Universität Passau
Fakultät für Informatik und Mathematik

Optimierung der Speichernutzung automatisch parallelisierter Programme

Diplomarbeit

Autor:
Benjamin Schulte

Aufgabensteller:
Priv.-Doz. Dr. Martin Griebl
Lehrstuhl für Programmierung
Universität Passau

Zweitgutachter:
Prof. Christian Lengauer, Ph.D.

Passau, 28. März 2007

Zusammenfassung

Durch die Verbreitung von geeigneter Hardware wird die parallele Ausführung von Programmen möglich. Sie ist vor allem bei großen zu verarbeitenden Datenmengen vorteilhaft; um dabei nicht auf verlangsamende Auslagerungsspeicher zurückgreifen zu müssen, ist eine gute Speichernutzung bei parallelen Programmen wichtiger als im Sequentiellen. Diese Arbeit stellt ein Verfahren vor, das die Speichernutzung von im Polytopmodell automatisch parallelisierten Programmen optimiert.

Danksagungen

Mein Dank gilt allen am LooPo-Projekt beteiligten Personen. Allen voran möchte ich natürlich Martin Griebl und Armin Größlinger danken, die bei allen konzeptionellen und technischen Schwierigkeiten erste und erstklassige Ansprechpartner waren und immer hilfreiche Anregungen und Aushilfen fanden. Das ganze Team hat in der LooPo-Implementation Datenstrukturen und Hilfsfunktionen entworfen und implementiert, mit denen man meist schneller und besser als erwartet zum erwünschten Ziel kam. In diesem Sinne danke ich auch den Entwicklern von Haskell: Es ist immer wieder überraschend, welch eine Produktivität diese funktionale Sprache ermöglicht.

Nicht zuletzt möchte ich auch Fabien Quilleré und Sanjay Rajopadhye danken, die diese Diplomarbeit mit ihrem Verfahren zur Optimierung der Speichernutzung funktionaler Programme erst möglich gemacht haben.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Parallele Ausführung von Programmen wird zunehmend wichtiger. Während dazu früher spezielle und teure Computer nötig waren, ist die Technik inzwischen in modernen Heimcomputerprozessoren integriert. Durch schnellere Netzwerktechnik ist sogar die parallele Ausführung von Programmen auf vernetzten Standardrechnern als virtueller Parallelcomputer möglich.

Die parallele Ausführung eines Programms reduziert die reine Berechnungszeit. Doch im Gegensatz zur sequentiellen Ausführung muss Zeit aufgewandt werden, um Daten zwischen den Recheneinheiten auszutauschen. Bei kleinen Problemgrößen führt dies teilweise zu längerer Ausführungszeit als im sequentiellen Fall. Vergrößert man die zu berechnende Datenmenge, kann es dagegen zu Speicherproblemen kommen: die Daten passen nicht mehr in den Cache oder gar in den Hauptspeicher und werden auf langsamere Ebenen der Speicherhierarchie ausgelagert. Wieder verliert das Parallelprogramm Vorteile gegenüber dem entsprechenden sequentiellen Programm. Es ist folglich wichtig, den verfügbaren Speicher so gut wie möglich zu nutzen.

Unter diesem Aspekt sollte ein automatischer Schleifenparallelisierer, der ein sequentielles Programm in ein semantisch äquivalentes paralleles Programm überführt, eine Optimierung der Speichernutzung vornehmen. Eine Möglichkeit hierzu stellt diese Arbeit vor.

1.2 Begriffe und Definitionen

Die Beschreibungen in diesem Abschnitt sind nur so weit ausgeführt, wie für das Verständnis der weiteren Kapitel nötig ist. Exakte Definitionen und Erklärungen finden sich in Martin Griebls Habilitationsarbeit [2].

1.2.1 Polytopmodell

Das Polytopmodell, 1993 von Lengauer [3] vorgestellt und seitdem oft erweitert, ist eine mächtige mathematische Grundlage zum Analysieren und Parallelisieren von Schleifenprogrammen. Jede Schleife des Ausgangsprogramms ist eine Dimension des Modellraums: ein Programm mit k verschachtelten Schleifen hat einen

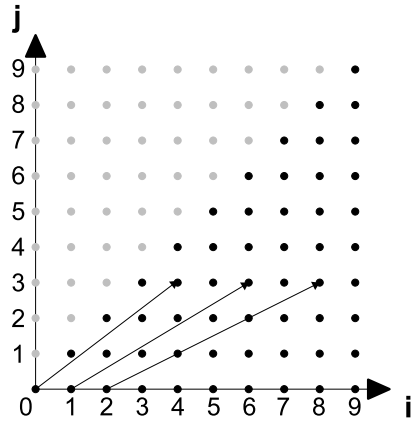

Abbildung 1.1: Darstellung von Indexraum und Abhängigkeiten im Polytopmodell

k-dimensionalen Modellraum. Die Schleifengrenzen bilden in diesem Raum ein k-dimensionales Polytop, den *Indexraum* der Anweisungen in den Schleifen. Die Anweisungen sind Zugriffe auf Arrays, deren Indizes affine Ausdrücke in umgebenden Schleifenvariablen und festen Parametern sein müssen. Abhängigkeiten zwischen *Operationen* bzw. *Instanzen* von Anweisungen bestimmter Schleifenschritte lassen sich als Vektoren zwischen den zugehörigen Punkten des Indexraums darstellen.

Beispiel 1.1 *Gegeben sei ein Schleifenprogramm der Form*

```
 for(i=0; i<=9; i++) {
    for(j=0; j<=i; j++)
S:        A[i,j] = ...;
 }
```

Der im Polytopmodell betrachtete Modellraum dieses Programms ist in Abbildung 1.1 dargestellt: Es gibt zwei verschachtelte Schleifen, der Raum ist zweidimensional. Der Indexraum ist gegeben durch $0 \leq i \leq 9$ *und* $0 \leq j \leq i$ *und spannt ein Polytop auf, das alle in der Abbildung schwarz dargestellten Punkte enthält. Weiterhin stellt die Abbildung Abhängigkeiten dar, z.B. zwischen Operationen von Schleifenschritt* `i=0, j=0` *und* `i=4, j=3`.

1.2.2 Abhängigkeiten

Abhängigkeiten legen die zwingend vorgeschriebene Reihenfolge von Operationen fest. Zwischen zwei Anweisungsinstanzen S und T besteht eine Abhängigkeit, wenn sie auf die gleiche Speicherzelle zugreifen und eine der beiden Operationen schreibt. Beide Instanzen müssen tatsächlich im Programm ausgeführt werden und es darf zeitlich keine andere Operation dazwischen ausgeführt werden, die auf die gleiche Speicherzelle zugreift.

Es gibt drei Typen von relevanten Abhängigkeiten: Eine Abhängigkeit von einem schreibenden zu einem lesenden Statement nennt man *Flow-Abhängigkeit*. Eine Abhängigkeit von einem schreibenden zu einem überschreibenden Statement nennt man

Output-Abhängigkeit. Eine Abhängigkeit von einem lesenden zu einem schreibenden Statement nennt man *Anti-Abhängigkeit.*

Eine Abhängigkeit zwischen Quellstatement S und Zielstatement T, die von Quellindex i auf Zielindex i' zeigt und für die Indexmenge M bzw. M' existiert, wird in der Regel wie folgt notiert:

$$\langle i,\ S\rangle \overset{[Typ]}{\longrightarrow} \langle i',\ T\rangle \text{ für } i \in M,\ i' \in M'$$

Ist die Differenz von i und i' für alle i im Existenzbereich gleich, spricht man von einer *uniformen* Abhängigkeit, da alle Vektoren im Polytopmodell gleiche Ausrichtung und Länge haben.

Beispiel 1.2 *Die in Abbildung 1.1 dargestellten Abhängigkeiten können wie folgt notiert werden:*

$$\langle (i,0),\ S\rangle \longrightarrow \langle (2i+4,3),\ S\rangle \textit{ für } 0 \leq i \leq 2$$

1.2.3 Raum-Zeit-Abbildung

Als Raum-Zeit-Abbildung bzw. *Space-Time Mapping* wird die Abbildung jeder Operation des Quellprogramms auf einen Raum-Zeit-Punkt des parallelen Zielprogramms bezeichnet. Für jedes Statement wird der Raum dabei durch eine Placement-, die Zeit durch eine Schedule-Funktion festgelegt.

Schedule

Der Schedule Θ ist eine Funktion, die jeder Operation des Ursprungsprogramms einen Ganzzahlvektor zuweist, der die logische Zeit repräsentiert, zu der die Operation im parallelen Zielprogramm ausgeführt wird. Für jedes Statement wird eine eigene Schedulefunktion festgelegt, die ein affiner Ausdruck in umgebenden Schleifenvariablen und Parametern sein muss. Ein Schedule ist *gültig*, wenn für alle Abhängigkeiten $\langle i,\ S\rangle \rightarrow \langle i',\ T\rangle$ für $i \in M,\ i' \in M'$ gilt:

$$\forall i \in M,\ i' \in M' : \Theta_S(i) < \Theta_T(i')$$

Beispiel 1.3 *Für das Programm aus Beispiel 1.1 sei für Anweisung S der Schedule $\Theta_S(i,j) = i$ gegeben.*

$$\forall 0 \leq i \leq 2 : \Theta_S(i,0) < \Theta_S(2i+4,3) \Leftrightarrow i < 2i+4 \Leftrightarrow 0 < i+4$$

Der Schedule $\Theta_S(i,j) = i$ ist unter der gegebenen Abhängigkeit gültig.

Placement

Das Placement Π ist analog zum Schedule eine Funktion, die jeder Operation des Ursprungsvektors einen Ganzzahlvektor zuweist, der den virtuellen Prozessor repräsentiert, auf dem die Operation im parallelen Zielprogramm ausgeführt wird. Für jedes Statement wird eine eigene Placementfunktion festgelegt, die ein affiner Ausdruck in umgebenden Schleifenvariablen und Parametern sein muss. Nach Möglichkeit werden voneinander abhängige Operationen auf dem gleichen Prozessor durchgeführt.

Schedule und Placement spannen gemeinsam den kompletten Indexraum eines Statements auf.

Kapitel 2

Optimierung der Speichernutzung funktionaler Programme

Dieses Kapitel erläutert das von Fabien Quilleré und Sanjay Rajopadhye entwickelte Verfahren zur Optimierung der Speichernutzung funktionaler Programme im Polytopmodell [1]. Die Beweise sind im Original nur für Schedules ohne Parallelität ausgeführt, sie wurden entsprechend angepasst und zur besseren Verständlichkeit genauer erläutert.

2.1 Idee

Funktionale Sprachen sind in *Single Assignment*-Form, d.h. es ist unmöglich, Variablen zweimal zu beschreiben: ein einmal zugewiesener Wert darf nicht mehr verändert werden. Wenn ein Compiler einer funktionalen Sprache bei der Codegenerierung daran keine Veränderungen vornimmt, gehen die erzeugten Programme unnötig verschwenderisch mit dem verfügbaren Speicher um. Daher sollte das erzeugte Programm eine Speicherzelle mehrfach beschreiben können. Dabei muss jedoch sichergestellt werden, dass der überschriebene Wert nicht mehr *lebendig* ist, d.h. nicht mehr benötigt wird.

Für Variablen ist diese Analyse auf syntaktischer Basis möglich und in die meisten Compiler integriert, doch wäre es auch wünschenswert, dass Arrays nicht in ihrer gesamten Größe im Speicher gehalten werden müssen, wenn einzelne Arrayzellen nicht mehr lebendig sind. Eine erste Idee ist, statt des gesamten Arrays nur eine *Projektion* auf einige Dimensionen im Speicher zu halten – unter der Bedingung, dass dabei keine lebendigen Zellen überschrieben werden. Nicht immer können Dimensionen komplett ausprojiziert werden, daher entwickelten Quilleré und Rajopadhye sogenannte *Pseudoprojektionen*, die einzelne Dimensionen auf ihre kleinstnotwendige Größe unter Berücksichtigung der Lebenszeiten der Zellen reduzieren.

Ein Verfahren zur Optimierung der Speichernutzung muss demnach zuerst feststellen, wie lange jede Arrayzelle lebendig ist, und kann danach eine Projektion bzw. Pseudoprojektion generieren.

2.2 Voraussetzungen

Quilleré und Rajopadhye definieren ihr Verfahren auf Basis der funktionalen Sprache Alpha [4]. Alpha orientiert sich sehr nah an mathematischen Gleichungen, ein Programm zur Berechnung der Fibonacci-Zahlen bis $(n+1)$ wird semantisch wie folgt definiert:

$$0 \leq i \leq n$$
$$\text{fib}(i) = \begin{cases} 1 & i \leq 1 \\ \text{fib}(i-1) + \text{fib}(i-2) & i \geq 2 \end{cases}$$

Für Arrayindizes auf der Zuweisungsseite sind affine Ausdrücke *nicht* erlaubt; es darf nur eine faktorlose Indexvariable für ausschließlich eine Arraydimension verwendet werden. Weiterhin darf es im ganzen Programm pro Variable bzw. Array nur eine einzige schreibende Anweisung geben. In welcher Reihenfolge die Arrayzellen berechnet werden, kann in Alpha nicht angegeben werden. Es bestimmt selbsttätig einen gültigen Schedule für jede Variable respektive jede Anweisung.

2.3 Lebenszeit

Man habe ein Alpha-Programm mit k-dimensionalem Schedule. Also gibt es für jedes n_Y-dimensionale Array Y eine affine Funktion, die für jede Arrayzelle $z \in D_Y \subseteq \mathbb{Z}^{n_Y}$ den Zeitpunkt der Berechnung bzw. Zuweisung zurückgibt. Diese Funktion kann mittels $k \times n_Y$-Matrix Λ_Y und k-Vektor α_Y definiert werden:

Definition 2.1 *$Y[z]$ wird zu Zeitpunkt $\Lambda_Y z + \alpha_Y$ berechnet und zugewiesen.*

In Alpha erfolgt die Berechnung sequentiell: $\forall z_1, z_2 \in D_Y : z_1 \neq z_2 \Rightarrow \Lambda_Y z_1 \neq \Lambda_Y z_2$; diese Einschränkung wird hier im Weiteren jedoch nicht weiter verwendet.

Die Flow-Abhängigkeiten zwischen den Arrayzellen werden per Abhängigkeitsanalyse errechnet. Damit kann die Lebenszeit von $Y[z]$ unter einer Abhängigkeit F errechnet werden. Sei z' eine Arrayzelle desselben oder eines anderen Arrays X, zu dessen Berechnung der Wert von $Y[z]$ benötigt wird. Gemäß Definition 2.1 wird z' zu Zeitpunkt $\Lambda_X z' + \alpha_X$ geschrieben. Zelle $Y[z]$ ist an diesem Zeitpunkt noch lebendig, ihre Lebenszeit ist also *mindestens* $\Lambda_X z' + \alpha_X - \Lambda_Y z - \alpha_Y$ Zeiteinheiten groß.

Definition 2.2 *Seien alle Zellen $X[z']$, die von $Y[z]$ unter F abhängig sind, in der Menge $D_{F,Y,X}(z)$ enthalten. Die* partielle Lebenszeit *einer Arrayzelle $Y[z]$, ihre Lebenszeit unter einer Abhängigkeit F, ist gegeben durch:*

$$d_{F,Y,X}(z) = \max_{z' \in D_{F,Y,X}(z)} (\Lambda_X z' + \alpha_X - \Lambda_Y z - \alpha_Y)$$

max *bezeichnet hierbei das lexikographische Maximum*[1].

Um die *tatsächliche* Lebenszeit einer Arrayzelle $Y[z]$ zu erhalten, müssen *alle* von Y ausgehenden Abhängigkeiten beachtet und das lexikographische Maximum der entsprechenden partiellen Lebenszeiten berechnet werden:

[1] Auch im gesamten folgenden Text beziehen sich $<$, $>$, $\leq$, $\geq$ und max angewandt auf Vektoren auf die lexikographische Ordnung.

Definition 2.3 *Sei F_Y die Menge aller von Y ausgehenden Flow-Abhängigkeiten. Die* totale Lebenszeit *einer Arrayzelle $Y[z]$ ist gegeben durch:*

$$d_Y(z) = \max_{(F,Y,X)\in F_Y} (d_{F,Y,X}(z)))$$

Für den weiteren Verlauf des Verfahrens ist nicht die Lebenszeit jeder einzelnen Zelle nötig, sondern nur die maximale Lebenszeit aller Zellen eines Arrays:

Definition 2.4 *Sei $D_Y \subseteq \mathbb{Z}^{n_Y}$ die Menge aller Zellen von Y. Die* maximale Lebenszeit *des Arrays Y ist gegeben durch:*

$$d_Y = \max_{z\in D_Y} (d_Y(z))$$

Beispiel 2.1 *Es soll die maximale Lebenszeit des Arrays fib für das oben angegebene Programm zur Berechnung der Fibonacci-Zahlen errechnet werden. Als Schedule für die fib schreibende Anweisung ist $\Lambda_{fib} = (1)$ und $\alpha_{fib} = (0)$ gültig.*

Es existieren zwei Flow-Abhängigkeiten:

$$\begin{array}{lrcll} F_1: & \langle (i),\ fib\rangle & \rightarrow & \langle (i+1),\ fib\rangle & \text{für } 0 \le i \le n-1 \\ F_2: & \langle (i),\ fib\rangle & \rightarrow & \langle (i+2),\ fib\rangle & \text{für } 0 \le i \le n-2 \end{array}$$

Für jede Abhängigkeit wird die partielle Lebenszeit nach Definition 2.2 berechnet:

$$\begin{array}{rcrcl} d_{F_1,fib,fib}(i) & = & \max_i((1)(i+1)+(0)-(1)(i)-(0)) & = & (1) \\ d_{F_2,fib,fib}(i) & = & \max_i((1)(i+2)+(0)-(1)(i)-(0)) & = & (2) \end{array}$$

Entsprechend berechnet man die totale Lebenszeit nach Definition 2.3:

$$d_{fib}(i) = \max((1),(2)) = (2)$$

Da beide Abhängigkeiten uniform sind, ist die maximale Lebenszeit nach Definition 2.4 gleich der totalen Lebenszeit: $d_{fib} = d_{fib}(i)$.

2.4 Projektion

Bei einer Projektion soll eine Anzahl von m_Y Vektoren ρ_i, die *Projektionsvektoren*, angeben, welche Zellen eines Arrays auf eine Speicherzelle projiziert werden sollen. Die Vektoren spannen im Array-Indexraum ein Gitter auf und genau die Punkte bzw. Arrayzellen, die durch dieses Gitter berührt und verbunden werden, sollen durch die Projektion auf die gleiche Speicherzelle projiziert werden. Formal: Zwei Arrayzellen z_1 und z_2 werden auf die gleiche Speicherzelle projiziert genau dann, wenn $\exists q \in \mathbb{Q}^{m_Y} : (z_1 - z_2 = (\rho_1, \ldots, \rho_{m_Y})q)$.

Jeder Projektionsvektor verringert die Anzahl der Dimensionen der Projektion: ein vormals n_Y-dimensionales Array Y wird mit m_Y linear unabhängigen Projektionsvektoren auf ein $(n_Y - m_Y)$-dimensionales Array projiziert.

Definition 2.5 *Eine Projektionsmatrix Π_Y bestimmt, auf welche Speicherzelle eine Arrayzelle von Y abgebildet wird. Zwei Arrayzellen z_1 und z_2 werden auf die gleiche Speicherzelle projiziert, wenn $\Pi_Y z_1 = \Pi_Y z_2$ bzw. $(z_1 - z_2) \in Ker(\Pi_Y)$.*

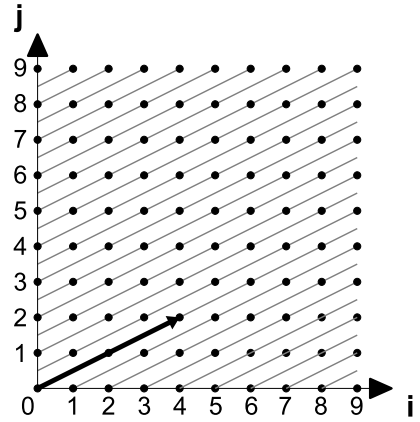

Abbildung 2.1: Projektionsvektor und aufgespanntes Gitter

Für Projektionsmatrix und Projektionsvektoren gilt also die Äquivalenz $\Pi_Y z_1 = \Pi_Y z_2 \Leftrightarrow \exists q \in \mathbb{Q}^{m_Y} : (z_1 - z_2 = (\rho_1, \ldots, \rho_{m_Y})q)$.

Beispiel 2.2 *Abbildung 2.1 stellt ein zweidimensionales Array und einen Projektionsvektor $\rho = (4,2)^T$ sowie das vom Projektionsvektor aufgespannte Gitter dar. Es verbindet alle Punkte $(i_1, j_1)^T$ und $(i_2, j_2)^T$, für die gilt: $\exists q \in \mathbb{Q} : (i_1 - i_2, j_1 - j_2)^T = q\rho$. Eine mögliche Projektionsmatrix ist $\Pi = (1, -2)$, jede Arrayzelle wird auf $\Pi(i,j)^T = (i - 2j)$ projiziert – die Projektion ist eindimensional.*

Es lässt sich errechnen, welche Zelle z' des Ursprungsarrays nach z die nächste ist, die in der Projektion in die gleiche Zelle gespeichert wird, und auch, wie viele Zeiteinheiten z im Speicher gehalten wird.

Definition 2.6 *$Y[z']$ überschreibt in der Projektion Y[z], wenn gilt:*

1. $z, z' \in D_Y$ *(Existenz)*
2. $\Pi_Y z = \Pi_Y z'$ *(Konflikt)*
3. $\Lambda_Y z + \alpha_Y \leq \Lambda_Y z' + \alpha_Y$ *(Ordnung)*
4. $\forall z'' \in D_Y : \Pi_Y z'' \neq \Pi_Y z \vee \neg(\Lambda_Y z < \Lambda_Y z'' < \Lambda_Y z')$ *(Optimierung)*

Dann setze $\sigma_Y(z) := z' - z$. $Y[z]$ wird nach $\Lambda_Y \sigma_Y(z)$ Zeiteinheiten überschrieben.

Damit kann überprüft und gegebenenfalls garantiert werden, dass keine lebendige Arrayzelle überschrieben wird:

Korollar 2.7 *Eine Projektion Π_Y ist unter einem Schedule Λ_Y gültig, wenn keine lebendige Arrayzelle überschrieben wird und kein gleichzeitiger Schreibzugriff auf eine Speicherzelle erfolgt:*

$$\forall z \in D_Y : d_Y \leq \Lambda_Y \sigma_Y(z) \wedge 0 < \Lambda_Y \sigma_Y(z)$$

Beispiel 2.3 *Die einzig mögliche Projektion für das eindimensionale Fibonacci-Programm ist die Projektion auf einen Skalar:* $\Pi_{fib} = (0)$*. Dann ist* $\sigma_{fib}(i) = (1)$*, und* $fib[i]$ *wird nach* $\Lambda_{fib}\sigma_{fib}(i) = (1)$ *Zeiteinheiten überschrieben. Die Gültigkeit der Projektion kann mit der in Beispiel 2.1 berechneten Lebenszeit überprüft werden:*

$$\Lambda_{fib}\sigma_{fib}(i) = (1) \ngeq (2) = d_{fib}$$

Eine Projektion auf einen Skalar ist beim Fibonacci-Programm nicht *gültig.*

2.5 Pseudoprojektion

Nach Beispiel 2.3 lässt sich beim Fibonacci-Programm mittels Projektion kein Speicher sparen. Allerdings ist leicht ersichtlich, dass die Berechnung auch mit nur einem zweielementigen Array gemacht werden könnte. Imperativ könnte das Programm beispielsweise so aussehen:

```
fib[0] = 1; fib[1] = 1;
for (i=2; i<=n; i++)
  fib[i%2] = fib[(i-1)%2] + fib[(i-2)%2];
```

Die Dimension wird also nicht komplett ausprojiziert, sondern auf nur noch zwei Elemente reduziert; dies ist eine sogenannte *Pseudoprojektion*. Die definierenden Vektoren werden dementsprechend nicht mehr Projektionsvektoren, sondern *Allokationsvektoren* genannt.

