Optimierung in Übersetzern
Zusammenfassung des Wichtigsten

Sommersemester 2016

Prof. Dr. Philippsen

Autor:

- Christian Strate

Inhaltsverzeichnis

0 Organisatorisches

Note:

Bei dieser Zusammenfassung handelt es sich um eine inoffizielle Mitschrift von Studenten. Entsprechend sind weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet.

1 Einführung, Überblick

- gcc-flags -o/-O1 gcc-Flags aktiviert einige Flags zur Optimierung: slide 1-12
- Möglichkeiten der Optimierung
  - auf Zwischen-Code: Transformationen, maschinen-unabhängig
  - auf dem Zielprogram: Schlüsselloch-Optimierung, maschinen-abhängig Scheduler
- Compiler verfügt (außer bei Laufzeitcompilern) nicht über Profiler, entsprechend müssen Laufzeiten geschätzt werden.
- Analyse-/Transformationsaufwand muss sich lohnen. Faustregel: quadratisch ist häufig schon zu schlecht.
- Das Schwierige an Transformationen ist die Analyse, ob diese angewendet werden dürfen (bsp. Dead-Code-Eliminiation), nicht die Transformation selbst
- Optimierung: Besserer Code als ohne Optimierung

Grundblock, Basisblock:

Folge von Instruktionen, bei der die Ausführung die Folge *nur* bei der ersten Instruktion betritt und *nur* bei der letzten Instruktion verlässt. Die Instruktionen werden sequentiell ausgeführt. Sprünge und Verzweigungen dürfen nur am Ende eines Grundblocks auftreten.

Minimaler Grundblock: eine Instruktion
Maximaler Grundblock: Zum Block kann keien Instruktion hinzugenommen werden, ohne die Grundblockeigenschaft zu zerstören.

Lokale Optimierungen sind nicht möglich, wenn Funktionen Seiteneffekte vorweisen. Die Grundblock-Repräsentation ist fehlerhaft, wenn eine Funktion nicht nur über die Rückkehr zur Aufrufstelle verlassen werden kann. Berechnete Sprünge machen es dem Optimierer schwer.

Grundblockberechnung - Bsp. 1-26f:

1. Blockanfänge markieren
   - Erster Befehl
   - Jedes Ziel eines Sprungs
   - Jeder auf einen Sprung folgende Befehl
2. Grundblöcke ablesen
   - Grundblockanfang bis zum nächsten Grundblockanfang bzw. Programmende

Lokale Optimierung:

Sammelt Informationen nur innerhalb eines Grundblocks und wendet Optimierungen nur auf die dort befindlichen Instruktionen an.

- Algebraische Optimierung
- Gemeinsame Teilausdrücke
- Kopienfortschreibung (Original statt Kopie)
- Konstantenfaltung
- Strength Reduction (Ersetzen teurer durch billige Ausdrücke - bsp Multiplikation durch Additionen) - Beispiele 01-33
  Schönes Beispiel - Adressberechnung beim Array-Zugriff $a[i, j]$, also i äußere Schleife, j innere:
  $base\_a + ((i - lo1) * (hi2 - lo2 + 1) + j - lo2) * w$, wobei $w, lo1, hi1, lo2, hi2$ bekannt
  $base\_a - (lo1 * (hi2 - lo2 + 1) - lo2) * w$ +i * (hi2-lo2+1)*w $+j * w$
  Also eine Aufteilung in einen konstanten Wert, einen nicht von j abhängigen Wert und einen sich stetig verändernden

  1-43: j: Schleifen-Laufvariable kann eliminiert werden, da im Rumpf bereits andere, von j abhängige vars existieren, dank derer man eine andere Abbruchbedingung definieren kann
- Ein Array-Zugriff ist teuer. Die einmal berechnete Adresse kann jedoch durch Instruktionsumordnung für zukünftige Zugriffe verwendet werden.
- Vereinfachte Annahme in CB2: Es gibt keine Aliase (verschiedene Namen für eine Speicherstelle), für Optimierungen ist die Kenntnis aller Aliase wichtig.

Globale Optimierung:

Gemeinsame Teilausdrücke. Wiederverwendung von Werten anderer Grundblöcke (Schleifenoptimierung), vorziehen von identischem Code (Achtung. siehe Transformations-Legalität). Flussgraph (Kontrollfluss + Datenfluss) muss hierfür bekannt sein (keine Änderung und tatsächliche Berechnung der gemeinsamen Teilausdrücke)

Legalität einer Transformation:

Semantisch korrektes Originalprogram und die transformierte Version liefert für alle identischen Ausführungen gleiche Ausgaben. Hier sind (je nach Sprachdespezifikation) anscheinend sogar Überläufe relevant und müssen nach der Trasformation identisch ablaufen. Je nach dem ob soetwas definiertes Verhalten ist oder nicht. Beachtung der Genauigkeit von Gleitkommaarithmetik. Beim Vorziehen von Instruktionen ist zu beachten: Zugriff auf Speicherbereiche ist unter Umständen nicht immer legitim und nur in einzelnen Fällen erlaubt. Selbes gilt bei Trap-Instruktionen, wie Division durch 0.

Aliase:

verschiedene Namen für eine Speicherstelle

In Schleifen lohnt es sich häufig die Schleifenzähler zu entfernen und die Abbruchbedingung entsprechend anzupassen. Das geht, wenn Variablen konstant abhängig zu dem Counter gesetzt werden. - Beispiel 01-45ff.
Wichtig ist es den Kontrollfloss, den Datenfluss zu verstehen und zu erkennen wann Transformationen erlaubt sind.

2 Kontrollflussgraph, Dominatoren, Bestimmung von Schleifen, Reduzierbarkeit

Datenabhängigkeiten:

Bei diesen ist eine Umordnung im Allgemeinen verboten.

- read after write
- write after write
- write after read

Weg von Grundblöcken zu Schleifen und Strukturen:

1. Grundblock
2. Kontrollflussgraph
3. Dominatorbaum
4. Dominanzgrenze
5. Hierarchischer Flussbaum
6. Schleifen und Strukturen

Kontrollflussgraph KFG:

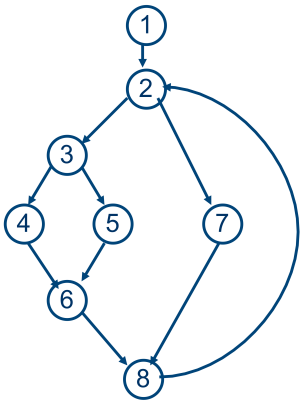
Gerichteter Graph, dessen Knoten Grundblöcke sind, die durch eine Kante im KFG verbunden sind, genau dann wenn in einer Programmausführung der zweite unmittelbar nach dem ersten ausgeführt werden kann. Künstlich erweitert wird er häufig mittels eines Entry- und Exit-Knotens und einer Kante von Entry zu Exit.

(strikte) Dominanz und Dominator (bei Kontrollflussgraph) - fig. 2:

$x \in G$ *dominiert* y, genau dann wenn von der Wurzel zu y x durchlaufen werden muss ($x \geq\geq y$).
x ist der *Dominator* von y. Die Dominanz ist *strikt* ($x >> y$) $\Leftrightarrow x \geq\geq y \wedge x \neq y$
Jeder Knoten dominiert sich selbst, der Wurzelknoten wird nur von sich selbst dominiert.

ImmDom[y], Dominatorbaum - figs. 2 and 3:

immediate dominator ImmDom[y] ist der nächste Dominator von y. Dieser ist eindeutig.
Der *Dominatorbaum* lässt sich aus dem ImmDom ablesen. Knoten y ist Kind seines ImmDom[y]. Weiterhin enthält er dieselben Knoten wie der KFG.

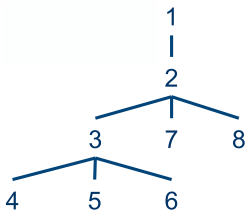


(a) KFG

1	$>>$	$2, 3, 4, 5, 6, 7, 8$
2	$>>$	$3, 4, 5, 6, 7, 8$
3	$>>$	$4, 5, 6$
4	$>>$	$-$
5	$>>$	$-$
6	$>>$	$-$
7	$>>$	$-$
8	$>>$	$-$

(b) strikte Dominanz

Abbildung 1: Dominatoren ablesen



(a) Dominatorbaum

y	$ImmDom[y]$
2	1
3	2
4	3
5	3
6	3
7	2
8	2

(b) Immediate Dominator

Abbildung 2: Dominator Baum - Immediate Dominators

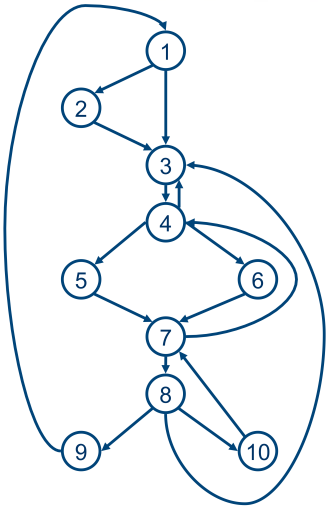
(wichtige) Region - figs. 4 and 5:

Eine *Region* ist ein Untergraph des KFG mit einem Header, der als einziger die Möglichkeit hat über mehr als eine Eingangskante von außen zu verfügen. Wichtige Regionen (die Schleifen sein können) lassen sich mithilfe des Dominatorbaums auslesen.

Eine *wichtige Region* entspricht einer maximalen Region, deren Eingangsknoten alle Knoten der Region dominieren. Sie bilden einen *hierarchischen Flussbaum*. Mögliche Schleifenköpfe - Köpfe der wichtigen Regionen

Rückwärtskante:

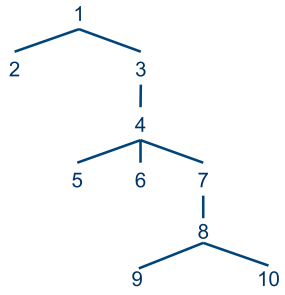
Eine Kante (n, d) heißt genau dann *Rückwärtskante* eines KFG, wenn $d \geq\geq n$. Diese Dominanz ist tatsächlich wichtig, da sonst mehrere Einträge existieren, was schwer optimierbar ist.



(a) KFG

y		$Dominators$	$ImmDom[y]$
1	>>	$2-10$	–
2	>>	–	1
3	>>	$4-10$	1
4	>>	$5-10$	3
5	>>	–	4
6	>>	–	4
7	>>	$8-10$	4
8	>>	$9-10$	7
9	>>	–	8
10	>>	–	8

(b) Dominatoren



(c) Dominatorebaum

Abbildung 3: Zweites Beispiel (Immediate) Dominator

natürliche Schleife - figs. 6 and 7:

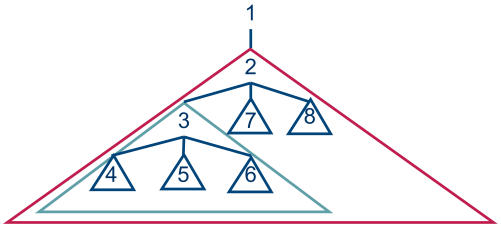
Sei (n, d) eine Rückwärtskante. Dann sind die folgenden Knoten Teil der *natürlichen Schleife*: n, d und $d \geq\geq k$ und es gibt einen Pfad von k zu n, der nicht durch d geht. Also: Alle Knoten, die vom Schleifenkopf dominiert werden und man von dort zum Startpunkt der RW-Kante gelangen kann.

Es genügt also Rückwärtskanten und die zugehörigen natürlichen Schleifen in den wichtigen Regionen zu suchen. Aus einer Rückwärtskante kann die zugehörige Schleife gebildet werden, indem von dem Schleifenende sämtliche Vorgänger, die vom Schleifenkopf dominiert werden, hinzugefügt werden (und dessen Vorgänger, ...), bis man beim Schleifenkopf angekommen ist.

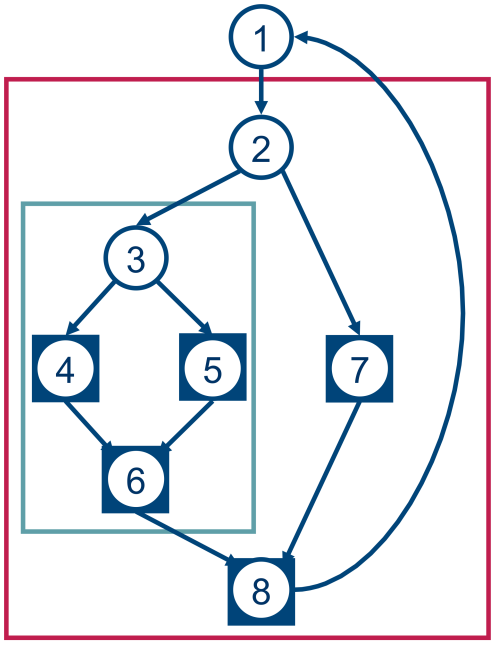
unsaubere Region, unnatürliche Schleife, Reduzierbarkeit:

Befindet sich in einem KFG ein Zyklus *(unnatürliche Schleife)* so ist dies eine *unsaubere Region* und lässt sich nicht *reduzieren*, da keine Rückwärtskante existiert. Dies ist ein zu überprüfendes Kriterium bevor Optimierungen am KFG vorgenommen werden. Ist im Grunde nur mit goto möglich.

Die Überprüfung auf unnatürliche Schleifen lässt sich durchführen, indem sämtliche Rückwärtskanten aus dem KFG entfernt werden. Bilden die verbleibenden (Vorwärts)-Kanten einen azyklischen Graphen, in dem jeder Knoten von der Wurzel aus erreichbar ist, dann ist der KFG frei von unnatürlichen Schleifen. Also der Graph ist *reduzierbar*. (Transformationen s.u.)



(a) Wichtige Region - Dominatorbaum



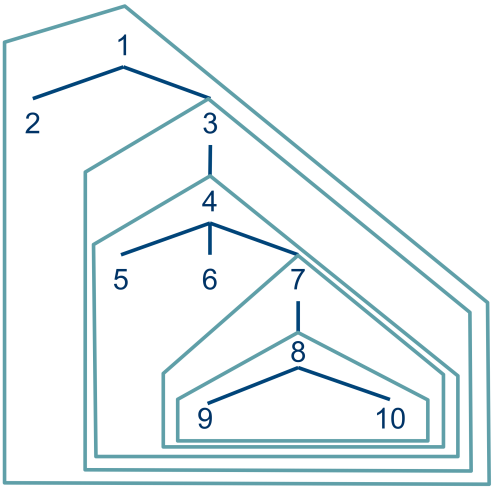
(b) Wichtige Region - KFG

Abbildung 4: Wichtige Region

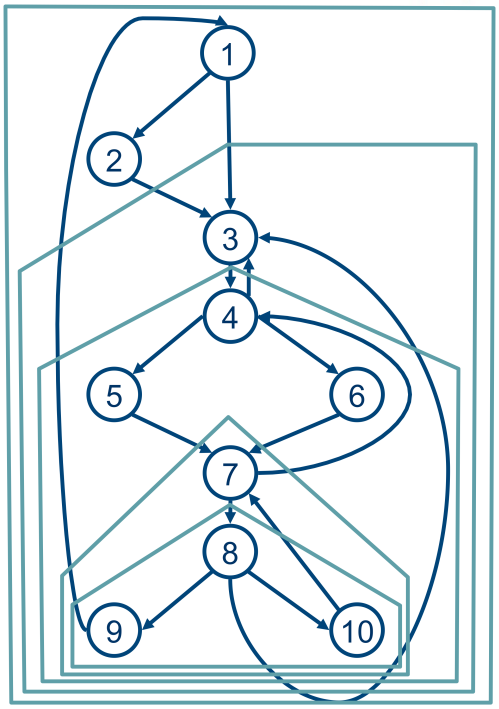
Reduktions-Transformationen - Test auf unnatürliche Schleifen - 02-39ff:

- $T1$: Selbstschleife, also Kante (n, n). Kann gelöscht werden
- $T2$: n verfügt über eindeutigen Vorgänger m. Verschmelzung beider Knoten zu m'

Reduzierbarkeit ist gegeben, wenn eine $T1 - T2$-Folge gefunden werden kann, bei der Reduktion auf einen Knoten möglich ist.



