Seminar-Beiträge

Cache-Optimierungen

Holger Hopp[*], Daniela Genius[†], Michael Philippsen[‡](Hrsg.),

Philipp Bender, Bert Blaha, Sven-Olaf Braun,
Gerd Flaig, Thorwald Franke, Matthias Grutzeck,
Florian Liekweg, Felix Schröter

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
76128 Karlsruhe, Germany

29. Januar 1998

Interner Bericht 05/98

Zusammenfassung

Dieser Bericht enthält die Ausarbeitungen von Vorträgen aus einem Seminar gleichen Namens, das am 29. Januar 1998 am Institut für Programmstrukturen und Datenorganisation unter Leitung von Holger Hopp, Daniela Genius und Michael Philippsen stattfand.

Die Ausarbeitungen geben einen Überblick über Techniken und Modelle, um die Zwischenspeicher (*Caches*), die in praktisch allen heutigen Rechnern eingesetzt werden, effektiv einzusetzen. Dabei stehen insbesondere solche Techniken im Vordergrund, die von Übersetzern ausgeführt werden können wie diverse Schleifentransformationen, Speicherabbildungen, Vorzeitiges Laden von Daten. Außerdem werden Konsistenzmodelle für Parallelrechner untersucht.

[*]http://wwwipd.ira.uka.de/~hopp
[†]http://i44www.info.uni-karlsruhe.de/~genius
[‡]http://wwwipd.ira.uka.de/~phlipp

Inhaltsverzeichnis

1 Einführung

Die Rechnerentwicklung der letzten Jahren erbrachte sehr große Leistungssteigerungen von Mikroprozessoren, jedoch nur kleine Verbesserungen in der Speichergeschwindigkeit. Dies führt zu dem Problem, daß heutige Prozessoren zwar sehr schnell rechnen können, aber die notwendigen Daten nicht schnell genug vom Speicher geliefert werden können. Um diesen Mißstand zu beheben, werden in praktisch allen heutigen Rechnern schnellere Zwischenspeicher, sogenannte *Caches* eingesetzt. Ein Cache oder mehrere Caches werden dabei zwischen Hautspeicher und Prozessor plaziert, um für einen schnellen Datenaustausch zu sorgen.

Die Caches basieren auf Lokalität, d. h. daß Daten, die mehrmals kurz hintereinander gelesen werden, oder Daten, die sich „in der Nähe“ kurz vorher gelesener Daten befinden, sich (automatisch) im schnelleren Cache befinden und damit schnell vom Prozessor gelesen werden können. Caches haben zwar den Vorteil, daß sie schneller die Daten an den Prozessor liefern können, haben jedoch auch einige Nachteile. Dies sind im wesentlichen die relativ geringe Größe und die geringe Assoziativität.

Dies bereitet einige Schwierigkeiten: Bei größeren Datenmengen ist es von entscheidender Bedeutung möglichst viele Daten, die verwendet werden, aus dem schnellen Cache und nicht aus dem langsamen Speicher zu lesen. Um dies zu erreichen, ist es notwendig den Programmcode in dieser Hinsicht zu optimieren. Dies kann zum Beispiel durch Entwicklung „guter“ Algorithmen und durch „gute“ Programmierung geschehen. Das alleine ist aber nicht ausreichend, da die Speicherhierarchie und deren Auswirkungen auf jedem Rechner sehr verschieden sein können. Deshalb ist es naheliegend, solche Optimierungen in Übersetzern zu implementieren, und den Programmierer möglichst wenig mit maschinenspezifischer Feinoptimierung zu beschäftigen. Derartige Techniken sind Gegenstand dieser Sammlung von Seminarbeiträgen.

2 Überblick

Die ersten drei Kapitel beschäftigen sich mit Schleifentransformationen, d. h. Umordnungen von Schleifen, um eine bessere Ausnutzung des (der) Caches zu erreichen. Schleifen zu optimieren ist offensichtlich ein naheliegender Ansatzpunkt, da die meisten Programme einen großen Teil ihrer Laufzeit in Schleifen verbrauchen. In Kapitel I werden einfache, sog. *unimodulare* Schleifentransformationen betrachtet, wie Vertauschen, Richtungsumkehr und Neigen von Schleifen. Das Kapitel II beschäftigt sich mit Blockungen. Dies ist eine Technik, die es ermöglicht, größere mehrdimensionale Arrays in mehreren Blöcken zerteilt zu verarbeiten. Diese Blöcke sollen so gestaltet sein, daß innerhalb eines Blockes möglichst viele Daten aus dem Cache wiederverwendet werden. Das III. Kapitel beschäftigt sich mit Techniken, die mehrere Schleifen zu einer einzigen zusammenfassen oder umgekehrt. Es werden Rumpfteilen, Verschmelzen und Ausrollen sowie dafür noch notwendigen Hilfstechniken diskutiert.

Ergänzend, nicht alternativ, zu Schleifentransformationen kann man versuchen, durch Umordnen der Daten im Speicher bessere Lokalität zu erreichen. Dies ist das Thema von Kapitel IV. Bei der Matrixmultiplikation kann man z. B. durch Transposition der spaltenweise zugegriffenen Matrix Lokalität verbessern, kann diesen Effekt jedoch nicht durch Schleifentransformation erreichen. Zudem sollten Reihungen für Caches geringer Assoziativität so im Speicher abgelegt werden, daß das vorzeitige Verdrängen noch benötigter Cachezeileninhalte vermindert wird.

Problematisch ist, dass sich mit der Speicherabbildung auch die Indizierung im gleiche Maße kompliziert, d.h. die Laufzeit größer wird. Es gibt Fälle, in denen z.B. innerhalb einer Schleifenschachtelung nur auf einen Teil der Reihung wirklich zugegriffen wird. Diese gilt es, herauszufinden. Die in Kapitel V vorgestellte Datenflußanalyse für Reihungen ist hierbei ein grundlegendes Werkzeug.

Die Mehrzahl der bisher vorgestellten Optimierungen zielte darauf ab, durch Erhöhung der Lokalität die beschränkte Speicherbandbreite gut zu nutzen. Die Speicherlatenz (Wartezeit auf das Eintreffen von Daten aus dem Hauptspeicher) ist ein weiterer einschränkender Faktor für die Leistung der Speicherhierarchie. Komplementär zu Schleifen- und Speicherabbildungsoptimierungen wird in Kapitel VI der Ansatz des Versteckens von Latenzzeiten durch sog. Vorladen, engl. *prefetching*, verfolgt. Daten können, vorausgesetzt entsprechende Hardwaremöglichkeiten, vorsorglich in den Cache geladen werden.

Parallelrechner werfen im Zusammenhang mit Caches und der Speicherhierarchie besondere Probleme

auf, die dadurch verursacht werden, daß jeder Prozessor bei diesen Systemen zumeist seinen eigenen Cache hat und bei Systemen mit verteiltem Speicher eine zusätzliche Ebene in der Speicherhierarchie hinzutritt. Greifen verschiedene Prozessoren mit zeitlicher Lokalität auf gemeinsame Daten zu, werden diese in verschiedenen Caches (oder in den lokalen Speichern, die jedem der Prozessoren zugeordnet sind) repliziert gespeichert. Wie üblich, verursacht Replikation Konsistenzprobleme aller Art.

In Kapitel VII werden unterschiedliche Konsistenzmodelle untersucht und ihre Beziehungen untereinander herausgearbeitet. Kapitel VIII stellt mit Treadmarks ein System vor, bei dem durch Einsatz einer konsistenzgarantierenden Software-Schicht, ein Parallelrechner mit verteiltem Speicher so benutzt werden kann, als handele es sich um ein System mit einem gemeinsamen Speicher.

Seminar - Ausarbeitung I

Bert Blaha

Unimodulare Schleifentransformationen

Zusammenfassung

Obwohl die Benutzung eines Caches für den Programmierer transparent erscheint, hat er Einfluß darauf, wie gut die Datenlokalität eines Programmstückes tatsächlich ist. Die Aufgabe, Schleifenschachtelungen entsprechend der wenigsten Hauptspeicherzugriffe anzuordnen ist zeitraubend und fehleranfällig. Es liegt daher nahe, solche Optimierungen, auch im Hinblick auf Portabilität dem Compiler zu überlassen.

Ziel dieser Ausarbeitung ist es zunächst, einfache Schleifentransformationen wie Vertauschen, Richtungsumkehr und Neigen vorzustellen, sowie der Frage nachzugehen, wann diese angewendet werden dürfen ohne die Semantik des Programmes zu verändern. Im zweiten Teil werden Mechanismen vorgestellt, die eine mathematische Darstellung von Transformationen erlauben und dadurch die algorithmische Anwendung in restrukturierenden Compilern ermöglichen.

1 Einleitung

Bei den folgenden Beispielen zu verschachtelten Schleifen spielen Arrayzugriffe eine große Rolle. Zum Verständnis, welche Rolle der Cache dabei spielt, ist wichtig zu wissen, wie Arrays in verschiedenen Sprachen im Hauptspeicher abgelegt werden.

Bei eindimensionalen Arrays liegt das erste Element, also dasjenige mit dem kleinsten Index an der Startadresse des Arrays und alle nachfolgenden Elemente liegen direkt dahinter:

A[1]	A[2]	A[3]

Interessanter ist der Fall bei zwei- oder mehrdimensionalen Arrays. Hier hängt es von der Programmiersprache ab, ob die Arrays spalten- oder zeilenweise durchlaufen werden. Fortran beispielsweise legt mehrdimensionale Arrays von der hintersten zur vordersten Dimension ab:

A[1,1,1]	A[2,1,1]	A[1,2,1]	A[2,2,1]	A[1,1,2]	A[2,1,2]	A[1,2,2]	A[2,2,2]

In C ist die Reihenfolge der Dimensionen gerade umgekehrt:

A[1,1,1]	A[1,1,2]	A[1,2,1]	A[1,2,2]	A[2,1,1]	A[2,1,2]	A[2,2,1]	A[2,2,2]

O.b.d.A. wird im folgenden die Speicheranordnung von Fortran angenommen.
Die betrachteten Schleifenschachtelungen werden alle als perfekt angenommen, d.h. sämtliche Anweisungen befinden sich innerhalb des innersten Schleifenrumpfes.

2 Schleifenvertauschung

Gegeben sei folgendes Codestück:

```
for I := 1 to M do
  for J := 1 to N do
    A[I,J] := A[I,J] * 2;
  end
end
```

Das Array A enthalte Werte der Größe ein Wort. Die Schleifengrenzen N und M seien hierbei so groß gewählt, daß der Cache nicht in der Lage ist, das komplette Array A zu halten (z.B. N=M=10000). Beim Durchlaufen der Schleifen werden nacheinander folgende Speicherstellen bearbeitet:

A[1,1] A[1,2] ... A[1,N-1] A[1,N] A[2,1] A[2,2] A[2,N] A[3,1] ... A[M,N-1] A[M,N]

Angenommen A[1,1] liegt so im Hauptspeicher, daß es zu Beginn einer Cachezeile zu liegen käme. Bei einer Cachezeilengröße von 16 Wort werden mit dem ersten Zugriff auf A[1,1] zusätzlich noch die Elemente A[1,2] bis A[1,16] in den Cache gelegt (wenn A[1,1] nicht an den Beginn einer Cachezeile positioniert wird, ist diese Zahl entsprechend kleiner). Für jedes Element, das im Hauptspeicher weiter als 16 Byte entfernt liegt , muß also eine neue Cachezeile gefüllt werden. Da M in diesem Beispiel größer als die Kapazität des Caches in Cachezeilen und N größer als 16 ist, kommt es beim Durchlaufen der I-Schleife bei jedem Zugriff auf A[I,J] zu einem Cache-Miss.
Zum Vergleich dazu ist hier obiges Codestück noch einmal, jetzt allerdings mit vertauschten Schleifen:

```
for J := 1 to N do
  for I := 1 to M do
    A[I,J] := A[I,J] * 2;
  end
end
```

Da in jeder Iteration der innersten Schleife nur ein einziges Element bearbeitet wird, ändert die Vertauschung der Schleifen zwar die Reihenfolge der Bearbeitung der Arrayelemente, nicht jedoch die Semantik des Codes. Das muß nicht immer so sein.
Die Elemente von A werden jetzt in folgender Reihenfolge bearbeitet:

A[1,1] A[2,1] ... A[M-1,1] A[M,1] A[1,2] A[2,2] ... A[M,2] A[1,3] ... A[M-1,N] A[M,N]

Analog zu oben wird mit einem Zugriff auf das Array A gleich die ganze zugehörige Cachezeile gefüllt. Der Unterschied zu der Codeversion oben liegt aber darin, daß die Elemente jetzt in der Reihenfolge bearbeitet werden, in der sie auch im Hauptspeicher aufeinanderfolgen. Somit wird idealerweise jedes zusätzlich in die Cachezeile gelesene Element auch von dort statt aus dem langsamen Hauptspeicher geholt. Liegt A[1,1] wieder am Anfang einer Cachezeile so kommen auf ein Cache-Miss (zum Füllen einer neuen Cachezeile) 15 Cache-Hits. Die Zahl der Cache-Misses sinkt somit auf das Minimum von (N*M)/16.

3 Datenabhängigkeit

3.1 Arten von Datenabhängigkeiten

Leider kann man in der Regel verschachtelte Schleifen nicht nach Bedarf umstellen, ohne daß man die Semantik des Programmes und damit seine Richtigkeit verändert. Die Notwendigkeit, daß ein Befehl zeitlich nach einem anderen ausgeführt werden muß wird Datenabhängigkeit genannt. Folgende verschiedene Datenabhängikeiten können auftreten:

Fluß-Abhängigkeit	δ^f	Speicherstelle wird geschrieben und später gelesen.
Output-Abhängigkeit	δ^o	Speicherstelle wird geschrieben und später erneut geschrieben.
Anti-Abhängigkeit	δ^a	Speicherstelle wird gelesen und später geschrieben.

```
(1) A := f(J);
(2) C := A;
```

Es besteht eine Flußabhängigkeit zwischen Anweisung (1) und Anweisung (2), kurz (1) δ^f (2).

Für Schleifentransformationen ist wichtig, ob die Abhängigkeiten durch die Existenz des Schleifenkonstruktes bestehen, oder ob sie auch dann bleiben, wenn man die Schleife entfernen würde. Man unterscheidet:

3.2 Schleifenunabhängige Datenabhängigkeit

Diese Abhängigkeit besteht innerhalb einer einzelnen Iteration und ist nicht vom Schleifenkonstrukt abhängig:

```
(1) for J := 1 to M do
(2)   A[J] := f(J);
(3)   ...
(4)   C := A[J];
(5) end
```

Hier besteht eine Flußabhängigkeit zwischen Zeile (2) und (4), (2) δ^f (4).

Da schleifenunabhängige Datenabhängigkeiten von Schleifentransformationen nicht beeinflußt werden, müssen sie bei der Frage nach der Legalität von Schleifentransformationen nicht weiter betrachtet werden.

3.3 Schleifengetragene Abhängigkeiten

Hier besteht die Datenabhängigkeit nicht innerhalb einer Iteration, sondern vielmehr von einer Iteration zu einer der nachfolgenden (das muß nicht die unmittelbar nächste sein).

```
for I := 1 to 5 do
  for J := 1 to 5 do
    A[I,J] := f(I,J);
    ...
    C := A[I-1,J+1];
  end
end
```

In Abhängigkeit von I und J werden folgende Elemente aus A bearbeitet:

I	J		
2	1	A[2,1] wird geschrieben	A[1,2] wird gelesen
2	2	A[2,2] wird geschrieben	A[1,3] wird gelesen
2	3	A[2,3] wird geschrieben	A[1,4] wird gelesen
...	...	...	...
3	1	A[3,1] wird geschrieben	A[2,2] wird gelesen
3	2	A[3,2] wird geschrieben	A[2,3] wird gelesen
3	3	A[3,3] wird geschrieben	A[2,4] wird gelesen

Man kann erkennen, daß das Element A[x,y], welches bei I = x und J = y geschrieben wurde, in der Iteration bei I = x+1 und J = y-1 wieder gelesen wird. Es ist also wichtig, daß bei einer eventuellen Transformation diese Reihenfolge eingehalten wird; man sagt es besteht eine Datenabhängigkeit zwischen diesen beiden Iterationen.

Um die Notation zu erleichtern, benutzt man Iterationsvektoren. Dieser Vektor enthält einfach die Laufvariablen von der äußersten zur innersten perfekt geschachtelten Schleife:

```
for i1 := ...
  for i2 := ...
    ...
      for in := ...
```

Der zugehörige Iterationsvektor hat die Form: i=(i_1, i_2, ... , i_n). Nach dieser Notation wird im obigen Beispiel A[2,2] in Iteration (2,2) geschrieben und in Iteration (3,1) gelesen. Die zwischen diesen Iterationen bestehende Datenabhängigkeit wird als (2,2) δ^f (3,1) geschrieben.

