Zur Generierung von effizient ausführbarem Code aus SAC-spezifischen Schleifenkonstrukten

— korrigierte Fassung —

Diplomarbeit
von
Dietmar Kreye

am Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität zu Kiel

Kiel
1. August 1998

Gutachter:	Prof. Dr. W. Kluge
Betreuer:	Dr. S.-B. Scholz, Dipl.-Inf. C. Grelck
Tag der Ausgabe:	3. November 1997
Tag der Abgabe:	1. August 1998

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Ein wichtiges Einsatzgebiet für Rechner ist die Implementierung und Berechnung von numerischen Algorithmen. Für eine Programmiersprache, die zur Lösung numerischer Probleme verwendet werden soll, sind vor allem zwei Eigenschaften wichtig: Zum einen muß sie Datenstrukturen für mehrdimensionale Arrays und Konstrukte zur Beschreibung von Operationen auf diesen bereitstellen, wobei sich der Programmierer ein möglichst hohes Abstraktionsniveau wünscht, um den Aufwand für die Programmentwicklung zu minimieren. Zum anderen muß der erzeugte Code sehr effizient bezüglich Laufzeit und Speicherbedarf sein, da Problemstellungen der Numerik im allgemeinen sehr komplexe Berechnungen erfordern und auf großen Datenmengen operieren.

SAC (Single Assignment C) ist eine funktionale Sprache, die unter besonderer Berücksichtung der Bedürfnisse numerischer Anwendungen entworfen wurde [Scho96, Grel96]. SAC stellt ein universelles Array-Konzept zur Verfügung, das mehrdimensionale Arrays und APL-ähnliche [Iver62] primitive Operationen umfaßt. Zur selektiven elementweisen Behandlung von Arrays existiert ferner das sogenannte `with`-Konstrukt, das mit den Array-Comprehensions in CLEAN [PE97] und HASKELL [HP97] oder den `for`-Loops in SISAL [Cann93] vergleichbar ist. Eine wesentliche Eigenschaft des Array-Konzeptes von SAC ist die Möglichkeit, Array-Operationen dimensionsunabhängig zu spezifizieren. Außerdem abstrahiert SAC vollständig von der konkreten Ablage der Arrays im Speicher des Rechners, sodaß Speicherverwaltung auf Hochsprachenebene überflüssig wird. Da SAC eine funktionale Sprache ist, sind alle SAC-Programme seiteneffektfrei und erfüllen die Church-Rosser-Eigenschaft, d. h. die Bedeutung des Programmes ist unabhängig von der Berechnungsreihenfolge seiner Teilausdrücke.

Im Rahmen der Diplomarbeiten von H. Wolf [Wolf95], A. Sievers [Siev95] und C. Grelck [Grel96] wurde ein Compiler entwickelt, der SAC-Programme in semantisch äquivalente C-Programme [KR88] übersetzt (SAC2C).[1] Dank der Integration umfangreicher Optimierungen, die auf der Seiteneffektfreiheit von SAC beruhen und nicht vom C-Compiler durchgeführt werden können, erreichen SAC-Programme Laufzeiten, die mit denen entsprechender SISAL-Programme vergleichbar sind (siehe [Scho96], Kapitel 5). Insbesondere die momentane Compilation von Array-Operationen bietet aber Ansatzpunkte für Verbesserungen.

[1] Homepage von SAC und SAC2C im WWW unter `http://www.informatik.uni-kiel.de/~sacbase`

Eine komplexe benutzerdefinierte Array-Operation in SAC besteht aus einer Komposition elementarer Operationen. Der SAC-Compiler transformiert diese durch Einführung temporärer Arrays zur Speicherung von Zwischenergebnissen in eine Folge atomarer Array-Operationen, um anschließend jede dieser Elementaroperationen separat zu übersetzen. Diese Vorgehensweise hat jedoch negative Auswirkungen auf die Laufzeit des Compilats, da die dynamische Verwaltung der temporären Arrays einen erheblichen Aufwand bedeutet und möglicherweise redundante Zwischenresultate explizit berechnet werden. Zur Lösung dieses Problems wird in den SAC-Compiler eine neue Hochsprachen-Optimierung integriert — das sogenannte `with`-Loop-Folding (siehe [Scho98] und vor allem [Schw98]). `With`-Loop-Folding sorgt dafür, daß aufeinanderfolgende `with`-Konstrukte in ein einziges `with`-Konstrukt überführt werden, wodurch temporäre Arrays und redundante Berechnungen vermieden werden. Da sich alle Array-Primitiva in SAC alternativ mit Hilfe des `with`-Konstruktes spezifizieren lassen, ist diese Optimierung auf alle Array-Operationen anwendbar. Allerdings ist die Syntax des `with`-Konstruktes bezüglich des `with`-Loop-Folding nicht abgeschlossen. Daher kann ein aus `with`-Loop-Folding hervorgegangenes `with`-Konstrukt im allgemeinen nicht mit dem derzeitigen Compilations-Schema übersetzt werden.

Da das `with`-Konstrukt in der Regel auf großen Datenmengen operiert, hat außerdem die Trefferrate im Cache starken Einfluß auf die Laufzeit. Eine Ablaufanalyse von Numerik-Benchmarks auf typischen Einprozessorarchitekturen hat zum Beispiel ergeben, daß bis zu 50 % der Gesamtlaufzeit durch Speicherzugriffe verursacht werden [MLG92]. Dieses Ergebnis zeigt sehr deutlich, wie wichtig die Optimierung der Cache-Performance auf diesen Maschinen ist. Erfahrungen mit anderen Compilern zeigen, daß sich die Anzahl der Cache-Fehlzugriffe — auch unabhängig von der konkreten Ausprägung des Caches (Assoziativität, Größe des Caches, usw.) — durch ein geeignetes Datenlayout und günstige Zugriffspfade deutlich verringern läßt [MA95, AAL95, LRW91, Lam94, WL91]. Wünschenswert ist ein hoher Grad an zeitlicher und örtlicher Datenlokalität, damit Daten, die einmal in das Cache geladen wurden, möglichst häufig wiederverwendet werden können. Um dieses Ziel für das Compilat eines `with`-Konstruktes zu erreichen, muß unter Umständen durch Code-Transformationen die Ausführungsreihenfolge verändert werden. Das ist möglich, da die Elemente eines `with`-Konstruktes konzeptionell nebenläufig berechnet werden. Leider kann dies ein C-Compiler häufig aber nicht inferieren, da er auf Grund der Verwendung von Zeigern mit Aliasing und eventuellen Seiteneffekten rechnen muß. Daher kann man sich nicht darauf verlassen, daß diese Optimierungen vom C-Compiler durchgeführt werden. Vielmehr müssen sie explizit in das Compilations-Schema des SAC-Compilers aufgenommen werden.

Ziel der vorliegenden Arbeit ist es, ein neues Compilations-Schema für das `with`-Konstrukt zu entwickeln, das deren Übersetzung in effizient ausführbaren Code ermöglicht, indem es zum einen mit `with`-Loop-Folding zusammenarbeitet und zum anderen die Cache-Performance optimiert. Ferner soll das Compilations-Schema modular aufgebaut und damit leicht erweiterbar sein.

Die Arbeit gliedert sich wie folgt: Kapitel 2 vermittelt die nötigen Vorkenntnisse über das Array-Konzept von SAC. Insbesondere wird dort das `with`-Konstrukt ausführlich vorgestellt. In Kapitel 3 werden einige wichtige Aspekte diskutiert, die die Compilation von Array-Operationen im allgemeinen betreffen. Vor diesem Hintergrund werden im Kapitel 4 die Me-

thoden entwickelt, die die Compilation des `with`-Konstruktes in effizient ausführbaren Code ermöglichen. Kapitel 5 stellt anschließend die Implementation im Detail vor. Es wird eine Zwischendarstellung für das `with`-Konstrukt eingeführt, auf deren Basis dann das Compilations-Schema definiert wird. Um die Leistungsfähigkeit des vorgestellten Compilations-Schemas zu demonstrieren, werden in Kapitel 6 Laufzeitmessungen für ein Anwendungsbeispiel präsentiert. Die Arbeit schließt mit einer Zusammenfassung in Kapitel 7.

Kapitel 2

Arrays in SAC

Dieses Kapitel umreißt die Aspekte der Sprache SAC und des SAC-Compilers, welche Arrays betreffen und daher für diese Arbeit unmittelbar von Bedeutung sind. Die komplette Sprachdefinition von SAC befindet sich in [Scho96] (allgemein) und [Grel96] (Module, Klassen, I/O), für eine ausführliche Darstellung des SAC-Compilers siehe auch [Wolf95, Siev95].

2.1 Grundlagen

Das Array-Konzept von SAC basiert auf einem Array-Kalkül, dem sogenannten ψ-Kalkül [Mull88]. Ein Array wird dabei durch zwei Vektoren repräsentiert: durch einen Datenvektor, der alle Elemente des Arrays enthält, und einen Formvektor (auch Shape genannt), der die Struktur des Arrays beschreibt. Die Länge des Formvektors entspricht gerade der Dimension des Arrays; die einzelnen Komponenten des Formvektors geben die Zahl der Elemente in der entsprechenden Dimension an. Beispiele dazu sind in Abbildung 2.1 gegeben.

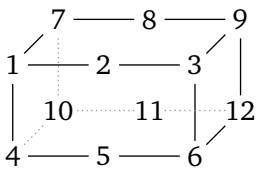
$$A_1 = \begin{pmatrix} 7 \\ 8 \\ 9 \end{pmatrix} = [7,8,9]$$

Dimension: 1
Formvektor: $[3]$
Datenvektor: $[7,8,9]$

$$A_2 = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

Dimension: 2
Formvektor: $[2,3]$
Datenvektor: $[7,8,9,10,11,12]$

$A_3 =$



Dimension: 3
Formvektor: $[2,2,3]$
Datenvektor: $[1,2,3,4,5,6,7,8,9,10,11,12]$

Abbildung 2.1: Beispiele für die Array-Darstellung in SAC.

2.2 Primitive Array-Operationen

SAC stellt zahlreiche dimensionsunabhängige primitive Operationen auf Arrays zur Verfügung:

- Arithmetische Operationen:

  Die Operationen `+`, `-`, `*`, `/`, `%` sind nicht nur über Skalaren definiert, sondern ermöglichen auch die Verknüpfung zweier Arrays mit identischen Formvektoren bzw. eines Skalars mit einem Array.

- `dim(` A`)`:

  liefert die Dimension des Arrays A;

- `shape(` A`)`:

  liefert den Formvektor des Arrays A;

- `reshape(` sh`,` A`)`:

  Falls das Produkt der Komponenten von sh gleich der Anzahl der Elemente von A ist, liefert diese Funktion ein Array mit dem Datenvektor von A und dem Formvektor sh. Ansonsten ist das Ergebnis undefiniert.

- `psi(` iv`,` A`)`:

  Diese Funktion selektiert ein Element oder ein Teil-Array des Arrays A.

  Die Selektion aus einem Array erfolgt mittels eines sogenannten Indexvektors (Vektor $\equiv$ eindimensionales Array). Sei n die Dimension, $sh = (sh_0, \ldots, sh_{n-1})$ der Formvektor und $d = (d_0, \ldots, d_{m-1})$ mit $m = \prod_{j=0}^{n-1} sh_j$ der Datenvektor des Arrays A. Dann ist die Menge der vollständigen legalen Indexvektoren bezüglich sh definiert durch:

$$\begin{aligned} I(sh) &:= \{i = (i_0, \ldots, i_{n-1}) \in {\mathbb{N}_0}^n \mid \forall j \in \{0, \ldots, n-1\} : 0 \leq i_j < sh_j\} \\ &= \{i \in {\mathbb{N}_0}^n \mid 0 \leq i < sh\} \quad [1] \quad . \end{aligned}$$

  Dies läßt sich alternativ als Intervall formulieren:

$$I(sh) \;=\; [\,0;\; sh\,) \quad .$$

  Ein vollständiger legaler Indexvektor i korrespondiert mit dem Element d_l des Datenvektors, wobei gilt:

$$l \;=\; \sum_{j=0}^{n-1} \left(i_j \cdot \prod_{k=j+1}^{n-1} sh_k \right) \quad .$$

[1] Sei $n \in \mathbb{N}$ fest. Dann läßt sich die Relation $<$ (und damit auch $\leq$) über $\mathbb{N}_0 \times \mathbb{N}_0$ auf natürliche Weise auf ${\mathbb{N}_0}^n \times {\mathbb{N}_0}^n$ übertragen:

$$(a_0, \ldots, a_{n-1}) < (b_0, \ldots, b_{n-1}) \quad :\Longleftrightarrow \quad \forall j \in \{0, \ldots, n-1\} : a_j < b_j \quad .$$

Man beachte, daß $<$ mit $n \geq 2$ *keine* Ordnungsrelation ist, da Trichotomie nicht gegeben ist:

$$(1, 3, \ldots) \neq (4, 2, \ldots) \quad \wedge \quad (1, 3, \ldots) \not< (4, 2, \ldots) \quad \wedge \quad (1, 3, \ldots) \not> (4, 2, \ldots) \quad .$$

Falls also iv ein vollständiger legaler Indexvektor von A ist, liefert `psi` das mit iv korrespondierende Element von A. Falls iv ein um hintere Komponenten verkürzter legaler Indexvektor von A ist, erhält man als Ergebnis das entsprechende Teil-Array von A — mit allen Elementen deren erste Indexkomponenten mit iv übereinstimmen. In allen anderen Fällen ist das Ergebnis undefiniert. Beispiele:

$$\texttt{psi([1,1],} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}\texttt{)} \;=\; 5 \quad ,$$

$$\texttt{psi([1],} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}\texttt{)} \;=\; \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \quad .$$

Alternativ ist für `psi` auch die Schreibweise A`[`iv`]` möglich.

- `genarray(` sh`,` $expr$`)`:

  Es wird ein Array der Form sh erzeugt, wobei jede Indexposition des Arrays mit $expr$ belegt wird. Falls $expr$ ein Skalar ist, wird also ein Array mit dem Formvektor sh erzeugt, dessen Elemente alle den Wert $expr$ haben. Falls $expr$ selbst ein Array mit dem Formvektor sh' ist, erweitert sich der Formvektor des Resultates zu $(sh_0, sh_1, \ldots, sh'_0, sh'_1, \ldots)$. Dazu ein Beispiel:

  $$\texttt{genarray([2], [1,2,3])} \;=\; \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \quad .$$

- `take(` v`,` A`)`:

  Die Funktion `take` liefert ein Teilarray von A, das bezüglich jeder Dimension soviel Elemente hat, wie der Vektor v vorgibt. Dabei werden jeweils die ersten Elemente jeder Dimension selektiert. Hat der Selektionsvektor v weniger Komponenten als die Dimensionalität von A, so werden in den übrigen Dimensionen alle Elemente selektiert. So gilt etwa:

  $$\texttt{take([2,2],} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}\texttt{)} \;=\; \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix} \quad ,$$

  $$\texttt{take([2],} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}\texttt{)} \;=\; \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad .$$

- `drop(` v`,` A`)`:

  `drop` arbeitet komplementär zur Funktion `take`. Anstatt die durch v spezifizierten Elemente aus A zu selektieren, entfernt `drop` entsprechend viele. Fehlende Komponenten in v werden dabei als 0 angenommen. Beispiele:

  $$\texttt{drop([1,1],} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}\texttt{)} \;=\; \begin{pmatrix} 5 & 6 \\ 8 & 9 \end{pmatrix} \quad ,$$

WithExpr ⇒ `with (` *Generator* `) [` *AssignBlock* `]` *ConExpr*

Generator ⇒ *GenCube* `[ step` *Expr* `[ width` *Expr* `] ]`

GenCube ⇒ *ExprOrDot GenRel GenIdx GenRel ExprOrDot*

ExprOrDot ⇒ *Expr* | `.`

GenRel ⇒ `<=` | `<`

GenIdx ⇒ *Id*
| `[` *Id* `[ ,` *Id* `]* ]`
| *Id* `= [` *Id* `[ ,` *Id* `]* ]`

ConExpr ⇒ `genarray (` *ConstVec* `,` *Expr* `)`
| `modarray (` *Expr* `,` *Id* `,` *Expr* `)`
| `fold (` *FoldFun* `,` *Expr* `,` *Expr* `)`

FoldFun ⇒ *FoldPrf*
| `[` *Id* `: ]` *Id*

FoldPrf ⇒ `+` | `*`

Abbildung 2.2: Syntax des `with`*-Konstruktes.*

$$\texttt{drop([1],} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}\texttt{)} = \begin{pmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} .$$

- `cat(` d`,` A`,` B`)`:

  Diese Funktion konkateniert die Arrays A und B in der Dimension d. Das Ergebnis ist genau dann definiert, wenn $\dim(A) = \dim(B)$ gilt, d die Eigenschaft $0 \leq d < \dim(A)$ besitzt, und die Formvektoren von A und B sich höchstens in der d-ten Komponente unterscheiden.

- `modarray(` A`,` iv`,` $expr$`)`:

  liefert das an der Stelle iv mit dem Wert $expr$ modifizierte Array A;

- `rotate(` d`,` num`,` A`)`:

  liefert das in Dimension d um num Elemente rotierte Array A.

2.3 Das `with`-Konstrukt

Mit dem `with`-Konstrukt existiert in SAC ein sehr flexibles Sprachelement zur Manipulation von Arrays. Die Syntax des `with`-Konstruktes ist in Abbildung 2.2 in Backus-Naur-Form (BNF) spezifiziert.

Das `with`-Konstrukt besteht im wesentlichen aus zwei Teilen: einem Generator und einem Operator. Der Generator spezifiziert durch Angabe einer unteren und oberen Grenze sowie

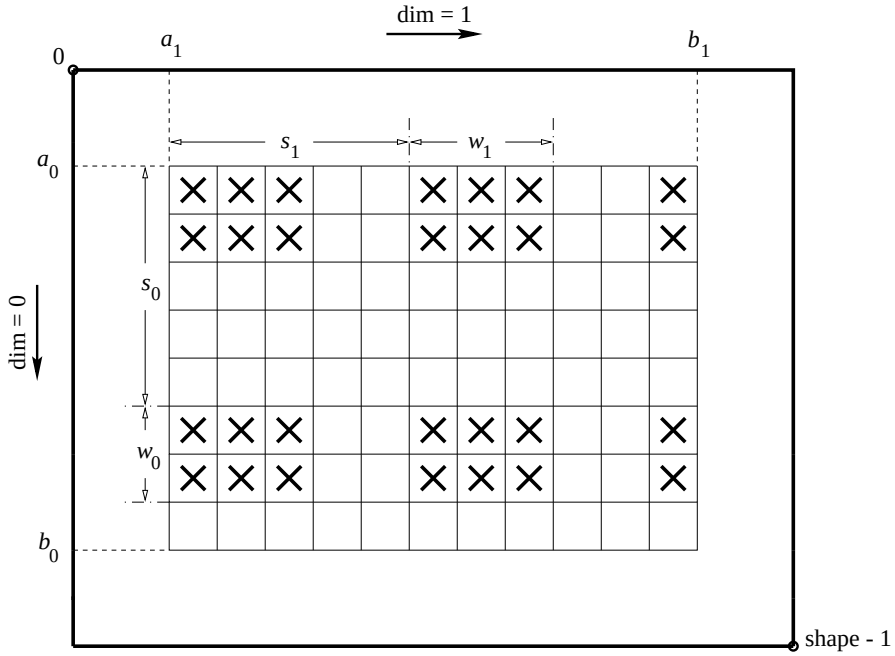


Abbildung 2.3: Layout einer zweidimensionalen Generatormenge.

einer Rasterung eine Menge von Indexvektoren. Desweiteren bestimmt er eine Generatorvariable, mit deren Hilfe die Elemente dieser Menge innerhalb des Operators referenziert werden können. Diese Generatorvariable kann bei Bedarf explizit in Skalare aufgeteilt werden. Beispielsweise bezeichnet

```
([0,0] <= iv=[iv0,iv1] < [10,10] step [2,2] width [1,1])
```

die Indexvektormenge, die alle zweistelligen Vektoren enthält, deren Komponenten kleiner als 10 und geradzahlig sind. Als Generatorvariable wird `iv` verwendet, auf deren Komponenten direkt mit Hilfe der Bezeichner `iv0` und `iv1` zugegriffen werden kann.

Allgemein hat der Generator die Form

$$\texttt{(}a \prec iv \lhd b \texttt{ step } s \texttt{ width } w\texttt{)} \quad ,$$

wobei $\prec, \lhd \in \{\leq, <\}$ gilt. Die durch ihn spezifizierte Indexvektormenge $I(a, b, s, w, \prec, \lhd)$ ist genau dann definiert, wenn a, b, s, w Vektoren gleichen Shapes sind. Es gilt dann:

$$\begin{aligned} I(a, b, s, w, \prec, \lhd) &:= \{i \in \mathbb{N}_0{}^* \mid \forall j : (a_j \prec i_j \lhd b_j) \wedge ((i_j - a_j) \bmod s_j < w_j)\} \\ &= \{i \in \mathbb{N}_0{}^* \mid (a \prec i \lhd b) \wedge ((i - a) \bmod s < w)\} \quad . \end{aligned}$$

$I(a, b, s, w, \prec, \lhd)$ wird Generatormenge genannt. Anstelle von $I(a, b, s, w, \leq, <)$ wird nachfolgend kurz $I(a, b, s, w)$ geschrieben. Abbildung 2.3 verdeutlicht das Layout einer Generatormenge für ein zweidimensionales Array.

Um eine prägnantere Generator-Spezifikation zu ermöglichen, existieren einige abkürzende Schreibweisen. So können die Generatorgrenzen a, b alternativ als Punkt (`.`) angegeben

werden, wenn der kleinste bzw. größte zulässige Indexvektor verwendet werden soll. Welcher konkrete Wert durch den Punkt repräsentiert wird, richtet sich nach dem verwendeten Operator (siehe nächsten Absatz). Ferner sind die Rasterungsparameter s und w optional. Falls sie nicht angegeben sind, werden sie standardmäßig auf 1 (als Vektor aufgefaßt) gesetzt.