(a) Wichtige Region - Immdom



(b) Wichtige Region - KFG

Abbildung 5: Wichtige Region - Finden von Schleifen, es gibt mehr Regionen als hier eingezeichnet, $\{3,4\}, \{4,5,6,7\}$ sind keine Regionen, da mehrere Eingänge, daher sind das auch keine Schleifen im eigentlichen Sinne.

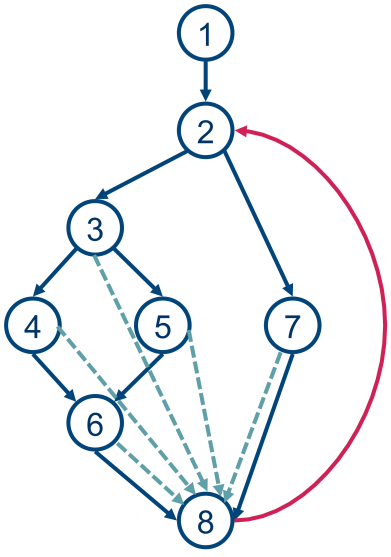


Abbildung 6: Natürliche Schleife

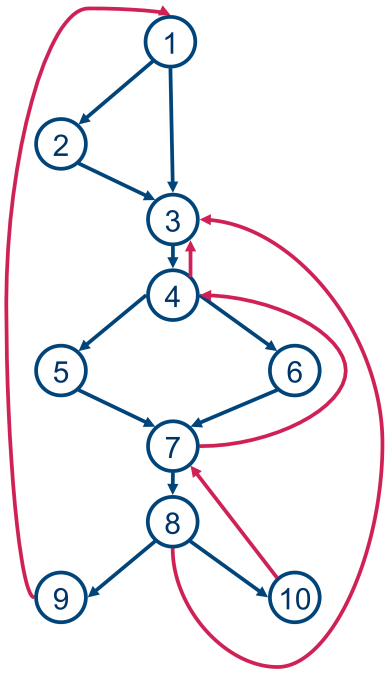


Abbildung 7: Rückwärtskanten und natürliche Schleifen. Vergleich mit wichtigen Regionen: fig. 5b
Natürliche Schleifen der Rückwärtskanten:
$(10, 7) : \{7, 8, 10\}$
$(9, 1) : \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
$(8, 3) : \{3, 4, 5, 6, 7, 8, 10\}$
$(7, 4) : \{4, 5, 6, 7, 8, 10\}$
$(4, 3) : \{3, 4, 5, 6, 7, 8, 10\}$

3 Berechnung von Dominatoren mit Fixpunktalgorithmus und Lengauer-Tarjan, Dominanzgrenzen

Berechnung der Dominanz:

Gegeben sein ein KFG mit $|V|$ Knoten und $|E|$ Kanten, ein einfacher Algorithmus benötigt $\mathcal{O}(|E| \cdot |V|^2)$, wohingegen kompliziertere lediglich $\mathcal{O}(|E| \cdot \log(|V|))$ benötigen und in der Praxis häufig sogar nur $\mathcal{O}(|E|)$.

Die Dominatoren eines Knotens lassen sich aus den Dominatoren der Vorgänger bestimmen.

Algorithm 1 Iterativer Fixpunkt-Algorithmus

Terminierung, da $D(v)$ in jedem Schritt nur kleiner oder gleich bleiben kann und $D(S)$ eine kleine Menge ist und die Verkleinerung frisst sich von der Wurzel zu den Blättern durch.

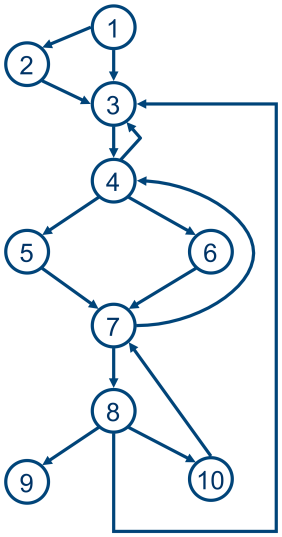
Terminiert am schnellsten, wenn Knoten in DFS besucht werden, weil dann schnell eine Verkleinerung der Nachfolger erfolgt. Die Komplexität ist in $\mathcal{O}(|E| \cdot |V|^2)$, da jede Kante einmal untersucht wird und die Mengenoperationen den Aufwand $|V|$ haben.

Beispiel - fig. 8

▷Input: KFG, mit V Knoten, E Kanten und Startknoten S
▷Output: Die Menge der Dominatoren von $v : D(v)$ für jeden Knoten $v \in V$
▷Idee: $D(v)$ initial zu groß wählen und Mithilfe der Vorgänger-Dominatoren $D(p)$ sukzessive verkleinern bis Stabilität erreicht.

$D(S) := \{S\}$
for $v \in V \setminus \{S\}$ **do**
  $D(n) := V$
end for

while Änderung in einem $D(v)$ **do**
  for $v \in V \setminus \{S\}$ **do**
    ▷Berechnung aus Dominatoren aller Vorgänger
    $D'(v) = \{v\} \cup \bigcap_{(p,v) \in E} D(p)$
  end for
end while



(a) CFG

		D'(v) nach Iteration		
v	$Pred(v)$	0	1	2
1	–	$\{1\}$	*same*	*same*
2	1	$[1, 10]$	$\{1, 2\}$	*same*
3	$1, 2, 4, 8$	$[1, 10]$	$\{1, 3\}$	*same*
4	$3, 7$	$[1, 10]$	$\{1, 3, 4\}$	*same*
5	4	$[1, 10]$	$\{1, 3, 4, 5\}$	*same*
6	4	$[1, 10]$	$\{1, 3, 4, 6\}$	*same*
7	$5, 6, 10$	$[1, 10]$	$\{1, 3, 4, 7\}$	*same*
8	7	$[1, 10]$	$\{1, 3, 4, 7, 8\}$	*same*
9	8	$[1, 10]$	$\{1, 3, 4, 7, 8, 9\}$	*same*
10	8	$[1, 10]$	$\{1, 3, 4, 7, 8, 10\}$	*same*

(b) Tabelle

Abbildung 8: Beispiel Iterativer Fixpunkt Algorithmus - algorithm 1 - Dieser endet sobald keine Änderungen mehr erfolgen
um Dominatorbaum zu erhalten ist noch der ImmDom zu berechnen

Arten übriger Kanten in einem Tiefenbaum T - Beispiel fig. 10:

- Rückschreitende Kante:
  - Führen zu einem Vorgänger in T
  - laufen zu dem Vorgänger weiter nach oben
  - dfn-Nummer wird kleiner
- Fortschreitende Kante:
  - Führen zu einem Nachfolger in T
  - laufen vorwärts nach unten
  - dfn-Nummer wird größer
- Kreuzkante:
  - Führen in einen bei der DFS früher besuchten Ast. (Weil die Tiefensuchnummern rechts meistens in der Darstellung größer sind, als links, führen diese Kanten in der Regel von rechts nach links)

Algorithm 2 Spannender Tiefenbaum -Fixpunkt-Algorithmus
Berechnet gleich den ImmDom. Im Allgemeinen gibt es zu G mehrere spannende Tiefenbäume - Beispiel fig. 9

▷Input: G, v
▷Output: $DFS(G, v)$
▷dfn nummeriert die Knoten in der DFS-Reihenfolge
$dfn := 1$
mark v
$v.dfn := dfn$
$dfn++$
▷Garantiert den einmaligen Besuch eines Knotens und dessen Unterbaums (Existenz mehrerer Pfade von der Wurzel zu diesem Knoten)
for $(v, w) \in E$ **do**
  if w is unmarkiert **then**
    ▷T enthält diejenigen Kanten $\in G$, die noch zu unmarkierten Knoten führen. Diese Kanten verbinden zwei Knoten mit wachsender dfn
    $T.add(v, w)$
    ▷Nachfolger gleichwertig $\Rightarrow$ mehrere Besuchsreihenfolgen möglich
    $DFS(G, w)$
  end if
end for

Dominanz im spannenden Tiefenbaum:

Alle Pfade zu einem bestimmten Knoten zu berechnen (inklusive sämtlicher Umwege) ist sehr aufwändig, weshalb man sich auf die ImmDoms beschränkt. Umwege starten stets auf einem Knoten des direkten Weges von der Wurzel zu dem entsprechenden Knoten. Gibt es also einen direkten Weg von $1 \to 2 \to 3 \to 4$, so können Umwege zu 4 lediglich bei den Knoten 2 und 3 starten.

Semi-Dominator (SemDom):

SemDom$[y]$ = min {direkter Vorgänger, Umwege} Für sämtliche Knoten auf dem direkten Weg zu y gilt $v_i.dfn > y.dfn$. Weshalb derjenige Startknoten einer Umgehung des T-Pfades gesucht wird, der möglichst nah an der Wurzel beginnt. Dh. für jede eingehende Kante (p, y) von y ist p ein Kandidat für SemDom$[y]$. Da Dominanz notwendig ist, können Kreuzkanten nicht Teil der Optionsmenge für SemDom sein. Also reicht es neben den direkten Vorgängern die Umwege über fortschreitende Kanten zu betrachten. Dann betrachte für alle $u \in \{v,$ alle Vorgänger von v auf $T\}, u.dfn > y.dfn$. Möglicher Kandidat für SemDom$[y]$ ist dann das Minimum.

Folglich ist es sinnvoll die SemDoms in Reihenfolge fallender dfn-Nummern auszurechnen Beispiele 03-29ff

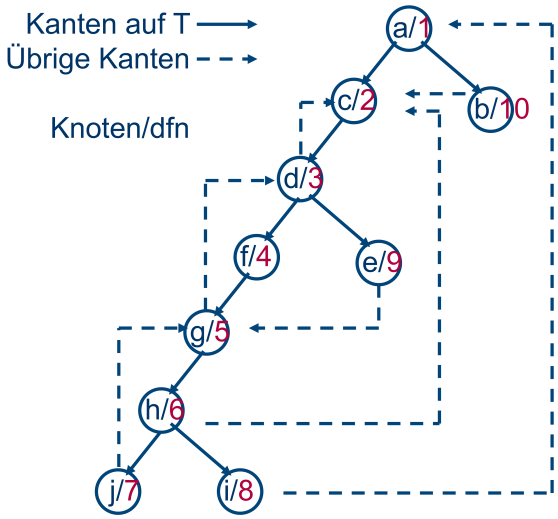


Abbildung 9: Spannender Tiefenbaum Beispiel - algorithm 2
Der zugehörige KFG: fig. 8a

Die Berechnung des SemDoms:
Für alle übrige unmittelbaren Vorgänger (letzte Kante ist eine übrige) v von w - Im Beispiel auf 03-29 fig. 10 $w = 5$, betrachte sämtliche Vorgänger, für die die dfn kleiner ist, als die von w. Das sind hier $6, 7$, nicht jedoch 2, dessen dfn kleiner ist. Das Minimum der $SemDom[6]$ und $SemDom[7]$ ist ein Kandidat für $SemDom[w] = SemDom[5].SemDom[7] = 3$ (direkter Vorgänger), $SemDom[6] = \min\{5, SemDom[7]\} = 3$

warum ist hier 5 in der Suche enthalten? es gibt doch keine übrige Kante von 5 nach 6?!?

SemDom[5] scheint auch 2 zu sein, da 03-30 sagt, dass sich 1,8,3 um SemDom[5] herummogelte, jedenfalls scheint SemDom[3] = 1 und SemDom[4] = 3 zu gelten

Um aus dem SemDom den ImmDom zu bestimmen:
Von $SemDom[w]$ bis zu w die Vorgänger u untersuchen
Finde u mit dem kleinsten $SemDom[u]$
If $SemDom[w] == SemDom[u] : ImmDom[w] := SemDom[u]$
sonst: $ImmDom[w] := ImmDom[u]$

Kontrollflussabhängigkeit:

Sei ein Verzweigungsknoten v mit den direkten Nachfolgern a, b gegeben. Ein Knoten y heißt genau dann *kontrollflussabhängig* von v (κ_{vy}), wenn es mindestens einen Pfad von a zum Exit-Knoten gibt, der nicht über y geht und jeder Pfad von b zum Exit-Knoten über y geht.

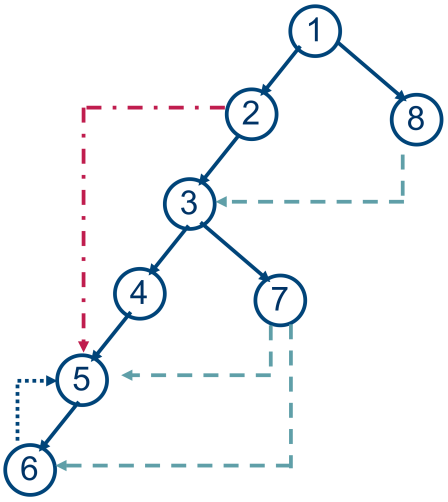


Abbildung 10: Beispiel anderer Kanten
Die Knoten-Nummern entsprechen den jeweiligen dfn. Rückschreitende Kanten, Fortschreitende Kanten, Kreuzkanten

Also salopp: In v wird die letzte Entscheidung gefällt, ob y durchlaufen wird, oder nicht.

das “letzte” in ”die letzte Entscheidung“ müsste falsch sein, da es ja trotzdem einen Pfad in a geben kann, der durch y zum Exitknoten führt. das müsste ein fehler im Foliensatz sein

- Beispiel fig. 11

Dominanzgrenze ($DG[x]$) eines Knotens x des KFG - Beispiel fig. 13a:

Ist die Menge aller Knoten y des KFG, wobei x mindestens einen Vorgänger von y dominiert, aber y selbst nicht **streng dominiert**. D.h. insbesondere ein Knoten kann eine Dominanzgrenze von sich selbst sein!!! Wichtig bei der Überlegung und dem Vergleich mit DG_{local}: x dominiert sich selbst, local betrachtet also nicht die $Succ(Succ(x))$, sondern wirklich nur die $Succ(x)$.

$$DG[x] = \{y | \exists u \in Pred(y) : x \geq\geq u, x \not> \not> y\}$$

Diese Dominanzgrenzen entsprechen der Kontrollflussabhängigkeit des inversen Graphen (vgl. fig. 12)

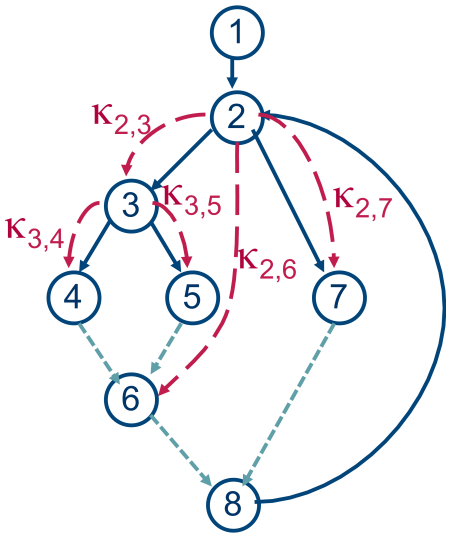


Abbildung 11: Beispiel Kontrollflussabhängigkeiten - $\kappa_{v,y}$, v Verzweigungsknoten, y kontrollflussunabhängiger Knoten. Es besteht keinerlei Kontrollflussabhängigkeit zwischen 3 und 6, besteht darüberhinaus keine Datenabhängigkeit, so kann 6 auch vor 3 ausgeführt werden.
Kontrollflussabhängig, erzwingen keine Reihenfolge

Berechnung einer Teilmenge $DG_{local}[x]$ der Dominanzgrenzen $DG[x]$ - Beispiel fig. 13a:

$DG_{local}[x] = \{y \in Succ(x) | ImmDom[y] \neq x\}$ = Direkte Nachfolger von x, die nicht mehr von x strikt dominiert werden. DG_{local} approximiert also DG
$DG_{local}[x] \subseteq DG[x]$
Denn $DG[x]$ = Nachfolger (nicht nur direkte), die gerade nicht mehr dominiert werden. Achtung es geht bei local um die direkten Nachfolger von x, nicht um dessen Nachfolger!