3.4 Abhängigkeitsdistanz

Wenn i_{source} und i_{target} die Iterationsvektoren zweier abhängiger Iterationen sind, bezeichnet man d mit

i_{source} + d = i_{target} als die Abhängigkeitsdistanz. Da $i_{source} < i_{target}$ ist, gilt d > 0. Für das Beispiel

oben bedeutet das konkret:

i_{source}	i_{target}	d
(2,2)	(3,1)	(1,-1)
(3,2)	(4,1)	(1,-1)
(4,4)	(5,3)	(1,-1)
...	...	...

Bei vorwärts laufenden Schleifen haben Abhängigkeitsdistanzen im allgemeinen folgendes Aussehen: $(d_1, d_2, d_3, \ldots, d_p, \ldots, d_n)$

wobei $d_m = 0$ für m < p
$d_p > 0$ weil eine Abhängigkeit nur zu nachfolgenden Iterationen bestehen kann.
d_m beliebig für m > p.

In diesem Fall sagt man: Schleife p trägt die Abhängigkeit.

Folgende Überlegung zeigt auf, warum ein gültiger Distanzvektor nur so aussehen kann: Wie erwähnt beschränken wir uns auf vorwärtslaufende Schleifen. *Daß* eine Abhängigkeit zwischen zwei Iterationen i_{source} und i_{target} besteht, bedeutete doch, daß i_{source} zeitlich *nach* i_{target} ausgeführt werden muß. Innerhalb einer Schleifenschachtelung wird die Laufvariable i_m einer Schleife m nur dann erniedrigt, wenn i_m die obere Schleifengrenze erreicht hat und auf die untere Schleifengrenze gesetzt wird. Dabei wird konsequenterweise die Laufvariable der davor liegenden Schleife m-1 um eins erhöht. Da die äußerste Schleife 1 aber keine weitere Schleife um sich hat, wird die Verschachtelung verlassen, sobald die letzte Iteration der äußersten Schleife abgearbeitet ist. Somit wird ihr Schleifenindex im Laufe der Abarbeitung immer nur inkrementiert.

Es ist daher nicht möglich, daß eine Iteration I_1 eine Abhängigkeit zu einer weiteren I_2 hat, deren erster Vektoreintrag kleiner ist als der von I_1. Dann würde I_1 nach I_2 ausgeführt, und das verletzt die Abhängigkeitsbedingung.

Sollte es innerhalb verschachtelter Schleifen lediglich schleifenunabhängige Datenabhängigkeiten geben, so ist der Distanzvektor der Nullvektor. Falls sich die Distanz nicht konstant angeben läßt, beispielsweise bei nicht linearen Indexausdrücken, vereinfacht sich der Distanzvektor zum Richtungsvektor, in dem statt Zahlen nur noch Relationen angegeben werden um festzuhalten, in welcher Richtung die Abhängigkeit in der jeweiligen Dimension liegt: (<, >, =) statt (1, -2, 0).

4 Legalität von Schleifentransformationen

4.1 Schleifenvertauschung

Es ist einsichtig, daß sich der Distanzvektor verändern kann, wenn man Schleifentransformationen durchführt. Wie beschrieben muß stets gelten, daß er lexikographisch positiv ist, d.h. das erste Vektorelement, das nicht Null ist, darf nicht negativ sein. Eine Schleifentransformation ist genau dann legal durchführbar, wenn sie diese Eigenschaft erhält.

Wenn eine Schleifenschachtelung aus n Schleifen gegeben ist, so haben die vorkommenden Datenabhängigkeiten n-dimensionale Distanzvektoren d = $(d_1, \ldots, d_n)$. Vertauscht man die Schleifen i und j, so werden auch die zu diesen Schleifen gehörenden Einträge in d vertauscht. Aus

$(d_1, \ldots, d_i, d_j, \ldots, d_n)$ wird somit

$(d_1, \ldots, d_j, d_i, \ldots, d_n)$.

Betrachten wir die folgende Schleifenvertauschung:

```
for J := 2 to M do
  for I := 1 to N do
    A[I,J] := A[I,J-1] +  B[I,J];
  end
end
```

Es existiert genau eine Datenabhängigkeit, die von der äußeren Schleife getragen wird; der zugehörige Distanzvektor lautet (1, 0). Vertauscht man die beiden Schleifen, so lautet der Distanzvektor (0, 1) und auch dieser ist lexikographisch positiv. Damit ist das Vertauschen der Schleifen legal.
Natürlich gibt es auch Fälle in denen das Vertauschen nicht erlaubt ist:

```
for H := 1 to N do
  for I := 2 to N do
    for J := 1 to N-1 do
      A[H,I,J] := A[H,I-1,J+1];
    end
  end
end
```

Hier lautet der Distanzvektor (0, 1, -1). Würde man die Schleifen vertauschen, würde er (0, -1,1) lauten. Das ist aber nach obiger Definition nicht mehr lexikographisch positiv, d.h. es werden Abhängigkeiten verändert und die Vertauschung ist nicht zulässig.

4.2 Richtungsumkehr

Eine weitere elementare Transformation ist die Richtungsumkehr. Darunter versteht man die Umkehrung der Laufrichtung einer Schleife. Da eine Cachezeile stets mit den umgebenden Hauptspeicherzellen gefüllt wird, kann durch Richtungsumkehr alleine die Laufzeit der Schleife nicht verkürzt werden. Die Transformation kann aber trotzdem sinnvoll sein, um andere Optimierungen, wie Vertauschen oder Neigen zu ermöglichen.
Richtungsumkehr der Schleife p verändert Distanzvektoren derart, daß der zu p gehörende Eintrag mit -1 multipliziert wird. Ein Distanzvektor der Form

$(d_1, \ldots, d_p, \ldots, d_n)$ wird somit zu
$(d_1, \ldots, -d_p, \ldots, d_n)$.

Folgende Schleifen dürfen in der vorliegenden Form nicht vertauscht werden:

```
for I := 2 to N do
  for J := 1 to N do
    A[I,J] := A[I-1,J+1] + 42;
  end
end
```

Der Distanzvektor lautet (1, -1) und nach dem Vertauschen der Schleifen würde er zu (-1, 1).
Läuft die innerste Schleife rückwärts entsteht folgendes Codestück:

```
for I := 2 to N do
  for J := N to 1 step -1 do
    A[I,J] := A[I-1,J+1] + 42;
  end
end
```

Der zur inneren Schleife gehörende Eintrag im Distanzvektor wird negiert, so daß dieser jetzt (1, 1) lautet. Nun darf man die Schleifen vertauschen, da der Distanzvektor auch danach noch lexikographisch positiv ist. Die Richtungsumkehr trägt also insofern zur Laufzeitverringerung eines Programmes bei, als die nachfolgend möglich gewordene Vertauschung die Lokalität erhöhen kann.

4.3 Schleifenneigen

Sollte Richtungsumkehr nicht zum Erfolg führen, bietet sich noch das Schleifenneigen als Alternative an. Darunter versteht man eine Indexverschiebung der inneren Schleife derart, daß der Index der äußeren Schleife mit einem Neigungsfaktor f multipliziert und anschließend auf die Schleifengrenzen der inneren Schleife addiert wird. Innerhalb der inneren Schleife muß dieser Summand bei einer Nutzung des inneren Schleifenindex wieder abgezogen werden. Ein Beispiel zur Verdeutlichung:

```
for I := 1 to N do
  for J := LO to HI do
    A[I,J] := A[I,J] + 1;
  end
end
```

wird nach Schleifenneigen mit Neigungsfaktor f zu:

```
for I := 1 to N do
  for J := LO + f*I to HI + f*I do
    A[I, J - f*I] := A[I, J - f*I] + 1;
  end
end
```

Schleifenneigen mit Faktor f verändert Distanzvektoren wie folgt: Aus

(d_1, d_2) wird
$(d_1, d_2 + \text{f*}d_1)$.

Wie man sehen kann, läßt diese Transformation den Distanzvektor stets lexikographisch positiv, ist also immer legal anwendbar. Entweder d_1 ist größer Null, dann ist $d_2 + \text{f*}d_1$ beliebig, oder d_1 ist Null, dann muß d_2 größer Null gewesen sein und ist es auch danach, da $d_2 + \text{f*}0 = d_2$ ist.

Auch Schleifenneigen bringt alleine nicht unbedingt eine Verkürzung der Laufzeit, sondern ist vielmehr eine Transformation, die weitere Optimierungen ermöglicht.

5 Darstellung von Transformationen mit Hilfe von Matrizen

Allen vorgestellten Transformationen war gemeinsam, daß sie die Abhängigkeitsvektoren der Zugriffe manipuliert haben. Haben die Vektoren etwa die Form

$$d = \ d_1, d_2, \ldots, d_n \ , d_n \in \mathbf{Z},$$

so lassen sich Vertauschung, Neigen und Richtungsumkehr als lineare Abbildungen auffassen, die auf den Abhängigkeitsvektoren operieren. Die zugehörigen Abbildungsmatrizen U sind von der Form $n \times n$ und haben folgende Eigenschaften :

- Da die Elemente von Abhängigkeitsvektoren natürlich ganzzahlig bleiben müssen, sind auch die Elemente von U ganzzahlig.
- Jede Transformation ist eindeutig umkehrbar, also besitzt U ein Inverses U^{-1}, das ebenfalls wieder ganzzahlig ist.
- Für die Determinante gilt: $|U| = 1$.

Diese Matrizen werden *unimodulare* Matrizen genannt.

5.1 Legalität bei unimodularen Transformationen

Wenn in einem Programmstück eine Abhängigkeit zwischen Iteration i und Iteration j bestanden hat, muß das natürlich auch nach der Transformation so sein:

$$i \;\delta\; j \quad \Rightarrow \quad i < j \quad \Rightarrow \quad iU \;\delta\; jU$$

Da es sich bei U um eine lineare Abbildung handelt, folgt für die Distanzvektoren:

$$Ui < Uj \quad \Rightarrow \quad 0 < jU - iU \quad \Rightarrow \quad 0 < (j-i)U \quad \Rightarrow \quad dU > 0$$

Eine unimodulare Transformation ist also genau dann legal durchführbar, wenn

$$dU > 0$$

gilt, d.h. der neue Abhängigkeitsvektor lexikographisch positiv bleibt. Für die vorgestellten Transformationen sehen die Abbildungsmatrizen folgendermaßen aus:

5.2 Vertauschung

U entsteht aus der Einheitsmatrix I, indem die Zeilen der zugehörigen Schleifen vertauscht werden:

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

Eine Vertauschung wird durch dieselbe Vertauschung wieder rückgängig gemacht, daher ist

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

5.3 Richtungsumkehr

U entsteht hier aus der Einheitsmatrix I, indem die zur umzukehrenden Schleife gehörende Zeile mit -1 multipliziert wird:

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

Auch hier kommt man durch erneutes Umkehren wieder zur Ausgangsmatrix:

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

5.4 Schleifenneigen

Schleifenneigen mit Faktor f bedeutete die Addition des f-fachen der Laufvariable der darüberliegenden Schleife zur eigenen im Schleifenkopf und die Subtraktion bei jeder Benutzung im Schleifenrumpf. U ist also:

$$U_f = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & f & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

U_f^{-1} hat die Form:

$$U_f^{-1} = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & -f & & & \\ & & & 1 & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix}$$

Eine Matrix für mehrere Transformationen, die hintereinander auszuführen sind, läßt sich einfach durch aufmultiplizieren konstruieren. Sollen beispielsweise zwei Schleifen vertauscht werden, von denen die eine wegen bestehender Datenabhängigkeiten zuvor noch umgekehrt werden muß, so lautet die passende Transformationsmatrix:

$$U_{Umkehr} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, U_{Vertauschen} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, U = U_{Vertauschen} U_{Umkehr} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

5.5 Anwenden einer Transformationsmatrix auf ein Codestück

In einem abschließenden Beispiel soll gezeigt werden, wie eine konstruierte unimodulare Transformationsmatrix auf ein Codestück angewendet wird, um die gewünschten Datenabhängigkeiten zu erhalten. Mit dieser Matrix

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

soll folgendes Codestück transformiert werden:

```
for I := 1 to N do
  for J := 1 to I+1 do
    for K := J to M do
      A[I,J,K] = B[I,J] + C[J,K]
    end
  end
end
```

Eine Iteration i wird durch die Transformation U auf $j = Ui$ abgebildet. Da U regulär ist, gibt es U^{-1}, so daß $U^{-1}j = i$ gilt. Wenn die ursprünglichen Schleifengrenzen als $Ai \leq c$ dargestellt sind, so bekommt man die transformierten Schleifengrenzen durch $AU^{-1}j \leq c$. Eine anschließend durch geführte Fourier-Motzkin-Elimination bestimmt konkret die Grenzen jeder einzelnen Schleife. Das Verfahren von Fourier-Motzkin, sowie dieses Beispiel finden sich in [1].

Der transformierte Indexbereich lautet:

$$j = \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} = U \begin{pmatrix} I \\ J \\ K \end{pmatrix} = \begin{pmatrix} J+K \\ I+J \\ I \end{pmatrix}$$

Über das Inverse von U erhält man die ursprünglichen Schleifengrenzen durch die transformierten ausgedrückt:

$$\begin{pmatrix} I \\ J \\ K \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} = \begin{pmatrix} j_3 \\ j_2 - j_3 \\ j_1 - j_2 + j_3 \end{pmatrix}$$

Die ursprünglichen Schleifengrenzen lassen sich folgendermaßen darstellen:

$$\begin{pmatrix} -1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & -1 & 0 \\ -1 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} I \\ J \\ K \end{pmatrix} \leq \begin{pmatrix} -1 \\ N \\ -1 \\ 1 \\ 0 \\ M \end{pmatrix}$$

Die neuen Schleifengrenzen ergeben sich aus $AU^{-1}j \leq c$:

$$\begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 1 \\ 0 & -1 & 1 \\ 0 & 1 & -2 \\ -1 & 2 & -2 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix} \leq \begin{pmatrix} -1 \\ N \\ -1 \\ 1 \\ 0 \\ M \end{pmatrix}$$

Die Fourier-Motzkin-Elimination ermittelt aus diesen Ungleichungen die neuen Schleifengrenzen, so daß der transformierte Code jetzt folgendes Aussehen hat:

```
for j1 := 2 to M+N+1 do
  for j2 := max(2,2*j1-2*M-1,j1-M+1) to min(2*N+1,N+floor(j1/2)) do
    for j3 := max(1,ceil((j2-1)/2),ceil((2*j2-j1)/2)) to min(N,j2-1,M-j1 + j2) do
      I := j3
      J := j2 - j3
      K := j1 - j2 + j3
      A[I,J,K] := B[I,J] + C[J,K]
    end
  end
end
```

6 Algorithmus von McKinley et al.

Der von McKinley et al. [2] vorgestellte Algorithmus bietet einen mathematischen Weg, aus allen möglichen Permutationen einer Schleifenschachtelung diejenige auszuwählen, die am meisten Lokalität erreicht und somit den Cache am besten ausnutzt. Er gliedert sich in drei Teile:

6.1 Einteilen in Zugriffsgruppen

Um den Grad der Wiederverwendung bei Speicherzugriffen angeben zu können, werden zunächst Zugriffe auf Arrayelemente bezüglich einer bestimmten Schleife in Zugriffsgruppen eingeteilt. Zwei Zugriffe werden genau dann ein und derselben Gruppe zugeteilt, wenn sie während der Programmabarbeitung dieselbe Cachezeile benutzen. Für die Praxis läßt sich dieser Sachverhalt folgendermaßen ausdrücken:

Zwei Zugriffe Z_1 und Z_2 gehören dann zur selben Zugriffsgruppe in Bezug auf Schleife l wenn es eine Datenabhängigkeit zwischen Z_1 und Z_2 gibt, die entweder

1. (a) schleifenunabhängig ist, oder
   (b) für den Distanzvektor d = $(d_1, \ldots, d_l, \ldots, d_n)$ gilt: $|d_l| \leq 2 \wedge d_i{=}0$, $i \neq l$, oder
2. Z_1 und Z_2 greifen auf dasselbe Array zu und unterscheiden sich um höchstens d' in der ersten Dimension, wobei d' $\leq$ Anzahl der Arrayelemente, die in eine Cachezeile passen.

Während Bedingung (1) zeitliche Lokalität zusammenfaßt, d.h. Zugriffe auf ein und dieselbe Speicherstelle, erkennt Bedingung (2) die meisten Arten von räumlicher Lokalität, d.h. Zugriffe auf Elemente, die aufgrund früherer Hauptspeicherzugriffe bereits im Cache liegen. Räumliche Lokalität schließt zeitliche Lokalität ein. Die Zahl zwei in Bedingung 1.(b) beruht auf Erfahrungswerten. Einerseits treten nur selten größere Konstanten auf, andererseits garantiert diese Einschränkung, daß bei einer Cachezeilen-Größe von mindestens zwei Elementen höchstens zwei Cachezeilen für die Zugriffe benutzt werden.

Beispiel:

```
for K := 2 to N-1 do
  for J := 2 to N-1 do
    for I := 2 to N-1 do
      A[I,J,K] := A[I+1,J+1,K] + B[I,J,K] + B[I,J+1,K] + B[I+1,J,K];
    end
  end
end
```

Zugriffsgruppen bzgl. Schleife J:	Aufgrund von Regel
{A[I,J,K]}	erfüllt keine Regel
{A[I+1,J+1,K]}	erfüllt keine Regel
{B[I,J,K], B[I,J+1,K], B[I+1,J,K]}	1.(b) und 2.

