Ausgewählte Kapitel aus dem Übersetzerbau

Prof. Gerhard Goos
Fakultät für Informatik
Universität Karlsruhe

Sommersemester 2006

http://www.info.uni-karlsruhe.de/

Cache Optimierung

—

Optimierungen für schnellen Speicherzugriff

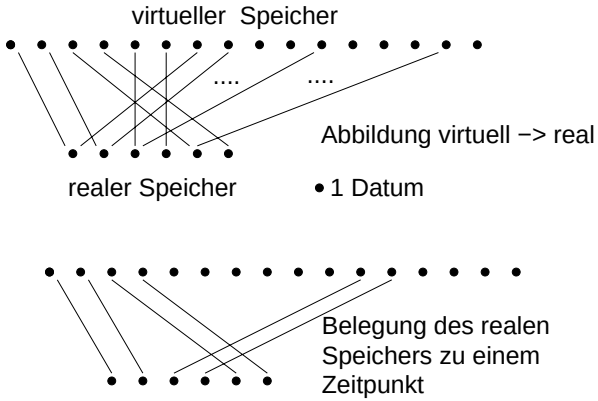
Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
  - Speicherzugriffe
  - *Caches*
  - Fehlzugriffe
  - *Caches* in der Praxis
  - Fehlzugriffe: Beispiele
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
  - Abhängigkeitsanalyse
  - Schleifentransformationen
  - Schleifenrestrukturierung
  - Anordnungstransformationen
- Optimierungen für Dynamische Datenstrukturen
- Fehlzugriffe tolerieren

Speicherzugriffe

- Jedes Programm greift auf Daten zu.
- Zugriffe auf die Daten werden anhand virtueller Adressen in virtuellen Speicherzellen verwaltet.
- Virtuelle Speicherzellen werden bei Bedarf in realen Speicher geladen.
- Betriebssystem bildet virtuelle auf reale Adressen des Systems ab:
  - Hauptspeicher
  - ausgelagerte Seiten auf Festplatte
- Es wird immer nur eine Teilmenge des virtuellen Adreßraums auf reale Adressen eines Speichers abgebildet (da dieser kleiner ist).
- Wir sagen: Eine virtuelle Speicherzelle oder Adresse ist in einem realen Speicher vorhanden.

Beispiel: Abbildung des virtuellen Adreßraums



Latenz eines Speicherzugriffs

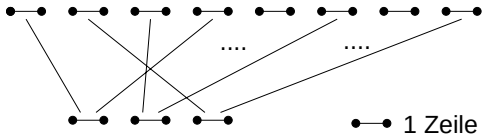
- Latenz eines Speichers: Dauer, bis ein Datum aus einem Speicher geladen ist.
- Die Dauer eines Zugriffs hängt davon ab, auf welchen realen Speicher eine Adresse abgebildet ist.
- Dauer eines Hauptspeicherzugriffs zu langsam: $\geq$ 50 Takte
- Einsatz schnellerer teurerer Pufferspeicher: *Caches*
- Praxis: Speicherhierarchie; wir betrachten nur einen *Cache*.

Caches

- Abbildung virtuelle auf reale Adressen in Hardware (schnell)
- Zeitvorteil erst beim 2. Zugriff auf eine Adresse: Beim ersten Zugriff muß diese immer noch aus dem Hauptspeicher geladen werden
- *Cache* klein: Abbildung virtuelle $\rightarrow$ reale Adresse nicht eineindeutig: Verdrängung von Daten.
- Problem: Verdrängung von Daten vor deren Wiederverwendung: Vorteil des *Caches* wird nicht ausgenutzt.
- Organisation der *Caches* in *Zeilen*: Mehrere aufeinander folgende virtuelle Speicherzellen bilden eine Zeile und werden gemeinsam verwaltet: Beim Zugriff auf eine Zelle in der Zeile wird die ganze Zeile geladen
- Eine reale Zeile im Hauptspeicher heißt auch Kachel.

Beispiel: Cachezeilen

- Es werden immer zwei benachbarte virtuelle Zellen auf benachbarte reale Zellen abgebildet



Fehlzugriffe

Definition Fehlzugriff:
Ein Zugriff auf eine Adresse ist ein Fehlzugriff in einem Speicher, wenn die zugehörige Zelle hier nicht vorhanden ist.

Definition Treffer:
Ein Zugriff auf eine Adresse ist ein Treffer in einem Speicher, wenn die zugehörige Zelle hier vorhanden ist.

Beobachtung: *Cache*zeilen ermöglichen, daß erste Zugriffe auf eine Zelle Treffer sind.

Gründe für Fehlzugriffe:

- Obligatorischer Fehlzugriff: erster Zugriff auf eine Zeile
- Kapazitätsfehlzugriff: Zelle wurde verdrängt, da *Cache* zu klein
- Zeilenkapazitätsfehlzugriff: Zelle wurde verdrängt, da Zeilen schlecht organisiert
- Konfliktfehlzugriff: Zelle wurde verdrängt, da zu viele andere virtuelle Zellen auf die benötigte reale Zelle abgebildet wurden.

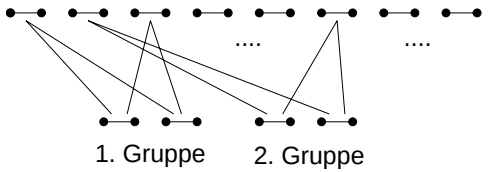
Caches in der Praxis

- Für eine virtuelle Zeile gibt es mehrere (1, 2, 4, 8, ...) mögliche Positionen im *Cache*.
  Die Anzahl möglicher Positionen heißt Assoziativität des *Cache*.
  - Assoziative Zeilen bilden eine Gruppe.
  - Assoziativität verringert Konfliktfehlzugriffe.
  - Hohe Assoziativität macht den *Cache* langsam oder teuerer (mehr Vergleiche nötig)
  - 1-Assoziativität heißt auch direkt abbildend.
  - Hauptspeicher: vollassoziativ, jede Position möglich.
  - Protokoll, das entscheidet, welche Zeilen ausgelagert werden (z.B. LRU)
- Die letzten Bits einer virtuellen Adresse adressieren die Zellen innerhalb einer Zeile,
- die nächsten Bits identifizieren eine Gruppe,
- die restlichen Bits werden mit der Zeile abgespeichert.

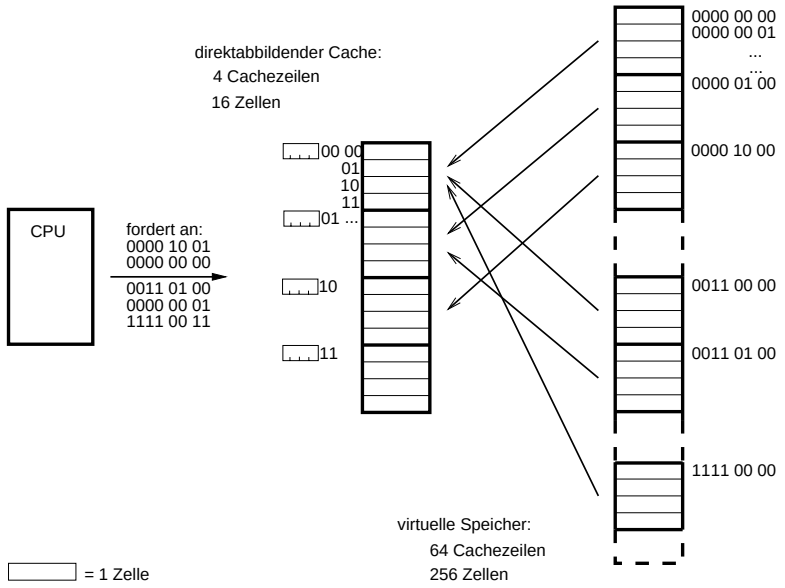
Assoziativität: Beispiel

- 2-assoziativer Speicher
- Jede virtuelle Zeile einer Gruppe zugeordnet
- Alle Zeilen innerhalb einer Gruppe mögliche Position für v. Zeile

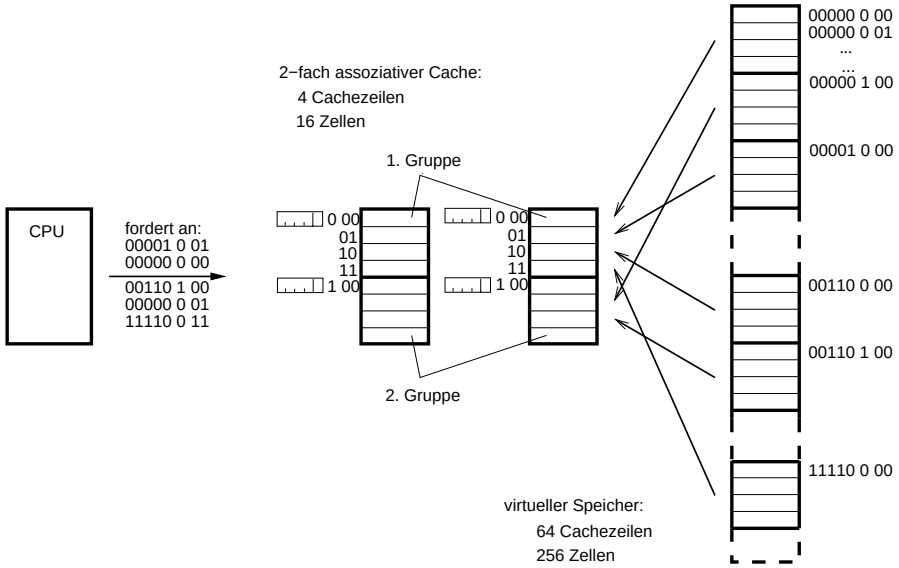
$\Rightarrow$ Abbildung hat Alternativen