Beispiel: 03-45 auch fig. 13

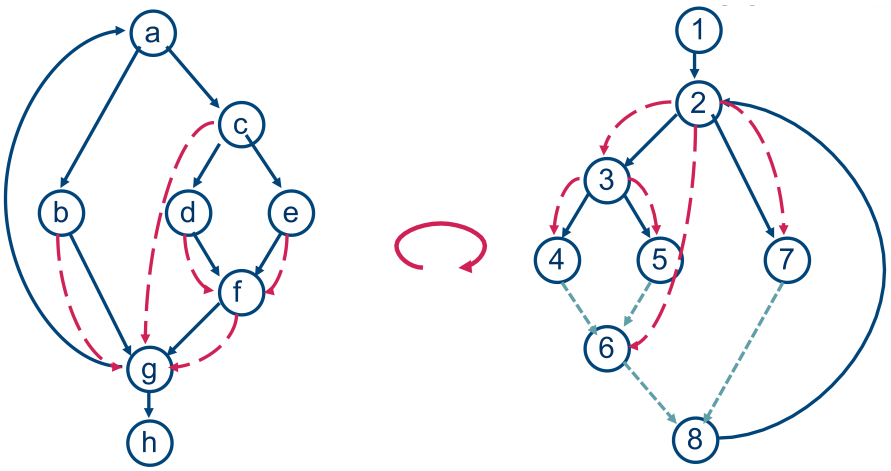
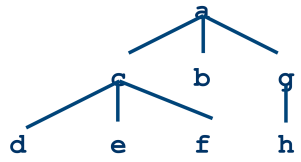


Abbildung 12: Dominanzgrenze vs. Kontrollflussabhängigkeit

($x \in DG_{iKFG[y]}, x \in$ Dominanzgrenze von y im inversen KFG.)
$\Leftrightarrow$
($\kappa_{xy} \in$ KFG, y kontrollflussabhängig von x im KFG)

x	$DG[x]$	$DG_{local}[x]$
a	a	–
b	g	g
c	g	–
d	f	f
e	f	f
f	g	g
g	a	a
h	–	–

(a) $DG[f] = \{g\}, DG_{local}[f] = \{g\} \subseteq DG[f]$ und
$DG[c] = \{g\}, DG_{local}[c] = \emptyset \subseteq DG[c]$ und
$DG[d] = DG[e] = \{f\} \not\subset DG[c]$
Also tragen die dominierten Nachfolger zu der Dominanzgrenze bei.



(b) Zugehöriger Dominatorbaum zum Ablesen der ImmDoms: $DG_{local}[x] = \{y \in Succ(x) | ImmDom[y] \neq x\}$

Abbildung 13: Dominanzgrenze ($DG[x]$) und dessen Teilmenge $DG_{local}[x]$ - Beispiel zum KFG in fig. 12

$$DG_{up}[x,z] = \{y \in DG[z] | ImmDom[y] \neq x\}$$
$$DG[x] = DG_{local}[x] \cup \bigcup_{\substack{z \in V \\ ImmDom[z]=x}} DG_{up}[x,z]$$

Abbildung 14: $DG_{up}[x,z]$ und der Zusammenhang mit Dominanzgrenzen ($DG[x]$) - Lässt sich wohl bottom-up aus dem Dominatorbaum ablesen - Beispiel 03-49f

4 Datenflussanalyse, typische Datenflussprobleme, Wertnummerierung, Datenflussgleichungen, Elimination von Array-Bereichstests

Datenflussanalyse:

Variableninhalt berechnen und Fluss dessen durch den KFG - dient als Anwendbarkeitsnachweis für Optimierungen. Verschiedene Einstufungen: gilt sicher, gilt höchstwahrscheinlich, gilt möglicherweise, gilt selten, gilt sicher nie. Dabei gilt es zu beachten, dass der Nachweis möglichst aggressiv sein sollte, aber *nicht zu scharf*, um nur wenige Fälle fälschlicherweise auszuschließen und *möglichst vollständig* - nur wenige Fälle möglicher Anwendung sollen übersehen werden.

Typische Datenflussprobleme:

- Erreichende Definition
- Konstantenweitergabe
- Kopienfortschreibung
- Verfügbare Ausdrücke
- Lebendige Variablen
- Erreichbare Nutzungen
- Vorhersehbare Ausdrücke

Starke vs. schwache Variablendefinition:

- *Starke*: Wissen $\mapsto$ sichere Zuweisung eines Wertes/Ausdruck zu einer Variablen
- *Schwache*: Möglich $\mapsto$ Werteveränderung einer Variable ist möglich - umfasst starke Variablendefinition

Erreichende Definition - reaching definition:

Eine Variablendefinition eines Knotens S *erreicht* einen anderen Knoten D gdw. es mindestens einen Pfad $S \to \cdots \to D$ gibt, in dem die Variable abgesehen von S nirgends schwach definiert wird, also es ausgeschlossen ist, dass der Wert sich ändert. Betrachtung von Lesen zur Zuweisung rückwärts durch den KFG.

Gibt es nur eine erreichende Definition oder sind diese alle gleich, dann sind Konstantenweitergabe und Kopienfortschreibung möglich.

Wissen wird vorwärts durch den KFG gesammelt/getragen. Hierfür sind Mengenoperationen recht hilfreich.

Verfügbarer Ausdruck - available expression:

Ausdruck ist in einem Knoten des KFG *verfügbar*, gdw. er auf jedem Pfad vom Entry zu diesem Knoten ausgewertet wird und anschließend keine der in diesem Ausdruck gelesenen Variablen schwach definiert wird.

Lebendigkeit vs. tote Variablen - live/dead variable:

Variable V ist in einem Knoten D *lebendig*, wenn es einen Pfad von diesem Knoten D zum Exit oder zu einer schwachen Definition von V gibt, auf dem die Variable gelesen wird. Andernfalls ist diese Variable V ab dem Knoten D *tot*.

Entfernung von Berechnungen toter Variablen kann Registerbelegung einsparen und die Sammlung des Wissens erfolgt rückwärts.

Erreichbare Nutzung:

Finden aller Nutzungen einer Variable. Eine Variablennutzung heißt *erreichbar* von einer Variablendefinition D, wenn es einen Pfad von D zum lesenden Zugriff gibt, auf dem die Variable nicht schwach definiert, also nicht manipuliert wird. Betrachtung von Zuweisung zum Lesen vorwärts durch den KFG.

vorhersehbare Ausdrücke - {anticipatable, predictable, down-safe, very busy} expression:

Ausdruck ist in einem Knoten D *vorhersehbar*, wenn er auf jedem Pfad von D zum Exit-Knoten ausgewertet wird und anschließend keine der im Ausdruck gelesenen Variablen schwach definiert wird. Also eine Spezialisierung des verfügbaren Ausdrucks (available expression). Hier ist die Auswertung notwendig, ganz gleich welcher Pfad durchlaufen wird.

Vorhersehbare Ausdrücke können vorgezogen werden und somit Register-Engpässe entschärfen. Das Wissen wird rückwärts gesammelt.

Vorverarbeitung:

- sicher ”must“ - Eigenschaft muss auf allen Eingangskanten erfüllt sein
- möglich ”may“ - Eigenschaft muss auf mindestens einer Eingangskante erfüllt sein

Um diese Vorverarbeitung sinnvoll und elegant vornehmen zu können verwendet man häufig Bitfeld, in denen gemerkt wird ob ein spezifischer Wert gesetzt ist oder nicht.

- Anfangsbelegung: false
- Setzen des Bits bei Variablendefinition
- Zurücksetzen bei schwacher Definition der Variable

Richtung	oder/möglich	und/sicher	sonst./sicher
Vorwärts	Erreichende Def.	Verfügbare Ausdr. Kopienfortschreibung	Konstantenweitergabe
Rückwärts	Lebendige Var.	Vorhersehbare Ausdr.	

Tabelle 1: Zusammenfassung Analyse

- Vorverarbeitung: *oder*, alle zuvor erreichenden Definitionen sind auch nach der Zusammenführung erreichend. - Beispiel 04-31
- Konstantenweitergabe: *und*, ist logisch, da nur eine oder nur gleiche erreichende Definitionen erlaubt sind. Es wird auch der konstante Wert mitgenommen
- Kopienfortschreibung: *und*, Bit für jede Wertgleichheitsbeziehung, das wieder zurückgesetzt wird, wenn original oder Kopie schwach definiert wird. Beispiel: X := Y, resultiert in dem true Bit in der Gleichheitsbeziehung.
- Verfügbare Ausdrücke: *und*, Bit für jeden (Teil-)Ausdruck, setzen bei Auswertung des Ausdrucks, zurücksetzen bei schwacher Definition einer in dem Ausdruck zu lesenden Variable
- Lebendige Variablen: *oder*, Bit pro Variable, setzen dieses bei Nutzung der Variable und löschen bei **starker** Definition dieser
- Vorhersehbare Ausdrücke: *und*, Bit pro (Teil-)Ausdruck, setzen bei Auswertung und löschen bei schwacher Definition mindestens einer für den Ausdruck zu lesenden Variable.

Bitvektoren sind aber prinzipiell doof, weil Transformationen zu unglaublichem Overhead führen weil große Teile dessen für einzelne Knoten gar nicht benötigt werden. Weiterhin ist die komplette Neuberechnung notwendig, nachdem eine Transformation durchgeführt wurde, weil die entsprechende Manipulation dieser ziemlich tricky ist. Besser sind Definitions-Nutzungs-Graphen.

Enthält der KFG Zyklen, so reicht es nicht mehr aus den KFG einmal vorwärts und einmal rückwärts zu durchlaufen, sondern so häufig, bis keine Änderung mehr erfolgt (iterativer Fixpunkt-Algorithmus) - Dieser kann verschiedene Lösungen liefern, da mehrere existieren können. Die gefundene Lösung muss nicht die beste sein.

```
1 //JVM-runtime-checks
2 i := 3
3 if (i <= N){
4    do{
5       if (i < 0) goto JAIL
6       if (i > N) goto JAIL
7       tmp := OLD[i]
8       if (i < 0) goto JAIL
```

```
9         if (i > N) goto JAIL
10        NEW[i] := tmp
11        i := i+1
12      }while(i > N)
13  }
```

Listing 1: Anwendungsbeispiel Iterativer Fixpunkt-Algorithmus
Weniger Laufzeit-Array-Bereichstests - fig. 15

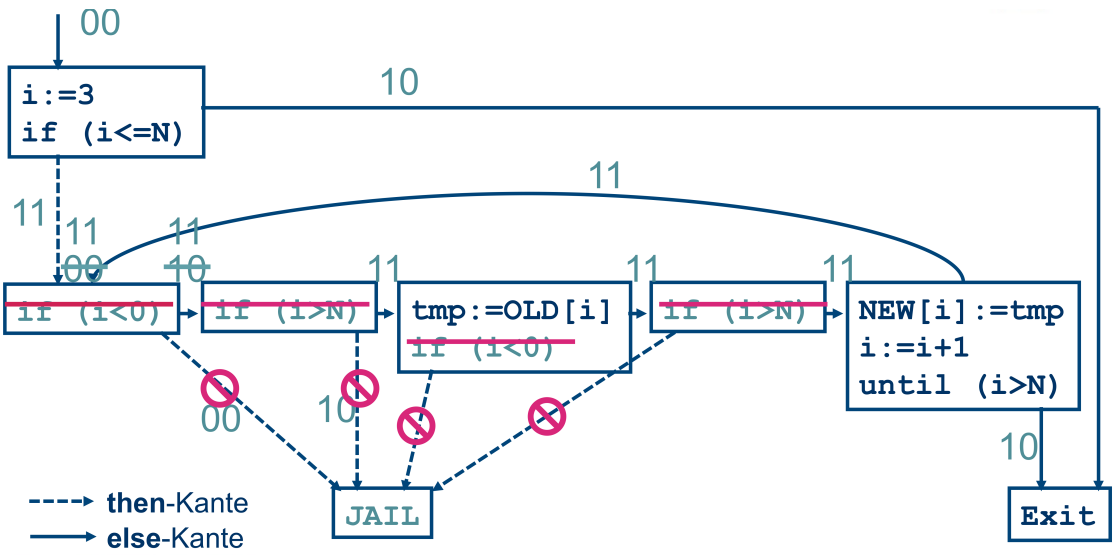


Abbildung 15: Anwendungsbeispiel Iterativer Fixpunkt-Algorithmus
listing 1 - Bits: $b1 : i \geq 0$, $b2 : i \leq N$ bei unbekannt restriktiv

5 SSA-Form

Alternativen zu Bitvektoren zur Speicherung von Informationen:

- Definitions-Nutzungs-Graph (DU-Chain)
- Nutzungs-definitions-Graph (UD-Chain)
- Variablen-Netze
- SSA - änderungsfreundliche, kompakte Basis für effizientere Datenflussanalyse

SSA - Static Single Assignment:

Programm liegt in *SSA-Form* vor, gdw, jeder Variable nur einmalig ein Wert zugewiesen wird. Wobei die Zuweisung auch innerhalb einer Schleife liegen kann und somit zur Laufzeit häufiger erfolgt.

Phi-Funktion:

Bei Kreuzungen mit unterschiedlichen Zuweisungen zur selben Variable lässt sich diese Variable nicht trennen. Entsprechend gibt es nun eine Orakelfunktion (Φ), die den tatsächlich zugewiesenen Wert korrekt errät. Also wird in den Verzweigungspfäden tatsächlich jeder Variabel lediglich einmal einen Wert zugewiesen. $if\ a_1 = x\ else\ a_2 = y;\ a_3 = \phi(a_1, a_2)$

Diese Phi-Funktion erzeugt nur dort neue Variablen, wenn einer alten Variable etwas zugewiesen wird, diese anschließend ausgelesen wird und wenn der Wert dieser variable je nach Kontrollfluss unterschiedliche Werte enthielte. Diese Bestimmung erfolgt an den *Dominanzgrenzen*, an denen Φ-Funktionen eingefügt werden, wobei in den Exitknoten keine Phi-Funktion eingefügt wird.

Das aufwändige dabei ist die Umbenennung der folgenden Variablen mit dem neuen Namen und der iterative Prozess, da auch hier für die neuen Variablen eine Dominanzgrenze existiert.

Elimination unnötiger Phi-Funktionen: Variablen nur dann betrachten, wenn sie in den darüberliegenden Zweigen verändert wurde und nach diesem Knoten lebendig ist. Weiterhin können Phi-Funktionen eliminiert werden, wenn sämtliche Eingänge über gleiche Werte verfügen.

Dominanzgrenzenkriterium - Beispiel 05-16ff:

Enthält ein Knoten S eine Definition der Variable v, so muss jeder Knoten in der Dominanzgrenze von S eine Φ-Funktion für v enthalten. Die Berechnung der Dominanzgrenzen muss iterativ erfolgen, da auch für neue Variablen die Dominanzgrenzen zu bestimmen sind und entsprechende Φ-Funktionen zu erzeugen.

Zweites Verfahren - Wertnummerierung - Beispiele 05-32ff:

Die SSA wird aus dem KFG erzeugt. Jeder Grundblock erhält eine Wertnummer. Beim Lesen von Variablen werden die Vorgänger gefragt - wissen die den Wert auch nicht, so muss dieser rekursiv berechnet werden.