Zugriffsgruppen bzgl. Schleifen I & K:	Aufgrund von Regel
{A[I,J,K]}	erfüllt keine Regel
{A[I+1,J+1,K]}	erfüllt keine Regel
{B[I,J,K], B[I+1,J,K]}	2.
{B[I,J+1,K]}	erfüllt keine Regel

6.2 Kosten der einzelnen Schleifen ermitteln

Im nächsten Schritt wird für jede Schleife berechnet, wieviele Cachezeilen gefüllt werden müßten, wenn sie an innerster Stelle säße. Folgende Daten werden benötigt:

EINGABE:

$$
\begin{aligned}
\mathcal{L} &= \{l_1, \ldots, l_n\} \\
\mathcal{R} &= \{Z_1, \ldots, Z_n\} \\
trip_l &= (hi_l - lo_l + step_l)/step_l \\
cls &= \text{die Cachezeilen-Größe gemessen in Datenelementen} \\
coeff(f, i_l) &= \text{der Koeffizient der Indexvariablen } i_l \text{ im Indexausdruck f} \\
stride(f_1, i_l, l) &= |step_l * coeff(f_1, i_l)|
\end{aligned}
$$

$\mathcal{L}$	ist eine n-fache Schleifenschachtelung in der Form $lo_l, hi_l, step_l$ (Untergrenze, Obergrenze, Schrittweite).
$\mathcal{R}$	ist je ein beliebigen Vertreter aus jeder Zugriffsgruppe.
$trip_l$	ist die Anzahl der tatsächlich durchlaufenen Iterationen in Schleife l.
$stride(f_1, i_l, l)$	gibt an, wie weit die Arrayelemente auseinanderliegen, die beim Durchlaufen der Schleife l referenziert werden.

AUSGABE:
LoopCost(l) = die Anzahl der benutzten Cachezeilen, falls Schleife l an innerster Stelle sitzt.

Die Kosten für eine bestimmte Schleife ergeben sich aus der Summe der Kosten der Zugriffsgruppen, multipliziert mit den $trip_l$ aller umliegenden Schleifen.

$$\textbf{LoopCost}(\mathrm{l}) = \sum_{k=1}^{m}(\textbf{RefCost}(\mathrm{Z_k}(\mathrm{f_1}(\mathrm{i_1},\ldots,\mathrm{i_n}),\ldots,\mathrm{f_j}(\mathrm{i_1},\ldots,\mathrm{i_n})),\mathrm{l}))\prod_{\mathrm{h}\neq\mathrm{l}}\mathrm{trip_h}$$

Die Kosten einer Zugriffsgruppe, dargestellt durch Vertreter Z_k, falls sich Schleife l an innerster Stelle befindet, errechnen sich folgendermaßen:

$$\textbf{RefCost}(Z_k, l) = \begin{cases} 1 & falls((coeff(f_1, i_l) = 0) \wedge \ldots \wedge (coeff(f_j, i_l) = 0)) \\ \frac{trip_l}{\left(\frac{cls}{stride(f_1, i_l, l)}\right)} & falls((stride(f_1, i_l, l) < cls) \wedge (coeff(f_2, i_l) = 0) \\ & \wedge \ldots \wedge (coeff(f_j, i_l) = 0)) \\ trip_l & sonst \end{cases}$$

Die Kostenberechnung soll an der folgenden Matrixmultiplikation verdeutlicht werden:

```
for J := 1 to N do
  for K := 1 to N do
    for I := 1 to N do
      C[I,J] := C[I,J] + A[I,K] * B[K,J];
    end
  end
end
```

Die Zugriffsgruppen sind bzgl. aller drei Schleifen gleich und lauten

{C[I,J]}
{A[I,K]}
{B[K,J]}.

Für dieses Beispiel sei die Cachezeilen-Größe cls=4.
Sowohl A[I,K] als auch C[I,J] bieten räumliche Lokalität bzgl. Schleife I und bekommen jeweils die Zugriffskosten n*(1/cls) zugewiesen. B[K,J] hat innerhalb der Schleife I zeitliche Lokalität. Es wird daher für B[K, J] nur eine Cachezeile benötigt; die Zugriffskosten sind folglich 1. Die Schleifenkosten von I sind also $\frac{1}{2} * n^3 + n^2$. Die Kosten für die Schleifen J und K ergeben sich analog, folgende Tabelle zeigt die Ergebnisse im Detail:

Zugriff	J	K	I
C(I,J)	$n * n^2$	$1 * n^2$	$\frac{1}{4}n * n^2$
A(I,K)	$1 * n^2$	$n * n^2$	$\frac{1}{4}n * n^2$
B(K,J)	$n * n^2$	$\frac{1}{4}n * n^2$	$1 * n^2$
gesamt	$2n^3 + n^2$	$\frac{5}{4}n^3 + n^2$	$\frac{1}{2}n^3 + n^2$

6.3 Schleifen permutieren

Der letzte Teil des Algorithmus ermittelt anhand der einzelnen Schleifenkosten eine Reihenfolge für die Schleifen, in der die Lokalität am größten ist. Danach wird versucht, diese Reihenfolge durch Schleifenvertauschung zu erreichen. Sollte dies aufgrund von Datenabhängigkeiten nicht möglich sein, wird versucht, Konflikte mit Richtungsumkehr aufzulösen. Schlägt auch das fehl, so liefert dieser Schritt wenigstens eine Schleifenreihenfolge, die der idealen am nächsten ist.

Bei der Ermittlung der idealen Schleifenanordnung kann man sich auf folgende Beobachtung stützen:

Wenn sowohl Schleife l als auch Schleife l' als innerste Schleife in Betracht kommen, l aber mehr Lokalität bietet, so wird l an einer beliebigen Position weiter außen wahrscheinlich mehr Lokalität bieten als l'.

Daher kann man die Schleifen ganz einfach anhand ihrer Zugriffskosten von außen nach innen so sortieren, daß die Schleifenkosten dabei abnehmen: $\text{LoopCost}(l_{i-1}) \geq \text{LoopCost}(l_i)$. Folgender Algorithmus ermittelt die bestmögliche Reihenfolge:

EINGABE:

$$\begin{aligned} \mathcal{O} &= \{i_1, \ldots, i_n\}, \text{die ursprüngliche Schleifenreihenfolge} \\ \mathcal{DV} &= \text{die ursprünglichen Abhänigkeitsvektoren von } l_n \\ \mathcal{L} &= \{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\} \end{aligned}$$

AUSGABE:

$\mathcal{P} = \mathcal{L}$, oder eine legale Anordnung von $\mathcal{O}$, die so nah an $\mathcal{L}$ liegt wie möglich

ALGORITHMUS:

if $\forall\ \mathcal{DV}$, $\mathcal{L}$ ist eine legale Vertauschung
  return $\mathcal{L}$
$\mathcal{P} = \emptyset$; k=0; m=n
while $\mathcal{L} \neq \emptyset$
  for $j = 1, m$
    $l = l_j \in \mathcal{L}$
    if Abhängigkeitsvektoren für $\{p_1, \ldots, p_k, l\}$ gültig sind
      $\mathcal{P} = \{p_1, \ldots, p_k, l\}$
      $\mathcal{L} = \mathcal{L} - \{l\}$; $k = k + 1$; $m = m - 1$
      break for
    endif
  endfor
endwhile

Wenn eine ideale Schleifenreihenfolge L=$\{i_{\sigma_1}, \ldots, i_{\sigma_n}\}$ der Schleifen $i_1, \ldots, i_n$ gegeben ist, wobei i_{σ_1} am wenigsten Lokalität erreicht und i_{σ_n} am meisten, erstellt der Algorithmus eine legale Permutation $\mathcal{P}$, indem er zunächst prüft, ob Schleife i_{σ_n}, ohne Datenabhängigkeiten zu verletzen an die äußerste Stelle gesetzt werden kann. Ist das möglich, wird sie zu $\mathcal{P}$ hinzugefügt, und aus $\mathcal{L}$ herausgenommen. Sollte die Schleife nicht an eine bestimmte Stelle plaziert werden können, wird die nächste Schleife in $\mathcal{L}$ überprüft, so lange, bis eine Schleife gefunden wird. Der Algorithmus läuft, solange sich noch Elemente in $\mathcal{L}$ befinden und enthält nach seiner Beendigung in $\mathcal{P}$ diejenige Schleifenreihenfolge, die der idealen am nächsten kommt.

7 Algorithmus von Wolf und Lam

Ziel von Wolf/Lam [3] ist es ebenfalls, einen Algorithmus zu präsentieren, der innerhalb eines restrukturierenden Compilers Schleifentransformationen vornehmen kann. Im wesentlichen basiert der Algorithmus auf der Beobachtung, daß durch Blockung die verschieden Arten von Lokalität am besten ausgenutzt werden können. Anhand einer algebraischen Darstellung von Wiederverwendung und Lokalität, wird zunächst festgestellt, welche Arten von Wiederverwendung überhaupt möglich sind, und dann versucht, die Schleifen so zu blocken, daß die Lokalität so gut wie möglich ausgenutzt werden kann. Um die Voraussetzungen zu schaffen, daß eine oder mehrere Schleifen geblockt werden dürfen kommen die bereits vorgestellten unimodularen Transformationen zum Einsatz.

Gegeben sei folgendes Codebeispiel:

```
for I := 0 to 5 do
  for J := 0 to 6 do
    A[J+1] := (1/3) * (A[J] + A[J+1] + A[J+2]);
  end
end
```

Die Abhängigkeitsvektoren der Zugriffe lauten:

$$D = \{(0,1),(1,0),(1,-1)\}$$

Die Zugriffe, die auf das Array A gemacht werden, sind einheitlich erzeugte Zugriffe, d.h.sie sind alle der Form:

$$A[f(i)], wobei f(i) = Hi + c_f$$

H ist eine $n \times m$ Matrix, mit n = Anzahl der Schleifen bzw. der Indexvariablen, m = Dimension des Arrays A. i ist der Indexvektor der Schleifenschachtelung, d.h. er enthält die Laufvariablen von der äußersten zur innersten Schleife. c_f ist eine von Zugriff zu Zugriff verschiedene Konstante.
Die Zugriffe des obigen Beispiels lassen sich schreiben als:

Zugriff	H	i	c_f
A[J]	$\begin{pmatrix} 0 & 1 \end{pmatrix}$	$\begin{pmatrix} I_1 \\ I_2 \end{pmatrix}$	$+(0)$
A[J+1]	$\begin{pmatrix} 0 & 1 \end{pmatrix}$	$\begin{pmatrix} I_1 \\ I_2 \end{pmatrix}$	$+(1)$
A[J+2]	$\begin{pmatrix} 0 & 1 \end{pmatrix}$	$\begin{pmatrix} I_1 \\ I_2 \end{pmatrix}$	$+(2)$

Wolf und Lam zeigen nun, wie sie anhand dieser Notation Wiederverwendung zwischen verschieden Iterationen ermitteln können. Dabei ist zu beachten, daß Wiederverwendung nicht automatisch auch Lokalität bedeutet. Letztere kann nur dann ausgenutzt werden, wenn die wiederverwendete Speicherstelle sich auch noch im Cache befindet. Ziel des Algorithmus wird es also sein, Schleifen so zu blocken, daß mögliche Lokalität auch ausgenutzt werden kann. Folgende Arten von Wiederverwendung werden unterschieden:

7.1 Eigen-zeitliche Wiederverwendung (self-temporal)

Man versteht darunter einen Zugriff auf immer dasselbe Cacheelement von immer derselben Arrayreferenz aus.

In unserem Beispiel besteht eigen-zeitliche Wiederverwendung bei allen Zugriffen auf das Array A bzgl. der Schleife I, da bei jedem Durchlauf von I genau die selben Elemente innerhalb von J bearbeitet werden.

Mathematisch ausgedrückt greifen zwei Iterationen i_1 und i_2 genau dann auf dasselbe Datenelement zu, wenn gilt: $Hi_1 + c = Hi_2 + c$, oder anders ausgedrückt: $H(i_1 - i_2) = 0$. Wenn man mit r die Richtung bezeichnet, in der Wiederverwendung stattfindet, erhält man diese durch $Hr = 0$. Das ist aber gerade der Kern von H. Der zur eigen-zeitlichen Wiederverwendung gehörende Vektorraum, kurz R_{ST} ist also:

$$R_{ST} = kernH$$

Nehmen wir etwa die Referenz A[J] heraus, so ist:

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

$$R_{ST} = kernH = span\{(1,0)\}$$

Mit $span\{(1,0)\}$ ist der von (1,0) aufgespannte Vektorraum gemeint. R_{ST} gibt als Richtung für die Wiederverwendung also wie erwartet genau die Dimension der äußersten Schleife an.

7.2 Eigen-räumliche Wiederverwendung (self-spatial)

Räumliche Wiederverwendung ist ein Oberfall von zeitlicher Wiederverwendung und bedeutet Zugriffe verschiedener Iterationen auf die gleiche Cachezeile. Auch hier wird immer nur der Zugriff ein und derselben Arrayreferenz betrachtet, nur daß diesmal die Zugriffe entlang der Speicheranordnung differieren dürfen. Während der Schleifenabarbeitung werden A[0], A[1], A[2], A[3], usw. referenziert. Da man die Richtung dieser Dimension (das ist ja gerade die Richtung, in der Räumlich Wiederverwendung möglich ist) dem Kern hinzufügen möchte, setzt man in der Matrix H einfach die zugehörige Dimension auf Null (ergibt Matrix H_S). Im obigen Beispiel betrifft das die erste Reihe von H. Die Richtung der Wiederverwendung wird somit durch Kern H angegeben:

$$R_{SS} = kernH_S$$

R_{SS} bezeichnet den zu eigen-räumlicher Wiederverwendung gehörenden Vektorraum, der in unserem Beispiel

$$kernH_S = kern \begin{pmatrix} 0 & 0 \end{pmatrix} = span\{(1,0),(0,1)\}$$

lautet.
Die Tatsache, daß räumliche zeitliche Lokalität einschließt, spiegelt sich in der Tatsache wieder, daß

$$kernH \subset kernH_S$$

gilt.

7.3 Gruppen-zeitliche und gruppen-räumliche Wiederverwendung

Diese beiden Möglichkeiten der Wiederverwendung verhalten sich analog zu den eben vorgestellten, mit dem Unterschied, daß jetzt auch Zugriffe verschiedener Arrayreferenzen auf ein und dasselbe Cacheelement (gruppen-zeitlich), oder lediglich auf dieselbe Cachezeile (gruppen-räumlich) betrachtet werden.

Um festzustellen, welche Zugriffe Elemente aus der selben Cachezeile benutzen können, werden sie in Zugriffsklassen eingeteilt, wie schon bei McKinley et al. [2] beschrieben, und anschließend wird innerhalb der Klassen auf Lokalität geprüft.

7.4 Der Lokalitäts-Vektorraum

Grundsätzlich kann man bei einer Schleifenschachtelung nicht davon ausgehen, daß jede mögliche Wiederverwendung auch wirklich zu Cache-Hits führt. Bei einer Schachtelung, die noch nicht durch Blockung optimiert worden ist, ist sowohl räumliche als auch zeitliche Wiederverwendung lediglich innerhalb der innersten Schleife sichergestellt. Der Raum, der angibt, in Richtung welcher Schleifendimensionen Wiederverwendung tatsächlich sicher ist, wird Lokalitätsvektorraum L genannt. Bei unmodifizierten Schachtelungen wie in unserem Beispiel, besteht er lediglich aus dem eindimensionalen Raum in der Richtung der innersten Schleife:

$$L = span\{(0,1)\}$$

Da Lokalität nur dann ausgenutzt werden kann, wo sich die Vektorräume für Wiederverwendung R und Lokalität L überschneiden. Daher versucht man nun die Schleifen so zu blocken, daß sich die Dimension von L, und damit auch die des Schnittes $R \cap L$ erhöht.

Blocken darf man Schleifen genau dann, wenn man sie auch Vertauschen dürfte. Es dürfen also keine Datenabhängigkeiten verletzt werden. Falls ein solcher Konflikt vorliegt, sucht man nach einer geeigneten Transformation wie Richtungsumkehr oder Schleifenneigen um den Abhängigkeitskonflikt aufzulösen.

In unserem Fall bringt uns Schleifenneigen ans Ziel:

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

ändert die Abhängigkeitsvektoren zu

$$D' = UD = \{(0,1),(1,1)(1,0)\}$$

Der negative Eintrag im letzten Vektor ist verschwunden, die Vektoren bleiben auch nach einer Vertauschung lexikographisch positiv. Der entstehende Code darf nun geblockt werden, so daß nicht nur die räumliche Lokalität aus Schleife J, sondern auch die zeitliche aus Schleife I genutzt werden kann.