Caches in der Praxis – direkt abbildend

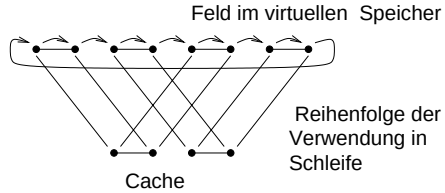


Caches in der Praxis – 2-fach Assoziativ



Kapazitätsfehlzugriffe: Beispiel

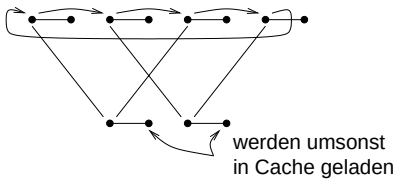
```
int A[8];
for i1 := 0 to 7 do
  for i2 := 0 to 7 do
    A[i2]:= ...
```



- Jedes Datenelement wird mehrfach verwendet
- Datenmenge paßt nicht in *Cache*
  Cache wird in jeder inneren Schleife zweimal neu geladen.
- Wiederverwendung wird von *Cache* nicht erfaßt. ⇒ Optimieren!
- (direktabbildender *Cache* mit 2 Zeilen à 2 Zellen)

Zeilenkapazitätsfehlzugriffe: Beispiel

```
int A[8];
for i1 := 0 to 7 do
  for i2 := 0 to 7 step 2 do
    A[i2]:= ...
```



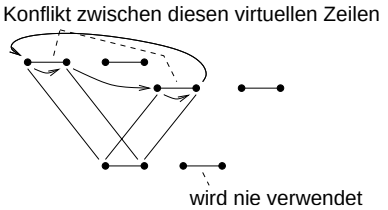
- Jedes Datenelement wird mehrfach verwendet
  (äußere Schleife)
- Datenmenge paßt in *Cache*
- Durch Zeilenbildung werden unnötige Daten in den *Cache* geladen.

⇒ daher ständig Fehlzugriffe.

- Wiederverwendung wird von *Cache* nicht erfaßt. ⇒ Optimieren!

Konfliktfehlzugriffe: Beispiele

```
int A[2][4];
for i1 := 0 to 7 do
  for i2 := 0 to 1 do
    for i3 := 0 to 1 do
      A[i2,i3]:= ...
```



- Jedes Datenelement wird mehrfach verwendet
- Datenmenge paßt in *Cache*
- Verwendete virtuelle Zeilen werden auf die gleiche(n) realen Zeilen abgebildet

⇒ ständig Fehlzugriffe.

- Wiederverwendung wird von *Cache* nicht erfaßt. ⇒ Optimieren!

Literatur

Beschreibt Funktionsweise der *Caches*:
Mark D. Hill, A Case for Direct-Mapped Caches, *IEEE Computer*, 21(12), S. 25–40, Dez. 1988.

Standardlehrbuch:
John L. Hennessy, David A. Patterson:
Computer Architecture: A Quantitative Approach,
Morgan Kaufmann Publishers, 3. Auflage, 2002.

Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
    - Datenabhängigkeiten
    - Iterationsraum
    - Abhängigkeitsdistanz
  - Abhängigkeitsanalyse
  - Schleifentransformationen
  - Schleifenrestrukturierung
  - Anordnungstransformationen
- Optimierungen für Dynamische Datenstrukturen
- Fehlzugriffe tolerieren

Schleifenoptimierungen für Reihungen

Warum Schleifenoptimierungen für Reihungen?
- Schleifenrümpfe sind am häufigsten ausgeführter Code
- Reihungen bedeuten große Datenmengen

$\Rightarrow$ wenig Code zu optimieren, großer Effekt.
- Zugriffe in Schleifen auf Reihungen sind analytisch handhabbar: Datenabhängigkeiten können analysiert werden.

Techniken sind i.d.R. auch für Parallelisierung wichtig.

Voraussetzungen

Wir setzen oft implizit voraus:
- Perfekt geschachtelte Schleifen (Rumpf der äußeren Schleifen besteht nur aus innerer Schleife).
- Reihungszugriffe in Abhängigkeit der Schleifenindizes $i_1, \ldots, i_d$ oder Schleifenkonstanten: $A[f(i_1, \ldots, i_d)]$
- Wir vernachlässigen Randbedingungen von Schleifen, Reihungen in allen Beispielen.

Wiederholung Datenabhängigkeiten

Seien S_1, S_2 zwei Programmstellen, S_1 vor S_2 im Code.

So bezeichnen wir

- Datenabhängigkeit (*Read-after-Write*): $S_1 \delta^f S_2$
- Ausgabe-Abhängigkeit (*Write-after-Write*): $S_1 \delta^o S_2$
- Anti-Abhängigkeit (*Write-after-Read*): $S_1 \delta^a S_2$
- Eingabe-Abhängigkeit (*Read-after-Read*): $S_1 \delta^e S_2$

Beispiel Datenabhängigkeiten

S_1: `A := 0;`
S_2: `B := A;`
S_3: `C := A + D;`
S_4: `D := 2;`
S_5: `C := C - 1;`

$S_1\ \delta^f\ S_2$
$S_1\ \delta^f\ S_3$
$S_2\ \delta^e\ S_3$
$S_3\ \delta^a\ S_4$
$S_3\ \delta^f\ S_5$
$S_3\ \delta^o\ S_5$

Datenabhängigkeiten

- Schleifenunabhängige Datenabhängigkeiten
  - innerhalb einer einzelnen Iteration
  - nimmt man Schleifenkonstrukt weg, bleibt die Abhängigkeit
- Schleifengetragene Datenabhängigkeiten (“loop carried”)
  - zwischen verschiedenen Iterationen

```
   for i := 0 to n do
S1    a[i] := ...;
S2    ...  := a[i];
S3    ...  := a[i-1];
```

$S_1\ \delta^f\ S_2$ schleifenunabhängig
$S_1\ \delta^f\ S_3$ schleifengetragen

Iterationsraum

Der *Iterationsraum* beschreibt alle Iterationen einer Schleifenschachtel. Ein *Iterationsvektor* beschreibt eine Iteration.

Schleifenschachtel:

```
for i1 := ...do
  for i2 := ...do
    ...
      for id := ...do
```

Iterationsvektor:

$$i = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_d \end{pmatrix}$$

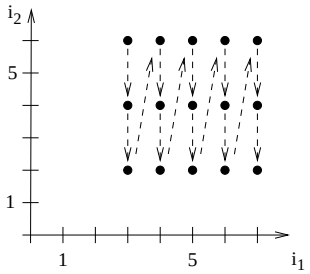
Eine Schleifenschachtel legt neben dem Iterationsraum auch die Reihenfolge der Iterationen fest. Diese ist eine Ordnung $\prec$ der Iterationen.

Iterationsraum: Beispiel

Schleifenschachtel:

```
for i1 := 3 to 7 do
  for i2 := 6 to 2 step -2 do
    ...
```

Iterationsraum und Reihenfolge:



Reihungsreferenzen als Funktionen des Iterationsvektors

Reihungsreferenzen sind Funktionen des Iterationsvektors.

Eine Referenz heißt *affin*, wenn sie eine lineare Abbildung des Iterationsvektors plus einem konstanten Vektor ist.

Beispiel: `A[2i1+1, i1+i2+3]`: $A\left[\begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix} i + \begin{pmatrix} 1 \\ 3 \end{pmatrix}\right]$

Zwei Reihungsreferenzen $A[f(i)]$ und $B[g(i)]$ heißen *uniform*, wenn sie affin sind und es eine lineare Abbildung H und konstante Vektoren c_f und c_g gibt so daß

$$f(i) = Hi + c_f \quad \text{und} \quad g(i) = Hi + c_g$$

.

Abhängigkeitsdistanz

Die *Abhängigkeitsdistanz* d einer Datenabhängigkeit beschreibt, zwischen welchen Iterationen Abhängigkeiten bestehen.

Sei $S_1 \delta S_2$ Abhängigkeit zwischen Iterationen i^{Quelle} und i^{Senke}, $i^{Quelle} \prec i^{Senke}$, dann gilt:

$$d = i^{Senke} - i^{Quelle}$$

Abhängigkeitsdistanzen können nur für Abhängigkeiten uniformer Referenzen angegeben werden.

Für nicht uniforme Referenzen kann man Richtungsvektoren angeben.

Abhängigkeitsdistanz: Beispiel

uniform	nicht uniform

`for i1 := 3 to 7 do`
`  for i2 := 2 to 6 step 2 do`
`  A[i1,i2] := A[i1,i2+2]+1;`
	`for i1 := 3 to 7 do`
	`  for i2 := 2 to 6 step 2 do`
	`  A[2*i1,i2]:=A[i1,i2+2]+1;`

$(3,2)\ \delta^a\ (3,4)$ $(3,4)\ \delta^a\ (3,6)$ $(4,2)\ \delta^a\ (4,4)$ $(4,4)\ \delta^a\ (4,6)$ $\vdots$ $(7,2)\ \delta^a\ (7,4)$ $(7,4)\ \delta^a\ (7,6)$	$(6,2)\ \delta^a\ (3,4)$ $(6,4)\ \delta^a\ (3,6)$ $(8,2)\ \delta^a\ (4,4)$ $(8,4)\ \delta^a\ (4,6)$ $\vdots$ $(14,2)\ \delta^a\ (7,4)$ $(14,4)\ \delta^a\ (7,6)$
$i^{Quelle} + (0,-2) = i^{Senke}$	$(>,<)$

Abhängigkeitsdistanz

Bei vorwärts laufenden Schleifen haben Abhängigkeitsdistanzen folgendes Aussehen:

$$d = (0, \ldots 0, d_p, \ldots, d_n), \quad d_p > 0$$

- Schleife p „trägt“ die Abhängigkeit
- > 0 da eine Abhängigkeit nur zu nachfolgenden Iterationen bestehen kann.