Es wird wiederum eine Menge von m_Y linear unabhängigen Allokationsvektoren ρ_i der Dimension n_Y ausgewählt (wie solch eine Auswahl gefunden wird, zeigt Abschnitt 2.6) und in einer $(n_Y \times m_Y)$-Matrix $\mathcal{R} = (\rho_1, \rho_2, \ldots, \rho_{m_Y})$ zusammengefasst. Im Gegensatz zu Projektionsvektoren werden bei Allokationsvektoren zwei Arrayzellen jedoch nur dann auf eine gemeinsame Speicherzelle gebildet, wenn sie untereinander durch eine *ganzzahlige* Linearkombination der Allokationsvektoren erreichbar sind: $\exists a \in \mathbb{Z}^{m_Y}$ mit $(z_1 - z_2) = \mathcal{R}a$.

Beispiel 2.4 *Im obigen Beispiel wird jede zweite Zelle des Ursprungsarrays auf eine Speicherzelle abgebildet. Der Allokationsvektor ist demnach* $\rho = (2)$*. Weitere Allokationsvektoren gibt es nicht, also ist* $\mathcal{R} = \rho = (2)$*.*

Bildlich gesprochen spannt $\mathcal{R}$ ein Gitter auf, das alle Punkte in D_Y miteinander verbindet, die auf eine Speicherzelle abgebildet werden sollen. Im Gegensatz zur reinen Projektion reicht es nicht mehr, wenn die Punkte vom Gitter nur durchlaufen werden. Abbildung 2.2 stellt dies in Kontrast zu Abbildung 2.1 graphisch dar: Nur noch Punkte, die auf einer Gitternetzlinie liegen *und* gleich gefärbt sind, sind untereinander durch ein ganzzahlig Vielfaches des Allokationsvektors erreichbar und sollen eine gemeinsame Speicherzelle nutzen.

Ziel ist es, eine Speicherallokationsfunktion Mem_Y zu finden, die genau dies zu einem gegebenen Satz Allokationsvektoren leistet.

Definition 2.8 *Sei* S *eine* $(m_Y \times m_Y)$*-Diagonalmatrix mit* $\forall 1 \leq i \leq m_Y : S_{i,i} \geq 0$ *und* S' *ihre Erweiterung mit Nullzeilen auf* n_Y *Zeilen. Dann existieren unimodulare Matrizen* U^{-1} *und* V^{-1}*, so dass gilt:*

$$S' = U^{-1}\mathcal{R}V^{-1}$$

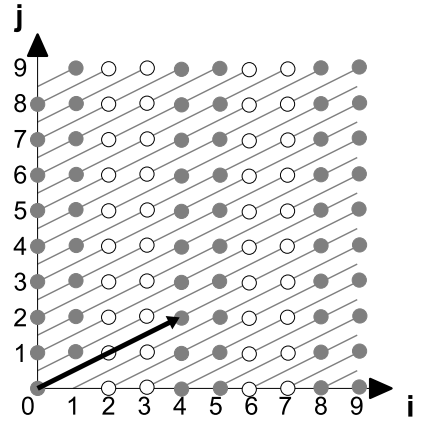

Abbildung 2.2: Allokationsvektor und dadurch verbundene Punkte

Definiere Ω_Y *als die* $(m_Y \times n_Y)$*-Matrix, die aus den ersten* m_Y *Zeilen von* U^{-1} *gebildet wird:*

$$\begin{array}{rl} m_Y \text{ Zeilen} \{ & \\ n_Y - m_Y \text{ Zeilen} \{ & \end{array} \left(\begin{array}{c} \Omega_Y \\ 0 \end{array} \right) = U^{-1}$$

Definiere S_Y *als den durch die Diagonaleinträge von* S *gegebenen Vektor:*

$$S_Y := (s_{1,1}, s_{2,2}, \ldots, s_{m_Y,m_Y})^T$$

Π_Y *ist wie im letzten Abschnitt definiert eine Matrix, für die gilt:*

$$\forall z_1, z_2 \in D_Y : \Pi_Y z_1 = \Pi_Y z_2 \Leftrightarrow (\exists q \in \mathbb{Q}^{n_Y} : (z_1 - z_2 = \mathcal{R}q))$$

Dann kann die Speicherallokationsfunktion für $z \in D_Y$ *wie folgt gebildet werden:*

$$\mathrm{Mem}_Y(z) = \left(\begin{array}{c} \Pi_Y z \\ \Omega_Y z \mod S_Y \end{array} \right)$$

mod *ist hierbei die zeilenweise Anwendung des Modulo-Operators auf Vektoren.*

Lemma 2.9 *Für die in Definition 2.8 gegebene* Mem*-Funktion gilt:*

$$\mathrm{Mem}_Y(z_1) = \mathrm{Mem}_Y(z_2) \Leftrightarrow \exists a \in \mathbb{Z}^{m_Y} : (z_1 - z_2) = \mathcal{R}a$$

Mem bildet also zwei Arrayzellen wie gefordert nur dann auf eine Speicherzelle ab, wenn sie untereinander durch eine ganzzahlige Linearkombination der Allokationsvektoren erreicht werden können.

Beweis.

- $\mathrm{Mem}_Y(z_1) = \mathrm{Mem}_Y(z_2) \Rightarrow \exists a \in \mathbb{Z}^{m_Y} : (z_1 - z_2) = \mathcal{R}a$
  Aus $\mathrm{Mem}_Y(z_1) = \mathrm{Mem}_Y(z_2)$ folgt sofort:

$$\Pi_Y z_1 = \Pi_Y z_2 \tag{2.1}$$
$$\Omega_Y z_1 \mod S_Y = \Omega_Y z_2 \mod S_Y \tag{2.2}$$

Aus (2.1) und der Definition von Π_Y erhält man $\exists q \in \mathbb{Q}^{m_Y} : (z_1 - z_2 = \mathcal{R}q)$ bzw.:

$$\exists q \in \mathbb{Q}^{m_Y} : z_1 = z_2 + \mathcal{R}q \tag{2.3}$$

Es ist noch zu zeigen, dass alle obigen q nur ganze Zahlen sein können.

Nach Definition des modulo-Operators gilt

$$a \mod b = d \mod b \Rightarrow \exists z \in \mathbb{Z} : bz + d = a. \tag{2.4}$$

Sind a, b und d n-Vektoren, die modulo-Operation zeilenweise angewandt und a_i bzw. b_i das i-te Element von a bzw. b, gilt trivialerweise

$$a \mod b = d \mod b \Leftrightarrow \begin{pmatrix} a_1 & \mod b_1 \\ a_2 & \mod b_2 \\ & \vdots \\ a_n & \mod b_n \end{pmatrix} = \begin{pmatrix} d_1 & \mod b_1 \\ d_2 & \mod b_2 \\ & \vdots \\ d_n & \mod b_n \end{pmatrix}$$

Der rechte Teil der Äquivalenz lässt sich mit (2.4) umformen zu:

$$\exists z \in \mathbb{Z}^n \quad : \quad \begin{pmatrix} b_1 z_1 + d_1 \\ b_2 z_2 + d_2 \\ \vdots \\ b_n z_n + d_n \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

$$\exists z \in \mathbb{Z}^n \quad : \quad \begin{pmatrix} b_1 z_1 \\ b_2 z_2 \\ \vdots \\ b_n z_n \end{pmatrix} + \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

$$\exists z \in \mathbb{Z}^n \quad : \quad \begin{pmatrix} b_1 & & & 0 \\ & b_2 & & \\ & & \ddots & \\ 0 & & & b_n \end{pmatrix} z + d = a$$

Diese Umformungen gelten auch für Gleichung (2.2): d entspricht $\Omega_Y z_2$, a entspricht $\Omega_Y z_1$ und b entspricht S_Y. Da S_Y der durch die Diagonaleinträge von S gegebene Vektor ist, folgt damit aus (2.2):

$$\exists x \in \mathbb{Z}^{n_Y} : \Omega_Y z_2 + Sx = \Omega_Y z_1$$

Einsetzen von (2.3):

$$\begin{aligned} \Omega_Y z_2 + Sx &= \Omega_Y (z_2 + \mathcal{R}q) \\ &= \Omega_Y z_2 + \Omega_Y \mathcal{R} q \\ Sx &= \Omega_Y \mathcal{R} q \end{aligned} \tag{2.5}$$

Nach Definition 2.8 ist $\mathcal{R} = US'V$ mit $S' = \begin{pmatrix} S \\ 0 \end{pmatrix} \begin{matrix} \} \, m_Y \text{ Zeilen} \\ \} \, (n_Y - m_Y) \text{ Zeilen} \end{matrix}$

Somit

$$Sx = \Omega_Y U \begin{pmatrix} S \\ 0 \end{pmatrix} V q \tag{2.6}$$

Da Ω_Y nach Definition aus den ersten m_Y Zeilen von U^{-1} besteht, gilt $\Omega_Y U = (E_{m_Y} \underbrace{0 \ldots 0}_{(n_Y - m_Y)})$. Eingesetzt in (2.6):

$$\begin{aligned} Sx &= (E_{m_Y} 0) \begin{pmatrix} S \\ 0 \end{pmatrix} Vq \\ Sx &= SVq \qquad (2.7) \\ x &= Vq \qquad (2.8) \end{aligned}$$

Da V nach Definition 2.8 unimodular ist, also nur Ganzzahlen enthält, und auch x nur aus ganzen Zahlen besteht, folgt $q \in \mathbb{Z}^{m_Y}$.

- $\text{Mem}_Y(z_1) = \text{Mem}_Y(z_2) \Leftarrow \exists a \in \mathbb{Z}^{m_Y} : (z_1 - z_2) = \mathcal{R}a$

  Aus der Voraussetzung $\exists a \in \mathbb{Z}^{m_Y} : (z_1 - z_2) = \mathcal{R}a$ folgt gemäß Definition von Π_Y sofort $\Pi_Y z_1 = \Pi_Y z_2$.

  Es ist noch zu zeigen, dass auch $\Omega_Y z_1 \mod S_Y = \Omega_Y z_2 \mod S_Y$.

  Aus der Voraussetzung folgt ebenfalls

  $$\Omega_Y z_1 - \Omega_Y z_2 = \Omega_Y (z_1 - z_2) = \Omega_Y \mathcal{R}a.$$

  Der rechte Gleichungsteil entspricht dem rechten Teil der Gleichung (2.5). Mit den im ersten Beweisteil gezeigten Umformungen bis zu (2.7) erhält man hier:

  $$\begin{aligned} \Omega_Y z_1 - \Omega_Y z_2 &= SVa \\ \Omega_Y z_1 &= \Omega_Y z_2 + SVa \qquad (2.9) \end{aligned}$$

  Da V eine unimodulare $(m_Y \times m_Y)$-Matrix ist und $a \in \mathbb{Z}^{m_Y}$, ist $x := Va \in \mathbb{Z}^{m_Y}$ ein ganzzahliger Vektor. Eingesetzt in (2.9) erhält man $\Omega_Y z_1 = \Omega_Y z_2 + Sx$ und somit gilt nach Definition des modulo-Operators wie im ersten Beweisteil gezeigt:

  $$\Omega_Y z_1 \mod S_Y = \Omega_Y z_2 \mod S_Y$$

□

Beispiel 2.5 *Beim Fibonacci-Programm soll jede zweite Arrayzelle auf die gleiche Speicherzelle projiziert werden. Wie in Beispiel 2.4 genannt, ist daher* $\mathcal{R} = (2)$.

Dann ist nach Definition 2.8 $\Pi_{fib} = (0)$ *und, da* $(2) = (1)(2)(1)$, *ist* $S = S_{fib} = (2)$ *sowie* $\Omega_{fib} = (1)$.

Damit ist eine Speicherallokationsfunktion gegeben durch

$$\text{Mem}_{fib}(i) = \begin{pmatrix} (0)(i) \\ (1)(i) \mod (2) \end{pmatrix} = \begin{pmatrix} 0 \\ i \mod 2 \end{pmatrix}.$$

Analog zu Projektionen lässt sich auch bei Pseudoprojektionen berechnen, welches $Y[z']$ unter einer Speicherallokationsfunktion Mem_Y $Y[z]$ überschreibt und damit auch, wie viele Zeiteinheiten $Y[z]$ im Speicher gehalten wird. Wiederum kann überprüft und gegebenenfalls garantiert werden, dass unter Λ_Y keine lebendige Arrayzelle überschrieben wird.

Definition 2.10 *$Y[z']$ überschreibt $Y[z]$, wenn gilt:*

1. *$z, z' \in D_Y$ (Existenz)*
2. *$Mem_Y z = Mem_Y z'$ (Konflikt)*
3. *$\Lambda_Y z + \alpha_Y \leq \Lambda_Y z' + \alpha_Y$ (Ordnung)*
4. *$\forall z'' \in D_Y : Mem_Y z'' \neq Mem_Y z \vee \neg(\Lambda_Y z < \Lambda_Y z'' < \Lambda_Y z')$ (Optimierung)*

Dann setze $\sigma_Y(z) := z' - z$. $Y[z]$ wird nach $\Lambda_Y \sigma_Y(z)$ Zeiteinheiten überschrieben.

Korollar 2.11 *Eine Speicherallokation Mem_Y ist unter einem Schedule Λ_Y gültig, wenn keine lebendige Arrayzelle überschrieben wird und kein gleichzeitiger Schreibzugriff auf eine Speicherzelle erfolgt:*

$$\forall z \in D_Y : d_Y \leq \Lambda_Y \sigma_Y(z) \wedge 0 < \Lambda_Y \sigma_Y(z) \text{ mit } \sigma_Y(z) \text{ gemäß Definition 2.10.}$$

2.6 Wahl der Allokationsvektoren

Beim obigen Beispiel des Fibonacci-Programms war die Wahl der Allokationsvektoren durch etwas Nachdenken zu finden, bei komplexeren Programmen ist dies nicht mehr so. Lemma 2.12 zeigt, wie *korrekte* Allokationsvektoren gefunden werden können, Lemma 2.14 zeigt, wie viele Allokationsvektoren maximal gewählt werden können. Lemma 2.15 schließlich zeigt, wie eine maximale Auswahl an korrekten Allokationsvektoren gefunden werden kann.

Lemma 2.12 *Ein Satz Allokationsvektoren ρ_i bestimmt nach Korollar 2.11 gültige Speicherallokation für ein Array Y, wenn für alle ganzzahligen Linearkombinationen η der Allokationsvektoren gilt:*

$$(\eta \neq 0 \Rightarrow 0 \neq \Lambda_Y \eta) \wedge (0 < \Lambda_Y \eta \Rightarrow d_Y \leq \Lambda_Y \eta)$$

Eine Linearkombination η der Allokationsvektoren verbindet wie im vorherigen Abschnitt beschrieben zwei Arrayzellen, die auf eine Speicherzelle abgebildet werden. Sei eine Arrayzelle mit z bezeichnet, so wird die Zelle $(z + \eta)$ in die gleiche Speicherzelle geschrieben. z wird nach Definition 2.1 zu Zeitpunkt $\Lambda_Y z + \alpha_Y$ beschrieben, $(z + \eta)$ zu Zeitpunkt $\Lambda_Y(z + \eta) + \alpha_Y = \Lambda_Y z + \Lambda_Y \eta + \alpha_Y$. Die Differenz zwischen beiden Zeitpunkten beträgt somit $\Lambda_Y \eta$ Zeitschritte.

Der erste Teil der Konjunktion in Lemma 2.12 verhindert somit, dass eine Speicherallokation Schreibkonflikte auslöst: Zwei verschiedene Arrayzellen dürfen nicht zur gleichen Zeit in eine Speicherzelle geschrieben werden.

Der zweite Teil der Konjunktion verhindert, dass eine Speicherallokation Arrayzellen in die gleiche Speicherzelle schreibt, deren Differenz der Zuweisungszeiten kleiner als die maximale Lebenszeit d_Y ist. Somit ist garantiert, dass jede Arrayzelle mindestens d_Y Zeitschritte im Speicher gehalten wird.

Beweis.

Zu zeigen: $\forall \eta : (\eta \neq 0 \Rightarrow 0 \neq \Lambda_Y \eta) \wedge (0 < \Lambda_Y \eta \Rightarrow d_Y \leq \Lambda_Y \eta) \Rightarrow \forall z \in D_Y : d_Y \leq \Lambda_Y \sigma_Y(z) \wedge 0 < \Lambda_Y \sigma_Y(z)$

$z' = z + \sigma_Y(z)$ ist nach Definition 2.10 die Arrayzelle, die z im Speicher überschreibt. z' wird nach Definition zeitlich nicht vor z geschrieben, also gilt:

$$0 \leq \Lambda_Y z' - \Lambda_Y z = \Lambda_Y(z' - z) \tag{2.10}$$

Da die Speicherallokation z' und z in die gleiche Speicherzelle abbildet, können sie durch eine ganzzahlige Linearkombination η der Allokationsvektoren verbunden werden: $\eta = z' - z = \sigma_Y(z)$. Eingesetzt in (2.10):

$$0 \leq \Lambda_Y(z' - z) = \Lambda_Y \eta = \Lambda_Y \sigma_Y(z)$$

Da z' und z nach Definition verschieden sind, ist $\eta \neq 0$. Somit ist nach Voraussetzung auch $\Lambda_Y \eta \neq 0$:

$$0 < \Lambda_Y \eta = \Lambda_Y \sigma_Y(z), \text{ q.e.d.} \tag{2.11}$$

Noch zu zeigen: $d_Y \leq \Lambda_Y \sigma_Y(z)$

Es gilt nach (2.11) $0 < \Lambda_Y \eta$. Aus der Voraussetzung folgt damit sofort $d_Y \leq \Lambda_Y \eta$ und somit $d_Y \leq \Lambda_Y \sigma_Y(z)$. □

Jeder zusätzliche, linear unabhängige Allokationsvektor verringert den von der Projektion benötigten Speicher. Es ist daher vorteilhaft, möglichst viele Allokationsvektoren zu finden; dabei muss die Speicherallokation aber nach Korollar 2.11 gültig sein.

Sei m_Y weiterhin die Anzahl der Allokationsvektoren ρ_i. Eine obere Schranke ist offensichtlich durch die Dimensionalität n_Y des Arrays Y vorgegeben: $m_Y \leq n_Y$. Unter Berücksichtigung der Gültigkeit gibt es noch eine weitere obere Schranke:

Definition 2.13 *Als* Grad *eines Vektors wird die Position seines ersten Eintrags ungleich* 0 *bezeichnet. Mit anderen Worten: Der Grad eines Vektors ist die Anzahl seiner führenden Nullen erhöht um* 1*.*

Lemma 2.14 *Die Anzahl der gültigen, linear unabhängigen Allokationsvektoren für ein Array Y ist nach oben beschränkt durch den Grad der maximalen Lebenszeit d_Y.*

Widerspruchsbeweis. Sei m_Y Grad der maximalen Lebenszeit d_Y, seien ρ_i $(m_Y + 1)$ linear unabhängige Allokationsvektoren.

Sei $\forall 1 \leq i \leq m_Y + 1 : \lambda_i = \Lambda_Y \rho_i$.

Nach Lemma 2.12 muss für alle ganzzahligen Linearkombinationen η einer gültigen Auswahl Allokationsvektoren gelten:

$$\eta \neq 0 \Rightarrow \Lambda_Y \eta \neq 0 \tag{2.12}$$

Gilt diese Bedingung für die hier betrachteten ρ_i nicht, ist die Auswahl ungültig – Widerspruch zur Voraussetzung, Beweisende. Gilt die Bedingung jedoch, muss der Widerspruch anders herbeigeführt werden: Aus (2.12) folgt dann insbesondere, dass für zwei Allokationsvektoren ρ_i, ρ_j gilt:

$$\forall \mu_i, \mu_j \in \mathbb{Z} : \mu_i \rho_i + \mu_j \rho_j \neq 0 \Rightarrow \Lambda_Y(\mu_i \rho_i + \mu_j \rho_j) \neq 0$$

Nach Voraussetzung sind die ρ_i linear unabhängig, für $i \neq j$ gilt also in jedem Fall $\mu_i\rho_i + \mu_j\rho_j \neq 0$. Somit gilt also nach der Implikation für alle $\mu_i, \mu_j \in \mathbb{Z}, i \neq j$:

$$\begin{aligned} \Lambda_Y(\mu_i\rho_i + \mu_j\rho_j) &\neq 0 \\ \Lambda_Y\mu_i\rho_i + \Lambda_Y\mu_j\rho_j &\neq 0 \\ \mu_i\Lambda_Y\rho_i + \mu_j\Lambda_Y\rho_j &\neq 0 \\ \mu_i\lambda_i + \mu_j\lambda_j &\neq 0 \end{aligned}$$

Mit anderen Worten: Die λ_i sind linear unabhängig. Eine $(k \times (m_Y + 1))$-Matrix M definiert durch $M = (\lambda_1, \ldots, \lambda_{m_Y+1})$ hat daher vollen Spaltenrang. Wenn es ein $\mu_0 \in \mathbb{Z}^{m_Y+1}$ gibt, so dass $0 < M\mu_0 < d_Y$ gilt, sind die Allokationsvektoren nach Lemma 2.12 ungültig. Dies ist zu zeigen.

Man teile M nach Zeile m_Y in eine Matrix M' mit m_Y Zeilen und eine Matrix A mit $(k - m_Y)$ Zeilen auf. Da M vollen Spaltenrang hat, gibt es kein $x \neq 0$ mit $Mx = 0$. Daraus folgt für die neuen Teilmatrizen:

$$\forall x \in \mathbb{Z}^{m+1}, x \neq 0 : M'x = 0 \Rightarrow Ax \neq 0 \tag{2.13}$$

Sei $S \subset \mathbb{Z}^{m+1}$, $S := \{x \in \mathbb{Z}^{m+1} \mid x \neq 0 \wedge M'x = 0 \wedge Ax > 0\}$. Da M' eine $(m_Y \times (m_Y + 1))$-Matrix ist und keinen vollen Rang haben kann, ist die Dimension des Kerns mindestens 1: Es gibt außer $x = 0$ Lösungen für $M'x = 0$. Daraus und aus (2.13) folgt: $S \neq \emptyset$.

Betrachten wir ein $x \in S$: Nach Definition gilt $M'x = 0$ und $Ax =: p > 0$, also

$$Mx = (\underbrace{0, \ldots, 0}_{m_Y}, \underbrace{p^T}_{(k-m_Y)})^T > 0.$$

d_Y hat jedoch nur $(m_Y - 1)$ führende Nullen, $Mx < d_Y$. x ist eine Lösung für μ_0. □

Kann man m_Y linear unabhängige und gültige Allokationsvektoren finden, ist das demnach eine optimale Auswahl.

Lemma 2.15 *Sei m_Y Grad der maximalen Lebenszeit $d_Y > 0$, seien $e_i := \left(\underbrace{0, \ldots, 0}_{(i-1)}, 1, \underbrace{0, \ldots, 0}_{(k-i)}\right)$ kanonische Basisvektoren von $\mathbb{Z}^k$. Dann lassen sich m_Y Vektoren ρ_i finden, für die gilt:*

$$\forall 1 \leq i < m_Y : \Lambda_Y\rho_i = e_i,\ \Lambda_Y\rho_{m_Y} = d_Y$$

So gefundene ρ_i sind gültige Allokationsvektoren.