8 Schluß

Die vorgestellten unimodularen Schleifentransformationen stellen einen elementaren Grundstock dar, um Schleifenschachtelungen hinsichtlich ihrer Lokalitätsausbeute zu optimieren. Ihre Stärken spielen sie gerade durch ihre Einfachheit aus: keine andere Schleifentansformation läßt sich so bequem durch Matrizen darstellen und somit algorithmisch verarbeiten. Lockert man die Beschränkung ein wenig auf und erlaubt lediglich invertierbare Matrizen, so kann man zusätzlich noch Skalieren durch Matrizen modellieren. Dies ist eine Transformation, die bei Parallelrechnern eine Rolle spielt [4].

Die unimodularen Transformationen sind im wesentlichen die Wegbereiter für nachfolgende Transformationen wie Blockung oder Schleifenverschmelzen. Lediglich die Vertauschung bringt mitunter eine erhebliche Laufzeitverbesserung. Auch die beiden Algorithmen, die aufgrund ihrer Komplexität und ihres Umfangs lediglich angerissen werden konnten, bauen auf den zuerst durchgeführten unimodularen Transformationen auf: McKinley et al. [2] benutzen zusätzlich Schleifenrumpfteilen und Schleifenverschmelzen, Wolf und Lam [3] erreichen bessere Cacheausnutzung durch Blockung.

Aber auch wenn man keinen restrukturierenden Compiler zur Verfügung hat, so kann man auch durch richtige Anordnung von Schleifen, beispielsweise bei einer einfachen Matrixmultiplikation, je nach Matrixgröße und Architektur Geschwindigkeitsverbesserungen zwischen Faktor 4 und 24 herausholen [2]. Das macht diese Transformationen doch wichtiger, als sie auf den ersten Blick zu sein scheinen.

Literatur

[1] Michael Wolfe, *High performance compilers for parallel computing.* Addison Wesley, 1996

[2] Kathryn McKinley, Steve Carr, Chau-Wen Tseng, Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424-453, Juli 1996

[3] Michael Wolf, Monica Lam, A data Locality optimizing algorithm. *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, ACM SIGPLAN Notices 26(6):30-44, Toronto, Canada, Juni 1991.

[4] W. Li, K. Pingali, Access normalization: Loop restructuring for NUMA compilers. *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM SIGPLAN Notices 27(9):285-295, Boston, MA, Oktober 1992.

Seminar - Ausarbeitung II

Matthias Grutzeck

Blockung

Zusammenfassung

Wenn mit Arrays, die zu groß sind, um in den Cache zu passen, Berechnungen durchgeführt werden sollen, wird häufig Blockung als Technik verwandt. Hier soll dieser Technik nachgegangen werden und erörtert werden, inwieweit Compiler Algorithmen selbstständig in geblockte Varianten umformen können. Verstärkt wird dann auf die Ermittlung von Blockgrößen eingegangen, um durch die Blockung ein möglichst gutes Cacheverhalten zu erzielen. Es werden zwei Algorithmen vorgestellt, die die Blöcke nach dem Gesichtspunkt der Vermeidung von Selbstinterferenz auswählen.

1 Einleitung

In vielen Bereichen wird heutzutage der Computer zu numerischen Berechnungen verwandt. Dabei handelt es sich meist um große Datenmengen, die die gleiche Berechnung durchlaufen. Die Daten sind dabei oftmals in Arrays gespeichert, die schnell die Größe von heute verwandten Caches übersteigt. Dadurch enden bei einem naiven Vorgehen die meisten Speicherzugriffe in einem Zugriff auf den langsamen Hauptspeicher. Dies schlägt sich deutlich in der Ausführungszeit des Programmes nieder. Hier sind Techniken gefragt, die dem Verhalten entgegenwirken und es ermöglichen, mehr Speicherzugriffe aus dem Cache zu bedienen. Hierzu werden wir die Technik der Blockung kennenlernen und Teilaspekte unter verschiedenen Gesichtspunkten betrachten.

Nun kann das Verhalten eines Cache besser ausgenutzt werden, indem man die Lokalität von Daten durch eine Umstrukturierung des Programmes erreicht. Verwendet das Programm bisher eine geschachtelte Schleife, in der Daten im Cache in einem Schleifendurchgang der äußeren Schleife wiederverwendet werden, ist es sinnvoll darauf zu achten, daß die innere Schleife auf Grund der Datenmenge dieses Vorhaben nicht zunichte macht. Es wird dann versucht, die innere Schleife in Blöcke aufzuteilen. Die äußere Schleife braucht dann nur über solch einen Block zu arbeiten. Eine weitere Schleife, die über die bisherigen Schleifen iteriert, sorgt dafür, daß die Blöcke abgearbeitet werden. Die Größe der Blöcke wird nun so gestaltet, daß der Cache eine möglichst minimale Anzahl von Cachemisses zu verbuchen hat.

2 Begriffsklärungen

Bevor wir uns dies mit einem Beispiel verdeutlichen, benötigen wir zuerst ein paar Begriffe. Diese klären wir hier zuerst, um eine einheitliche Terminologie zu verwenden.

2.1 Lokalität

Oft können Zugriffe mit Daten aus dem Cache bedient werden. Bei solch einer Lokalität von Daten unterscheidet man:

zeitliche Lokalität: Auf das gleiche Datum wird mehrmals zugegriffen und auf Grund dessen ist das Datum bereits im Cache vorhanden.

räumliche Lokalität: Bei einen Zugriff auf ein anderes Datum wurde das gewünschte Datum, bei dem räumliche Lokalität auftritt, bereits geladen und ist deswegen im Cache vorhanden. Dieses Verhalten kommt auf Grund der Cacheline, die auf einmal in den Cache geladen wird, zu Tage.

Die Anzahl der Verwendung eines Datums wird durch den *Wiederverwendungsfaktor* beschrieben.

2.2 Interferenzen

Ein Datum kann aus verschiedenen Gründen aus dem Cache gelöscht werden. Bei jedem Verdrängen eines Datums ist dies durch den Zugriff auf ein anderes Datum bedingt. Dies nennt man *Interferenz*. Man unterscheidet:

Selbstinterferenz: Das Datum, das ein anderes Datum aus dem Cache verdrängt, entstammt dem gleichen Array.

Fremdinterferenz: Das Datum, das ein anderes Datum aus dem Cache verdrängt, entstammt einer anderen Variable oder einem anderen Array.

2.3 Cachemiss

Ist ein Datum, auf das zugegriffen wird, nicht im Cache vorhanden, erhalten wir einen *Cachemiss*. Je nach Grund für den Cachemiss unterscheidet man:

unvermeidbarer Cachemiss: Es ist der erste Zugriff auf die Cacheline, in der das Datum steht. Dieser Zugriff ist dann zwangsweise nicht aus dem Cache zu bedienen.

Kapazitäts-Cachemiss: Auf das Datum wurde zuvor bereits zugegriffen, doch auf Grund der Größe des Cache mußte das Datum verdrängt werden. Bei dieser Aussage wird von einen voll-assoziativen Cache mit einer Cachelinelänge von einen Element ausgegangen.

Interferenz-Cachemiss: Im Cache-Speicher wäre noch genug Platz vorhanden, um das Datum zu behalten, doch auf Grund der Verdrängungsstrategie wurde das Datum verdrängt. Hier kann man noch nach den Begriffen Selbstinterferenz- und Fremdinterferenz-Cachemiss gemäß den obigen Definitionen unterscheiden.

3 Blockung am Beispiel Matrixmultiplikation

Am Beispiel Matrixmultiplikation soll das Grundprinzip der Blockungstechnik verdeutlicht werden. Seien A, B und C $N \times N$ Matrizen. Wir wollen $C{=}A\ B$ berechnen. Der erste Ansatz, dies zu programmieren, sieht wie folgt aus:

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

Für das Folgende nehmen wir an, daß die Matrizen spaltenweise im Speicher abgelegt sind wie zum Beispiel in FORTRAN. Wir beobachten eine räumliche Lokalität bei den Lesezugriffe auf die Matrixelemente $A[i][k]$ und $C[i][j]$ und eine zeitliche Lokalität für den Schreibzugriff auf das Matrixelement $C[i][j]$. Das Element $B[k][j]$ weist räumliche Lokalität auf, wenn die j-te Spalte von der Matrix C und die k-te Spalte von der Matrix A nicht zu einer Verdrängung von dem Element $B[k][j]$ geführt haben. Dies sind $2N$ Elemente, die der Cache minimal halten können muß.

Das Element $C[i][j]$ kann auch für den Lesezugriff zeitliche Lokalität aufweisen, wenn die k-te Zeile der Matrix A und die j-te Zeile der Matrix C und das Element $B[k+1][j]$ nicht zu einer Verdrängung geführt haben. Dies sind N zusätzliche Elemente, die nicht verdrängt werden sollten.

Eine zusätzliche zeitliche Lokalität für das Element $A[i][k]$ ist gegeben, wenn die gesamte Matrix A und die j-te Spalte der Matrix C nicht zu einer Verdrängung geführt haben. Dies sind zusätzliche $N^2 - N$ Elemente, die berücksichtigt werden sollten.

Wurde N so groß gewählt, daß keine zeitliche Lokalität vorkommt, haben wir für die Zugriffe auf die Matrix A und C jeweils ungefähr N^3/CLS Speicherzugriffe, wobei CLS die Größe der Cacheline in Anzahl Elementen ist. Dazu kommen N^2 Speicherzugriffe für die Beschaffung von Elementen $B[k][j]$. Dies ergibt $2 \cdot N^3/CLS + N^2$ Speicherzugriffe.

Wir wollen versuchen, die zeitliche Lokalität der Elemente $A[i][k]$ auch bei großen Matrizen zu erhöhen. Bei der jetzigen Konstellation muß der Cache N^2+N Elemente halten. Wenn die beiden inneren Schleifen nur über kleine Bereiche iterieren würden, bräuchte der Cache weniger Elemente halten, bevor es zu einer zeitlichen Lokalität kommt. Wir führen zwei weitere Schleifen ein:

```
for (kk=0; kk < N; kk+=blockgröße_k)
  for (ii=0; ii < N; ii+=blockgröße_i)
    for (j=0; j < N; ++j)
      for (k=kk; k < min(kk+blockgröße_k, N); ++k) {
        r=B[k][j];
        for (i=ii; i < min(ii+blockgröße_i, N); ++i)
          C[i][j] += A[i][k] * r;
      }
```

Wir wollen die drei inneren Schleifen, die einem Block entsprechen, auf Lokalität für die Matrizen untersuchen. Wir haben wieder räumliche Lokalität für die Zugriffe auf die Elemente $A[i][k]$ und $C[i][j]$. $B[k][j]$ hat räumliche Lokalität, wenn $2 \cdot blockgröße_i$ Elemente nicht zu einer Verdrängung geführt haben. Die Elemente $A[i][k]$ haben zeitliche Lokalität, wenn $blockgröße_i \cdot blockgröße_k$ Elemente der Matrix A, $blockgröße_i$ Elemente der Matrix C und $blockgröße_k$ Elemente der Matrix B im Cache gehalten werden können.

Man sieht, daß bei geeigneter Wahl der Parameter $blockgröße_k$ und $blockgröße_i$ zeitliche Lokalität auch für große Matrizen innerhalb eines Blocks entsteht. Hier erkennt man, daß man die Blockgrößen nicht so groß wählen darf, daß zuviel Interferenz innerhalb eines Blocks entsteht, aber auch nicht so klein, daß der Cache nicht komplett genutzt wird. Wir werden später noch feststellen, daß die Wahl der Blockgrößen von der Cachelinegröße, der Cachegröße in Anzahl Elementen und der Matrixgröße abhängig ist.

4 Blockung durch Compiler

Die Blockung von Algorithmen wurde bisher meist durch Handoptimierung erreicht. Diese Optimierungen sind leider meist maschinenspezifisch durchgeführt worden. Ein weiterer Nachteil von handoptimierten Code ist der Verlust an Verständlichkeit. Es stellt sich die Frage, ob die Optimierung, die Algorithmen blockt, durch einen Compiler vorgenommen werden kann. Dadurch würde man sowohl Zielplattformunabhängigkeit als auch eine Entlastung des Programmierers erreichen.

4.1 Verschiedene Transformationen

Für eine automatisierte Blockung eines Algorithmus werden hier einige Transformationen vorgestellt, die der Compiler anwenden kann. Diese sind Carr et al [CK92] entnommen. Oftmals ist es nötig, mehrere, verschiedene Transformationen nacheinander anzuwenden, um zu einem guten, geblockten Algorithmus zu gelangen.

4.1.1 Streifen-Schneiden-Und-Vertauschen

Bei der *Streifen-Schneiden-Und-Vertauschen* Transformation werden zwei Schleifen, die ineinander geschachtelt sind, genommen. Jetzt kann man eine von den beiden Schleifen auf Werte, die wiederverwendet werden, untersuchen. Die tragende Schleife, die für die Wiederverwertung von Werten verantwortlich ist, kann nun unterteilt werden. Man führt eine weitere Schleife vor der tragende Schleife ein, die eine Schrittweite der Größe des Blockungsfaktor hat und die ursprünglichen Grenzen besitzt. Die alte Schleife

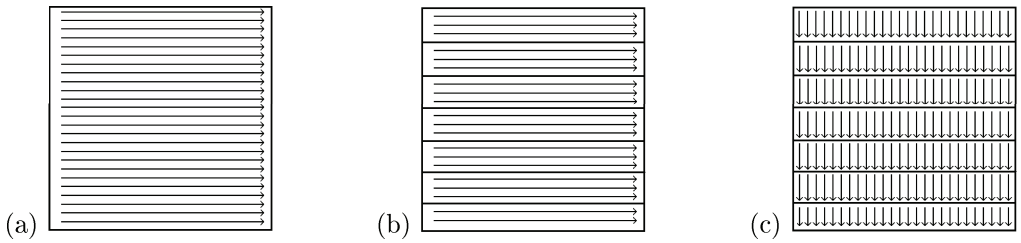


Abbildung 1: Reihenfolge der Zugriffe auf die Matrix B *(a) im Original*, nach dem *(b) Streifen-Schneiden* und nach dem *(c) Streifen-Schneiden-Und-Vertauschen*

erhält als neue Grenzen den aktuellen Wert der äußeren Schleife und das Minimum aus dem aktuellen Wert der äußeren Schleife addiert um Blockungsfaktor und der alten Schleifenobergrenze. Anschließend werden, wenn nötig, Schleifen vertauscht, um ein besseres Cache-Verhalten zu erreichen. Dieses Verfahren kann nur angewendet werden, wenn die Ausführungsreihenfolge irrelevant ist.

Es sei zum Beispiel folgender Algorithmus gegeben:

```
for (i=unter_i; i < ober_i; ++i)
  for (j=unter_j; j < ober_j; ++j)
    A[i]=A[i]+B[i][j]
```

In dem Vektor A werden hier die einzelnen Zeilensummen der Matrix B gespeichert. Die Reihenfolge der Zugriffe auf die Matrix B entspricht der Abbildung 1 (a). Wenn wir hier eine spaltenweise Haltung der Matrix B im Speicher voraussetzen (z.B. wie in FORTRAN), dann haben wir nur für kleine Problemgrößen zeitliche und räumlichen Lokalität für die Matrixelemente $B[i][j]$ und eine zeitliche Lokalität für die Vektorelemente $A[i]$.

Wir versuchen jetzt, die Transformation Streifen-Schneiden-Und-Vertauschen anzuwenden. Dazu führen wir zuerst eine weitere Schleife ein. Dies entspricht dem Streifen-Schneiden:

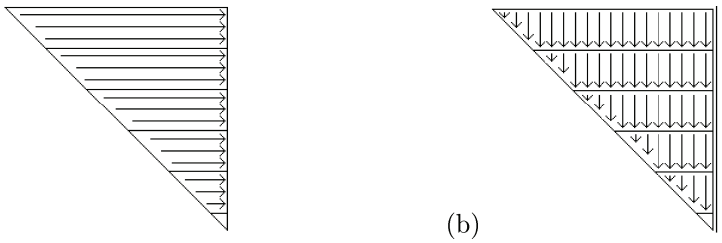
```
for (i=unter_i; i < ober_i; i+=blockgröße_i)
  for (ii=i; ii < min(i+blockgröße_i, ober_i); ++ii)
    for (j=unter_j; j < ober_j; ++j)
      A[ii]=A[ii]+B[ii][j]
```

Nach Abbildung 1 (b) hat sich die Reihenfolge der Zugriffe auf die Matrix B durch das in Streifen-Schneiden nicht geändert. Hier könnten andere Techniken wie zum Beispiel Vektorisierung oder Parallelität angewandt werden, um aus dem Streifen-Schneiden direkt Nutzen zu ziehen. Wir gehen zur nächsten Teiltransformation über und vertauschen die beiden inneren Schleifen und schauen, was wir dadurch gewinnen:

```
for (i=unter_i; i < ober_i; i+=blockgröße_i)
  for (j=unter_j; j < ober_j; ++j)
    for (ii=i; ii < min(i+blockgröße_i, ober_i); ++ii)
      A[ii]=A[ii]+B[ii][j]
```

Jetzt hat sich die Reihenfolge der Zugriffe auf die Matrix B nach Abbildung 1 (c) verändert. Räumliche Lokalität von den Matrixelementen $B[ii][j]$ wird durch direkt aufeinanderfolgenden Zugriffe auf die Matrix B getragen. Zusätzlich gewinnen wir räumliche Lokalität für die Zugriffe auf die Matrixelemente $B[ii][j]$ und die die Vektorelemente $A[ii]$, wenn die $blockgröße_i$ sinnvoll gewählt wird. Wie man die Blockgröße wählt, werden wir später noch betrachten. Durch den Gewinn von räumlicher und zeitlicher Lokalität bei guter Wahl des Parameters $blockgröße_i$ liegt hier mehr Lokalität im Vergleich zum Originalalgorithmus vor.