Der Operator legt die Bedeutung eines `with`-Konstruktes in Abhängigkeit von der Generatormenge fest. Für den Operator stehen drei Varianten zur Auswahl:

- `genarray(` sh`,` $expr$`)`:

  Ein `with`-Konstrukt mit diesem Operator liefert genau dann ein definiertes Resultat, wenn die Generatormenge G eine Teilmenge von $I(sh)$ ist. Es wird dann ein Array der Form sh erzeugt, wobei jede Indexposition aus G dieses Arrays mit $expr$ belegt wird, und alle anderen mit einem Default-Wert — bei Arrays über Zahlen ist es die 0 — initialisiert werden. In dem Zielausdruck $expr$ kann bei Bedarf die Generatorvariable referenziert werden, sodaß der Wert von $expr$ im allgemeinen von der Indexposition abhängt. Der optionale Zuweisungsblock vor dem Operator enthält Abstraktionen, die in $expr$ verwendet werden — dient also lediglich der übersichtlichen Notation von komplizierten Zielausdrücken. Nachfolgend ein Beispiel für ein `with`-Konstrukt mit `genarray`-Operator:

  ```
  with ([1,1] <= iv < [4,4]) {
    val = 10*iv[0] + iv[1];
  }
  genarray( [5,5], val)
  ```

  $$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 11 & 12 & 13 & 0 \\ 0 & 21 & 22 & 23 & 0 \\ 0 & 31 & 32 & 33 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} .$$

  Alternativ kann im Generator die Punkt-Schreibweise verwenden werden. Im Zusammenhang mit dem `genarray`-Operator steht ein Punkt als untere Grenze für $(0 \cdot sh)$ und als obere Grenze für $(sh - 1)$. Also läßt sich obiges `with`-Konstrukt auch wie folgt formulieren:

  ```
  with (. < iv < .) {
    val = 10*iv[0] + iv[1];
  }
  genarray( [5,5], val)     .
  ```

  Außerdem können die `psi`-Operationen in der Spezifikation von `val` entfallen, falls `iv` im Generator explizit in Skalare aufgeteilt wird:

  ```
  with (. < [iv0,iv1] < .) {
    val = 10*iv0 + iv1;
  }
  genarray( [5,5], val)   .
  ```

  Analog zur primitiven Funktion `genarray` (siehe Abschnitt 2.2) kann $expr$ nicht nur ein Skalar, sondern auch ein Array sein, wodurch sich der Formvektor des Resultates entsprechend erweitert:

```
with ([1] <= iv < [4]) {
  val = [iv[0], 4, 5];
}
genarray( [5], val)
```
$$= \begin{pmatrix} 0 & 0 & 0 \\ 1 & 4 & 5 \\ 2 & 4 & 5 \\ 3 & 4 & 5 \\ 0 & 0 & 0 \end{pmatrix} \quad .$$

- `modarray(` A`,` iv`,` $expr$`)`:

  Dieser Operator erzeugt ein Array, daß den gleichen Formvektor wie A aufweist ($sh_A := \mathrm{shape}(A)$). Dazu muß die Generatormenge G eine Teilmenge von $I(sh_A)$ sein. Alle Indexpositionen $iv \in G$ des zu erzeugenden Arrays werden mit $expr$ belegt, die übrigen mit $A[iv]$. Zum Beispiel gilt:

```
A = genarray( [5,6], 1);
B = with ([0,0] <= iv < [5,6]
          step [3,2] width [2,1])
    { val = 2; }
    modarray( A, iv, val);
```
$$\longrightarrow \quad \mathtt{B} = \begin{pmatrix} 2 & 1 & 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 & 2 & 1 \end{pmatrix} \quad .$$

  Als Generatorgrenzen verwendete Punkte symbolisieren in Verbindung mit dem `modarray`-Operator die Werte $(0 \cdot sh_A)$ bzw. $(sh_A - 1)$. Das heißt, das obige Beispiel kann auch als

```
...
B = with (. <= iv <= . step [3,2] width [2,1])
    modarray( A, iv, 2);
```

  geschrieben werden.

- `fold(` f`,` $neutral$`,` $expr$`)`:

  Der `fold`-Operator berechnet für jeden Indexvektor aus der Generatormenge den Zielausdruck $expr$ und faltet die erhaltenen Werte — sie seien $\{e_1, e_2, \ldots, e_N\}$ genannt — mit Hilfe der Funktion f und des neutralen Elementes $neutral$ (ist für primitive Funktionen optional) zusammen, sodaß am Ende ein einziger Wert entsteht:

$$(((neutral\ f\ e_1)\ f\ e_2)\ \ldots\ f\ e_N) \quad .$$

  Dazu muß f zweistellig sein, und der Typ des Funktionswertes muß den Typen der Argumente entsprechen. Da keine Faltungsreihenfolge vorgegeben ist, muß f außerdem assoziativ und kommutativ sein.

  Man beachte, daß bei Verwendung des `fold`-Operators der für die Generatormenge zulässige Wertebereich im allgemeinen nicht beschränkt ist. Es ist in diesem Fall also sinnlos und daher auch verboten, bei den Generatorgrenzen Punkte anzugeben.

  Das `with`-Konstrukt mit `fold`-Operator läßt sich zum Beispiel dazu verwenden, alle an geradzahligen Indexpositionen stehenden Elemente eines Vektors aufzuaddieren:

```
A = [1,2,3,4,5,6,7,8,9,10];
a = with ([0] <= iv < [10] step [2])
    fold( +, A[iv]);
```
$$\longrightarrow \quad \mathtt{a} = 30 \quad .$$

Das folgende Programmfragment multipiziert alle zweistelligen Vektoren über $\{1, 2, 3\}$ miteinander:

```
with ([1,1] <= iv <= [3,3])
fold( *, iv);
```
$$= \begin{pmatrix} 216 \\ 216 \end{pmatrix} \quad .$$

2.4 Interne Darstellung des `with`-Konstruktes

Wie bereits in der Einleitung erwähnt, faltet der SAC-Compiler eine Abfolge von `with`-Konstrukten nach Möglichkeit zusammen (`with`-Loop-Folding). Die prinzipielle Vorgehensweise soll hier nur an Hand eines Beispiels verdeutlicht werden — eine ausführliche Beschreibung dieser Optimierung befindet sich in [Scho98] und vor allem in [Schw98]. Im folgenden Programmfragment werden zwei Arrays `A` und `B` erzeugt:

```
A = with ([0,0] <= iv < [300,100])
    genarray( [300,300], 1);

B = with ([0,100] <= iv < [300,300])
    modarray( A, iv, 2);                .
```

Da das Array `A` nur ein Zwischenergebnis für die Berechnung von `B` darstellt, werden die beiden `with`-Konstrukte vom Compiler intern zu einem einzigen vereinigt. Die externe Syntax des `with`-Konstruktes ist bezüglich einer solchen Faltung jedoch nicht abgeschlossen. Soll das Array `B` aus dem Beispiel mit nur einem `with`-Konstrukt beschrieben werden, muß man sich etwa eines Conditionals bedienen, um zwischen den Elementen, die mit 1 bzw. 2 initialisiert werden, unterscheiden zu können:

```
B = with (. <= iv <= .) {
      if (iv[1] < 100)
        val = 1;
      else
        val = 2;
    }
    genarray( [300,300], val);    .
```

Diese Vorgehensweise ist jedoch sehr unbefriedigend, da der Zielausdruck hier Aufgaben übernehmen muß, die eigentlich Sache des Generators sein sollten. Die Lösung besteht darin, die Darstellung des `with`-Konstruktes intern gegenüber der externen Syntax dahingehend zu erweitern, daß multiple Generatoren mit jeweils eigenem Zielausdruck nebst optionalem Anweisungsblock möglich sind. Das obige Beispiel stellt sich dann nach der Faltung intern wie folgt dar:

```
B = with ([0,  0] <= iv < [300,100]) : 1
         ([0,100] <= iv < [300,300]) : 2
    genarray( [300,300]);                  .
```

In dieser Notation ist der Zielausdruck nun kein Argument des Operators mehr, sondern steht, durch einen Doppelpunkt (`:`) getrennt, hinter dem zugehörigen Generator.

Für jeden Indexvektor der einzelnen Generatormengen muß es genau einen Zielausdruck geben, also müssen die Generatormengen paarweise disjunkt sein. Im Falle eines `genarray` bzw. `modarray` bilden sie eine Partition der Menge vollständiger legaler Indizes des betreffenden Arrays. Das bedeutet, daß in der internen Darstellung das Komplement einer Generatormenge explizit gemacht wird. Es wird zum Beispiel

```
B = with ([0,0] <= iv < [300,100])
    genarray( [300,300], 1);
```

zu

```
B = with ([0,  0] <= iv < [300,100]) : 1
         ([0,100] <= iv < [300,300]) : 0
    genarray( [300,300]);
```

und

```
B = with ([0,0] <= iv < [300,100])
    modarray( A, iv, 1);
```

zu

```
B = with ([0,  0] <= iv < [300,100]) : 1
         ([0,100] <= iv < [300,300]) : A[iv]
    modarray( A);
```

umgewandelt.

Um die einzelnen Generatoren bei der Code-Erzeugung besser handhaben zu können, werden diese in der internen Darstellung dahingehend normiert, daß eventuell auftretende Punkte (`.`) durch konkrete Werte ersetzt werden, fehlende `step`- oder `width`-Angaben ergänzt werden, bei den unteren Grenzen nur die Relation $\leq$ und bei den oberen Grenzen nur $<$ zum Einsatz kommt:

```
A = with ([99,99] < iv <= .)
    genarray( [300,300], 1);
```

wird in die Darstellung

```
A = with ([  0,  0] <= iv < [100,300] step [1,1] width [1,1]) : 0
         ([100,  0] <= iv < [300,100] step [1,1] width [1,1]) : 0
         ([100,100] <= iv < [300,300] step [1,1] width [1,1]) : 1
    genarray( [300,300]);
```

WithExpr ⇒ `with` `[` *GenAndExpr* `]`$^+$ *ConExpr*

GenAndExpr ⇒ `(` *Generator* `)` `[` *AssignBlock* `]` `:` *Expr*

Generator ⇒ *GenCube* `[` `step` *Expr* `[` `width` *Expr* `]` `]`

GenCube ⇒ *Expr* `<=` *GenIdx* `<` *Expr*

GenIdx ⇒ *Id*
| `[` *Id* `[` `,` *Id* `]`* `]`
| *Id* `=` `[` *Id* `[` `,` *Id* `]`* `]`

ConExpr ⇒ `genarray` `(` *ConstVec* `)`
| `modarray` `(` *Expr* `)`
| `fold` `(` *FoldFun* `,` *Expr* `)`

FoldFun ⇒ *FoldPrf*
| *[Id :] Id*

FoldPrf ⇒ `+` | `*`

Abbildung 2.4: Interne Darstellung des `with`*-Konstruktes.*

transformiert.

In den bisherigen Beispielen haben die einzelnen Generatormengen eine sehr kompakte Form — das ist aber nicht immer so. Falls die Generatoren der zu faltenden `with`-Konstrukte Raster beschreiben, d. h. `step` größer als `width` ist, setzt sich die durch das Falten entstandene Partition aus untereinander verschränkten Indexvektormengen zusammen. Beispielsweise wird das Programmfragment

```
A = with ([0,0] <= iv < [300,300] step [2,1] width [1,1])
    genarray( [300,300], 1);

B = with ([0,0] <= iv < [300,300] step [1,2] width [1,1])
    genarray( [300,300], 2);

C = with (. <= iv < .)
    modarray( A, iv, A[iv] + B[iv]);
```

zu

```
A = with ([0,0] <= iv < [300,300] step [2,2] width [1,1]): 3
         ([0,1] <= iv < [300,300] step [2,2] width [1,1]): 1
         ([1,0] <= iv < [300,300] step [2,2] width [1,1]): 2
         ([1,1] <= iv < [300,300] step [2,2] width [1,1]): 0
    genarray( [300,300]);
```

gefaltet.

Allgemein hat das `with`-Konstrukt intern die in Abbildung 2.4 in BNF dargestellte Form.

Kapitel 3

Zur Compilation von Array-Operationen

Array-Operationen arbeiten im allgemeinen auf großen Datenmengen. Um für einen schnellen Austausch solcher Daten zwischen Hauptspeicher und Prozessor zu sorgen, verfügen moderne Rechnerarchitekturen in der Regel über sogenannte Cache-Speicher. Auf Grund ihres Aufbaus und ihrer Funktionsweise hängt die Trefferrate im Cache, und damit das Laufzeitverhalten des betreffenden Anwendungsprogramms, wesentlich von der Reihenfolge der Speicherzugriffe ab. Bei der Compilation von Array-Operationen sind daher bestimmte, aus der Cache-Architektur ableitbare, Grundsätze zu beachten, um effizient ausführbaren Code zu erhalten. Es ist also notwendig, die Funktionsweise von Caches zu verstehen.

3.1 Caches

Wichtigste Eckdaten des Speichers eines Rechners sind zum einen seine Kapazität und zum anderen seine Zugriffszeit. Wünschenswert ist eine große Kapazität und eine geringe Zugriffszeit; diese beiden Eigenschaften lassen sich gemeinsam jedoch nur mit erheblichen Kosten erfüllen. Um einen Kompromiß zwischen Leistungsfähigkeit und Kosten zu erzielen, verfügen moderne Rechner in der Regel über eine mehrstufige Speicherhierarchie. Je weiter man, ausgehend vom Prozessor, in dieser Hierarchie fortschreitet, desto länger werden die Zugriffszeiten und desto größer wird die Kapazität der betreffenden Schicht. Eine typische Speicherkonfiguration könnte sich beispielsweise von kleinen schnellen Register- und Pufferspeichern nahe des Prozessors über den Hauptspeicher bis zu langsamen Hintergrundspeichern großer Kapazität erstrecken. Die Daten innerhalb der Speicherhierarchie sind so organisiert, daß jede einzelne Schicht einen Ausschnitt des größeren Speichers der nächsten Stufe enthält (Inklusionseigenschaft). Eine zwischen Prozessor und Hauptspeicher angeordnete Schicht der Speicherhierarchie wird als Cache bezeichnet [LF93, DK95].

Um von den kurzen Zugriffszeiten des Caches profitieren zu können, muß dafür gesorgt werden, daß vom Prozessor benötigte Daten nicht aus dem Hauptspeicher geholt werden müssen, sondern sich im Cache befinden. Da das Cache eine wesentlich geringere Kapazität als der Hauptspeicher hat und in ihm deshalb immer nur ein Ausschnitt des Hauptspeichers gepuffert werden kann, müssen Methoden entwickelt werden, nach denen Speicherinhalte in

das Cache geladen bzw. aus ihm verdrängt werden.

3.1.1 Cache-Typen

Inhalte des Hauptspeichers werden in der Regel nicht byte- oder wortweise in das Cache geladen, sondern in größeren Einheiten. Diese Einheiten werden Blöcke genannt und könnten beispielsweise vier Worte umfassen. Entsprechend ist auch das Cache nicht byte- oder wortweise, sondern nach Zeilen organisiert, wobei ein Block aus dem Hauptspeicher gerade in eine Cache-Zeile paßt. Für das Plazieren der einzelnen Blöcke im Cache gibt es unterschiedliche Vorgehensweisen (line placement policies), die sich in unterschiedlichen Cache-Typen niederschlagen. Die drei wichtigsten Typen sind vollassoziative Caches, mehrfach assoziative Caches, und einfach assoziative Caches.

3.1.1.1 Vollassoziativer Cache

Beim vollassoziativen Cache (fully associative cache) kann jeder Block aus dem Hauptspeicher in einer beliebigen Cache-Zeile abgelegt werden. Um diese Flexibilität optimal nutzen zu können, werden beim Laden des Caches Strategien angewendet, die eine bestmögliche Nutzung des Caches gewährleisten. Dazu müssen sogenannte Alterungsmechanismen implementiert werden, die bei gefülltem Cache vorgeben, welche Cache-Zeile beim nächsten Ladevorgang überschrieben wird (siehe Abschnitt 3.1.2.2). Vollassoziative Caches benötigen also einen hohen technischen Aufwand und sind entsprechend teuer.

3.1.1.2 Einfach assoziativer Cache

Das einfach assoziative Cache (direct mapped cache) ordnet jedem Block des Hauptspeichers über eine sogenannte Blocknummer, die sich aus der Speicheradresse des Blockes berechnet, eine eindeutige Cache-Zeile zu. Das bedeutet, daß von allen Speicherblöcken mit gleicher Blocknummer immer nur einer im Cache nachgehalten werden kann. Durch die starre Zuordnung der Blöcke zu den Cache-Zeilen vereinfacht sich der Aufbau des Caches. Außerdem sind bei diesem Cache-Typ keine Alterungsmechanismen notwendig, da es für einen Block keine Alternativen für die Plazierung innerhalb des Caches gibt. Dafür erreicht ein einfach assoziatives Cache im allgemeinen nicht die Trefferrate, die mit einem vollassoziativen möglich ist. Ein problematischer Fall ergibt sich zum Beispiel, falls unterschiedliche Daten mit gleicher Blocknummer wechselseitig benötigt werden, da diese sich gegenseitig aus dem Cache verdrängen (Cache-Konflikt). Die Blocknummer eines Datums berechnet sich üblicherweise aus

$$(\textit{Speicheradresse} \bmod \textit{Cache-Größe}) \quad .$$

Das bedeutet, daß Cache-Konflikte zwischen zwei Daten nur dann ausgeschlossen werden können, wenn diese im Speicher dicht beieinander liegen.

3.1.1.3 Mehrfach assoziativer Cache

Einen Kompromiß zwischen einem voll- und einem einfach assoziativen Cache stellt das n-fach assoziative Cache (n-way set associative cache) dar. Hier ist das Cache in Mengen partitioniert, wobei jede Menge aus n Cache-Zeilen besteht. Über Blocknummern wird jedem Speicherblock eindeutig eine dieser Mengen zugeordnet. Das Laden eines Blockes in das Cache geschieht dann auf folgende Weise: Zuerst wird analog zum einfach assoziativen Cache aus der Speicheradresse des Blockes die zugehörige Blocknummer und damit die passende Cache-Zeilen-Menge bestimmt. Anschließend wird aus dieser Menge eine Cache-Zeile ausgesucht und der Block dort plaziert. Um dabei eine möglichst günstige Wahl zu treffen, erfolgt diese Auswahl wie beim vollassoziativen Cache meist über Alterungsmechanismen (siehe Abschnitt 3.1.2.2).

3.1.2 Lade- und Ersetzungsstrategien

Bei Lese- und Schreiboperationen wird zur Vermeidung unnötiger Speicherzugriffe grundsätzlich zuerst das Cache inspiziert. Falls das benötigte Datum im Cache aufgefunden wird, handelt es sich um einen Trefferzugriff (cache hit). Andernfalls spricht man von einem Fehlzugriff (cache miss), und der Zugriff erfolgt über den Hauptspeicher. Die Trefferrate eines Caches hängt neben dem Programmverhalten wesentlich von der Strategie ab, nach der Daten in das Cache geladen werden. Bei voll- und mehrfach assoziativen Caches sind ferner geeignete Ersetzungsstrategien von wesentlicher Bedeutung.

3.1.2.1 Ladestrategien

Eine Ladestrategie, die von allen Caches angewendet wird, ist das Demand-Fetching. Bei ihr erfolgt das Laden eines Datums immer dann, wenn ein Fehlzugriff aufgetreten ist. Dies ist sinnvoll, da angenommen werden kann, daß dieses Datum in Kürze ein weiteres Mal benötigt wird. Der Ladevorgang ist dabei nicht auf das adressierte Datum beschränkt, sondern umfaßt stets einen ganzen Block. Falls diese zusätzlich geladenen Daten anschließend ebenfalls referenziert werden, hat dies den positiven Nebeneffekt, daß die Zahl der Fehlzugriffe weiter reduziert wird.

Diese Technik des Vorgriffes (Prefetching) läßt sich erweitern, indem mit jedem Block auch eine gewisse Zahl der nachfolgend im Speicher befindlichen Blöcke in das Cache geholt wird.

3.1.2.2 Ersetzungsstrategien

Um einen Speicherblock in das Cache laden zu können, muß eine Cache-Zeile ausgewählt werden. Beim einfach-assoziativen Cache geschieht dies in eindeutiger Weise durch die Blocknummer. Beim voll- oder mehrfach assoziativen Cache stehen dagegen für jede Blocknummer mehrere Cache-Zeilen zur Auswahl. Falls alle diese Cache-Zeilen bereits vergeben sind, muß nach einer bestimmten Strategie die Entscheidung getroffen werden, welcher Zeileninhalt durch den neuen Block ersetzt wird (replacement policies).

```
for (i = 0; i < 1000; i++) {
  for (j = 0; j < 10; j++) {
    A[j] = B[i];
  }
}
```

Abbildung 3.1: Beispiel für zeitliche und örtliche Lokalität.

Dabei wird in der Regel das Alterungsprinzip angewendet, das jeweils den „ältesten“ Eintrag verdrängt. Dies kann zum Beispiel die Cache-Zeile sein, die sich bereits am längsten im Cache befindet (first-in-first-out, FIFO) oder die längste Zeit nicht benutzt wurde (least-recently-used, LRU).

3.1.3 Schlußfolgerung

Zusammenfassend läßt sich feststellen, daß die Funktionsweise eines Caches auf dem Prinzip der Lokalität beruht, das aus zwei Annahmen besteht:

- Zeitliche Lokalität:

  Falls ein Datum referenziert wird, ist zu erwarten, daß es kurz darauf ein weiteres Mal referenziert wird.

- Örtliche Lokalität:

  Falls ein Datum referenziert wird, ist zu erwarten, daß benachbarte Daten kurz darauf ebenfalls referenziert werden.

Ein Beispiel soll diese beiden Begriffe illustrieren: In dem C-Programm aus Abbildung 3.1 wird in der inneren Schleife für jedes `i` zehnmal in Folge das selbe Element `B[i]` referenziert (zeitliche Lokalität), außerdem werden dort zehn aufeinanderfolgende Elemente des Arrays `A` geschrieben (örtliche Lokalität).

3.2 Cache-Performance

Die bisherigen Ausführungen zeigen, daß die Cache-Performance im konkreten Anwendungsfall ganz entscheidend davon abhängt, ob die Annahmen über Datenlokalität zutreffen. Man betrachte dazu das Beispielprogramm in Abbildung 3.2. Dort werden in der ersten Schleife die Elemente von `A` mit geradem Index, in der zweiten Schleife die Elemente mit ungeradem Index referenziert. Dies bedeutet, daß das Prinzip der örtlichen Lokalität in diesem Beispiel nicht eingehalten wird. So liegen etwa die Elemente `A[0]` und `A[1]` auf benachbarten Adressen im Speicher — und daher mit hoher Wahrscheinlichkeit in der selben Cache-Zeile —, werden jedoch nicht aufeinanderfolgend referenziert. Der Wert von `A[1]` wird also beim Zugriff auf `A[0]` mit in das Cache geladen, muß aber, wenn er in der zweiten Schleife benötigt

```
sum1 = sum2 = 0;
for (i = 0; i < N; i += 2) {
  sum1 += A[i];
}
for (i = 1; i < N; i += 2) {
  sum2 += A[i];
}
```

Abbildung 3.2: Beispiel mit verminderter örtlicher Datenlokalität.

wird, bei großen Werten für `N` erneut aus dem Hauptspeicher gelesen werden, da er in der Zwischenzeit aus dem Cache verdrängt worden ist.

Um in einem gegebenen Code-Fragment die Datenlokalität zu verbessern, gibt es zwei Möglichkeiten:

- Änderung des Datenlayouts des Arrays;
- Optimierung des Zugriffspfads durch geeignete Code-Transformationen.

Das Programm in Abbildung 3.2 könnte zum Beispiel optimiert werden, indem die Array-Elemente nicht in der kanonischen Reihenfolge im Speicher abgelegt werden, sondern zuerst alle Elemente mit geradem Index, dann alle mit ungeradem Index. Transformationen dieser Art werden in [AAL95] näher beschrieben (Strip Mining mit anschließender Permutation der Dimensionen). Vorteil solcher Datentransformationen ist, daß die Ausführungsreihenfolge innerhalb des Programms nicht verändert wird, bestehende Datenabhängigkeiten also nicht beachtet werden müssen. Problematisch wird es aber, falls ein Array in einem Programm in verschiedenen Kontexten auftritt. In diesem Fall kann möglicherweise kein Datenlayout gefunden werden, daß in jeder dieser Situationen optimal ist. Dann muß entweder eine deutlich verschlechterte Cache-Leistung hingenommen werden, oder die Datenstruktur des Arrays muß zur Laufzeit in die jeweils günstige Darstellung transformiert werden. Beide Maßnahmen haben im allgemeinen erhebliche Laufzeiteinbußen zur Folge. Auf Grund dieses gravierenden Nachteils soll der Ansatz der Datentransformation an dieser Stelle nicht weiter verfolgt werden.