Beweis. Nach Lemma 2.12 sind die Allokationsvektoren ρ_i gültig, wenn für jede Linearkombination η von ihnen gilt:

$$(\eta \neq 0 \Rightarrow 0 \neq \Lambda_Y\eta) \wedge (0 < \Lambda_Y\eta \Rightarrow d_Y \leq \Lambda_Y\eta)$$

Mit $\mu_i \in \mathbb{Z}$ sei:

$$\begin{aligned} \eta &= \sum_{i=1}^{m_Y} \mu_i \rho_i \\ \Lambda_Y \eta &= \Lambda_Y \left(\sum_{i=1}^{m_Y} \mu_i \rho_i \right) = \sum_{i=1}^{m_Y} \Lambda_Y \mu_i \rho_i = \sum_{i=1}^{m_Y} \mu_i \Lambda_Y \rho_i \\ &= \sum_{i=1}^{m_Y - 1} \mu_i \Lambda_Y \rho_i + \mu_{m_Y} \Lambda_Y \rho_{m_Y} \end{aligned} \tag{2.14}$$

Nach Definition ist $\forall 1 \leq i < m_Y : \Lambda_Y \rho_i = e_i$, $\Lambda_Y \rho_{m_Y} = d_Y$. Eingesetzt in (2.14) erhält man:

$$\begin{aligned} \Lambda_Y \eta &= \sum_{i=1}^{m_Y - 1} \mu_i e_i + \mu_{m_Y} d_Y \\ &= \left(\mu_1, \ldots, \mu_{m_Y - 1}, \underbrace{0, \ldots, 0}_{k - m_Y + 1} \right)^T + \mu_{m_Y} d_Y \end{aligned} \tag{2.15}$$

Somit $\Lambda_Y \eta = 0 \Leftrightarrow \mu_1 = \mu_2 = \cdots = \mu_{m_Y} = 0 \Leftrightarrow \eta = 0$, mit anderen Worten:

$$\eta \neq 0 \Rightarrow 0 \neq \Lambda_Y \eta$$

Noch zu zeigen: $0 < \Lambda_Y \eta \Rightarrow d_Y \leq \Lambda_Y \eta$. Widerspruchsbeweis: Sei $0 < \Lambda_Y \eta < d_Y$. Einsetzen von (2.15) ergibt

$$0 < \left(\mu_1, \ldots, \mu_{m_Y - 1}, \underbrace{0, \ldots, 0}_{k - m_Y + 1} \right)^T + \mu_{m_Y} d_Y < d_Y.$$

Da m_Y der Grad von d_Y ist, kann die Ungleichung nur für $\mu_1 = \mu_2 = \cdots = \mu_{m_Y - 1} = 0$ erfüllt sein:

$$0 < \mu_{m_Y} d_Y < d_Y.$$

Diese Ungleichung ist nicht erfüllbar für $d_Y > 0$, $\mu_{m_Y} \in \mathbb{Z}$. □

Eine optimale Auswahl an Allokationsvektoren kann somit bestimmt werden.

2.7 Anwendung

Nach den Erklärungen und Beweisen hier eine Übersicht über die Schritte, die nach den Abschnitten 2.5 und 2.6 zur Optimierung der Speichernutzung eines n_Y-dimensionalen Arrays Y durchgeführt werden müssen.

1. Bestimmung der *maximalen Lebenszeit* d_Y aller Arrayzellen.

Dafür nötig:

$D_{F,Y,X}(z)$	Zellen von X, die unter F von Zelle z von Y abhängen;
$\Lambda_Y z + \alpha_Y$	Zeitpunkt, an dem z geschrieben wird;
F_Y	Menge aller von Y ausgehenden Flow-Abhängigkeiten;
$D_Y \subseteq \mathbb{Z}^{n_Y}$	Menge aller Zellen von Y.

Zunächst wird die Lebenszeit einer Arrayzelle z unter Abhängigkeit F (*partielle Lebenszeit*) berechnet:

$$d_{F,Y,X}(z) = \max_{z' \in D_{F,Y,X}(z)} (\Lambda_X z' + \alpha_X - \Lambda_Y z - \alpha_Y)$$

Damit kann die Lebenszeit *einer* Arrayzelle z von Y (*totale Lebenszeit*) und in einem dritten Schritt die *maximale* Lebenszeit d_Y *aller* Arrayzellen bestimmt werden:

$$\begin{aligned} d_Y(z) &= \max_{(F,Y,X) \in F_Y} (d_{F,Y,X}(z)) \\ d_Y &= \max_{z \in D_Y} (d_Y(z)) \end{aligned}$$

2. Bestimmung der Allokationsvektormatrix $\mathcal{R}$.

   Dafür nötig:

$0 < d_Y \in \mathbb{Z}^k$	maximale Lebenszeit aus Schritt 1;
$\Lambda_Y \in \mathbb{Z}^{k \times n_Y}$	Schedule des auf Y schreibenden Statements.

   Sei $1 \leq m_Y \leq n_Y$ die Position der ersten Stelle in d_Y, die nicht 0 ist,
   sei $\forall 1 \leq i \leq k : e_i$ kanonischer Basisvektor von $\mathbb{Z}^k$, so dass $e_i > e_{i+1}$.

   Die Allokationsvektoren ρ_i sowie die Allokationsvektormatrix $\mathcal{R}$ sind dann wie folgt definiert:

   $$\begin{aligned} \Lambda_Y \rho_i &= e_i \text{ für } 1 \leq i \leq m_Y - 1 \\ \Lambda_Y \rho_{m_Y} &= d_Y \\ \mathcal{R} &= \begin{pmatrix} \rho_1 & \rho_2 & \dots & \rho_{m_Y} \end{pmatrix} \end{aligned}$$

3. Finde eine Matrix Π_Y, so dass gilt: $\forall 1 \leq i \leq m_Y : \Pi_Y \rho_i = 0$.

4. Berechnen der Matrizen Ω_Y und S_Y.

   Sei $S = \begin{pmatrix} s_1 & & & 0 \\ & s_2 & & \\ & & \ddots & \\ 0 & & & s_{m_Y} \end{pmatrix}$ Diagonalmatrix mit $\forall 1 \leq i \leq m_Y : s_i \geq 0$.

   Finde ein $U \in \mathbb{Z}^{n_Y \times n_Y}$, $V \in \mathbb{Z}^{m_Y \times m_Y}$ und $S' \in \mathbb{Z}^{n_Y \times m_Y}$, so dass $\mathcal{R} = US'V$ und $S' = \begin{pmatrix} S \\ 0 \end{pmatrix}$.

   Ferner sei U^{-1} Inverses von U. Dann ist Ω_Y die Untermatrix von U^{-1}, die die ersten m_Y Zeilen bildet, $S_Y := (s_1, s_2, \dots, s_{m_Y})^T$.

5. Konstruktion der Mem-Funktion mit

   $$\text{Mem}_Y(z) = \begin{pmatrix} \Pi_Y z \\ (\Omega_Y z) \mod S_Y \end{pmatrix}$$

   Die mod-Funktion ist zeilenweise anzuwenden.

2.8 Beispiel

Die im vorherigen Abschnitt gezeigte Vorgehensweise wird hier zum besseren Verständnis an einem Beispiel demonstriert.

Betrachten wird das folgende funktionale Programm zur Berechnung des Pascalschen Dreiecks:

$$0 \leq i \leq n,\ 0 \leq j \leq i$$
$$PD[i,j] = \begin{cases} 1 & j = 0 \\ PD[i-1, j-1] + PD[i-1, j] & j < i \\ 1 & j = i \\ 0 & j > i \end{cases}$$

Es gilt $i \leq n$, $j \leq n$.
Die Berechnung soll sequentiell ausgeführt werden, damit ist $\Lambda_Y = E^{2\times 2}$ ein gültiger Schedule. In diesem Fall gilt außerdem $\Lambda_Y^{-r} = \Lambda_Y$.

Es existieren zwei Flow-Abhängigkeiten:

$$\begin{array}{lcll} F_1 : \langle (i,j),\ PD\rangle & \rightarrow & \langle (i+1,j),\ PD\rangle & \text{für } 1 \leq i < n,\ 1 \leq j \leq i \\ F_2 : \langle (i,j),\ PD\rangle & \rightarrow & \langle (i+1,j+1),\ PD\rangle & \text{für } 0 \leq i < n,\ 0 \leq j < i \end{array}$$

Nun kann versucht werden, die Speichernutzung nach der Vorgehensweise aus Abschnitt 2.7 zu optimieren.

1. Bestimmung der maximalen Lebenszeit d_{PD} aller Arrayzellen.

   Zunächst wird die partielle Lebenszeit für jede Abhängigkeit berechnet:

   $$\begin{array}{lll} d_{PD,F_1}(z) = & \max_{z' \in D_{F_1,PD,PD}(z)}(\Lambda_{PD} z' + \alpha_{PD} - \Lambda_{PD} z - \alpha_{PD}) & = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ d_{PD,F_2}(z) = & \max_{z' \in D_{F_2,PD,PD}(z)}(\Lambda_{PD} z' + \alpha_{PD} - \Lambda_{PD} z - \alpha_{PD}) & = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \end{array}$$

   Da beide Abhängigkeiten uniform sind, sind die partiellen Lebenszeiten unabhängig von den Quellarrayzellen. Daher ergibt sich sofort:

   $$d_{PD} = d_{PD}(z) = \max(d_{PD,F_1}(z), d_{PD,F_2}(z)) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

2. Bestimmung der Allokationsvektormatrix $\mathcal{R}$.

   Aus Schritt 1 folgt $m_{PD} = 1$, es gibt nur einen Allokationsvektor ρ mit

   $$\rho_{m_{PD}} = \Lambda_{PD}^{-r} d_{PD} = E^{2\times 2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} =: \mathcal{R}$$

3. Finde Π_{PD}, so dass gilt: $\Pi_{PD}\rho = 0$.

   $\Pi_Y = (1, -1)$ ist eine Lösung.

4. Berechnen der Matrizen Ω_{PD} und S_{PD}.

   Finde ein U und V, so dass $\mathcal{R}$ in Diagonalmatrix S überführt wird:

$$S = U^{-1}\mathcal{R}V^{-1} = \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} (1) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

   Somit $\Omega_{PD} = (1, 0)$, $S_{PD} = (1)$.

5. Konstruktion der Mem-Funktion.

$$\mathrm{Mem}_{PD}(z) = \begin{pmatrix} \Pi_{PD} z \\ (\Omega_{PD} z) \mod S_{PD} \end{pmatrix} = \begin{pmatrix} (1, -1) \cdot z \\ ((1, 0) \cdot z) \mod (1) \end{pmatrix}$$

   Setzt man wie im Originalprogramm $z = \begin{pmatrix} i \\ j \end{pmatrix}$ und multipliziert aus, erhält man

$$\mathrm{Mem}_{PD} \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} i - j \\ i \mod 1 \end{pmatrix} = \begin{pmatrix} i - j \\ 0 \end{pmatrix}$$

   Dimensionen der Größe 1 können entfallen. Es können daher alle Arrayzugriffe $PD(i, j)$ im Programm stattdessen in $PD(i - j)$ umgeschrieben werden, ohne dass ein Berechnungsschritt falsche Daten bekommt. Die Größe des Arrays verringert sich durch die Optimierung der Speichernutzung von $n \times n$ auf n.

   Das Programm nach der Transformation:

$$0 \le i \le n,\ 0 \le j \le i$$

$$PD[i - j] = \begin{cases} 1 & j = 0 \\ PD[i - j] + PD[i - j - 1] & j < i \\ 1 & j = i \\ 0 & j > i \end{cases}$$

Kapitel 3

Erweiterung auf imperative, automatisch parallelisierte Programme

Die im vorherigen Kapitel beschriebenen Möglichkeiten zur Optimierung der Speichernutzung funktionaler Programme und zur Auswahl dazu nötiger Allokationsvektoren werden in diesem Kapitel auf imperative, mit Hilfe des Polytopmodells parallelisierte Programme erweitert.

Gegenüber den in Abschnitt 2.2 vorgestellten Alpha-Programmen kann ein imperatives Parallelprogramm im Polytopmodell folgende für die Speicheroptimierung relevanten Erweiterungen nutzen:

- Schreibzugriffe auf Variablen sind nicht auf eine einzige Anweisung begrenzt.

  Die Schreibzugriffe auf ein Array Y besitzen keinen eindeutigen Schedule Λ_Y, sondern für jedes Y zuweisende Statement S einen Schedule Θ_S[1].

- Schleifen werden explizit notiert, der Schleifenindexraum ist nicht in jedem Fall gleich dem Arrayindexraum.

  Bei Alpha-Programmen gibt $(\Lambda_Y z + \alpha_Y)$ zu einer Arrayzelle $z \in D_Y \in \mathbb{Z}^{n_Y}$ zurück, zu welchem Zeitpunkt in z geschrieben wird. Im Imperativen gibt $\Theta_S(i)$ für eine Anweisung S zurück, zu welchem Zeitpunkt sie mit Index $i \in D_S \subseteq \mathbb{Z}^{c_S}$ (c_S Anzahl der umgebenden Schleifenvariablen) ausgeführt wird. Die Information, wann eine bestimmte Arrayzelle beschrieben wird, ist nicht explizit zugänglich und im Allgemeinen aufgrund von Mehrfachzuweisungen auch nicht eindeutig.

- Als Arrayindizes können auch auf der Zuweisungsseite affine Ausdrücke, bestehend aus umgebenden Schleifenvariablen und Parametern, verwendet werden.

  Die Arrayzelle z, die von einer Instanz von Anweisung S bei Index $i \in D_S$ geschrieben wird, wird von der *Access*-Funktion geliefert: $z = \mathrm{Acc}_S(i)$.

[1] Durch abhängigkeitseliminierende Techniken wie das *Index Set Splitting* [2], die originale Anweisungen des Quellprogramms in mehrere Anweisungen im Zielprogramm aufspalten, kann es sogar mehrere Schedules pro Statement geben.

Zur einfacheren Bezeichnung erhalten die von Schleifenvariablen abhängigen und konstanten Anteile von Schedule und Access-Funktion eigene Bezeichnungen: Θ_S wird aufgeteilt in seinen für gegebene Parameter konstanten Anteil α_S und dem vom Index abhängigen Teil τ_S: $\Theta_S(i) = \tau_S i + \alpha_S$. Acc_S wird aufgeteilt in seinen für gegebene Parameter konstanten Anteil ζ_S und dem vom Index abhängigen Teil $\mathcal{A}_S$: $\mathrm{Acc}_S(i) = \mathcal{A}_S i + \zeta_S$.

Die Menge der auf Y schreibenden Anweisungen wird mit $\mathcal{S}_Y$ bezeichnet.

3.1 Voraussetzungen

Um eine Speicherallokation auf Gültigkeit zu überprüfen (siehe Lemma 3.11) und entsprechende Allokationsvektoren zu finden, muss ein imperatives Programm ausschließlich *indexmonoton* auf das zu optimierende Array schreiben. Man kann sich bildlich eine Schreibmarke wie bei Bandmaschinen vorstellen, die im Array die nächste Schreibposition bezeichnet: Bei streng indexmonotonen Arrayzugriffen darf die Schreibmarke gleichmäßig vorrücken, aber niemals die Richtung wechseln oder still stehen.

Definition 3.1 *Sei $D_S \subseteq \mathbb{Z}^{c_S}$ wie bisher der Indexraum einer Anweisung S. Die Erweiterung des Indexraums um alle Indizes, die zeitlich später als die Indizes in D_S ausgeführt würden – auch, wenn sie tatsächlich niemals ausgeführt werden –, sei bezeichnet mit D_S^+:*

$$D_S^+ = \{i \mid i \in \mathbb{Z}^{c_S} \wedge (\exists i_0 \in D_S : \Theta_S i_0 \leq \Theta_S i)\}$$

Alle Schreibzugriffe auf ein Array Y sind streng indexmonoton steigend, *wenn nach einem Schreibzugriff auf eine Arayzelle z nur noch in Zellen z' mit $z < z'$ geschrieben wird. Formal ausgedrückt:*

$$\forall i_S \in D_S^+, i_T \in D_T^+, S, T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \Rightarrow \mathrm{Acc}_S(i_S) < \mathrm{Acc}_T(i_T)$$

Streng indexmonoton fallende *Schreibzugriffe sind analog zu definieren:*

$$\forall i_S \in D_S^+, i_T \in D_T^+, S, T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \Rightarrow \mathrm{Acc}_S(i_S) > \mathrm{Acc}_T(i_T)$$

Hinweis: Im Folgenden wird *strenge* Indexmonotonie der besseren Lesbarkeit wegen oftmals nur einfach mit Indexmonotonie bezeichnet.

Beispiel 3.1 *Es soll überprüft werden, ob im folgenden imperativen Programm zur Berechnung der Fibonacci-Zahlen auf Array fib streng indexmonoton steigend zugegriffen wird:*

```
S: fib[0] = 1;
T: fib[1] = 1;
   for (i=2; i<=n; i++)
U:   fib[i] = fib[i-1] + fib[i-2];
```

Als Schedules sind $\Theta_S() = \Theta_T() = (0)$, $\Theta_U(i) = (i-1)$ gültig.

Die Indexräume für die Statements sind $D_S = D_T = \{()\}$, $D_U = \{(i) \mid 2 \leq i \leq n\}$; $\mathrm{Acc}_S() = (0)$, $\mathrm{Acc}_T() = (1)$, $\mathrm{Acc}_U(i) = (i)$. *Die Arrayzugriffe sind streng indexmonoton steigend, wenn alle folgenden Implikationen erfüllt sind:*

$$\forall i_S \in D_S^+, i_T \in D_T^+ : \quad (0) < (0) \Rightarrow (0) < (1) \tag{3.1}$$
$$\forall i_T \in D_T^+, i_S \in D_S^+ : \quad (0) < (0) \Rightarrow (1) < (0) \tag{3.2}$$
$$\forall i_S \in D_S^+, i_U \in D_U^+ : \quad (0) < (i_U - 1) \Rightarrow (0) < (i_U) \tag{3.3}$$
$$\forall i_U \in D_U^+, i_S \in D_S^+ : \quad (i_U - 1) < (0) \Rightarrow (i_U) < (0) \tag{3.4}$$
$$\forall i_T \in D_T^+, i_U \in D_U^+ : \quad (0) < (i_U - 1) \Rightarrow (1) < (i_U) \tag{3.5}$$
$$\forall i_U \in D_U^+, i_T \in D_T^+ : \quad (i_U - 1) < (0) \Rightarrow (i_U) < (0) \tag{3.6}$$
$$\forall i_U \in D_U^+, i'_U \in D_U^+ : \quad (i_U - 1) < (i'_U - 1) \Rightarrow (i_U) < (i'_U) \tag{3.7}$$

Da D_S^+ *einelementig ist, ist die Implikationen zwischen Elementen von* D_S^+ *trivial erfüllt und nicht aufgeführt, ebenso die Implikationen zwischen Elementen von* D_T^+. (3.1), (3.2), (3.3), (3.5) *und* (3.7) *sind offensichtlich erfüllt. Da für alle* $(i_U) \in D_U^+ : i_U \geq 2$ *gilt, sind auch die Bedingungsseiten von* (3.4) *sowie* (3.6) *niemals wahr und die Gesamtimplikationen somit immer erfüllt: Die Arrayzugriffe auf* fib *sind streng indexmonoton steigend.*

Sei das Beispielprogramm für den Moment noch um ein schleifenloses Statement `V: fib[n+4] = fib[n] + fib[n-1]` *erweitert mit* $\Theta_V = (n)$. *Dann ist für die oben definierte Indexmonotonie der Arrayzugriffe auch*

$$\forall i_V \in D_V^+, (i_U) \in D_U^+ : (n) < (i_U - 1) \Rightarrow (n + 4) < (i_U)$$

zu erfüllen. Für $i_U = n + 2$ *ist die Implikation jedoch falsch, somit ist das erweiterte Programm* nicht *indexmonoton. Würde* V *stattdessen in* $fib[n + 1]$ *schreiben* oder *zum Zeitpunkt* $(n + 4)$ *ausgeführt, wäre die Indexmonotonie wieder hergestellt.*

Im zweiten Teil des Beispiels ist zu beachten, dass $i_U = n + 2 \notin D_U$. Die Indexmonotonie wird jedoch auf D_U^+ überprüft. Im nächsten Abschnitt wird an einem Beispiel gezeigt, warum dies sinnvoll ist.

Die Einschränkung des Gültigkeitsbegriffs von Speicherallokationen auf ausschließlich indexmonoton schreibende Programme ist auf den ersten Blick sehr hart. Tatsächlich sind jedoch Arrayzugriffe mit begrenzter maximaler Lebenszeit eines Arrays oft auch indexmonoton: beispielsweise ist jedes perfekt verschachtelte Programm indexmonoton, wenn bei den Zuweisungen jede Schleifendimension eine entsprechende Arraydimension erhält (*Single Assignment*-Form), nur die innersten Schleifen parallel und die äußeren Schleifen sequentiell ausgeführt werden (*synchrone* bzw. *horizontale* Parallelität):

```
for (i=0; i<=n; i++) {
  parallel for(j=0; j<=m; j++) {
    parallel for(k=1; k<=o; k++) {
      A[i,j,k] = ...;
    }
  }
}
```

Nicht indexmonoton ist es, die j- ohne die k- oder die i- ohne die j- und k-Schleifen parallel auszuführen. Auch ein Vertauschen der Reihenfolge der Arrayindizes kann Indexmonotonie gegebenenfalls zerstören oder im Umkehrfall herstellen. Ein Beispiel dazu wird in Abschnitt 3.8.1 gezeigt.

Um in folgenden Definitionen und Beweisen Fallunterscheidungen zwischen indexmonoton fallenden und indexmonoton steigenden Zugriffen zu vermeiden, wird eine Hilfsfunktion ψ_Y eingeführt.

Definition 3.2 *Die Indexmonotoniekonstante ψ_Y sei wie folgt festgelegt:*

$$\psi_Y = \begin{cases} 1 & \textit{Schreibzugriffe auf } Y \textit{ indexmonoton steigend} \\ -1 & \textit{Schreibzugriffe auf } Y \textit{ indexmonoton fallend} \end{cases}$$

Mittels dieser Definition kann auch sofort eine Beziehung zwischen einer Zeitspanne Δt und dem Verschieben der Schreibposition um Δz Zellen in dieser Zeit gefunden werden:

Lemma 3.3 *Seien $t_E, t_0 \in \mathbb{Z}^k$ Zeitpunkte mit k Dimensionalität des Schedules. Es gelte $t_E > t_0$, t_E liegt also zeitlich nach t_0. Für die Menge der Indexschritte Δi, die von einem Statement S innerhalb der Zeitspanne $\Delta t = t_E - t_0$ gemacht werden können, gilt*

$$\Delta t = \tau_S \Delta i.$$

Die Verschiebung der Schreibmarke eines Arrays Y von einem Statement S innerhalb einer Anzahl von Indexschritten $\Delta i = i_E - i_0$, wobei i_E zeitlich nach i_0 ausgeführt wird, ist gegeben durch

$$\Delta z' = \mathcal{A}_S(\Delta i).$$

Es werden im Betrag

$$\Delta z = \psi_Y \Delta z' = \psi_Y \mathcal{A}_S(\Delta i)$$

Zellen geschrieben oder übersprungen.

Es gilt außerdem: Wenn $\Delta t > 0$, dann ist für alle in dieser Zeitspanne möglichen Indexschritte und alle Statements $\Delta z > 0$.

Beweis. Der Beweis der Behauptungen erfolgt in gleicher Reihenfolge wie im Lemma.

Gemäß Definition ist der Zeitpunkt t, an dem Statement S mit Index i ausgeführt wird, gegeben durch

$$t = \Theta_S(i) = \tau_S i + \alpha_S.$$

Für $\Delta t = t_E - t_0$ gilt somit, wenn i_0 zur Zeit t_0 und i_E zur Zeit t_E ausgeführt wird:

$$\begin{aligned} \Delta t &= t_E - t_0 = \tau_S i_E + \alpha_S - \tau_S i_0 - \alpha_S \\ &= \tau_S i_E - \tau_S i_0 = \tau_S (i_E - i_0) \end{aligned}$$

Die letzte Umformung ist möglich, da der variable Teil des Schedule eine lineare Abbildung ist.

i_0 ist ein zur Zeit t_0, i_E ein zu t_E ausgeführter Index. In der Zeit von t_0 bis t_E führt Statement S also $\Delta i = i_E - i_0$ Indexschritte aus, q.e.d.

