Ergebnissen führt. Wird mehr Bedarf auf Codequalität gelegt, sollte der GAG eingesetzt werden, da er die Ergebnisse der naiven Lösung noch deutlich verbessert.

Das Problem der Gruppenanordnung hat das Ziel, die Programmgröße zu minimieren. Dazu müssen die vorliegenden Gruppen im Speicher angeordnet und Adressen zugewiesen werden. Es stellt sich jedoch als schwierig dar, eine vorgenommene Gruppenanordnung zu bewerten, da es sehr viele Freiheitsgrade gibt. Diese bestehen in der freien Auswahl eines im VLIW-Cache gespeicherten VLIWs, das für die Bildung eines neuen VLIWs zugrunde gelegt wird und den verschiedenen Möglichkeiten der Bildung eines Adreßgenerierungsteils, für den vier Adreßregister und vier Modify-Register zur Verfügung stehen und bei dem zwischen drei verschiedenen Adressierungsarten gewählt werden kann. Es besteht eine Analogie zu den Problemen SOA und GOA, da auch dort Variablen Adressen zugewiesen werden. Es gibt aber keine Verfahren, die direkt mit dem Problem der Gruppenanordnung zu vergleichen sind. Die Lösung des GOA-Problems als Teil des Gruppenanordnungs-Problems stellt bereits ein NP-hartes Problem dar. Deswegen wurde zur Problemlösung des Gruppenanordnungsproblems ein Genetischer Algorithmus verwendet, der GAA. Die Bewertung einer gegebenen Gruppenanordnung wurde mit einem heuristischen Verfahren durchgeführt. Die Ergebnisse lagen ab einer Gruppenanzahl von 15 bei durchschnittlich 15% weniger Befehlswechseln gegenüber einer zu Vergleichszwecken berechneten naiven Lösung. Für die Sequenz der DCT ergaben sich Einsparungen von knapp 10%.

5.2 Ausblick

Auch bei den zukünftigen Entwicklungen ist es sinnvoll, Gruppenbildung und Gruppenanordnung getrennt zu betrachten, da die Abhängigkeiten beider Probleme gering sind und die Komplexität jedes Einzelproblems schon NP-hart ist. Es ist zu überlegen, die Gruppenanordnung zu einem späteren Zeitpunkt im Compilierprozeß durchzuführen, wodurch sich eine größere Trennung von Gruppenbildung und Gruppenanordnung bei der Berechnung eines Speicherlayouts ergibt. Im folgenden werden die möglichen Erweiterungen beider entwickelter Problemlösungen, GAG und GAA, vorgestellt.

5.2.1 Gruppenbildung

Die eingegebenen Variablenzugriffssequenzen bestehen nur aus Einzelzugriffen auf Variablen und Constraints bezüglich des Speicherlayouts in Form von gegebenen Gruppen. Hierdurch wird es der Codegenerierung insbesondere ermöglicht, skalare Variablen anzuordnen. Spezielle Array-Optimierungen werden in dieser Arbeit nicht betrachtet. Diese Optimierung könnte in einem gesonderten Schritt erfolgen, dessen Ergebnisse wiederum als Constraints an den GAG weitergegeben werden könnten.

Das Tool kann für einzelne Teile eines Programms (Funktionen oder Basisblöcke) eingesetzt werden. Werden alle Zugriffssequenzen globaler oder lokaler Variablen als Eingabe für den GAG nur verkettet, entstehen allerdings eventuell Ungenauigkeiten, weil dann beim Übergang der letzten Variable eines Basisblocks zur ersten Variable eines nachfolgenden Basisblocks Nachbarschaftsbeziehungen entstehen, die gar nicht vorhanden sind[1],

[1] Eine weitere Möglichkeit bestünde im Einfügen von *don't-care-Variablen*, die im späteren Verlauf keiner Gruppe zugeteilt würden, zwischen zwei Basisblöcken.

weil diese Basisblöcke vielleicht nicht beide in einem Programmdurchlauf ausgeführt werden können (beispielsweise bei einer If-Then-Else Anweisung). Es wäre wünschenswert, diese Stellen in einer Variablenzugriffssequenz markieren zu können, so daß das Tool die Sequenz in mehrere Teilsequenzen unterteilen kann. Dann könnte im Tool ermittelt werden, welcher Basisblock das höchste Optimierungspotential besitzt, so daß die anderen Teilsequenzen erst nach der Optimierung dieses Basisblocks betrachtet werden. Zudem könnten dem Tool Aufrufwahrscheinlichkeiten der Basisblöcke übergeben werden.