Die Alternative besteht darin, das Datenlayout unverändert zu lassen und stattdessen den Code so zu transformieren, daß der Zugriffspfad eine günstige Form annimmt. In dem Beispielprogramm können etwa, wie in Abbildung 3.3, die beiden Schleifen zusammengefaßt werden. Dadurch wird erreicht, daß die Array-Elemente in der gleichen Reihenfolge referenziert werden, wie sie im Speicher abgelegt sind. Im folgenden Abschnitt werden diese und andere Code-Transformationen genauer untersucht.

3.3 Code-Transformationen

Nachfolgend werden einige wichtige Code-Transformationen vorgestellt, die sich bei der Compilation einsetzen lassen, um effizienter ausführbaren Code zu erhalten. Im Mittelpunkt steht

```
sum1 = sum2 = 0;
for (i = 0; i < N; i += 2) {
  sum1 += A[i];
  sum2 += A[i+1];
}
```

Abbildung 3.3: Variante zu Abbildung 3.2 mit optimierter örtlicher Datenlokalität.

```
sum = 0;
for (i1 = 0; i1 < N; i1++) {
  for (i0 = 0; i0 < N; i0++) {
    sum += A[i0,i1];
  }
}
```

```
sum = 0;
for (i0 = 0; i0 < N; i0++) {
  for (i1 = 0; i1 < N; i1++) {
    sum += A[i0,i1];
  }
}
```

Abbildung 3.4: Beispiel für Loop Interchange.

dabei zum einen die Optimierung der Datenlokalität, zum anderen die Reduzierung des Schleifen-Overheads. Eine umfassende Darstellung solcher Transformationen befindet sich in [BGS94].

3.3.1 Loop Interchange

Unter Loop Interchange wird das Vertauschen zweier ineinander verschachtelter Schleifen verstanden. Dadurch läßt sich in manchen Situationen ein besseres Cache-Verhalten erreichen. Es werde etwa das Beispiel in Abbildung 3.4 betrachtet. Wenn davon ausgegangen wird, daß Arrays — analog zu SAC — zeilenweise (Zeile $\equiv$ 0-te Dimension) im Speicher abgelegt werden, so wird im Ausgangsprogramm auf der linken Seite das Array `A` mit einer Schrittweite (Stride) von `N` durchlaufen. Dies hat bei großem `N` zur Folge, daß die im Vorgriff in das Cache geladenen Daten kaum genutzt werden können.

Lösen läßt sich das Problem, indem die beiden Schleifen vertauscht werden. Das ist in diesem Fall eine legale Transformation, da zwischen den einzelnen Iterationen keine Datenabhängigkeit besteht. Als Ergebnis erhält man das Programm auf der rechten Seite der Abbildung. In dieser optimierten Version wird das Array `A` mit der Schrittweite 1 durchlaufen; es liegt also eine gute örtliche Lokalität vor.

3.3.2 Loop Fusion

Die Vorgehensweise bei der Transformation Loop Fusion wurde bereits an Hand des Beispiels in den Abbildungen 3.2 und 3.3 demonstriert. Durch das Zusammenfassen von Schleifen läßt sich zum einen das Schleifen-Overhead reduzieren, zum anderen wird unter Umständen die Datenlokalität verbessert.

```
for (i0 = 0; i0 < N; i0++) {
  for (i1 = 0; i1 < N; i1++) {
    A[i0,i1] = B[i0];
  }
}
for (i0 = 0; i0 < N; i0++) {
  for (i1 = N; i1 < 2*N; i1++) {
    A[i0,i1] = 1 - B[i0];
  }
}
```

```
for (i0 = 0; i0 < N; i0++) {
  for (i1 = 0; i1 < N; i1++) {
    A[i0,i1] = B[i0];
  }
  for (i1 = N; i1 < 2*N; i1++) {
    A[i0,i1] = 1 - B[i0];
  }
}
```

Abbildung 3.5: Beispiel für Loop Fusion.

```
for (i = 0; i < N; i++) {
  A1[i] = B[i] + 1;
}
for (i = 0; i < 2*N; i++) {
  A2[i] = B[i] - 2;
}
```

```
for (i = 0; i < N; i++) {
  A1[i] = B[i] + 1;
}
for (i = 0; i < N; i++) {
  A2[i] = B[i] - 2;
}
for (i = N; i < 2*N; i++) {
  A2[i] = B[i] - 2;
}
```

Abbildung 3.6: Beispiel für Loop Peeling.

Für das Beispiel in Abbildung 3.5 erhält man durch Fusion der beiden äußeren Schleifen eine optimierte örtliche Lokalität bezüglich der Arrays `A`, sowie eine optimierte zeitliche Lokalität bezügliche des Arrays `B`.

Damit zwei Schleifen zusammengefaßt werden können, müssen sie die gleichen Schleifengrenzen und -schrittweiten besitzen. Falls diese Werte nicht gleich sind, die Schleifen aber eine identische Anzahl von Iterationen durchführen, können die Schleifengrenzen durch Translation auf die gleiche Form gebracht werden. Die beiden Schleifen in Abbildung 3.2 lassen sich zum Beispiel fusionieren, wenn für die zweite Schleife ein Offset von -1 eingeführt wird. Ist auch die Anzahl der Iterationen bei zwei Schleifen unterschiedlich, kann dieses Mißverhältnis bei Bedarf durch Loop Peeling oder Loop Unrolling (siehe folgende Abschnitte) behoben werden.

3.3.3 Loop Peeling

Loop Peeling bezeichnet das Heraustrennen von Iterationen am Beginn oder Ende einer Schleife. Dadurch läßt sich zum Beispiel der Iterationsbereich zweier Schleifen angleichen, um Loop Fusion zu ermöglichen. Ein Beispiel dazu zeigt Abbildung 3.6: Nach dem Aufteilen ergeben sich drei Schleifen, von denen nun die ersten beiden zusammengefaßt werden können.

```
A = ...;
for (i0 = 0; i0 < N; i0++) {
  for (i1 = 0; i1 < N; i1++) {
    B[i0,i1] = A[i1,i0];
  }
}
```

```
A = ...;
for (b0 = 0; b0 < N; b0 += BS0) {
  for (b1 = 0; b1 < N; b1 += BS1) {
    for (i0 = b0; i0 < min(b0 + BS0, N); i0++) {
      for (i1 = b1; i1 < min(b1 + BS1, N); i1++) {
        B[i0,i1] = A[i1,i0];
      }
    }
  }
}
```

Abbildung 3.7: Beispiel für Loop Blocking.

3.3.4 Loop Blocking

Loop Blocking ist eine Methode zur Verbesserung der Trefferrate im Cache. Die Idee besteht darin, nicht auf kompletten Zeilen oder Spalten eines Arrays zu operieren, sondern auf Unterarrays — sogenannten Blöcken. Den Nutzen des Loop Blocking demonstriert das Beispiel in Abbildung 3.7 (oberer Teil). Dieses Programm weist dem Array `B` das transponierte Array `A` zu. Es fällt auf, daß der Zugriffspfad auf die Elemente von `B` die optimale Schrittweite 1 hat, der Zugriff auf die Elemente von `A` aber mit der ungünstigen Schrittweite `N` erfolgt. Mit den bisher vorgestellten Transformationen läßt sich dieses Problem nicht lösen. Loop Interchange würde beispielsweise lediglich die Rollen der Arrays `A` und `B` vertauschen, am grundsätzlichen Dilemma aber nichts ändern.

Um trotz des ungünstigen Zugriffspfads jede Cache-Zeile nutzen zu können, wird der Iterationsraum in Blöcke gleicher Größe aufgeteilt. Die Blocklängen in den einzelnen Dimensionen (auch Blocking-Faktoren genannt; im Beispiel `BS0`, `BS1`) werden dabei so klein gewählt, daß ein kompletter Block in das Cache paßt. Durch das Einfügen je einer neuen Schleife pro Dimension läßt sich erreichen, daß nacheinander über diese Blöcke iteriert wird (siehe Abbildung 3.7, unterer Teil).

Eine ausführliche Darstellung dieser Optimierung findet sich in [LRW91]. Dort wird am Beispiel der Matrix-Multiplikation unter anderem ein Verfahren entwickelt, mit dessen Hilfe die optimale Blockgröße ermittelt werden kann.

Wie bereits im Abschnitt 3.1 erwähnt wurde, besteht das Cache im allgemeinen aus einer mehrere Ebenen umfassenden Hierarchie. Dieses Prinzip läßt sich auf das Blocking übertragen. Gegeben sei beispielsweise eine Einprozessorarchitektur mit zwei externen Caches und

```
for (i = 1; i < N-1; i++) {
  A[i] = A[i] + A[i-1] + A[i+1];
}
```

```
for (i = 1; i < N-2; i+=2) {
  A[i] = A[i] + A[i-1] + A[i+1];
  A[i+1] = A[i+1] + A[i] + A[i+2];
}
if ((N-2) % 2) {
  A[N-2] = A[N-2] + A[N-3] + A[N-1];
}
```

Abbildung 3.8: Beispiel für Loop Unrolling.

einem internen Prozessor-Cache. Dann könnte zum Beispiel ein zweidimensionales Array in Blöcke der Größe 100×100 aufgeteilt werden, um eine gute Ausnutzung des ersten externen Caches zu erreichen. Anschließend könnte man dazu übergehen, die Fehlzugriffe im zweiten externen Cache zu minimieren, indem jeder dieser Blöcke erneut aufgeteilt wird. Durch weiteres Blocking ließe sich dann die Ausnutzung des Prozessor-Caches optimieren. Dieses Vorgehen wird als hierarchisches Blocking bezeichnet.

3.3.5 Loop Unrolling

Die Transformation Loop Unrolling vergrößert die Iterations-Schrittweite einer Schleife um einen bestimmten Faktor (genannt Unrolling-Faktor) und dupliziert dafür den Code im Schleifenrumpf entsprechend. Da die Iterationsreihenfolge dabei unverändert bleibt, kann diese Optimierung unabhängig von bestehenden Datenabhängigkeiten durchgeführt werden.

Unrolling reduziert in jedem Fall das Schleifen-Overhead und verbessert unter Umständen die Lokalität. Ein Beispiel wird in Abbildung 3.8 gegeben. Der Unrolling-Faktor hat dort den Wert 2 — dadurch wird die Anzahl der Iterationen und mithin auch die Anzahl der Schleifensprünge und Inkrementationen der Iterationsvariablen halbiert. Ferner fällt auf, daß die Array-Elemente `A[i]` und `A[i+1]` nach dem Abrollen im Schleifenrumpf zweimal verwendet werden. Legt der Compiler diese Werte etwa in Registern ab, brauchen diese nun nicht mehr so oft geladen zu werden.

Falls nicht sichergestellt werden kann, daß die Anzahl der Schleifendurchläufe (I) ein Vielfaches des Unrolling-Faktors (u) ist, muß im Anschluß an die transformierte Schleife noch ein Epilog eingefügt werden, der die verbleibenden $(I \bmod u)$ Iterationen durchführt. Im Fall $u = 2$ kann dies mit Hilfe eines `if`-Statements gelöst werden (siehe Beispiel). Wenn $u > 2$ ist, wird auch dieser Epilog selbst wieder eine Schleife sein, die bei Bedarf abgerollt wird.

Kapitel 4

Zur Compilation des `with`-Konstruktes

In diesem Kapitel werden an Hand einiger Beispiele die Methoden entwickelt, die die Compilation des `with`-Konstruktes in effizient ausführbaren C-Code ermöglichen.

4.1 Naive Compilation

Die einfachste und naheliegenste Möglichkeit, ein `with`-Konstrukt in ein C-Compilat zu übersetzen, besteht darin, jeden Generator für sich in eine perfekte Schachtelung von `for`-Schleifen zu überführen. Für die j-te Komponente einer Generatormenge mit den Grenzen a, b und der Rasterung s, w (siehe Abschnitt 2.3)

$$I((\ldots, a_j, \ldots), (\ldots, b_j, \ldots), (\ldots, s_j, \ldots), (\ldots, w_j, \ldots))$$

ergibt sich im allgemeinen Fall folgende Schleifenkonstruktion:

```
for (iv_j = a_j; iv_j < b_j; iv_j += s_j-w_j) {
  stop_j = iv_j + w_j;
  for (; iv_j < stop_j; iv_j++) {
    /* Schleifen für nächste Dimension bzw. Zielausdruck */
  }
}
```

.

Falls $w_j = 1$ gilt, kann dies vereinfacht werden zu:

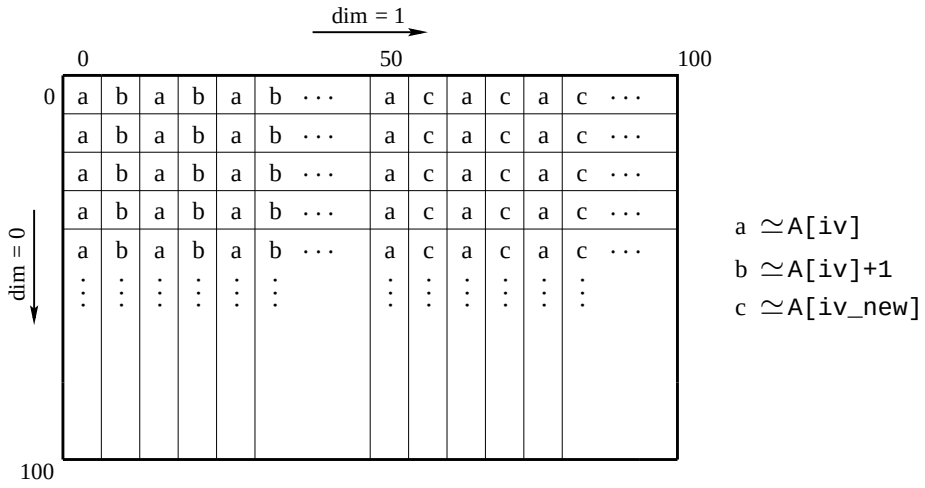
```
for (iv_j = a_j; iv_j < b_j; iv_j += s_j) {
  /* Schleifen für nächste Dimension bzw. Zielausdruck */
}
```

.

Daraus entsteht das naive Compilat, indem diese Schleifenausdrücke — nach aufsteigender Dimension geordnet — ineinander verschachtelt werden, und als Rumpf der innersten Schleife der Zielausdruck des `with`-Konstruktes nebst optionalem Zuweisungsblock eingefügt wird. Als Beispiel diene das `with`-Konstrukt aus Abbildung 4.1(a). Dort wird ein Array `A` mit dem

```
A = ...;
B = with ([0, 0] <= iv < [100,100] step [1,2] width [1,1]): A[iv]
         ([0, 1] <= iv < [100, 51] step [1,2] width [1,1]): A[iv] + 1
         ([0,51] <= iv < [100,100] step [1,2] width [1,1]) {
                                                iv_new = iv - [0,1];
                                              } : A[iv_new]
    genarray( [100,100]);
```

(a) SAC*-Code.*



(b) Diagramm der Indexvektormenge.

```
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 100; iv1 += 2) {
    B[iv] = A[iv];
  }
}
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 1; iv1 < 51; iv1 += 2) {
    B[iv] = A[iv] + 1;
  }
}
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 51; iv1 < 100; iv1 += 2) {
    iv_new = iv - [0,1];
    B[iv] = A[iv_new];
  }
}
```

(c) Naives Compilat.

Abbildung 4.1: Beispiel für die naive Compilation eines with*-Konstruktes.*

Formvektor $(100, 100)$ erzeugt — der Aufbau dieses Arrays ist Abbildung 4.1(b) zu entnehmen. Bei naiver Compilation ergibt sich für jeden der drei Generatoren eine perfekte Schachtelung von zwei `for`-Schleifen (siehe Abbildung 4.1(c)). Die im Compilat verwendete Schreibweise `A[iv]` soll dabei die Selektion des mit dem Indexvektor `[iv0,iv1]` korrespondierenden Elementes des Datenvektors von `A` andeuten.

Diese naive Compilation hat jedoch zwei gravierende Nachteile. Zum einen ist dadurch, daß jeder Generator quasi separat compiliert wird, das Schleifen-Overhead recht groß. Zum anderen ist die Datenlokalität des Compilats sehr schlecht, falls zumindest einer der Generatoren ein Raster spezifiziert ($s > w$). In dem obigen Beispiel werden etwa die Elemente des zu erzeugenden Arrays `B` nicht in der Reihenfolge berechnet, wie sie im Datenvektor abgelegt werden müssen, sondern mit einer Schrittweite von 2. Gleiches gilt für die Lesezugriffe auf das Array `A`.

4.2 Kanonische Reihenfolge

Um sowohl das Schleifen-Overhead zu reduzieren, als auch die Datenlokalität zu verbessern, muß das Schleifen-Layout des naiven Compilats optimiert werden. Es liegt nahe, die Schleifen so umzuorganisieren, daß die Iterationsreihenfolge exakt der Abfolge innerhalb des zu erzeugenden Datenvektors entspricht — diese Reihenfolge werde als kanonisch bezeichnet. Diese Umordnung gelingt durch systematische Anwendung einiger der in Abschnitt 3.3 vorgestellten Code-Transformationen. Dadurch wird nicht nur erreicht, daß die Schreibzugriffe mit der optimalen Schrittweite 1 erfolgen, sondern diese Eigenschaft überträgt sich auch auf alle anderen Arrayzugriffe, sofern sie affin von der Iterationsvariablen abhängen. Es ergibt sich also eine deutlich verbesserte Datenlokalität und mithin bessere Cache-Performance. Außerdem wird das Schleifen-Overhead minimiert.

Dies soll an Hand des Beispiels aus dem letzten Abschnitt verdeutlicht werden. Abbildung 4.2 demonstriert, wie das Compilat mit Hilfe der Transformationen Loop Fusion, Peeling und Translation in die kanonische Iterationsreihenfolge überführt werden kann. In dem optimierten Code erfolgen nun alle Arrayzugriffe mit der Schrittweite 1, und das Schleifen-Overhead ist auf ein Minimum reduziert.

In Bezug auf imperative Programmiersprachen und insbesondere die Sprache C tritt jedoch ein Problem auf: Eine solche Umordnung der Ausführungsreihenfolge innerhalb des Programms, in diesem Fall konkret die Durchführung des Loop Fusion, setzt voraus, daß zwischen den einzelnen Iterationen keine unerlaubten Datenabhängigkeiten bestehen. Für das Compilat eines `with`-Konstruktes ist dies sicherlich gegeben, da der Zielausdruck des `with`-Konstruktes für die Elemente der Generatormenge konzeptionell nebenläufig berechnet wird. Jedoch kann dies ein C-Compiler an Hand des naiven Compilats im allgemeinen nicht inferieren, da er dort auf Grund der Verwendung von Zeigern mit Aliasing und eventuellen Seiteneffekten rechnen muß. Daher kann man sich nicht darauf verlassen, daß diese Optimierung vom C-Compiler durchgeführt wird. Vielmehr muß die Überführung in die kanonische Reihenfolge explizit in das Compilations-Schema des `with`-Konstruktes aufgenommen werden.

```
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 100; iv1 += 2) {
    B[iv] = A[iv];
  }
}
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 1; iv1 < 51; iv1 += 2) {
    B[iv] = A[iv] + 1;
  }
}
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 51; iv1 < 100; iv1 += 2) {
    iv_new = iv - [0,1];
    B[iv] = A[iv_new];
  }
}
```

↓ Loop Fusion ↓

```
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 100; iv1 += 2) {
    B[iv] = A[iv];
  }
  for (iv1 = 1; iv1 < 51; iv1 += 2) {
    B[iv] = A[iv] + 1;
  }
  for (iv1 = 51; iv1 < 100; iv1 += 2) {
    iv_new = iv - [0,1];
    B[iv] = A[iv_new];
  }
}
```

↓ Loop Peeling ↓

```
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 50; iv1 += 2) {
    B[iv] = A[iv];
  }
  for (iv1 = 50; iv1 < 100; iv1 += 2) {
    B[iv] = A[iv];
  }
  for (iv1 = 1; iv1 < 51; iv1 += 2) {
    B[iv] = A[iv] + 1;
  }
  for (iv1 = 51; iv1 < 100; iv1 += 2) {
    iv_new = iv - [0,1];
    B[iv] = A[iv_new];
  }
}
```

Abbildung 4.2: Überführung in die kanonische Iterationsreihenfolge.
(wird fortgesetzt)

↓ Loop Translation ↓

```
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 50; iv1 += 2) {
    B[iv] = A[iv];
  }
  for (iv1 = 50; iv1 < 100; iv1 += 2) {
    B[iv] = A[iv];
  }
  for (iv1 = 0; iv1 < 50; iv1 += 2) {
    iv_trans = iv + [0,1];
    B[iv_trans] = A[iv_trans] + 1;
  }
  for (iv1 = 50; iv1 < 100; iv1 += 2) {
    iv_trans = iv + [0,1];
    iv_new = iv_trans - [0,1];
    B[iv_trans] = A[iv_new];
  }
}
```

↓ Loop Fusion ↓

```
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 50; iv1 += 2) {
    B[iv] = A[iv];

    iv_trans = iv + [0,1];
    B[iv_trans] = A[iv_trans] + 1;
  }
  for (iv1 = 50; iv1 < 100; iv1 += 2) {
    B[iv] = A[iv];

    iv_trans = iv + [0,1];
    iv_new = iv_trans - [0,1];
    B[iv_trans] = A[iv_new];
  }
}
```

Abbildung 4.2: Überführung in die kanonische Iterationsreihenfolge. (Fortsetzung)

4.3 Segmentierung

Im letzten Abschnitt wurde dargelegt, daß es aus Gründen der Cache-Performance wünschenswert ist, innerhalb des Compilats eines `with`-Konstruktes in der kanonische Reihenfolge über die Indexvektoren zu iterieren. Leider hat dies im Zusammenhang mit inhomogenen Generatormengen aber auch Nachteile. Das Beispiel in Abbildung 4.3 soll dies verdeutlichen. Das dort aufgeführte `with`-Konstrukt beschreibt ein Array, das aus zwei Teilen besteht. Die ersten

```
A = with ([0, 0] <= iv < [60,30] step [3,1] width [1,1]): 1
         ([1, 0] <= iv < [60,30] step [3,1] width [2,1]): 0
         ([0,30] <= iv < [60,60] step [2,1] width [1,1]): 1
         ([1,30] <= iv < [60,60] step [2,1] width [1,1]): 0
    modarray( A);
```

Abbildung 4.3: With*-Konstrukt mit inhomogenem Aufbau der Generatormengen.*

```
for (iv0 = 0; iv0 < 60; iv0 += 3) {
  for (iv1 = 0; iv1 < 30; iv1++) {
    A[iv] = 1;
  }
}
for (iv0 = 1; iv0 < 60; iv0++) {
  stop0 = iv0 + 2;
  for (; iv0 < stop0; iv0++) {
    for (iv1 = 0; iv1 < 30; iv1++) {
      A[iv] = 0;
    }
  }
}
for (iv0 = 0; iv0 < 60; iv0 += 2) {
  for (iv1 = 30; iv1 < 60; iv1++) {
    A[iv] = 1;
  }
}
for (iv0 = 1; iv0 < 60; iv0 += 2) {
  for (iv1 = 30; iv1 < 60; iv1++) {
    A[iv] = 0;
  }
}
```

Abbildung 4.4: Naives Compilat des with*-Konstruktes aus Abbildung 4.3.*

30 Spalten des Arrays enthalten in jeder dritten Zeile eine 1 und sonst nur 0 — also ein Muster der Periode 3. Die nächsten 30 Spalten enthalten abwechselnd 0 und 1 — also ein Muster der Periode 2. Naiv compiliert ergibt sich der in Abbildung 4.4 angegebene C-Code. Um dieses Code-Fragment in die kanonische Reihenfolge zu überführen, müssen alle vier Schleifen über iv0 fusioniert werden. Das bedeutet jedoch, daß sich die fusionierte Schleife sowohl mit der Periode 2 des linken Teil-Arrays, als auch mit der Periode 3 des rechten Teil-Arrays vertragen muß. Mithin muß sie zumindest die Periode $\mathrm{kgV}(2,3) = 6$ besitzen, wodurch der Schleifenrumpf entsprechend umfänglicher wird. Bei ungünstigen Rasterungsverhältnissen zwischen den Generatormengen führt also die Überführung in die kanonische Reihenfolge zu sehr komplexen Schleifen, wodurch sich die Größe des Compilats signifikant erhöht.