Neue Wertnummer bei mindestens einem unbesuchten Vorgänger ⇒ neue (vorläufige) Phi-Funktion, die vielleicht noch einen Wert erhält, wenn nicht dann kann die Funktion anschließend entfernt werden Ziel ist es den Rekursions-Aufwand zu minimieren Rekursive Bestimmung erfolgt bei nur einem Vorgänger, wenn dieser die Wertnummer noch nicht kennt Die vorläufigen Phi-Funktionen werden nach der kompletten Bestimmung aller Wertnummern betrachtet: Entweder gelöscht, falls keine weiteren Änderungen oder aber Ersetzung durch endgültige Phi-Funktionen.

Das kann dazu führen, dass endgültige Phi-Funktionen bestehen bleiben, die zwischen zweimal den selben Variablen mit selber Wertnummer entscheiden müssen. Aber das lässt sich ebenfalls nachträglich neutralisieren.

1. Initialisierung für aktuellen Block Z
   - $wn_Z(const) := const$
   - Falls Z Startknoten ist, dann setze Wertnummern sämtlicher Tupel in Z auf ungültig
   - Hat Z die beiden Vorgänger X, Y, so
     - Falls $wn_X(t) \neq wn_Y(t)$, dann setze $wn_Z(t) :=$ neue Wertnummer und generiere $wn_Z(t) := \Phi(wn_X(t), wn_Y(t))$
     - Falls $wn_X(t) == wn_Y(t)$, dann setze $wn_Z(t) := wn_X(t)$
   - Hat Z einen Vorgänger X und ist $wn_X(t)$ in diesem undefiniert, so initialisiere $wn_X(t)$ rekursiv
   - Hat Z zwei Vorgänger X, Y und X ist bisher unbesucht, dann setze $wn_Z(t) :=$ neue Wertnummer und $wn_X(t) :=$ neue besondere Wertnummer. Generiere vorläufig $wn_Z(t) := \Phi'(wn_X(t), wn_Y(t))$
2. Für alle Tupel t in Z: Grundblockwertnummerierung
3. Korrektur der Φ'-Funktionen
   - Ersetze besondere Wertnummern in X durch die letzten in X gültigen Wertnummern
   - Ersetze offene Φ'-Funktionen durch abgeschlossene Φ-Funktionen mit den ersetzten Werten
   - Lösche $wn_Z(t) := \Phi'(wn_X(t), wn_Y(t))$, falls t in unbesuchten Blöcken nicht geändert wurde und ersetze dann alle weiteren Verwendungen von $wn_Z(t)$ durch $wn_X(t)$

SSA-Optimierung - Beispiel 05-51ff:

Das kann im dritten Schritt ersetzt werden. Ebenso Kopienfortschreibung + Konstantenweitergabe bei Zuweisung: textuelle Ersetzung der linken im Folgenden durch den rechten Operanden. Abschließend arithmetische Vereinfachung

Konstantenweitergabe in SSA:

- $v_i := c$ - es können alle Vorkommen von v_i durch c ersetzt werden
- $v_i = \phi(c_1, c_2, \ldots c_k)$ kann durch $v_i := c_1$ ersetzt werden, wenn alle c_j über den gleichen Wert verfügen
- bei normaler DFA muss nach Ersetzung erneut DFA gemacht werden, hier genügt der Worklist-Algorithmus

Kopienfortschreibung in SSA:

- $v_i := y_j$ - es können alle Vorkommen von v_i durch y_j ersetzt werden
- Änderung von y_j kann sich nicht auf v_i auswirken, sondern erst nach einer Φ-Funktion auf ein v_k

Lebendigkeit in SSA:

Kann sofort abgelesen werden. Wird ein v_i niemals gelesen, so ist sie tot. Weiterhin sind Wertnummerierungen dank SSA auch auf den gesamten Code und nicht nur Blockweise möglich.

6 SSA-Optimierungen, Rücktransformation aus SSA-Form, Aliase

Beispiel SSA-Umwandlung und Optimierung: 06-03ff
Annahme es gibt keine Endlosschleifen bei der Bestimmung von Rückgabewerten

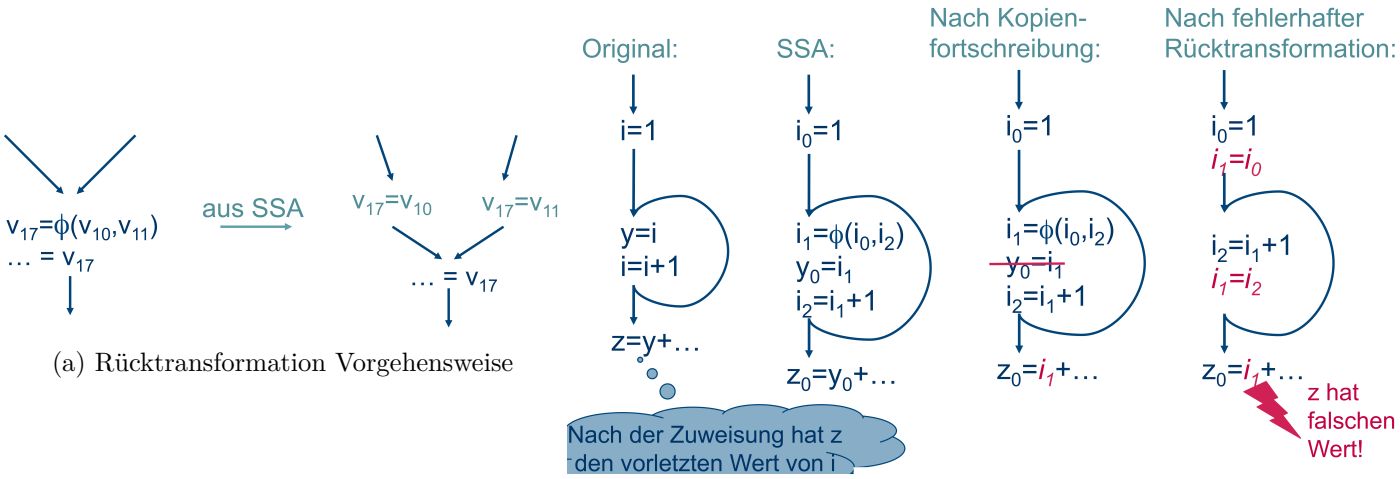
Worklist-Algorithmus:

Alles was nicht als lebendig markiert ist, ist tot. Anweisung wird als lebendig markiert, falls:

- sie I/O macht, in den Speicher schreibt, eine Funktion mit einem Wert beendet, eine andere Funktion aufruft, Seiteneffekte hat.
- Schreibt eine Variable, die von einer anderen lebendigen Anweisung gelesen wird
- Ist eine Verzweigung, von der eine andere lebendige Anweisung kontrollflussabhängig ist.

Probleme:

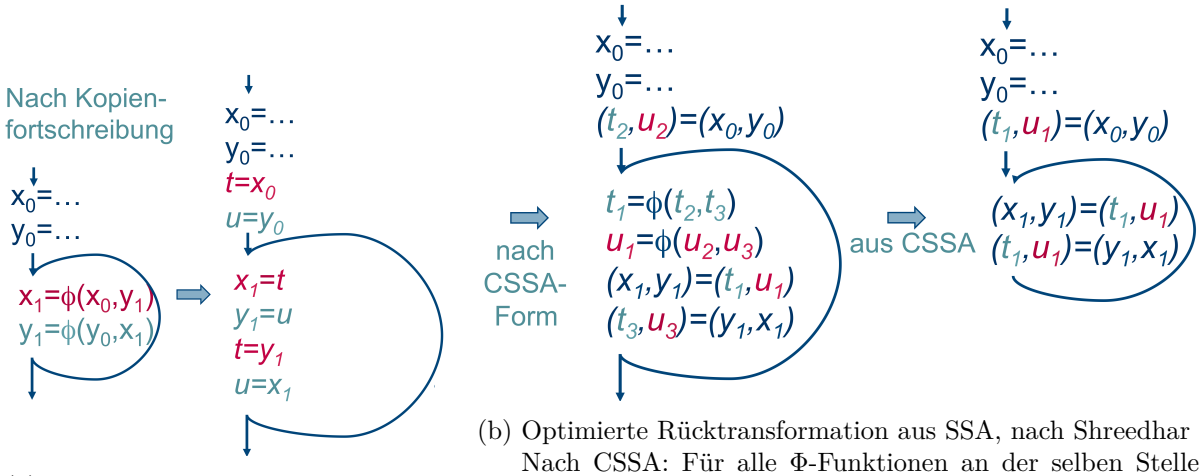
1. Problem 1: Die Φ-Funktion ist nicht implementierbar, es muss eine Rücktransformation stattfinden, um tatsächlichen Code zu erzeugen.
   - Triviale Rücktransformation: Keine Optimierung wurde durch die Φ-Funktion durchgeführt ⇒ Ersetzen der Argumente durch das Ergebnis der Φ-Funktion - das funktioniert nicht immer. Weil Lebendigkeitsintervalle unzureichend betrachtet werden - Die Dominanzbeziehung kann sich ändern (vgl. fig. 16). Überführung in eine *konventionelle SSA-Form (CSSA)*
   - Korrekte Rücktransformation - Beispiel 06-27ff: Im Grunde das selbe, nur mit einer temporären Variable. D.h. vor der Verzweigung wird der neuen temporären Variable der Wert der Φ-Funktion zugewiesen, und in der Verzweigung wird der tatsächlichen Variable der Wert der temporären Variable zugewiesen. Dies erzeugt wahnsinnig viel Overhead, der mittels Optimierung nach Shreedar, reduziert werden kann. Dies erfolgt durch die Betrachtung der Move-Kanten und parallele Kopieroperationen für alle Φ-Funktionen die sich an der selben Stelle eines Blocks befinden. - Beispiel 06-31ff und fig. 17
2. Problem: werden Pointer dereferenziert und dort Werte hingeschrieben, funktioniert SSA nicht, weil der Pointer auf ALLES Zeigen kann, auch mitten rein in irgendwelche Variablen. Falls das Alignmnet auf das Zeigen auf ganze Variablen erzwingt, so müssen alle Möglichkeiten betrachtet werden: a2 = isAlias(p, a1); b2 = isAlias(p, b2)



(a) Rücktransformation Vorgehensweise

(b) Tatsächlicher Fehler

Abbildung 16: SSA - Fehlerhafte Rücktransformation mit der trivialen Rücktransformation
Das Problem ist die Interferierung der Lebendigkeitsintervalle von i_1 und i_2, wodurch die Dominanzbeziehung vertauscht wurde.



(a) Rücktransformation aus SSA - Existenz einer nicht notwendigen Zuweisung

(b) Optimierte Rücktransformation aus SSA, nach Shreedhar
Nach CSSA: Für alle Φ-Funktionen an der selben Stelle eines Blocks gibt es parallele Kopieroperationen. Die Lebensspannen sind paarweise nicht überlappend.
Aus CSSA: Bei der Rücktransformation wird erst Kopienfortschreibung und anschließend eine Sequentialisierung der parallelen Zuweisungen vorgenommen.

Abbildung 17: SSA - Rücktransformation optimiert - Shreedhar

7 Alias-Analyse, Teil 1

Motivation: Aliase verhindern aggressive Optimierung. Insbesondere in SSA-Form, jede Zuweisung zu einer Variable erzwingt bei allen Aliasen an den Dominanzgrenzen Φ-Funktionen und frische Variablen einzuführen.

Aliase sind verschiedene Möglichkeiten auf die selbe Speicherstelle zuzugreifen. Prinzipiell darf der Compiler in C wohl Überlappungsfreiheit annehmen.

Existieren Aliase?:

- gibt es Zeiger
- was können Zeiger referenzieren. Können Zeiger nicht umgesetzt werden
- überlappende Speicherbereiche - union
- Referenzen auf Array/-Abschnitte
- Zeigerarithmetik

restrict keyword in C erlaubt dem Compiler mehr Optimierungen, durch die Garantie des Entwicklers auf die nicht Existenz von Aliasen unter bestimmten Gesichtspunkten

Alias-Analyse ist nicht trivial möglich. Es ist nicht möglich zu sagen “diese alle sind Aliase”, es ist einfacher zu fragen “das hier könnte ein Alias sein”. Es gibt also eine Unterscheidung zwischen

Alias-Wissen-Kategorisierung - Beispiele 07-27f:

- Wahrscheinlichkeit
  - mögliche may-Aliase - also sowohl die Variablen, die bei einem Pfad ein Alias bilden könnten, als auch diejenigen, über die keinerlei Aussage getroffen werden kann
    Führt zu relativ großen Alias-Mengen und behindert dadurch aggressive Optimierungen
  - sichere must-Aliase - betrachtete Variablen sind auf jedem Pfad Aliase
- Betrachtung von KFG?
  - Fluss-ignorierend: Können Variablen irgendwo im KFG aliasen? Fluss-ignorierend interprozedurale Analyse, schnell analysierbar, Problem ist in Teilprobleme zerlegbar und getrennt lösbar. Hierbei wird eine Relation $Alias(a, b)$ eingeführt, die für eine ganze Prozedur angibt, dass a und b Aliase sind. Diese Relation ist stets reflexiv und symmetrisch, aber nur dann transitiv wenn es sich um eine must-Analyse handelt.

    Häufig für interprozedurale Analyse verwendet

  - Fluss-sensitiv: Können Variablen in einem Knoten aliasen? Alias-Menge kann sich ändern. Nicht trivial in Teilprobleme zerlegbar, da der komplette Kontrollfluss betrachtet werden muss. Hier ist es relativ einfach für relevante Programmpunkte p und Variablen v eine Abbildung auf abstrakte Speicherstellen SL (storage location) abzubilden, wobei SL eine Menge ist, und dafür eine Funktion bereitzustellen: $Alias(p, v) = SL$ Es wird also eine Analyse auf den Speicherstellen vorgenommen, auf die ein Pointer verweist, dies deckt auch überlappende Elemente auf (union).

    Häufig für intraprozedurale Analyse verwendet

- Betrachtung von aufgerufenen Prozeduren?
  - Intraprozedurale - Hier wird davon ausgegangen, dass ein Funktionsaufruf alles kaputt machen kann, da nur eine Prozedur betrachtet wird.
  - Interprozedurale - Beeinflussung der Aliasanalyse anhand der Aliasinformationen bereits durchgeführter Analyse anderer Funktionen (Standard-Verfahren, Andersens Algorithmus, Steensgaards Algorithmus)

Für schnelle C-Code Analyse wird oft *fluss-ignorierende may-Analyse* verwendet. Also nur für Variablen deren Adressen berechnet werden können, können Aliase haben und jede Zeiger-Variable kann auf jede dieser Variablen zeigen. Dies wird sehr schnell sehr schlecht, wenn viele Adressen bestimmt werden.

Alias-Analyse in 2 Phasen:

1. Sammler-Phase Gatherer
   Wie beeinflussen einzelne Instruktionen das bisheriges Alias-Wissen (Flussfunktion)
   Sprachabhängig
   braucht eigentlich kein KFG-Wissen

2. Verbreiter-Phase Propagator
   Auswirkungen auf andere Variabeln
   Sprachunabhängig
   braucht KFG-Wissen

- dynamisch allozierter Speicher wird separat betrachtet, möglicherweise unbegrenzt viele Aufrufe, möglicherweise unbegrenzte Laufzeit der Alias-Analyse
- dynamisch allozierter Speicher wird zusammengeführt, schlechtere Aliasanalyse
- Betrachtung der Speicherstellen auf Unterschiedlichkeit

Unterscheidung schwache vs starke Definition: Ein Pointer zeigt nur auf eine Speicherstelle oder mehrere. D.h. entweder man weiß wenn dort eine Zuweisung auf die Dereferenzierte Speicherstelle passiert, wo das hin kommt, oder nicht.