Der GAG kann bei einer geeigneten Erstellung der Variablenzugriffssequenz auch für andere Partitionierungsprobleme verwendet werden, deren Problemstellungen zu denen der Gruppenbildung ähnlich sind.

5.2.2 Gruppenanordnung

Zur Gruppenanordnung kann auch auf das von Fabian David [8] entwickelte Tool zur Lösung des GOA mit einem Genetischen Algorithmus verwiesen werden. Die beim GAA vorkommenden Variablengruppen können nicht mehr aufgeteilt werden, so daß sie wie Variablen betrachtet werden können, die einer Adresse zugewiesen werden müssen. Das hier betrachtete Optimierungsziel besteht beim Gruppenanordnungsproblem jedoch in einer Reduzierung der Adreßwechsel, und nicht in einer Reduzierung der nicht mit automodify durchgeführten Veränderungen der Adreßregister zur Adressierung wie bei dem von David entwickelten Tool. Eine Anpassung an diese geänderte Zielfunktion ist denkbar durch Verwendung einer Kombination aus dem von Fabian David verwendeten Verfahren und der Bewertungsfunktion des GAA. Die Parameter des M3-DSP, wie die Zahl der Adreßregister und Modify-Register, können von dem von David entwickelten Tool gut berücksichtigt werden. Auch der parametrisierbare VLIW-Cache könnte eingearbeitet werden.

Eine Verbindung der beiden Probleme GAA und GAG besteht in der Bildung von Variablengruppen beim GAG, die wiederum aus mehreren Teilgruppen bestehen, deren Variablen zu den anderen **Teilgruppen** keine Nachbarschaftsbeziehungen besitzen. Durch eine andere Anordnung dieser Teilgruppen zu Gruppen würden sich die Ergebnisse des GAG nicht ändern, da keine zusätzlichen Nachbarschaftsbeziehungen der Variablen ausgenutzt werden können. Beim GAA kann dies jedoch Auswirkungen haben. Es wäre denkbar, diese Teilgruppen beim GAA wieder als einzelne Gruppen zu betrachten, und die Gruppenanordnung mit diesen Teilgruppen durchzuführen.

Anhang A

Benutzerhandbuch

In diesem Abschnitt werden die Möglichkeiten vorgestellt, Parameter für das entwickelte Tool einzustellen. Da der M3-DSP ein Derivat der M3-DSP Plattform ist und diese parametrisierbar ist, können die unterschiedlichen Parameter für eine gegebene Variablenzugriffssequenz vom Anwender vorgegeben werden. Zudem können auch die genetischen Parameter des GAG und des GAA eingestellt werden. Das Tool wird aufgerufen mit der Befehlszeile:

pgaprog $< var\_sequ\_name > < parameter >$

Um die Möglichkeiten der Parametrisierbarkeit und unterschiedliche Einstellungen der genetischen Parameter nutzen zu können, können weiterhin folgende Parameter an diesen Aufruf angehängt werden:

Parameter für die M3-DSP Plattform

-gg $< parameter >$: Gruppengröße $gg \in \mathbb{N}$
Standard: 16 (M3-DSP)

-sz $< parameter >$: Speicherzeilen $sz \in \mathbb{N}$
Standard: 100

-vliw $< parameter >$: Anzahl der Befehle im VLIW-Cache $vliw \in \mathbb{N}$
Standard: 4 (M3-DSP)

Genetische Parameter

-gagmut $< parameter >$: Mutations-Wahrscheinlichkeit $prob_{gagmut} \in [0, 1]$ des GAG
Standard: 1,875/Anzahl Gene