Um einen guten Kompromiß zwischen Cache-Performance und Compilat-Größe zu errei-

```
for (iv0 = 0; iv0 < 60; ) {
  for (iv1 = 0; iv1 < 30; iv1++) {
    A[iv] = 1;
  } iv0++;

  stop0 = iv0 + 2;
  for (; iv0 < stop0; iv0++) {
    for (iv1 = 0; iv1 < 30; iv1++) {
      A[iv] = 0;
    }
  }
}
for (iv0 = 0; iv0 < 60; ) {
  for (iv1 = 30; iv1 < 60; iv1++) {
    A[iv] = 1;
  } iv0++;

  for (iv1 = 30; iv1 < 60; iv1++) {
    A[iv] = 0;
  } iv0++;
}
```

Abbildung 4.5: Compilat des `with`*-Konstruktes aus Abbildung 4.3 mit kanonischer Reihenfolge auf zwei Segmenten.*

chen, müssen daher Bereiche der Indexvektormenge, die sich in ihrer Rasterung sehr unterscheiden, vor der Überführung in die kanonische Reihenfolge voneinander entkoppelt werden. Dazu wird die Menge der Indexvektoren in paarweise disjunkte Bereiche unterteilt — sogenannte Segmente —, die sich idealerweise durch eine homogene Rasterung auszeichnen. Anschließend wird nicht die gesamte Indexvektormenge in die kanonische Reihenfolge überführt, sondern jedes Segment für sich.

Dieser Systematik folgend könnte etwa das Array aus dem obigen Beispiel in zwei Segmente — eine linke ($S_1 = [\,(0,0);\,(60,30)\,)$) und eine rechte Hälfte ($S_2 = [\,(0,30);\,(60,60)\,)$) — zerlegt werden. Daraus wird der in Abbildung 4.5 abgebildete Code erzeugt. Er besteht aus zwei aufeinanderfolgenden Schleifen-Nestings: zuerst wird in kanonischer Reihenfolge über S_1 iteriert, anschließend über S_2. Diese Zweiteilung des Arrays hat zwar zur Folge, daß die Cache-Performance etwas nachläßt, da der Datenvektor von `A` nicht durchgängig mit der Schrittweite 1 durchlaufen wird, sondern nach jeweils 30 Elementen die nächsten 30 übersprungen werden. Auf der anderen Seite kann aber das Aufblähen der äußere Schleife auf die Periode 6 verhindert und damit der Schleifenaufbau vereinfacht werden.

4.3.1 Definition

Der Begriff des Segmentes läßt sich wie folgt formalisieren: Gegeben sei ein with-Konstrukt mit N Generatoren. Dann sei definiert:

- Die k-te ($k \in \{1, 2, \ldots, N\}$) Generatormenge sei mit $G_k := I(a_k, b_k, s_k, w_k)$ bezeichnet;
- $\mathcal{G} := \{G_1, \ldots, G_N\}$;
- $I_{Dom} := \bigcup \mathcal{G}$. Falls der Operator des with-Konstruktes ein genarray(sh) ist, gilt dann $I_{Dom} = I(sh)$. Für einen Operator der Form modarray(A) erhält man $I_{Dom} = I(sh_A)$.

Eine Segmentierung bezeichnet dann eine Partition $\mathcal{S} := \{S_1, S_2, \ldots, S_M\}$ von I_{Dom}, wobei für alle $l \in \{1, 2, \ldots, M\}$ gilt:

$$\exists\, a, b \in {\mathbb{N}_0}^* : S_l = [\, a;\, b\,) \cap I_{Dom} \quad . \tag{4.1}$$

Die in dieser Definition vorkommende Schnittbildung erreicht, daß Segmente — soweit möglich — vollständige Intervalle über ${\mathbb{N}_0}^*$ sind. An dieser Stelle stets auch Raster zuzulassen ist wenig sinnvoll, da sich auf diese Weise kaum gute Cache-Performance erreichen läßt. Man beachte jedoch, daß es im allgemeinen keinen Sinn macht, $S_l = [\, a;\, b\,)$ zu fordern, da die Menge I_{Dom} bei with-Konstrukten mit fold-Operator isolierte Raster enthalten kann, sodaß sich gerasterte Segmente nicht prinzipiell vermeiden lassen. Das (eindeutig bestimmte) *minimale* Intervall $[\, a;\, b\,)$ mit der Eigenschaft (4.1) wird Umriss des Segments S_l genannt und als $\overline{S_l}$ notiert.[1]

4.3.2 Quader

Für ein gegebenes with-Konstrukt gibt es im allgemeinen mehrere Möglichkeiten einer sinnvollen Segmentierung. In jedem Fall eindeutig bestimmt ist dagegen eine spezielle Art der Segmentierung, nämlich die Einteilung der Indexvektormenge in sogenannte Quader. Ein Quader ist ein spezielles Segment, dadurch ausgezeichnet, daß es aus *einem einzigen* sich periodisch wiederholendem Muster besteht. Die Quader sind von großer Bedeutung, da sich alle anderen sinnvollen Segmente durch Mengenvereinigung aus diesen ableiten lassen. Quader sind also die Bausteine jeder sinnvollen Segmentierung.

Dies soll an Hand eines Beispiels demonstriert werden. Abbildung 4.6 zeigt das Diagramm einer zu einem with-Konstrukt mit sechs Generatoren $\{G_1, \ldots, G_6\}$ gehörigen Indexvektormenge. In Richtung der Dimension $\dim = 0$ besitzen die Generatormengen $\{G_1, G_2, G_5, G_6\}$ die Periode 2, und die Generatormengen $\{G_3, G_4\}$ die Periode 4. Entlang der Achse $\dim = 1$ haben alle Generatormengen die Schrittweite 1.

In der linken oberen Ecke des Diagramms beginnt ein Muster mit der Periode $(2, 1)$ — d. h. Schrittweite 2 in Richtung der Achse $\dim = 0$ und Schrittweite 1 in Richtung $\dim = 1$

[1] Der Begriff des Umrisses läßt sich ohne weiteres auf beliebige Indexvektormengen ausdehnen und wird nachfolgend auch so allgemein verwendet.

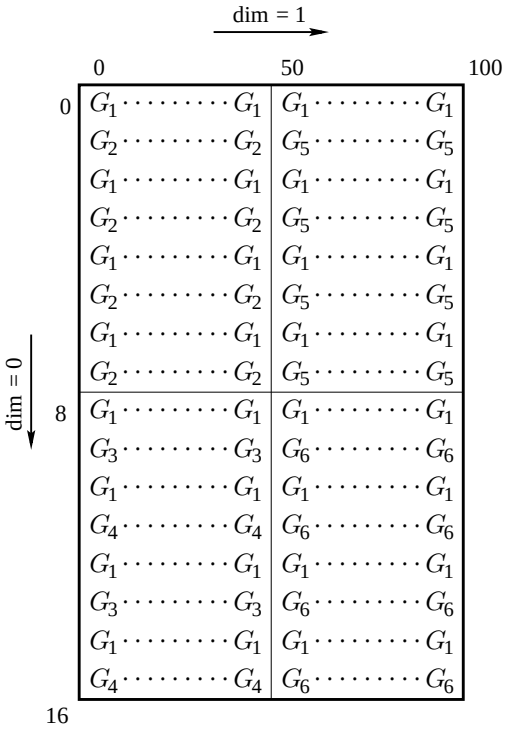


Abbildung 4.6: Beispiel für die Berechnung der Quader einer Indexvektormenge.

—, welches sich aus je einem Element von G_1 und G_2 zusammensetzt. Dieses Muster füllt genau das Intervall $[\,(0,0);\,(8,50)\,)$ aus. Da dieses Intervall ein komplettes Segment darstellt, ist damit einer der gesuchten Quader (Q_1) identifiziert. Auf analoge Weise lassen sich die Quader $Q_2 := [\,(0,50);\,(8,100)\,)$ und $Q_4 := [\,(8,50);\,(16,100)\,)$ finden. Das noch verbleibende Intervall $Q_3 := [\,(8,0);\,(16,50)\,)$ ist ebenfalls ein Quader, da es aus einem Muster der Periode $(4,1)$, bestehend aus zwei Elementen von G_1 und je einem Element von G_3 und G_4, aufgebaut ist.

Diese vier Quader können nun direkt als Segmente übernommen werden, oder es wird durch Mengenvereinigung aus ihnen eine geeignete Segmentierung gewonnen. In diesem konkreten Fall würde es sich etwa anbieten, die Quader Q_1 und Q_2 zu einem Segment zusammenzufassen, da diese entlang der gemeinsamen Achse $\dim = 0$ in ihren Rasterperiodenlängen übereinstimmen.

Es soll nun ein Algorithmus angegeben werden, der zu einer vorgegebenen Menge $\mathcal{G}$ von N Generatormengen der Dimension n

$$\mathcal{G} := \{G_1, \ldots, G_N\}\,, \quad \text{wobei} \quad G_k := I(a_k, b_k, s_k, w_k) \quad ,$$

die entsprechende Quadermenge $\mathcal{Q}$ bestimmt. Um diesen Algorithmus elegant spezifizieren zu können, wird der Begriff des maximalen Umrisses eingeführt. Sei $I := I(a, b, s, w)$ eine Indexvektormenge, dann ist der maximale Umriss $\overline{\overline{I}}$ von I definiert durch

$$\overline{\overline{I}} := [\,a';\,b'\,) \quad ,$$

wobei gilt:

$$
\begin{aligned}
a' &:= \max(0, a - (s - w)), \\
b'_j &:= \left\{ \begin{array}{ll} b_j + s_j - ((b_j - a_j) \bmod s_j) & ; \text{ falls } (b_j - a_j) \bmod s_d \geq w_d \\ b_j & ; \text{ falls } (b_j - a_j) \bmod s_d < w_d \end{array} \right\}, \\
& \qquad\qquad\qquad\qquad\qquad\qquad \text{für } j \in \{0, \ldots, n-1\} \quad .
\end{aligned}
$$

$\overline{\overline{I}}$ bezeichnet also das *maximale* Intervall, das sich aus I unter Vernachlässigung der Rasterung ($s, w := 1$) ergibt.[2] Unter Verwendung dieser Notation läßt sich die Berechnung der Quadermenge $\mathfrak{Q}$ wie folgt spezifizieren:

1. Zuerst wird die Menge

$$
\begin{aligned}
\mathfrak{I}^{(0)} := & \left\{ G_k \cap \overline{\overline{G_l}} \mid (G_k, G_l \in \mathfrak{G}) \wedge (G_k \neq G_l) \wedge (\overline{G_k} \cap \overline{G_l} \neq \emptyset) \right\} \\
& \cup \left\{ G_k \mid (G_k \in \mathfrak{G}) \wedge (\forall\, (G_l \in \mathfrak{G}) : (G_k \neq G_l) \rightarrow (\overline{G_k} \cap \overline{G_l} = \emptyset)) \right\}
\end{aligned}
$$

   berechnet, d. h. jede Generatormenge wird mit den maximalen Umrissen der übrigen Generatormengen geschnitten. Die Menge $\mathfrak{I}^{(0)}$ stellt dann (ebenso wie $\mathfrak{G}$) eine Partition der Grundmenge I_{Dom} dar.

Nachtrag: Leider ist $\mathfrak{I}^{(0)}$ im allgemeinen *keine* Partition von I_{Dom}, wie sich an Hand eines einfachen Beispiels demonstrieren läßt. Sei etwa die folgende Generatormenge gegeben:

$$\mathfrak{G} := \{G_1, G_2\} \quad \text{mit } G_1 := I(0,3,2,1), G_2 := I(1,2,1,1) \quad .$$

Dann ist $G_1 \cap \overline{\overline{G_2}} = \emptyset$ und $G_2 \cap \overline{\overline{G_1}} = G_2$, also ergibt sich $\mathfrak{I}^{(0)} = \{\emptyset, G_2\}$, somit gilt $\bigcup \mathfrak{I}^{(0)} = G_2 \neq \bigcup \mathfrak{G}$. Dies führt dazu, daß der Algorithmus zur Quader-Berechnung für dieses Beispiel nicht korrekt funktioniert.
Um sicherzustellen, daß $\mathfrak{I}^{(0)}$ stets eine Partition von I_{Dom} ist, dürfen keine Generatormengen auftreten (hier: G_1), in die sich andere Generatormengen zwischen zwei aufeinanderfolgenden Rasterperioden einbetten lassen (hier: G_2). Also müssen in einem vorbereitenden Arbeitsschritt alle Generatormengen dieser Art an der entsprechenden Stelle aufgetrennt werden. Für das angegebene Beispiel würde G_1 in $I(0,1,1,1)$ und $I(2,3,1,1)$ zerlegt werden, d. h. $\mathfrak{I}^{(0)}$ würde an Hand der modifizierten Generatormenge $\mathfrak{G} := \{I(0,1,1,1), I(1,2,1,1), I(2,3,1,1)\}$ berechnet werden.

2. Die formale Spezifikation von Indexvektormengen mit Hilfe der Parameter a, b, s, w ist bezüglich dieser Schnittbildung abgeschlossen. Jedes Element der Menge $\mathfrak{I}^{(0)}$ läßt sich also wieder auf die Form $I(a, b, s, w)$ bringen.

   Daher kann diese Schnittbildung ausgehend von $\mathfrak{I}^{(0)}$ sukzessiv fortgeführt werden

$$
\begin{aligned}
\mathfrak{I}^{(m+1)} := & \left\{ I_k \cap \overline{\overline{I_l}} \mid (I_k, I_l \in \mathfrak{I}^{(m)}) \wedge (I_k \neq I_l) \wedge (\overline{I_k} \cap \overline{I_l} \neq \emptyset) \right\} \\
& \cup \left\{ I_k \mid (I_k \in \mathfrak{I}^{(m)}) \wedge (\forall\, (I_l \in \mathfrak{I}^{(m)}) : (I_k \neq I_l) \rightarrow (\overline{I_k} \cap \overline{I_l} = \emptyset)) \right\} \quad ,
\end{aligned}
$$

   bis der, auf Grund der Konstruktion stets vorhandene, Fixpunkt $\mathfrak{I}^* := \mathfrak{I}^{(m+1)} = \mathfrak{I}^{(m)}$ erreicht ist. Auch $\mathfrak{I}^*$ ist eine Partition von I_{Dom}.

[2] Zur Erinnerung: Bei $\overline{I}$ ist es das *minimale* Intervall.

3. Aus der Menge $\mathfrak{I}^*$ läßt sich nun die Quadermenge $\mathfrak{Q}$ gewinnen, indem alle Indexvektormengen aus $\mathfrak{I}^*$, die in übereinstimmenden Intervallen liegen, vereinigt werden.

   Dazu sei eine Äquivalenzrelation $\sim$ über $\mathfrak{I}^*$ wie folgt definiert:

   $$I_k \sim I_l \;:\Leftrightarrow\; (I_k \subseteq \overline{\overline{I_l}} \;\wedge\; I_l \subseteq \overline{\overline{I_k}}) \quad .$$

   Wenn die Menge der Äquivalenzklassen von $\sim$ mit

   $$[\mathfrak{I}^*]_\sim \;=:\; \{\mathfrak{I}_1, \mathfrak{I}_2, \dots\}$$

   bezeichnet wird, dann gilt:

   $$\mathfrak{Q} \;=\; \left\{\bigcup \mathfrak{I}_1, \bigcup \mathfrak{I}_2, \dots\right\} \quad .$$

4.3.3 Segmentierungs-Strategien

Auf der Basis der berechneten Quadermenge $\mathfrak{Q}$ kann nun eine geeignete Segmentierung festgelegt werden, wobei ein Ausgleich zwischen Compilat-Größe und Datenlokalität gefunden werden muß.

Eine naheliegende Möglichkeit der Segmentierung ist die Wahl

$$\mathfrak{S} \;:=\; \mathfrak{Q} \quad ,$$

d. h. die Quader werden als Segmente verwendet. Dies hat den Vorteil, daß die Compilat-Größe minimiert wird, da jedes Segment eine einheitliche Rasterung aufweist. Allerdings ist es aus Gründen der Datenlokalität sinnvoll, möglichst große Segmente zu wählen. Quader mit zumindest annähernd übereinstimmender Rasterung sollten also zu einem Segment zusammengefaßt werden. Im Extremfall führt dies zu einem einzigen Segment (triviale Segmentierung)

$$\mathfrak{S} \;:=\; \left\{\bigcup \mathfrak{Q}\right\} \;=\; \left\{\bigcup \mathfrak{G}\right\}$$

und zu maximaler Datenlokalität, die unter Umständen durch einen größeren Umfang und komplizierteren Aufbau des compilierten Codes erkauft wird.

Über die genaue Auswirkung der verschiedenen Segmentierungen auf die Laufzeit eines Compilats gibt es bisher nur wenig Erkenntnisse. Insbesondere ist noch kein Algorithmus bekannt, der für ein gegebenes `with`-Konstrukt die optimale Segmentierung bestimmt. Daher verfügt der SAC-Compiler über eine sehr variable Schnittstelle, mit deren Hilfe sich verschiedenste Segmentierungsvorgaben und -strategien verwirklichen und auf ihre Eignung testen lassen (siehe Abschnitt 4.7).

```
A = ...;
B = with ([0,0] <= iv=[iv0,iv1] < [100,100] step [1,1] width [1,1]):
      A[[iv1,iv0]]
    modarray( A);
```

Abbildung 4.7: With*-Konstrukt mit nicht-affinem Array-Zugriff.*

4.4 Loop Blocking

Indem in der kanonischen Reihenfolge über die Segmente des with-Konstruktes iteriert wird, läßt sich in vielen Fällen eine gute Datenlokalität erreichen. Problematisch bleiben aber all diejenigen Fälle, in denen ein Zielausdruck Array-Zugriffe enthält, die nicht-affin von der Iterationsvariablen abhängen. Ein Beispiel dafür ist das with-Konstrukt in Abbildung 4.7, das die Transponierte der Matrix A berechnet. Der Zugriff auf das Array A erfolgt hier nicht mit Hilfe von iv und einem konstanten Offset, vielmehr werden die Komponenten von iv vertauscht. Dadurch ist es prinzipiell unmöglich, die Schreibzugriffe auf B und die Lesezugriffe auf A gemeinsam mit der Schrittweite 1 stattfinden zu lassen.

Um dennoch möglichst gute Laufzeiten zu erhalten, wird es in diesen Fällen nicht bei der kanonischen Iterationsreihenfolge belassen, sondern zusätzlich die Transformation Loop Blocking (siehe Abschnitt 3.3.4) angewendet, damit auch die nicht-affinen Array-Zugriffe das Cache besser nutzen können. Es können dabei für jede Dimension individuelle Blocking-Faktoren vergeben werden. (Ein kompletter Satz von Blocking-Faktoren sei Blocking-Vektor genannt.) Desweiteren muß das Blocking nicht auf eine Ebene beschränkt bleiben, sondern kann hierarchisch sein (maximal drei Level) — das Blocking ist dann durch eine Liste von bis zu drei Blocking-Vektoren beschrieben.

4.5 Loop Unrolling-Blocking

Insbesondere im Zusammenhang mit hierarchischem Blocking haben die Komponenten des innersten Blocking-Vektors unter Umständen sehr kleine Werte. In diesem Fall kann es sinnvoll sein, die Rümpfe der innersten Blocking-Schleifen abzurollen, um die Schleifen einzusparen. Diese Kombination aus Loop Blocking und anschließendem Unrolling wird Unrolling-Blocking genannt.

Abbildung 4.8 demonstriert diese Vorgehensweise für das Beispiel der Matrix-Transponierung. Es ist entsprechender C-Code einmal ohne Blocking, einmal mit Blocking (Blocking-Vektor $(2, 2)$) und einmal mit Unrolling-Blocking (ebenfalls $(2, 2)$) aufgeführt.

```
A = ...;
B = with ([0,0] <= iv=[iv0,iv1] < [100,100]): A[[iv1,iv0]]
    modarray( A);
```

(a) SAC*-Code.*

```
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 100; iv1++) {
    B[iv] = A[[iv1,iv0]];
  }
}
```

(b) Compilat ohne Blocking.

```
for (iv0 = 0; iv0 < 100; ) {
  bstart0 = iv0; bstop0 = iv0 + 2;
  for (iv1 = 0; iv1 < 100; ) {
    bstart1 = iv1; bstop1 = iv1 + 2;
    for (iv0 = bstart0; iv0 < bstop0; iv0++) {
      for (iv1 = bstart1; iv1 < bstop1; iv1++) {
        B[iv] = A[[i1,i0]];
      }
    }
  }
}
```

(c) Compilat mit Blocking.

```
for (iv0 = 0; iv0 < 100; ) {
  bstart0 = iv0;
  for (iv1 = 0; iv1 < 100; ) {
    bstart1 = iv1;
    iv0 = bstart0;
      iv1 = bstart1;
      B[iv] = A[[i1,i0]]; iv1++;
      B[iv] = A[[i1,i0]]; iv1++;
    iv0++;
      iv1 = bstart1;
      B[iv] = A[[i1,i0]]; iv1++;
      B[iv] = A[[i1,i0]]; iv1++;
    iv0++;
  }
}
```

(d) Compilat mit Unrolling-Blocking.

Abbildung 4.8: `With`*-Konstrukt mit und ohne Blocking bzw. Unrolling-Blocking compiliert.*

4.6 Arrayzugriffe

In den bisher aufgeführten Compilat-Fragmenten wurden die Array-Zugriffe immer in der Form

$$A\texttt{[}iv\texttt{]}$$

notiert, wobei A das Array und iv der Zugriffsvektor ist. Konkret bedeutet ein solcher Arrayzugriff die Selektion des mit iv korrespondierenden Elementes des Datenvektors von A. Unter Verwendung der Bezeichner A_{Data} für den Datenvektor, sh für den Formvektor und n für die Dimension von A, impliziert dies also einen Arrayzugriff der Form

$$A_{Data}\left[\ \sum_{j=0}^{n-1}\left(iv_j \cdot \prod_{k=j+1}^{n-1} sh_k\right)\right] \quad .$$

Für Arrayzugriffe innerhalb eines with-Konstruktes, die von der Generatorvariablen abhängen, kann die Position innerhalb des Datenvektors nicht statisch berechnet werden. Da es sehr teuer wäre, im Compilat diese Position bei jeder Iteration neuzuberechnen, wird sie in einer Offsetvariablen off nachgehalten. Bei der Pflege dieses Offsets kann ausgenutzt werden, daß die Iteration über die Indexvektoren innerhalb eines Segmentes bzw. eines Blockes in der kanonischen Reihenfolge erfolgt und daher beim Übergang zur nächsten Indexposition häufig nur ein Inkrement notwendig ist. Lediglich an den Rändern von Segmenten und Blöcken sind unter Umständen geringfügig aufwendigere Operationen notwendig, um off auf den aktuellen Wert zu setzen.