Es sei weiterhin $\Delta i = i_E - i_0$, sei außerdem z_E die von S mit Index i_E geschriebene Arrayzelle, z_0 analog. Die Verschiebung der Schreibmarke ist gegeben durch die Differenz der Arrayzellindizes:

$$\Delta z' = z_E - z_0 \tag{3.8}$$

Gemäß Definition der Access-Funktion wird die Zelle z von Statement S mit Index i geschrieben genau dann, wenn

$$z = \mathrm{Acc}_S(i) = \mathcal{A}_S(i) + \zeta_S.$$

Einsetzen in (3.8):

$$\begin{aligned} \Delta z' &= z_E - z_0 = \mathcal{A}_S(i_E) + \zeta_S - \mathcal{A}_S(i_0) - \zeta_S \\ &= \mathcal{A}_S(i_E) - \mathcal{A}_S(i_0) = \mathcal{A}_S(i_E - i_0) \\ &= \mathcal{A}_S(\Delta i), \text{ q.e.d.} \end{aligned}$$

Δz ist die Anzahl der Zellen zwischen z_E und z_0, also

$$\Delta z = |\Delta z'|$$

Die Behauptung $\Delta z = \psi_Y \Delta z'$ ist wahr genau dann, wenn

$$\Delta z' < 0 \Leftrightarrow \psi_Y = -1 \wedge \Delta z' \geq 0 \Leftrightarrow \psi_Y = 1.$$

$\Delta z' = z_E - z_0$ ist negativ genau dann, wenn $z_E < z_0$ bzw. $\mathrm{Acc}_S(i_E) < \mathrm{Acc}_S(i_0)$. i_E wird zeitlich nach i_0 ausgeführt, also $\Theta_S(i_E) > \Theta_S(i_0)$. Da alle Zugriffe auf Y indexmonoton sind und $\Theta_S(i_E) > \Theta_S(i_0)$ und $\mathrm{Acc}_S(i_E) < \mathrm{Acc}_S(i_0)$ gilt, sind die Zugriffe *indexmonoton fallend*. Also ist $\psi_Y = -1$ laut Definition. Die Argumentation funktioniert in beide Richtungen, also ist die erste Äquivalenz gezeigt.

Der Beweis der zweiten Äquivalenz, $\Delta z' \geq 0 \Leftrightarrow \psi_Y = 1$, erfolgt analog. □

3.2 Lebenszeit

Die maximale Lebenszeit eines Arrays ist mit leicht veränderten Definitionen auch bei imperativen Parallelprogrammen problemlos ermittelbar. Die Abhängigkeitsanalyse muss jedoch die Abhängigkeiten zwischen den Indizes i zweier Anweisungsinstanzen erfassen, nicht zwischen den Arrayzellen z wie in Abschnitt 2.3 gefordert. Entsprechende Abhängigkeitsanalysemethoden sind nicht nur verfügbar, sondern gängig [2, 7].

Definition 3.4 *Seien in der Menge $D_{F,S,T}(i)$ alle Schleifenindizes i' der Anweisung T enthalten, die von der in Iteration i von Anweisung S geschriebenen Arrayzelle unter F abhängig sind. Die partielle Lebenszeit von Arrayzelle $Y[\mathrm{Acc}_S(i)]$ unter Abhängigkeit $F : S \to T$ ist gegeben durch:*

$$d_{F,S,T}(i) = \max_{i' \in D_{F,S,T}(i)} (\Theta_T i' - \Theta_S i)$$

Definition 3.5 *Sei $D_S \subseteq \mathbb{Z}^{c_S}$ wie oben beschrieben der Indexraum von S. Die maximale Lebenszeit aller Zellen eines Arrays unter Abhängigkeit $F : S \to T$ ist gegeben durch:*

$$d_{F,S,T} = \max_{i \in D_S}(d_{F,S,T}(i)))$$

Definition 3.6 *Sei F_Y die Menge aller Flow-Abhängigkeiten, die von einer Y schreibenden Anweisung ausgehen. Die* maximale Lebenszeit *aller Zellen des Arrays Y ist gegeben durch:*

$$d_Y = \max_{(F,S,T)\in F_Y} (d_{F,S,T})$$

Bei imperativen Parallelprogrammen sind neben den Flow- im Allgemeinen auch Anti- und Outputabhängigkeiten vorhanden. Die Definition der Indexmonotonie garantiert jedoch, dass keine Arrayzelle mehrmals beschrieben wird, somit können bei hier untersuchten Programmen von schreibenden Anweisungen nur Flow-Abhängigkeiten ausgehen.

Beispiel 3.2 *Es soll die maximale Lebenszeit des Arrays fib für das in Beispiel 3.1 gezeigte Programm mit den dort genannten Schedules errechnet werden.*

Es existieren fünf Flow-Abhängigkeiten:

$$\begin{array}{lrcll}
F_1: & \langle(),\ S\rangle & \rightarrow & \langle(2),\ U\rangle & \\
F_2: & \langle(),\ T\rangle & \rightarrow & \langle(2),\ U\rangle & \\
F_3: & \langle(),\ T\rangle & \rightarrow & \langle(3),\ U\rangle & \\
F_4: & \langle(i),\ U\rangle & \rightarrow & \langle(i+1),\ U\rangle & \text{für } 2 \leq i \leq n-1 \\
F_5: & \langle(i),\ U\rangle & \rightarrow & \langle(i+2),\ U\rangle & \text{für } 2 \leq i \leq n-2
\end{array}$$

Für jede Abhängigkeit wird die partielle Lebenszeit nach Definition 3.4 berechnet:

$$\begin{array}{rcrcl}
d_{F_1,S,U}() & = & \max_{()}((1)(2)+(-1)-()-(0)) & = & (1) \\
d_{F_2,T,U}() & = & \max_{()}((1)(2)+(-1)-()-(0)) & = & (1) \\
d_{F_3,T,U}() & = & \max_{()}((1)(3)+(-1)-()-(0)) & = & (2) \\
d_{F_4,U,U}(i) & = & \max_i((1)(i+1)+(-1)-(1)(i)-(-1)) & = & (1) \\
d_{F_5,U,U}(i) & = & \max_i((1)(i+2)+(-1)-(1)(i)-(-1)) & = & (2)
\end{array}$$

Da alle partiellen Lebenszeiten unabhängig von i sind, sind sie gleich den maximalen Lebenszeiten unter allen fünf Abhängigkeiten. Die endgültige maximale Lebenszeit ist nach Definition 3.6 das lexikographische Maximum der maximalen Lebenszeiten aller Abhängigkeiten:

$$d_{fib} = \max((1),(1),(2),(1),(2)) = (2)$$

An dieser Stelle kann nun gezeigt werden, wieso die Indexmonotonie in Abschnitt 3.1 nicht auf dem tatsächlichen Indexraum der Schleifen, sondern auf einer Erweiterung definiert wurde:

Sei das Beispielprogramm noch einmal wie in Beispiel 3.1 um das Statement `V: fib[n+4] = fib[n] + fib[n-1]` mit $\Theta_V = (n)$ erweitert, außerdem um `W: ret = fib[n]` mit $\Theta_W = (n+1)$. Die maximale Lebenszeit des Programms ändert sich nicht, jede Arrayzelle ist weiterhin nicht mehr als 2 Zeitschritte lebendig und somit ist, wie im vorherigen Kapitel gezeigt, auch die Nutzung von lediglich zwei Speicherzellen möglich. Das um V und W erweiterte, in seiner Speichernutzung optimierte Programm sähe dann wie folgt aus:

```
S:  fib[0] = 1;
T:  fib[1] = 1;
    for (i=2; i<=n; i++)
```

```
U:    fib[i%2] = fib[(i-1)%2] + fib[(i-2)%2];
V:  fib[(n+4)%2] = fib[n%2] + fib[(n-1)%2];
W:  ret = fib[n%2];
```

U schreibt im letzten Schleifenschritt zu Zeitpunkt $(n-1)$ in Speicherzelle $(n \mod 2)$. Einen Zeitschritt später schreibt V auf $(n+4) \mod 2 = n \mod 2$, also in die gleiche Speicherzelle. Der vorher gespeicherte Wert wird damit weniger als d_{fib} Zeitschritte gehalten, W liest dadurch fälschlicherweise den Wert von $fib[n+4]$! Grund dafür ist der unnötige Versatz der Arrayindizes: Stellt man sich das fib-Array und die Schreibzugriffe darauf wiederum wie eine Bandmaschine vor, läuft die Schreibposition während der Iterationen durch die i-Schleife gleichmäßig voran. Das Statement V führt jedoch einen „ruckartigen Sprung“ der Schreibposition durch. Dieser Sprung führt in der Speicherallokation wegen des Modulo-Operators zum vorzeitigen Überschreiben von Zellen. Würde V in $fib[n+1]$ schreiben, wäre die Verwendung des zweielementigen Arrays wieder korrekt; dann ist das Programm aber auch streng indexmonoton in seinen Schreibzugriffen. Die Definition der Indexmonotonie auf dem erweiterten Indexraum ist somit sinnvoll und schützt Werte in der Speicherallokation vor vorzeitigem Überschreiben.

3.3 Speicherallokation

Wie in Abschnitt 2.5 für funktionale Programme soll auch für imperative Parallelprogramme eine Speicherallokation Mem_Y gefunden werden, die zwei Arrayzellen $Y[z_1]$ und $Y[z_2]$ auf eine Speicherzelle abbildet genau dann, wenn für eine gegebene Allokationsmatrix $\mathcal{R} \in \mathbb{Z}^{n_Y \times m_Y}$ ein $a \in \mathbb{Z}^{m_Y}$ existiert, für das $(z_1 - z_2) = \mathcal{R}a$ gilt. Wiederum sind die Spalten von $\mathcal{R}$ die einzelnen, linear unabhängigen Allokationsvektoren ρ_i.
Definition 2.8 definiert eine Mem_Y-Funktion, die dies leistet. Da der Beweis von Lemma 2.9 ohne Änderungen auch im Imperativen gültig ist, kann die Definition der Speicherallokation unverändert genutzt werden.

Für die Erweiterung der weiteren im Abschnitt 2.5 gezeigten Lemmas für imperative Parallelprogramme wird in allen folgenden Beweisen die in Abschnitt 3.1 eingeführte Indexmonotonie benötigt und daher als gegeben angenommen. Für Speicherallokationen in Programmen, die nicht indexmonoton in ihren Schreibzugriffen auf ein Array sind, kann mit den folgenden Ausführungen keine Gültigkeit garantiert werden.

Definition 3.7 *Sei $\mathcal{N}_Y(z)$ die Menge aller Arrayzellen, die unter der Speicherallokation zeitlich nach $Y[z]$ in die gleiche Speicherzelle geschrieben werden.*

$$\mathcal{N}_Y(z) = \{z' \mid Mem_Y(z) = Mem_Y(z') \wedge \psi_Y z' > \psi_Y z\}$$

$z_0 := \psi_Y \min_{z' \in \mathcal{N}_Y(z)}(\psi_Y z')$. Dann wird $Y[z]$ von $Y[z_0]$ überschrieben.
Setze

$$\sigma_Y(z) := z_0 - z.$$

Lemma 3.8 *Für ein indexmonotones Array Y gilt für alle $z \in D_Y$: $\psi_Y \sigma_Y(z) > 0$.*

Beweis. Nach Definition ist für alle z: $\sigma_Y(z) = z_0 - z$ mit $\psi_Y z_0 > \psi_Y z$. Durch Umformungen erreicht man die Behauptung:

$$\begin{aligned} \psi_Y z_0 &> \psi_Y z \\ \psi_Y z_0 - \psi_Y z &> 0 \\ \psi_Y (z_0 - z) &> 0 \\ \psi_Y \sigma_Y(z) &> 0 \end{aligned}$$

□

Eine Speicherallokation ist auch im Imperativen gültig, wenn keine lebendige Arrayzelle überschrieben wird und kein gleichzeitiger Schreibzugriff auf eine Speicherzelle erfolgt, doch Korollar 2.11 gilt nur für Alpha-Programme. Mit ausschließlich indexmonotonen Schreibzugriffen gibt es jedoch auch im Imperativen eine Gültigkeitsbedingung. Dazu müssen jedoch erst einige Definitionen gemacht werden.

Definition 3.9 *Sei $\mathcal{S}_Y^{\neq 0}$ die Menge aller auf Y schreibenden Anweisungen, die nicht zu einem einzigen konstanten Zeitpunkt ausgeführt werden. Formal ausgeschrieben:*

$$\mathcal{S}_Y^{\neq 0} = \{S \mid S \in \mathcal{S}_Y \wedge \tau_S \neq 0\}$$

$\mathcal{Z}_Y^{\neq 0}$ ist die Menge aller Arrayzellen von Y, die die nicht zu konstanter Zeit ausgeführten Statements schreiben:

$$\mathcal{Z}_Y^{\neq 0} = \bigcup_{S \in \mathcal{S}_Y^{\neq 0}} \mathrm{Acc}_S(D_S)$$

Entsprechend ist $\mathcal{Z}_Y$ die Menge aller Arrayzellen von Y, die von irgendeinem Statement beschrieben werden:

$$\mathcal{Z}_Y = \bigcup_{S \in \mathcal{S}_Y} \mathrm{Acc}_S(D_S)$$

Das kleinste Element von $\mathcal{Z}_Y$ bzw. $\mathcal{Z}_Y^{\neq 0}$ wird mit κ_Y bzw. $\kappa_Y^{\neq 0}$ bezeichnet:

$$\kappa_Y = \min\{\psi_Y z \mid z \in \mathcal{Z}_Y\},\ \kappa_Y^{\neq 0} = \min\left\{\psi_Y z \mid z \in \mathcal{Z}_Y^{\neq 0}\right\}$$

Da $\mathcal{Z}_Y^{\neq 0} \subseteq \mathcal{Z}_Y$, gilt $\kappa_Y \leq \kappa_Y^{\neq 0}$ und somit auch $\kappa_Y^{\neq 0} - \kappa_Y \geq 0$. Die beiden minimalen Elemente existieren bei indexmonoton zugreifenden Programmen und sind diejenigen Arrayzellen, die im ersten auf Y schreibenden Zeitpunkt beschrieben werden.

Definition 3.10 *Die Menge $\mathcal{I}_S(\Delta t)$ ist gegeben durch*

$$\mathcal{I}_S(\Delta t) = \{\Delta i \mid \Delta i \in \mathbb{Z}^{c_S} \wedge \tau_S \Delta i \geq \Delta t\}.$$

Für ein Statement $S \in \mathcal{S}_Y^{\neq 0}$ ist die Menge $\psi_Y \mathcal{A}_S(\mathcal{I}_S(\Delta t))$ gegeben durch

$$\psi_Y \mathcal{A}_S(\mathcal{I}_S(\Delta t)) = \{\psi_Y \mathcal{A}_S \Delta i \mid \Delta i \in \mathcal{I}_S(\Delta t)\}$$

Ein beliebiges Element dieser Menge sei mit $\Delta \check{z}_S(\Delta t)$ bezeichnet:

$$\Delta \check{z}_S(\Delta t) \in \psi_Y \mathcal{A}_S(\mathcal{I}_S(\Delta t))$$

Mit den in Lemma 3.3 gezeigten Beziehungen erkennt man: $\mathcal{I}_S(\Delta t)$ ist die Menge der Indexschritte, die ein Statement $S \in \mathcal{S}_Y$ nach $\Delta t \in \mathbb{Z}^k$ oder mehr Zeiteinheiten ausführen kann. $\mathcal{A}_S(\mathcal{I}_S(\Delta t))$ sind dann die Indizes der Arrayzellen, die ein Statement S zum oder nach Zeitpunkt $t_0 + \Delta t$ relativ zu den zu Zeit t_0 geschriebenen Zellenindizes schreibt. Das bedeutet aufgrund der Indexmonotonie, dass der Absolutbetrag (hier gemäß Lemma 3.3 durch ψ_Y ausgedrückt) *jeden* Elements der Menge, also insbesondere auch $\Delta\check{z}_S(\Delta t)$, eine obere Schranke für die Anzahl der Arrayindizes ist, um die die Schreibmarke auf Array Y in *weniger* als Δt Zeitschritten verschoben wird.

Offensichtlich gilt für $\Delta t > 0 : S \notin \mathcal{S}_Y^{\neq 0} \Leftrightarrow \mathcal{I}_S(\Delta t) = \emptyset$: Ein zu konstanter Zeit ausgeführtes Statement kann nicht nach einer positiven Anzahl von Zeitschritten weiterhin ausgeführt werden. In allen anderen Fällen ist die Menge absolut nach unten beschränkt. Also ist aufgrund der Indexmonotonie und der Linearität von $\mathcal{A}_S$ auch $\psi_Y \mathcal{A}_S(\mathcal{I}_S(\Delta t))$ nach unten beschränkt und alle enthaltenen Elemente positiv.

Mit Hilfe dieser Definitionen lässt sich ein Gültigkeitsbegriff für eine Speicherallokation formal definieren und seine Korrektheit beweisen, der auf folgender Grundidee basiert:

> Aufgrund der Indexmonotonie wissen wir, dass sich die Schreibmarke auf dem Array gleichmäßig verschiebt: Es lässt sich errechnen, um wie viele Zellen sie sich nach einer bestimmten Zeit verschoben hat ($\Delta\check{z}_S(\Delta t)$); also lässt sich auch errechnen, um wie viele Zellen die Schreibposition in der maximalen Lebenszeit verschoben wird ($\Delta\check{z}_S(d_Y)$). In *weniger* als dieser Zeit werden nur die Arrayzellen beschrieben, die zwischen der Ausgangsposition der Schreibmarke und ihrer Position nach Ablauf der Zeit liegen – Voraussetzung für die Gültigkeit einer Speicherallokation ist also, dass für jede Arrayzelle gilt, dass sie im Speicher frühestens von einer Arrayzelle überschrieben wird, bei der die Schreibmarke erst nach Ablauf der maximalen Lebenszeit ankommt.

Lemma 3.11 *Eine Speicherallokation Mem_Y ist gültig, wenn keine lebendige Arrayzelle überschrieben wird und kein gleichzeitiger Schreibzugriff auf eine Speicherzelle erfolgt. Das ist gegeben, wenn für ein streng indexmonotones Array Y, $d_Y > 0$, $\mathcal{S}_Y^{\neq 0} \neq \emptyset$ und alle $z \in D_Y$ gilt:*

$$\left(\kappa_Y^{\neq 0} - \kappa_Y\right) + \max_{S \in \mathcal{S}_Y^{\neq 0}} \left(\Delta\check{z}_S(d_Y)\right) \leq \psi_Y \sigma_Y(z)$$

Im nun folgenden Beweis wird auf viele zuvor definierte Variablen zurückgegriffen. Der besseren Übersicht wegen sind sie daher mit kurzer Beschreibung in Tabelle 3.1 noch einmal aufgeführt. Alle im Beweis lokal definierten Variablen werden nach folgender Konvention benannt:

> Zeitpunkte werden mit t bezeichnet;
> Arrayzellenindizes mit z bezeichnet;
> Zeitspannen werden mit Δt bezeichnet;
> Arrayzellenindexdifferenzen werden mit Δz bezeichnet.

Name	Typ und Beschreibung	Definition
d_Y	maximale Lebenszeit bzw. Zeitspanne, in der eine Arrayzelle im Speicher verbleiben muss, um nicht vorzeitig überschrieben zu werden	3.6
$\psi_Y \sigma_Y(z)$	Arrayzellenindexdifferenz zwischen Zelle z und der in der Speicherallokation z überschreibenden Zelle	3.7
$\Delta\check{z}_S(\Delta t)$	Arrayzellenindexdifferenz, um die die Schreibmarke von Statement S in Δt Zeitschritten maximal verschoben wird	3.10
$\kappa_Y^{\neq 0}$	kleinster Arrayzellenindex, der von allen mehrfach ausgeführten Statements beschrieben wird	3.9
κ_Y	kleinster Arrayzellenindex, der beschrieben wird	3.9

Tabelle 3.1: Im Beweis zu Lemma 3.11 genutzte Variablen

$\Delta z(\Delta t)$ gibt in Abhängigkeit der Zeitspanne Δt an, um wieviele Arrayzellen die Schreibmarke verschoben wurde, ist also der Konvention folgend eine Arrayzellenindexdifferenz.

Beweis. Es gilt wie oben ausgeführt $\kappa_Y^{\neq 0} - \kappa_Y \geq 0$ sowie $\max_{S \in \mathcal{S}_Y^{\neq 0}} (\Delta\check{z}_S(d_Y)) \geq 0$. Wenn $a + b \leq c \wedge a \geq 0 \wedge b \geq 0$ gilt, gilt $a \leq c$ sowie $b \leq c$. Somit folgt aus der Voraussetzung:

$$\forall z \in D_Y : \kappa_Y^{\neq 0} - \kappa_Y \leq \psi_Y \sigma_Y(z) \tag{3.9}$$

$$\forall z \in D_Y : \max_{S \in \mathcal{S}_Y^{\neq 0}} (\Delta\check{z}_S(d_Y)) \leq \psi_Y \sigma_Y(z) \tag{3.10}$$

Man nehme an, die Speicherallokation für Y sei ungültig. Fallunterscheidung:

1. **Eine lebendige Speicherzelle wird überschrieben.**

   Es existiert zu Zeitpunkt t_0 ein $z_0 \in D_Y$, das vor seiner letzten Verwendung, also in $\Delta t := t - t_0 < d_Y$ Zeiteinheiten, von $z_0' := z_0 + \sigma_Y(z_0)$ überschrieben wird. Durch die Indexmonotonie müssen in der Zwischenzeit die Arrayzellen zwischen z_0 und $(z_0 + \sigma_Y(z_0))$ geschrieben oder übersprungen worden sein, die Schreibposition muss in weniger als d_Y Zeiteinheiten um mindestens $\psi_Y \sigma_Y(z_0)$ Arrayzellen verschoben worden sein:

   $$\psi_Y \sigma_Y(z_0) \leq \Delta z(\Delta t). \tag{3.11}$$

   Fallunterscheidung:

   (a) **Mindestens ein nicht zu konstanter Zeit ausgeführtes Statement $T \in \mathcal{S}_Y^{\neq 0}$ hat schon einmal in das Array geschrieben.**
   Da $\Delta t < d_Y$ gilt, gilt aufgrund der Indexmonotonie

   $$\Delta z(\Delta t) < \Delta z(d_Y). \tag{3.12}$$

Annahme: T wird nach d_Y oder mehr Zeiteinheiten noch einmal ausgeführt, dann schreibt T unter anderem in $Y[z_0 + \Delta\check{z}_T(d_Y)]$. Die Schreibmarke wurde also um maximal $\Delta\check{z}_T(d_Y)$ Zellen verschoben:

$$\Delta z(d_Y) \leq \Delta\check{z}_T(d_Y)$$

Nach Definition der Maximum-Funktion gilt dann auch

$$\Delta z(d_Y) \leq \max_{S \in \mathcal{S}_Y^{\neq 0}} \left(\Delta\check{z}_S(d_Y)\right). \tag{3.13}$$

Mit (3.12) und (3.10) erhält man

$$\Delta z(\Delta t) < \max_{S \in \mathcal{S}_Y^{\neq 0}} \left(\Delta\check{z}_S(d_Y)\right) \leq \psi_Y \sigma_Y(z_0).$$

Widerspruch zu (3.11).

Es wurde angenommen, dass T nach d_Y oder mehr Zeiteinheiten noch einmal ausgeführt wird. Dies ist jedoch nicht immer gegeben, da T einen begrenzten Indexraum D_T besitzt – irgendwann wird die letzte Instanz von T ausgeführt. Die Indexmonotonie wurde in Abschnitt 3.1 allerdings so definiert, dass sie auch für Indexschritte gilt, die nicht in D_T existieren und somit nicht ausgeführt werden, die aber – wenn sie denn existierten – zeitlich später ausgeführt würden (D_T^+, Definition 3.1). Auch wenn T also in Wirklichkeit nicht noch einmal ausgeführt wird, kann man dies trotzdem annehmen und alle mit seiner nochmaligen Ausführung verbundenen Garantien gelten trotzdem.