Die Minimumsbildung bei der Obergrenze der innersten Schleife kann verhindert werden, indem man von der äußeren Schleife die Obergrenze um die Blockgröße verringert und nach dem gesamten Block die fehlende Schleife ergänzt. In dem Beispiel sieht das so aus:



(a) (b)

Abbildung 2: Ausführungsreihenfolge nach *(a) Streifen-Schneiden* und nach *(b) Streifen-Schneiden-Und-Vertauschen*

```
for (i=unter_i; i < (ober_i/blockgröße_i)*blockgröße_i; i+=blockgröße_i)
  for (j=unter_j; j < ober_j; ++j)
    for (ii=i; ii < i+blockgröße_i; ++ii)
      A[ii]=A[ii]+B[ii][j]
for (j=unter_j; j < ober_j; ++j)
    for (ii=(ober_i/blockgröße_i)*blockgröße_i; ii < ober_i; ++ii)
      A[ii]=A[ii]+B[ii][j]
```

Dieses Verfahren läßt sich auch auf Dreiecks-Iterationsstrukturen übertragen. Der Vorgang des Schleifentauschs wird hierbei aufwendig. Hier eine allgemeine Schleife nach dem Vorgang Streifen-Schneiden. m sei die Steigung und b die Verschiebung der Dreiecks-Iterationsstruktur:

```
for (i=unter_i; i < ober_i; i+=blockgröße_i)
  for (ii=i; ii < min(i+blockgröße_i, ober_i); ++ii)
    for (j=m*ii+b; j < ober_j; ++j)
      Anweisung(ii,j);
```

Bisher ist eine weitere Schleife entstanden, dennoch hat sich die Ausführungsreihenfolge nicht verändert, wie die Abbildung 2 (a) verdeutlicht. Um den Austausch durchführen zu können, müssen nun die Grenzen der ii-Schleife verändert werden:

```
for (i=unter_i; i < ober_i; i+=blockgröße_i)
  for (j=m*i+b; j < ober_j; ++j)
    for (ii=i; ii < min((j-b)/m, i+blockgröße_i, ober_i); ++ii)
      Anweisung(ii,j);
```

Erst jetzt erfolgt eine Blockung der Anweisung, wie auch die Abbildung 2 (b) verdeutlicht.

4.1.2 Schleifenbereiche Teilen

Die zuvor besprochene Streifen-Schneiden-Und-Vertauschen Transformation läßt sich nur auf Rechteck- oder Dreieck-Iterationsstrukturen anwenden. Sind diese nicht gegeben, kann die Transformation *Schleifenbereiche Teilen* solche Voraussetzungen schaffen. Bei dieser Transformation wird die Schleife in mehrere, nacheinander ausgeführte Schleifen aufgeteilt. Das heißt, daß der Schleifenkörper genauso oft und in der gleichen Reihenfolge ausgeführt wird, wie zuvor, nur daß die Schleifen andere Grenzen haben und mehrmals nacheinander vorkommen.

Ein einfaches Beispiel kann dies verdeutlichen:

```
for (i=unter_i; i < ober_i; ++i)
  Anweisungen;
```

Diese einfache Schleife wird in mehrere Schleifen auseinandergebrochen. N sei dabei ein beliebiger, sinnvoller Wert:

for (i=$unter_i$; $i < \min(ober_i,N)$; $++i$)
  Anweisungen;
for (i=$\min(ober_i,N)$; $i < ober_i$; $++i$)
  Anweisungen;

Ein praktisches Beispiel für diese Transformation stellt die Teilung von einer Trapez-Iterationsstruktur dar. Diese kann durch Schleifenbereiche Teilen in zwei Schleifen mit Dreieck-Iterationsstruktur und eine Schleife mit Rechteck-Iterationsstruktur geteilt werden, ohne die Ausführungsreihenfolge zu verändern. Anschließend wendet man die Streifen-Schneiden-Und-Vertauschen Transformation auf die drei Schleifen an und erhält das gewünschte Ergebnis.

Muß ein Wert von einem Array berechnet werden, bevor ein anderer Wert im gleichen Array berechnet werden kann, und dies durch den gleichen Schleifenkörper gemacht wird, haben wir eine Abhängigkeit, die Streifen-Schneiden-Und-Vertauschen verhindert. Diese kann durch Schleifenbereiche Teilen aufgelöst werden, indem man die Schleife in mehrere Schleifen zerteilt, die diese Abhängigkeit auf Grund von kleineren Intervallen nicht mehr aufweisen. Dazu ist es nötig, den größtmöglichen Index-Set zu finden. Dies ist algorithmisch ohne größere Schwierigkeiten möglich. An diese Stelle sei für eine genauere Auseinandersetzung auf Carr et al [CK92] verwiesen.

4.1.3 IF-Inspektion

Wenn in einem Schleifenkörper eine IF-Anweisung vorhanden ist, die unnötige Berechnungen einspart, kann Streifen-Schneiden-Und-Vertauschen nicht sinnvoll zur Beschleunigung eingesetzt werden. Hier liegt die Idee nahe, sich zuerst die Bereiche zu berechnen, in denen die Berechnung durchgeführt werden muß und dann über die berechneten Bereiche mit dem eigentlichen Schleifenkörper zu iterieren. Die Schleifen, die die Bereiche ermitteln und die Schleifen, die über die Bereiche iterieren, können dann nach bekannten Verfahren optimiert werden. Eine Umsetzung dieser Grundidee findet man bei Carr et al [CK92].

4.2 Mögliche Spracherweiterung

Durch die verschiedenen vorgestellten Verfahren lassen sich viele, nicht geblockte Algorithmen in entsprechende Blockalgorithmen übernehmen. Dennoch gibt es Ausnahmen, in denen dies nicht möglich ist. Hier stellt sich die Frage, wie man Blockalgorithmen programmieren kann, die nicht maschinenabhängig optimiert sind und dennoch eine gute Performance erreichen. Hier scheint es nötig, die Algorithmen unabhängig vom Blockungsfaktor angeben zu können und diese vom Compiler ermitteln zu lassen beziehungsweise zur Laufzeit des Programms zu berechnen.

Es liegt nahe, für diesen Zweck die Programmiersprache zu erweitern. Der Vorschlag, zwei neue Schleifenkonstrukte einzuführen (siehe in [CK92]), erscheint nicht abwegig. Zuerst braucht man einen Konstrukt *BLOCK DO*, bei dem die Grenzen angegeben werden und das nach Übersetzung eine Schrittweite von *Blockgröße* verwendet. Das zweite benötigte Konstrukt *IN DO* entspricht der Schleife über die Region, die den geblockten Schleifenkörper enthält. Normalerweise werden hier keine Grenzen angegeben. Eine Schrittweite kann angegeben werden. Um Grenzen beeinflussen zu können, wird ein Konstrukt *LAST* eingeführt, das die aktuelle Obergrenze für den im Moment ausgeführten Block angibt. Der Compiler kann nun die *Blockgröße* durch Analysieren des entsprechenden *IN DO*-Block ermitteln und übersetzen.

Weiterhin erscheint es sinnvoll, ein Sprachkonstrukt einzuführen, das ausdrückt, daß die Ausführungsreihenfolge des Schleifenkörpers veränderbar ist. Damit könnte der Compiler auch bei komplexen Funktionen, die im Schleifenkörper vorhanden sein könnten, Veränderungen vornehmen, ohne Seiteneffekte befürchten zu müssen. Ein Problem stellt dabei die Anlayse vom Cacheverhalten der komplexen Funktionen dar.

5 Wahl eines Blockungsfaktors

Bisher wurde untersucht, wie man einen Algorithmus möglichst automatisiert in eine geblockte Variante überführen kann. Dabei wurde die Wahl des Blockungsfaktors vernachlässigt. Diesem Thema soll im

Folgenden nachgegangen werden.

5.1 Modellierung eines Caches nach Lam et al

Nach dem bisher gehörten soll durch die Blockung ein besseres Cacheverhalten erreicht werden. Dies drückt sich durch möglichst starke Lokalität und möglichst wenigen Cachmisses aus. Daraus folgt, daß eine Modellierung des Caches unter den Gesichtspunkte Cachemissrate sinnvoll ist. Wir verwenden dabei das Modell nach Lam et al [LRW91]. Die Annahme eines nicht-assoziativen Caches erleichtert das Modellieren erheblich. Bei einen n-fach assoziativen Cache wird man weniger Cachemisses betrachten, da ein Satz n verschiedene Cachelines aufnehmen kann, die bei einem nicht-assoziativen Cache auf die gleiche Cacheline fallen könnten. Dadurch werden wir mit dem nicht-assoziativen Cache einen schlechteren Fall beobachten, der uns dafür das Modellieren erheblich vereinfacht.

Zuerst werden wir einige Definitionen voranstellen, die dann zu Ausdrücken zusammengesetzt werden:

$D(v)$	Gesamtanzahl aller Zugriffe auf das Array v
$R(v)$	Wiederverwendungsfaktor für Zugriffe auf das Array v
$S(v)$	Ein Maß für die Anzahl der Zugriffe auf das Array v, die Selbstinterferenz erzeugen, ausgedrückt in Anteilen des Caches. Also ein Maß für die Anzahl von Zugriffen auf das Array v, die auf Cachepositionen von vorherigen Zugriffen auf das Array v abgebildet werden.
$F(u)$	Der *Footprint* ist der Anteil des Caches, der durch Zugriffe auf das Array u in einem Schleifendurchgang gefüllt wird. Der Footprint ist ein Maß für die Anzahl unterscheidbarer Cachepositionen für die Zugriffe auf Elemente des Array u.
V	Menge aller Arrays, auf die zugegriffen wird

Zuerst haben wir die unvermeidbaren Cachemisses auf ein Array v. Diese können mit der Anzahl aller Zugriffe auf das Array $D(v)$ und dem Wiederverwendungsfaktor $R(v)$ bestimmt werden. Dies ergibt dann $D(v)/R(v)$.

Zusätzlich müssen noch Cachemisses auf Grund von Interferenzen berücksichtigt werden. Wir führen dazu die Miss-Rate $M(v)$ ein. Sie gibt die Wahrscheinlichkeit für einen Cachemiss bei einen Zugriff auf das Array v an. Wir können für den konkreten, zu untersuchenden Algorithmus die Selbstinterferenz $S(v)$ und den Footprint der anderen Variablen $F(u)$ berechnen. Wenn die anderen Arrays $u \in V \backslash \{v\}$ $F(u)$ Anteile des Caches benutzen, dann drückt das Produkt der Gegenwahrscheinlichkeiten $\prod(1 - F(u))$ die Wahrscheinlichkeit für keine Fremdinterferenz aus. Die Wahrscheinlichkeit, daß keine Selbstinterferenz auftritt ist entsprechend $(1 - S(v))$. Nehmen wir alles zusammen und bilden die Gegenwahrscheinlichkeit, erhalten wir:

$$M(v) = 1 - (1 - S(v)) \prod_{u \in V \backslash \{v\}} (1 - F(u))$$

Mit Hilfe der Miss-Rate können wir dann die Anzahl der Cachemisses berechnen. Dazu nehmen wir die unvermeidbaren Cachemisses. Zusätzlichen berechnen wir die durch die Interferenz hervorgerufenen Cachemisses mit Hilfe der Miss-Rate. Beides zusammen ergibt dann die Anzahl der Cachemisses in diesem Modell:

$$\sharp Cachemisses = D(v) \left(\frac{1}{R(v)} + \frac{R(v) - 1}{R(v)} M(v) \right)$$

Die beiden Faktoren, die nach dem Modell beeinflußt werden können, um eine möglichst kleine Miss-Rate zu erreichen, sind der Footprint und die Selbstinterferenz. Bekommt man diese Werte klein, wird auch die Miss-Rate kleiner. Man versucht meist zuerst Selbstinterferenz zu vermeiden.

Verschiedene Untersuchungen mit Hilfe des Cachemodells und realen Experimenten haben ergeben, daß ein fester Blockungsfaktor für verschiedene Problemgrößen nur schlechte Ergebnisse gegenüber abhängig von der Problemgröße berechneten Blockungsfaktor erreicht. Der optimale Blockungsfaktor

schwankt vor allen bei quadratischen Blöcken bereits bei kleinen Änderungen der Problemgröße. Der Zeitaufwand zur Bestimmung des Blockungsfaktor fällt gegenüber dem Geschwindigkeitsgewinn nicht ins Gewicht.

Wir werden im Folgenden zwei verschiedene Algorithmen vorstellen, die beide versuchen, durch einen gut gewählten Blockungsfaktor Selbstinterferenz zu vermeiden und dabei den Block möglichst groß halten wollen. Das Verfahren nach Lam et al [LRW91] verwendet ausschließlich quadratische Blöcke im Gegensatz zu dem Verfahren von Coleman et al [CK95], das rechteckige Blöcke verwendet und dadurch bessere Ergebnisse erzielt.

5.2 Blockungsfaktor nach Lam et al

Der Ansatz von Lam et al [LRW91] versucht einen kritischen Blockfaktor B_0 zu finden, bei dem Selbstinterferenz vermieden wird. Dabei wird ein gegebenes Array auf den größtmöglichen, quadratischen Block untersucht, bei dem keine Selbstinterferenz auftritt.

Bei dem Beispiel der Matrixmultiplikation, wird die Matrix A von den beiden geblockten Schleifen getragen. Wir optimieren die Blockgröße demnach nach der Matrix A. Wir benötigen für das folgende noch die Cachegröße, die wir mit CS abkürzen werden.

Sei A ein quadratisches Array, N die Dimension von A und $\&A[i,j]$ die Adresse des Elementes i,j des Arrays A. Selbstinterferenz tritt genau dann auf, wenn zwei Elemente des Blocks des Arrays A auf die gleiche Cacheposition fallen. Dies ist genau dann der Fall, wenn die Adressen zweier Elemente modulo der Cachegröße das gleiche Ergebnis haben. Mathematisch ausgedrückt haben wir dann $\&A[i,j] \equiv_{mod\ CS} \&A[i',j']$. Dann gilt auch $\&A[i+a,j+b] \equiv_{mod\ CS} \&A[i'+a,j'+b]$ für alle a,b. Auf unsere Begebenheiten übersetzt heißt das, daß, gültige Indizes vorausgesetzt, auch die Elemente $\&A[i+a,j+b]$ und $\&A[i'+a,j'+b]$ Selbstinterferenz aufweisen. Von dieser Feststellung halten wir fest, daß sich ein gefundenes Selbstinterferenzmuster im Array wiederholt.

Mit Hilfe dieser Feststellung ist es möglich, einen maximalen Blockungsfaktor für einen quadratischen Block abhängig von der Cachegröße CS und der Arraydimension N zu ermitteln. Um die Arbeit zu vereinfachen, wird angenommen, daß die Anfangsadresse vom untersuchten Array A auf die Anfangsadresse vom Cache fällt, also $\&A[0,0] = 0$ gilt. Dies ist erlaubt, da im anderen Fall im Cache lediglich eine Verschiebung der Elemente des Arrays um $\&A[0,0]\ mod\ CS$ auftritt.

Man nimmt das Element $\&A[N/2,0]$ aus dem Array und prüft, ob Selbstinterferenz mit dem Element $\&A[N/2 \pm dj, di]$ auftritt. Tritt Selbstinterferenz auf, kann der Block maximal die Größe $max(\mid di \mid, \mid dj \mid)$ haben. Daraus ergibt sich der folgende Algorithmus in Abbildung 3.

```
int findB(int N, int CS)
{
    int addr, di, dj, maxWidth;

    maxWidth = min(N,CS);
    addr = N/2;
    while (1) {
        addr += CS;
        di = addr div N ;
        dj = abs((addr mod N) - N/2);
        if (di ≥ min(maxWidth, dj))
            return min(maxWidth, di);
        maxWidth = min(maxWidth, dj);
    }
}
```

Abbildung 3: Algorithmus zum Bestimmen von quadratischen Blöcken ohne Selbstinterferenz

Ein Beispiel kann dies verdeutlichen. Nehmen wir an, die Matrix habe die Dimension 400 und der Cache kann 1024 Elemente fassen. Dann ergeben sich folgende Werte:

addr	*di*	*dj*	$maxWidth$	$di * maxWidth$	
1224	3	176	176	528	
2248	5	48	48	240	
3272	8	128	48	384	
4296	10	96	48	480	N=400, CS=1024, B_0=23
5320	13	80	48	624	
6344	15	144	48	720	
7368	18	32	32	576	
8392	20	192	32	640	
9416	23	16	32	736	

Der Wert *di* gibt an, wieviele Zeilen in den Cache passen, wenn in einer Spalte Anzahl Schleifendurchläufe Zeilen passen, ohne mit einer anderen Schleife zu interferieren. Der Wert *dj* gibt gerade die Verschiebung für die nächste Iteration im Cache an. Der Wert $maxWidth$ gibt die aktuelle maximale Blockgröße an. Bei dem Vergleich muß *di* größer wie die Verschiebung im Cache sein, damit die weiteren Spalten in die entstandenen Lücken passen.