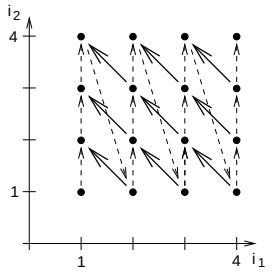
Schleife q ist parallel ausführbar, wenn für alle Abhängigkeitsdistanzen $d = (d_1, \ldots, d_n)$ gilt:

$$d_q = 0 \quad \vee \quad \exists q' < q : d_{q'} > 0$$

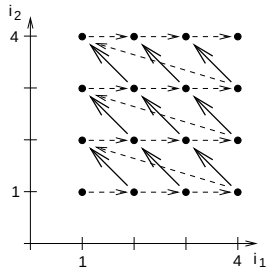
Bedeutung der Datenabhängigkeiten

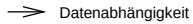
- Schränken Optimierungen ein

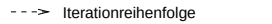
```
for i1 := 1 to 4 do
  for i2 := 1 to 4 do
    A[i1,i2]:=A[i1-1,i2+1]+1;
```




```
for i2 := 1 to 4 do
  for i1 := 1 to 4 do
    A[i1,i2]:=A[i1-1,i2+1]+1;
```







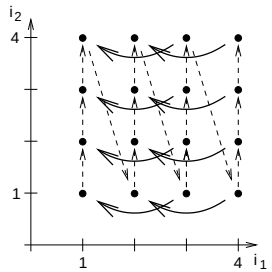
- Verwendung einer Definition aus bereits ausgeführter Iteration
- Verwendung einer Definition aus zukünftiger Iteration

Schleifen vertauschen illegal!

Bedeutung der Datenabhängigkeiten

- Schränken Parallelisierung ein

```
for i1 := 1 to 4 do
  for i2 := 1 to 4 do
    A[i1,i2]:=A[i1-2,i2]+1;
```



- i_1-Schleife kann wegen Datenabhängigkeiten nicht parallel ausgeführt werden.
- i_2-Schleife kann parallel ausgeführt werden.

Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
  - Abhängigkeitsanalyse
    - Problembeschreibung
    - GGT-Test
    - Ungleichungstest
    - Fourier-Motzkin Elimination
  - Schleifentransformationen
  - Schleifenrestrukturierung
  - Anordnungstransformationen
- Optimierungen für Dynamische Datenstrukturen
- Fehlzugriffe tolerieren

Abhängigkeitsanalyse

Wie klärt der Übersetzer, welche Reihungsreferenzen Abhängigkeiten implizieren?

- Simulation wie im Beispiel vorher (Folie 28) nicht praktikabel.

Allgemeine Problemformulierung:
Gegeben: zwei Referenzen S_1, S_2 mit zwei Indexfunktionen f_1, f_2.
Gesucht: Alle Paare (i, j) Iterationsvektoren, $i \prec j$ so daß $f_1(i) = f_2(j)$

Gibt es mindestens ein solches Paar, sind die Referenzen abhängig.

- Lösung muß innerhalb der Schleifengrenzen liegen.
- Lösung muß ganzzahlig sein.

Abhängigkeitsanalyse

- Problem ist NP-vollständig, weil äquivalent zum Lösen von Systemen von linearen diophantischen Gleichungen, was ein NP-vollständiges Problem ist.
- Einschränkung: Abhängigkeitsanalyse nur für affine Referenzen in perfekten Schleifenschachteln
- Aliase der Reihungen? Überlappungen?
- pessimistischer Ansatz: zunächst sind alle Referenzen voneinander abhängig
- Verschiedene Test auf Unabhängigkeit durchführen – diese können auch bestimmte Abhängigkeiten feststellen
- Gefundene Abhängigkeiten analysieren

Abhängigkeitsanalyse

Bei der Auswahl einer Abhängigkeitsanalyse müssen folgende Punkte abgewogen werden

- Effizienz: Laufzeit des Testalgorithmus
- Präzision:
  - Wie konservativ ist der Algorithmus?
  - D.h. wie viele Unabhängigkeiten übersieht der Algorithmus?
- Anwendbarkeit auf Problem

Abhängigkeitsanalyse: GGT-Test

```
for i := 2 to 10 do
  A[2*i+2] := A[2*i-2] +1;
```

$S_1 =$ `A[2*i+2]`; $S_2 =$ `A[2*i-2]`;
$f_1 = 2 * i_{def} + 2$; $f_2 = 2 * i_{use} - 2$;

$$2i_{def} + 2 = 2i_{use} - 2$$

(Zum Feststellen der Abhängigkeitsdistanz muß Herkunft der Indizes gemerkt werden.)

$$\Leftrightarrow 2i_{def} - 2i_{use} = -4$$

Wenn überhaupt, dann gibt es nur eine Abhängigkeit, wenn der GGT der Koeffizienten die Konstante teilt.

$ggt(2, 2) = 2$; 2 teilt -4

eventuell abhängig.

$$i_{def} - i_{use} = -2$$

Abhängigkeitsanalyse: GGT-Test

$$i_{def} - i_{use} = -2$$

1. Möglichkeit: δ^f

$$i_{def} + 2 = i_{use}$$

δ^f mit Abhängigkeitsdistanz 2

Kontrolle der Schleifengrenzen und Schrittweite ergibt: Abhängigkeit kann auftreten

2. Möglichkeit: δ^a

$$i_{use} - 2 = i_{def}$$

Da $i_{use} < i_{def}$ sein muß, kann es diese Abhängigkeit nicht geben.

Weitere Abhängigkeitsanalysen

Ungleichungstest
- Testet, ob Abhängigkeitsdistanz innerhalb der Schleifengrenzen
- z.B. für `A[i]` und `A[i+100]`: wenn Schleife nur bis 10 iteriert, liegt keine Abhängigkeit vor.

Fourier-Motzkin Elimination
- Formuliere LGS für die Abhängigkeiten und löse dieses.
- Aufwendigerer, aber effektiver Algorithmus.

Literatur

D. E. Maydan and J. L. Hennessy and M. S. Lam, Efficient and Exact Data Dependence Analysis, *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6), Toronto, Canada, Jun. 1991.

G. Goff and K. Kennedy and C. Tseng, Practical Dependence Testing, *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, 26(6), Toronto, Canada, Jun. 1991

Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
  - Abhängigkeitsanalyse
  - Schleifentransformationen
    - Schleifen ausrollen
    - Schleifenbereichsteilen
    - Schleifen schälen
    - Produktschleifenbildung
    - Streifenschneiden
  - Schleifenrestrukturierung
  - Anordnungstransformationen
- Optimierungen für Dynamische Datenstrukturen
- Fehlzugriffe tolerieren

Schleifentransformationen

- erhalten die Reihenfolge der Iterationen

$\Rightarrow$ sind unabhängig von Datenabhängigkeiten

- dienen hauptsächlich dazu, Potential anderer Optimierungen zu verbessern
- keine *Cache*-Optimierungen.

Schleifen ausrollen

("loop unrolling")

Vorgehen:

- Schleifenrumpf vervielfachen ($k - fach$)
- Schrittweite der Schleife anpassen ($*k$)
- Schleifentest:
  - Schleifengrenze (n) konstant: $n \ mod \ k$ Iterationen vor oder nach Schleife ausführen (Schleifen schälen)
  - Schleifengrenzen unbekannt: Test auch vervielfachen (schlecht) oder Schleifenbereichsteilen.
- Rolle kleine Schleifen komplett aus.

Schleifen ausrollen

Vorteile:

- Grundblock für Rumpf wird größer: Fließband des Prozessors besser ausgenutzt
- Reduziert Schleifenmehraufwand
- Mehr Möglichkeiten für Standardoptimierungen durch größeren Grundblock

Nachteile:

- Rumpf muß immer noch in Befehlscache passen.
- Andere Schleifenoptimierungen schwieriger, da Schleifenstruktur komplizierter.
- Programm wird größer.

Schleifen ausrollen: Beispiel

- Auf Quellcodeebene
- Setzt Kenntnis der Schleife, Indizes . . . voraus

```
   for i := 0 to 17 do
      A[i]:=A[i-1]+A[i+1];
```

- kleiner Schleifenrumpf

```
for i := 0 to 16 step 4 do {
  A[i]:=A[i-1]+A[i+1];
  A[i+1]:=A[i]+A[i+2];
  A[i+2]:=A[i+1]+A[i+3];
  A[i+3]:=A[i+2]+A[i+4];
}
A[17]:=A[16]+A[18];
```

- Schleife 4-mal ausgerollt.
- Schrittweite vervierfacht.
- Restliche Iterationen am Ende nachgeholt (geschält).
- Großer Schleifenrumpf.
- Neue Optimierungsmöglichkeit: z.B. `A[i]`, `A[i+1]` . . . in Register.

Schleifen ausrollen: Beispiel

- Auf Zielsprachebene
- Anpassen der Schleifenindizes . . . durch Standardoptimierungen nötig.

```
LOOP:  COMPUTE ...
       ADD i, 1
       CMPge (i,n), CONT
       GOTO LOOP
CONT:  ...
```

- kleiner Schleifenrumpf

Schleifen ausrollen: Beispiel

```
LOOP:  COMPUTE ...
       ADD i, 1
       CMPge (i,n), CONT
       COMPUTE ...
       ADD i, 1
       CMPge (i,n), CONT
       COMPUTE ...
       ADD i, 1
       CMPge (i,n), CONT
       GOTO LOOP
CONT:  ...
```

- Schleife 3-mal ausgerollt
- Extrem einfach durchzuführen
- Sprungoperationen bleiben erhalten
  $\Rightarrow$ sollten noch entfernt werden!
  So wenig Möglichkeiten für Optimierungen
- Besseres BefehlsCacheverhalten, evtl. gut für Befehlsanordnung

Schleifenbereichsteilen

("index set splitting")

Vorgehen:
- Teile eine Schleife in zwei
- $(0 \ldots n) \Rightarrow (0 \ldots n'), (n'+1 \ldots n)$

Vorteile:
- Ermöglicht andere Optimierungen
  - Ausrollen wenn $n' = n \ mod \ k$
  - Schleifen verschmelzen (siehe später)
- Bedingte Anweisung in Schleife mit Schleifenindex in Bedingung kann konstant werden

Nachteile:
- mehr Schleifenmehraufwand
- Programm wird größer

Schleifenbereichsteilen: Beispiel

```
for i := 0 to n do
  A[i]:=A[i-1]+A[i+1];
```

- Iterationsraum von 0 bis n.

```
for i := 0 to (n mod 4) do
  A[i]:=A[i-1]+A[i+1];
for i := (n mod 4)+1 to n do
  A[i]:=A[i-1]+A[i+1];
```

- Zwei Schleifen.