(b) **Es wurden bisher nur Statements zu konstanter Zeit ausgeführt.**

Da nach Voraussetzung $\mathcal{S}_Y^{\neq 0} \neq \emptyset$, gibt es mindestens ein mehrfach ausgeführtes Statement $U \in \mathcal{S}_Y^{\neq 0}$, das $\kappa_Y^{\neq 0}$ zu Zeit t_U schreibt. Dies kann innerhalb von d_Y Zeitschritten oder erst später passieren, man kann aber wie in Fall 1a beschrieben aufgrund der Definition der Indexmonotonie auf jeden Fall annehmen, dass U nach d_Y Zeitschritten ausgeführt wird. Aufgrund der Indexmonotonie wird U von allen wiederholt ausgeführten Statements als erstes ausgeführt.

z_0 kann nicht kleiner als die kleinste überhaupt geschrieben Arrayzelle κ_Y sein. Daher können bis zum Zeitpunkt t_U, an dem U seinen ersten Schreibvorgang in die Zelle $\kappa_Y^{\neq 0}$ ausführt, maximal $\kappa_Y^{\neq 0} - \kappa_Y$ Arrayzellen beschrieben oder übersprungen worden sein: $\Delta z(t_U - t_0) = \kappa_Y^{\neq 0} - \kappa_Y$.

Fallunterscheidung mit t Zeitpunkt, an dem z_0 überschrieben wird:

i. $t_U > t$: **Es wird kein wiederholt ausgeführtes Statement ausgeführt, bevor z_0 überschrieben wird.** Das heißt mit anderen Worten: z_0 wird von einem zu konstanter Zeit ausgeführten Statement überschrieben.

Dann gilt aufgrund der Indexmonotonie und mit $\Delta t = t - t_0$:

$$\Delta z(\Delta t) < \Delta z(t_U - t_0) = \kappa_Y^{\neq 0} - \kappa_Y$$

Mit (3.9) erhält man daraus $\Delta z(\Delta t) < \psi_Y \sigma_Y(z_0)$, Widerspruch zu (3.11).

ii. $t_0 \leq t_U \leq t$: z_0 **wird erst überschrieben, nachdem bereits ein wiederholt ausgeführtes Statement ausgeführt wurde.** Das bedeutet, dass z_0 von einem konstant ausgeführten oder von einem wiederholt ausgeführten Statement überschrieben werden kann.
Da $\Delta t = t - t_0 < d_Y$ ist, ist $t < d_Y + t_0$. An die Voraussetzung angehängt ergibt sich

$$t_0 \leq t_U \leq t < d_Y + t_0.$$

Durch Ausnutzen der ersten Beziehung $t_0 \leq t_U$ und Einsetzen in den letzten Term erhält man

$$t_0 \leq t_U \leq t < d_Y + t_U.$$

Von den letzten beiden Termen wird nun t_0 abgezogen:

$$\begin{aligned} t - t_0 &< t_U - t_0 + d_Y \\ \Delta t &< (t_U - t_0) + d_Y \end{aligned}$$

Wie im ersten Fall erhält man durch Ausnutzen der Indexmonotonie

$$\Delta z(\Delta t) < \Delta z(t_U - t_0 + d_Y).$$

Da $d_Y > 0$ ist, ist

$$\Delta z(t_U - t_0 + d_Y) = \Delta z(t_U - t_0) + \Delta z(d_Y)$$

$\Delta z(t_U - t_0)$ ist in Fall 1(b)i bereits bestimmt worden. Die Verschiebung der Schreibmarke $\Delta z(d_Y)$, die ein beliebiges $T \in \mathcal{S}_Y^{\neq 0}$ in d_Y Zeitschritten ausführt, ist in Fall 1a, Gleichung (3.13) schon bestimmt worden. Damit ergibt sich:

$$\begin{aligned} \Delta z(\Delta t) \quad < \quad & \Delta z(t_U - t_0 + d_Y) \leq \\ & \kappa_Y^{\neq 0} - \kappa_Y + \max_{S \in \mathcal{S}_Y^{\neq 0}} (\Delta \check{z}_S(d_Y)) \leq \\ & \psi_Y \sigma_Y(z_0) \end{aligned}$$

Auch hier ergibt sich ein Widerspruch zu (3.11).

2. **Es existiert ein gleichzeitiger Schreibzugriff auf eine Speicherzelle von Y.**

Dann gibt es ein $z_0 \in D_Y$ mit Schreibzugriff zur Zeit t und $z' = (z_0 + \sigma_Y(z_0))$ mit Schreibzugriff zur gleichen Zeit. Innerhalb eines Zeitschritts, $\Delta t = 0$, muss die Schreibmarke dann um

$$\Delta z(\Delta t) \geq \psi_Y \sigma_Y(z_0) \tag{3.14}$$

Arrayzellen verschoben werden, da wegen der Indexmonotonie z' ansonsten erst später und nicht zur gleichen Zeit wie z_0 geschrieben würde. Da $d_Y > 0$, ist $\Delta t < d_Y$. In Fall 1 wurde für alle z und $\Delta t < d_Y$ bewiesen, dass $\Delta z(\Delta t) < \psi_Y \sigma_Y(z)$ gilt, also gilt insbesondere

$$\Delta z(\Delta t) < \psi_Y \sigma_Y(z_0).$$

Widerspruch zu (3.14).

Alle Fälle zu einem Widerspruch geführt. □

Lemma 3.11 zeigt, dass die Anzahl der Arrayzellen, die in weniger als d_Y Zeitschritten beschrieben oder übersprungen werden können, nach oben beschränkt ist. Die im Betrag kleinste Schranke sei mit $\widehat{z}_Y(d_Y)$ bezeichnet.

Definition 3.12 *$\widehat{z}_Y(\Delta t)$, eine Anzahl an Arrayzellen, die alle Statements $S \in \mathcal{S}_Y$ in $0 < \Delta t \in \mathbb{Z}^k$ Zeiteinheiten* nicht *schreiben oder überspringen können, wenn $\mathcal{S}_Y^{\neq 0} \neq \emptyset$ gilt, ist gegeben durch*

$$\widehat{z}_Y(\Delta t) = \left(\kappa_Y^{\neq 0} - \kappa_Y\right) + \max_{S \in \mathcal{S}_Y^{\neq 0}} \left(\Delta \check{z}_S(\Delta t)\right).$$

Für alle $S \in \mathcal{S}_Y^{\neq 0}$ ist $\Delta \check{z}_S(\Delta t)$ gemäß Definition 3.10 ein beliebiges Element der Menge $\psi_Y \mathcal{A}_S(\mathcal{I}_S(\Delta t))$.

Um nach der im nächsten Abschnitt definierten Auswahl von Allokationsvektoren möglichst kleine Allokationsvektoren und damit eine möglichst hohe Optimierung der Speichernutzung zu erhalten, sollten alle $\Delta \check{z}_S(\Delta t)$ so gewählt werden, dass $\widehat{z}_Y(\Delta t)$ im Betrag möglichst klein ist. Diese Forderung ist für die Korrektheit natürlich nicht nötig, eine Pseudoprojektion in eine „zu große“ Speicherallokation führt schließlich nur zu einer längeren Lebenszeit jeder einzelnen Arrayzelle im Speicher.

Ist $\mathcal{S}_Y^{\neq 0} = \emptyset$, gibt es im Programm ausschließlich zu einmaliger, konstanter Zeit ausgeführte Anweisungen. Lemma 3.11 gibt für diesen Fall keine Angaben über die Gültigkeit einer Speicherallokation.

3.4 Wahl der Allokationsvektoren

In diesem Abschnitt wird $\mathcal{S}_Y^{\neq 0} \neq \emptyset$ vorausgesetzt. Dann kann eine Bedingung aufgestellt werden, unter der ein Satz Allokationsvektoren $\rho_1, \ldots, \rho_{m_Y}$ gültig ist. Wie in Kapitel 2 bestimmen die Allokationsvektoren alle Arrayzellen, die in die selbe Speicherzelle geschrieben werden.

Lemma 3.13 *Eine Menge Allokationsvektoren ρ_i bestimmt eine nach Lemma 3.11 gültige Speicherallokation für ein indexmonotones Array Y und eine maximale Lebenszeit $d_Y > 0$, wenn für alle ganzzahligen Linearkombinationen η der Vektoren gilt:*

$$\eta > 0 \Rightarrow \widehat{z}_Y(d_Y) \leq \eta$$

Beweis. Nach Definition bestimmen die Allokationsvektoren alle Arrayzellen, die in die selbe Speicherzelle geschrieben werden. Somit ist $\sigma_Y(z)$ für beliebiges z eine ganzzahlige Linearkombination der Allokationsvektoren. Also ist auch $\psi_Y \sigma_Y(z)$ eine ganzzahlige Linearkombination:

$$\psi_Y \sigma_Y(z) =: \eta_\sigma.$$

Nach Lemma 3.8 gilt $\forall Y,\ \forall z : \psi_Y \sigma_Y(z) > 0$, somit auch $\eta_\sigma > 0$.

Die Voraussetzung

$$\eta > 0 \Rightarrow \widehat{z}_Y(d_Y) \leq \eta$$

gilt für alle η, also insbesondere auch für $\eta_\sigma > 0$. Daraus folgt

$$\begin{aligned} \widehat{z}_Y(d_Y) &\leq \eta_\sigma \\ \widehat{z}_Y(d_Y) &\leq \psi_Y \sigma_Y(z) \end{aligned}$$

Durch Einsetzen der Definition von $\widehat{z}_Y(d_Y)$ erhält man die in Lemma 3.11 geforderte Gültigkeitsbedingung der Allokationsvektoren. □

Nun kann eine Möglichkeit vorgestellt werden, korrekte Allokationsvektoren für ein gegebenes imperatives Parallelprogramm zu finden.

Lemma 3.14 *Sei m_Y Grad von $\widehat{z}_Y(d_Y)$ gemäß Definition 2.13, Seite 14. Sei $d_Y > 0$, seien $e_i := \left(\underbrace{0,\ldots,0}_{(i-1)},1,\underbrace{0,\ldots,0}_{(n_Y-i)}\right)$ kanonische Basisvektoren von $\mathbb{Z}^{n_Y}$. Dann lassen sich m_Y Vektoren ρ_i finden, für die gilt:*

$$\forall 1 \leq i < m_Y : \rho_i := e_i,\ \rho_{m_Y} := \widehat{z}_Y(d_Y)$$

So gefundene ρ_i sind gültige Allokationsvektoren.

Beweis. Die Allokationsvektoren ρ_i sind *nicht* gültig, wenn es eine Linearkombination η von ihnen gibt mit

$$\widehat{z}_Y(d_Y) > \eta > 0. \tag{3.15}$$

Mit $\mu_i \in \mathbb{Z}$ und den Definitionen von η und ρ_i gilt:

$$\begin{aligned} \eta &= \sum_{i=1}^{m_Y} \mu_i \rho_i = \sum_{i=1}^{m_Y-1} \mu_i \rho_i + \mu_{m_Y} \rho_{m_Y} = \sum_{i=1}^{m_Y-1} \mu_i e_i + \mu_{m_Y} \widehat{z}_Y(d_Y) \\ &= \left(\mu_1,\ldots,\mu_{m_Y-1},\ \underbrace{0,\ldots,0}_{n_Y-m_Y+1}\right)^T + \mu_{m_Y} \widehat{z}_Y(d_Y) \end{aligned}$$

Eingesetzt in (3.15) erhält man die Forderung

$$\widehat{z}_Y(d_Y) > \left(\mu_1,\ldots,\mu_{m_Y-1},\ \underbrace{0,\ldots,0}_{n_Y-m_Y+1}\right)^T + \mu_{m_Y} \widehat{z}_Y(d_Y) > 0.$$

Da m_Y der Grad von $\widehat{z}_Y(d_Y)$ ist, kann die Ungleichung nur für $\mu_1 = \mu_2 = \cdots = \mu_{m_Y-1} = 0$ erfüllt sein:

$$\widehat{z}_Y(d_Y) > \mu_{m_Y} \widehat{z}_Y(d_Y) > 0.$$

Diese Ungleichung ist nicht erfüllbar mit $\mu_{m_Y} \in \mathbb{Z}$, Widerspruch. □

Lemma 3.15 *Die Anzahl der gültigen, linear unabhängigen Allokationsvektoren für ein Array Y ist nach oben beschränkt durch den Grad von $\widehat{z}_Y(d_Y)$.*

Beweis. Der Beweis erfolgt analog zum Beweis zu Lemma 2.14 auf Seite 14 mit $M := (\rho_1, \ldots, \rho_{m+1})$. Zu finden ist ein μ_0 mit $0 < M\mu_0 < \widehat{z}_Y(d_Y)$. □

Die nach Lemma 3.14 gewählten Allokationsvektoren sind also optimal in dem Sinne, dass keine größere Anzahl Vektoren unter der in Lemma 3.13 definierten Gültigkeitsbedingung gefunden werden kann. Die Gültigkeitsbedingung ist jedoch konservativ, daher kann es bessere Auswahlen geben (siehe dazu auch Abschnitt 3.6 und 3.7).

3.5 Anwendung

Nach den Erklärungen und Beweisen hier eine Übersicht über die Schritte, die bei imperativen Parallelprogrammen zur Optimierung der Speichernutzung eines n_Y-dimensionalen Arrays Y durchgeführt werden müssen.

1. Überprüfung, ob schreibende Arrayzugriffe streng indexmonoton.

   Dafür nötig:

$\mathcal{S}_Y$	Menge der Statements, die auf Y schreiben;
alle $D_S \subseteq \mathbb{Z}^{c_S}$	Indexraum der S umgebenden Schleifen;
alle $\Theta_S(i)$	Zeit, zu der Index i von Statement S ausgeführt wird;
alle $\mathrm{Acc}_S(i)$	Arrayzelle, auf die Statement S mit Index i schreibt.

   Sei $D_S^+ = \{i \mid i \in \mathbb{Z}^{c_S} \wedge (\exists i_0 \in D_S : \Theta_S i_0 \leq \Theta_S i)\}$. Zu überprüfen ist für alle möglichen Paare $S \in \mathcal{S}_Y,\ T \in \mathcal{S}_Y$, ob

   $$\forall i_S \in D_S^+, i_T \in D_T^+, S, T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \Rightarrow \mathrm{Acc}_S(i_S) < \mathrm{Acc}_T(i_T)$$

   oder

   $$\forall i_S \in D_S^+, i_T \in D_T^+, S, T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \Rightarrow \mathrm{Acc}_S(i_S) > \mathrm{Acc}_T(i_T)$$

   gilt. Im ersten Fall setze $\psi_Y := 1$, im zweiten $\psi_Y := -1$.

   Sollte keine der beiden Bedingungen gelten, kann die Speichernutzung nicht optimiert werden, Abbruch.

2. Bestimmung der *maximalen Lebenszeit* d_Y aller Arrayzellen.

   Dafür nötig:

F_Y	Menge aller Flow-Abhängigkeiten, die von einer Y schreibenden Anweisung ausgehen;
$D_S \subseteq \mathbb{Z}^{c_S}$	Indexraum der S umgebenden Schleifen;
$D_{F,S,T}(i)$	Schleifenindizes einer Anweisung T, die unter $F : S \to T \in F_Y$ von Index i von S abhängen;
$\Theta_S(i)$	Zeitpunkt, an dem Statement S mit Index i ausgeführt wird.

   Berechnung der Lebenszeit einer Arrayzelle $Y[\mathrm{Acc}_S(i)]$ unter $F : S \to T$:

   $$d_{F,S,T}(i) = \max_{i' \in D_{F,S,T}(i)} (\Theta_T i' - \Theta_S i)$$

Bestimmung der Lebenszeit aller Zellen des Arrays Y unter Abhängigkeit $F : S \to T$:

$$d_{F,S,T} = \max_{i \in D_S}(d_{F,S,T}(i)))$$

Bestimmung der maximalen Lebenszeit von Y unter allen Abhängigkeiten:

$$d_Y = \max_{(F,S,T) \in F_Y}(d_{F,S,T})$$

3. Bestimmung von $\widehat{z}_Y(d_Y)$, der minimalen Anzahl an Arrayzellen, die nicht in d_Y Zeitschritten geschrieben werden können.

   Dafür nötig:

$\mathcal{S}_Y$	Menge der Statements, die auf Y schreiben;
alle $D_S \subseteq \mathbb{Z}^{c_S}$	Indexraum der S umgebenden Schleifen;
alle $\tau_S(i)$	Nichtkonstante Komponente von $\Theta_S(i)$;
alle $\mathrm{Acc}_S(i)$	Arrayzelle, auf die Statement S mit Index i schreibt;
alle $\mathcal{A}_S(i)$	Nichtkonstante Komponente von $\mathrm{Acc}_S(i)$;
ψ_Y	Indexmonotoniekonstante aus Schritt 1;
$0 < d_Y \in \mathbb{Z}^k$	maximale Lebenszeit aus Schritt 2.

   $\mathcal{S}_Y^{\neq 0} = \{S \mid S \in \mathcal{S}_Y \wedge \tau_S \neq 0\}$ ist die Menge aller auf Y schreibenden Anweisungen, die nicht zu einem einzigen konstanten Zeitpunkt ausgeführt werden. Falls $\mathcal{S}_Y^{\neq 0} = \emptyset$, ist $\widehat{z}_Y(d_Y)$ nicht definiert: Abbruch.

   $\mathcal{Z}_Y^{\neq 0} = \bigcup_{S \in \mathcal{S}_Y^{\neq 0}} \mathrm{Acc}_S(D_S)$ ist die Menge der Arrayzellen von Y, die die nicht zu konstanter Zeit ausgeführten Statements schreiben.

   $\mathcal{Z}_Y = \bigcup_{S \in \mathcal{S}_Y} \mathrm{Acc}_S(D_S)$ ist die Menge der Arrayzellen von Y, die alle Statements schreiben.

   $\mathcal{I}_S(\Delta t) = \{i \mid i \in \mathbb{Z}^{c_S} \wedge \tau_S i \geq \Delta t\}$ ist die Menge der Indexschritte, die ein Statement $S \in \mathcal{S}_Y$ nach $\Delta t \in \mathbb{Z}^k$ Zeiteinheiten ausführen kann, $\Delta\check{z}_S(\Delta t)$ ist ein Element der Menge $\psi_Y \mathcal{A}_S(\mathcal{I}_S(\Delta t))$.

   Bestimme $\kappa_Y := \min\{\psi_Y z \mid z \in \mathcal{Z}_Y\}$, $\kappa_Y^{\neq 0} := \min\left\{\psi_Y z \mid z \in \mathcal{Z}_Y^{\neq 0}\right\}$.

   Damit kann auch $\widehat{z}_Y(d_Y)$ bestimmt werden:

   $$\widehat{z}_Y(d_Y) = \left(\kappa_Y^{\neq 0} - \kappa_Y\right) + \max_{S \in \mathcal{S}_Y^{\neq 0}}\left(\Delta\check{z}_S(d_Y)\right).$$

   Für alle $S \in \mathcal{S}_Y^{\neq 0}$ ist $\Delta\check{z}_S(d_Y)$ so zu wählen, dass $|\widehat{z}_Y(d_Y)|$ möglichst klein ist.

4. Bestimmung der Allokationsvektormatrix $\mathcal{R}$.

   Dafür nötig:

$\widehat{z}_Y(d_Y)$	aus Schritt 3.

   Sei $1 \leq m_Y \leq n_Y$ die Position der ersten Stelle in d_Y, die nicht 0 ist,
   sei $\forall 1 \leq i \leq n_Y : e_i$ kanonischer Basisvektor von $\mathbb{Z}^{n_Y}$, so dass $e_i > e_{i+1}$.

   Die Allokationsvektoren ρ_i sowie die Allokationsvektormatrix $\mathcal{R}$ sind dann wie folgt definiert:

   $$\begin{aligned} \rho_i &= e_i \text{ für } 1 \leq i \leq m_Y - 1 \\ \rho_{m_Y} &= \widehat{z}_Y(d_Y) \\ \mathcal{R} &= \begin{pmatrix} \rho_1 & \rho_2 & \dots & \rho_{m_Y} \end{pmatrix} \end{aligned}$$

5. Finde eine Matrix Π_Y, so dass gilt: $\forall 1 \leq i \leq m_Y : \Pi_Y \rho_i = 0$.

6. Berechnen der Matrizen Ω_Y und S_Y.

   Sei S $(m_Y \times m_Y)$-Diagonalmatrix mit $\forall 1 \leq i \leq m_Y : s_i \geq 0$.

   Finde ein $U \in \mathbb{Z}^{n_Y \times n_Y}$, $V \in \mathbb{Z}^{m_Y \times m_Y}$ und $S' \in \mathbb{Z}^{n_Y \times m_Y}$, so dass $\mathcal{R} = US'V$ und $S' = \begin{pmatrix} S \\ 0 \end{pmatrix}$.

   Ferner sei U^{-1} Inverses von U. Dann ist Ω_Y die Untermatrix von U^{-1}, die die ersten m_Y Zeilen bildet, $S_Y := (s_1, s_2, \ldots, s_{m_Y})^T$.

7. Konstruktion der Mem-Funktion mit

$$\mathrm{Mem}_Y(z) = \begin{pmatrix} \Pi_Y z \\ (\Omega_Y z) \mod S_Y \end{pmatrix}$$

   Die mod-Funktion ist zeilenweise anzuwenden.

3.6 Beispiele

Die im vorherigen Abschnitt gezeigte Vorgehensweise wird hier zum besseren Verständnis an zwei Beispielen demonstriert. Es wurden die gleichen Beispiele wie in Kapitel 2 gewählt, um eine leichte Vergleichbarkeit des funktionalen und des imperativen Verfahrens zu bieten.

3.6.1 Fibonacci-Zahlen

Ein imperatives Programm zur Berechnung der Fibonacci-Zahlen wurde bereits in Beispiel 3.1 mit gültigen Schedules vorgestellt:

```
S: fib[0] = 1;
T: fib[1] = 1;
   for (i=2; i<=n; i++)
U:   fib[i] = fib[i-1] + fib[i-2];
```

Die vorhandenen von fib schreibenden Anweisungen ausgehenden Flow-Abhängigkeiten wurden in Beispiel 3.2 aufgezählt. Somit sind alle für das Verfahren nötigen Informationen vorhanden.

1. Überprüfung, ob schreibende Arrayzugriffe streng indexmonoton.

   Die Überprüfung wurde bereits in Beispiel 3.1, Seite 22 durchgeführt. Alle Schreibzugriffe auf fib sind streng indexmonoton steigend.

$$\psi_{fib} = 1$$

2. Bestimmung der maximalen Lebenszeit d_{fib} aller Arrayzellen.

   Die maximale Lebenszeit wurde bereits in Beispiel 3.2, Seite 26 berechnet:

$$d_{fib} = (2)$$

3. Bestimmung von $\widehat{z}_{fib}(d_{fib})$, der minimalen Anzahl an Arrayzellen, die nicht in d_{fib} Zeitschritten geschrieben werden können.

   Nur Statement U wird nicht einmalig ausgeführt, daher $\mathcal{S}_{fib}^{\neq 0} = \{U\}$. U schreibt in $fib[z]$ mit $2 \leq z \leq n$, also $\mathcal{Z}_{fib}^{\neq 0} = \{(z) \mid 2 \leq z \leq n\}$. Von allen Statements gemeinsam werden im Array alle Zellen zwischen 0 und n beschrieben: $\mathcal{Z}_{fib} = \{(z) \mid 0 \leq z \leq n\}$. Also ist $\kappa_{fib} = (0)$, $\kappa_{fib}^{\neq 0} = (2)$.