Die Vorgehensweise wird in Abbildung 4.9 an Hand eines Beispiels verdeutlicht, wobei die triviale Segmentierung und kein Blocking verwendet wird.

4.7 Pragmas

Eine wichtige Folgerung aus dem bisher Gesagten ist, daß für die Compilation eines with-Konstruktes in effizient ausführbaren Code einige Parameter geeignet gewählt werden müssen. Im einzelnen sind dies:

- eine Segmentierung;
- für jedes Segment
  - eine Liste von Blocking-Vektoren;
  - ein Unrolling-Blocking-Vektor.

Eine konkrete Ausprägung dieser Parameter werde als Konfiguration eines with-Konstruktes bezeichnet. Ziel muß es sein, das der Compiler für ein auftretendes with-Konstrukt die geeignete Konfiguration selbständig an Hand des Quellcodes inferiert. Mit dem aktuellen Wissensstand ist dies jedoch noch nicht möglich. Daher verwendet der Compiler standardmäßig immer die folgende triviale Konfiguration:

```
A = ...;
B = ...;
C = with ([0, 0] <= iv < [100, 50]): A[iv]
         ([0,50] <= iv < [100,100]): B[iv+[2,4]]
    genarray( [100,100]);
```

(a) SAC-Code.

```
A = ...;
B = ...;
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 50; iv1++) {
    off = 100*iv0 + iv1;
    C_data[off] = A_data[off];
  }
  for (iv1 = 50; iv1 < 100; iv1++) {
    off = 100*iv0 + iv1;
    off_2_4 = off + (100*2 + 4);
    C_data[off] = B_data[off_2_4];
  }
}
```

(b) Compilat mit naiver Offset-Berechnung.

```
A = ...; B = ...;
off = 0;
for (iv0 = 0; iv0 < 100; iv0++) {
  for (iv1 = 0; iv1 < 50; iv1++) {
    C_data[off] = A_data[off];
    off++;
  }
  for (iv1 = 50; iv1 < 100; iv1++) {
    off_2_4 = off + (100*2 + 4);
    C_data[off] = B_data[off_2_4];
    off++;
  }
}
```

(c) Compilat mit effizienter Offset-Berechnung.

Abbildung 4.9: Beispiel für die Compilation von Array-Zugriffen.

- triviale Segmentierung;
- kein Blocking, kein Unrolling-Blocking.

Davon abweichende Vorgaben können innerhalb des Quellcodes mit Hilfe eines Pragmas annotiert werden.

Jedes Pragma spezifiziert eine konkrete Konfiguration bzw. eine Strategie zur Gewinnung einer solchen und kann in einem SAC-Programm auf zwei Arten eingesetzt werden: entweder mit lokalem, oder mit globalem Gültigkeitsbereich. Lokale Pragmas entfalten ihre Wirkung nur für ein einzelnes `with`-Konstrukt und stehen direkt vor dem Schlüsselwort `with`. Globale Pragmas werden vor eine Funktion gestellt und gelten für alle `with`-Konstrukte ohne lokalem Pragma, die in dem nachfolgenden Programmtext, aber vor dem nächsten globalen Pragma, auftreten.

Die Syntax des Pragmas in BNF zeigt Abbildung 4.10. Das Pragma besteht aus dem Schlüsselwort `wlcomp`[3] gefolgt von einer Konfigurations-Spezifikation, die sich aus einer Schachtelung von Funktionsanwendungen (in der Syntaxdefinition als $ConfFun$ bezeichnet) zusammensetzt. Jede dieser Funktionen konsumiert eine Konfiguration und eine variable Anzahl

[3] SAC kennt im Zusammenhang mit dem Modul- und Klassensystem noch weitere Pragmas, siehe [Grel96, Abschnitt 5.2].

$$
\begin{array}{lcl}
\mathit{Pragma} & \Rightarrow & \texttt{\#pragma wlcomp } \mathit{Conf} \\
\mathit{Conf} & \Rightarrow & \mathit{ConfFun} \texttt{ (} \mathit{ConfArgs} \ \mathit{Conf} \texttt{)} \\
 & | & \texttt{Default} \\
\mathit{ConfArgs} & \Rightarrow & \mathit{Expr} \texttt{ , } \mathit{ConfArgs} \\
 & | & \varepsilon
\end{array}
$$

Abbildung 4.10: Syntax des `wlcomp`*-Pragmas.*

weiterer Argumente, um daraus eine neue Konfiguration zu generieren. Die Standardkonfiguration wird dabei durch das Symbol `Default` repräsentiert.

Bisher sind für das Pragma die folgenden Funktionen implementiert:

- `All(` conf`)`:

  liefert die Konfiguration mit trivialer Segmentierung, sowie deaktiviertem Blocking und Unrolling-Blocking — entspricht also dem derzeitigen `Default`;

- `Cubes(` conf`)`:

  liefert die Konfiguration mit den Quadern als Segmenten, sowie deaktiviertem Blocking und Unrolling-Blocking;

- `ConstSegs(` a_1`,` b_1`,` $\ldots$`,` a_M`,` b_M`,` conf`)`:

  es werden die Segmente mit den Umrissen $[\, a_1;\, b_1\,)$, $\ldots$, $[\, a_M;\, b_M\,)$ verwendet, Blocking und Unrolling-Blocking werden deaktiviert;

- `BvL0(` bv_1`,` $\ldots$`,` bv_M`,` conf`)`:

  Diese Funktion liefert die Konfiguration conf mit neuen Blocking-Vektoren bezüglich Level 0 (äußerer Block) für alle Segmente. Die Blocking-Vektoren der ersten $(M-1)$ Segmente werden auf die Werte $bv_1, \ldots, bv_{M-1}$ gesetzt, der Parameter bv_M gilt für *alle weiteren*[4] Segmente. (Eine Ordnung der Segmente ist entweder durch ein vorgeschaltetes `ConstSegs` gegeben, oder läßt sich aus den Segmentgrenzen ablesen.[5]) Ist M größer als die Anzahl der Segmente, so werden die überschüssigen Argumente ignoriert. Falls für einige Segmente kein Blocking erwünscht ist, kann dies mit Hilfe des 1-Vektors spezifiziert werden.

- `BvL1(` bv_1`,` $\ldots$`,` bv_M`,` conf`)`:

  setzt analog zu `BvL0` neue Blocking-Vektoren für Level 1 (mittlerer Block);

[4] Dies ermöglicht die einfache Spezifikation eines einheitlichen Blocking auch in Fällen, in denen die Segmentanzahl sehr groß oder unbekannt ist.

[5] Für zwei Segmente S_1 und S_2 mit $\overline{S_1} = [\, a_1;\, b_1\,)$ und $\overline{S_2} = [\, a_2;\, b_2\,)$ gelte:

$$S_1 \leq S_2 \ :\Leftrightarrow\ (S_1 = S_2) \ \vee\ (\exists k : (\forall j \in \{0, \ldots, k-1\} : a_{1,j} = a_{2,j}) \wedge (a_{1,k} < a_{2,k})) \quad .$$

Dann ist über den Segmenten mit $\leq$ eine Ordnungsrelation definiert.

- `BvL2(` bv_1`, ...,` bv_M`,` $conf$`)`:

  setzt analog zu `BvL0` neue Blocking-Vektoren für Level 2 (innerer Block);

- `Ubv(` ubv_1`, ...,` ubv_M`,` $conf$`)`:

  setzt analog zu `BvL0` neue Unrolling-Blocking-Vektoren;

Da die Auswertung der `wlcomp`-Pragmas in einem gekapselten Modul des Compilers erfolgt, kann diese Liste ohne großen Aufwand um weitere Funktionen ergänzt werden.

Zwei Anwendungsbeispiele sollen den Gebrauch des `wlcomp`-Pragmas verdeutlichen. Das Pragma

```
#pragma wlcomp Default
```

steht für die triviale Konfiguration und kann etwa dazu eingesetzt werden, ein voranstehendes globales Pragma aufzuheben. Um eine Konfiguration mit drei zweidimensionalen Segmenten zu definieren, von denen nur das erste geblockt, aber alle mit Unrolling-Blocking compiliert werden sollen, könnte das folgende Pragma verwendet werden:

```
#pragma wlcomp Ubv( [2,2], BvL0( [50,50], [1,1],
                ConstSegs( [  0,  0], [500,250],
                           [  0,250], [250,500],
                           [250,250], [500,500], Default)))   .
```

Kapitel 5

Implementation

Aufbauend auf den Erkenntnissen des vorherigen Kapitels wird nun das Compilationsschema für das `with`-Konstrukt im Detail vorgestellt. Um die semantische Lücke zwischen SAC und der Zielsprache C zu verringern, erfolgt die Compilation in mehreren Stufen. Zuerst wird das `with`-Konstrukt in eine Zwischendarstellung transformiert, in der die Generatormengen in ihre Komponenten-Projektionen zerlegt sind. Auf dieser Zwischendarstellung werden dann die in Kapitel 4 entwickelten Transformationen ausgeführt (Segmentierung, Blocking, kanonische Reihenfolge, ...). In der abschließenden Code-Erzeugungsphase wird aus der Zwischendarstellung C-Code mit sogenannten Intermediate Code Macros (ICMs) generiert. Diese ICMs werden dann implizit während der Übersetzung durch den C-Compiler expandiert.

5.1 Zwischendarstellung

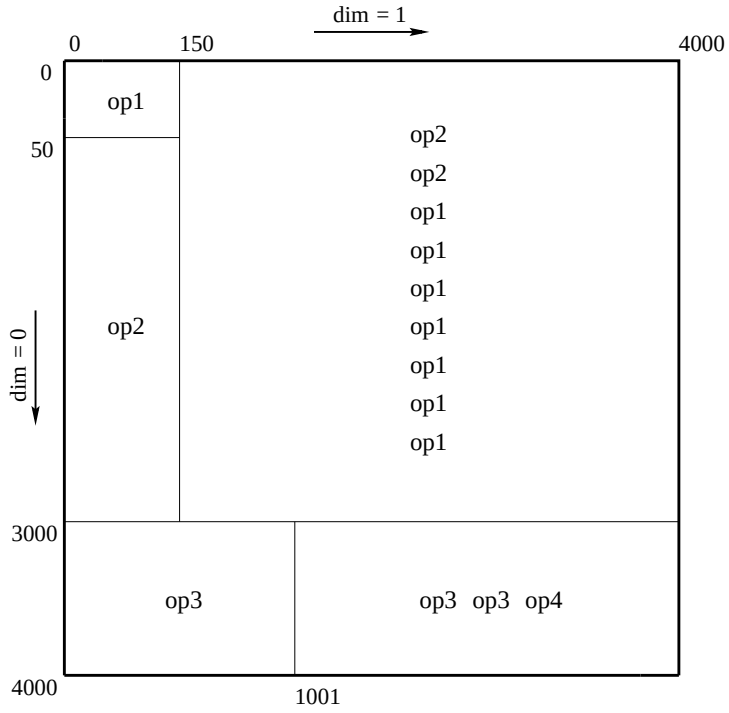
Das Compilations-Schema soll transparent und leicht zu erweitern sein, daher wird die Einführung einer Zwischendarstellung nötig, die näher am zu erzeugenden Code liegt. Auf Grundlage dieser Zwischendarstellung werden dann Transformationen zu definieren sein, die die in Kapitel 4 vorgestellten Optimierungen realisieren.

Als Anwendungsbeispiel soll in diesem Abschnitt das `with`-Konstrukt in Abbildung 5.1(a) dienen. Um das Verständnis zu erleichtern, wird jeder Transformationsschritt an Hand dieses `with`-Konstruktes nachvollzogen. Abbildung 5.1(b) zeigt das Layout der Indexvektormenge, wobei in den einzelnen Bereichen jeweils nur das sich periodisch wiederholende Muster aufgeführt ist. Der Bereich oben rechts besitzt beispielsweise eine Rasterung in Richtung der Achse $\mathrm{dim} = 0$ mit der Periode 9, wobei sich eine Periode aus zwei Indexpositionen mit dem Zielausdruck `op2` und sieben Indexpositionen mit `op1` zusammensetzt.

Die Idee der Zwischendarstellung ist es, die Generatormengen nicht mehr als Ganzes zu betrachten, sondern in ihre Komponenten-Projektionen zu zerlegen, um anschließend die verschiedenen Dimensionen getrennt manipulieren zu können. Im Wesentlichen ist dies die Vorgehensweise, die auch in Abschnitt 4.1 bei der naiven Compilation angewendet wurde. Die Zwischendarstellung dient nur der Manipulation der Generatoren, der Operator des `with`-Konstruktes bleibt unverändert und braucht daher nicht betrachtet zu werden.

```
A = with ([   0,   0] <= iv < [  50, 150] step [1,1] width [1,1]): op1
         ([   0, 150] <= iv < [3000,4000] step [9,1] width [2,1]): op2
         ([   2, 150] <= iv < [3000,4000] step [9,1] width [7,1]): op1
         ([  50,   0] <= iv < [3000, 150] step [1,1] width [1,1]): op2
         ([3000,   0] <= iv < [4000,1001] step [1,1] width [1,1]): op3
         ([3000,1001] <= iv < [4000,4000] step [1,3] width [1,1]): op3
         ([3000,1002] <= iv < [4000,4000] step [1,3] width [1,2]): op4
    genarray( [4000,4000]);
```

(a) SAC*-Code.*



(b) Diagramm der Indexvektormenge.

Abbildung 5.1: Beispiel eines zu compilierenden `with`*-Konstruktes.*

```
  0→  50, step[0] 1
                0→1:    0→ 150, step[1] 1
                                      0→1: op1
  0→3000, step[0] 9
                0→2:  150→4000, step[1] 1
                                      0→1: op2
  2→3000, step[0] 9
                0→7:  150→4000, step[1] 1
                                      0→1: op1
 50→3000, step[0] 1
                0→1:    0→ 150, step[1] 1
                                      0→1: op2
3000→4000, step[0] 1
                0→1:    0→1001, step[1] 1
                                      0→1: op3
3000→4000, step[0] 1
                0→1: 1001→4000, step[1] 3
                                      0→1: op3
3000→4000, step[0] 1
                0→1: 1002→4000, step[1] 3
                                      0→2: op4
```

Abbildung 5.2: Die Generatoren des Beispiels nach Anwendung der Transformation $\mathcal{T}_{Interm}$.

Die Überführung der Generatoren eines `with`-Konstruktes in die Zwischendarstellung geschieht mit Hilfe der in Abbildung A.1 definierten Transformation $\mathcal{T}_{Interm}$.[1]Nach Anwendung dieser Transformation falten sich die Generatoren des Beispiels in den in Abbildung 5.2 dargestellten Baum auf.

5.1.1 Berechnung der Quader

Nachdem die Generatoren des `with`-Konstruktes in die Zwischendarstellung überführt worden sind, werden sie analog des in Abschnitt 4.3.2 angegebenen Algorithmus in die Quader-Darstellung transformiert — diese Transformation sei $\mathcal{T}_{Cubes}$ genannt.[2] Für das Beispiel führt dies zu dem in Abbildung 5.3 aufgeführten Ergebnis. In der Abbildung sind die einzelnen Quader zur besseren Übersicht durch waagerechte Linien voneinander getrennt.

[1]Aus Platzgründen sind die formalen Spezifikationen dieser und aller nachfolgenden Transformationen im Anhang aufgeführt.

[2]Der in Abschnitt 4.3.2 spezifizierte Algorithmus zur Quaderberechnung arbeitet mit speziellen Indexvektormengen, die durch vier Parameter a, b, s, w charakterisiert sind. Da jede dieser Indexvektormengen eine Repräsentation in der Zwischendarstellung besitzt (und umgekehrt), läßt sich der Algorithmus leicht auf die Zwischendarstellung übertragen. Daher kann hier auf eine exakte Spezifikation von $\mathcal{T}_{Cubes}$ verzichtet werden.

```
  0→  50, step[0] 1
                0→1:    0→ 150, step[1] 1
                                      0→1: op1
-------------------------------------------------
  0→3000, step[0] 9
                0→2:  150→4000, step[1] 1
                                      0→1: op2
                2→9:  150→4000, step[1] 1
                                      0→1: op1
-------------------------------------------------
 50→3000, step[0] 1
                0→1:    0→ 150, step[1] 1
                                      0→1: op2
-------------------------------------------------
3000→4000, step[0] 1
                0→1:    0→1001, step[1] 1
                                      0→1: op3
-------------------------------------------------
3000→4000, step[0] 1
                0→1: 1001→4000, step[1] 3
                                      0→1: op3
                                      1→3: op4
```

Abbildung 5.3: Das Beispiel nach Anwendung von $\mathcal{T}_{Cubes}$.

5.1.2 Wahl der Segmente

Im nächsten Schritt wird eine Segmentierung $\mathcal{S} = \{S_1, \ldots, S_M\}$ bestimmt. Die Compilation wird dann auf jedem Segment getrennt fortgeführt und am Ende der Code-Erzeugung werden die für die einzelnen Segmente erhaltenen Code-Fragmente hintereinander gesetzt.

Die Selektion des zu einem Segment S_l gehörigen Teilbaumes aus der Zwischendarstellung T erfolgt durch die in Abbildung A.2 definierte Transformation $\mathcal{T}_{Seg}$. In der Regel werden sich die Segmente durch Mengenvereinigung aus den Quadern ableiten, in diesem Fall macht $\mathcal{T}_{Seg}[\![S_l, T]\!]$ nichts anderes, als die nicht zu S_l gehörenden Quader aus T zu entfernen. Für das Beispiel bedeutet dies, daß der Baum an einigen der in Abbildung 5.3 markierten Stellen gekürzt wird. In dem konkreten Fall sei jedoch die triviale Segmentierung gewählt, so daß die nachfolgenden Transformationen weiterhin auf dem vollständigen Baum operieren.

Für jedes Segment sind eine Liste von Blocking-Vektoren $\langle bv_0, bv_1, bv_2 \rangle$ und ein Unrolling-Blocking-Vektor ubv zu wählen. Bei der Festlegung dieser Werte müssen einige Einschränkungen beachtet werden:

- Falls in einer Dimension j Unrolling-Blocking durchgeführt werden soll — d. h. es wurde $ubv_j > 1$ gewählt —, ist ubv_j so zu dimensionieren, daß stets vollständige Rasterperioden abgerollt werden können, da sonst die Periodizität verloren geht. Daher muß ubv_j ein Vielfaches aller Schrittweiten sein, die innerhalb des Segmentes entlang der Achse $\mathrm{dim} = j$ auftreten.

Bei Bedarf ist es möglich, daß das Unrolling-Blocking nicht schon in der 0-ten, sondern erst in hinteren Dimensionen einsetzt. Es kann also $ubv_0 = ubv_1 = \ldots = 1$ gewählt werden. Sobald aber in einer Dimension k $ubv_k > 1$ gesetzt wurde, muß auch in allen nachfolgenden Dimensionen ein Unrolling-Blocking erfolgen.

Zusammenfassend muß ubv folgender Bedingung genügen, wobei sv das kgV aller innerhalb des Segmentes auftretener Schrittweiten bezeichne:

$$\exists k : (\forall j < k : ubv_j = 1) \wedge (\forall j \geq k : sv_j \mid ubv_j) \quad .$$

Für das Beispiel berechnet sich sv zu

$$sv = (\mathrm{kgV}(1,9), \mathrm{kgV}(1,3)) = (9,3) \quad ,$$

d. h. legale Werte für ubv sind dort alle Vielfachen von $(9,3)$, sowie $(1,\ldots\cdot 3)$ und $(1,1)$.

- Für jeden der drei Blocking-Vektoren bv_0, bv_1, bv_2 muß gelten:

$$\exists k : (\forall j < k : bv_{.,j} = 1) \wedge (\forall j \geq k : bv_{.,j} \geq \max(sv_j, ubv_j)) \quad .$$

Analog zum Unrolling-Blocking muß also auch das Blocking nicht schon in der 0-ten Dimension beginnen. Falls geblockt wird, muß in dem Block mindestens eine abgerollte Rasterperiode ($\max(sv_j, ubv_j)$) Platz finden. Im Gegensatz zum Unrolling-Blocking wird dabei jedoch *nicht* gefordert, daß das Blockende mit einer vollständigen Rasterperiode abschließt, da dies automatisch durch eine spezielle Transformation sichergestellt wird (siehe Abschnitt 5.1.8).

Ferner müssen die Blocking-Vektoren untereinander der Beziehung

$$bv_0 \geq bv_1 \geq bv_2$$

genügen, da ein innerer Block natürlich nicht größer als ein äußerer sein kann.

Für das Beispiel werden nachfolgend die beiden folgenden Konfigurationen verwendet:

- $\langle bv_0 = (180,158), bv_1 = (1,1), bv_2 = (1,1)\rangle$, $ubv = (1,6)$;
- $\langle bv_0 = (1,158), bv_1 = (1,1), bv_2 = (1,1)\rangle$, $ubv = (1,6)$.

5.1.3 Splitting

Als Vorbereitung auf das spätere Zusammenfassen identischer Projektionen (siehe Abschnitt 5.1.6) werden nach der Segmentierung — in Analogie zum Loop Peeling (siehe Abschnitt 3.3.3) — alle sich paarweise überlappenden Projektionen aufgetrennt. Dieser als Splitting bezeichnete Vorgang wird durch die in Abbildung A.3 angedeutete Transformation $\mathcal{T}_{Split}$ ausgeführt:

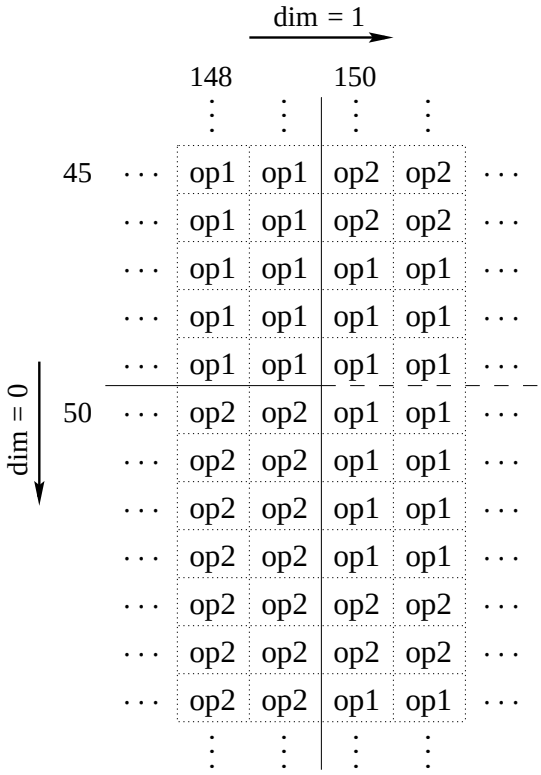


Abbildung 5.4: Detail aus dem Indexvektormengen-Diagramm des Beispiels.

Zuerst wird das Splitting in der 0-ten Dimension durchgeführt. Die Teilbäume werden dabei entlang der Achse $\dim = 0$ so lange zerteilt, bis die Projektionen ihrer Grenzen in der 0-ten Dimension paarweise disjunkt oder identisch sind. Anschließend lassen sich Gruppen mit jeweils identischen 0-Projektionen bilden. Auf jeder dieser Gruppen wird dann das Splitting in der 1-ten Dimension fortgesetzt, usw.