Schleifen schälen

("loop peeling") Vorgehen:
- Führe einige Iterationen am Anfang oder Ende der Schleife getrennt aus.
- $(0 \ldots n) \Rightarrow 0, 1, (2 \ldots n-1), n$
- Sonderfall Bereichsteilen
- Achtung: evtl. trotzdem Bedingung testen!

Vorteile:
- Iterationsraum kann für andere Optimierungen angepaßt werden

Nachteile:
- Programm wird größer

Legalität:
- immer anwendbar

Schleifen schälen: Beispiel

```
n := ...;
for i := 0 to n do
  body
```

```
if (0 < n) {
  i := 0; body;}
for i := 1 to n do
  body
```


Schleifen schälen: Größeres Beispiel

```
for i := 2 to n do
  b[i] := b[i]+b[2];
for i := 3 to n do
  a[i] := a[i] + c
```

```
if (2 <= n) {
  b[2] := b[2] + b[2] }
for i := 3 to n do {
  b[i] := b[i] + b[2]; a[i] := a[i] + c }
```


Weitere Schleifentransformationen

Produktschleifenbildung:
- Wandle zwei perfekt ineinander geschachtelte Schleifen in eine um
- $(0 \ldots n)(0 \ldots m) \Rightarrow (0 \ldots n * m)$

Streifenschneiden:
- Wandle eine Schleife in zwei perfekt ineinander geschachtelte um
- $(0 \ldots n * m) \Rightarrow (0 \ldots n)(0 \ldots m)$
- Umkehrung von Produktschleifenbildung

Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
  - Abhängigkeitsanalyse
  - Schleifentransformationen
  - Schleifenrestrukturierung
    - Schleifenpermutation
    - Schleifenumkehr
    - Schleifenneigen
    - Lineare Schleifenrestrukturierung
    - Kacheln
    - Schleifen verschmelzen
    - Schleifen teilen
  - Anordnungstransformationen
- Optimierungen für Dynamische Datenstrukturen
- Fehlzugriffe tolerieren

Schleifenrestrukturierungen

- Ändern im Gegensatz zu Schleifentransformationen die Reihenfolge der Iterationen
- Müssen Datenabhängigkeiten beachten
- Verbessern Speicher/*Cache*leistung
- Ermöglichen Parallelisierung

Schleifenpermutation

("loop permutation, loop interchange")

Vorgehen:
- Vertausche innere gegen äußere Schleifen
- zwei Schleifen: Schleifenvertauschung
- mehrere Schleifen: Schleifenpermutation

Vorteile:
- Schleifen, die Abhängigkeiten tragen, nach außen ziehen: innen bleibt u.U. parallelisierbare Schleife
- Innere Schleife soll Reihungselemente in Speicheranordnung durchlaufen:
  reduziert Zeilenkapazitätsfehlzugriffe
- Reihungszugriffe unabhängig von einer Schleife:
  nach innen, Reihungselemente in Register

Schleifenpermutation: Beispiel

- Fortran Speicheranordnung: `B[`i_1`,`i_2`]` und `B[`i_1+1`,`i_2`]` benachbart (spaltenweise Speicherung, "column major")

```
for i1 := 0 to n1 do
  for i2 := 0 to n2 do
    A[i1]:=A[i1] + B[i1,i2];
```

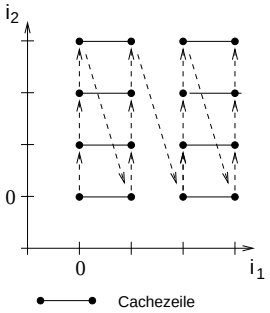
- `A[`i_1`]` kann für n_2 Iterationen in Register bleiben: $n_1 * n_2$ Speicheroperationen, $\approx n_1$ Fehlzugriffe
- `B[`i_1`,`i_2`]`: jeder Zugriff ein Fehlzugriff: $n_1 * n_2$ Fehlzugriffe

```
for i2 := 0 to n2 do
  for i1 := 0 to n1 do
    A[i1]:=A[i1] + B[i1,i2];
```

- `A[`i_1`]`: $n_1 * n_2$ Speicheroperationen, $n_1 * n_2 \frac{\text{Elementgröße}}{\textit{Cache}\text{größe}}$ Fehlzugriffe
- `B[`i_1`,`i_2`]`: nur $n_1 * n_2 \frac{\text{Elementgröße}}{\textit{Cache}\text{größe}}$ Fehlzugriffe

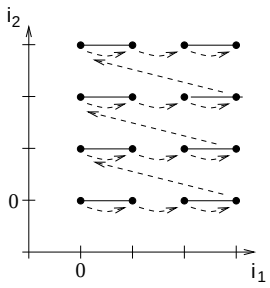
Schleifenpermutation: Beispiel

```
for i1 := 0 to 4 do
  for i2 := 0 to 4 do
    A[i1,i2]:=A[i1,i2] + 1;
```



- Iteration gegen Speicherlayout.
- Bei großem Feld Zeile bis zur Wiederverwendung verdrängt.

```
for i2 := 0 to 4 do
  for i1 := 0 to 4 do
    A[i1,i2]:=A[i1,i2] + 1;
```

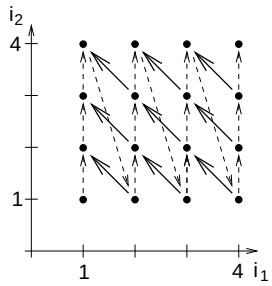


- Iteration mit Speicherlayout.
- Sofortiges Verwenden aller Zeilenelemente.

Schleifenpermutation: Legalität

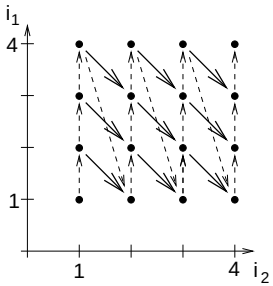
- Dimensionen der Abhängigkeitsvektoren werden wie Schleifen permutiert.
- Richtung der Abhängigkeit darf sich nicht ändern (Vorzeichen des 1. Eintrags $\neq 0$ darf sich nicht ändern).

```
for i1 := 1 to 4 do
  for i2 := 1 to 4 do
    A[i1,i2]:=A[i1-1,i2+1]+1;
```



Abhängigkeit $(<,>)$ bzw. $(1,-1)$

```
for i2 := 1 to 4 do
  for i1 := 1 to 4 do
    A[i1,i2]:=A[i1-1,i2+1]+1;
```



Abhängigkeit $(>,<)$ bzw. $(-1,1)$

Vertauschung illegal!

Schleifenumkehr

(“loop reversal”)

Vorgehen:

- Vertausche Laufrichtung einer Schleife

Vorteil:

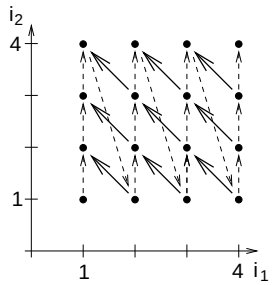
- Abhängigkeiten ändern sich: kann andere Restrukturierungen ermöglichen
- Maschinenbefehle „Springe bei Null“ können besser ausgenutzt werden

Legalität:

- Alle Abhängigkeiten müssen von äußeren Schleifen getragen werden (d ist der Abhängigkeitsvektor):
  p umkehrbar wenn $\forall d = (d_1, \ldots d_p, \ldots d_N)\ \exists i \in \{1, \ldots p-1\}\ d_i \neq 0$ oder wenn $p = 0$ gilt.
- Ist das nicht der Fall, kehrt sich eine Abhängigkeit um!

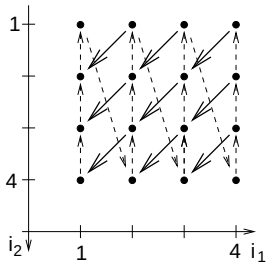
Schleifenumkehr: Beispiel

```
for i1 := 1 to 4 do
  for i2 := 1 to 4 do
    A[i1,i2]:=A[i1-1,i2+1]+1;
```



- Abhängigkeit (1, -1)

```
for i1 := 1 to 4 do
  for i2 := 4 downto 1 do
    A[i1,i2]:=A[i1-1,i2+1]+1;
```



- Abhängigkeit (1, 1)
- Jetzt Vertauschen erlaubt!

- Umkehr der i_1-Schleife ergäbe Abhängigkeit (-1, -1): illegal!

Schleifenneigen

(“loop skewing”)

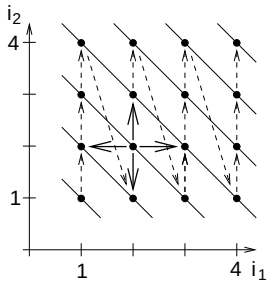
Vorgehen:
- Lege Neigungsfaktor f fest.
- Addiere Produkt aus Neigungsfaktor und äußerem Schleifenindex ($f * \mathtt{i}_1$) zu inneren Schleifengrenzen.
- Subtrahiere ($f*\mathtt{i}_1$) von Verwendungen der inneren Schleifengrenzen

Vorteile:
- Änderung der Datenabhängigkeiten: (d_1, d_2) wird zu $(d_1, f * (d_1 + d_2)$.
- Ermöglicht dadurch andere Restrukturierungen.
- Man kann f so festlegen, dass gewünschte Restrukturierung möglich.
- Beachte: Spezieller Algorithmus für Vertauschen, wenn der äußere Schleifenindex in Grenzen der inneren Schleife vorkommt.