Notation von Alias-Wissen - Beispiel 07-38ff:

Programmpunkte befinden sich stets zwischen zwei Instruktionen

- $ovr_P(x)$ - overlap - Menge der Speicherstellen, die diese Variable x im Grundblock/Programmpunkt P überlappen kann (Arrays, Structs, Unions)
- $ptr_P(x)$ - Wissen über Zeiger - Menge der Speicherstellen, auf die Variable x am Grundblock P zeigen kann (eine Dereferenzierung)
- $ref_P(x)$ - beliebig viele Dereferenzierungen - Menge der Speicherstellen, die von x im Grundblock P über beliebig viele Dereferenzierungen erreichbar ist
  $ref_P^1(x) = ptr_p(x)$
  $ref_P^i(x) = ptr_p(fields(ref_P^{i-1}(x))$
  $ref_p(x) = \bigcup_{i=1}^{\infty} ref_p^i(x)$ - terminiert häufig nicht

Notation für Speicherbereiche - Beispiel 07-37ff:

- $star(o)$ - statischer Speicher für Objekt o
- $anon(ty)$ - Speicher für einen dynamischen Typen vom Typ ty, in C nicht sinnvoll, da malloc generisch
- $anon$ - dynamischer Speicher für ein Objekt unbekannten Typs

Folien ab 07-38 nochmal anschauen

Zuweisung p =& a; $ptr_P(p) = \{mem_P(a)\} = \{mem'_P(a)\}$ ptr sind nun alle Speicherstellen, an denen a liegen könnte.

7-45 merken der Mengen-kopieroperation, noch keine Durchführung, da später Fixpunkt-Iteration, bei der sich die Mengen noch ändern kann

Verbreitung des Alias-Wissens erfolgt über die Ovr(P,x) und Ptr(P,p)

8 Alias-Analyse, Teil 2

Aggressive Optimierung wird erst möglich, wenn interprozedurale Alias-Analyse vorgenommen wird.

Aliase können auftreten (Adressoperator ausgenommen), wenn:

- Zweifache Übergabe einer Variable als Argument
- globale Variable als Argument

Prozeduraufrufgraph (PCG) - Beispiel fig. 18a and listing 2:

Knoten sind die Prozeduren des Programms und eine gerichtete Kante von Prozedurknoten p nach q gibt es genau dann, wenn p q aufrufen könnte.

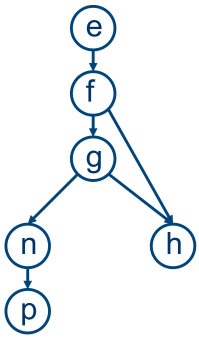
```
1  global a,b
2  procedure e(){
3     f(1,b)
4  }
5  procedure f(x,y){
6     y := h(1,2,x,x)
7     return g(a,y,) + b + y
8  }
9  procedure g(z,w){
10    local c
11    procedure n(m){
12       a:= p(z,m)
13       return m+a+c
14    }
15    w := n(c) + h(1,z,w,4)
16    return h(z,z,a,w)+n(w)
17 }
18 procedure h (i,j,k,l){...}
19 procedure p (r,s){...}
```

Listing 2: Beispielcode PCG (fig. 18a) + Variablenbindungsgraph(fig. 18b)

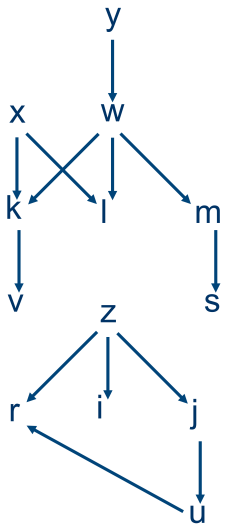
Variablenbindungsgraph - fig. 18b and listing 2:

Knoten sind die formalen Parameter des Programms und eine gerichtete Kante vom formalen Parameter x nach y gibt es genau dann, wenn (p, q) im PCG existiert und x ein formaler Parameter von p und x ein Argument für den formalen Parameter y von q ist. Also z.B. $p(type\ x)\{\ldots q(x);\ldots\}$; $q(type\ y)\{\ldots\}$

Alias-Detektierung:



(a) Beispiel PCG - listing 2



(b) Beispiel Variablenbindungsgraph - Code: listing 2, PCG: fig. 18a

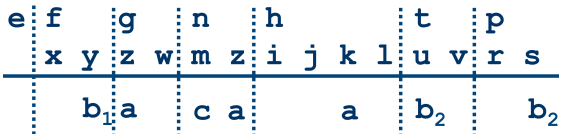
Abbildung 18: Prozeduraufrufgraph, Variablenbindungsgraph zum Code von listing 2

1. Aliase globaler Variablen
   Alias entsteht bei Übergabe einer globalen Variable als Argument. Lösung des Problems:
   a) Entdeckung von Bindungen von globalen Variablen an formale Parameter fig. 19a
   b) Anhand des Variablenbindungsgraphen detektieren an welche andere formalen Parameter die globale Variable ebenfalls weitergegeben werden können fig. 19b
   c) Aliase der globalen Variablen ablesen
2. Parameterpaare als Aliase
   Aliase entstehen durch zwei formale Parameter, wenn die gleiche Variable an beide übergeben wird oder wenn eine globale Variable an einen formalen Parameter und ein Alias an den anderen übergeben wird. Das Verfahren ist wohl super aufwändig: $\mathcal{O}(v^2 + v * e)$
   a) Betrachte alle Paare formaler Parameter - table 2
   b) Markiere diejenigen Paare, die beim Aufruf zu Alias-Paaren werden können
   c) propagiere Alias-Informationen durch den PCG
      Wird ein Alias-Paar (x, y) bei einem Prozeduraufruf an formale Parameter (X, Y) der gerufenen Prozedur gebunden, dann ist auch (X, Y) ein Alias-Paar - hier kommt es auf die paarweise Weitergabe an, weshalb die Betrachtung des Variablenbindungsgraph nicht genügt - fig. 20

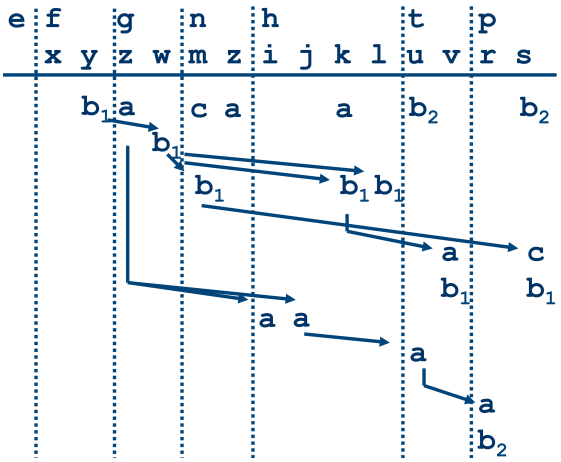
Funktion	Paare
f	$(x,x),(x,y),(y,x),(y,y)$
g	$(z,z),(z,w),(w,z),(w,w)$
n	$(m,m),(m,z),(z,m),(z,z)$
h	$(i,i),(i,j),(i,k),(i,1),(j,i),(k,i),(l,i),(j,j)$
	$(j,k),(j,l),(k,j),(l,j),(k,k),(k,l),(l,k),(l,l)$
t	$(u,u),(u,v),(v,u),(v,v)$
p	$(r,r),(r,s),(s,r),(s,s)$

Tabelle 2: Formale Parameter - listing 2

d) Aliase formaler Parameter ablesen



(a) Erkennung v. Bindung globaler Variablen - listing 2



(b) Bindungs-Propagation - listing 2 and fig. 18b

Abbildung 19: Bindungen globaler Variablen

Speichergraph und Steensgaards Algorithmus:

Weil das obere Verfahren so schlechte Laufzeit hat, betrachten wir nun ein schnelleres, aber weniger präzises Verfahren, das insbesondere für C geeignet ist und Adressoperatoren, Funktionszeiger behandelt. Im Steensgaard Algorithmus wird der Speichergraph fig. 21a approximiert, indem jeder Knoten über höchstens eine ausgehende Kante verfügt. Sollte eine Variable auf verschiedene Speicherstellen zeigen können, so fasst der Algorithmus diese zu einem einzelnen Knoten zusammen. Der Speichergraph von fig. 21a wird zu fig. 21b vereinfacht. Der Algorihtmus führt für jede Variable einen Knoten im Speichergraphen ein und Kanten für repräsentierende Zeiger. Analyse erfolgt fluss-ignorierend.

Iteratives vorgehen, bis sich nichts mehr ändert: Zeigt ein Zeiger auf eine Variable, füge eine Kante in den Speichergraphen ein. Falls ein Alias eingeführt werden könnte, vereinige entsprechende Knoten im Speichergraphen. - Beispiel 08-38ff

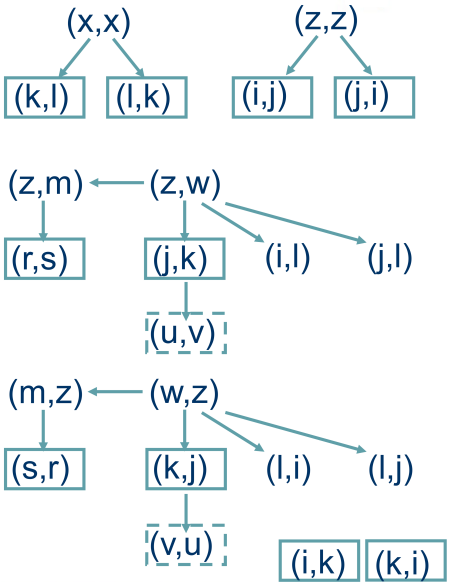


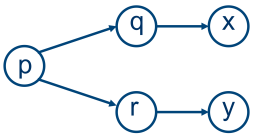
Abbildung 20: Paarbindungsgraph - listing 2
Markierung eintragen, Markierung propagieren

Falls ein Knoten noch nicht als Zeiger erkannt wurde, aber einer Verschmelzung hervorrufen würde, wäre er einer, so wird sich dies vorgemerkt. Regel für $a := b$ 08-41f, $a := *b$ 08-43, $*a := b$ 08-44, $a := b \otimes c$ 08-45f

Prinzipiell scheint der Algorithmus sich das Typsystem zu nutze zu machen und entsprechend Funktionszeiger auf ihre Signatur hin zu betrachten. Aufwand steigt pro Variable und Parameter um einen Knoten und höchstens eine Verschmelzung. Also ist der Speicherplatzaufwand linear. Die Laufzeigt pro Anweisungstyp ist nahezu konstant, das verschmelzen macht das jedoch schlechter. Dessen Aufwand wächst jedoch sehr langsam (inverse Ackermann-Funktion)

```
1 q = &x
2 r = &y
3 if(...)
4     p = &q
5 else
6     p = &r
```

Listing (3) Code für Speichergraph-Beispiel (fig. 21a)



(a) Speichergraph - Beispiel - listing 3
Jeder Knoten repräsentiert eine oder mehrere Speicherstellen und die Kanten sind “zeigt auf”-Beziehungen

(b) Speichergraph - Beispiel nach Steensgaard - listing 3 and fig. 21a

Abbildung 21: Speichergraph mit Umwandlung nach Steensgaard

9 Herausziehen schleifeninvarianten Codes, Induktionsvariableneliminierung

Schleifeninvariante Instruktion:

Instruktion heißt *schleifeninvariant* gdw. für alle Operanden v mindestens einer der folgenden Punkte gilt:

- v ist konstant oder
- v ist nur außerhalb der Schleife berechnet oder
- v wird innerhalb der Schleife in einer schleifeninvarianten Instruktion berechnet

Algorithmus:

1. Vorbereitung
   - Alle natürlichen Schleifen finden
   - erreichende Definitionen berechnen
   - Konstantenfaltung vornehmen und konstante Werte bestimmen
2. alle konstanten Ausdrücke/Variablen und alle Variablen, deren erreichende Definitionen alle außerhalb der Schleife liegen als schleifeninvariant markieren
3. Solange sich noch was ändert: markiere Anweisungen/Variablen als schleifeninvariant, wenn:
   - sie konstant sind
   - alle erreichenden Definitionen außerhalb der Schleife liegen
   - oder wenn genau eine erreichende Definition existiert und schleifeninvariant ist - Beispiel 09-06

Herausziehen von Anweisungen:

- *invariante Ausdrücke* können im Grunde jedesmal vor die Schleife rausgezogen werden - Speicherung in temporären Variablen möglich
- *invariante Zuweisungen* können höchstens bei strikter Dominanz aller folgenden Knoten rausgezogen werden und in der Schleife nicht wo anders auf einen anderen Wert gesetzt wird (also einzige Zuweisung ist)
  - Vorbereitung
    - Dominanz, erreichende Definitionen, Konstantenfaltung, schleifeninvariant-Eigenschaft berechnen
    - Schleifenausgänge bestimmen, also die Knoten im KFG, die Nachfolger außerhalb der Schleife haben.

Bedingungen für das Rausziehen von Zuweisungen - Beispiele 09-11f:

- invariant
- strikte Dominanz aller Benutzungen innerhalb der Schleife (Zuweisung in einem Grundblock)
- Dominanz aller Ausgänge
- Zuweisung ist die einzige zu dieser Variable innerhalb der Schleife

Betrachtung von minimalen Grundblöcken vereinfacht die Analyse. Beim Herausziehen sollte die Reihenfolge der Anweisungen beibehalten bleiben

while-Schleifen müssen also in if-not-then-do-while umgebaut werden, damit das wirklich nur vor das gezogen wird, wenn die Eintrittsbedingung erfüllt ist. s. auch 09-14f (while-Schleifenausgang wird von keinem anderen Block dominiert) .

Loop-Unswitching:

Loop-Unswitching. Rausziehen von invarianten Überprüfungen. Also man macht aus einer Schleife zwei Schleifen, wobei die Bedingung vor die Schleife gezogen wird

Vorteile:

- bedingter Sprung wird nur einmal ausgewertet
- die kleinen Schleifen passen womöglich vollständig in den Cache
- bessere Sprungvorhersageeigenschaften

Nachteile:

- Code-Vergrößerung
- Erzeugung komplexerer Struktur, wodurch womöglich andere Transformationen verhindert werden.