Bei der Ausführung des Splittings ist zu beachten, daß sich nach dem Auftrennen einer Projektion unter Umständen die Rasterung verschiebt. (In der Spezifikation von $\mathcal{T}_{Split}$ in Abbildung A.3 wird dies durch den Übergang von R_1, R_2 zu R'_1, R'_2 angedeutet.) Dieses Phänomen läßt sich gut an Hand des Beispiels verdeutlichen. Abbildung 5.4 zeigt einen vergrößerten Ausschnitt des Indexvektormengen-Diagramms. Die Quadergrenzen sind mit durchgezogenen Linien markiert; die gestrichelte Linie gibt die Position an, an der der rechte Quader aufgetrennt werden muß. Es fällt auf, daß die Trennstelle im Inneren einer Rasterperiode liegt, sich daher die Rasterung im unteren abgetrennten Teil entsprechend verschiebt. Somit ergibt sich für das Beispiel nach dem Splitting der in Abbildung 5.5 darstellte Baum.

Wie bereits erwähnt wurde, ist das Splitting eine Vorbereitung für das Merging. Es erscheint auf den ersten Blick sinnvoll, Splitting und Merging dimensionsweise verschränkt durchzuführen. Die nach dem Splitting in einer Dimension gebildeten Gruppen stellen ja genau die Projektionen dar, die später beim Merging vereinigt werden. Allerdings gibt es gute Gründe die beiden Phasen strikt zu trennen: Das Merging kann auf jeden Fall erst nach dem Blocking (siehe nächsten Abschnitt) durchgeführt werden, das Splitting findet jedoch am besten vor dem Blocking statt, da sich durch das Splitting im allgemeinen die Raster verschieben

```
 0→  50, step[0] 1
                0→1:    0→ 150, step[1] 1
                                        0→1: op1
 0→  50, step[0] 9
                0→2:  150→4000, step[1] 1
                                        0→1: op2
                2→9:  150→4000, step[1] 1
                                        0→1: op1
50→3000, step[0] 1
                0→1:    0→ 150, step[1] 1
                                        0→1: op2
50→3000, step[0] 9
                0→4:  150→4000, step[1] 1
                                        0→1: op1
                4→6:  150→4000, step[1] 1
                                        0→1: op2
                6→9:  150→4000, step[1] 1
                                        0→1: op1
3000→4000, step[0] 1
                0→1:    0→1001, step[1] 1
                                        0→1: op3
3000→4000, step[0] 1
                0→1: 1001→4000, step[1] 3
                                        0→1: op3
                                        1→3: op4
```

Abbildung 5.5: Das Beispiel nach Anwendung von $\mathcal{T}_{Split}$.

— sich also der Inhalt eines Blockes noch ändern würde!

5.1.4 Blocking

Die nächste Transformationsphase führt das Blocking durch. Der Baum wird dabei um Elemente ergänzt, die ähnlich wie die Quaderprojektionen aufgebaut sind. Anstelle des Schlüsselwortes `step` wird je nach Blocking-Level `block0`, `block1` bzw. `block2` verwendet.

Um etwa in dem gegebenen Beispiel den letzten Quader mit $bv_0 = (180, 158)$ zu blocken, muß der folgende Baum gebildet werden:

```
3000→4000, block0[0] 180:
  1001→4000, block0[1] 158:
    /*
     * Dies ist das Koordinatensystem für die einzelnen Blöcke.
     * Hier muß noch definiert werden, was innerhalb eines Blockes
     * passieren soll.
     */                                                          .
```

An den Blättern dieses Blocking-Baumes können jetzt die Beschreibungen für die Blockinhalte eingefügt werden:

```
3000→4000, block0[0] 180:
  1001→4000, block0[1] 158:
                  0→180, step[0] 1
                              0→1: 0→156, step[1] 3
                                                  0→1: op3
                                                  1→3: op4   .
```

Diese Vorgehensweise läßt sich auf hierarchisches Blocking ausweiten. Mit $bv_1 = (40, 50)$ würde sich beispielsweise ergeben:

```
3000→4000, block0[0] 180:
  1001→4000, block0[1] 158:
                  0→180, block1[0] 40:
                    0→158, block1[1] 50:
                                0→40, step[0] 1
                                           0→1: 0→50, step[1] 3
                                                              0→1: op3
                                                              1→3: op4 .
```

Soll in allen oder einigen Dimensionen kein Blocking stattfinden, läßt sich dies — zumindest im Fall $sv_j > 1$ — wegen der vorhandenen Raster nicht wie oben und einer Blocking-Schrittweite von 1 erreichen, da in den entsprechenden Zweigen des Baumes eine komplette Rasterperiode keinen Platz mehr finden würde. Daher wird stattdessen so vorgegangen, daß für die entsprechende Dimension kein `block?[`j`]`-Zweig gebildet wird, sondern sofort der `step[`j`]`-Zweig im Baum erscheint. Dem kann sich dann die Blocking-Hierarchie für die nachfolgenden Dimensionen anschließen. Also etwa im Beispiel mit $bv_0 = (1, 156)$:

```
3000→4000, step[0] 1
                0→1: 1001→4000, block0[1] 156:
                                       0→156, step[1] 3
                                                   0→1: op3
                                                   1→3: op4   .
```

Entsprechend dieser Systematik läuft das Blocking nach dem in Abbildung A.4 angegebenen Transformationsschema $\mathcal{T}_{Block}$ ab. $\mathcal{T}_{Block}$ erwartet als Parameter neben dem zu transformierenden Baum einen Blocking-Vektor bv. Hierarchisches Blocking mit den Blocking-Vektoren $\langle bv_0, bv_1, bv_2 \rangle$ wird durch Hintereinanderausführung von $\mathcal{T}_{Block}$ erreicht:

$$\mathcal{T}_{Block}\,[\![bv_2, \mathcal{T}_{Block}\,[\![bv_1, \mathcal{T}_{Block}\,[\![bv_0, \ldots]\!]]\!]\!] \quad .$$

Für das Beispiel ergeben sich nach dem Blocking mit $bv_0 = (180, 158)$ bzw. $bv_0 = (1, 158)$ die in den Abbildungen 5.6 und 5.7 aufgeführten Resultate. Bei beiden Baumstrukturen fällt ins Auge, daß das Blocking eine Konstruktion der Art

```
0→150, block0[1] 158: ...
```

nicht sofort zu

```
0→150, block0[1] 150: ...
```

vereinfacht, oder gar völlig entfernt. Dies liegt darin begründet, daß diese Informationen für die spätere Optimierung noch benötigt werden (siehe Abschnitt 5.1.7).

5.1.5 Unrolling-Blocking

Im Anschluß an das Blocking wird ein Unrolling-Blocking durchgeführt. Die dafür zuständige Transformation

$$\mathcal{T}_{UBlock}\,[\![ubv, \ldots]\!]$$

arbeitet analog zu $\mathcal{T}_{Block}$, verwendet allerdings als Schlüsselwort `ublock` statt `block?`. Das Unrolling wird explizit erst während der Code-Erzeugung ausgeführt (siehe Abschnitt 5.2).

Mit $ubv = (1, 6)$ ergeben sich für die beiden Beispiele nach Anwendung von $\mathcal{T}_{UBlock}$ die in den Abbildungen 5.8 und 5.9 gezeigten Zwischenergebnisse.

5.1.6 Merging

Die nächste Transformation wird Merging genannt und faßt — analog zu Loop Fusion (siehe Abschnitt 3.3.2) — Projektionen mit identischen Umrissen zusammen. Dabei müssen unter Umständen die einzelnen Rasterperiodenlängen durch kgV-Bildung angeglichen werden. Dank des vorher durchgeführten Splittings repräsentiert der Generatorbaum nach dieser Transformation $\mathcal{T}_{Merge}$ eine kanonische Iterationsreihenfolge. Die formale Spezifikation von $\mathcal{T}_{Merge}$ findet sich in Abbildung A.5.

Merging transformiert die beiden Beispiele in die in den Abbildungen 5.10 und 5.11 angegebenen Baumstrukturen.

5.1.7 Optimierung

In der anschließenden Optimierungsphase werden alle Projektionen mit aufeinanderfolgenden Indexvektor-Bereichen und identischen Unterbäumen zusammengefaßt. In dem für das Beispiel mit $bv_0 = (1, 158)$ erhaltenen Baum (siehe Abbildung 5.11) ist etwa die Konstellation

```
0→50 step[0] 9
          ...
          2→9:   0→ 150 block0[1] 158: R
               150→4000 block0[1] 158: R
```

enthalten. Da in den beiden letzten Zeilen die Schrittweite (158) und der Unterbaum (R) identisch sind, kann dies zu

```
0→50 step[0] 9
          ...
          2→9: 0→4000 block0[1] 158: R
```

vereinfacht werden. Man beachte, daß die eine der beiden Projektionen einen Blocking-Faktor besitzt, der größer als ihre Breite ist. Hier zeigt sich, daß durch diese auf den ersten Blick absurd erscheinende Konstruktion die Baumvereinfachung erheblich erleichtert wird.

Die Optimierung wird mit Hilfe der in Abbildung A.6 definierten Transformation $\mathcal{T}_{Opt}$ durchgeführt.

5.1.8 Fitting

In der vorletzten Transformationsphase, dem sogenannten Fitting, wird dafür gesorgt, daß in jedem Teilbaum und jeder Dimension sowohl die Differenz der äußersten Grenzen als auch alle Blocking-Faktoren ein Vielfaches der Anzahl abzurollender Elemente — also $\max(\texttt{ublock}, \texttt{step})$ — sind. Ersteres wird sichergestellt, indem unvollständige Perioden an den Enden abgetrennt werden, letzteres läßt sich erreichen, indem die Blocking-Faktoren bei Bedarf geringfügig modifiziert werden.

Das Abtrennen unvollständiger Perioden ermöglicht die Erzeugung erheblich effizienteren Codes. Es kann dann nämlich im Compilat zwischen den abgerollten Schleifendurchläufen — diese entstehen entweder explizit durch Unrolling-Blocking, oder implizit durch die Rasterung — auf einen 'Schleifenende'-Test verzichtet werden.

Das Anpassen der Blocking-Faktoren an die Anzahl abzurollender Elemente ist unumgänglich, um die Periodizität innerhalb des Blockes zu erhalten.

In dem Beispiel mit $bv_0 = (1, 158)$ wird in der 0-ten Dimension 0→50 in 0→45 und 45→50 zerlegt, da $(50 - 0)$ kein Vielfaches von 9 (= `step[0]`) ist, sondern vielmehr fünf Iterationen überhängen. In der 1-ten Dimension ist beispielsweise 150→4000 betroffen, da $(4000 - 150)$ kein Vielfaches von 6 (= `ublock[1]`) ist, sondern vier Iterationen überhängen. Außerdem wird der Blocking-Faktor `block0[1]` von 158 auf 156 geändert, da 156 das nächstgelegende Vielfache von 6 ist (siehe Abbildung 5.12).

Die für das Fitting zuständige Transformation $\mathcal{T}_{Fit}$ ist in Abbildung A.7 definiert.

5.1.9 Normalisierung

Als letzter Transformationsschritt wird in dem Baum eine Normalisierung vorgenommen ($\mathcal{T}_{Norm}$). In dieser Phase werden alle Projektionen bereinigt, deren Schrittweite größer als ihre Breite ist. Insbesondere werden in den durch $\mathcal{T}_{Fit}$ abgespaltenen unvollständigen Rasterperioden alle überflüssigen Teile entfernt. Abbildung 5.13 zeigt die Auswirkungen dieser Transformation auf das Beispiel mit $bv_0 = (1, 158)$.

```
0000→0050, block0[0] 180:
    0000→0150, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, step[1] 1
                                                0→1: op1
0000→0050, block0[0] 180:
    0150→4000, block0[1] 158:
                0→180, step[0] 9
                            0→2: 0→158, step[1] 1
                                                0→1: op2
                            2→9: 0→158, step[1] 1
                                                0→1: op1
0050→3000, block0[0] 180:
    0000→0150, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, step[1] 1
                                                0→1: op2
0050→3000, block0[0] 180:
    0150→4000, block0[1] 158:
                0→180, step[0] 9
                            0→4: 0→158, step[1] 1
                                                0→1: op1
                            4→6: 0→158, step[1] 1
                                                0→1: op2
                            6→9: 0→158, step[1] 1
                                                0→1: op1
3000→4000, block0[0] 180:
    0000→1001, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, step[1] 1
                                                0→1: op3
3000→4000, block0[0] 180:
    1001→4000, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, step[1] 3
                                                0→1: op3
                                                1→3: op4
```

Abbildung 5.6: Das Beispiel mit $bv_0 = (180, 158)$ nach Anwendung von $\mathcal{T}_{Block}$.

```
 0→  50, step[0] 1
              0→1:    0→150, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op1
 0→  50, step[0] 9
              0→2:  150→4000, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op2
              2→9:  150→4000, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op1
50→3000, step[0] 1
              0→1:    0→150, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op2
50→3000, step[0] 9
              0→4:  150→4000, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op1
              4→6:  150→4000, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op2
              6→9:  150→4000, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op1
3000→4000, step[0] 1
              0→1:    0→1001, block0[1] 158:
                                  0→158, step[1] 1
                                              0→1: op3
3000→4000, step[0] 1
              0→1: 1001→4000, block0[1] 158:
                                  0→158, step[1] 3
                                              0→1: op3
                                              1→3: op4
```

Abbildung 5.7: Das Beispiel mit $bv_0 = (1, 158)$ nach Anwendung von $\mathcal{T}_{Block}$.

```
0000→0050, block0[0] 180:
    0000→0150, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
0000→0050, block0[0] 180:
    0150→4000, block0[1] 158:
                0→180, step[0] 9
                            0→2: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op2
                            2→9: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
0050→3000, block0[0] 180:
    0000→0150, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op2
0050→3000, block0[0] 180:
    0150→4000, block0[1] 158:
                0→180, step[0] 9
                            0→4: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
                            4→6: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op2
                            6→9: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
3000→4000, block0[0] 180:
    0000→1001, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op3
3000→4000, block0[0] 180:
    1001→4000, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 3
                                                            0→1: op3
                                                            1→3: op4
```

Abbildung 5.8: Das Beispiel mit $bv_0 = (180, 158)$ und $ubv = (1, 6)$ nach Anwendung von $\mathcal{T}_{UBlock}$.

```
   0→  50, step[0] 1
               0→1:    0→ 150, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op1
   0→  50, step[0] 9
               0→2:  150→4000, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op2
               2→9:  150→4000, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op1
 50→3000, step[0] 1
               0→1:    0→ 150, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op2
 50→3000, step[0] 9
               0→4:  150→4000, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op1
               4→6:  150→4000, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op2
               6→9:  150→4000, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op1
3000→4000, step[0] 1
               0→1:    0→1001, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 1
                                                               0→1: op3
3000→4000, step[0] 1
               0→1: 1001→4000, block0[1] 158:
                                    0→158, ublock[1] 6:
                                                  0→6, step[1] 3
                                                               0→1: op3
                                                               1→3: op4
```

Abbildung 5.9: Das Beispiel mit $bv_0 = (1, 158)$ und $ubv = (1, 6)$ nach Anwendung von $\mathcal{T}_{UBlock}$.

```
0000→0050, block0[0] 180:
    0000→0150, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
    0150→4000, block0[1] 158:
                0→180, step[0] 9
                            0→2: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op2
                            2→9: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
0050→3000, block0[0] 180:
    0000→0150, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op2
    0150→4000, block0[1] 158:
                0→180, step[0] 9
                            0→4: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
                            4→6: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op2
                            6→9: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op1
3000→4000, block0[0] 180:
    0000→1001, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 1
                                                            0→1: op3
    1001→4000, block0[1] 158:
                0→180, step[0] 1
                            0→1: 0→158, ublock[1] 6:
                                                0→6, step[1] 3
                                                            0→1: op3
                                                            1→3: op4
```

Abbildung 5.10: Das Beispiel mit $bv_0 = (180, 158)$ nach Anwendung von $\mathcal{T}_{Merge}$.

```
    0→  50, step[0] 9
                0→2:    0→ 150, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op1
                      150→4000, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op2
                2→9:    0→ 150, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op1
                      150→4000, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op1
  50→3000, step[0] 9
                0→4:    0→ 150, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op2
                      150→4000, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op1
                4→6:    0→ 150, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op2
                      150→4000, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op2
                6→9:    0→ 150, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op2
                      150→4000, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op1
 3000→4000, step[0] 1
                0→1:    0→1001, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 1
                                                                    0→1: op3
                     1001→4000, block0[1] 158:
                                        0→158, ublock[1] 6:
                                                        0→6, step[1] 3
                                                                    0→1: op3
                                                                    1→3: op4
```

Abbildung 5.11: Das Beispiel mit $bv_0 = (1, 158)$ nach Anwendung von $\mathcal{T}_{Merge}$.

```
 0→45, step[0] 9
             0→2:    0→ 150, block0[1] 156:
                                  0→156, ublock[1] 6:
                                                 0→6, step[1] 1
                                                           0→1: op1
                    150→3996, block0[1] 156:
                                  0→156, ublock[1] 6:
                                                 0→6, step[1] 1
                                                           0→1: op2
                   3996→4000, block0[1] 156:
                                  0→156, ublock[1] 6:
                                                 0→6, step[1] 1
                                                           0→1: op2
             2→9: ...
45→50, step[0] 9
             0→2: ...
             2→9: ...
...
```

Abbildung 5.12: Das Beispiel mit $bv_0 = (1, 158)$ nach Anwendung von $\mathcal{T}_{Fit}$.

```
 0→45, step[0] 9
             0→2:    0→ 150, block0[1] 150:
                                  0→150, ublock[1] 6:
                                                 0→6, step[1] 1
                                                           0→1: op1
                    150→3996, block0[1] 156:
                                  0→156, ublock[1] 6:
                                                 0→6, step[1] 1
                                                           0→1: op2
                   3996→4000, block0[1]   4:
                                  0→  4, ublock[1] 4:
                                                 0→4, step[1] 1
                                                           0→1: op2
             2→9: ...
45→50, step[0] 5
             0→2: ...
             2→5: ...
...
```

Abbildung 5.13: Das Beispiel mit $bv_0 = (1, 158)$ nach Anwendung von $\mathcal{T}_{Norm}$.

Nachtrag: Im Zuge der Normalisierung kann die Situation eintreten, daß sich ein (unvollständiger) Unrolling-Block nicht mehr mit dem vorhandenen Raster verträgt, wie dies im Beispiel etwa in der rechten unteren Ecke des Arrays auftritt. Nach der (naiven) Normalisierung ergibt sich dort:

```
3000→4000, step[0] 1
                0→1: ...
                     3995→4000, block0[1] 5:
                                          0→5, ublock[1] 5:
                                                       0→5, step[1] 3
                                                                 0→1: op3
                                                                 1→3: op4   .
```

Nun ist aber `ublock[1]` ($= 5$) kein Vielfaches von `step[1]` ($= 3$) mehr. Daher muss an diesem Fall innerhalb des Unrolling-Blockes erneut ein Fitting durchgeführt werden:

```
3000→4000, step[0] 1
                0→1: ...
                     3995→4000, block0[1] 5:
                                          0→5, ublock[1] 5:
                                                       0→3, step[1] 3
                                                                 0→1: op3
                                                                 1→3: op4
                                                       3→5, step[1] 2
                                                                 0→1: op3
                                                                 1→2: op4   .
```

5.1.10 Zusammenfassung

Durch die vorgestellten Transformationen $\mathcal{T}_{Interm}, \ldots, \mathcal{T}_{Norm}$ werden die Generatoren G eines gegebenen `with`-Konstruktes von einer abstrakten mengenorientierten Darstellung in eine baumartige projektionsorientierte Darstellung überführt. Sei $\mathcal{S}^+$ die gewünschte Segmentierung $\{S_1, \ldots, S_M\}$ inklusive der Parameter $\langle bv_{0,l}, bv_{1,l}, bv_{2,l} \rangle$ und ubv_l für jedes Segment S_l, dann lassen sich die einzelnen Phasen wie folgt zu einer einzigen Transformation zusammenfassen:

$$\mathcal{T}^* \left[\!\left[\mathcal{S}^+, G \right]\!\right] \longmapsto \begin{cases} \mathcal{T}_{Norm} \left[\!\left[... \mathcal{T}_{UBlock} \left[\!\left[ubv_1 \ , ... \mathcal{T}_{Block} \left[\!\left[bv_{0,1} \ , \mathcal{T}_{Split} \left[\!\left[\mathcal{T}_{Seg} \left[\!\left[S_1 \ , T_{\mathbb{Q}} \right]\!\right] \right]\!\right] ... \right]\!\right] ... \right]\!\right] \\ \vdots \\ \mathcal{T}_{Norm} \left[\!\left[... \mathcal{T}_{UBlock} \left[\!\left[ubv_M, ... \mathcal{T}_{Block} \left[\!\left[bv_{0,M}, \mathcal{T}_{Split} \left[\!\left[\mathcal{T}_{Seg} \left[\!\left[S_M, T_{\mathbb{Q}} \right]\!\right] \right]\!\right] ... \right]\!\right] ... \right]\!\right], \\ \qquad\qquad \text{wobei } T_{\mathbb{Q}} := \mathcal{T}_{Cubes} \left[\!\left[\mathcal{T}_{Interm} \left[\!\left[G \right]\!\right] \right]\!\right] \quad . \end{cases}$$

5.2 Code-Erzeugung

In der letzten Phase der Compilation wird aus der internen Repräsentation eines `with`-Konstruktes C-Code erzeugt, der ICMs enthält. Die ICMs ermöglichen es, Änderungen an der konkreten C-Implementation vorzunehmen, ohne in die eigentliche Code-Erzeugung eingreifen zu müssen.

C_{ICM} bezeichne die Menge aller C-Programme, die ICM-Befehle enthalten. Dann läßt sich die Code-Erzeugung für ein SAC-Programm durch ein Compilations-Schema

$$\mathcal{C}\,[\![\ \text{SAC-}\mathit{Programm}\ ]\!] \longmapsto C_{ICM}\text{-}\mathit{Programm}$$

beschreiben (siehe [Scho96]). Die Compilations-Regeln für das `with`-Konstrukt ergeben sich wie im folgenden dargestellt.

Für ein `with`-Konstrukt mit `genarray`- oder `modarray`-Operator wird Code erzeugt, der im wesentlichen aus dem Compilat der Generatoren besteht. Dieses wird von zwei ICMs umschlossen, die das Environment in Form von lokalen Hilfsvariablen bereitstellen:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} A\texttt{ = with(}iv\texttt{)} \\ \qquad \mathit{Generatoren} \\ \texttt{genarray(}\mathit{shape}\texttt{);} \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \texttt{/* Bereitstellung von Speicher für }A\texttt{ */} \\ \texttt{WL\_BEGIN(}A\texttt{,}\ iv\texttt{)} \\ \quad \mathcal{C}_{Gen}\,[\![\ A, iv, \bot, \emptyset,\ \mathcal{T}^*\,[\![\ \mathit{Generatoren}\ ]\!]\ ]\!] \\ \texttt{WL\_END(}A\texttt{,}\ iv\texttt{)} \end{array}\right.$$

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} A\texttt{ = with(}iv\texttt{)} \\ \qquad \mathit{Generatoren} \\ \texttt{modarray(}B\texttt{);} \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \texttt{/* Bereitstellung von Speicher für }A\texttt{ */} \\ \texttt{WL\_BEGIN(}A\texttt{,}\ iv\texttt{)} \\ \quad \mathcal{C}_{Gen}\,[\![\ A, iv, \bot, \emptyset,\ \mathcal{T}^*\,[\![\ \mathit{Generatoren}\ ]\!]\ ]\!] \\ \texttt{WL\_END(}A\texttt{,}\ iv\texttt{)} \end{array}\right.$$

Ein `with`-Konstrukt mit `fold`-Operator wird in ähnlicher Weise compiliert. Zusätzlich muß zu Beginn die Akkumulationsvariable der `fold`-Operation mit dem neutralen Element initialisiert werden:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l} A\texttt{ = with(}iv\texttt{)} \\ \qquad \mathit{Generatoren} \\ \texttt{fold(}\mathit{foldfun}\texttt{, }\mathit{neutral}\texttt{);} \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \mathcal{C}\,[\![\ A\texttt{ = }\mathit{neutral}\texttt{;}\ ]\!] \\ \texttt{WL\_BEGIN(}A\texttt{,}\ iv\texttt{)} \\ \quad \mathcal{C}_{Gen}\,[\![\ A, iv, \mathit{foldfun}, \emptyset,\ \mathcal{T}^*\,[\![\ \mathit{Generatoren}\ ]\!]\ ]\!] \\ \texttt{WL\_END(}A\texttt{,}\ iv\texttt{)} \end{array}\right.$$

Um die Generatoren zu übersetzen, werden diese zunächst mit Hilfe der im letzten Abschnitt definierten Transformation $\mathcal{T}^*$ in die Zwischendarstellung transformiert. Anschließend wird daraus entsprechend dem zweiten Compilations-Schema $\mathcal{C}_{Gen}$ C-Code generiert. Dieses zusätzliche Schema wird notwendig, da die Zwischendarstellung nicht durchgängig kontextfrei compiliert werden kann, weshalb dem Schema einige zusätzliche Argumente übergeben werden müssen. Im einzelnen sind dies der Name der Ergebnisvariablen (A), der Name der Generatorvariablen (iv), der Name der `fold`-Funktion (f) — soweit vorhanden — und die Menge der abzurollenden Dimensionen (urd).