Legalität:
- immer: erhält Iterationsreihenfolge

Schleifenneigen: Beispiel

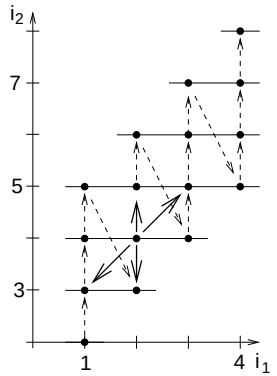
```
for i1 := 1 to 4 do
  for i2 := 1 to 4 do
    A[i1,i2]:=
      (A[i1-1,i2]+A[i1,i2-1]
       A[i1+1,i2]+A[i1,i2+1])/4
```



- Abhängigkeiten (1,0), (0,1), andere uninteressant.
- keine Schleife parallel ausführbar
- Vertauschen illegal
- Wellenfront parallel ausführbar

Schleifenneigen: Beispiel

```
for i1 := 1 to 4 do
  for i2 := i1+1 to i1+4 do
    A[i1,i2-i1]:=
      (A[i1-1,i2-i1]+A[i1,i2-i1-1]
       A[i1+1,i2-i1]+A[i1,i2-i1+1])/4
```



- Abhängigkeiten (1,1), (0,1)
- Vertauschen erlaubt
- nach Vertauschen: (1,1), (1,0)
  $\Rightarrow$ innere Schleife parallel ausführbar

Lineare Schleifenrestrukturierung

- Mathematisches Modell für
  - Vertauschen
  - Umkehr
  - Neigen
- Verwende lineare Abbildung/Matrix, um Schleifenrestrukturierung darzustellen
- Man bezeichnet diese Abbildungen als *unimodular*.
- Verknüpfen mehrerer Abbildungen zu komplexer Restrukturierung
- Wende Abbildung z.B. auf Abhängigkeitsvektoren an, um ihre Legalität zu testen.

Unimodulare Schleifenrestrukturierung

Eine unimodulare Schleifentransformation ist eine lineare Abbildung U, die mit einer unimodularen Matrix beschrieben werden kann. Eine unimodulare Matrix ist eine quadratische Matrix mit ganzzahligen Einträgen, deren Determinante 1 oder -1 ist.

Intuition bzw. Fakten über Matrizen:

- Nur quadratische Matrizen haben eine Determinante $\neq 0$.
- Determinante positiv: Orientierung des Volumens der abgebildeten Mengen bleibt erhalten.
- Determinante negativ: Orientierung des Volumens der abgebildeten Mengen ändert sich. Im zweidimensionalen Raum: z.B. Spiegelung.
- Determinante ist 1 oder -1: Volumen der abgebildeten Mengen bleibt gleich.
- Inverse Matrix hat im 2-dimensionalen Fall auch nur ganzzahlige Einträge: Einträge der Inversen sind Permutation der Einträge der ursprünglichen Matrix, geteilt durch die Determinante. Das gilt auch für den allgemeinen Fall. (genauere Infos in Standardliteratur zur linearen Algebra)

Lineare Schleifenrestrukturierung und Legalität

Gab es vor der Restrukturierung eine Abhängigkeit zwischen Iterationen i und j, so muß nach der Restrukturierung Iteration iU vor jU ausgeführt werden:

$$i \ \delta \ j \ \Rightarrow \ iU < jU.$$

D.h. für alle Abhängigkeitsdistanzen d muß gelten:

$$dU > 0.$$

(d wird zu dU in restrukturierter Schleife.)

Vertauschen als Lineare Schleifenrestrukturierung

Folgende lineare Abbildung realisiert Vertauschung benachbarter Schleifen:

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

$$U^{-1} = U$$

Umkehr als Lineare Schleifenrestrukturierung

Folgende lineare Abbildung realisiert Umkehr einer Schleife:

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

$$U^{-1} = U$$

Neigen als Lineare Schleifenrestrukturierung

Folgende lineare Abbildung realisiert Neigen einer Schleife:

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

Kacheln

("loop tiling, loop blocking")

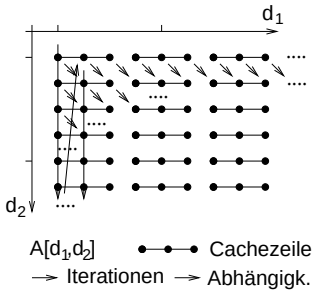
Vorgehen:
- Teile zweidimensionale Reihung in Rechtecke.
- Führe Algorithmus auf diesen Rechtecken aus.
- Äußere Schleifen iterieren von Rechteck zu Rechteck.
- Allgemein für Reihungen mit beliebig vielen Dimensionen möglich.
- Läßt sich als Kombination von Streifenschneiden und Vertauschen darstellen.

Vorteile:
- Vermeidet Kapazitätsfehlzugriffe
  - Wenn es eine Schleife gibt, von der die Reihungszugriffe unabhängig sind (Index nie verwendet).
  - Wenn in jeder Iteration benachbarte Reihungselemente aus beiden Reihungsstufen verwendet werden.
- Vermeidet evtl. Seitenkonflikte

Nachteile:
- Kann extreme Konfliktfehlzugriffe auslösen: Kachelgröße vorsichtig wählen.

Kacheln: Beispiel

```
for d1 := 0 to n do
  for d2 := 0 to n do
    A[d1,d2]:=A[d1+1,d2+1]+1;
```



Optimierung für *Cache* mit 4 Zeilen à 3 Reihungselementen. Sei n durch 3 teilbar, n > 12.

- Da nicht mehrere Spalten in den *Cache* passen, Layout ungünstig
- Vertauschen sei nicht möglich / unnütz wegen weiterem Code (z.B. Matrixmultiplikation).
- Bilde 3x3 Kacheln
- (Bild stellt Feld dar.)

Kacheln: Beispiel

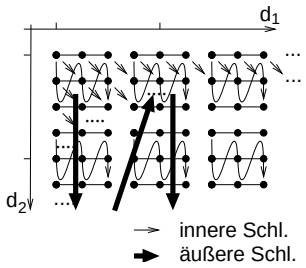
```
for TI1 := 0 to n step 3 do
  for i1 := TI1 to TI1 + 2 do
    for TI2 := 0 to n step 3 do
      for i2 := TI2 to TI2 + 2 do
        A[i1,i2]:=A[i1+1,i2+1]+1;
```

Streifenschneiden für beide Schleifen

```
for TI1 := 0 to n step 3 do
  for TI2 := 0 to n step 3 do
    for i1 := TI1 to TI1 + 2 do
      for i2 := TI2 to TI2 + 2 do
        A[i1,i2]:=A[i1+1,i2+1]+1;
```

Schleifen vertauschen

Kacheln: Beispiel



- Kachel und Randgebiete passen in *Cache*
- schlechte Wiederverwendung der Randgebiete: äußere Schleifen noch vertauschen

```
for TI2 := 0 to n step 3 do
  for TI1 := 0 to n step 3 do
    for i1 := TI1 to TI1 + 2 do
      for i2 := TI2 to TI2 + 2 do
        A[i1,i2]:=A[i1+1,i2+1]+1;
```

- Äußere Schleifen vertauscht

Kacheln: Formal

l_k Kantenlänge der Kachel in dieser Stufe.
Annahme: $(n_{max} - n_0) \mod l_k = 0$
Ersetze Schleife

```
for i := n0 to nmax do
```

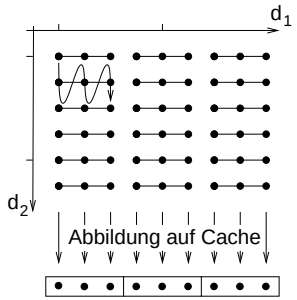
durch

```
for TI := 1 to (nmax − n0) div lk do
  for i := n0 + (TI − 1) ∗ lk to n0 + (TI − 1) ∗ lk + lk − 1 do
```


Kacheln: Konflikte

- Ist die Zeilenlänge (bei zeilenweiser Speicherung) ein Vielfaches der *Cache*größe, werden in den Spalten benachbarte Elemente auf *Cache*-Zeilen der gleichen Assoziativitätsgruppe abgebildet
- Ist dann eine Kachel breiter als die Assoziativität, kann trotz ausreichender *Cache*größe nicht die ganze Kachel im *Cache* gehalten werden.

Kacheln: Konflikte – Beispiel



- Direkt abbildender *Cache* mit 3 Zeilen à 3 Reihungselementen
- Speicherlayout entlang 2. Stufe
- Eine Zeile der Reihung paßt genau in den *Cache*
- Kacheln bewirken Neuladen der *Cache*zeile bei jedem Zugriff
- Bei anderer Reihungsgröße funktioniert diese Kachelung!
- Dieses Bild zeigt eine Reihung, keinen Iterationsraum!

Schleifen verschmelzen

("loop fusion, loop jamming")

Vorgehen:
- Füge zwei aufeinander folgende Schleifen zu einer zusammen

Vorteile:
- Erhöhte Lokalität: Operieren beide Schleifenrümpfe auf den gleichen Daten, müssen diese nur ein mal geladen werden.
- Größerer Schleifenrumpf: Vorteil für viele andere Optimierungen
- Reduziert Schleifenmehraufwand

Nachteile:
- Datenmenge kann *Cache* auch überlasten.
- Schleifenrumpf könnte nicht mehr in Befehlscache passen.

Schleifen verschmelzen: Beispiel

```
for i := 0 to n do
  A[i] := i;
  B[i] := C[i] - 1;
for i := 0 to n do
  A[i] := A[i] + B[i];
```

- Gleicher Indexbereich
- Keine problematischen Abhängigkeiten

```
for i := 0 to n do
  A[i] := i;
  B[i] := C[i] - 1;
  A[i] := A[i] + B[i];
```

- Schleifen verschmolzen

```
for i := 0 to n do
  A[i]:= i + C[i]-1;
```

- Geschaffenes Optimierungspotential genutzt.
- (B nicht weiter verwendet – tot)

Schleifen verschmelzen: Beispiel Cacheeffekte

```
for i := 0 to n do
  A[i] := 5 * i + B[i];
for i := 0 to n do
  A[i] := 1 / A[i];
```

⇒

```
for i := 0 to n do
  A[i] := 1 / (5*i+B[i]);
```

positiver *Cache*-Effekt:
- Gleiche Datenmenge wird nur einmal durchlaufen.
- Fehlzugriffe halbiert.