Parallelität nicht mehr so schön möglich, wenn dadrum noch eine Schleife liegt

In SSA-Form ist die Entscheidung, ob Dinge invariant sind wesentlich einfacher und auch relativ viel kann als invariant rausgezogen werden

Arrayzugriffe in Schleifen immernoch teuer - will man optimieren - macht der Compiler inzwischen

Induktionsvariable:

Variable x ist *Induktionsvariable* einer Schleife, wenn x bei jedem Durchlauf um einen Konstanten Wert ink-/dekrementiert wird

einfache Induktionsvariable:

x ist eine *einfache Induktionsvariable*, wenn sie innerhalb der Schleife nur mit Zuweisungen der Form $x := x \pm c$ verändert wird, wobei c schleifeninvariant.
Durch Inspektion des Schleifenrumpfs leicht zu entdecken

abhängige Induktionsvariable:

y ist eine *abhängige Induktionsvariable*, wenn sie innerhalb der Schleife einmal oder mehrfach, aber dafür genau gleich, als lineare Funktion einer anderen Induktionsvariable berechnet wird: $y := a * x \pm b$, wobei a, b schleifeninvariant und x Induktionsvariable

Familie einer einfachen Induktionsvariablen:

Wird definiert durch eine einfache Induktionsvariable x, wobei sämtliche y zu dieser Familie gehören, wenn es schleifeninvariante Ausdrücke c_{1y}, c_{2y} gibt, sodass $y == c_{1y} * x + c_{2y}$ immer gilt, wenn y in der Schleife definiert wird. Aus c_{1y}, c_{2y} kann y berechnet werden Also erfolgt die Berechnung in Abhängigkeit von einem Familienmitglied - wichtig, dass zwischendrin niemand relevante Werte verändert haben darf. Also wenn $y := c_{1y} * z \pm c_{2y}, z \in$Familie$(x)$, dann darf zwischen der Zuweisung zu z und derer zu y x nicht verändert werden. Auch darf keine Definition von z die Zuweisung zu y von außerhalb der Schleife erreichen. - Beispiele 09-29f

Reduktion Induktionsvariablen - Schleifenmanipulation - Beispiel 09-36ff:

Für jede einfache Induktionsvariable x und für jede von x abhängige Induktionsvariable y mit einem Tripel (x, a, b), also $y := x * a + b$:

- erzeuge neue Variable y'
- Ersetze die Zuweisung zu y durch $y := y'$
- Nach jeder Zuweisung $x := x \pm c$ füge neue Zuweisung $y' := y' \pm a * c$ ein
- Neue Zuweisung $y' := x * a + b$ vor dem Schleifenkopf einfügen

Terminierungsbedingung der Schleife kann auch durch andere Invarianten ersetzt werden

9-44 Bedingung warum der Überlauf nicht so wild ist: Der Überlauf ist auch im Schleifenrumpf bereits passiert - die Abbruchbedingung steht ja nun am Ende der Schleife

Die Bereichstests, die in Java bei Arrayzugriffen vorgenommen werden, können ebenfalls transformiert werden: 09-57ff

10 Abhängigkeitsdistanzen, Richtungsvektoren, parallele Ausführbarkeit von Schleifen

Datenabhängigkeiten δ:

- *Fluss-Abhängigkeit* δ^f: Speicherstelle A wird geschrieben und später gelesen
- *Ausgabe-Abhängigkeit* δ^o: Speicherstelle A wird geschrieben und später erneut geschrieben
- *Anti-Abhängigkeit* δ^a: Speicherstelle A wird gelesen und später geschrieben

Lokalität:

- Zeitlich - temporal locality
  Nahe am Prozessor und häufig verwendet werden, um Cache-Miss-Kosten zu amortisieren
- Räumlich - spatial locality
  Nebeneinander im Speicher liegende Daten nacheinander verwenden, Cache-Miss-Kosten durch Verwendung der Nachbarn amortisieren

```
1 for i:= 1 to 100 do
2    for j := 1 to i do
3       A[i,j] := A[i,j] / (A[i,i]*A[i,i+1])
```

Listing 4: Beispiel Array - Datenabhängigkeit - fig. 22 and listing 5

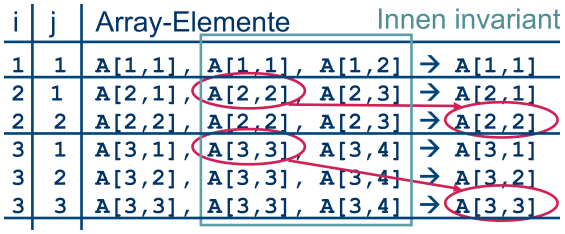
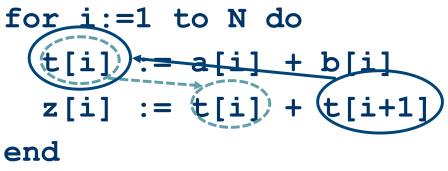


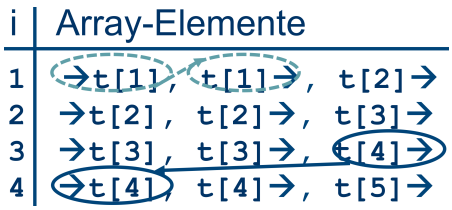
Abbildung 22: Datenabhängigkeit - Beispiel Array - listings 4 and 5
Erst mit letzter Iteration der inneren Schleife wird $A[i,i]$ verändert

```
1 for i:= 1 to 100 do
2    t := 1.0 / (A[i,i] * A[i,i+1])
3    for j := 1 to i do
4       A[i,j] := A[i,j] * t
```

Listing 5: Datenabhängigkeit - Beispiel Array - optimiert - fig. 22 and listing 4
Nur möglich, weil $A[i,i]$ als letztes in der inneren Schleife verändert wird



(a) Datenabhängigkeit - Beispiel 2 Array - Code zu den Abhängigkeiten in fig. 23b



(b) Datenabhängigkeit - Beispiel 2 Array - Abhängigkeiten zu dem Code in fig. 23a

Abbildung 23: Datenabhängigkeit - Beispiel 2 Array
Abhängig in selber Iteration, Abhängig in nächster Iteration - Code wird optimiert zu listing 6

```
1 for i:= 1 to N do
2    reg0 := a[i] + b[i]
3    z[i] := reg0 + t[i+1]
4    t[i] := reg0
```

Listing 6: Datenabhängigkeit - Beispiel 2 Array optimiert - fig. 23
1 array store und 3 Registerzugriffe, statt 1 array store und 1 array load

Verständnis der Schleifen- und Arrayausdrücke ermöglicht ...:

- weitergehende Anwendungen der bisher vorgestellten Techniken
- bessere Registerverwendung - skalare, statt Array-Ausdrücke, zeitliche Lokalität - mehrfach gleicher Array-Ausdruck
- Ausnutzung räumlicher Lokalität, effizientere Cache-Nutzung
- Parallelisierung bzw. Vektorisierung von Schleifen
- Lokalitäts- und Kommunikationsoptimierung für Parallelrechner

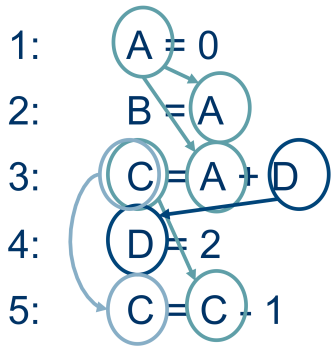


Abbildung 24: Datenabhängigkeit Beispiel 3 - $S_1\delta^f S_2$, $S_1\delta^f S_3$, $S_3\delta^f S_5$, $S_3\delta^a S_4$, $S_3\delta^o S_5$

Schleifenunabhängige/Schleifengetragene Datenabhängigkeiten:

- *Schleifenunabhängige Datenabhängigkeiten - loop independent dependence* sind Abhängigkeiten innerhalb einer einzelnen Iterationen, diese verbleiben auch beim entfernen der Schleife.
- *Schleifengetragene Datenabhängigkeiten - loop carried dependence* sind Abhängigkeiten, die durch die Existenz der Schleife verursacht werden.

Iterations-Foo:

Jede Iteration einer Schleife kann durch einen d-dimensionalen Iterationsvektor $i = (i1, i2, \ldots, id)$ eindeutig beschrieben werden. Hierbei bildet die Menge aller möglichen Iterationsvektoren den *Iterationsraum*. Es gilt $i \angle j \Leftrightarrow$ die Iteration i kommt zur Laufzeit vor der Iteration j. Vektoren i, j sind lexikographisch aufsteigend geordnet.

Abhängigkeitsdistanz-/Vektor:

Besteht eine Datenabhängigkeit zwischen einer Anweisung $S1$ in Iteration i_{source} und einer Anweisung $S2$ in Iteration j_{target}, wobei $i_{source} \angle j_{target}$, so heißt $i_{source} + d = j_{target}$ die *Abhängigkeitsdistanz*. Weiterhin bildet d den zugehörigen *Abhängigkeitsdistanzvektor*
Beispiele: für alle $(3,2)\delta^a(3,4), (3,4)\delta^s(3,6), (4,2)\delta^a(4,4), (4,4)\delta^a(4,6)$ ist die Distanz $+(0,2)$

Bei inkrementierenden Schleifen sehen gültige Abhängigkeitsdistanzen wie folgt aus: $\underbrace{d_1, d_2, \ldots,}_{0}\ \underbrace{d_p}_{>0}, \underbrace{d_{p+1}, \ldots, d_n}_{\text{egal}}$, also der erste Wert ungleich 0 muss größer 0 sein.
In diesem Fall *trägt* die Schleife die *Abhängigkeit*, weil die Abhängigkeit zu nachfolgenden Iterationen besteht. Der entsprechende Richtungsvektor: $(=, =, \ldots, <, ?, \ldots, ?)$

Abhängigkeitsrichtung, Richtungsvektor:

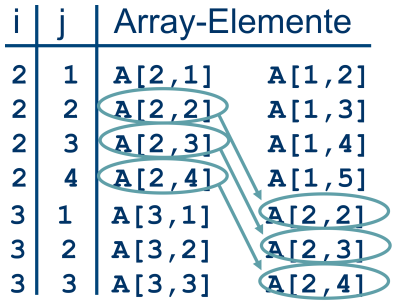
Die *Abhängigkeitsrichtung* ist eine Vergröberung der Abhängigkeitsdistanz, wobei für die Berechnung jede Position des Abhängigkeitsdistanzvektors ersetzt wird:
$0 = d_i \Rightarrow$ " $=$", $0 < d_i \Rightarrow$ " $<$", $0 > d_i \Rightarrow$ " $>$", wobei $i_{source} < i_{target}$,
Achtung: 0 steht links und d auf der rechten Seite, d.h. also aus $(1, -1)$ wird $(<, >)$ und aus $(-1, 0)$ wird $(>, =)$

Sämtliche Abhängigkeitsrichtungen einer Schleife können mit weiterer Vergröberung zum *Richtungsvektor* der Schleife zusammengefasst werden

$=,= \Rightarrow =$ $=,< \Rightarrow \leq$ $>,< \Rightarrow *$
$<,< \Rightarrow <$ $=,> \Rightarrow \geq$
$>,> \Rightarrow >$

```
1 for i := 2 to n do
2     for j := i to n-1 do
3         A[i,j] = A[i,j] + A[i-1, j+1]
```

Listing (7) Abhängigkeitsdistanz - Beispiel - fig. 25a



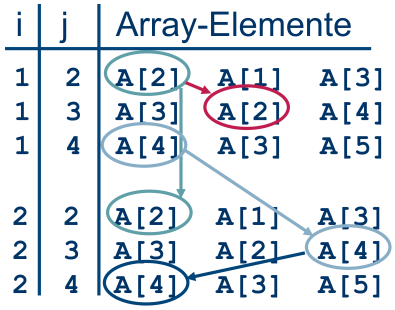
(a) Abhängigkeitsdistanz - Beispiel Distanz - listing 7
$\delta^f : (2,2) + (1,-1) = (3,1)$ Also Abhängigkeitsdistanz: $(1,-1)$ mit Richtungsvektor $(<,>)$

Parallele Schleifenausführung:

Es kann passieren, dass die Abhängigkeisdistanz nicht für alle Iterationen konstant ist, dann ist keine Angabe möglich, dennoch ist womöglich ein Richtungsvektor findbar. Dieser Richtungsvektor kann angeben, ob eine Schleife parallel ausführbar ist, oder nicht. 10-44 - Dies ist möglich, wenn der Richtungsvektor für alle Indices ein $=$ enthält. Dies ist weiterhin möglich, wenn **jede** schleifengetragene Abhängigkeit von einer die Schleife p umgebenden Schleife getragen wird, dann können alle Iterationen der Schleife p parallel ausgeführt werden, also $\exists q < p, d_q > 0$ also für alle Positionen $q < p$ steht $<$ im Richtungsvektor, also $(d_1, d_2, \ldots, d_q, \ldots, d_p, \ldots, d_n)$

```
1 for i := 1 to n do
2    for j := 2 to n-1 do
3       A[j] = (A[j] +A[j-1]+A[j+1])/3
```

Listing (8) Abhängigkeitsdistanz - Beispiel 2 - fig. 26a



(a) Abhängigkeitsdistanz - Beispiel 2 Distanz - listing 8
$\delta^f\delta^o : (i,j)+(1,0) = (i+1,j) \Rightarrow (<,=)$
$\delta^f : (i,j)+(0,1) = (i,j+1) \Rightarrow (=,<)$
$\delta^f : (i,j)+(1,-1) = (i+1,j-1) \Rightarrow (<,>)$
$\delta^a : (i,j)+(0,1) = (i,j+1) \Rightarrow (=,<)$
$\Rightarrow (\leq^*)$, i und j nicht parallel ausführbar

Beispiele: für $(0,1,1),(=,<,<)$ und $(1,0,1),(<,=,<)$ ist jeweils die innere Schleife parallel ausführbar.
listing 4: $(=,<)$ i parallel, j nicht
fig. 23 Distanzvektor (1), Richtungsvektor $(<)$, i nicht parallel ausführbar
fig. 25a i nicht parallel, j parallele ausführbar Weitere 10-49ff

11 Indexanalyse, Schleifentransformationen

zwei Zugriffe auf ein Array - pessimistische Annahme, es wird auf das selbe Element zugegriffen - und später versucht zu zeigen, dass es doch unterscheidliche sind

existenz einer Datenabhängigkeit:

Es liegt genau dann eine Datenabhängigkeit $S1\delta S2$ vor, wenn:

- mindestens ein Schreibzugriff vorgenommen wird
- $\exists I, J : I = (i_1, \ldots, i_d) \angle J = (j_1, \ldots, j_d)$ Ungleichungen existieren
- beide innerhalb der Schleifengrenzen, wobei die Nebenbedingung ist: die Existenz einer ganzzahlige Lösung innerhalb der Schleifengrenzen
- $\forall p : f_p(I) = g_p(J)$, also ein Gleichungssystem in dem die Variablen die Schleifenlaufvariablen sind und die Koeffizienten die Konstanten in den linearen Index-Ausdrücken.

Das lässt sich nur mit dümmlichen Tests überprüfen, da np-hart Verschiedende Tests zum bestimmen ob eine Abhängigkeit vorliegt oder nicht:

- ggt - Abhängigkeit ist nur möglich, wenn der ggT die Koeffizienten der Konstante teilt.
  Beispiel: $A[2i+2] := A[2i-2]$; $2i_{def} + 2 = 2i_{use} - 2 \Leftrightarrow 2i_{def} - 2i_{use} = -4$, da $ggt(2,2) = 2, 2|4$ ist eine Abhängigkeit vielleicht möglich: $i_{def} - i_{use} = -2$
  Eine weitere Analyse zeigt, $i_{def} + 2 = i_{use}$, dass δ^f mit einer Abhängigkeitsdistanz von 2 tatsächlich möglich ist, wobei Schleifengrenzen und Schrittweiten zu beachten sind
- Ungleichungstest - Es werden obere und untere Grenzwerte der Array-Zugriffe analysiert, Ist in der Schnittmenge des Intervalls die 0 nicht im Ergebnis enthalten, so liegt keine Abhängigkeit vor: $\neq 0 \Rightarrow$ unabhängig
- Fourier-Motzkin - s.u.

Dabei ist der allgemeine Testansatz in fig. 27 dargestellt.

Fourier-Motzkin-Test:

Das Ungleichungssystem kann in verschiedene Formen gebracht werden, entweder man sucht eine untere Schranke: $\beta \leq bz, \quad z, \beta > 0$ oder aber eine obere: $az \leq \alpha, \quad z, \alpha > 0$

Anhand der Fourier-Motzkin-Elimination lässt sich implizieren: $a\beta \leq azb \leq b\alpha \Rightarrow a\beta \leq b\alpha$, wodurch z für die Gleichung irrelevant wird und das Problem kleiner wird, dies lässt sich rekursiv fortführen. Am Ende gibt es zwei Fälle:
Entweder es gibt *keine Lösung*, wodurch die *Unabhängigkeit* gezeigt wurde, oder aber *es gibt eine Lösung*, wobei weiterhin die *Existenz einer ganzzahligen Lösung* für *azb nachzuweisen* gilt.