Bei aufeinanderfolgenden Teilbäumen der gleichen Dimension wird das Compilations-Schema rekursiv auf alle diese Teilbäume angewandt:

$$\mathcal{C}_{Gen}\left[\!\!\left[\ A, iv, f, urd,\ \begin{array}{r} T_1 \\ T_2 \\ \vdots \end{array}\ \right]\!\!\right] \longmapsto \left\{\begin{array}{l} \mathcal{C}_{Gen}\,[\![\ A, iv, f, urd, T_1\ ]\!] \\ \mathcal{C}_{Gen}\,[\![\ A, iv, f, urd, T_2\ ]\!] \\ \vdots \end{array}\right.$$

Für eine Projektion mit dem Schlüsselwort `block?` wird — hinter ICMs verborgen — eine Schleife nebst öffnender und schließender Klammer generiert, die über die entsprechenden Blöcke iteriert:

$$\mathcal{C}_{Gen}\left[\!\!\left[A, iv, f, urd, \begin{array}{c} a_j \rightarrow b_j \ \texttt{block}\,l\texttt{[}j\texttt{]} \ s_j \\ T_1 \\ T_2 \\ \vdots \end{array} \right]\!\!\right] \longmapsto \left\{ \begin{array}{l} \texttt{WL\_BLOCK\_BEGIN(}j\texttt{,}\ iv\texttt{,}\ l\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} \\ \quad \mathcal{C}_{Gen}\left[\!\!\left[A, iv, f, urd, \begin{array}{c} T_1 \\ T_2 \\ \vdots \end{array} \right]\!\!\right] \\ \texttt{WL\_BLOCK\_END(}j\texttt{,}\ iv\texttt{,}\ l\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} \end{array} \right.$$

Projektionen mit dem Schlüsselwort `ublock` werden ebenfalls in eine Blocking-Schleife übersetzt. Desweiteren wird mit Hilfe des Parameters *urd* dafür gesorgt, daß der Schleifenrumpf abgerollt wird:

$$\mathcal{C}_{Gen}\left[\!\!\left[A, iv, f, urd, \begin{array}{c} a_j \rightarrow b_j \ \texttt{ublock[}j\texttt{]} \ s_j \\ T_1 \\ T_2 \\ \vdots \end{array} \right]\!\!\right] \longmapsto \left\{ \begin{array}{l} \texttt{WL\_UBLOCK\_BEGIN(}j\texttt{,}\ iv\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} \\ \quad \mathcal{C}_{Gen}\left[\!\!\left[A, iv, f, (urd \cup \{j\}), \begin{array}{c} T_1 \\ T_2 \\ \vdots \end{array} \right]\!\!\right] \\ \texttt{WL\_UBLOCK\_END(}j\texttt{,}\ iv\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} \end{array} \right.$$

Projektionen vom Typ `step` und Rasterfragmente werden, entsprechend dem in *urd* vorgefundenen Status, entweder abgerollt oder aber zu einer Schleife compiliert:

$$\mathcal{C}_{Gen}\left[\!\!\left[A, iv, f, urd, \begin{array}{c} a_j \rightarrow b_j \ \texttt{step[}j\texttt{]} \ s_j \\ T_1 \\ T_2 \\ \vdots \end{array} \right]\!\!\right]$$

$$\longmapsto \left\{ \begin{array}{ll} \texttt{WL\_STRIDE\_BEGIN(}j\texttt{,}\ iv\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} & \\ \quad \mathcal{C}_{Gen}\left[\!\!\left[A, iv, f, urd, \begin{array}{c} T_1 \\ T_2 \\ \vdots \end{array} \right]\!\!\right] & ; \ \begin{array}{l} \text{falls } j \notin urd \\ \text{und } ((b_j - a_j)/s_j) > 1 \end{array} \\ \texttt{WL\_STRIDE\_END(}j\texttt{,}\ iv\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} & \\ \hline \texttt{WL\_STRIDE\_UNROLL\_BEGIN(}j\texttt{,}\ iv\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} & \\ \quad \ldots & \\ \quad \texttt{/* folgendes } ((b_j - a_j)/s_j)\texttt{-mal abrollen: */} & \\ \quad \mathcal{C}_{Gen}\left[\!\!\left[A, iv, f, urd, \begin{array}{c} T_1 \\ T_2 \\ \vdots \end{array} \right]\!\!\right] & ; \ \begin{array}{l} \text{falls } j \in urd \\ \text{oder } ((b_j - a_j)/s_j) = 1 \end{array} \\ \quad \ldots & \\ \texttt{WL\_STRIDE\_UNROLL\_END(}j\texttt{,}\ iv\texttt{,}\ a_j\texttt{,}\ b_j\texttt{,}\ s_j\texttt{)} & \end{array} \right.$$

$$\mathcal{C}_{Gen}\ [\![\ A, iv, f, urd,\ c_j \rightarrow d_j\texttt{:}\ R\ ]\!]$$

$$\longmapsto \begin{cases} \texttt{WL\_GRID\_BEGIN(}j\texttt{,}\ iv\texttt{,}\ c_j\texttt{,}\ d_j\texttt{)} \\ \quad \mathcal{C}_{Gen}\ [\![\ A, iv, f, urd,\ R\ ]\!] \\ \texttt{WL\_GRID\_END(}j\texttt{,}\ iv\texttt{,}\ c_j\texttt{,}\ d_j\texttt{)} \end{cases} ; \quad \begin{array}{l}\text{falls } j \notin urd \\ \text{und } (d_j - c_j) > 1\end{array}$$

$$\longmapsto \begin{cases} \texttt{WL\_GRID\_UNROLL\_BEGIN(}j\texttt{,}\ iv\texttt{,}\ c_j\texttt{,}\ d_j\texttt{)} \\ \quad \ldots \\ \quad \texttt{/* folgendes } (d_j - c_j)\texttt{-mal abrollen: */} \\ \quad \mathcal{C}_{Gen}\ [\![\ A, iv, f, urd,\ R\ ]\!] \\ \quad \ldots \\ \texttt{WL\_GRID\_UNROLL\_END(}j\texttt{,}\ iv\texttt{,}\ c_j\texttt{,}\ d_j\texttt{)} \end{cases} ; \quad \begin{array}{l}\text{falls } j \in urd \\ \text{oder } (d_j - c_j) = 1\end{array}$$

Der Wert eines Zielausdrucks wird in die Akkumulationsvariable gefaltet, falls der Operator des aktuellen `with`-Konstuktes ein `fold` ist, andernfalls wird er an der aktuellen Indexvektorposition in das Ergebnis-Array geschrieben:

$$\mathcal{C}_{Gen}\ [\![\ A, iv, f, urd,\ \texttt{\{}\ assigns\ \texttt{\}:}\ expr\ ]\!] \longmapsto \begin{cases} \texttt{\{}\ \mathcal{C}\ [\![\ assigns\ ]\!]\ \texttt{\}} \\ \texttt{WL\_ASSIGN(}\ A\texttt{,}iv\texttt{,}\ \mathcal{C}\ [\![\ expr\ ]\!]\texttt{)} \end{cases} ; \quad \text{falls } f = \bot$$

$$\begin{cases} \texttt{\{}\ \mathcal{C}\ [\![\ assigns\ ]\!]\ \texttt{\}} \\ \mathcal{C}\ [\![\ A\ \texttt{=}\ f\texttt{(}\ A\texttt{,}\ expr\texttt{);}\ ]\!] \end{cases} ; \quad \text{falls } f \neq \bot$$

Kapitel 6

Laufzeitmessung: Multigrid-Relaxation

In diesem Kapitel soll an Hand eines Anwendungsbeispiels nachgewiesen werden, daß das in dieser Arbeit entwickelte Compilationsschema in der Lage ist, für das `with`-Konstrukt effizient ausführbaren Code zu erzeugen. Als Beispielproblem dient die numerische Approximation von Poisson-Gleichungen durch Multigrid-Relaxation [Hack85] — ein in der Praxis häufig eingesetztes Verfahren.

Der in den NAS-Benchmarks [BBB+94] enthaltene Multigrid-Kern 'MG' führt eine vorgebbare Anzahl kompletter Multigrid-Zyklen auf einem dreidimensionalen Array mit $2^m (m \in \{3, 4, ...\})$ Elementen pro Dimension durch. Aus dieser FORTRAN-Implementation wurde eine äquivalente — allerdings dimensionsunabhängig formulierte — SAC-Implementation abgeleitet. Für beide Programme wurden die Laufzeiten mit drei verschiedenen Array-Größen auf einem Rechner des Typs SUN ULTRASPARC mit 192 MB Arbeitsspeicher und dem Betriebssystem SUNOS 5.5 ermittelt. Das FORTRAN-Programm wurde dabei mit Hilfe des SUN FORTRAN-Compilers (F77, Version 4.2) übersetzt; der vom SAC-Compiler generierte C-Code wurde zum einen mit dem SUN C-Compiler (CC, Version 4.2) und zum anderen mit dem GNU C-Compiler (GCC, Version 2.7.2.1) compiliert. Um zu ermitteln, welchen Einfluß die Art der Compilation des `with`-Konstruktes auf die Laufzeit des gesamten Programms hat, wurde die SAC-Variante ferner mit drei verschiedenen Konfigurationen des SAC-Compilers übersetzt:

- altes `wK`:

  Das `with`-Konstrukt wird nach dem alten Compilations-Schema übersetzt, insbesondere werden keine Hochsprachen-Optimierungen vorgenommen.

- neues `wK`, ohne WLO:

  Die Compilation des `with`-Konstruktes erfolgt mit Hilfe des in dieser Arbeit vorgestellten Compilations-Schemas. In Ermangelung geeigneter Inferenzmechanismen wird allerdings kein Blocking, o. ä. durchgeführt. Zusätzlich werden die beiden Optimierungen `with`-Loop-Folding und `with`-Loop-Unrolling (siehe [Schw98]) deaktiviert.

- neues `wK`, mit `WLO`:

  Es wird das neue Compilationsschema für das `with`-Konstrukt (ohne Blocking) mit vollständigen Hochsprachen-Optimierungen verwendet.

Aus Tabelle 6.1 geht hervor, daß bei dem SAC-Programm allein die Verwendung des neuen Compilations-Schemas einen Laufzeitgewinn von ca. 9 % erbringt. In Kombination mit den Optimierungen `with`-Loop-Folding und -Unrolling verbessert sich die Laufzeit im Mittel um weitere 17 %. Dadurch wird der Laufzeit-Vorsprung des FORTRAN-Compilats bei Verwendung des CC auf ca. 14 % verkürzt. Falls der SAC-Compiler zusammen mit dem GCC eingesetzt wird, führt dies zu einem Code, der sogar 5–9 % schneller ausgeführt wird.

Array-Größe, Anz. Zyklen	SAC mit CC			SAC mit GCC			F77
	altes wK	neues wK ohne WLO	neues wK mit WLO	altes wK	neues wK ohne WLO	neues wK mit WLO	
32^3, 50	8,0 s	7,0 s	5,9 s	7,1 s	5,6 s	4,7 s	5,0 s
64^3, 10	6,2 s	5,7 s	4,7 s	5,5 s	4,7 s	3,8 s	4,0 s
128^3, 1	9,8 s	8,9 s	7,4 s	8,7 s	7,3 s	5,9 s	6,5 s

Tabelle 6.1: Laufzeiten einer 3D-Multigrid-Relaxation.

Kapitel 7

Zusammenfassung

Die vorliegende Arbeit stellt für das `with`-Konstrukt der Programmiersprache SAC ein verbessertes Compilations-Schema vor. Wesentliche Neuerungen dieses Compilations-Schemas sind zum einen die Unterstützung der Hochsprachen-Optimierung `with`-Loop-Folding, zum anderen das Vorhandensein mächtiger Mechanismen zur Verbesserung der Cache-Ausnutzung.

Vor allem die Einführung des `with`-Loop-Folding stellt hohe Anforderungen an die Code-Erzeugung. Durch das Zusammenfalten mehrerer aufeinanderfolgender `with`-Konstrukte zu einem einzigen entstehen im allgemeinen multiple und sehr inhomogene Generatormengen. Um im Compilat das Schleifen-Overhead auf ein Minimum zu reduzieren und für eine gute Trefferrate im Cache zu sorgen, muß daraus ein Code generiert werden, der ein hohes Maß an Datenlokalität aufweist. Dies ist jedoch nur mit Hilfe ausgefeilter Code-Transformationen möglich. Da deren Durchführung von einem üblichen C-Compiler nicht gewährleistet werden kann, müssen diese explizit in das Compilations-Schema integriert werden. Durch die Verwendung einer Zwischendarstellung gelingt es, die semantische Lücke zwischen SAC und der Zielsprache C soweit zu verringern, daß eine konzise Spezifikation und Implementierung der notwendigen Transformationen möglich wird.

Durchgeführte Laufzeitmessungen zeigen, daß das entwickelte Compilations-Schema zusammen mit `with`-Loop-Folding eine substantielle Leistungsverbesserung des SAC-Compilers bewirkt. Bei Anwendungen wie z. B. einem Multigrid-Verfahren werden sogar Laufzeiten erreicht, die denen vergleichbarer, aber für eine feste Dimensionalität spezifizierten, FORTRAN-Programmen ebenbürtig sind.

Es bleibt aber noch einiges Potential ungenutzt. So sind etwa die beiden Optimierungen Blocking und Unrolling-Blocking bereits im Compiler angelegt, werden aber im Normalfall nicht aktiviert, da die dafür notwendigen Parameter noch nicht automatisch an Hand des Quellcodes inferiert werden können. Statt dessen existiert mit dem `wlcomp`-Pragma aber eine sehr flexible Schnittstelle, über die derartige Informationen dem Compilations-Schema übermittelt werden können. Zur Zeit wird dieser Mechanismus dazu verwendet, Compilations-Parameter im Programmtext annotieren zu können. Von großem Interesse wäre aber die Entwicklung eines Moduls, das die Blocking-Faktoren, u. ä. selbständig ermittelt und dann in Form eines `wlcomp`-Pragmas dem Compiler bekanntmacht.

Nicht behandelt wurde in dieser Arbeit der Themenkomplex der nicht-sequentiellen Programmausführung. Dabei eignet sich gerade das `with`-Konstrukt besonders gut zur nebenläufigen Berechnung, da es zwischen den einzelnen Elementen der Generatormenge prinzipiell keine Datenabhängigkeiten gibt. Ferner ist die Berechnung eines `with`-Konstruktes in der Regel sehr zeitaufwendig, so daß eine Verteilung auf mehrere Prozessoren sehr lohnend erscheint. Aus diesem Grund wurde bei der Implementierung des neuen Compilations-Schemas ausdrücklich darauf geachtet, daß es leicht dahingehend erweitert werden kann, auch nichtsequentiell ausführbaren Code zu generieren. In der Tat gibt es bereits eine erste lauffähige Version des SAC-Compilers, die in der Lage ist, zu einem `with`-Konstrukt multi-threadingfähigen Code zu erzeugen. Es ist auf diesem Gebiet aber noch einige Forschungs- und Entwicklungsarbeit notwendig.

Anhang A

Generator-Transformationen

Dieser Abschnitt enthält die formalen Spezifikationen der in Abschnitt 5.1 vorgestellten Transformationen. $\mathcal{T}_{Interm}$ dient zur Überführung der Generatormenge eines `with`-Konstruktes in einen Baum der Zwischendarstellung, alle übrigen Transformationen modifizieren einen solchen. Da einige Transformationen und vor allem die anschließende Code-Erzeugung (kanonische Reihefolge!) einen Baum erwarten, in dem alle Projektionen einer Dimension nach aufsteigenden Startpunkten sortiert sind, ist es wichtig, daß alle Transformationen wieder einen Baum dieser Eigenschaft liefern. Diese Anforderung wurde in den Transformations-Schemata jedoch *nicht* explizit umgesetzt, da großer Wert auf eine möglichst kompakte Darstellung gelegt wurde.

$$\mathcal{T}_{Interm} \left[\!\!\left[\begin{array}{l} G_1 : op_1 \\ G_2 : op_2 \\ \vdots \end{array} \right]\!\!\right] \longmapsto \left\{ \begin{array}{l} \mathcal{T}_{Interm} \ [\![G_1 : op_1]\!] \\ \mathcal{T}_{Interm} \ [\![G_2 : op_2]\!] \\ \vdots \end{array} \right.$$

$$\mathcal{T}_{Interm} [\![\texttt{(}(a_0, ..., a_{n-1}) \ \leq \ iv \ < \ (b_0, ..., b_{n-1}) \ \texttt{step} \ (s_0, ..., s_{n-1}) \ \texttt{width} \ (w_0, ..., w_{n-1})\texttt{):} \ op]\!]$$

$$\longmapsto \left\{ \begin{array}{r} a_0 \ \rightarrow \ b_0 \ \texttt{step[0]} \ s_0 \\ 0 \ \rightarrow \ w_0\texttt{:} \ \ldots \\ a_{n-1} \ \rightarrow \ b_{n-1} \ \texttt{step[}n\texttt{-1]} \ s_{n-1} \\ 0 \ \rightarrow \ w_{n-1}\texttt{:} \ op \end{array} \right.$$

Abbildung A.1: Die Transformation $\mathcal{T}_{Interm}$.

$$\mathcal{T}_{Seg}\left[\!\!\left[\begin{array}{l} T_1 \\ S,\ T_2 \\ \vdots \end{array}\right]\!\!\right] \longmapsto \begin{cases} \mathcal{T}_{Seg}\left[\!\!\left[\ S,\ T_1\ \right]\!\!\right] \\ \mathcal{T}_{Seg}\left[\!\!\left[\ S,\ T_2\ \right]\!\!\right] \\ \vdots \end{cases}$$

$$\mathcal{T}_{Seg}\left[\!\!\left[\begin{array}{ll} & a_0 \ \to\ b_0\ \texttt{step[0]}\ s_0 \\ & \qquad\quad c_{1,0} \ \to\ d_{1,0}\texttt{:}\ a_1 \ \to\ b_1\ \texttt{step[1]}\ \ldots \\ S, & \qquad\quad c_{2,0} \ \to\ d_{2,0}\texttt{:}\ a_1 \ \to\ b_1\ \texttt{step[1]}\ \ldots \\ & \qquad\quad \vdots \end{array}\right]\!\!\right]$$

$$\longmapsto \begin{cases} \begin{array}{ll} a'_0 \ \to\ b'_0\ \texttt{step[0]}\ s_0 & \\ \qquad\quad c_{1,0} \ \to\ d_{1,0}\texttt{:}\ a'_1 \ \to\ b'_1\ \texttt{step[1]}\ \ldots & \\ \qquad\quad c_{2,0} \ \to\ d_{2,0}\texttt{:}\ a'_1 \ \to\ b'_1\ \texttt{step[1]}\ \ldots & ; \\ \qquad\quad \vdots & \end{array} & \begin{array}{l}\text{falls } S \cap [\,a;\,b\,) \neq \emptyset, \\ \text{wobei } [\,a';\,b'\,) := \overline{S} \cap [\,a;\,b\,)\end{array} \\ \hline ./. \quad ; & \text{falls } S \cap [\,a;\,b\,) = \emptyset \end{cases}$$

Abbildung A.2: Die Transformation $\mathcal{T}_{Seg}$ bezüglich eines Segmentes S.

$$\mathcal{T}_{Split}\left[\!\!\left[\begin{array}{ll} a_{1,0} \ \to\ b_{1,0}\ \texttt{step[0]}\ s_{1,0} \\ \qquad\qquad\qquad\quad\ R_1 \\ a_{2,0} \ \to\ b_{2,0}\ \texttt{step[0]}\ s_{2,0} \\ \qquad\qquad\qquad\quad\ R_2 \\ T \end{array}\right]\!\!\right], \quad \begin{array}{l}\text{wobei die } (a_{.,0}, b_{.,0}) \text{ o. B. d. A. aufsteigend sortiert sind,} \\ \text{d. h. } (a_{1,0} < a_{2,0}) \vee ((a_{1,0} = a_{2,0}) \wedge (b_{1,0} \leq b_{2,0}))\end{array}$$

$$\longmapsto \begin{cases} \begin{array}{l} a_{1,0} \ \to\ b_{1,0}\ \texttt{step[0]}\ s_{1,0} \\ \qquad\qquad\qquad\quad\ R_1 \\ \mathcal{T}_{Split}\left[\!\!\left[\begin{array}{l} a_{2,0} \ \to\ b_{2,0}\ \texttt{step[0]}\ s_{2,0} \\ \qquad\qquad\qquad\quad\ R_2 \\ T \end{array}\right]\!\!\right] \end{array} \quad ; & \begin{array}{l}\text{falls } [\,a_{1,0};\,b_{1,0}\,), [\,a_{2,0};\,b_{2,0}\,) \\ \text{gleich oder disjunkt}\end{array} \\ \hline \begin{array}{l} a_{1,0} \ \to\ a_{2,0}\ \texttt{step[0]}\ s_{1,0} \\ \qquad\qquad\qquad\quad\ R_1 \\ a_{2,0} \ \to\ b_{1,0}\ \texttt{step[0]}\ s_{1,0} \\ \qquad\qquad\qquad\quad\ R'_1 \\ \mathcal{T}_{Split}\left[\!\!\left[\begin{array}{l} a_{2,0} \ \to\ b_{1,0}\ \texttt{step[0]}\ s_{2,0} \\ \qquad\qquad\qquad\quad\ R_2 \\ b_{1,0} \ \to\ b_{2,0}\ \texttt{step[0]}\ s_{2,0} \\ \qquad\qquad\qquad\quad\ R'_2 \\ T \end{array}\right]\!\!\right] \end{array} \quad ; & \text{sonst} \end{cases}$$

R'_1, R'_2 bezeichnen die aus R_1, R_2 hervorgegangenen Unterbäume
mit an die neuen Grenzen angepaßter Rasterung.

Paare von Teilbäumen, deren Umrisse sich erst in tieferen Dimensionen unterscheiden,
werden analog transformiert.