   $\mathcal{I}_S(\Delta t) = \{i \mid i \in \mathbb{Z}^{c_S} \wedge \tau_S i \geq \Delta t\}$ ist die Menge der Indexschritte, die ein Statement $S \in \mathcal{S}_Y$ nach $\Delta t > 0$ und mehr Zeiteinheiten ausführen kann. Für die drei vorhandenen Statements sind das für $\Delta t = d_{fib} = (2)$:

$$\begin{aligned} \mathcal{I}_S(d_{fib}) &= \emptyset \\ \mathcal{I}_T(d_{fib}) &= \emptyset \\ \mathcal{I}_U(d_{fib}) &= \{2, 3, 4, \dots\} \end{aligned}$$

   Es ist zu beachten, dass $\mathcal{I}_U(d_{fib})$ nicht nach oben durch n beschränkt ist.

$$\Delta\check{z}_U(d_{fib}) \in \psi_{fib}\mathcal{A}_U(\{2, 3, 4, \dots\}) = \{2, 3, 4, \dots\}$$

   Somit ergibt sich für $\widehat{z}_{fib}(d_{fib})$:

$$\begin{aligned} \widehat{z}_{fib}(d_{fib}) &= \left(\kappa_{fib}^{\neq 0} - \kappa_{fib}\right) + \max_{S \in \mathcal{S}_{fib}^{\neq 0}} (\Delta\check{z}_S(d_{fib})) \\ &= ((2) - (0)) + \Delta\check{z}_U(d_{fib}) \\ &= (2) + (2) = (4) \end{aligned}$$

4. Bestimmung der Allokationsvektormatrix $\mathcal{R}$.

   Da der Grad von $\widehat{z}_{fib}(d_{fib})$ gemäß Definition 2.13 1 ist, gibt es nur einen Allokationsvektor $\rho = \widehat{z}_{fib}(d_{fib}) = (4)$. Die Allokationsvektormatrix ist dementsprechend

$$\mathcal{R} = (4).$$

Die weiteren Schritte des Verfahrens gleichen dem funktionalen Fall und sind daher nicht weiter ausgeführt.

Es fällt auf, dass der erhaltene Allokationsvektor $\rho = (4)$ nur jede vierte Arrayzelle auf die gleiche Speicherzelle abbilden lässt, obwohl das Programm auch mit zwei Speicherzellen auskommt. Lemma 3.11 garantiert jedoch nicht, dass die kleinstmögliche Speicherallokation gefunden wurde, sondern nur ihre Gültigkeit. Ob die bestmögliche Allokationsmatrix gefunden werden kann, hängt unter anderem vom gegebenen Schedule ab. Siehe dazu auch Beispiel 3.4, Abschnitt 3.7.3.

3.6.2 Pascalsches Dreieck

Ein Programm zur Berechnung des Pascalschen Dreiecks ist in Abbildung 3.1 gezeigt. Die folgenden Schedules für die auf PD schreibenden Anweisungen sind gemeinsam

```
  CONSTANT n,m;

S PD[0,0]=1;
  for(i=1; i<=n; i++) {
T  PD[i,0]=1;
U  PD[i,i]=1;
   for(j=1; j<i; j++) {
V   PD[i,j] = PD[i-1,j-1]
      + PD[i-1,j];
   }
  }
```

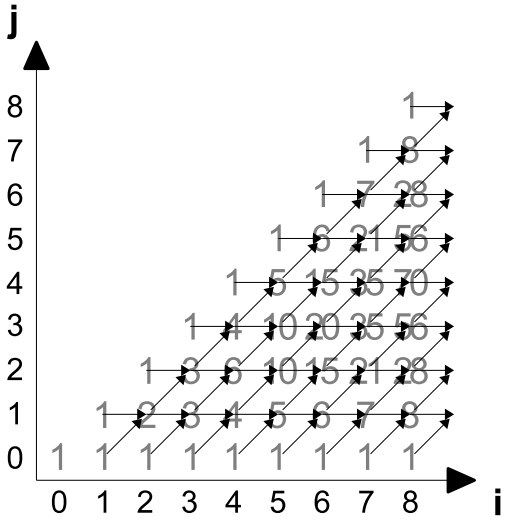

Abbildung 3.1: Programm Pascalsches Dreieck mit Abhängigkeiten im Quellraum

gültig:

$$\begin{aligned}\Theta_S() &= (0)\\ \Theta_T(i) &= (i-1)\\ \Theta_U(i) &= (i-1)\\ \Theta_V(i,j) &= (i-1)\end{aligned}$$

Die Abhängigkeiten sind in der Abbildung bereits eingezeichnet:

$$\begin{array}{lrcll} F_1: & \langle(i),\ T\rangle & \rightarrow & \langle(i+1,1),\ V\rangle & \text{für } 1 \le i \le n-1\\ F_2: & \langle(i),\ U\rangle & \rightarrow & \langle(i+1,i),\ V\rangle & \text{für } 1 \le i \le n-1\\ F_3: & \langle(i,j),\ V\rangle & \rightarrow & \langle(i+1,j),\ V\rangle & \text{für } 1 \le i \le n-1\\ F_4: & \langle(i),\ V\rangle & \rightarrow & \langle(i+1,j+1),\ V\rangle & \text{für } 1 \le i \le n-1 \end{array}$$

1. Überprüfung, ob schreibende Arrayzugriffe streng indexmonoton.

   Hierfür müssen formal 16 Gleichungen überprüft werden, die hier aus Platzgründen nicht angegeben sind. Man erkennt jedoch durch bloßes Hinsehen: Jeder Schritt der äußeren Schleife ist ein Zeitschritt, und in jedem Schritt der Schleife werden die Arrayzellen $PD[i,0]$ bis $PD[i,i]$ beschrieben. Somit werden in jedem weiteren Zeitschritt nur Zellen mit echt größerem Index beschrieben.

   Also sind alle Schreibzugriffe auf PD streng indexmonoton steigend, $\psi_{PD} = 1$.

2. Bestimmung der maximalen Lebenszeit d_{fib} aller Arrayzellen.

   Für jede Abhängigkeit wird die partielle Lebenszeit nach Definition 3.4 berechnet:

$$\begin{aligned} d_{F_1,T,V}() &= \max_{(i)}((1)(i+1)+(-1)-(1)(i)-(-1)) &&= (1)\\ d_{F_2,U,V}() &= \max_{(i)}((1)(i+1)+(-1)-(1)(i)-(-1)) &&= (1)\\ d_{F_3,V,V}() &= \max_{(i)}((1)(i+1)+(-1)-(1)(i)-(-1)) &&= (1)\\ d_{F_4,V,V}(i) &= \max_{(i,j)}((1)(i+1)+(-1)-(1)(i)-(-1)) &&= (1) \end{aligned}$$

Alle partiellen Lebenszeiten sind von einem konkreten i unabhängig, für die maximale Lebenszeit ergibt sich $d_{PD} = \max\{(1),(1),(1),(1)\} = (1)$.

3. Bestimmung von $\widehat{z}_{PD}(d_{PD})$, der minimalen Anzahl an Arrayzellen, die nicht in d_{PD} Zeitschritten geschrieben werden können.

$$\begin{aligned}
\mathrm{Acc}_S(D_S) &= \{(0,0)\} \\
\mathrm{Acc}_T(D_T) &= \{(i,0) \mid 1 \le i \le n\} = \{(1,0),(2,0),\ldots,(n,0)\} \\
\mathrm{Acc}_T(D_U) &= \{(i,i) \mid 1 \le i \le n\} = \{(1,1),(2,2),\ldots,(n,n)\} \\
\mathrm{Acc}_T(D_V) &= \{(i,j)^T \mid 1 \le i \le n,\ 1 \le j \le i-1\} \\
&= \{(2,1),(3,1),(3,2),(4,1),\ldots,(4,3),\ldots,(n,n-1)\}
\end{aligned}$$

Der insgesamt kleinste beschriebene Arrayzellenindex aller Statements ist $[0,0]$:

$$\kappa_{PD} = \min\{\psi_{PD} z \mid z \in \mathcal{Z}_{PD}\} = \min \mathcal{Z}_{PD} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Nur Statement S wird einmalig ausgeführt, daher $\mathcal{S}_{PD}^{\neq 0} = \{T,U,V\}$. Der kleinste von diesen Statements beschriebene Arrayzellenindex ist $[1,0]$:

$$\kappa_{PD}^{\neq 0} = \min\left\{\psi_{PD} z \mid z \in \mathcal{Z}_{PD}^{\neq 0}\right\} = \min \mathcal{Z}_{PD}^{\neq 0} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$\mathcal{I}_S(d_{PD}) = \{i \mid i \in \mathbb{Z}^{c_S} \wedge \tau_S i \ge d_{PD}\}$ ist die Menge der Indexschritte, die ein Statement nach d_{PD} oder mehr Zeiteinheiten ausführen kann:

$$\begin{aligned}
\mathcal{I}_S(d_{PD}) &= \emptyset \\
\mathcal{I}_T(d_{PD}) &= \mathcal{I}_U(d_{PD}) = \{1,2,3,\ldots\} \\
\mathcal{I}_V(d_{PD}) &= \{(i,j) \mid i \ge 1, j \in \mathbb{Z}\}
\end{aligned}$$

Nun werden noch $\check{z}_S(d_{PD}) \in \psi_{PD}\mathcal{A}_S(\mathcal{I}_S(d_{PD}))$ für alle $S \in \mathcal{S}_{PD}^{\neq 0}$ so benötigt, dass $\left(\kappa_{PD}^{\neq 0} - \kappa_{PD}\right) + \check{z}_S(d_{PD})$ möglichst klein im Betrag ist:

$$\begin{aligned}
\check{z}_T(d_{PD}) &= \min\left\{\left|\begin{pmatrix} i \\ 0 \end{pmatrix}\right| \mid i \in \{1,2,3,\ldots\}\right\} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
\check{z}_U(d_{PD}) &= \min\left\{\left|\begin{pmatrix} i \\ i \end{pmatrix}\right| \mid i \in \{1,2,3,\ldots\}\right\} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
\check{z}_V(d_{PD}) &= \min\left\{\left|\begin{pmatrix} i \\ j \end{pmatrix}\right| \mid \begin{matrix} i \ge 1 \\ j \in \mathbb{Z} \end{matrix}\right\} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}
\end{aligned}$$

Somit ergibt sich für $\widehat{z}_{PD}(d_{PD})$:

$$\begin{aligned}
\widehat{z}_{PD}(d_{PD}) &= \left(\kappa_{PD}^{\neq 0} - \kappa_{PD}\right) + \max_{S \in \mathcal{S}_{PD}^{\neq 0}} (\check{z}_S(d_{PD})) \\
&= \left(\begin{pmatrix} 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \end{pmatrix}\right) + \max\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}\right\} \\
&= \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}
\end{aligned}$$

4. Bestimmung der Allokationsvektormatrix $\mathcal{R}$.

   Da der Grad von $\widehat{z}_{PD}(d_{PD})$ 1 ist, gibt es nur einen Allokationsvektor $\rho = \widehat{z}_{PD}(d_{PD}) = (2,1)^T$. Die Allokationsvektormatrix ist dementsprechend

$$\mathcal{R} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

Wiederum ist der Allokationsvektor größer als nötig: Auch $(1,0)^T$ würde genügen.

3.7 Sonderfälle

In den beiden in Abschnitt 3.6 gezeigten Beispielen wurden korrekte, aber keine optimalen im Sinne von möglichst kleinen Allokationsvektoren gefunden. Es lassen sich auch noch beliebig mehr solche Beispiele finden, doch auch Beispiele, in denen optimale Allokationsvektoren gefunden werden – die in Abschnitt 3.4 gezeigte Auswahl von Allokationsvektoren ist also nicht allgemein zu schlecht, sondern nur für einige Sonderfälle. Einige davon sollen hier definiert und eine für den Fall bessere Auswahl an Vektoren vorgestellt werden. Die Sonderfälle sind auch kombinierbar.

Definition 3.16 Θ_S^1 *bezeichnet die erstmögliche Ausführungszeit eines Statements* $S \in \mathcal{S}_Y$.

3.7.1 Nichtkonstant ausgeführte Statements nur gemeinsam

Die Auswahl von Allokationsvektoren berücksichtigt, dass es im Allgemeinen Zeitpunkte geben kann, an denen ein nichtkonstantes Statement ausgeführt wurde, ein anderes jedoch noch nicht: Im Beweis zu Lemma 3.11 heißt es im Fall 1a, dass *ein* $T \in \mathcal{S}_Y^{\neq 0}$ bereits ausgeführt wurde und man erhält

$$\Delta z(d_Y) \leq \Delta \check{z}_T(d_Y) \leq \max_{S \in \mathcal{S}_Y^{\neq 0}} (\Delta \check{z}_S(d_Y)).$$

Betrachtet man jedoch folgendes (nutzlose) Programm mit $\Theta_S = \Theta_T = (i)$, sieht man, dass nach diesem Verfahren die Auswahl des Allokationsvektors viel zu groß ist:

```
   for  (i=1; i<=n; i++) {
     for (j=0; j<(5*i); j++) {
S:     A[i,5*j] = A[i-1,0];
      }
T:   A[i,5*i] = A[i-1,0];
    }
```

Die Lebenszeit ist leicht erkennbar $d_A = 1$ und die Schreibzugriffe auf das Array sind streng indexmonoton.
Nach einem Zeitschritt wird mindestens ein i-Indexschritt durchgeführt:

$$\begin{aligned} \check{z}_S(d_A) &= \min \left\{ \left| \begin{pmatrix} i \\ 5j \end{pmatrix} \right| \; | \; \begin{matrix} i \geq 1 \\ j \in \mathbb{Z} \end{matrix} \right\} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \check{z}_T(d_A) &= \min \left\{ \left| \begin{pmatrix} i \\ 5i \end{pmatrix} \right| \; | \; i \in \{1,2,3,\dots\} \right\} = \begin{pmatrix} 1 \\ 5 \end{pmatrix} \end{aligned}$$

Es wird das Maximum der beiden Werte genommen, also ist $\mathcal{R} = (1,5)^T$. Leicht nachvollziehbar wäre jedoch auch ein Allokationsvektor $(1,0)^T$ ausreichend.

Im Unterschied zur allgemeinen Situation, für die der Beweis zu Lemma 3.11 konzipiert ist, gilt hier $\forall S,\ T \in \mathcal{S}_Y^{\neq 0} : \Theta_S^1 = \Theta_T^1$: es gibt keinen Zeitpunkt, an denen nur T und nicht auch S ausgeführt wurde. In Fall 1a des Beweises kann man daher davon ausgehen, dass *alle* $T \in \mathcal{S}_Y^{\neq 0}$ bereits ausgeführt wurden, und die hergeleitete Gleichung gilt daher für alle T:

$$\forall T \in \mathcal{S}_Y^{\neq 0} : \Delta z(d_Y) \leq \Delta \check{z}_T(d_Y)$$

Hieraus folgt sofort $\Delta z(d_Y) \leq \min_{S \in \mathcal{S}_Y^{\neq 0}} (\Delta \check{z}_S(d_Y))$ und der für die Allokationsvektorauswahl maßgebliche Wert $\widehat{z}_Y(d_Y)$ kann neu defniniert werden:

$$\widehat{z}_Y(d_Y) = \left(\kappa_Y^{\neq 0} - \kappa_Y\right) + \min_{S \in \mathcal{S}_Y^{\neq 0}} (\check{z}_S(d_Y))$$

Falls $\left|S \in \mathcal{S}_Y^{\neq 0}\right| = 1$, ist die genannte Bedingung trivial erfüllt. Da dann allerdings auch $\min_{S \in \mathcal{S}_Y^{\neq 0}} (\check{z}_S(d_Y)) = \max_{S \in \mathcal{S}_Y^{\neq 0}} (\check{z}_S(d_Y))$ gilt, ist die ursprüngliche Definition von $\widehat{z}_Y(d_Y)$ in diesem Fall genauso gut.

3.7.2 Konstant und nichtkonstant ausgeführte Statements nicht gemeinsam

Es gelte $\forall S \in (\mathcal{S}_Y - \mathcal{S}_Y^{\neq 0}),\ T \in \mathcal{S}_Y^{\neq 0} : \Theta_S^1 + d_Y \leq \Theta_T^1$.

Unter dieser Bedingung gilt im Beweis zu Lemma 3.11 in Fall 1b:

z_0 wird von $S \notin \mathcal{S}_Y^{\neq 0}$ zum Zeitpunkt t_0 geschrieben und zum Zeitpunkt t überschrieben. $\Delta t = t - t_0 < d_Y$. Ein $U \in \mathcal{S}_Y^{\neq 0}$ wird aufgrund der Bedingung frühestens zum Zeitpunkt $(t_0 + d_Y)$ aktiv:

$$t_U \geq (t_0 + d_Y) > t.$$

Somit kann Fall 1(b)ii entfallen und für Fall 1b gilt nur noch:

$$\Delta z(\Delta t) < \kappa_Y^{\neq 0} - \kappa_Y$$

Der für die Allokationsvektorauswahl maßgebliche Wert $\widehat{z}_Y(d_Y)$ kann damit unter Berücksichtigung von Fall 1a wie folgt definiert werden:

$$\widehat{z}_Y(d_Y) = \max\left(\left(\kappa_Y^{\neq 0} - \kappa_Y\right), \max_{S \in \mathcal{S}_Y^{\neq 0}} (\check{z}_S(d_Y))\right)$$

Beispiel 3.3 *Das in Abbildung 3.1 gezeigte Programm zur Berechnung des Pascalschen Dreiecks erfüllt die Vorbedingung mit den gegebenen Schedules nicht, wohl aber mit den Schedules* $\Theta_S() = 0$, $\Theta_T(i) = \Theta_U(i) = \Theta_V(i,j) = i$.

Mit der somit gültigen obigen Definition für $\widehat{z}_Y(d_Y)$ *erhält man eine bessere Allokationsvektormatrix* $\mathcal{R} = (1,1)^T$. *Diese ist jedoch immer noch nicht optimal.*

Falls $\mathcal{S}_Y - \mathcal{S}_Y^{\neq 0} = \emptyset$, ist die genannte Bedingung trivial erfüllt. Dann ist allerdings auch $\left(\kappa_Y^{\neq 0} - \kappa_Y\right) = 0^{n_Y}$ und die ursprüngliche Definition von $\widehat{z}_Y(d_Y)$ genauso gut.

3.7.3 Starke Indexmonotonie

Die auf Seite 22 gezeigte Definition der Indexmonotonie wird wie folgt verschärft:

Definition 3.17 *Alle Schreibzugriffe auf ein Array Y sind* stark indexmonoton, *wenn eine der folgenden Bedingungen gilt:*

$$\forall i_S \in \mathbb{Z}^{c_S}, i_T \in \mathbb{Z}^{c_T}, S,T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \Rightarrow \mathrm{Acc}_S(i_S) < \mathrm{Acc}_T(i_T)$$

$$\forall i_S \in \mathbb{Z}^{c_S}, i_T \in \mathbb{Z}^{c_T}, S,T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \Rightarrow \mathrm{Acc}_S(i_S) > \mathrm{Acc}_T(i_T)$$

Die Indexmonotonie gilt demnach für jede Anweisung im gesamten *Indexvektorraum*.

Für den Beweis zu Lemma 3.11 folgt aus der starken Indexmonotonie: Es kann immer angenommen werden, dass bereits eine Anweisung $T \in \mathcal{S}_Y^{\neq 0}$ ausgeführt wurde. Auch wenn dies noch gar nicht der Fall war, garantiert die starke Indexmonotonie, dass alle schreibenden Arrayzugriffe so ausgeführt werden, als ob T bereits aktiv gewesen wäre. Der komplette Fall 1b des Beweises entfällt damit. Doch auch Fall 1a muss geändert werden: Da für *jede* Anweisung $S \in \mathcal{S}_Y^{\neq 0}$ angenommen werden kann, dass sie bereits ausgeführt wurde, ändert sich der Fall wie in Abschnitt 3.7.1 gezeigt. Insgesamt erhält man unter der starken Indexmonotonie für $\widehat{z}_Y(d_Y)$ die folgende Definition:

$$\widehat{z}_Y(d_Y) = \min_{S \in \mathcal{S}_Y^{\neq 0}} (\check{z}_S(d_Y))$$

Beispiel 3.4 *Das in Beispiel 3.1 gezeigte Programm zur Berechnung der Fibonacci-Zahlen ist* nicht *stark indexmonoton. Die Voraussetzung ist für Statement S und U mit $i_U = 0$ nicht erfüllt:*

$$\begin{aligned} \Theta_U(i_U) < \Theta_S(i_S) &\Rightarrow \mathrm{Acc}_U(i_U) < \mathrm{Acc}_S(i_S) \\ (i_U - 1) < (0) &\Rightarrow (i_U) < (0) \\ (-1) < (0) &\Rightarrow (0) \not< (0) \end{aligned}$$

Werden stattdessen die Schedules $\Theta_S() = 0$, $\Theta_T() = 1$ und $\Theta_U(i) = i$ verwendet, ist das Programm stark indexmonoton und das Verfahren kann die optimale Allokationsmatrix $\mathcal{R} = (2)$ errechnen.

Auch das Programm Pascalsches Dreieck ist mit den in Beispiel 3.3 gegebenen Schedules stark indexmonoton. Mit der somit gültigen Definition für $\widehat{z}_Y(d_Y)$ erhält man die optimale Allokationsvektormatrix $\mathcal{R} = (1,0)^T$.

Korollar 3.18 *Sei in einem Programm mit indexmonotonen Schreibzugriffen auf Y $|\mathcal{S}_Y| = \left|\mathcal{S}_Y^{\neq 0}\right| = 1$. Dann sind die Schreibzugriffe auf Y* stark *indexmonoton. In diesem besonderen Fall liefert auch die ursprüngliche Definition von $\widehat{z}_Y(d_Y)$ den optimalen Satz Allokationsvektoren, da $\left(\kappa_Y^{\neq 0} - \kappa_Y\right) = 0^{n_Y}$ und $\min_{S \in \mathcal{S}_Y^{\neq 0}} (\check{z}_S(d_Y)) = \max_{S \in \mathcal{S}_Y^{\neq 0}} (\check{z}_S(d_Y))$ gelten.*

3.8 Erweiterungen

Auch trotz der im vorherigen Abschnitt definierten Sonderfälle gibt es Programme, deren Speichernutzung sich nicht oder nicht so gut wie möglich mit dem hier gezeigten Vorgehen optimieren lässt. Mit den folgenden Vorschlägen zur Erweiterung könnte das Verfahren auch in diesen Fällen zu guten bzw. besseren Ergebnissen führen.

3.8.1 Herstellung von Indexmonotonie

Wie schon in Abschnitt 3.1 erwähnt, ist die Reihenfolge der Arrayindizes entscheidend für die Indexmonotonie. Die Matrixmultiplikation ist ein weiteres Beispiel für so ein Programm:

```
parallel for(i=0; i<n; i++) {
   parallel for(j=0; j<n; j++) {
      C[i,j,0]=0;
      for(k=0; k<n; k++) {
          C[i,j,k+1] = C[i,j,k] + A[i,k] * B[k,j];
}}}
```

Das Schreibzugriffe auf C sind nicht indexmonoton. Vertauscht man jedoch die Reihenfolge der Indexdimensionen so, dass k die erste Dimension ist, wird Indexmonotonie im Parallelen hergestellt[2], und das Programm führt weiterhin die gleichen Berechnungen durch.

Auch kann Indexmonotonie durch die Reihenfolge der Statements zerstört werden. Als Beispiel dient noch einmal das in Abbildung 3.1, Seite 39 vorgestellte Programm zur Berechnung des Pascalschen Dreiecks: Mit sequentiellem Schedule schreibt Statement U in $PD[2, 2]$, bevor Statement V in $PD[2, 1]$ schreibt. Würde U jedoch als letzte Anweisung innerhalb der i-Schleife ausgeführt, wäre das Programm semantisch äquivalent und indexmonoton.

Ein Verfahren, das solche Fälle erkennt und – beispielsweise durch die Verwendung einer anderen als der lexikographischen Ordnung, durch syntaktischen Veränderung des Programms oder Anpassung der Statement-Schedules – Indexmonotonie herstellt, würde die Optimierung der Speichernutzung in mehr Programmen ermöglichen.