-gagcr $< parameter >$: Crossover-Wahrscheinlichkeit $prob_{gagcr} \in [0, 1]$ des GAG
Standard: 0,7

-gagpop $< parameter >$: Populationsgröße $gagpop \in \mathbb{N}$ des GAG
Standard: 50

-gagit $< parameter >$: Iterationen $gagit \in \mathbb{N}$ des GAG
Standard: 1000

-*gaamut* $<$ *parameter* $>$: Mutations-Wahrscheinlichkeit $prob_{gaamut} \in [0, 1]$ des GAA
Standard: 1,875/Anzahl Gene

-*gaacr* $<$ *parameter* $>$: Crossover-Wahrscheinlichkeit $prob_{gaacr} \in [0, 1]$ des GAA
Standard: 0,7

-*gaapop* $<$ *parameter* $>$: Populationsgröße $gaapop \in I\!N$ des GAA
Standard: 50

-*gaait* $<$ *parameter* $>$: Iterationen $gaait \in I\!N$ des GAA
Standard: 200

Sonstige Parameter

-*lea* $<$ *parameter* $>$: Mit oder ohne Left-Edge-Algorithmus (LEA), $lea \in \{J, N\}$
Standard: N (ohne LEA)

-*kl* $<$ *parameter* $>$: Initialisierung mit oder ohne Kernighan-Lin-Algorithmus (KL), $kl \in \{J, N\}$
Standard: J (mit KL)

-*fs* $<$ *parameter* $>$: Zugelassene freie Speicherzeilen $fs \in I\!N$
Standard: 0

-*s* $<$ *parameter* $>$: Defaultwert bei Eingabe dynamischer Schleifen, $s \in I\!N$
Standard: 10

Beispielaufruf für einen Start des GAG mit Gruppengröße 4 und 200 Generationen Laufzeit bei einer Crossover-Rate von $0, 6$:
pgaprog $<$ *var_sequ_name* $>$ -gg 4 -gagit 200 -gagcr 0,6

Literaturverzeichnis

[1] Baker, J.E.: *Reducing Bias and Inefficiency in the Selection Algorithm*, Genetic Algorithms and their Applications, Proceedings of the Second International Conference on Genetic Algorithms, Hillsdale/NJ: Lawrence Erlbaum 1987

[2] Bartley, D.H.: *Optimizing Stack Frame Access for Processors with Restricted Addressing Modes*, Software Practice and Experience, vol. 22(2), pp. 101-110, 1992

[3] Belady, L.: *A Study of Replacement Algorithms for a Virtual-Storage Computer*, IBM System Journal 5(2): pp 78-101, 1966

[4] Cheng, W.H.; Lin, Y.-L.: *Addressing Optimization for Loop Execution Targeting DSP with Auto-Increment/Decrement Architecture*, 11th International Symposium on System Synthesis (ISSS), Hsinchu, Taiwan, R.O.C., December 2-4, 1998

[5] Cong, J.; Li, Z.; Bagrodia, R.: *Acyclic Multi-Way Partitioning of Boolean Networks*, Proceedings of the 31st ACM/IEEE Design Automation Conference DAC'94, San Diego, CA, pp. 670-675, June 1994

[6] Cong, J.; Smith, M.: *A Parallel Bottom-Up Clustering Algorithm with Applications to Circuit Partitioning in VLSI Design*, pp. 755-760, 30. DAC 1993: Dallas, Texas, USA, 1993

[7] Cong, J.; Li, H. P.; Lim, S.K.; Shibuya, T; X., D.: *Large Scale Circuit Partitioning With Loose/Stable Net Removal And Signal Flow Based Clustering*, International Conference of Computer-Aided Design, ICCAD 97, pp. 441-446, 1997

[8] David, F.: *Optimierte Adreßzuweisung in DSP-Compilern* Diplomarbeit am Lehrstuhl XII, Universität Dortmund, Januar 1998

[9] Fiduccia, C.M.; Mattheyses, R.M.: *A Linear-Time Heuristic for Improving Network Partitions*, 19th DAC '82, pp. 175-181, 1982