Abbildung A.3: Die Transformation $\mathcal{T}_{Split}$.

$$\mathcal{T}_{Block}\left[\!\left[\begin{array}{l} T_1 \\ bv,\ T_2 \\ \vdots \end{array}\right]\!\right] \longmapsto \left\{\begin{array}{l} \mathcal{T}_{Block'}\left[\!\left[\ bv,\ 0,\ T_1\ \right]\!\right] \\ \mathcal{T}_{Block'}\left[\!\left[\ bv,\ 0,\ T_2\ \right]\!\right] \\ \vdots \end{array}\right.$$

$$\mathcal{T}_{Block'}\left[\!\left[\begin{array}{ll} & a_j \ \rightarrow\ b_j\ \texttt{block}\,l'\texttt{[}j\texttt{]}\ s_j \\ bv,\ l, & \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ R_1 \\ & \quad c_{2,j}\ \rightarrow\ d_{2,j}\texttt{:}\ R_2 \\ & \quad \vdots \end{array}\right]\!\right]$$

$$\longmapsto \left\{\begin{array}{ll} a_j \ \rightarrow\ b_j\ \texttt{block}\,l'\texttt{[}j\texttt{]}\ s_j & \\ \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ \mathcal{T}_{Block'}\left[\!\left[\ bv,\ (l'+1),\ R_1\ \right]\!\right] & \\ \quad c_{2,j}\ \rightarrow\ d_{2,j}\texttt{:}\ \mathcal{T}_{Block'}\left[\!\left[\ bv,\ (l'+1),\ R_2\ \right]\!\right] & \\ \quad \vdots & \end{array}\right.$$

$$\mathcal{T}_{Block'}\left[\!\left[\begin{array}{ll} & a_j \ \rightarrow\ b_j\ \texttt{step[}j\texttt{]}\ s_j \\ bv,\ l, & \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ R_1 \\ & \quad c_{2,j}\ \rightarrow\ d_{2,j}\texttt{:}\ R_2 \\ & \quad \vdots \end{array}\right]\!\right], \quad \text{wobei}\ \ bv_j = 1$$

$$\longmapsto \left\{\begin{array}{ll} a_j \ \rightarrow\ b_j\ \texttt{step[}j\texttt{]}\ s_j & \\ \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ \mathcal{T}_{Block'}\left[\!\left[\ bv,\ l,\ R_1\ \right]\!\right] & \\ \quad c_{2,j}\ \rightarrow\ d_{2,j}\texttt{:}\ \mathcal{T}_{Block'}\left[\!\left[\ bv,\ l,\ R_2\ \right]\!\right] & \\ \quad \vdots & \end{array}\right.$$

$$\mathcal{T}_{Block'}\left[\!\left[\begin{array}{ll} & a_j \ \rightarrow\ b_j\ \texttt{step[}j\texttt{]}\ s_j \\ bv,\ l, & \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ R \\ & \quad \vdots \end{array}\right]\!\right], \quad \text{wobei}\ \ bv_j > 1$$

$$\longmapsto \left\{\begin{array}{l} a_j \ \rightarrow\ b_j\ \texttt{block}\,l\texttt{[}j\texttt{]}\ bv_j \\ \qquad \mathcal{T}_{Block'}\left[\!\left[\ bv,\ l,\ R\ \right]\!\right] \\ \qquad\qquad \mathcal{T}_{Inner}\left[\!\left[\begin{array}{ll} & a_j \ \rightarrow\ b_j\ \texttt{step[}j\texttt{]}\ s_j \\ bv, & \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ R \\ & \quad \vdots \end{array}\right]\!\right] \end{array}\right.$$

$$\mathcal{T}_{Block'}\left[\!\left[\ bv,\ op\ \right]\!\right] \longmapsto ./.$$

$$\mathcal{T}_{Inner}\left[\!\left[\ bv,\ a_j \ \rightarrow\ b_j\ \texttt{step[}j\texttt{]}\ \texttt{1}\ \ \texttt{0}\ \rightarrow\ \texttt{1:}\ R\ \right]\!\right], \quad \text{wobei}\ \ bv_j = 1$$

$$\longmapsto \mathcal{T}_{Inner}\left[\!\left[\ bv,\ R\ \right]\!\right]$$

$$\mathcal{T}_{Inner}\left[\!\left[\begin{array}{ll} & a_j \ \rightarrow\ b_j\ \texttt{step[}j\texttt{]}\ s_j \\ bv, & \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ R_1 \\ & \quad c_{2,j}\ \rightarrow\ d_{2,j}\texttt{:}\ R_2 \\ & \quad \vdots \end{array}\right]\!\right], \quad \text{wobei}\ \ bv_j > 1$$

$$\longmapsto \left\{\begin{array}{ll} \texttt{0}\ \rightarrow\ bv_j\ \texttt{step[}j\texttt{]}\ s_j & \\ \quad c_{1,j}\ \rightarrow\ d_{1,j}\texttt{:}\ \mathcal{T}_{Inner}\left[\!\left[\ bv,\ R_1\ \right]\!\right] & \\ \quad c_{2,j}\ \rightarrow\ d_{2,j}\texttt{:}\ \mathcal{T}_{Inner}\left[\!\left[\ bv,\ R_2\ \right]\!\right] & \\ \quad \vdots & \end{array}\right.$$

$$\mathcal{T}_{Inner}\left[\!\left[\ bv,\ op\ \right]\!\right] \longmapsto op$$

Abbildung A.4: Die Transformation $\mathcal{T}_{Block}$ mit Blocking-Vektor bv.

$$\mathcal{T}_{Merge}\left[\!\!\left[\begin{array}{llll} a_{1,j} & \rightarrow & b_{1,j} & \texttt{step[}j\texttt{]}\ s_{1,j} \\ & & & \quad R_1 \\ a_{2,j} & \rightarrow & b_{2,j} & \texttt{step[}j\texttt{]}\ s_{2,j} \\ & & & \quad R_2 \\ T & & & \end{array}\right]\!\!\right], \quad \text{wobei die } a_{\cdot,j} \text{ o. B. d. A. aufsteigend sortiert sind}$$

$$\longmapsto \begin{cases} \mathcal{T}_{Merge}\left[\!\!\left[\begin{array}{llll} a_{1,j} & \rightarrow & b_{1,j} & \texttt{step[}j\texttt{]}\ s'_j \\ & & & \quad R'_{12} \\ T & & & \end{array}\right]\!\!\right] & ; \quad \begin{array}{l}\text{falls } a_{1,j} = a_{2,j} \text{ und } b_{1,j} = b_{2,j}, \\ \text{wobei } s'_j := \mathrm{kgV}(s_{1,j}, s_{2,j})\end{array} \\ \hline \begin{array}{l} a_{1,j} \rightarrow b_{1,j}\ \texttt{step[}j\texttt{]}\ s_{1,j} \\ \qquad\qquad\qquad\quad \mathcal{T}_{Merge}\left[\!\left[R_1 \right]\!\right] \\ \mathcal{T}_{Merge}\left[\!\!\left[\begin{array}{llll} a_{2,j} & \rightarrow & b_{2,j} & \texttt{step[}j\texttt{]}\ s_{2,j} \\ & & & \quad R_2 \\ T & & & \end{array}\right]\!\!\right] \end{array} & ; \quad \text{sonst} \end{cases}$$

$$\mathcal{T}_{Merge}\left[\!\!\left[\begin{array}{llll} a_j & \rightarrow & b_j & \texttt{step[}j\texttt{]}\ s_j \\ & & & \quad R \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{llll} a_j & \rightarrow & b_j & \texttt{step[}j\texttt{]}\ s_j \\ & & & \quad \mathcal{T}_{Merge}\left[\!\left[R \right]\!\right] \end{array}\right.$$

$$\mathcal{T}_{Merge}\left[\!\!\left[\begin{array}{lll} c_{1,j} & \rightarrow & d_{1,j}\colon\ R_1 \\ c_{2,j} & \rightarrow & d_{2,j}\colon\ R_2 \\ T & & \end{array}\right]\!\!\right], \quad \text{wobei die } c_{\cdot,j} \text{ o. B. d. A. aufsteigend sortiert sind}$$

$$\longmapsto \begin{cases} \mathcal{T}_{Merge}\left[\!\!\left[\begin{array}{lll} c_{1,j} & \rightarrow & d_{1,j}\colon\ R_1 \\ & & \quad R_2 \\ T & & \end{array}\right]\!\!\right] & ; \quad \text{falls } c_{1,j} = c_{2,j} \text{ und } d_{1,j} = d_{2,j} \\ \hline \begin{array}{l} c_{1,j} \rightarrow d_{1,j}\colon\ \mathcal{T}_{Merge}\left[\!\left[R_1 \right]\!\right] \\ \mathcal{T}_{Merge}\left[\!\!\left[\begin{array}{lll} c_{2,j} & \rightarrow & d_{2,j}\colon\ R_2 \\ T & & \end{array}\right]\!\!\right] \end{array} & ; \quad \text{sonst} \end{cases}$$

$$\mathcal{T}_{Merge}\left[\!\left[c_j \rightarrow d_j\colon\ R \right]\!\right] \longmapsto c_j \rightarrow d_j\colon\ \mathcal{T}_{Merge}\left[\!\left[R \right]\!\right]$$

$$\mathcal{T}_{Merge}\left[\!\left[op \right]\!\right] \longmapsto op$$

R'_{12} bezeichnet den aus R_1, R_2 hervorgegangenen Unterbaum
mit an die neue Schrittweite s'_j angepaßter Rasterung.

Analoge Regeln gelten für Projektionen mit den Schlüsselwörtern `block?` und `ublock`.

Abbildung A.5: Die Transformation $\mathcal{T}_{Merge}$.

$$\mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_{1,j} & \to & b_{1,j} & \texttt{block}l\texttt{[}j\texttt{]} & s_{1,j} \\ & & & & R_1 \\ a_{2,j} & \to & b_{2,j} & \texttt{block}l\texttt{[}j\texttt{]} & s_{2,j} \\ & & & & R_2 \\ T & & & & \end{array}\right]\!\!\right], \quad \text{wobei die } a_{\cdot,j} \text{ o. B. d. A. aufsteigend sortiert sind}$$

$$\longmapsto \begin{cases} \mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_{1,j} & \to & b_{2,j} & \texttt{block}l\texttt{[}j\texttt{]} & s_{1,j} \\ & & & & R_1 \\ T & & & & \end{array}\right]\!\!\right] & ; \quad \begin{array}{l}\text{falls } b_{1,j} = a_{2,j},\ s_{1,j} = s_{2,j} \\ \text{und } R_1 = R_2\end{array} \\ \hline a_{1,j} \to b_{1,j}\ \texttt{block}l\texttt{[}j\texttt{]}\ s_{1,j}\ \mathcal{T}_{Opt}\left[\!\!\left[R_1 \right]\!\!\right] & \\ \mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_{2,j} & \to & b_{2,j} & \texttt{block}l\texttt{[}j\texttt{]} & s_{2,j} \\ & & & & R_2 \\ T & & & & \end{array}\right]\!\!\right] & ; \quad \text{sonst} \end{cases}$$

$$\mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_{1,j} & \to & b_{1,j} & \texttt{ublock[}j\texttt{]} & s_{1,j} \\ & & & & R_1 \\ a_{2,j} & \to & b_{2,j} & \texttt{ublock[}j\texttt{]} & s_{2,j} \\ & & & & R_2 \\ T & & & & \end{array}\right]\!\!\right] \longmapsto \text{analog zu oben}$$

$$\mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_{1,j} & \to & b_{1,j} & \texttt{step[}j\texttt{]} & s_{1,j} \\ & & & & R_1 \\ a_{2,j} & \to & b_{2,j} & \texttt{step[}j\texttt{]} & s_{2,j} \\ & & & & R_2 \\ T & & & & \end{array}\right]\!\!\right], \quad \text{wobei die } a_{\cdot,j} \text{ o. B. d. A. aufsteigend sortiert sind}$$

$$\longmapsto \begin{cases} \mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_{1,j} & \to & b_{2,j} & \texttt{step[}j\texttt{]} & s_{1,j} \\ & & & & R_1 \\ T & & & & \end{array}\right]\!\!\right] & ; \quad \begin{array}{l}\text{falls } b_{1,j} = a_{2,j},\ s_{1,j} = s_{2,j} \\ \text{und } s_{1,j} \mid (b_{1,j} - a_{1,j}) \\ \text{und } R_1 = R_2\end{array} \\ \hline a_{1,j} \to b_{1,j}\ \texttt{step[}j\texttt{]}\ s_{1,j}\ \mathcal{T}_{Opt}\left[\!\!\left[R_1 \right]\!\!\right] & \\ \mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_{2,j} & \to & b_{2,j} & \texttt{step[}j\texttt{]} & s_{2,j} \\ & & & & R_2 \\ T & & & & \end{array}\right]\!\!\right] & ; \quad \text{sonst} \end{cases}$$

$$\mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_j & \to & b_j & \texttt{block}l\texttt{[}j\texttt{]} & s_j \\ & & & & R \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{lllll} a_j & \to & b_j & \texttt{block}l\texttt{[}j\texttt{]} & s_j \\ & & & & \mathcal{T}_{Opt}\left[\!\!\left[R \right]\!\!\right] \end{array}\right.$$

$$\mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_j & \to & b_j & \texttt{ublock[}j\texttt{]} & s_j \\ & & & & R \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{lllll} a_j & \to & b_j & \texttt{ublock[}j\texttt{]} & s_j \\ & & & & \mathcal{T}_{Opt}\left[\!\!\left[R \right]\!\!\right] \end{array}\right.$$

$$\mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{lllll} a_j & \to & b_j & \texttt{step[}j\texttt{]} & s_j \\ & & & & R \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{lllll} a_j & \to & b_j & \texttt{step[}j\texttt{]} & s_j \\ & & & & \mathcal{T}_{Opt}\left[\!\!\left[R \right]\!\!\right] \end{array}\right.$$

$$\mathcal{T}_{Opt}\left[\!\!\left[\begin{array}{ll} c_{1,j} \to d_{1,j}\texttt{:} & R_1 \\ c_{2,j} \to d_{2,j}\texttt{:} & R_2 \\ \vdots & \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{ll} c_{1,j} \to d_{1,j}\texttt{:} & \mathcal{T}_{Opt}\left[\!\!\left[R_1 \right]\!\!\right] \\ c_{2,j} \to d_{2,j}\texttt{:} & \mathcal{T}_{Opt}\left[\!\!\left[R_2 \right]\!\!\right] \\ \vdots & \end{array}\right.$$

$$\mathcal{T}_{Opt}\left[\!\!\left[op \right]\!\!\right] \longmapsto op$$

Abbildung A.6: Die Transformation $\mathcal{T}_{Opt}$.

$$\mathcal{T}_{Fit}\left[\!\!\left[\begin{array}{l} T_1 \\ T_2 \\ \vdots \end{array}\right]\!\!\right] \longmapsto \left\{\begin{array}{l} \mathcal{T}_{Fit'}\left[\!\!\left[\ 0,\ T_1\ \right]\!\!\right] \\ \mathcal{T}_{Fit'}\left[\!\!\left[\ 0,\ T_2\ \right]\!\!\right] \\ \vdots \end{array}\right.$$

$$\mathcal{T}_{Fit'}\left[\!\!\left[\ dim,\ \begin{array}{l} a_{1,j} \ \rightarrow\ b_{1,j}\ \texttt{block}l\texttt{[}j\texttt{]}\ \ s_{1,j} \\ \quad R_1 \\ \quad\ a_{2,j} \ \rightarrow\ b_{2,j}\ \texttt{ublock[}j\texttt{]}\ \ s_{2,j} \\ \quad\quad R_2 \end{array}\ \right]\!\!\right]$$

$$\longmapsto \left\{\begin{array}{ll} \begin{array}{l} a_{1,j} \ \rightarrow\ b_{1,j}\ \texttt{block}l\texttt{[}j\texttt{]}\ \ s_{1,j} \\ \quad R_1 \\ \quad\ a_{2,j} \ \rightarrow\ b_{2,j}\ \texttt{ublock[}j\texttt{]}\ \ s_{2,j} \\ \quad\quad R_2 \end{array} & ;\quad \text{falls } j \neq dim \\ \hline \begin{array}{l} a_{1,j} \ \rightarrow\ b_{1,j}\ \texttt{block}l\texttt{[}j\texttt{]}\ \ s'_{1,j} \\ \quad \mathcal{T}_{Fit'}\left[\!\!\left[\ (j+1),\ \begin{array}{l} R_1 \\ a_{2,j} \ \rightarrow\ b_{2,j}\ \texttt{ublock[}j\texttt{]}\ \ s_{2,j} \\ R_2 \end{array}\ \right]\!\!\right] \end{array} & ;\quad \begin{array}{l}\text{falls } j = dim \\ \text{und } s_{2,j} \mid (b_{1,j} - a_{1,j})\end{array} \\ \hline \begin{array}{l} a_{1,j} \ \rightarrow\ b'_{1,j}\ \texttt{block}l\texttt{[}j\texttt{]}\ \ s'_{1,j} \\ \quad \mathcal{T}_{Fit'}\left[\!\!\left[\ (j+1),\ \begin{array}{l} R_1 \\ a_{2,j} \ \rightarrow\ b_{2,j}\ \texttt{ublock[}j\texttt{]}\ \ s_{2,j} \\ R_2 \end{array}\ \right]\!\!\right] \\ b'_{1,j} \ \rightarrow\ b_{1,j}\ \texttt{block}l\texttt{[}j\texttt{]}\ \ s'_{1,j} \\ \quad \mathcal{T}_{Fit'}\left[\!\!\left[\ (j+1),\ \begin{array}{l} R_1 \\ a_{2,j} \ \rightarrow\ b_{2,j}\ \texttt{ublock[}j\texttt{]}\ \ s_{2,j} \\ R_2 \end{array}\ \right]\!\!\right] \end{array} & ;\quad \begin{array}{l}\text{falls } j = dim \\ \text{und } s_{2,j} \nmid (b_{1,j} - a_{1,j})\end{array} \\ \hline & ;\quad \begin{array}{l}\text{wobei } b'_{1,j} := b_{1,j} - ((b_{1,j} - a_{1,j}) \bmod s_{2,j}) \\ \text{und } s'_{1,j} :\approx s_{1,j} \text{ mit } s_{2,j} \mid s'_{1,j}\end{array} \end{array}\right.$$

$$\mathcal{T}_{Fit'}\left[\!\!\left[\ dim,\ \begin{array}{l} a_{1,j} \ \rightarrow\ b_{1,j}\ \texttt{block}l\texttt{[}j\texttt{]}\ \ s_{1,j} \\ \quad R_1 \\ \quad\ a_{2,j} \ \rightarrow\ b_{2,j}\ \texttt{step[}j\texttt{]}\ \ s_{2,j} \\ \quad\quad R_2 \end{array}\ \right]\!\!\right], \quad \text{wobei } R_1 \text{ kein } \texttt{ublock[}j\texttt{]} \text{ enthält}$$

$$\longmapsto \text{analog zu oben}$$

$$\mathcal{T}_{Fit'}\left[\!\!\left[\ dim,\ \begin{array}{l} a_{1,j} \ \rightarrow\ b_{1,j}\ \texttt{step[}j\texttt{]}\ \ s_{2,j} \\ \qquad\qquad c_{1,j} \ \rightarrow\ d_{1,j}\texttt{:}\ \ R_1 \\ \qquad\qquad c_{2,j} \ \rightarrow\ d_{2,j}\texttt{:}\ \ R_2 \\ \qquad\qquad \vdots \end{array}\ \right]\!\!\right] \longmapsto \text{analog zu oben}$$

Abbildung A.7: Die Transformation $\mathcal{T}_{Fit}$.

Literaturverzeichnis

[AAL95] J. M. Anderson, S. P. Amarasinghe, M. S. Lam: *Data and Computation Transformations for Multiprocessors*. In: Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP '95), Santa Barbara, California, USA. ACM Press, 1995, S. 166–178.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, et al.: *The NAS Parallel Benchmarks*. RNR 94-007, NASA Ames Research Center, 1994.

[BGS94] D. F. Bacon, S. L. Graham, O. J. Sharp: *Compiler Transformations for High-Performance Computing*. ACM Computing Surveys, 26(4), S. 345–420, 1994.

[Cann93] D. C. Cann: *The Optimizing Sisal Compiler (Version 12.0)*. Lawrence Livermore National Laboratory, Livermore, California, USA, 1993. Part of the Sisal distribution.

[DK95] E. van der Deijl, G. Kanbier: *The Cache Visualization Tool*. University of Leiden, The Netherlands, 1995.

[Grel96] C. Grelck: *Integration eines Modul- und Klassen-Konzeptes in die funktionale Programmiersprache SAC — Single Assignment C*. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.

[Hack85] W. Hackbusch: *Multi-Grid Methods and Applications*. Springer, 1985. ISBN 3-540-12761-5.

[HP97] K. Hammond, J. Peterson: *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language (Version 1.4)*. Yale University, New Haven, Connecticut, USA, 1997.

[Iver62] K. E. Iverson: *A Programming Language*. John Wiley & Sons, 1962.

[KR88] B. W. Kernighan, D. M. Ritchie: *The C Programming Language*. Prentice Hall, 2. Auflage, 1988. ISBN 0-13-110362-8.

[Lam94] M. S. Lam: *Locality Optimization for Parallel Machines*. In: Third Joint International Conference on Vector and Parallel Processing. 1994.

[LF93] H. Liebig, T. Flik: *Rechnerorganisation*. Springer, 2. Auflage, 1993. ISBN 3-540-54632-4.

[LRW91] M. S. Lam, E. E. Rothberg, M. E. Wolf: *The Cache Performance and Optimizations of Blocked Algorithms*. In: Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, California, USA. 1991, S. 63–74.

[MA95] N. Manjikian, T. S. Abdelrahman: *Array Data Layout for the Reduction of Cache Conflicts*. In: Proceedings of the International Conference on Parallel and Distributed Computing Systems. 1995.

[MLG92] T. C. Mowry, M. S. Lam, A. Gupta: *Design and Evaluation of a Compiler Algorithm for Prefetching*. In: Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems. 1992, S. 62–73.

[Mull88] L. M. R. Mullin: *A Mathematics of Arrays*. Dissertation, Syracuse University, New York, USA, 1988.

[PE97] R. Plasmeijer, M. van Eekelen: *Concurrent Clean 1.2 Language Report*. University of Nijmegen, The Netherlands, 1997.

[Scho96] S.-B. Scholz: *Single Assignment C — Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen*. Dissertation, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996. ISBN 3-8265-3138-8.

[Scho98] S.-B. Scholz: *WITH-Loop-Folding in SAC — Condensing Consecutive Array Operations*. In C. Clack, T. Davie, K. Hammond (Hrsg.): Implementation of Functional Languages, 9th International Workshop (IFL '97), St. Andrews, Scotland, UK, Selected Papers. Band 1467 von: LNCS. Springer, 1998, S. 72–91. ISBN 3-540-64849-6.

[Schw98] S. Schwartz: *Zur Code-Optimierung von Schleifenkonstrukten der Programmiersprache SAC*. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1998.

[Siev95] A. Sievers: *Maschinenunabhängige Optimierungen eines Compilers für die funktionale Programmiersprache Single Assignment C*. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1995.

[WL91] M. E. Wolf, M. S. Lam: *A Data Locality Optimizing Algorithm*. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '91), Toronto, Ontario, Canada. ACM Press, 1991, S. 30–44.

[Wolf95] H. Wolf: *SAC → C: Ein Basiscompiler für die funktionale Programmiersprache SAC*. Diplomarbeit, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1995.

Index

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, den 1. August 1998