Schleifen verschmelzen: Beispiel Cacheeffekte

```
for i := 0 to n do
  A[i] := 5 * i;
for i := 0 to n do
  B[i] := 1 / C[i];
```

⇒

```
for i := 0 to n do
  A[i] := 5 * i;
  B[i] := 1 / C[i];
```

ungünstiger *Cache*effekt:
- Alle Reihungen werden nach wie vor einmal durchlaufen.
- Vereinigte Schleife kann Konflikte zwischen A und B oder C verursachen.
- Zugriff auf Hauptspeicher abwechselnd für A, B, C ineffizient
- Seitenfehler

Schleifen verschmelzen: Legalität

Legalität:

- Beide Schleifen müssen dieselben Indexbereiche haben.
  Dies kann erreicht werden durch:
  - Schälen
  - Bereichsteilen
- Es darf keine Programmstellen S_1 in der ersten Schleife und S_2 in der zweiten Schleife geben, für die nach Verschmelzung eine Abhängigkeit $S_1 \ \delta^{<} \ S_2$ bestehen würde.
- In der ersten Schleife dürfen keine Skalare definiert werden, die in der zweiten benutzt werden.

Legalität Verschmelzen: Beispiel

```
for i := 0 to n do
  A[i] := 5 * i;
for i := 0 to n do
  B[i] :=1 / A[i+1];
```

$\Rightarrow$

```
for i := 0 to n do
  A[i] := 5 * i;
  B[i] := 1 / A[i+1];
```

Verschmelzen illegal:

- Verschmolzene Schleife verwendet uninitialisiertes `A[i+1]`.

```
for i := 0 to n do
  X := A[i];
  B[i] := B[i] + X;
for i := 0 to n do
  C[i] := C[i] + X;
```

$\Rightarrow$

```
for i := 0 to n do
  X := A[i];
  B[i] := B[i] + X;
  C[i] := C[i] + X;
```

Verschmelzen illegal:

- Verschmolzene Schleife addiert zu `C[i]` `A[i]` statt `A[n]`.

Schleifen teilen

("loop splitting, loop distribution, loop fission")

Vorgehen:

- Teile eine Schleife auf in zwei einzelne Schleifen.
- Gegenteil von Verschmelzen.

Vorteile:

- In der Schleife verwendete Datenmenge reduzieren:
  vermeidet Konfliktfehlzugriffe.
- Parallelisierbare Teile eines Schleifenrumpfes herausziehen.
- Schafft evtl. perfekte Schleifenschachteln.

Nachteil:

- Schleifenrumpf wird kleiner
- erhöhte Lokalität: Cache evtl. nicht ganz genutzt

Schleifen teilen: Beispiel

```
for i := 0 to n do
  A[i] := A[i] + c;
  B[i+1] := B[i] + A[i];
```

$\Rightarrow$

```
for i := 0 to n do
  A[i] := A[i] + c;
for i := 0 to n do
  B[i+1] := B[i] + A[i];
```

- Schleifenkonstrukt replizieren.
- Anweisungen in Schleife auf neue Rümpfe aufteilen.
- Erste Schleife ist nun parallelisierbar.

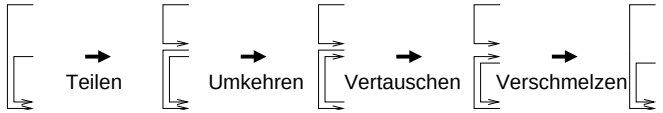
Teilen: Legalität

Legalität:
- Abhängigkeiten dürfen nicht verletzt werden

Vorgehen:
- Konstruiere Abhängigkeitsgraph der Schleife
  - Knoten: Alle Programmstellen S
  - Kanten: Abhängigkeiten
  - (Unterscheide von Abhängigkeitsgraph bei SSA bzw. Befehlsanordnung)
- Alle Programmstellen in starken Zusammenhangskomponenten müssen in einer Schleife bleiben
- Seien S_1 und S_2 Programmstellen mit $S_1 \rightarrow^* S_2$ im Abhängigkeitsgraph. Dann muß S_2 in der gleichen oder einer späteren der neuen Schleifen stehen als S_1.

Kombination von Optimierungen

- Für optimale Ergebnisse müssen alle präsentierten Techniken kombiniert werden.
- Es gibt verschiedene Ansätze, einige Techniken in einem gemeinsamen Rahmenwerk zu betrachten
- Als einziges formales Rahmenwerk haben sich unimodulare Transformationen durchgesetzt
- Es gibt keine gute Reihenfolge für die Optimierungen.



Literatur

Übersichtsartikel:
D. Bacon and S. Graham and O. Sharp, Compiler Transformations for High-Performance Computing, *ACM Computing Surveys*, 28(4), S. 345–420, Dez. 1994

Lineare Schleifenrestrukturierung:
Michael E. Wolf and Monica Lam, A Data Locality Optimizing Algorithm, *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, S. 30–44, Toronto, Canada, Jun. 1991

Kacheln:
S. Coleman and K. S. McKinley, Tile Size Selection Using Cache Organization and Data Layout, *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, S. 279–290, La Jolla, USA, Jun. 1995

Rahmenwerk für Teilen und Verschmelzen:
S. Carr and K. S. McKinley and C. Tseng, Compiler Optimizations for Improving Data Locality, *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, S. 252–262, San Jose, USA, Okt. 1994

Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
  - Abhängigkeitsanalyse
  - Schleifentransformationen
  - Schleifenrestrukturierung
  - Anordnungstransformationen
    - Grundsätzliche Überlegungen
    - Transposition
    - Kopieren
    - Auffüttern
    - Vereinigen
- Optimierungen für Dynamische Datenstrukturen
- Fehlzugriffe tolerieren

Anordnungstransformationen

Idee:
- Verändern der Anordnung der Daten im virtuellen Speicher.
- Dies ändert die vom Programm verwendeten Adressen
  $\Rightarrow$ *Cache* wird anders belegt.
- Algorithmus muß nicht verändert werden.

Vorteile:
- Leichter durchführbar: Datenabhängigkeiten egal.
- Besonders geeignet gegen Konfliktfehlzugriffe.
- Kann obligatorische Fehlzugriffe verhindern.
  (Wiederverwendung einer Adresse für eine andere Variable.)

Nachteile:
- Nicht anwendbar, wenn Anordnung durch Sprache vorgegeben.
- Datenanordnung muß bei Übersetzung bekannt sein.
- Kann Kapazitätsfehlzugriffe prinzipiell nicht verhindern.
- Aliasprobleme beachten.

Statische und dynamische Anordnungen

Anordnung kann statisch, beim Übersetzen, festgelegt werden:
- Veränderung der Deklaration des Datenobjekts
  (Reihungsdimensionen, Anordnung der Reihungen eines Objekts).
- Anordnung steht für den ganzen Programmlauf fest.
- Anordnung kann nicht für eine spezielle Schleife optimiert werden.

Anordnung kann dynamisch, zur Laufzeit, hergestellt werden:
- Umkopieren der Datenstruktur.
- Umkopieren ist teuer!
- Anordnung für eine Schleife möglich.
- Analysen nötig: Alle Zugriffe müssen angepasst werden.

Permutation

Vorgehen:
- Stufen einer Reihung vertauschen.
- Indizes in Reihungsreferenzen vertauschen.
- Analog zu Permutation für Schleifen.

Vorteile:
- Stufe, die entlang einer innersten Schleife läuft, als innerste Reihungsstufe (so dass geringste Schrittweite im Speicher):
  reduziert Zeilenkapazitätsfehlzugriffe.
- Bei Änderung der Deklaration keine zusätzliche Laufzeit.

Nachteile:
- Bei Änderung der Deklaration muß Anordnung für alle Schleifen passen.
- Sonst Kopierkosten.

Legalität:
- Immer, wenn von Sprache erlaubt.

Kopieren

Vorgehen:
- Kopiere einen Datensatz
  - der in den *Cache* paßt und
  - auf dessen Elemente mehrfach zugegriffen wird

  in ein fortlaufendes Stück Speicher
- Typische Anwendung: Kacheln kopieren

Vorteile:
- Garantiert keine Konfliktfehlzugriffe.

Nachteile:
- Kopieren vor und bei Änderung der Daten nach den eigentlichen Zugriffen

Legalität:
- Immer, wenn von Sprache erlaubt.

Auffüttern ("Padding")

Vorgehen:
- Füge Füllfelder zwischen eigentlichen Daten ein.
- Verschiebt Daten relativ zueinander.
- Statisch und dynamisch möglich ⇒ Vorteile und Nachteile

Typische Anwendung:
- zwischen verschiedenen Reihungen: kleine Reihungen einfügen
- zwischen Reihungsstufen: Dimension ändern

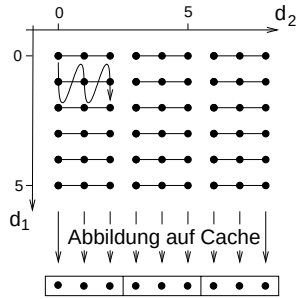
Vorteile:
- verringert Konfliktfehlzugriffe
  ⇒ Anwendung bei gekachelten Schleifen

Nachteile:
- Speicherverschnitt

Legalität:
- Immer, wenn von Sprache erlaubt.