Der Algorithmus zur Berechnung vom Blockfaktor benötigt $O(N/\sqrt{CS})$ Zeitschritte.

Dieses Verfahren weist starke Schwankungen bei der Ausführungszeit der resultierenden Schleife auf. Der Grund liegt in der Beschränkung auf quadratische Blöcke. Dadurch werden unter Umständen Blockfaktoren der Größe 1 bis 6 gewählt. Im Vergleich dazu liegen Blockfaktoren normalerweise im zweistelligen Bereich. Im Durchschnitt erhält man zwar einen guten Geschwindigkeitsvorteil, der aber schon bei kleinen Schwankungen der Problem-Größe sehr gering werden kann. Dies läßt darauf schließen, daß man hier einen verfolgenswerten Ansatz vorfindet, der so nicht sinnvoll einsetzbar ist.

5.3 Blockungsfaktor nach Coleman et al

Bei dem Verfahren von Coleman et al [CK95] handelt es sich eigentlich nicht um die Bestimmung eines Blockfaktors, sondern um die Bestimmung eines rechteckigem Blocks. Deswegen ist zu berücksichtigen, wie das Array im Speicher abgespeichert ist. Bei diesem Algorithmus wird von einer Aneinanderreihung von Spalten innerhalb des Speichers ausgegangen. (Dies ist bei FORTRAN der Fall.)

Zuerst geht es wieder um die Vermeidung von Selbstinterferenz innerhalb eines Blocks. Der naive Ansatz nimmt alle Spalten, die in den Cache passen und nimmt diesen als einen Block. Nehmen wir wieder an, daß CS die Cachegröße sei, A die Matrix, die wir betrachten und N die Dimension der Matrix.

In der Abbildung 4 (a) wird die Abbildung des Blocks im Cache verdeutlicht. Der Block hat dabei die Größe $N \times (CS \text{ div } N)$. Beim Lesen einer weiteren Spalte würde Selbstinterferenz mit der ersten Spalten auftreten. Dadurch wird ein Reststück der Größe $r_1 = CS \ mod \ N$ vom Cache nicht verwandt.

Um die Möglichkeit zu haben, mehr Elemente im Cache halten zu können, müssen wir eine Spaltenlänge $\leq r_1$ wählen. Die Abbildung 4 (b) verdeutlicht die dadurch entstehenden Löcher. Wenn diese groß genug sind und die Verschiebung $setDiff = N - r_1$ günstig ausfällt, dann können in die Löcher weitere Spalten ohne Selbstinterferenz abgelegt werden. Dabei wurde der Cache genau mit einem Set gefüllt. Ein Set ist die Menge der Spalten, die zeitlich nacheinander in den Cache an größer werdenden Positionen abgelegt werden. Abbildung 4 (c) zeigt ein weiteres Set, mit dem die Löcher vom Cache wieder gefüllt werden. Es wird sooft mit einer Verschiebung *setDiff* durchgeführt, bis keine weitere Spalte in ein Loch paßt (Abbildung 4 (d)). Es entsteht wieder ein Reststück, daß gerade die Größe N mod r_1 hat. Dies setzt sich so weiter fort bis es keine Reststücke mehr gibt. Eine Betrachtung dieser Werte führt zu Spaltenbreite, die den Resten vom Algorithmus größten gemeinsamen Teiler von der Cachegröße und der Matrixgröße *ggT(CS,N)* entsprechen. Diese Reste sind potentielle Spaltenbreiten, für die wir die Spaltenanzahlen berechnen müssen. Daraus wählen wir später dann den endgültigen Block.

Wir betrachten nun eine Spalte der Breite N und beobachten, wieviele Spalten der Breite N in das entstandene Loch unter Berücksichtigung von der Verschiebung *setDiff* passen. Dazu definieren wir zuerst ein paar Begriffe und Größen.

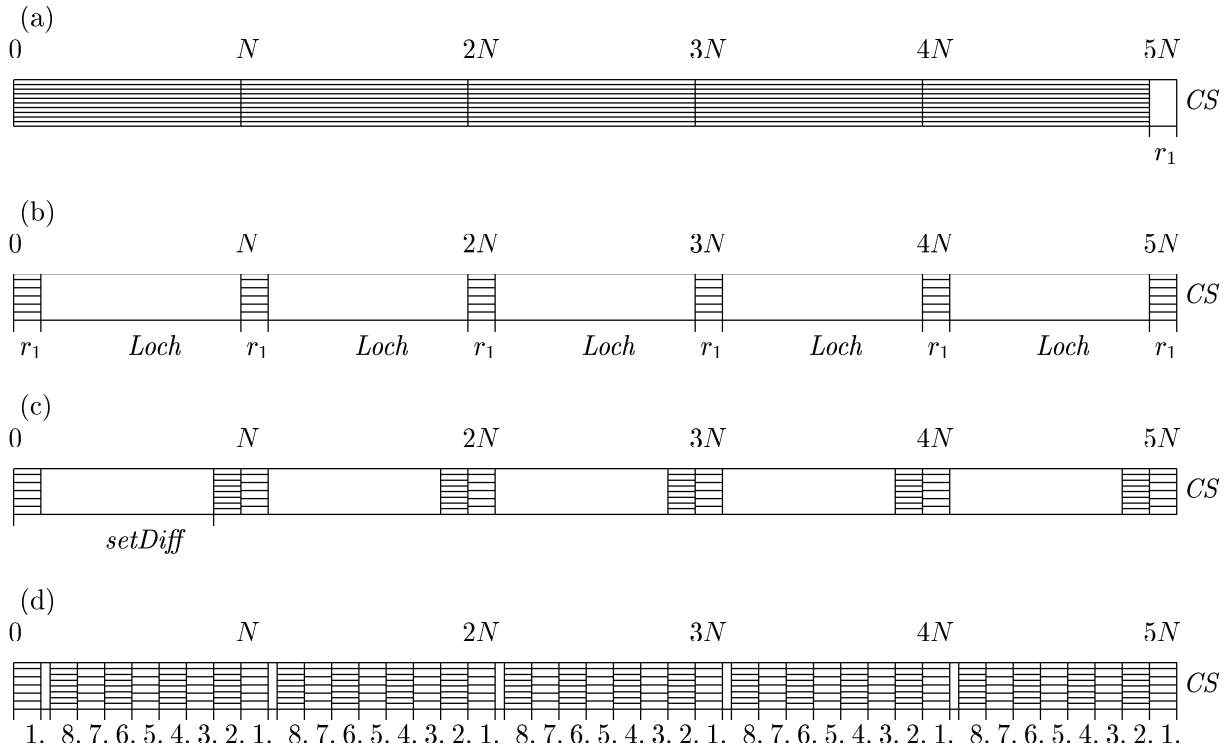


Abbildung 4: Abbildung der Arrays im Speicher bei einer Spaltenbreite von *(a)* N und von *(b)* r_1 *mit einem Set, (c) mit zwei Sets* bzw. *(c) nach acht Sets (mit Angabe des Sets)*

Die Spaltenbreite, zu der wir die Spaltenanzahl berechnen wollen, nennen wir *colSize*. Die Anzahl Spalten der Breite *colSize*, die in eine Verschiebung passen, nennen wir *colsPerSetDiff*. *colsPerSetDiff* berechnet sich aus der Verschiebung und der Spaltenbreite wie folgt:

$$colsPerSetDiff = setDiff \text{ div } colSize$$

Die Anzahl der Verschiebungen *setDiff*, die in eine Spalte N unterkommen, nennen wir $colsPerN$. Es gilt

$$colsPerN = N \text{ div } setDiff$$

Wenn wir die Werte *colsPerSetDiff* und $colsPerN$ multiplizieren, erhalten wir genau die Anzahl Spalten der Breite *colSize*, die in eine Spalte der Breite N unter Berücksichtigung von der Verschiebung *setDiff* passen.

Zusätzlich erhalten wir dabei weitere Löcher der Größe $gap = N \ mod \ setDiff$. Diese haben wir bisher nicht berücksichtigt, da wir bisher nur Spalten, die im *setDiff* Platz finden, berücksichtigt haben. In diese Löcher passen genau $colsPerGap = gap$ div $colSize$ weitere Spalten der Breite *colsize* hinein.

Damit haben wir jetzt die Beobachtung für die Spalte der Breite N beendet. Zusammengenommen passen $colsPerSetDiff * colsPerN + colsPerGap$ Spalten der Breite *colSize* in eine der ursprünglichen Spalten N. Ein Set hat genau $colsPerSet = CS$ div N Spalten der Breite N. Wenn wir das Reststück r_1 unbetrachtet lassen, kommen wir somit auf $(colsPerSetDiff * colsPerN + colsPerGap) * colsPerSet$ Spalten der Breite *colSize*.

Betrachten wir jetzt das Reststück r_1. Zuerst berechnen wir, wieviele Spalten der Breite $colSize$ mit Hilfe der Verschiebung *setDiff* in das Reststück passen. Dies sind genau

$$colsPerSetDiff * (r_1 \text{ div } setDiff)$$

Spalten der Breite *colSize*. Zusätzlich haben wir hier das gleiche Loch wie im vorigen Teil, in das entsprechend ebenfalls noch *colsPerGap* weitere Spalten passen.

Wir berücksichtigen noch den Sonderfall, daß gerade die Spaltenbreite N verwandt wird. Dann geben wir die Anzahl der Spalten zurück, die komplett in den Cache passen. Im Sonderfall, daß der erste Rest r_1 vom ggT(CS,N) verwandt wird, und *setDiff* kleiner als r_1 ist, paßt nur noch das Reststück zusätzlich in den Cache und es ergibt eine zusätzliche Spalte wie bei dem Ansatz, ganze Spalten abzulegen.

Zusammengenommen erhält man dann folgenden Algorithmus, der in Abbildung 5 abgedruckt ist.

```
int CS;                                  /* Cachegröße */
int N;                                   /* Zeilenanzahl */

int ComputeRows(int colSize)
{
    int colsPerSet = CS / N;
    int r1 = CS mod N;
    int setDiff = N-r1;
    int gap = N mod setDiff;
    int colsPerN = N / setDiff;

    int colsPerSetDiff, colsPerGap, rowSize;

    if (colSize=N)                       /* Für naiven Ansatz */
        return colsPerSet;
    if (colSize=r1 && colSize > setDiff)
        return colsPerSet+1;
    colsPerSetDiff = setDiff / colSize;
    colsPerGap = gap / colSize;
    rowSize = colsPerSetDiff * colsPerN * colsPerSet +
        colsPerGap * colsPerSet +
        colsPerSetDiff * (r1 / setDiff) +
        colsPerGap;
    return rowSize;
}
```

Abbildung 5: Algorithmus zum Bestimmen von Spaltenanzahlen zu gegebener Spaltenbreite, die ohne Interferenz im Cache abgebildet werden können.

Jetzt können wir für Spaltenbreiten die entsprechende Spaltenanzahl berechnen, wenn die Breiten Reste aus dem ggT-Algorithmus sind. Ein Durchprobieren aller Reste mit entsprechenden Spaltenanzahlen liegt nahe. Für eine Bewertung ist das Betrachten des Arbeitsbereiches $WSet$ hilfreich. $WSet$ ist im Allgemeinen die Summe aller Footprints der verwendeten Schleife bezüglicher aller Variablen, wie wir sie im vorigen Kapitel definiert haben. Dieser Arbeitsbereich sollte kleiner als die Cachegröße sein, da sonst auf alle Fälle Cachemisses auftreten. Andererseits versuchen wir den Wert zu maximieren. Ein weiteres Anliegen ist es, Fremdinterferenzen zu minimieren. Dazu betrachten wir die Fremdinterferenzen-Miss-Rate CIR. Diese Rate kann nur schwer berechnet werden. Wir werden sie im Algorithmus, der in der Abbildung 6 zu sehen ist, trotz dieser Unschönheit verwenden.

Das Berechnen der Spaltenanzahl für eine gegebene Spaltenbreite benötigt $O(1)$, das Ausführen des ggT-Algorithmus $O(log_2(CS))$. Insgesamt haben wir damit ein Zeitverhalten von $O(log_2(CS))$. Anzumerken bleibt, daß bei vielen Wertkombinationen die Berechnung von dem Algorithmus *ComputeRows* (siehe Abbildung 5) zu viele Zeilen berechnet. Dies war unschwer durch einen kleines Testprogramm zu bemerken.

Eine Möglichkeit, dies zu verhindern, ist es, das Cacheverhalten zu simulieren. Dazu belegt man den

Cache nacheinander mit Spalten der Breite *colSize* bis Interferenz auftritt und gibt die entsprechende Spaltenanzahl zurück (siehe Abbildung 7).

```
int CS;                                   /* Cachegröße */
int CLS;                                  /* Cachelinegröße */
int N;                                    /* Zeilenanzahl */
int M;                                    /* Spaltenanzahl */

int TTS(void)
{
   int bestCol = N;
   int oldCol = N;
   int bestRow = CS/N;
   int oldRow = CS/N;
   int colSize = CS mod N;

   int tmp;

   while (colSize>CLS &&
     oldCol mod colSize ≠ 0 &&
     rowSize<M) {
       rowSize = ComputeRows(colSize);
       if (colSize ≠ N)                   /* colSize an CLS anpassen */
           tmp=(colSize/CLS)*CLS;
       else
           tmp=colSize;
       if (WSet(tmp,rowSize) > WSet (bestCol, bestRow) &&
         WSet(tmp,rowSize) < CS &&
         CIR(tmp,rowSize) < CIR(bestCol,bestRow)) {
           bestCol = tmp;
           bestRow = rowSize;
       }
       tmp=colSize;
       colSize = oldCol mod colSize;      /* Nächster Rest von ggT(CS,N) */
       oldCol = tmp;
   }
   return (bestCol,bestRow);
}
```

Abbildung 6: Algorithmus zum Berechnen von rechteckigen Blöcken

Es ist zu beachten, daß der Algorithmus von Grutzeck zwar korrekt ist, dies aber einen erheblich größeren Zeitaufwand von $O(Spaltenbreite \cdot Spaltenanzahl)$ im Gegensatz zum Zeitaufwand von $O(1)$ bei *ComputeRows* erfordert. Der Fehler von *ComputeRows* führt nicht immer zu Fehlern beim endgültig verwandten Block, da dies nur ein möglicher Block ist, der mit anderen Blöcken verglichen wird.

Abschließend wollen wir die beiden Verfahren, die wir zur Ermittlung von Blockgrößen kennengelernt haben, gegenüberstellen. Die Nachteile der kleinen Blöcke, die beim Verfahren von Lam et al auftreten, treten bei diesen Verfahren nicht mehr auf. Durch das verwenden von rechteckigen Blöcken erhalten wir hier einen guten, gleichmäßigen Geschwindigkeitsvorteil. Untersuchungen haben gezeigt, daß die durchschnittliche Verbesserung der Geschwindigkeit durch Verringern der Cachemissrate um Faktor 14 gegenüber der ungeblockten Variante erwähnenswert ist. Dabei wurde ein Algorithmus festgestellt, bei dem eine Verbesserung bis zum Faktor 200 ermittelt wurde. Nimmt man diesen Algorithmus aus der

```
int GetRowSize (int CS, int N, int colSize)
{
  BOOL cache[CS];                    /* Bildet Cache nach */
  int i, j, pos;

  for (i=0; i<CS;++i)                /* Initialisieren */
    cache[i]=false;
  pos=0;                             /* Position im Cache */
  for (i=0; i<N; ++i) {              /* Schleife über Spaltenanzahl */
    for (j=0; j<colSize; ++j) {      /* Schleife über Spaltenbreite */
      if (cache[pos])                /* Test auf Interferenz */
        return i;
      cache[pos]=true;               /* Speichern der Position */
      pos=(pos+1) mod CS;            /* Nächste Position im Cache */
    }
    pos=(pos+N-colSize) mod CS;      /* Überspringe Loch */
  }
  return N;
}
```

Abbildung 7: Algorithmus nach Grutzeck zum Berechnen von Spaltenanzahlen, der richtige Ergebnisse liefert

Mittelwertbildung heraus, wurde die Cachemissrate um Faktor 2,5 verringert. Gegenüber dem Verfahren von Lam et al stellt dieser Algorithmus nach vorliegenden Untersuchungen stets eine Verbesserung dar.

6 Zusammenfassung und Ausblick

Wir haben die Verfahren Streifen-Schneiden-Und-Vertauschen, Schleifenbereiche Teilen und IF-Inspektion betrachtet, durch die manche Algorithmen in eine geblockte Formen überführt werden können. Implementierungen dieser Transformationen müssen zeigen, ob die Theorie auch sinnvoll in die Praxis umsetzbar ist.