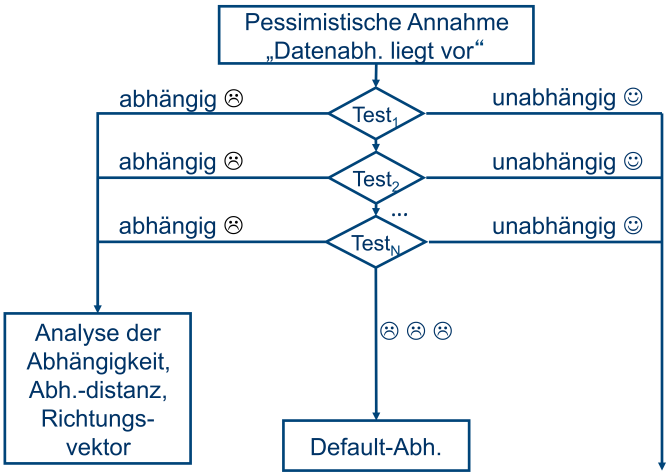


Abbildung 27: Test auf Existenz von Abhängigkeiten

Beispiel: Gegeben sei folgendes Ungleichungssystem

$$\begin{aligned} 2x_1 - 11x_2 &\leq 3 \\ -3x_1 + 2x_2 &\leq -5 \\ x_1 + 3x_2 &\leq 4 \\ -2x_1 &\leq 3 \end{aligned}$$

Als erstes erfolgt eine Normalisierung für einen der Parameter, hier x_2:

$$\begin{aligned} \frac{2}{11}x_1 - \frac{3}{11} \leq x_2 \\ x_2 \leq \frac{3}{2}x_1 - \frac{5}{2} \\ x_2 \leq -\frac{1}{3}x_1 + \frac{4}{3} \end{aligned}$$

Anschließend wird der normalisierte Parameter (hier x_2) eliminiert:

$$\begin{aligned} \frac{2}{11}x_1 - \frac{3}{11} &\leq \frac{3}{2}x_1 - \frac{5}{2} \\ \frac{2}{11}x_1 - \frac{3}{11} &\leq -\frac{1}{3}x_1 + \frac{4}{3} \\ -2x_1 &\leq 3 \end{aligned}$$

Abschließend wird versucht das reduzierte Problem zu lösen:

$$1.69 \approx \frac{49}{29} \leq x_1 \leq \frac{53}{17} \approx 3.1$$

Entsprechend kann x_1 nur entweder 2 oder 3 sein, da wir ganzzahlige Lösungen suchen. Setzen wir also diese Werte in vorherige Gleichungen ein:

$x_1 := 2$

$$\frac{1}{11} \leq x_2 \leq \min\left\{\frac{1}{2}, \frac{2}{3}\right\} \lightning$$

$x_1 := 3$

$$\frac{3}{11} \leq x_2 \leq \min\left\{2, \frac{1}{3}\right\} \lightning$$

Für keine der beiden Werte konnte eine Lösung gefunden werden, entsprechend existiert keine und die Variablen sind unabhängig.

Entfernung schleifengetragener Datenabhängigkeiten:

Skalarvevielfachung oder Skalarprivatisierung (das selbe, nur eine skalare variable pro Prozessor), Umwandlung schleifengetragener Datenabhängigkeiten in schleifenunabhängige Datenabhängigkeiten durch Schleifenneuausrichtung oder Schleifenrumpfaufteilung.

Schleifen-Neuausrichtung: schleifengetragene Abhängigkeit mit bekannter Abhängigkeitsdistanz, wird in eine schleifenunabhängige datenabhängigkeit umgewandelt, wodurch sich die Schleife parallelisieren lässt

```
1 //Abhängigkeitsdistanz 1
2 for i := 2 to n
3    A[i] := B[i] + 2
4    C[i] := A[i-1]*2
5
6 //wird zu, also nun Abhängigkeitsdistanz 0, wodurch die
     Schleife parallelisierbar wird
7 for i := 1 to n
8    if i > 1
9       A[i] := B[i] + 2
10   if i < n
11      C[i+1] := A[i]*2
```

Listing 9: *Loop Alignment* - Schleifenneuausrichtung, Verschieben der Anweisungs-Ausführung um eine konstante Abhängigkeitsdistanz an Iterationen. Dadurch lässt sich die Schleife parallelisieren. Gibt es in einer Schleife mehrere schleifengetragene Datenabhängigkeiten, lassen sich diese selten alle umwandeln. Die entsprechende Transformation/Restrukturierung heißt *legal*, gdw. die relative Reihenfolge für jede Datenabhängigkeit erhalten bleibt.

```
for i := 2 to n-1
    A[i] := A[i] + A[i-1]*A[i+1]

//f-faches unrolling, indem Fall f=2
for i := 2 to n-2 by 2
    A[i] := A[i] + A[i-1]*A[i+1]
    A[i+1] := A[i+1] + A[i]*A[i+2]

if mod(n-2,2)
    A[n-1] := A[n-1] + A[n-2]*A[n]
```

Listing 10: *Loop Unrolling*, Schrittweite anpassen, Rumpf replizieren, Randbedingung nach der Schleife, falls kein gleichmäßiges ausrollen möglich war.

Vor-Nachteile des loop unrollings - listing 10:

- Vorteile
  - Ist immer legal
  - Schleifenoverhead reduziert
  - lokale Optimierungen eher möglich
  - Instruktionscache könnte womöglich besser ausgenutzt werden
  - Höhere Lokalität/Registerverwendung
  - Parallelität
- Nachteile
  - Code-Wachstum
  - Behinderung anderer Optimierungen, da diese häufig kleinere Schleifenrümpfe bevorzugen
  - Randbedingung ist zu beachten

```
for i := 0 to tc-1
    ...

//umwandeln zu
for i := 0 to s-1
    ...
for i := s to tc-1
    ...
```

Listing 11: *Index Set Splitting* - Schleifenbereichsteilen

Vor-Nachteile des Index Set Splitting - listing 11:

- Vorteile
  - Ist immer legal
  - Eine der Schleifen kann anhand anderer Optimierungen optimiert werden, da nun die gewünschte Breite vorliegt
  - Beispielsweise sinnvoll, wenn der Schleifenrumpf eine if-condition der folgenden Form hat, kann diese gespart werden: if $i > s$ then.
  - Einige Array-Bereichstests können in einer der Schleifen womöglich entfallen
- Nachteile
  - Code-Wachstum
  - Behinderung anderer Optimierungen möglich

```
1 for i := 0 to tc-1
2     ...
3
4 //abschälen der 1. Iteration
5 if tc > 0
6     i := 0
7     ...
8     for i := 1 to tc-1
9         ...
```

Listing 12: *Loop Peeling* - Schleifenschälen - ist ein Sonderfall des Schleifenbereichsteilens - Beispiele listings 13 and 14

Vor-Nachteile des Index Loop Peeling - listing 12:

- Vorteile
  - Ist immer legal
  - Wie bei Schleifenbereichsteilen
  - Invarianten werden in der ersten Iteration ausgewertet und später verwendet
  - Datenabhäniggkeiten zu den ersten und letzten Iterationen lassen sich wunderbar herausziehen, wodurch Schleifen parallelisierbar werden
  - Gute Kombinierbarkeit mit Schleifen-Neuausrichtung
- Nachteile
  - Wie bei Schleifenbereichsteilen

```
for i := 2 to n
   B[i] := B[i] + B[2]
for i := 3 to n
   A[i] := A[i] + c

//erste Iteration abschälen und Schleifen verschmelzen
if 2 <= n
   B[2] := B[2] + B[2]
for i := 3 to n
   B[i] := B[i] + B[2]
   A[i] := A[i] + c
```

Listing 13: Schleifenschälen - Beispiel 1 - mit Schleifenverschmelzung - listing 12

```
for i := 1 to n
   if i > 1
      A[i] := B[i] + 2
   if i < n
      C[i+1] := A[i]*2

//Abschälen der ersten und n.ten Iteration
if 1 < n
   C[2] := A[1]*2
for i := 2 to n-1
   A[i] := B[i] + 2
   C[i+1] := A[i]*2
if n > 1
   A[n] := B[n] + 2
```

Listing 14: Schleifenschälen - Beispiel 1 - mit Schleifen-Neuausrichtung - listing 12

```
for i := 1 to n
   for j := 1 to m
      A[i,j] := A[i,j] + c

//Produktschleifenbildung
for t := 1 to n*m
   i :=((t-1) / m) + 1
   j := mod(t-1, m) + 1
   A[i,j] := A[i,j] + c
```

Listing 15: *Loop Coalescing* - Produktschleifenbildung. Wird schwieriger, wenn die Bereichsgrenzen voneinander abhängen, also die Iterationsräume nicht rechteckig sind.

Vor-Nachteile des Index Loop Coalescing - listing 15:

- Vorteile
  - Ist immer legal, außer es gibt einen Überlauf
  - Produktgröße ist für Vektor- oder Parallelrechner häufig besser geeignet
  - Wird nachträglich eine Induktionsvariablenoptimierung durchgeführt, so kann der Schleifen-Overhead reduziert werden
  - Weitere Vereinfachungen sind möglich, wenn die Speicheranordnung der zugegriffenen Arrays passend ist
- Nachteile
  - Einschränkung anderer Optimierungen, z.B. Schleifenvertauschung

```
1  for i := 1 to n
2     A[i] := A[i]*c
3
4  //Streifenschneiden in Größe 64
5  Tn := (n/64) * 64
6  for t := 1 to Tn by 64
7     for i := t to t+63
8        A[I] := A[i]*c
9  //Randbehandlung
10 for i := Tn+1 to n
11    A[i] := A[i]*c
```

Listing 16: *Strip Mining* - Streifenschneiden - Die Parallelisierbarkeit wird noch erhöht, wenn man erst Loap Coalescing vornimmt und anschließend Strip Mining

Vor-Nachteile des Strip Minings - listing 16:

- Vorteile
  - Ist immer legal
  - Massive Ausnutzung von Parallelismus beispielsweise mit SIMD
  - gut Kombinierbar mit Loap Coalescing
- Nachteile
  - Höherer Schleifen-Overhead

```
1 for i := ...
2    body1
3 for i := ...
4    body2
```

```
5
6 //Schleifenverschmelzzen
7 for i := ...
8    body1
9    body2
```

Listing 17: *Loop Jamming* - Loop Fusion - Schleifenverschmelzung. Die Abbruchbedingungen beider Schleifen sollten gleich sein, sonst muss erst noch mit Loop Peeling und Index Set Splitting nachgeholfen werden. Weiterhin ist diese Operation nicht bei allen Datenabhängigkeiten legal. Es darf in der ersten Schleife keine Anweisung $S1$ und in der zweiten Schleife kein $S2$ geben, so dass durch die Verschmelzung eine Abhängigkeit $S2\delta^{>}S1$ entstehen würde (wobei $>$ eine spätere Iteration meint).

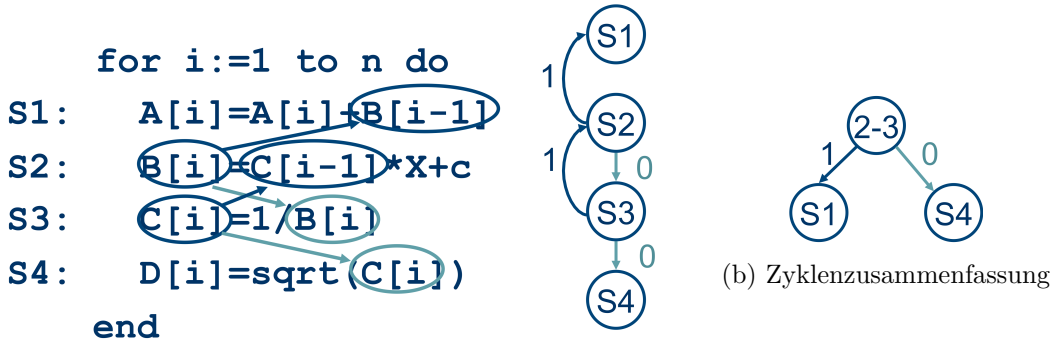
Vor-Nachteile des Loop Jammings - listing 17:

- Vorteile
  - Reduktion des Schleifen-Overheads
  - effizientere lokale Optimierungen werden aufgrund größerer Grundblöcke möglich
  - Schleifen mit Zugriff auf die selben Datenelemente weisen nach der Verschmelzung häufig eine zeitliche Lokalität in der Speicherhierarchie vor
- Nachteile
  - Womöglich könnten größere Schleifemrümpfe zu groß für den Instruktions-Cache werden

Um die Legalität zu gewährleisten muss vorab ein Abhängigkeitsgraph der zu teilenden Schleife erstellt werden. Alle Anweisungen, die in starken Zusamenhangskomponenten liegen, dürfen nicht auf unterschiedliche Schleifen verteilt werden. Falls für zwei Anweisungen $S1, S2$ die Abhängigkeit $S1 \rightarrow^{*} S2$ gilt, so muss die Schleife, die $S1$ aufnimmt, nach dem Schleifenrumfteilen vor der Schleife stehen, die $S2$ aufnimmt. Induktionsvariablen sind zu vernachlässigen. - Beispiel für diesen Graphen: fig. 28

```
1 for i := 1 to n
2    B[i] := C[i-1]*X +c
3    C[i] := 1/B[i]
4 for i := 1 to n
5    A[i] := A[i] + B[i-1]
6 for i := 1 to n
7    D[i] := sqrt(C[i])
```

Listing 18: Transformierter Code nach *Schleifenrumpfteilen* - Loop Splitting, Loop Distribution, Loop Fission - Gegenteil von Verschmelzen - fig. 28



(a) Abhängigkeitsgraph

(b) Zyklenzusammenfassung

Abbildung 28: Legalität Schleifenrumpfteilen - Beispiel - wird transformiert zu listing 18

Vor-Nachteile des Schleifenrumpfteilens - fig. 28:

- Vorteile
  - Es kann perfekte Schleifenschachtelung (siehe 11-5) entstehen, wenn einige Anweisungen der zu teilenden Schleife ebenfalls Schleifen sind
  - womöglich werden Registerengpässe anhand kleinerer Grundblöcke überbrückt
  - die kleineren Rümpfe passen u.U. eher in den Instruktions-Cache
  - Der Cache kann besser genutzt werden, wenn in den kleineren Schleifen weniger auf verschiedene Arrays zugegriffen wird
  - Abhängigkeiten liegen nun womöglich zwischen Schleifen, wodurch parallele Ausführung nun nicht mehr behindert wird
- Nachteile
  - wie bei Schleifenverschmelzung

12 Schleifenrestrukturierungen

```
for i := a
    for j := b
        ...
//Durch Schleifenvertauschung umwandeln in
for j := b
    for i := min(a, j)
        ...
```

Listing 19: *Loop Intechange* - Schleifenvertauschung - hierbei werden ebenfalls Richungsvektoren und Abhängigkeitsdistanzen angepasst: $(d_1, d_2, \ldots, d_i, d_j, ..., d_n)$ wird zu $(d_1, d_2, \ldots, d_j, d_i, ..., d_n)$
Entsprechende Legailitätsregel lautet: Alle Abhängigkeitsdistanzen sind per Definition lexikographisch positiv und die Abhängigkeitsdistanzen müssen das nach der Vertauschung auch bleiben. Beispiele:
Datenabhängigkeit vor die äußere Schleife tragen, Distanz innen ist 0, Richtungsvektor $(<, =)$, Vertauschen legal
Datenabhängigkeit von innerer Schleife getragen, Distanz außen 0, Richtungsvektor $(=, <)$, Vertauschen legal
Datenabhängigkeit hat bei beiden Schleifen positive Distanz mit Richtungsvektor $(<, <)$, Vertauschen legal
Vertauschen illegal wenn: Richtungsvektor $(<, >)$, da durch Vertauschung $(>, <)$ entstünde.
Beispiel zur Legalität - listing 25

Vor-Nachteile der Schleifenvertauschung - listing 19:

- Vorteile
  - Schleifen mit Abhängigkeiten nach außen tragbar, wodurch innen parallel ausführbare Schleife verbleibt
  - Vektorisierungsförderung und Prozentual weniger Schleifen-Overhead durch größere Schleifenbreite
  - räumliche Lokalität durch Schleifen entlang der Array-Dimension nach innen gerichtete, bzw. bei Invarianz anderer Schleifen nach außen und zeitliche lokalität
  - Gut mit anderen Techniken kombinierbar
- Nachteile
  - -

```
//Vertauschen verboten, da Abh.Distanz (1,-1)
for i := 2 to n
```

```
3     for j := 1 to n
4        A[i, j] := A[i-1, j+1]+1
5
6  //Richtungsumkehr
7  //Vertauschen nun erlaubt, da Abh. Distanz (1,1) durch den
       Vorzeichenwechsel
8  for i := 2 to n
9     for j := n to 1 by -1
10       A[i, j] := A[i-1, j+1]+1
```