Auffüttern: Beispiel

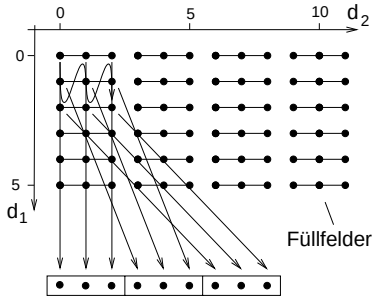
```
int A[6,9];
for TI1 := 0 to 8 step 3 do
  for TI2 := 0 to 5 step 3 do
    for i1 := TI1 to TI1 + 3 do
      for i2 := TI2 to TI2 + 3 do
        A[i2,i1]:= ...
```



- Direkt abbildender *Cache* mit 3 Zeilen à 3 Reihungselementen
- Speicheranordnung entlang 2. Dimension (0-8)
- Eine Zeile der Reihung paßt genau in *Cache*
- Alle drei virtuellen Zeilen einer Kachel werden auf die gleiche *Cache*zeile abgebildet.
- Dieses Bild zeigt eine Reihung, keinen Iterationsraum!

Auffüttern: Beispiel

```
int A[6,12];
for TI1 := 0 to 8 step 3 do
  for TI2 := 0 to 5 step 3 do
    for i1 := TI1 to TI1 + 3 do
      for i2 := TI2 to TI2 + 3 do
        A[i2,i1]:= ...
```



- Abbildung durch Füllfelder verändert.
- Kachel wird genau auf *Cache* abgebildet.

Vereinigen

Vorgehen:
- Füge mehrere Reihungen zu einer zusammen
- Wechsle dabei zwischen Elementen der Reihungen ab

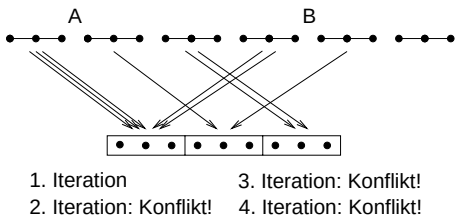
Vorteile:
- keine Konflikte, falls Reihungsreferenzen affin.
- Gleichmäßige Speicherauslastung, keine Spitzen:
  in jeder Iteration eine *Cache*zeile benötigt

Nachteile:
- Verschnitt, falls Reihungen verschieden groß
- Verschnitt, falls Referenzen nicht uniform.

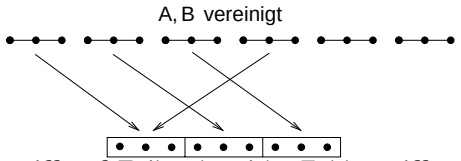
Legalität:
- Immer, wenn von Sprache erlaubt.

Vereinigen: Beispiel

```
int A[6], B[12];
for i := 0 to 5 do
  A[i] := B[2i] + B[2i+1]
```




```
int C[18];
for i := 0 to 6 do
  C[3i] := C[3i+1] + C[3i+2]
```



- Vorher: ungeordneter *Cache*zugriff auf Zeilen bewirkt Fehlzugriffe
- Nachher: Jede Iteration benötigt eine Zeile.

Literatur

Übersicht:
A. R. Lebeck and D. A. Wood, Cache Profiling and the SPEC Benchmarks: A Case Study, *IEEE Computer*, Okt. 1994

Kopieren (für gekachelte Schleifen):
O. Temam and E. Granston and W. Jalby, To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts, *Proceedings of Supercomputing '93*, S. 410–419, Portland, USA, Nov. 1993

Auffüttern:
Gabriel Rivera and Chau Wen Tseng, Data Transformations for Eliminating Conflict Misses, *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, Montreal, Canada, Jun. 1998.

Umfassender Ansatz: Daniela Genius, Handling Cross Interferences by Cyclic Cache Line Coloring, *The 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, Frankreich, Okt. 1998

Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
  - Abhängigkeitsanalyse
  - Schleifentransformationen
  - Schleifenrestrukturierung
  - Anordnungstransformationen
- Optimierungen für Dynamische Datenstrukturen
  - Anhäufung
  - Strukturteilen
  - Haldenorganisation
- Fehlzugriffe tolerieren

Optimierungen für Dynamische Datenstrukturen

Dynamische Datenstrukturen
- Listen, Bäume, Graphen; bestehend aus Knoten.
- Werden zur Laufzeit erzeugt (`malloc`, `new`).
- Struktur beim Übersetzen unbekannt.
- Struktur ändert sich zur Laufzeit.

$\Rightarrow$ schwerer zu optimieren als Reihungen in Schleifen.

Probleme bei Optimierung:
- geringe Lokalität (probleminhärent)
- Adressen der Knoten unbekannt
  $\Rightarrow$ Fehlzugriffe schwer vorhersagbar,
  Datenabhängigkeiten nicht bestimmbar (Aliasprobleme).
- ungenaue Heuristiken
- i.d.R. nicht automatisiert

Optimierungen für Dynamische Datenstrukturen

Keine Programmtransformationen:
Wegen der oben genannten Probleme gibt es keine automatischen Optimierungen von Schleifen/Rekursionen.
Es existieren cachesensitive Algorithmen $\Rightarrow$ Algorithmentechnik.

Anordnungstransformationen können angewendet werden:
dynamische Allokation lässt beliebige Platzierung eines Knotens zu.

Anhäufung

Vorgehen:
- Platziere gemeinsam verwendete Knoten in einer *Cache*zeile
- Heuristik: Knoten und seine Kinder werden gemeinsam verwendet
- Wähle anhand weiterer Heuristiken wichtige Kinder aus.

Vorteile:
- Vermeidet Fehlzugriffe
- Anhäufung kann bei Allokation oder Kopierphasen (Speicherbereinigung) hergestellt werden

Nachteile:
- Knoten oft größer als *Cache*zeile
- Erzeugt Speicherverschnitt

Strukturteilen

Vorgehen:
- Unterteile Knoten in häufig und selten genutzte Anteile
- Optimiere *Cache*verhalten des häufig genutzten Anteils
- Lagere selten genutzten Anteil aus.

Vorteile:
- Es passen mehr Knoten in den Cache
- Anhäufung effizienter

Nachteile:
- Zusätzlicher Zeiger benötigt
- Zusätzliche Indirektion und Fehlzugriff bei Zugriff auf selten benutzte Elemente

Legalität:
- Aliasprobleme?
- Wenn es die Sprache zulässt.

Strukturteilen: Beispiel

```
struct node {
  node *n1, *n2;
  data *d;
  int  key;
  long l; }

struct node {
  node *n1;
  int  key;
  rest *r; }
struct rest {
  node *n2;
  data *d;
  long l; }
```

- Häufig benötigt: `n1` und `key` (Doppelt verkettete Liste, Suche)
- Selten benötigte Reihungen ausgelagert in Knoten des Typs `rest`.
- Zugriffe `n.d` ersetzt durch `n.r->d` (für `node n;`).

Haldenorganisation

Vorgehen:
- Verwende mehrere Halden für ein Programm
- Platziere häufig verwendete Knoten in einer eigenen Halde
- Auswahl nach Datentyp

Vorteile:
- Häufig verwendeter Datensatz wird kleiner:
  - Kapazitätsfehlzugriffe
  - Seitenfehler
- Verwendeter Datensatz ist kompakter: Effekte wie bei Anhäufung möglich

Nachteile:
- Zugriff auf Knoten auf anderer Halde verursacht wahrscheinlich Seitenfehler
- Organisationsmehraufwand für die Halden

Haldenorganisation: Beispiel

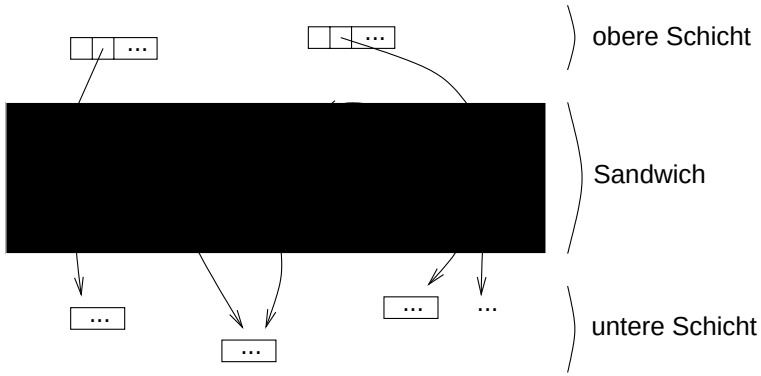
Welche Knoten sollen auf einer eigenen Halde abgelegt werden?

Knoten eines Sandwichtyps:
- Nur erreichbar von Knoten der oberen Schicht: Kopfknoten.
- Enthalten nur Referenzen auf ihresgleichen oder Knoten der unteren Schicht
- Knoten der unteren Schicht nur durch Sandwichknoten erreichbar.

```
class roots {                  • Obere Schicht
  sandwich[]root; }            • Wahrscheinlich nur selten verwendet
                                 (zum Betreten der Datenstruktur)
class sandwich {
  sandwich  son1;              • Inneres des Sandwich.
  sandwich  son2;              • Hier Navigation in Datenstruktur.
  leave     data; }            • In eigener Halde allozieren.

class leave {                  • Untere Schicht.
  int       ...; }             • Wahrscheinlich seltener Zugriff.
                               • Nicht bevorzugt behandeln.
```


Haldenorganisation: Beispiel



Literatur

Anhäufen und Strukturteilen:
Trishul M. Chilimbi and Mark D. Hill and James R. Larus, Cache-Conscious Structure Layout, *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, Mai 1999.
Gleiche Konferenz mit Bob Davidson: Cache-Conscious Structure Definition

Haldenorganisation:
Daniela Genius and Martin Trapp and Wolf Zimmermann, An Approach to improve Locality using Sandwich Types, *Proceedings of the 2nd Types in Compilation workshop*, S. 194–214, Kyoto, Japan, Mär. 1998.