3.8.2 Feinere Granularität

Das vorliegende Verfahren bestimmt die Möglichkeit zur Optimierung der Speichernutzung über die maximale Lebenszeit aller Zellen eines Arrays und aller darauf schreibenden Statements. Stattdessen könnte die maximale Lebenszeit der geschriebenen Zellen für jedes Statement gesondert berechnet werden. Die Berechnung der Speicherallokation müsste umfangreich angepasst werden, doch bei Programmen mit verschiedenen auf das Array zugreifenden Statements kann die höhere Genauigkeit das Ergebnis verbessern.

Als Beispiel hierfür kann das Programm zur Berechnung der Fibonacci-Zahlen mit Schedule aus Beispiel 3.1 betrachtet werden: Die maximale Lebenszeit ist (2). Das Verfahren gibt also eine Speicherallokation an, bei der alle Zellen, also auch die von Statement S geschriebenen, zwei Zeitschritte im Speicher gehalten werden. S und T schreiben zur gleichen Zeit $fib[0]$ bzw. $fib[1]$, im darauf folgenden Zeitschritt schreibt U in $fib[2]$ und erst dann kann $fib[0]$ überschrieben werden; es müssen also die Arrayzellen 0 bis 2, insgesamt *drei* Zellen, im Speicher gehalten werden. Wie bereits gezeigt rechnet das Programm auch mit nur zwei Arrayzellen im Speicher korrekt, da die von Statement S beschriebene Zelle nur *einen* Zeitschritt im Speicher gehalten

[2] Interessanterweise wird durch die Vertauschung der Indexdimensionen in diesem Fall die Indexmonotonie im *Sequentiellen* zerstört.

werden muss. Würde die maximale Lebenszeit pro Statement berechnet, kann ein Verfahren diese Information nutzen und beim gegebenen Schedule die (optimale) Speicherallokation der Größe 2 finden.

3.8.3 Nutzung von verteiltem Speicher

Es wurden bisher nur Parallelrechner mit gemeinsamen Speicher betrachtet. Bei verteiltem Speicher ist es unter Umständen möglich, den genutzten Speicher auf jedem Einzelrechner zu reduzieren, da lediglich die Daten für die auf dem Einzelrechner ausgeführten Operationen benötigt werden. Eine Ausweitung des Verfahrens auf Systeme mit verteiltem Speicher ist daher sinnvoll, aber die Betrachtungsweise ist zunächst völlig anders, als die hier vorgestellte: Die vorliegende Arbeit setzt sich intensiv mit dem Schedule des Parallelprogramms auseinander, für die Speichernutzung auf einem Einzelrechner muss vorrangig das Placement betrachtet werden.

Für weitere Forschungen möchte ich an dieser Stelle eine Grundidee für eine Vorgehensweise bei verteiltem Speicher aufzeigen, die sich mit dem hier gezeigten Verfahren nicht nur kombinieren lässt, sondern durch die Herstellung *starker* Indexmonotonie auch gegebenenfalls erst ermöglicht:

Das *Space-Time Mapping* gibt für jede Operation eindeutig an, *wann* und *wo* sie durchgeführt wird; die Abbildung ist bijektiv, es ist also auch umgekehrt für jeden Punkt (t, p) eindeutig festgelegt, welche Instanz welcher Anweisung zu dieser Zeit-/Raum-Koordinate ausgeführt wird und damit auch, in welche Zelle welches Originalarrays geschrieben wird. Es spricht folglich nichts dagegen, alle Schreibzugriffe im Zielprogramm so umzuformen, dass jeder Prozessor nicht in das Originalarray schreibt, sondern in ein temporäres Array an Position $[t, p]$. Die Transformation der Lesezugriffe könnte wie bei Feautriers *Single Assignment-Konversion* [7], jedoch angewandt auf das Zielprogramm und die Zielraumabhängigkeiten, erfolgen.

Die Schreibzugriffe auf das neue Array wären dann stark indexmonoton gemäß Definition 3.17 (und damit auch gut verträglich mit dem Hardware-Cache). Das Array könnte somit unter besten Voraussetzungen nach dem hier vorgestellten Verfahren speichernutzungsoptimiert werden. Auch müsste bei verteiltem Speicher nicht auf jedem Prozessor das gesamte Array im Speicher gehalten werden: Prozessor p müsste neben Zeile p, auf die er selber schreibt, nur die den Prozessoren zugeordneten Zeilen halten, von deren Berechnungen seine eigenen Operationen abhängig sind.

Kapitel 4

Einfluss des Schedules

Das vorgestellte Verfahren zur Optimierung der Speichernutzung arbeitet auf dem bereits parallelisierten Programm und verwendet die gegebenen Schedules. Diese bestimmen darüber, ob und wieviel unnötig genutzter Speicher überhaupt eingespart werden kann: oft bedeutet mehr Parallelität auch eine größere Speichernutzung und eine kleinere Speichernutzung weniger Parallelität. Ein einfaches Beispiel verdeutlicht das:

```
   for (i=0; i<=n; i++)
S:   a[i] = 42;
```

Mit sequentiellem Schedule lässt sich das Array zu einem Skalar wandeln. Ist dagegen $\Theta_S(i) = 0$, werden also alle Zuweisungen parallel ausgeführt, muss das Array in vollem Umfang im Speicher abgebildet werden.

Nicht immer bedeutet eine höhere Parallelität jedoch auch eine geringere Programmlaufzeit, daher kann es vorkommen, dass die Möglichkeit zur Optimierung der Speichernutzung nutzlos verhindert wird. Darauf wird im nächsten Abschnitt genauer eingegangen und in Abschnitt 4.2 ein einfaches Konzept zur Verbesserung des Schedules in solchen Fällen gezeigt.

Außerdem wurde von William Thies [8] eine Methode erarbeitet, die auf Speicherverbrauch *und* Parallelität optimierte Schedules berechnet.

4.1 Probleme durch Free Schedules

Um kurze Programmausführungszeiten zu garantieren, wird oft ein sogenannter *Free Schedule* eingesetzt, der alle Operationen zum frühest möglichen Zeitpunkt ausführt. Dadurch wird jedoch schon einfachsten Programmen die Möglichkeit einer verbesserten Speichernutzung entzogen. Als Beispiel sei hier wieder das Programm zur Berechnung des Pascalschen Dreiecks aus Abbildung 3.1, Seite 39 genutzt.

In Beispiel 3.3 wurden Schedules angegeben, mit denen in Beispiel 3.4 die Allokationsmatrix $\mathcal{R} = (1, 0)^T$ gefunden werden konnte. Mit dieser Allokationsmatrix erhält man die Mem-Funktion

$$\mathrm{Mem}(i, j) = (j).$$

Eine ganze Dimension des Arrays kann eingespart werden!

Schedule	
$\Theta_S()$	$= 0$
$\Theta_T(i)$	$= 0$
$\Theta_U(i)$	$= 0$
$\Theta_V(i,j)$	$= i - 1$

Maximale Lebenszeit	
$F_1 : T \to V$:	∞
$F_2 : U \to V$:	∞
$F_3 : V \to V$:	1
$F_4 : V \to V$:	1

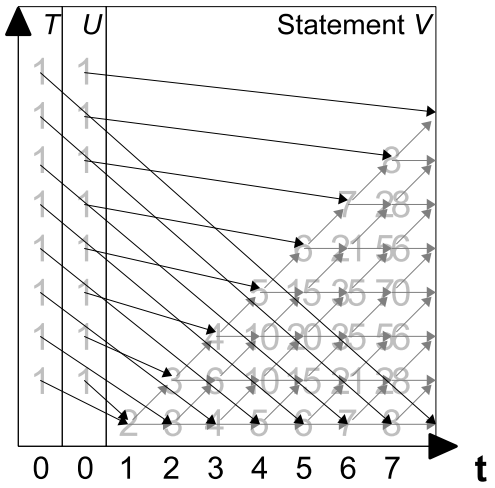

Abbildung 4.1: Free Schedule verhindert verbesserte Speichernutzung

Ein Free Schedule lässt jedoch die unabhängigen Statements T und V zum Zeitpunkt 0 ausführen. Dies macht das Gesamtprogramm nicht schneller – $\Theta_V(i)$ bleibt gleich –, doch jede Anweisungsinstanz wird dann so früh wie möglich ausgeführt. Die Auswirkung auf die Abhängigkeiten im Zielraum sind in Abbildung 4.1 dargestellt: In jedem späteren Zeitpunkt müssen Operationen auf Arrayzellen zugreifen, die zu Zeit 0 geschrieben wurden; die Lebenszeiten sind nicht mehr nach oben beschränkt. Eine Verbesserung der Speichernutzung ist unter diesem Schedule nicht möglich.

Dies betrifft alle Programme, die Initialisierungsdaten schreiben. Da Initialisierungen meist keine Abhängigkeit haben, wird ein Free Schedule Initialisierungsstatements ganz an den Anfang der Ausführungszeit stellen, und viel später ausgeführte Operationen müssen auf sehr früh geschriebene Daten zugreifen. Da fast jedes Programm mit Initialisierungsdaten arbeitet, ist auch fast jedes Programm unter einem Free Schedule nicht in seiner Speichernutzung verbesserbar. Auch wirkt dieses Verhalten dem Hardware-Cache entgegen und führt daher in der Praxis sogar zu einer Verlängerung der Ausführungszeit.

4.2 Heuristik zur Schedule-Verbesserung

In diesem Abschnitt wird eine Heuristik vorgestellt, die in einfachen Fällen einen Free Schedule soweit modifiziert, dass die Optimierung der Speichernutzung ermöglicht wird. Die Heuristik ist allerdings nicht optimal.

4.2.1 Idee

Der Free Schedule zerstört die Möglichkeit des Speichersparens dadurch, dass er Code möglichst *früh* ausführt. Es muss dafür gesorgt werden, dass Quellstatements, die zu unbeschränkten Lebenszeiten der Arrayzellen führen, stattdessen möglichst *spät* ausgeführt werden, ohne die Latenz des Schedules zu ändern.

Beim Beispielprogramm sind – unter Verwendung des Free Schedules aus Abbildung 4.1 – unter anderem die Lebenszeiten der in Statement T in das Array

Schedule	
$\Theta_S()$	$= 0$
$\Theta_T(i)$	$= i - 1$
$\Theta_U(i)$	$= i - 1$
$\Theta_V(i,j)$	$= i - 1$

Maximale Lebenszeit	
$F_1 : T \to V$:	1
$F_2 : U \to V$:	1
$F_3 : V \to V$:	1
$F_4 : V \to V$:	1

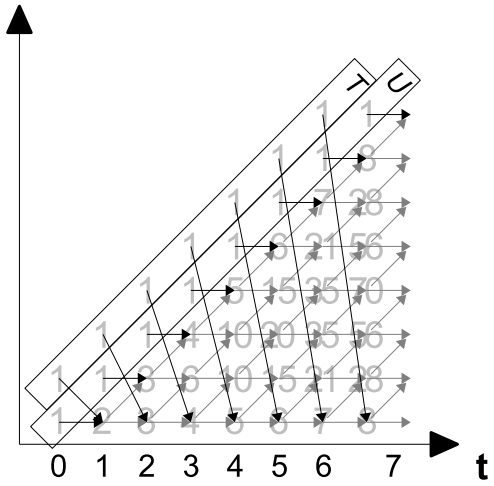

Abbildung 4.2: Änderung des Schedules zur Optimierung der Speichernutzung

geschriebenen und in Statement V gelesenen Werte nicht nach oben beschränkt, da jede Instanz von Statement T zu Zeitpunkt 0 ausgeführt wird, die Instanzen von Statement V jedoch in Abhängigkeit des i-Wertes. Es ist leicht zu sehen: Ändert man den Schedule von T so, dass er von i abhängt, ist die Lebenszeit der zugehörigen Arrayzellen wie gewünscht nach oben beschränkt. Dies lässt sich analog auch für Statement U durchführen. In Abbildung 4.2 sind die Abhängigkeitsvektoren unter Verwendung solch eines Schedules dargestellt: Statement T und U sind bildlich an Statement V heran „verkippt“, die Abhängigkeitsvektoren haben nun eine beschränkte Größe in Richtung der Zeitachse.

Die Heuristik arbeitet genau nach dem an diesem Beispiel aufgezeigten Konzept: Der Schedule der Quellstatements von Flow-Abhängigkeiten, bei denen die Lebenszeit nicht beschränkt ist, wird an den Schedule des zugehörigen Zielstatements angepasst und so spät ausgeführt, wie es unter Einhaltung der Gültigkeit möglich ist. Die genaue Arbeitsweise wird in Abschnitt 4.2.3 erläutert.

4.2.2 Wahl der Reihenfolge der Statements

Hat jedes Quellstatement nur zu einem einzigen anderem Statement führende Abhängigkeiten, führt das gerade vorgestellte Vorgehen immer zum Erfolg; sonst jedoch ist die Reihenfolge wichtig, in der die Statement-Schedules bei diesem Verfahren bearbeitet werden.

Die zu ändernden Statement-Schedules sollten für gute Ergebnisse der Heuristik in folgender Reihenfolge gewählt werden: Alle Abhängigkeiten, deren maximale Lebenszeit unbeschränkt ist, werden nach der Ausführungszeit der ersten Instanz ihres *Zielstatements aufsteigend* sortiert; alle Abhängigkeiten mit gleichem Zielstatement werden nach der ersten Ausführungszeit ihres *Quellstatements absteigend* sortiert. Dieses Vorgehen ergibt nicht zwingend die besten Ergebnisse und kann in Einzelfällen auch sehr schlechte Ergebnisse verursachen, ist aber für die meisten Programme vorteilhaft:

Die absteigende Sortierung der Quellstatements sichert in jedem Schritt den Er-

halt eines gültigen Schedules. Betrachten wir noch einmal Abbildung 4.2: Statement T und U sind an Statement V „gekippt“. Im allgemeinen Fall kann es eine Abhängigkeit zwischen Instanzen von Statement T zu U geben, dann würde T im Originalschedule zeitlich vor U ausgeführt. Würde dann zuerst T „verkippt“, würden Instanzen von U eventuell vor den Instanzen von T, von denen sie abhängig sind, ausgeführt – der Schedule wäre damit ungültig. Wird Statement U zuerst bearbeitet, wird das vermieden. In diesem speziellen Beispiel allerdings gibt es keine Abhängigkeiten zwischen T und U – sie werden deshalb auch zur gleichen Zeit ausgeführt –, die Bearbeitungsreihenfolge ist daher unwichtig.

Die aufsteigende Sortierung nach Zielstatements ist aus zwei Gründen vorteilhaft:

Der Schedule des Quellstatements soll so verändert werden, dass es so spät wie möglich vor dem Zielstatement ausgeführt wird. Gibt es für ein Quellstatement mehrere Zielstatements, so muss es natürlich so spät wie möglich vor dem *ersten* Zielstatement ausgeführt werden, sonst wäre der Schedule ungültig.

Wichtiger jedoch: Programme mit Berechnungen in Schleifen, mit denen wir uns maßgeblich beschäftigen, haben meist den Aufbau *Initialisierung – Schleifenberechnung – Rückgabe*. Auch die hier vorgestellten Beispielprogramme lassen sich so kategorisieren. Beim Beispielprogramm aus Abbildung 3.1 wird in Statement S bis U initialisiert und Statement V berechnet. Ein Rückgabestatement ist im Beispiel nicht dargestellt, aber denkbar. Abhängigkeiten zeigen von den Initialisierungen zu allen anderen Statements und von der Berechnung zur Rückgabe. Der Free Schedule weist der Initialisierung einen festen Zeitpunkt ganz am Anfang der Ausführung zu (im Beispiel: 0), dann folgt die Berechnung (im Beispiel: $i - 1$) und am Schluss wird die Rückgabe zu einem eventuell konstanten, von der Größe der Parameter abhängigen Zeitpunkt ausgeführt. Die problematischen Abhängigkeiten führen von den Initialisierungsstatements ab. Würde zuerst eine Abhängigkeit von Initialisierung zu Rückgabe von der Heuristik bearbeitet, würde der Schedule eines Initialisierungsstatements an den des Rückgabestatements angepasst. Da dieses unter Umständen zu einer konstanten Zeit ausgeführt wird, könnten auch die Initialisierungen zu einer konstanten Zeit ausgeführt werden – was jedoch genau das ist, was die Heuristik vermeiden soll. Es ist also besser, wenn zuerst die Abhängigkeiten zu den Berechungsstatements bearbeitet werden.

4.2.3 Algorithmus

Hier wird ein algorithmisches Verfahren vorgestellt, das das in Abschnitt 4.2.1 vorgestellte Heuristikprinzip zur Verbesserung eines Schedule zur Verfügung stellt. Da die Heuristik die Anwendung des in Kapitel 3 vorgestellten Verfahrens ermöglichen soll, das nur für Programme definiert ist, die indexmonoton auf Arrays zugreifen und somit in *Single Assignment*-Form sind, werden nur Flow- und keine Anti- und Output-Abhängigkeiten beachtet.

1. Setze Liste der veränderten Statements $ChangedStats := \{\}$.

   Diese Liste speichert, welche Statements bisher von der Heuristik verändert wurden. Dabei wird eine Verschiebung des Statements um lediglich eine Konstante nicht gewertet.

2. Führe für jede Flow-Abhängigkeit eine Analyse auf maximale Lebenszeit im Zielraum durch.

   Sind alle Lebenszeiten beschränkt, springe direkt zu Schritt 12.
   Gibt es eine Abhängigkeit mit unbeschränkter Lebenszeit, deren Quell- und Zielstatement gleich ist, breche ab. Eine Anpassung des Schedules des Quellstatements an sich selbst ist obsolet.
   Ansonsten weiter mit Schritt 3.

3. Wie im vorherigen Abschnitt erläutert, arbeitet die Heuristik am besten, wenn sie Ausführungszeiten der frühest ausgeführten Zielstatements mit den spätest ausgeführten zugehörigen Quellstatements abgleicht. Wähle eine Abhängigkeit mit entsprechenden Quell- und Zielstatements $curDep : srcStmt \rightarrow dstStmt$ möglichst wie folgt aus:

   (a) Die in Schritt 2 berechnete maximale Lebenszeit von $curDep$ ist unbeschränkt.

   (b) $dstStmt$ ist von allen Zielstatements, auf die Abhängigkeiten zeigen, bei denen Voraussetzung 3a zutrifft, das frühest ausgeführte.

   (c) $srcStmt$ ist von allen Quellstatements von Abhängigkeiten, bei denen Voraussetzung 3a zutrifft und deren Ziel $dstStmt$ ist, das spätest ausgeführte.

   Es wird nun versucht, den Schedule von $srcStmt$ so zu verändern, dass die Lebenszeit von $curDep$ nach oben beschränkt ist.

4. Falls $srcStmt \in ChangedStats$, breche ergebnislos ab.
   Falls der Schedule des gewählten Quellstatements $srcStmt$ schon einmal von der Heuristik verändert worden ist, wurde damit die maximale Lebenszeit einer Abhängigkeit beschränkt; trotzdem ist die Lebenszeit von $curDep$ unbeschränkt. Würde der Schedule von $srcStmt$ nun so angepasst, dass die Lebenszeit von $curDep$ beschränkt wird, würde damit eventuell die vorherige Verbesserung wieder rückgängig gemacht – die Heuristik könnte in eine Endlosschleife geraten. Daher wird das zweimalige Verändern des Schedules eine Statements, abgesehen um das Verschieben um eine Konstante, unterbunden.

   Ansonsten setze $ChangedStats := ChangedStats \cup srcStmt$.

5. Führe für $curDep$ eine Analyse auf minimale Lebenszeit im Zielraum durch. Ist die minimale Lebenszeit nicht beschränkt – das kann z.B. passieren, wenn sie immer einen Parameter enthält –, so kann auch die maximale Lebenszeit nicht beschränkt werden: der Algorithmus wird abgebrochen.

6. Der Schedule von $srcStmt$ wird an den von $dstStmt$ angeglichen: dazu werden alle Indexraumvariablen, deren Koeffizienten im $dstStmt$-Schedule ungleich 0 gesetzt sind (und somit die Ausführungszeit von $dstStmt$-Instanzen bestimmen) und auch im Indexraum von $srcStmt$ vorhanden sind, auch dort auf den gleichen Wert wie im $dstStmt$-Schedule gesetzt. Ist eine der mitbestimmenden Indexraumvariablen von $dstStmt$ im Indexraum von $srcStmt$ jedoch nicht vorhanden, ist das Angleichen nicht möglich, die Heuristik wird abgebrochen. Der konstante Anteil des $srcStmt$-Schedules wird in erster Näherung gleich

dem konstanten Anteil des *dstStmt*-Schedules weniger 1 gesetzt und in den folgenden drei Schritten überprüft und gegebenenfalls verbessert.

Der neue Schedule zählt mindestens so viele Zeitpunkte auf wie vor der Anpassung, die Gesamtausführungszeit von *srcStmt* ist daher länger oder gleich lang wie vorher.

7. Für jede von *srcStmt* ausgehende Flow-Abhängigkeit wird die minimale Lebenszeit ermittelt und der kleinste Wert übernommen. Existiert eine minimale Lebenszeit kleiner als 1, ist der Schedule ungültig; ist jede minimale Lebenszeit dagegen echt größer als 1, ist die Gesamtausführungszeit länger als nötig. Auch wird aufgrund der verlängerten Lebenszeit die Möglichkeit, Speicher einzusparen, verringert. Daher wird die Konstante im *srcStmt*-Schedule nun so verändert und die Ausführungszeit der Statement-Instanzen damit so verschoben, dass die kleinste minimale Lebenszeit aller ausgehenden Abhängigkeiten 1 ist.

8. Analog zum vorherigen Schritt wird die kleinste minimale Lebenszeit aller auf *srcStmt* zeigenden Flow-Abhängigkeiten berechnet und *srcStmt* sowie alle davon abhängenden Statements entsprechend verschoben.

9. Durch die Angleichung und Verschiebung von *srcStmt* kann dessen erste Ausführungszeit früher als vor der Schedule-Änderung sein. Es kann daher vorkommen, dass ein Schedule erzeugt wird, der vor Zeitpunkt 0 beginnt. Ist dies der Fall, werden alle Statements so nach hinten verschoben, dass 0 der erste Schedule-Zeitpunkt ist.

10. Die Heuristik nimmt am Schedule nur Änderungen vor, bei denen die Gültigkeit erhalten bleiben sollte. Jedoch wurde die Korrektheit nicht bewiesen und eventuell vorhandene Anti- und Outputabhängigkeiten werden nicht beachtet, daher wird zur Sicherheit noch einmal überprüft, ob ein gültiger Schedule erzeugt wurde. Ist dies nicht der Fall, wird abgebrochen.

11. Wenn bis hierhin nicht abgebrochen wurde, sollte eine vorher unbeschränkt große Lebenszeit einer Abhängigkeit beschränkt worden sein. Da aber noch andere existieren können, wird zurück zu Schritt 2 gesprungen.

12. Sind keine unbeschränkt großen Lebenszeiten mehr vorhanden, kann die Optimierung der Speichernutzung durchgeführt werden.

Kapitel 5

Implementation

Im Rahmen dieser Diplomarbeit wurden das in Kapitel 3 vorgestellte Verfahren zur Optimierung der Speichernutzung sowie die Heuristik zur Schedule-Verbesserung aus Abschnitt 4.2 zur Verwendung im automatischen Schleifenparallelisierer LooPo [5] in der funktionalen Programmiersprache Haskell implementiert.

Es ist im Unterverzeichnis `QuilMem` aller Haskell-Quellcodes zu finden und besteht aus drei Modulen: dem User-Interface `Main.hs`, `BetterSchedule.hs` beinhaltet die Routinen der Heuristik und `QuilMem.hs` stellt das Verfahren zur Optimierung der Speichernutzung zur Verfügung.