Es wurde festgestellt, daß rechteckige Blöcke bessere Ergebnisse wie quadratische liefern können. Die Begrenzung der bisherige Algorithmen zur Ermittlung von Blöcken liegen in der Berücksichtigung von ausschließlich aufeinanderfolgende Zugriffen auf ein Array im Schleifenkörper. Hier ist Forschungsarbeit für komplexere Zugriffsmuster und Berücksichtigung von Zugriffen auf mehrere Arrays im Schleifenkörper nötig. Die Berechnung der Blockgrößen zur Programmlaufzeit führt zu Geschwindigkeitsvorteilen durch besser an die Problemgröße angepaßte Blockgrößen.

Es ist offensichtlich, daß Blockung nicht nur zur besseren Nutzung von Firstlevel-Caches verwendet werden kann. Es liegt nahe, daß Konzept auf Speicherhierarchien zu übertragen. Hierbei stellt sich die Frage, von was die Blockgröße unter solch einem Betrachtungwinkel abhängig ist. Bisherige Verfahren optimieren Blöcke für jede Hierarchiestufe einzeln. In Mitchell et al [MCFH97] wird durch verschiedene Beispiele beleuchtet, daß ein ganzheitlicher Ansatz bessere Ergebnisse erzielen kann. Die Gruppe von Forschern hat es sich zu ihrer Aufgabe gemacht, in diesem Bereich durch Kostenfunktionen, die die gesamte Speicherhierarchie betrachten, geeignetere Blockgrößen zu ermitteln.

Literatur

[CK92] Steve Carr und Ken Kennedy. Compiler blockability of numerical algorithms. In *The journal of Supercomputing*, Seiten 114-124, Minneapolis, MN, November 1992.

[LRW91] Monica S. Lam, Edward E. Rothberg and Michael E. Wolf. The cache performance and optimizations of blockes algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM SIGPLAN Notices 26(4), Santa Clara, CA, April 1991.

[CK95] Stephanie Colman und Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, ACM SIGPLAN Notices 30(6), Seiten 279-290, La Jolla, CA, Juni 1995.

[MCFH97] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, Karin Högstedt. Quantifying the Multi-Level Nature of Tiling Interactions. *10th International Workshop on Languages and Compilers for Parallel Computing (LCPC '97)*. August 1997.

Seminar - Ausarbeitung III

Philipp Bender

Rumpfteilen, Verschmelzen und Ausrollen von Schleifen

Zusammenfassung

Durch Schleifenrestrukturierungen kann die Ausführungszeit eines Programmes erheblich verkürzt werden. Dies kann einerseits durch eine bessere Ausnutzung des Prozessorcaches erreicht werden als auch durch das gleichzeitige Ausführen von Programmcode auf verschiedenen Prozessoren. In diesem Artikel werden das *Schleifenrumpfteilen*, das *Schleifenverschmelzen* und das *Unroll-and-jam* vorgestellt. Durch die erste Restrukturierungsmaßnahme wird sowohl die Datenlokalität innerhalb einer Schleife als auch die Parallelausführbarkeit der einzelnen Iterationen erhöht. Wie es gemacht wird und was für Probleme dabei auftreten können, zeigt der erste Abschnitt des Artikels. Da moderne Prozessoren mehrere Gleitkommaoperationen und Speicherzugriffe gleichzeitig ausführen können, ist es sinnvoll, das Verhältnis Anzahl der Speicherzugriffe zu Anzahl der Gleitkommaoperationen im Schleifenrumpf so anzupassen, daß die Prozessorleistung optimal ausgenutzt wird. Unroll-and-jam ist eine Möglichkeit dieses Verhältnis anzugleichen.

1 Einführung

Um ein Programm mit optimaler Geschwindigkeit auf einem Rechner laufen zu lassen, sollte es die ihm angebotene Rechnerarchitektur bestmöglich ausnutzen.

Moderne Mikroprozessoren unterstützen eine Speicherhierarchie, die meistens aus dem Hauptspeicher, den Caches und den Prozessorregistern besteht, um den Zugriff auf Daten zu beschleunigen. Der Code eines Programmes sollte so angelegt werden, daß zum Beispiel die während des Durchlaufens einer Schleife benötigten Daten fast vollständig im Cache gehalten werden. Somit fallen teure Hauptspeicherzugriffe weg.

Viele Computer sind außerdem auf den Mehrprozessorbetrieb ausgelegt, so daß sie verschiedene Befehle auf verschiedenen Prozessoren gleichzeitig ausführen können. Im Programm kann diese zum Beispiel dadurch ausgenutzt werden, daß es ermöglicht wird, die einzelnen Iterationen einer Schleife auf verschiedenen Prozessoren gleichzeitig abarbeiten zu lassen.

Wie bekommt man aber nun eine Schleife oder eine Menge von Schleifen in die gewünschte Form, um das letzte Stück Geschwindigkeit herauszukitzeln? Dieser Artikel beschreibt einige Schleifenrestrukturierungen, um sowohl die Parallelität der Ausführung zu erhöhen als auch den Cache so gut wie möglich auszunutzen. Experimente haben gezeigt, daß sich durch die vorgestellten Verfahren die Ausführungszeit einer Schleife um 20 bis 30% verringern läßt. Auch die Anzahl der Cachekonflikte während einer Schleifeniteration geht deutlich zurück.

Als erste Restrukturierung wird kurz das *Schleifenrumpfteilen (loop distribution)* behandelt, das geschachtelte Schleifen in eine günstige Form für das *Schleifenverschmelzen (loop fusion)* bringt. Beim Schleifenverschmelzen werden verschiedene Schleifen unter der Beachtung verschiedener Optimierungskriterien zu einer neuen zusammengefaßt. Hier werden dann verschiedene Verfahren und Algorithmen vorgestellt, die das gewünschte Ergebnis liefern. Anschließend folgt die *Unroll-and-jam* Schleifenrestrukturierung, die die Anweisungen einer Schleife an die Rechnerhardware anpaßt.

2 Begriffe

Bei einer *Programmtransformation* können die einzelnen Anweisungen innerhalb eines Programmes verschoben werden. Ihre Abarbeitungsreihenfolge bleibt aber erhalten. Im Gegensatz dazu kann sich bei einer *Programmrestrukturierung* die Reihenfolge ändern. Damit sich aber die Semantik des Programms nicht ändert, müssen die Abhängigkeiten zwischen den einzelnen Anweisungen erhalten bleiben.

Zwischen zwei Anweisungen besteht eine *Kontrollflußabhängigkeit*, wenn ihre Ausführungsreihenfolge zwingend festgelegt ist. Zwischen beiden existiert eine *Datenflußabhängigkeit*, falls es eine Kontrollflußabhängigkeit gibt und sie Daten verwenden, die an der gleichen Speicherstelle stehen. Bei den Datenflußabhängigkeiten spricht man von einer

- *Fluß-Abhängigkeit (true dependence)*, wenn die erste Anweisung an eine Speicherstelle einen Wert schreibt, der dann später von der zweiten gelesen wird.
- *Anti-Abhängigkeit (anti dependence)*, falls die erste Anweisung einen Wert liest, der von der zweiten später geschrieben wird.
- *Ausgabe-Abhängigkeit (output dependence)*, wenn die erste Anweisung einen Wert schreibt und die zweite diesen dann überschreibt.
- *Eingabe-Abhängigkeit (input dependence)*, wenn beide Anweisung nacheinander einen Wert von einer Speicherstelle einlesen.

Bei einer Programmrestrukturierung müssen alle Abhängigkeiten, außer der Eingabe-Abhängigkeit, erhalten bleiben, da sich sonst der Sinn des Programmes ändert.

Eine Abhängigkeit heißt *schleifengetragen (loop carried)* , wenn die Quelle und die Senke einer Abhängigkeit in verschiedenen Iterationen einer Schleife liegen. Liegt die Abhängigkeit innerhalb einer Iteration ist sie *schleifenunabhängig*. Die Anzahl der Durchläufe einer Schleife zwischen Quelle und Senke einer Abhängigkeit heißt *Abhängigkeitsdistanz d (dependence distance)*. Hat eine Abhängigkeit eine Distanz von null, ist sie folglich schleifenunabhängig.

Geschachtelte Schleifen werden von außen nach innen durchnumeriert und die Abhängigkeitsdistanzen als Vektor angegeben ($d = (d_1, d_2, ...)$), wobei d_n die Anzahl der Iterationen der n-ten Schleife zwischen Quelle und Senke einer Abhängigkeit angibt. Sind alle Einträge d_i mit $i < k$ im Vektor gleich null und ist d_k größer als null, dann ist die k-te Schleife der *Träger der Abhängigkeit*.

Eine Schleife heißt *parallel*, wenn sie keine schleifengetragenen Abhängigkeiten besitzt, und *sequentiell*, falls es mindestens eine gibt. Die einzelnen Iterationen einer parallelen Schleife können auf einem Mehrprozessorrechner dann gleichzeitig auf verschiedenen Prozessoren ausgeführt werden, was wiederum die Ausführungszeit verringert.

Für jede betrachtete Schleife wird ein Knoten in einem Graphen angelegt. Die gerichteten oder ungerichteten Kanten zwischen den Knoten repräsentieren die Abhängigkeiten zwischen den Schleifen. Der resultierende Graph ist immer ein DAG (directed acyclic graph), d.h. es existieren keine Abhängigkeitszyklen.

Eine Fluß-, Anti- oder Ausgabe-Abhängigkeit wird durch eine gerichtete Kante dargestellt. Da es bei Eingabe-Abhängigkeiten nicht auf eine Richtung ankommt, werden hier ungerichtete Kanten verwendet.

Abbildung 1 a) zeigt ein kleines, konstruiertes Beispiel mit sechs Schleifen $L1$ bis $L6$. Zwei Flußabhängigkeiten gibt es zwischen $L1$ und $L3$ ($a[i]$s und $b[i]$s). Ebenso eine zwischen $L2$ und $L6$ ($g[i]$ und $g[i]$). Da die beiden Schleifen $L2$ und $L6$ beide den Wert $r[i]$ verwenden, besteht hier eine Eingabe-Abhängigkeit. In Abbildung 1 b) sind alle weiteren Abhängigkeiten eingezeichnet.

3 Schleifenrumpfteilen

Beim *Schleifenrumpfteilen (loop distribution)* wird der Rumpf einer Schleife zerlegt und die einzelnen Teile in verschiedene neue Schleifen gesteckt. Im Beispiel kann die Schleife $L1$ in zwei neue Schleifen $L1.1$ und $L1.2$ geteilt werden.

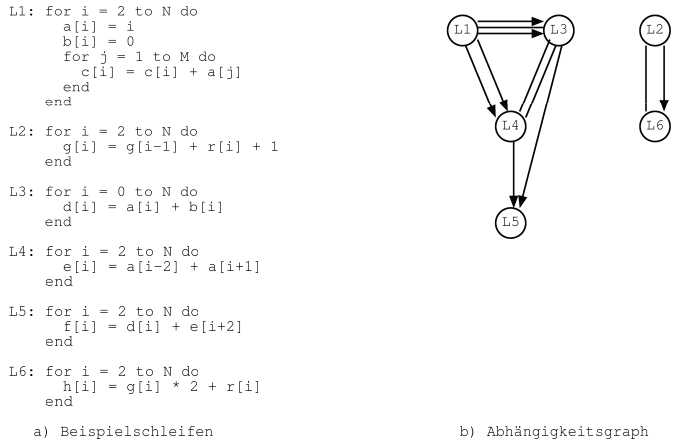


Abbildung 1: Schleifen und ihr Abhängigkeitsgraph

```
L1: for i = 2 to N do                          L1.1: for i = 2 to N do
      a[i] = i                                         a[i] = i
      b[i] = 0                  Rumpfteilen            b[i] = 0
      for j = 1 to N do            ===>              end
        c[i] = c[i] + a[j]                     L1.2: for i = 2 to N do
      end                                              for j = 1 to N do
    end                                                  c[i] = c[i] + a[j]
                                                     end
                                                   end
```

Das Schleifenrumpfteilen kann verwendet werden, um *perfekte Schleifenschachtelungen* zu erreichen. Eine perfekte Schachtelung hat die Form:

```
for i = ... do
  for j = ... do
    for n = ... do
     Anweisungen
    end
  end
end
```

d.h. es stehen keine weiteren Anweisungen zwischen zwei Schleifenköpfen. Viele Schleifenrestrukturierungen fordern perfekt geschachtelte Schleifen, um überhaupt anwendbar zu sein. Außerdem kann es zu einer besseren Ausnutzung des Caches kommen, da die Anweisungen innerhalb einer Schleife nun weniger geworden sind und somit auch weniger Variablen verwenden.

Aber ein Schleifenrumpf kann nicht beliebig aufgeteilt werden. Alle Anweisungen in starken Zusammenhangskomponenten im Abhängigkeitsgraphen des Schleifenrumpfes müssen nach dem Teilen auch wieder in einer Schleife liegen. Besteht zwischen zwei Anweisungen im Graphen eine Kante $A_1 \rightarrow A_2$, dann muß die Schleife, die A_1 aufnimmt, vor der Schleife, die A_2 aufnimmt, ausgeführt werden.

3.1 Ergebnisse

Gauss'sches Eliminationsverfahren. Dieses Beispiel aus [3] zeigt, daß Schleifenrestrukturierungen nicht isoliert gesehen werden dürfen. Das Programm ist in Fortran 77 geschrieben, wo Arrays spaltenweise gespeichert werden. Im ursprünglichen Algorithmus (siehe Abbildung 2) wird durch die J Schleife eine

Prozessor	Original	RT+V	Verbesserung
Sun Sparc2	12.23	5.22	57 %
Intel i860	8.35	2.63	69 %
IBM RS/6000	27.29	4.84	82 %

Tabelle 1: Geschwindigkeitsverbesserung beim Gauss-Algorithmus

effiziente Ausnutzung des Caches verhindert. Erst durch das Schleifenrumpfteilen und einem anschließenden Schleifenvertauschen wird ein Geschwindigkeitsgewinn erzielt. Tabelle 1 zeigt den Geschwindigkeitzuwachs bei einer 256x256 Matrix.

```
{KIJ Form}     =>  Rumpfteilen + Vertauschen  =>  {KJI Form}

DO K = 1,N                                  DO K = 1,N
 DO I = K+1,N                                DO I = K+1,N
  A(I,K) = A(I,K) / A(K,K)                    A(I,K) = A(I,K) / A(K,K)
  DO J = K+1,N                                DO J = K+1,N
   A(I,J) = A(I,J)-A(I,K)*A(K,J)               DO I = K+1,N
                                                A(I,J) = A(I,J)-A(I,K)*A(K,J)
```

Abbildung 2: Gauss-Algorithmus

4 Das Schleifenverschmelzen

Das Schleifenverschmelzen ist eine Restrukturierung, bei der mehrere Schleifen zu einer verschmolzen werden, d.h. die Rümpfe der beteiligten Schleifen werden in eine gepackt:

```
L1.1: for i = 2 to N do                          for i = 2 to N do
       a[i] = i                                    a[i] = i
       b[i] = 0              Verschmelzen          b[i] = 0
      end                        ==>               d[i] = a[i] + b[i]
L3':  for i = 2 to N do                          end
        d[i] = a[i] + b[i]
      end
```

Was bringt das Ganze nun? Zuerst einmal wird der Schleifenoverhead des Programmstücks reduziert. Es gibt nur noch eine Schleife, die einmal durchlaufen werden muß.

Außerdem sieht man am Beispiel, daß der Cache besser ausgenutzt wird. In der neuen Schleife werden $a[i]$ und $b[i]$ zuerst geschrieben und dann in der dritten Zuweisung gleich wiederverwendet. Dies macht es wahrscheinlich, daß sich die beiden Werte noch im Cache befinden und nicht aus dem Hauptspeicher geladen werden müssen. Bei den zwei ursprünglichen Schleifen liegen die Arrayzugriffe N Iterationen auseinander. Dort müssen die Werte aus dem Hauptspeicher geholt werden, wenn N groß ist und die gesamten Arrays nicht in den Cache passen.

Als dritten Vorteil ergibt sich die Tatsache, daß kein Synchronisieren mehr nach der ersten Schleife nötig ist. Werden die Iterationen der ursprünglichen Schleifen parallel ausgeführt, muß vor Beginn der zweiten Schleife gewartet werden, bis alle fertig ausgeführt sind. Im verschmolzenen Fall fällt das Warten weg.

Experimente haben gezeigt, daß das Verschmelzen zu einem Geschwindigkeitszuwachs von 30 oder mehr Prozent in der Ausführungsgeschwindigkeit führen kann.

4.1 Die Vorbedingungen

Nicht alle Schleifen können miteinander verschmolzen werden. Um das Verschmelzen zu ermöglichen, müssen die beteiligten Schleifen einige Vorbedingungen erfüllen.