Inhalt - Cache-Optimierungen

- *Caches* und deren Problematik
- Schleifenoptimierungen für Reihungen
  - Grundsätzliches
  - Abhängigkeitsanalyse
  - Schleifentransformationen
  - Schleifenrestrukturierung
  - Anordnungstransformationen
- Optimierungen für Dynamische Datenstrukturen
- Fehlzugriffe tolerieren
  - Grundsätzliches
  - Vorladen
  - Befehlsanordnung

Fehlzugriffe tolerieren

- Optimierungen können nicht alle Fehlzugriffe vermeiden
  - Obligatorische Fehlzugriffe
  - Optimierungen sind nicht optimal; Heuristiken
  - Abwägen: Fehlzugriffe $\Leftrightarrow$ Mehrkosten durch Optimierung

Idee:
Organisiere Programm so, dass Fehlzugriff Ausführung nicht verzögert: verstecke Latenzzeiten der Fehlzugriffe hinter Ausführzeiten anderer Befehle.

Nachteil:
- Teure *Cache*hardware umsonst.
- Hohe Auslastung auf Datenpfad *Cache* $\Leftrightarrow$ Speicher.
- Wenn Ladebefehl Treffer, dann war Optimierung umsonst.

Für die folgenden zwei Folien wird die Kenntnis von Anordnungsalgorithmen vorausgesetzt (siehe später).

Befehlsanordnung

1. Idee:
Führe Ladebefehl lange vor Verwendung des geladenen Datums aus.

Vorteil:
- Genug Zeit, um Datum aus Hauptspeicher zu laden.

Nachteile:
- „lange vor Verwendung“ wegen Abhängigkeitsgraph oft nicht möglich.
- Datum belegt Register für eine ebenso lange Zeit
  (weniger relevant bei “register renaming” im Prozessor).

Vorgehen:
- Modifiziere Algorithmen zur Befehlsanordnung
- Blockanordnung: Passe Heuristik zur Auswahl aus Bereitliste an.
- Software Fließband: Passe Heuristik zur Wahl der parallel anzuordnenden Iterationen an

Befehlsanordnung: Blockanordnung

(“list scheduling”)

Für die Priorität bei Blockanordnung werden i.d.R. Latenzen der Befehle verwendet.
- Für Ladebefehle wird i.d.R. die Trefferlatenz verwendet
- Zu viel Ladebefehle, um für alle Fehlzugrifflatenz zu verwenden
  $\Rightarrow$ Parallelität auf Befehlsebene (ILP) begrenzender Faktor
- Entscheide durch Heuristiken, für welche Ladebefehle Fehlzugriffslatenzen verwendet werden sollen.
- Verwende hierzu Informationen aus vorangegangenen *Cache*-Optimierungen

Vorsicht:
- Latenzen der Fehlzugriffe treten nicht immer auf (Treffer).
- Latenzen anderer Befehle (Div) treten immer auf.

$\Rightarrow$ Latenzen der Fehlzugriffe sind zweitrangig

Befehlsanordnung: Software Fließband

("modulo scheduling", "software pipelining")

Die Heuristik zur Festlegung der Anzahl parallel angeordneter Iterationen (II) verwendet i.d.R. den längsten Pfad durch den DAG des Schleifenrumpfes.

- Auch für diesen DAG werden i.d.R. Trefferlatenzen verwendet.
- Durch Fehlzugriffslatenzen verlängert sich der längste Pfad
- Dies erhöht II und damit den Registerdruck
- Wäge ab.

Vorteil (im Gegensatz zur Blockanordnung):

- Durch Erhöhen des II kann ausreichend ILP geschaffen werden.

Vorladen

("prefetching")

2. Idee: Vorladen
Hole rechtzeitig den benötigten Block in den *Cache*,
lade Datum kurzfristig.

Vorteile:

- Kein Register wird gebunden.
- Platzierung des Vorladebefehls unabhängig von Datenflußabhängigkeiten/Abhängigkeitsgraph.

Nachteile:

- Vorladen zu früh:
  - Noch benötigte Zeile kann verdrängt werden.
  - Vorgeladene Zeile kann vor Verwendung verdrängt werden.
- Vorladen zu spät: Zeile nicht rechtzeitig da.
- Wenn vor Datenabhängigkeit, dann Vorladen umsonst.
- Zusätzlicher Befehl: Programm wird größer.

Vorladen: Beispiele für Probleme

```
Prefetch A[i]
LOAD B[i]
...
LOAD A[i]
```

- Vorladen unnütz, wenn Konflikt zwischen `B[i]` und `A[i]`:
  Zugriff auf `B[i]` verdrängt vorgeladene Zeile.

```
for i := 1 to n do
  Prefetch A[i]
  LOAD A[i]
```

- Nicht genug Zeit zwischen `Prefetch` und `LOAD`
- Da `A[i]` entlang Speicherlayout, Vorladebefehle auch für Treffer.

Vorladen: Umsetzung

Umsetzung 1.Variante:
Verwende standard `LOAD`-Befehl, als Ziel ein fest verdrahtetes Register.

Vorteile:

- Auf jedem Prozessor möglich
- Standard Befehl – keine Erweiterung des Befehlssatzes

Nachteile:

- Adresse muß korrekt sein.
- Wird immer ausgeführt.
- Beansprucht auch Datenpfad *Cache* $\Rightarrow$ CPU
- Nur wenn *Cache* nicht blockierend, d.h. CPU arbeitet während `LOAD` weiter bis Datenabhängigkeit

Vorladen: Umsetzung

Umsetzung 2.Variante:
Spezieller Vorladebefehl im Prozessor:

- Prüft, ob Zeile in *Cache*.
- Wenn nicht, lädt Zeile in *Cache*
- Umgeht Speichereinheit – Entlastung
- Wird nicht ausgeführt, wenn
  - Adresse ungültig
  - Speichereinheit überlastet (Ladepuffer voll)
- muß nicht blockierend sein – sonst bringt es nichts.

(Nicht alle Vorladebefehle erfüllen all diese Kriterien.)

Bei blockierenden *Caches* (`LOAD` hält CPU an, wenn Fehlzugriff) sind Vorladebefehle unbedingt nötig.

Heute: die meisten *Caches* sind nicht blockierend.

Vorladen für Reihungen

Nutze Informationen aus vorangegangenen *Cache*-Optimierungen:

- Durch Schleifenpermutation Schleife, die entlang Speicheranordnung nach innen iteriert.
- Nun lässt sich von der Schrittweite und Datengröße ableiten, welche Zugriffe Fehlzugriffe sind, z.B. jede 4. Iteration
- Schleife vierfach ausrollen
- Einen Vorladebefehl einfügen
- Je nach Latenzzeit in vorheriger Iteration einfügen

Vorladen für Reihungen: Beispiel

```
for i1 := 0 to n do
  for i2 := 0 to n do
    A[i1,i2] := ...;
```

```
for i2 := 0 to n do
  for i1 := 0 to n step 4 do
    Prefetch A[i1 + 8,i2]
    A[i1,i2] := ...;
    A[i1 + 1,i2] := ...;
    A[i1 + 2,i2] := ...;
    A[i1 + 3,i2] := ...;
```

- $n >>$ *Cache*, spaltenweise Speicherung
- Jeder Zugriff ein Fehlzugriff
- Vertauscht, 4 mal ausgerollt
- Vorladen der übernächsten Cachezeile: genug Zeit zum Laden der Cachezeile.
- Alle Vorladebefehle sind Fehlzugriffe
- Alle Ladebefehle sind Treffer (außer `A[0,i2]`,`A[4,i2]`).
- Prozessor muss nie auf Fehlzugriff warten.

Vorladen für dynamische Datenstrukturen

Idee ("Greedy Prefetching"):
Lade Kinderknoten vor, bevor auf diese zugegriffen wird :

Vorteile:

- Der Zeiger auf den Kinderknoten ist im aktuellen Knoten.
- Dieser ist im *Cache*.

Nachteil:

- Berechnungen auf dem aktuellen Knoten oft zu kurz, um Vorladelatenz zu verdecken.

Abhilfe:

- Lade Enkel/Urenkel vor

Problem: Adresse zum Vorladen nicht bekannt

- Speichere Zeiger auf Enkel im Knoten ("history pointer").
  - Evtl. Speicherintensiv: welche Enkel sind wichtig?
- Linearisiere Datenstruktur: Zeiger auf Enkel berechenbar.

Literatur

Befehlsanordnen, Blockanordnung:
J. L. Lo and S. J. Eggers, Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism, *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, S. 151–162, San Diego, USA, Jun. 1995.

Befehlsanordnen, Software Fließband:
F. Jesus Sanchez and Antonio Gonzalez, Cache Sensitive Modulo Scheduling, *The 1997 International Conference on Parallel Architectures and Compilation Techniques*, S. 261–271, San Francisco, USA, Nov. 1997.

Vorladen für Reihungen in Schleifen:
Todd C. Mowry and Monica S. Lam and Anoop Gupta, Design and Evaluation of a Compiler Algorithm for Prefetching, *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, S. 62–73, Boston, USA, Okt. 1992.

Vorladen für Dynamische Datenstrukturen:
Chi-Kueng Luk and Todd C. Mowry, Compiler-Based Prefetching for Recursive Data Structures, *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, S. 222–233, Boston, USA, Okt. 1996.

Zusammenfassung Cache-Optimierung

- Optimierungen für Reihungen in Schleifen weit verbreitet; Stichworte „Fortran“, „high performance computing, HPF“, „Parallelisieren“.
- Schleifentransformationen:
  - verändern Iterationsraum nicht
  - ermöglichen und unterstützen andere Optimierungen (Standard- und *Cache*-Optimierungen).
- Schleifenrestrukturierung
  - verändern Iterationsraum
  - als *Cache*-Optimierung und zur Parallelisierung
  - Datenabhängigkeiten beachten!
  - Datenabhängigkeiten aufwendig zu finden.
- Anordnungstransformationen:
  - Ähnliche Effekte wie Schleifenrestrukturierung
  - Datenabhängigkeiten egal
  - Problem: Anordnungstransformationen für einzelne Schleifen
- Dynamische Datenstrukturen
  - Schwer analytisch zu beschreiben
  - aktuelles Forschungsgebiet
- Fehlzugriffe tolerieren
  - `LOAD` früh anordnen
  - Spezieller Vorladebefehl im Prozessor
  - Jeweils vorsichtig verwenden.