5.1 Main.hs

Das Interface erwartet als Argumente ein LooPo-Temp-Verzeichnis und den Namen eines Programms, das mit LooPo analysiert und als Zielprogramm generiert wurde. Die `main`-Funktion delegiert an die `optMem`-Funktion, die folgende Schritte ausführt:

1. Auflistung aller im Programm vorhandenen Arrays.

   Der Benutzer gibt an, welches der Arrays in seiner Speichernutzung optimiert werden soll.

2. Berechnung der Lebenszeit des Arrays

   Es werden die entsprechenden Funktionen von `QuilMem.hs` aufgerufen.

3. Ist die Lebenszeit unbeschränkt: Versuch, den Schedule mittels Heuristik zu verbessern.

   Es werden die entsprechenden Funktionen von `BetterSchedule.hs` aufgerufen.

   Ist die Lebenszeit weiterhin unbeschränkt, Abbruch.

4. Überprüfung der Voraussetzung: Sind die Schreibzugriffe auf das gegebene Array indexmonoton?

   Es werden die entsprechenden Funktionen von `QuilMem.hs` aufgerufen.

   Falls die Arrayzugriffe nicht indexmonoton sind, Abbruch.

5. Bestimmung von $\hat{z}$, der Anzahl Arrayzellen, die nicht innerhalb der maximalen Lebenszeit beschrieben werden können, gemäß Definition 3.12.

   Es werden die entsprechenden Funktionen von `QuilMem.hs` aufgerufen.

6. Bestimmung der Allokationsvektormatrix sowie der Speicherallokation.

   Es werden die entsprechenden Funktionen von `QuilMem.hs` aufgerufen.

7. Ausgabe des Ergebnisses.

   Es werden die entsprechenden Funktionen von `QuilMem.hs` aufgerufen.

Nach dem Anwendungsschema aus Abschnitt 3.5 sollte zuerst die Indexmonotonie überprüft und dann erst die Lebenszeit berechnet werden. Die Reihenfolge dieser beiden Schritte ist jedoch für das Verfahren selbst unwichtig. In dieser Implementierung kann der Schedule aber bei unbeschränkter Lebenszeit noch in Schritt 3 verändert werden: dies beeinflusst die Indexmonotonie, daher wird sie erst in Schritt 4 überprüft.

5.2 QuilMem.hs

`QuilMem.hs` ist das Kernstück der Implementierung und stellt alle für das Verfahren nötigen Funktionen sowie eine Reihe Hilfsfunktionen zur Verfügung. Die Funktionen sind in der Datei dokumentiert, hier werden daher nur die wichtigsten Funktionsnamen sowie die in der Implementierung genutzten Techniken vorgestellt.

5.2.1 Lebenszeitanalyse

Die maximale Lebenszeit eines Arrays unter *einer einzigen* Abhängigkeit (Definition 3.5) erhält man mit `getDepLifeTimeDTree` bzw. `getDepLifeTimeDTrees`. Die maximale Lebenszeit des Arrays unter allen Abhängigkeiten, also d_Y, erhält man durch Aufruf von `maxLT` auf den Ergebnissen.

Zur Berechnung werden die in der LooPo-Datenstruktur vorhandenen Ungleichungssysteme der Abhängigkeiten im Zielraum genutzt. Die Berechnung der maximalen Lebenszeit unter einer Abhängigkeit geschieht mittels einer bereits implementierten Funktion zur Bestimmung des lexikographischen Maximums eines Ungleichungssystems.

Die Funktionen besitzen einen Parameter, um bei Bedarf statt der maximalen die *minimale* Lebenszeit zu bestimmen. Dies wird von der Heuristik zur Verbesserung des Schedule benötigt (siehe Abschnitt 5.3).

5.2.2 Indexmonotonie

Zur Überprüfung der Indexmonotonie nach Definition 3.1 wird zunächst für jedes Statement S der erweiterte Indexraum

$$D_S^+ = \{i \mid i \in \mathbb{Z}^{cs} \wedge (\exists i_0 \in D_S : \Theta_S i_0 \leq \Theta_S i)\}$$

benötigt. Bezeichne Θ_S^1 wie in Abschnitt 3.7 den erstmöglichen Ausführungszeitpunkt einer Instanz von Statements S, dann kann D_S^+ auch wie folgt ausgedrückt werden:

$$D_S^+ = \left\{i \mid i \in \mathbb{Z}^{cs} \wedge \Theta_S^1 \leq \Theta_S i\right\}$$

`getEarliestTime` berechnet Θ_S^1 und auf dieser Basis stellt `getDSPlus` das Ungleichungssystem für D_S^+ auf. Da der Schedule mehrdimensional sein kann und der Vergleichsoperator $\leq$ sich auf die lexikographische Ordnung bezieht, kann das Ungleichungssystem Oder-Verknüpfungen enthalten. Die Ungleichungssysteme der LooPo-Datenstruktur können nur Und-Verknüpfungen darstellen, daher gibt `getDSPlus` eine Liste von Ungleichungssystemen zurück. Werden diese Teilmengen Oder-verknüpft, erhält man das Ungleichungssystem für D_S^+. Aus dem gleichen Grund werden auch im Weiteren Listen von Ungleichungssystemen verwendet.

Arrayzugriffe sind indexmonoton steigend, wenn

$$\forall i_S \in D_S^+, i_T \in D_T^+, S, T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \Rightarrow \mathrm{Acc}_S(i_S) < \mathrm{Acc}_T(i_T)$$

gilt. Also sind Arrayzugriffe *nicht* indexmonoton steigend, wenn die Negation des Ausdrucks erfüllbar ist:

$$\exists i_S \in D_S^+, i_T \in D_T^+, S, T \in \mathcal{S}_Y : \Theta_S(i_S) < \Theta_T(i_T) \wedge \mathrm{Acc}_S(i_S) \geq \mathrm{Acc}_T(i_T).$$

`getSchedsIneqSyss` liefert für zwei Statements eine Liste von Ungleichungssystemen zurück, die zusammengenommen $\Theta_S(i_S) < \Theta_T(i_T)$ beschreiben, `getAccsIneqSyss` liefert die Liste von Ungleichungssystemen für $\mathrm{Acc}_S(i_S) \geq \mathrm{Acc}_T(i_T)$. Um zu überprüfen, ob die Arrayzugriffe zweier Statements indexmonoton steigend sind, müssen alle möglichen Kombinationen der Ungleichungssysteme für D_S^+, $\Theta_S(i_S) < \Theta_T(i_T)$ und $\mathrm{Acc}_S(i_S) \geq \mathrm{Acc}_T(i_T)$ auf Erfüllbarkeit getestet werden: ist eines erfüllbar, liegt *keine* steigende Indexmonotonie vor. Fallende Indexmonotonie lässt sich analog testen.

Starke Indexmonotonie ist nach Definition 3.17 nicht auf D_S^+ beschränkt, sonst aber exakt wie die „normale“ Indexmonotonie definiert. Um starke Indexmonotonie zu prüfen, muss also lediglich die Hinzunahme der Ungleichungssysteme für D_S^+ entfallen.

Für zwei Statements wird dies alles von `testQMSIM` geleistet. `testQMSIM` wird von `testIM` für jedes mögliche Statement-Paar aufgerufen, um ein Gesamtergebnis über alle Statements zu erhalten.

`checkIndexMonotonity` schließlich prüft mittels Aufrufen von `testIM`, ob ein Programm indexmonoton fallend, indexmonoton steigend, stark indexmonoton fallend, stark indexmonoton steigend oder nichts von alledem ist und liefert das Ergebnis zurück.

5.2.3 Bestimmung von $\hat{z}_Y(d_Y)$

Für die Berechnung von $\hat{z}_Y(d_Y)$ benötigt man nach Definition 3.12 $\left(\kappa_Y^{\neq 0} - \kappa_Y\right)$ und für alle auf Y schreibenden, nicht zu konstanter Zeit ausgeführten Anweisungen $\Delta\check{z}_S(d_Y)$.

$\kappa_Y^{\neq 0}$ ist nach Definition 3.9 der kleinste Arrayzellindex, den die nicht zu konstanter Zeit ausgeführten Anweisungen auf Y schreiben, κ_Y der kleinste beschriebene Zellindex aller auf Y schreibenden Anweisungen. `getSmallestWrittenCell` berechnet für jedes vorhandene Statement den kleinsten beschriebenen Zellindex, indem es Access-Funktion und Indexraumungleichungen zusammenführt und an die Funktion zur Bestimmung des lexikographischen Minimums übergibt. `getKappaDif` berechnet κ_Y als das Minimum von `getSmallestWrittenCell` über alle Statements, $\kappa_Y^{\neq 0}$

als das Minimum von `getSmallestWrittenCell` über alle nicht zu konstanter Zeit ausgeführten Statements und liefert die Differenz $\left(\kappa_Y^{\neq 0} - \kappa_Y\right)$ zurück.

$\Delta\check{z}_S(d_Y)$ ist nach Definition 3.10 ein Element der Menge $\psi_Y \mathcal{A}_S(\mathcal{I}_S(d_Y))$ mit $\mathcal{I}_S(d_Y)$ Menge der Indexschritte, die ein Statement S nach d_Y Zeiteinheiten ausführen kann. `getTimeDifISPC` bildet exakt nach Definition das Ungleichungssystem für $\mathcal{I}_S(d_Y)$, `getCellDifSPC` das Ungleichssystem für $\psi_Y \mathcal{A}_S(\mathcal{I}_S(d_Y))$. `getSmallElement` wählt aus diesem System ein Element, also $\Delta\check{z}_S(d_Y)$, aus, das im Betrag möglichst klein ist. Hier liegt ein Unterschied gegenüber Abschnitt 3.5: dort soll $\Delta\check{z}_S(d_Y)$ so gewählt werden, dass $|\hat{z}_Y(d_Y)|$ möglichst klein ist; das wäre jedoch einiges aufwändiger. Auf die hier angewandte Weise sind die gefundenen Allokationsvektoren daher unter Umständen etwas größer, als sie sein könnten. Größere Allokationsvektoren führen zwar dazu, dass etwas weniger Speicher gespart wird, aber die durch sie gegebene Speicherallokation ist nach Lemma 3.13, das eine untere Schranke für die Allokationsvektorgröße angibt, gültig.

`computeZDach` schließlich berechnet mittels der oben genannten Hilfsfunktionen $\hat{z}_Y(d_Y)$. Dabei werden die in Abschnitt 3.7 vorgestellten Sonderfälle erkannt und gesondert behandelt.

5.2.4 Berechnung der Speicherallokation

Für die Bestimmung der Speicherallokation wird die Allokationsvektormatrix $\mathcal{R}$ benötigt, die von `getAllocationVects` exakt nach Definition berechnet wird.

Die Berechnung von S_Y und Ω_Y durch `calculateSOmega` folgt exakt der Definition. Für die Berechnung der Matrizen S, U und V, für die $S = U\mathcal{R}V$ gilt mit S Diagonalmatrix, wurde mit `diagMatrix` ein Algorithmus von Bannerjee [6] implementiert.

Die Speicherallokationsfunktion Mem_Y ist lediglich die korrekte Zusammenstellung von Π_Y, S_Y und Ω_Y und wird von den Ausgabefunktionen erstellt (siehe nächster Abschnitt).

Π_Y wird von `calculatePi` nach einem selbst entwickelten Algorithmus berechnet: $\mathcal{R}$ hat nach Definition m Spalten und $n > m$ Zeilen und die Eigenschaft, dass die ersten $(m-1)$ Spalten die Einheitsmatrix sind. Die letzte Spalte hat $(m-1)$ führende Nullen:

$$\begin{array}{r} (m-1) \text{ Zeilen } \{ \\ (n-m+1) \text{ Zeilen } \{ \end{array} \begin{pmatrix} E & 0 \\ \underbrace{0}_{(m-1)} & \underbrace{*}_{1} \end{pmatrix} = \mathcal{R}$$

Es soll $\Pi_Y \mathcal{R} = 0$ gelten mit $\Pi_Y \in \mathbb{Z}^{(n-m \times n)}$. Da $\mathcal{R}$ vollen Spaltenrang hat, muss also für jede Spalte $\rho_i : \Pi_Y \rho_i = 0$ gelten. Außerdem sollen die Zeilen von Π_Y linear unabhängig sein.

Als Initialisierung generiere Π_Y als $(n - m \times n)$-Nullmatrix. Die ersten $(m - 1)$ Spalten werden 0 bleiben, da so die Forderung $\Pi_Y \rho_i = 0$ schon einmal für $1 \leq i \leq (m-1)$ erfüllt ist:

$$\begin{array}{r} (n-m) \text{ Zeilen } \{ \end{array} \begin{pmatrix} \underbrace{0}_{(m-1)} & \underbrace{*}_{(n-m+1)} \end{pmatrix} = \Pi_Y$$

Setze $p := 1$. p iteriert über die Zeilen von Π_Y. Jede Zeile multipliziert mit ρ_m muss 0 ergeben. Da die ersten $(m-1)$ Stellen jeder Zeile schon fest 0 gesetzt sind,

werden im Folgenden nur die letzten $(n - m + 1)$ Stellen verändert.

Solange $p \leq n - m$:

1. Setze $k := m + p - 1$.

   Das Element an Zeile p, Spalte k ist somit auf der Diagonalen der letzten $(n - m + 1)$ Spalten von Π_Y. Spalte k wird mit dem k-ten Element von ρ_m multipliziert, um das p-te Element von $(\Pi_Y \rho_m)$ zu berechnen.

2. Ist k-te Stelle von $\rho_m = 0$? Dann setze Element $\Pi_Y(p, k) = 1$. Alle anderen Elemente der Zeile bleiben 0, somit ist Stelle p des Vektors $(\Pi_Y \rho_m)$ gleich 0.

   Ansonsten setze $i := \rho_m(k)$. Suche $l > k$ mit $\rho_m(l) \neq 0$. Fallunterscheidung:

   (a) Wenn kein l gefunden, sind folglich alle $\rho_m(l) = 0$ mit $l > k$:

   $$\rho_m = \begin{pmatrix} * \\ 0 \end{pmatrix} \begin{matrix} \} \ k \text{ Zeilen} \\ \} \ (n-k) \text{ Zeilen} \end{matrix}$$

   Setze in den verbleibenden Zeilen z mit $p \leq z \leq (n - m)$ jeweils Spalte $m + z$ auf 1 und breche ab.
   Es ergibt sich dadurch für Π_Y:

   $$\begin{matrix} (p-1) \text{ Zeilen} \{ \\ (n-m-p+1) \text{ Zeilen} \{ \end{matrix} \begin{pmatrix} * & * \\ \underbrace{0}_{k} & \underbrace{E}_{(n-k)} \end{pmatrix} = \Pi_Y$$

   Somit

   $$\Pi_Y \rho_m = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{matrix} \} \ (p-1) \text{ Zeilen} \\ \} \ (n-m-p+1) \text{ Zeilen} \end{matrix}$$

   Die ersten $(p-1)$ Elemente sind aufgrund der vorherigen Iterationsschritte gleich Null.

   (b) Wenn l gefunden, sei $j := \rho_m(l)$. Setze in Π_Y, Zeile p, das Element in Spalte k auf $(j/\gcd(j,i))$, das Element in Spalte l auf $((-i)/\gcd(j,i))$. Wiederum ist Stelle p des Vektors $(\Pi_Y \rho_m)$ gleich 0, da er nach Definition der Matrixmultiplikation durch $(ij/\gcd(j,i) - ij/\gcd(j,i))$ gegeben ist.

3. p:=p+1.

Der Algorithmus setzt in jeder Zeile p das k-te Element auf einen Wert ungleich Null. In jeder späteren Zeile werden nur spätere Spalten verändert. Π_Y ist demnach in Zeilenstufenform, jede Zeile wie gefordert linear unabhängig.

5.2.5 Ausgabe

`QuilMem.hs` verfügt über zwei Arten, sein Berechnungsergebnis auszugeben:

`showAllocFunction` bzw. `showMemFunction` geben in ansprechender Formatierung aus, wie alle Zugriffe auf Array Y verändert werden können, um den Speicher besser zu nutzen.

`modifyPTST` modifiziert alle Zugriffe auf Y selbsttätig in einem gegebenen Syntaxbaum mit Symboltabelle. Es nutzt dazu die gegebenen Funktionen höherer Ordnung der Syntaxbaummodule bzw. der Symboltabellenmodule.

5.3 BetterSchedule.hs

Das Modul implementiert exakt den in Abschnitt 4.2.3 vorgestellten Algorithmus zur Verbesserung des Schedules. Die aufzurufende Funktion heißt `changeSched`.

Zur Überprüfung der Gültigkeit des Schedules wird nach jeder Änderung die Lebenszeitanalyse von Modul `QuilMem.hs` aufgerufen: es darf keine Lebenszeit einer Abhängigkeit kleiner als ein Zeitschritt sein kann. Daher muss nach jeder Änderung des Schedules in LooPo die Berechnung des Zielprogramms erneut aufgerufen werden, um die Ungleichungssysteme der Zielabhängigkeiten zu aktualisieren.

Nach Durchlauf der Heuristik wird der neue Schedule noch einmal mittels von `hsloopo` zur Verfügung gestellten Funktionen überprüft.

5.4 Benutzung

Die Benutzung der Implementation wird am Beispiel der Matrixmultiplikation vorgestellt; das Beispielprogramm ist unter `hsloopo/QuilMem/matmult-qm.src` zu finden:

```
CONSTANT n;

for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
      C[0,i,j]=0;
      for(k=0; k<n; k++) {
         C[k+1,i,j] = C[k,i,j] + A[i,k] * B[k,j];
}}}
```

Im Gegensatz zur in Abschnitt 3.8.1 gezeigten Version wurde hier die k-Dimension des Arrays C an die erste Position gestellt, um Indexmonotonie im Parallelen herzustellen.

Das Programm wird mit LooPo analysiert und ein Zielprogramm generiert. Mit dem Feautrier-Scheduler wird für das erste Statement ein Schedule $\Theta_1(i,j) = 0$, für das zweite Statement $\Theta_2(i,j,k) = k + 1$ generiert.

Nun kann man die Implementation des in dieser Arbeit vorgestellten Verfahrens anwenden. Die folgende Ein- und Ausgabe[1] wurde im Haskell-Interpreter ghci erzeugt:

```
Prelude> :l QuilMem/Main.hs
*Main> optMem "/tmp/loopo2458" "matmult-qmem.src.out"
0 C
1 A
2 B

Enter number of array to check: 0

Longest lifetime per target dependence:
TrDependenceID 1 (StatementID 1 --> StatementID 2): Optimum [1%1]
```

[1] um Statusausgaben und Debugmeldungen gekürzt

```
TrDependenceID 2 (StatementID 2 --> StatementID 2): Optimum [1%1]

Maximal lifetime: Vector 1


Checking index monotonity...
Index monotonity is Just (IMGrowing IMExtended)
Allocation matrix:
Matrix with 3 rows and 1 columns
 1
 0
 0

Every occurence of the array can be changed the following way:
"C[a,b,c] -> C[b,c]"


New target code:
for (t1=min(0, 1); t1<=max(0, n); t1+=1)
  parallel for (p1=min(0, 0); p1<=max(n-1, n-1); p1+=1)
    parallel for (p2=min(0, 0); p2<=max(n-1, n-1); p2+=1) {
      if (0 <= t1 && t1 <= 0 && 1 && 0 <= p1 && p1 <= n-1 && 1 &&
          0 <= p2 && p2 <= n-1 && 1)
        C[p1, p2] = 0;
      if (1 <= t1 && t1 <= n && 1 && 0 <= p1 && p1 <= n-1 && 1 &&
          0 <= p2 && p2 <= n-1 && 1)
        C[p1, p2] = C[p1, p2]+A[p1, t1-1]*B[t1-1, p2];
      }
```

Weitere verfügbare Beispielprogramme unter `loopo/lib/examples` sind die Berechnung der Fibonacci-Zahlen `fib.src` sowie die Berechnung des Pascalschen Dreiecks `pascal.src`.

Kapitel 6

Fazit

Kapitel 2 hat ein optimales Verfahren zur Optimierung der Speichernutzung von Arrays funktionaler Programme im Polytopmodell vorgestellt. Dabei wurden sogenannte *Pseudoprojektionen* eingeführt, die Dimensionen nur teilweise ausprojizieren. Das Originalverfahren von Fabien Quilleré und Sanjay Rajopadhye wurde nur für sequentielle Programme definiert, die Beweise wurden für Parallelprogramme erweitert.

In Kapitel 3 wurde die Erweiterung des Verfahrens auf imperative, mit Hilfe des Polytopmodells automatisch parallelisierte Programme gezeigt. Es wurde ein Verfahren entwickelt, das die Speichernutzung von Arrays in imperativen Parallelprogrammen verringert, auf die ausschließlich *streng indexmonoton* geschrieben wird. Diese Speichernutzung ist zwar niemals kleiner, als wenn bei der Programmerstellung von vornherein eine möglichst gute Nutzungsrate der Arrayzellen durch viele Mehrfachzuweisungen implementiert wurde; doch ein von vornherein manuell auf geringen Speicherverbrauch optimiertes Programm erschwert die Datenabhängigkeitsanalyse und es entstehen gegebenenfalls Output- sowie Anti-Abhängigkeiten, die die mögliche Parallelität verringern können. Das gezeigte Speicherallokationsverfahren wird erst nach der Parallelisierung angewandt, daher wird keinerlei Parallelität zerstört. Weiterhin können bei manueller Speicheroptimierung Fehler gemacht werden, während das vorliegende Verfahren *garantiert*, dass kein Wert überschrieben wird, der noch benötigt wird.

Das Verfahren arbeitet in der Praxis erfolgreich auf Programmen in *Single Assignment*-Form. Problematisch erweist sich jedoch die Erfüllung der Voraussetzung Indexmonotonie: hier können automatische Transformationen des Zielprogramms, wie in Abschnitt 3.8 vorgeschlagen, die Situation noch verbessern.

Auch verhindert der sogenannte Free Schedule in vielen Fällen die Optimierung der Speichernutzung, ohne die Programmausführungszeit damit zu verkürzen. Es wurde daher eine Heuristik entwickelt und in Abschnitt 4.2 vorgestellt, die einen Free Schedule, der durch Initialisierungsanweisungen unbegrenzte Lebenszeiten hat, in einen Schedule umwandelt, der Speicheroptimierung zulässt und dessen Programmlaufzeit nur um eine additive Konstante größer ist.

Die Heuristik sowie das Speicherallokationsverfahren wurden implementiert und stehen im automatischen Schleifenparallelisierer LooPo zur Verfügung.

Literaturverzeichnis

[1] Fabien Quilleré, Sanjay Rajopadhye: Optimizing Memory Usage in the Polyhedral Model, ACM Transations on Programming Languages and Systems, Vol. 22, No. 5, September 2000

[2] Martin Griebl: Automatic Parallelization of Loop Programs for Distributed Memory Architectures, Juni 2004

[3] Christian Lengauer: Loop parallelization in the polytope model, Eike Best, editor, CONCUR'93, LNCS 715, Springer-Verlag, 1993.

[4] H. Le Verge, C. Mauras, P. Quinton: The Alpha language and its use for design of systolic arrays, Journal of VLSI signal processing 3, September 1991.

[5] Martin Griebl, Christian Lengauer: The loop parallelizer LooPo: Announcement, Languages and Compilers for Parallel Computing, LNCS 1239, Springer-Verlag, 1997. Neuere Informationen auch im Web unter http://www.infosun.fmi.uni-passau.de/cl/loopo.

[6] Utpal Banerjee: Loop Transformations for Restructuring Compilers: The Foundations, Kluwer 1993.

[7] P. Feautrier: Dataflow analysis of array and scalar references, Int. J. Parallel Programming, 20(1):23–53, February 1991.

[8] William Thies, Frederic Vivien, Jeffrey Sheldon, Saman Amarasinghe: A unified framework for schedule and storage optimization, Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation, 2001

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet habe, sowie dass diese Diplomarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Passau, den 28. März 2007

(Benjamin Schulte)