Listing 20: *Loop Reversal* - Richtungsumkehr
Legal, wenn alle Abhängigkeiten von der umgebenden Schleife getragen werden, also $(\underbrace{d_1, d_2, \ldots}_{d_i>0}, d_p, \ldots, d_n)$, Weil d_p nach der Richtungsumkehr sonst < 0 wäre, da > 0. Kurz legal, wenn Abhängigkeitsdistanzvektoren legal bleiben. Durch die Richtungswechsel der Schleife ändert sich das Vorzeichen. Die Schleifenrichtungen sind explizit Teil der Faktoren! - Beispiel Richtungsumkehr zur Erzeugung der Legalität von Vertauschung: listing 26

Vor-Nachteile der Richtungsumkehr - listing 20:

- Vorteile
  - Einige Architekturen haben nur einen Springe-wenn-Null-Befehl und kein CMP-And-jmp
  - Durch das Ändern des Vorzeichens der Abhängigkeitsdistanz werden evtl. weitere Optimierungen möglich
- Nachteile
  - -

```
1  for i := 1 to n
2     for j := lo to up
3        body(i, j)
4
5  //Schleifenneigen, Neigungsfaktor f
6  for i := 1 to n
7     for j := lo + f*i to up+f*i
8        body(i, j-f*i)
```

Listing 21: *Loop Skewing* - Schleifenneigen
Der Index der äußeren Schleife wird mit dem Neigungsfaktor f multipliziert und auf die Schleifengrenze der inneren Schleife addiert und bei der Verwendung des inneren Zählerindexes von diesem subtrahiert. - weitere Beispiel mit Vertauschungslegalität: 12-24, 12-26

```
1 //Abhängigkeitsdistanzen: (0,1) und (1,-1)
2 for i := 2 to n
3    for j := 0 to n-i
4       A[i,j+i] := 0.5*(A[i,j+i-1]+A[i-1, j+i])
5
6 //Schleifenneigen mit Faktor 1
7 //Abhängigkeitsdistanzen: (0,1 = 1+1*0), (1,0=-1+1*1)
8 for i := 2 to n
9    for j := i to n
10      A[i,j] := 0.5*(A[i,j-1]+A[i-1, j])
```

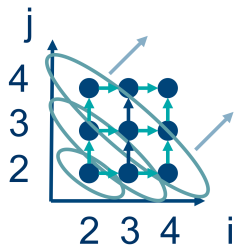
Listing 22: *Loop Skewing* - Beispiel Faktor 1

Vor-Nachteile der Schleifenneigung- listing 21:

- Vorteile
  - Immer legal (außer bei Überlauf)
  - Distanzabhängigkeit (d_1, d_2) wird zu $(d_1, d_2 + f * d_1)$
  - Dadurch kann die Legalität anderer Transformationen erzeugt werden - s. 12-24,12-26
- Nachteile
  - -

```
1 //Abhängigkeitsdistanzen: (1,0), (0,1)
2 for i := 2 to n-1
3    for j := 2 to m-1
4       A[i,j] := (A[i-1,j]+A[i,j-1] +
           A[i+1,j] + A[i, j+1]) / 4
```

Listing (23) *Loop Skewing* Aufgrund gegebener Abhängigkeitsdistanzen ist also keine der Schleifen parallelisierbar

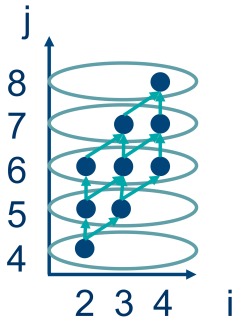


(a) *Loop Skewing* - Schleifenneigen - Wellenparallelität

Abbildung 29: *Loop Skewing* - Schleifenneigen - Beispiel für Wellenparallelität
Hier ist eine wellenförmige Parallelisierung möglich - Entsprechende Umwandlung: fig. 30

```
1 //Abhängigkeitsdistanzen: (1,1), (0,1)
2 for i := 2 to n-1
3    for j := 2+i to m-1+i
4       A[i,j-i] := (A[i-1,j-i] +
            A[i,j-1-i] + A[i+1,j-i] +
            A[i,j+1-i])/4
```

Listing (24) *Loop Skewing* - Beispiel für Wellenparallelität nach der Umwandlung



(a) *Loop Skewing* - Schleifenneigen - Wellenparallelität nach der Umwandlung
innere Schleife ist nach vertauschen parallelisierbar, da Abhängigkeitsdistanzen nach Vertauschung (1,1), (1,0)

Abbildung 30: *Loop Skewing* - Schleifenneigen - Beispiel für Wellenparallelität nach der Umwandlung - original: fig. 29
geneigter Iterationsraum, Welle entlang einer Schleife - Beispiel listing 27

Lineare Schleifenstrukturierung:

Ist eine Veralgemeinerung von Schleifenvertauschung, Richtungsumkehr und Schleifenneigen und hilft bei gezielter Konstruktion von Restrukturierung. Also man möchte eine bestimmte Restrukturierung vornehmen und versucht diese legal werden zu lassen, indem legale Bedingungen geschaffen werden.

Matrix U - unimodular:

Matrix U kann zur Transformation von Indexvektoren verwendet werden. Identitätsmatrix (I)

$$\forall_{ij} u_{ij} \in \mathbb{Z}, U \in \mathbb{Z}^{n \times n}$$

$\exists U^{-1} : UU^{-1} = I$ und $U^{-1} \in \mathbb{Z}^{n \times n}$

U ist genau dann ganzzahlig invertierbar, wenn $\exists U^{-1} : UU^{-1} = I, U^{-1} \in \mathbb{Z}^{n \times n} \Leftrightarrow detU \in \{-1, +1\}$ Solche Matrizen heißen *unimodular*. Es gibt drei Zeilentransformationen, die die Unimodularität beibehalten:

- Addition eines ganzzahligen Vielfachen einer Zeile zu einer anderen
- Vertauschung zweier Zeilen
- Multiplikation einer Zeile mit -1

Legalitätsregel von Restrukturierungen:

Gab es vor der Restrukturierung eine Abhängigkeit zwischen Iteration i und j, so muss diese auch im Nachhinein bestehen, also die Iteration iU muss vor jU ausgeführt werden: $i\delta j \Rightarrow iU \angle jU$.

Entsprechend muss für alle Abhängigkeitsdistanzen d gelten: $0 \angle dU$

Schleifenvertauschung: Vertauschungsmatrix:

Eine Schleifenvertauschung lässt sich darstellen als:

$$U = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & 0 & 1 & & \\ & & 1 & 0 & & \\ & & & & 1 & \\ & & & & & 1 \end{pmatrix} \quad U^{-1} = U, \text{ beliebiger Dimension}$$

```
1 //Abhängigkeitsdistanzen (0,1), (1,-1)
2 for i := 2 to n
3    for j := 0 to n-1
4       A[i,j+i] := 0.5*(A[i,j+i-1]+A[i-1,j+i])
```

Listing 25: Ist Schleifenvertauschung (Loop interchange) in diesem Fall legal?

$$U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$0 \angle (0,1)U = (1,0)$

$0 \not\angle (1,-1)U = (-1,1)$ ↯

Vertauschung ist illegal

Richtungsumkehr - Sonderfall linearer Schleifenrestrukturierung:

$$U = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & -1 & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix} \quad U^{-1} = U$$

```
1  //Abhängigkeitsdistanz: (1,-1)
2  for i := 1 to n
3     for j := 1 to n
4        A[i,j] := A[i-1,j+1]+1
5
6  //Richtungsumkehr
7  //Abhängigkeitsdistanz: (1,1)
8  for i := 1 to n
9     for j := n to 1 by -1
10       A[i,j] := A[i-1,j+1]+1
```

Listing 26: Vertauschung wird durch die Richtungsumkehr legal, vorher illegal

Richtungsumkehrmatrix: $U = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \Rightarrow 0 \angle (1,-1)U = (1,1)$

Schleifenneigen:

$$U = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1f & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix} \quad U^{-1} = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1-f & & \\ & & & 1 & \\ & & & & 1 \end{pmatrix}$$

```
1  //Abhängigkeitsdistanzen: (0,1), (1,-1)
2  for i := 2 to n
3     for j := 0 to n-1
4        A[i,j+i] := 0.5 * (A[i,j+i-1] + A[i-1,j+i])
5
6  //Schleifenneigen - Faktor f = 1
7  //Abhängigkeitsdistanzen: (0,1), (1,0)
8  for i := 2 to n
9     for j := i to n
10       A[i,j] := 0.5 * (A[i,j-1] + A[i-1,j])
```

Listing 27: Anwendung der Neigungsmatrix - Schleifenneigen

Neigungsmatrix: $U = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \Rightarrow 0\angle(0,1)U = (0,1)$ und $0\angle(1,-1)U = (1,0)$

```
1 //Abhängigkeitsdistanzen: (0,1), (2,-3), (3,-7)
2 for i := 5 to 100
3    for j := 5 to 100
4       A[i,j] := A[i,j-1] + A[i-2,j+3] + A[i-3,j+7]
```

Listing 28: Schleifenrestrukturierung - Wird umgeformt zu listing 29

Abhängigkeitsdistanzvektoren $D : \begin{pmatrix} 0,1 \\ 2,-3 \\ 3,-7 \end{pmatrix}$ Keine der Schleifen kann parallel ausgeführt werden und sie sind auch nicht vertauschbar. Gesucht ist also ein $U^T, U^T D^T = D^{T'}, D^{T'}$ gewisse Bedingungen erfüllt Die Umrechnungen:

$$\left(\begin{array}{cc|ccc} 1 & 0 & 0 & 2 & 3 \\ 0 & 1 & 1 & -3 & -7 \end{array}\right) \xrightarrow{\text{tauschen der beiden Zeilen}} \left(\begin{array}{cc|ccc} 0 & 1 & 1 & -3 & -7 \\ 1 & 0 & 0 & 2 & 3 \end{array}\right)$$

das ist ein illegaler Zustand, aufgrund der negativen Werte in der ersten Zeile, also addiere nun untere Zeile 3 mal auf obere

$$\left(\begin{array}{cc|ccc} 3 & 1 & 1 & 3 & 2 \\ 1 & 0 & 0 & 2 & 3 \end{array}\right)$$

Der linke Teil ist nun U^T und der Rechte $D^{T'}$, entsprechend ist $U^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix}$ und die Transformation sieht wie folgt aus $\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} j' \\ i' - 3j' \end{pmatrix}$ Und der entstandene Code sieht wie folgt aus

```
1 A[j',i'-3j'] := A[j,i'-3j'-1] + A[j'-2,i'-3j'+3] +
     A[j'-3,i'-3j'+7]
```

Listing 29: Anpassung der Indices nach Schleifenrestrukturierung - nach listing 28 - komplett dann in listing 30

Jetzt noch die Schleifengrenzen mit Fourier-Motzkin anpassen:

$$\begin{pmatrix} 5 \leq i \leq 100 \\ 5 \leq j \leq 100 \end{pmatrix} \Rightarrow \begin{pmatrix} 5 \leq j' \leq 100 \\ 5 \leq i' - 3j' \leq 100 \end{pmatrix}$$

$$-\frac{1}{3}(100 - i') \leq j' \leq -\frac{1}{3}(5 - i')$$

$$\max\left(5, \frac{1}{3}(i' - 100)\right) \leq j' \leq \min\left((100, \frac{1}{3}(i' - 5)\right)$$

$\uparrow$ neue Schleifengrenzen für j', nun daraus 4 Ungleichungen für i' bilden:

$$5 \leq 100$$

$$5 \leq \frac{1}{3}(i' - 5)$$

$$\frac{1}{3}(i' - 100) \leq 100$$

$$\frac{1}{3}(i' - 100) \leq \frac{1}{3}(i' - 5)$$

$$\Rightarrow 20 \leq i' \leq 400$$

```
for i' := 20 to 400
    for j' := ⌈max(5, 1/3(i' − 100))⌉ to ⌊min(100, 1/3(i' − 5))⌋
        A[j',i'-3j'] := A[j,i'-3j'-1] + A[j'-2,i'-3j'+3] +
            A[j'-3,i'-3j'+7]
```

Listing 30: Anpassung der ganzen Schleife nach Schleifenrestrukturierung - nach listings 28 and 29

j'-Schleife ist nun parallelisierbar mit dem Abhängigkeitsdistanzvektor $\begin{pmatrix} 1, 0 \\ 3, 2 \\ 2, 3 \end{pmatrix}$

```
for i := 5 to 100
    for j := 16 to 80
        A[i,j] := A[i-2,j-4] + A[i-3,j-6]
```

Listing 31: Schleifenrestrukturierung - Äußere Schleife parallelisieren - Anpassung der Indices nach listing 32

Entsprechender Abhängigkeitsvektor: $\begin{pmatrix} 2, 4 \\ 3, 6 \end{pmatrix}$ aus diesem folgen die Umformungen Das swap in der folgenden Berechnung müsste notwendig sein, damit der negative Wert sich auf der Diagonalen befindet.

$$\left(\begin{array}{cc|cc} 1 & 0 & 2 & 3 \\ 0 & 1 & 4 & 6 \end{array}\right) \xrightarrow{\text{II-2*I}} \left(\begin{array}{cc|cc} 1 & 0 & 2 & 3 \\ -2 & 1 & 0 & 0 \end{array}\right) \xrightarrow{\text{swap}} \left(\begin{array}{cc|cc} -2 & 1 & 0 & 0 \\ 1 & 0 & 2 & 3 \end{array}\right)$$

Wobei der linke Teil U^T und der rechte D^T', entsprechend ist $U^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}$ und

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{pmatrix} j' \\ i' + 2j' \end{pmatrix}$$

```
1 A[j',i'+2j'] := A[j'-2,i'+2j'-4] + A[j'-3, i'+2j'-6]
```

Listing 32: Anpassung der Indices nach Schleifenrestrukturierung - Äußere Schleife parallelisieren - original: listing 31, komplette Schleife: listing 33

Schleifengrenzen anpassen mit Fourier-Motzkin:

$$\begin{pmatrix} 5 \leq i \leq 100 \\ 16 \leq j \leq 80 \end{pmatrix} \Rightarrow \begin{pmatrix} 5 \leq j' \leq 100 \\ 16 \leq i' + 2j' \leq 80 \end{pmatrix}$$

$\Rightarrow$

$5 \leq j' \leq 100$

$\frac{1}{2}(16 - i') \leq j' \leq \frac{1}{2}(80 - i')$

$\Rightarrow \max\left(5, 8 - \frac{i'}{2}\right) \leq j' \leq \min\left(100, 40 - \frac{i'}{2}\right)$

$\uparrow$ Schleifengrenzen für j', nun daraus 4 Ungleichungen für i' bilden:

$5 \leq 100$

$5 \leq 50 - \frac{i'}{2}$

$8 - \frac{i'}{2} \leq 100$

$8 - \frac{i'}{2} \leq 40 - \frac{i'}{2}$

$\Rightarrow -184 \leq i' \leq 70$

```
1 for i' := -184 to 70
2     for j' := ⌈max(5, 8 − i'/2)⌉ to ⌊min(100, 40 − i'/2)⌋
3         A[j',i'+2j'] := A[j'-2,i'+2j'-4] + A[j'-3, i'+2j'-6]
```

Listing 33: Anpassung der gesamten Schleife nach Schleifenrestrukturierung - Äußere Schleife parallelisieren - nach listings 31 and 32

mit neuem Abhängigkeitsvektor $\begin{pmatrix} 0, 2 \\ 0, 3 \end{pmatrix}$ also ist die äußere Schleife nun parallelisierbar

12-52ff

13 Abbildungsverzeichnis

Abbildungsverzeichnis

14 Tabellenverzeichnis

Tabellenverzeichnis

15 Listingverzeichnis

List of Listings

Liste der noch zu erledigenden Punkte