[10] Hansen, C.: *MicroUnity's MediaProcessor Architecture*, IEEE Micro, Vol. 16, No. 4, pp. 34-41, August 1996

[11] Holland, J.: *Adaptation in Natural and Artificial Systems*, Ann Arbor: University of Michigan Press, 1975

[12] De Jong, K.: *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Dissertation, University of Michigan, Ann Arbor 1975

[13] Kernighan, B.W.;Lin, S.: *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell System Technical Journal, Vol. 49, pp. 291-307, 1970

[14] Kruskal, J.B.: *On the shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*, Proceedings of the American Mathematical Society 71, pp. 48-50, 1956

[15] Kurdahi, F.J.; Parker, A.C.: *REAL: Programm for REgister ALlocation*, Procedings of the 24th ACM/IEEE Design Automation Conference, pp 210-215, 1987

[16] Leupers, R.: *Retargetable Code Generation for Digital Signal Prozessors*, Kluwer Academic Publishers, 1997

[17] Leupers, R.: *Novel Code Optimization Techniques for DSPs*, 2nd European DSP Education and Research Conference, Paris/France, September 1998

[18] Leupers, R.; Basu, A; Marwedel, P.: *Optimizing Array Index Computation in DSP Programs* ASP-DAC 1998, Yokohama/Japan, Februar 1998

[19] Leupers, R.; Marwedel: *Algorithms for Address Assignment in DSP Code Generation*, ICCAD 1996

[20] Leupers, R.; Marwedel: *Optimierende Compiler für DSPs: Was ist verfügbar?*, DSP Deutschland 1997, München, Oktober 1997

[21] Liao, S., Devadas, S. et. al.: *Storage Assignment to Decrease Code Size*, Programming Language Design and Implementation (PLDI) 1995

[22] Nissen, V.: *Einführung in Evolutionäre Algorithmen: Optimierung nach dem Vorbild der Evolution*, Braunschweig, Wiesbaden: Vieweg 1997

[23] Saghir, M.A.R.; Chow, P.; Lee, C.G.: *Exploiting Dual Data-Memory Banks in Digital Signal Processors*, ASPLOS 1996, pp. 234-243, 1996

[24] Schneburg, Heinzmann, Feddersen: Genetische Algorithmen und Evolutionsstrategien, Addison-Wesley 1994

[25] Sung, W.; Kim, J.; Ha, S.: *Memory Efficient Synthesis from Dataflow Graph*, 11th Internationl Symposium on System Synthesis (ISSS), pp. 137-142, 1998

[26] Sudarsanam, A.: *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*, PhD, Princeton University, USA, May 1998

[27] Sudarsanam, A.; Malik, S.: *Memory Bank and Register Allocation in Software Synthesis for ASIPs*, ICCAD '95, pp. 388-392, 1995

[28] Vemuri, R.; Vemuri,R.: *A Genetic Algorithm for Multichip Partitioning*, TM-ECE-DDE-92-26, June 1992.

[29] Wang, M.; Lim, S.K.; Cong, J.; Sarrafzadeh, M.: *Multi-way Partitioning Using Bi-partition Heuristics*, Asia and South Pacific Design Automation Conference 2000 (ASP-DAC 2000), 2000

[30] McWilliams, P.: *Optimisation Methods for Dynamic Load Balancing*, EURO-CM-PAR 97 Conference at Lochinver, Scotland, April 1997

[31] Weiss, M.H.; Fettweiss, G.P.: *Dynamic Codewith Reduction for VLIW Instruction Set Architectures in Digital Signal Processors*, 3rd International Workshop on Image and Signal Processing, pp. 517-520, November 1996

[32] Weiss, M.H.; Fettweiss, G.P.; Lorenz, M.; Leupers, R.; Marwedel, P.: *Toolumgebung für plattformbasierte DSPs der nächsten Generation*, DSP Deutschland, 1999

[33] Weiss, M.H.; Schwann, P.; Fettweiss, G.P.: *TVLIW++: Instruktionssatzsimulator für VLIW-DSPs, M.H. Weiss*, DSP Deutschland, S.190ff., 1997