Alle zu verschmelzenden Schleifen müssen über denselben Indexbereich laufen und die gleiche Laufvariable besitzen. Damit der Indexbereich paßt, kann man zuerst andere Schleifentransformationen mit der Schleife durchführen. Dies könnte zum Beispiel ein *Schleifenbereichsteilen (index-set-splitting)* oder ein *Schleifenschälen (loop peeling)* sein. Einen Überblick über die verschiedenen Transformationen liefert [2]. Beim Schleifenbereichsteilen wird nicht mehr der gesamte Indexbereich der ursprünglichen Schleife in einer Schleife durchlaufen, sondern in zwei oder mehr neuen Schleifen. Lief zum Beispiel die alte Schleife von 1 bis N, dann könnten die Indexbereiche der neuen Schleifen, die denselben Schleifenrumpf wie die Originalschleife besitzen, von 1 bis M und von $M+1$ bis N gehen. Das Schleifenschälen ist eine Sonderform des Schleifenbereichsteilens, bei der einzelne Iterationen aus einer Schleife herausgezogen werden. Die Namensgleichheit der Laufvariablen kann man durch Umbenennen erreichen. Aus der Schleife $L3$ des Beispiels kann man die ersten beiden Iterationen $d[0] = ...$ und $d[1] = ...$ herausziehen und vor die Schleife schreiben. Dann läuft auch sie von 2 bis N.

Beim Schleifenverschmelzen müssen auch alle Fluß-, Anti- und Ausgabe-Abhängigkeiten aus dem ursprünglichen Abhängigkeitsgraphen erhalten bleiben. Zwischen den zwei zu verschmelzenden Schleifen kann es

1. keine Abhängigkeiten oder
2. eine schleifengetragene Abhängigkeit geben, die von einer äußeren Schleife getragen wird, oder
3. eine schleifenunabhängige Abhängigkeit geben.

Im ersten Fall geht alles klar und die Verschmelzung kann durchgeführt werden, da es ja überhaupt keine zu beachtenden Abhängigkeiten gibt (siehe im Beispiel $L1$ und $L2$). Der zweite Fall macht auch keine Probleme, weil die Abhängigkeit nichts mit den beiden Schleifen zu tun hat. Nur beim dritten Fall muß man aufpassen. Hier kann aus der Abhängigkeit entweder

1. wieder eine schleifenunabhängige oder
2. eine vorwärts-schleifengetragene oder
3. eine rückwärts-schleifengetragene

Abhängigkeit werden. Bleibt sie schleifenunabhängig, kann verschmolzen werden (Im Beispiel: $a[i]$ aus $L1$ und $a[i]$ aus $L3$). Wird die Abhängigkeit vorwärts-schleifengetragen, gibt es auch kein Problem. Dies ist der Fall bei $a[i]$ aus $L1$ und $a[i-2]$ aus $L4$. Nur im letzten Fall wird eine unzulässige Abhängigkeit eingeführt. Hier wird eine Variable verwendet, bevor ihr der richtige Wert zugewiesen wurde oder umgekehrt. Dies trifft etwa auf $e[i]$ aus $L4$ und $e[i+2]$ aus $L5$ zu. Die ursprüngliche Abhängigkeit wird dann als *verschmelzungsverhindernd (fusion-preventing)* bezeichnet. Im Abhängigkeitsgraphen werden diese Kanten ungerichtet und durchgestrichen dargestellt.

Bei der Verwendung von Skalaren darf es keine erreichende Definition in einer anderen Schleife geben. Die Schleifen in Abbildung 3 dürfen nicht verschmolzen werden, da die Variable c ihren Wert erst in der letzten Iteration der ersten Schleife zugewiesen bekommt.

Vor der Schleifenverschmelzung kann noch ein Schleifenrumpfteilen durchgeführt werden, um alle Schleifenrümpfe in die kleinstmögliche Granularität zu zerteilen. Bei Verschmelzen kann es dann zu besseren Ergebnissen kommen.

Wurden all diese Fälle überprüft und die passenden Schleifen gefunden, steht einer Verschmelzung nichts mehr im Wege.

```
for j = 0 to M do
  c = a[j] * b[j]
  d[j] = 2 * c
end

for j = 0 to M do
  e[j] = e[j] + c
end
```

Abbildung 3: Zwei Schleifen, bei denen ein Verschmelzen nicht möglich ist

---

GreedyPartition (G)
EINGABE: $G = (V, E)$ ein DAG
AUSGABE: P Menge von Partition des DAG
ALGORITHMUS:
$P = \{\{n \epsilon V \mid \nexists\ m : (m,n) \epsilon E\}\}$
$V = V \backslash \{n \epsilon \mathrm{p} \mid \forall \mathrm{p} \epsilon P\}$
foreach $n\ \epsilon\ V$ in topologischer Ordnung **do**
  foreach $(m,n)\ \epsilon\ E$ und $\nexists\ \ q\ \epsilon\ p$ mit $p =$ Partition (m) und
    mit (q,n) nicht verschmelzungsverhindernd **do**
      $p = p \cup \{n\}$
    else
      $P = P \cup \{\{n\}\}$
    end
  end
end

Abbildung 4: Greedy-Algorithmus zur Zerlegung eines Graphen

4.2 Verschmelzen mit einem Greedy-Algorithmus

Ein einfacher Greedy-Algorithmus (siehe [3]) versucht, den Abhängigkeitsgraphen der zu verschmelzenden Schleifen so zu zerlegen, daß die Schleifen, die sich dann in einer Partition befinden, gefahrlos verschmolzen werden können. Abbildung 4 zeigt diesen Algorithmus.

Der Algorithmus legt zuerst alle Knoten ohne einen Vorgänger in eine gemeinsame Partition. Danach versucht er alle Nachfolger der topologischen Ordnung nach hinzuzufügen. Ein Knoten kann in die Partition eines Vorgängers eingefügt werden, wenn es keine verschmelzungsverhindernde Kante zwischen ihm und irgendeinem anderen Knoten der Partition gibt. Wenn ein Knoten keiner schon bestehenden Partition zugeordnet werden kann, wird eine neue Partition für ihn angelegt.

Der Algorithmus kann so implementiert werden, daß er eine Laufzeit von $O(\mid V \mid + \mid E \mid)$ hat. Er behandelt alle Knoten im Graphen gleich und erzeugt hierbei immer eine optimale Zerlegung, bei der die Anzahl der Partitionen minimiert und die Parallelität maximiert ist. Zum Beweis siehe [3].

Abbildung 5 zeigt eine gültige Zerlegung für das Beispiel, die mit dem Algorithmus erzeugt wurde.

Der Greedy-Algorithmus versucht möglichst viele Schleifen auf einmal zu verschmelzen. Daher kann es vorkommen, daß parallele und serielle Schleifen in einer Partition landen. Werden diese Schleifen dann verschmolzen, tauchen in der neuen Schleife die schleifengetragenen Abhängigkeiten der seriellen Schleifen wieder auf. Dies hat zur Folge, daß die neue Schleife auch seriell ist und Parallelität verloren geht. Im Beispiel ist $L2$ seriell und alle anderen Schleifen parallel. Die Zerlegung mit dem Greedy-Algorithmus legt $L1.1$, $L1.2$, $L2$, $L3$ und $L6$ in eine Partition und die durch Verschmelzen entstehende Schleife wäre seriell. Würde man nun $L2$ nicht mit den anderen verschmelzen, könnten die Iterationen der aus $L1.1$, $L1.2$, $L3$ und $L6$ entstandenen Schleife parallel ausgeführt werden. Unter diesem Gesichtspunkt wäre die gefundene Lösung des Greedy-Algorithmus nicht mehr optimal.

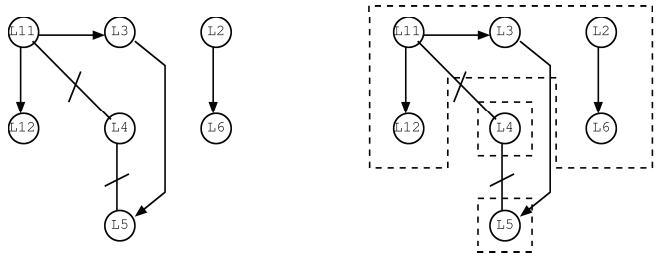


Abbildung 5: Abhängigkeitsgraph vor und nach dem Zerlegen

4.3 Verschmelzen mit zwei Graphen

Um zu gewährleisten, daß es nicht zu neuen Synchronisationsbarrieren kommt, kann man den ursprünglichen Graphen in zwei neue aufteilen, wobei der eine alle parallelen Schleifen und der andere alle sequentiellen enthält. Dann werden beide getrennt zerlegen und wieder zusammengebaut. So liegen dann nur Schleifen des gleichen Typs in einer Partition. Damit ergibt sich nun folgender Algorithmus (siehe [3]):

1. Erzeuge einen Graphen G_p aus dem ursprünglichen Graphen G_0, der alle Knoten der parallelen Schleifen und alle Kanten zwischen diesen Knoten enthält.

2. Füge in G_p verschmelzungsverhindernde Kanten ein, die denen in G_0 entsprechen aber in G_p nicht vorkommen.

3. Erzeuge aus G_p mit dem Greedy-Algorithmus den Graphen $G_{p'}$ und übertrage die Zerlegung auf G_0, indem die ursprünglichen Knoten wie in $G_{p'}$ zusammengefaßt werden. Dies ergibt den Graphen $G_{0'}$.

4. Erzeuge einen Graphen G_s aus dem Graphen $G_{0'}$, der alle seriellen Knoten und Kanten zwischen seriellen Knoten enthält und füge wieder die verschmelzungsverhindernden Kanten ein, die es dort noch nicht gibt.

5. Zerlege G_s mit dem Greedy-Algorithmus und übertrage die Partitionierung auf $G_{0'}$. Dies ergibt dann den Lösungsgraphen.

Ein einfacher Weg, um die neuen verschmelzungsverhindernden Kanten in den beiden Teilgraphen zu berechnen, ist es die transitive Hülle G_{tr} von G_0 zu bestimmen und erst dann den Teilgraphen zu erzeugen. Neue verschmelzungsverhindernde Kanten in den Teilgraphen werden nur benötigt, wenn es im ursprünglichen Graphen einen Pfad zwischen zwei Knoten gibt, auf dem ein Knoten des anderen Typs liegt.

Die einzelnen Teilgraphen können dann mit dem **GreedyPartition** Algorithmus zerlegt werden.

Das Rückübertragen der Zerlegungen vom Teilgraphen eines Typs auf den gesamten Graphen erfolgt dadurch, daß in G_0 einfach die Knoten zusammengefaßt werden, die auch im Teilgraphen in einer Partition liegen.

Der Algorithmus hat eine Laufzeit von $O(\mid V \mid * \mid E \mid)$ und erzeugt optimale Teilgraphen. Der zusammengesetzte Lösungsgraph muß nicht immer optimal zerlegt sein. Der Beweis hierzu findet sich in [3].

4.4 Verbesserung der Datenlokalität

Um eine bessere Ausnutzung des Caches zu erreichen, werden nun auch die Eingabe-Abhängigkeiten im DAG betrachtet. Diese interessieren nun, da die Zeit für das mehrmalige direkt hintereinander liegende Einlesen eines Wertes verringert wird, wenn sich der Wert im Cache befindet.

Zusätzlich werden jetzt alle Kanten im Graph bewertet. Das Gewicht einer Kante gibt an, wie hoch der Wiederverwendungsnutzen ist, wenn die zwei zugehörigen Knoten verschmolzen werden. Zur Vereinfachung werden alle Fluß- und Eingabe-Abhängigkeiten mit eins gewichtet, alle Ausgabe- und Anti-Abhängigkeiten mit null. Alle verschmelzungsverhindernden Kanten haben einen Wert $\epsilon << 1$. Gibt es zwischen zwei Knoten mehr als eine Kante, so werden diese Kanten zu einer zusammengefaßt und mit der Summe der Gewicht bewertet. War eine der Kanten gerichtet, ist die neue Kante auch gerichtet, sonst ist sie ungerichtet.

Maximale Datenlokalität wird nun erreicht, wenn nach dem Zerlegen die Summe der Gewichte von Kanten, die ihre Endpunkte in verschiedenen Partitionen haben, minimal ist.

Das Problem, beim Schleifenverschmelzen maximale Datenlokalität zu erreichen, ist NP-vollständig, wie in [3] bewiesen. Es kann polynomial auf das *Multiway-Cut-Problem*, dessen NP-Vollständigkeit in [7] gezeigt wird, reduziert werden. Beim Multiway-Cut-Problem sind ein Graph $G = (V, E)$ und eine Menge $S = \{s_1, s_2, ..., s_k\}$ von speziellen Knoten $s_i \; \epsilon \; V$ gegeben, die im folgenden *Terminale* genannt werden. Alle Kanten des Graphs sind ungerichtet und bekommen das Gewicht $w(e) = 1$. Nun soll eine Teilmenge E' von Kanten aus E gefunden werden, bei der die Summe über alle Kantengewichte minimal ist und deren Entfernen aus dem Graphen bewirkt, daß es keine Pfade mehr zwischen zwei beliebigen Terminalen gibt.

Mit dem folgenden Algorithmus, der eine polynomiale Laufzeit besitzt, kann nun das Multiway-Cut-Problem auf ein Verschmelzungsproblem abgebildet werden und umgekehrt. Ohne Beschränkung der Allgemeinheit werden Kanten von Eingabe-Abhängigkeiten nicht betrachtet.

1. Wähle einen beliebigen Terminalkonten $s \; \epsilon \; V$ und bezeichne ihn als Startknoten. Er erhält die Nummer 1. In die Menge *Bereit* kommen nun alle Nachfolger von s. Dann wird ein Nicht-Terminal aus *Bereit* ausgewählt oder ein Terminal, falls *Bereit* nur Terminale enthält. Der ausgewählte Konten bekommt die nächste noch nicht vergebene Nummer, wird aus *Bereit* entfernt und alle seine Nachfolger, die noch keine Nummer haben und nicht in *Bereit* sind, werden in *Bereit* aufgenommen. Es werden solange Knoten ausgewählt bis alle eine Nummer besitzen.

2. Der Graph G wird nun in einen DAG $G1$ für das Verschmelzungsproblem umgebaut. Alle Knoten aus G stellen in $G1$ Schleifenknoten dar und alle Kanten aus G bekommen in $G1$ eine Richtung, so

daß die Kante vom Knoten mit der kleineren Nummer zu dem mit der größeren Nummer zeigt. Diese Kanten, von denen keine verschmelzungsverhindernd ist, werden mit dem Wiederverwendungsnutzen bewertet.

3. Seien $\{s_1, s_2, ..., s_k\}$ die Terminale aus dem Multiway-Cut-Problem nach größerwerdenden Nummern sortiert. In den DAG des Verschmelzungsproblems werden folgende verschmelzungsverhindernde Kanten eingefügt:

$$(s_1, s_2), ..., (s_1, s_k), ..., (s_j, s_{j+1}), ..., (s_j, s_k), ...(s_{k-1}, s_k)$$

Die Korrektheit des Konstruktionsalgorithmus wird in [3] bewiesen.

Das konstruierte Verschmelzungsproblem hat eine Lösung mit minimalem Gewicht genau dann, wenn das zugehörige Multiway-Cut-Problem die gleiche Lösung besitzt. Die Lösung des Verschmelzungs- bzw. Multiway-Cut-Problems gibt dann mit E' diejenigen Kanten an, deren Endpunkte jeweils in einer anderen Partition liegen. Die Endpunkte aller anderen Kanten liegen innerhalb einer Partition, womit der Graph wieder partitioniert ist.

Mit Hilfe eines heuristischen Ansatzes kann man sich aber dem Optimum annähern. Der DAG wird wieder mit dem **GreedyPartition** Algorithmus in Partitionen eingeteilt. Gibt es nun in der entstandenen Zerlegung zwei Partitionen g_1 und g_2 mit einer gerichteten Kante (g_1, g_2) dazwischen, so kann keiner der Knoten aus g_2 nach g_1 verschoben werden. Dies hätte sonst der Algorithmus getan. Aber trotzdem kann das Verschieben sinnvoll sein, wenn es *sicher* und *gewinnbringend* ist. Ein Knoten $n \in g_l$ kann genau dann nach g_h verschoben werden wenn, er keine Nachfolger in g_l besitzt und es keine verschmelzungsverhindernde Kante (n, r) gibt mit entweder $r \in g_h$ oder $r \in g_k$, wobei g_k ein Vorgänger von g_h ist. Gewinnbringend ist es, wenn der Knoten verschoben werden kann und innerhalb der neuen Partition die Summe der Kantengewichte höher ist als die Summe der Kantengewichte innerhalb der alten Partition des Knotens.

Der Algorithmus **ImprovePartition** (Abbildung 6) überprüft nun diese Kriterien und verschiebt eine Knoten, wenn sich dadurch der Wiederverwendungsnutzen erhöht.

ImprovePartition (G_g)
EINGABE: G_g ein greedy partitionierter Graph
AUSGABE: $G_{g'}$ eine Partitionierung mit verbesserter Cacheausnutzung
ALGORITHMUS:
foreach Partitionen $g_l \in G_g$ in umgekehrter topologischer Sortierung **do**
  foreach Knoten $n \in g_l$ in umgekehrter topologischer Sortierung **do**
    Wähle g_h mit $(n, p) \in G_g$ und $p \in g_h$
    if es ist sicher und profitabel n nach g_h zu verschieben **then**
      verschiebe n nach g_h
    end
  end
end

Abbildung 6: Einfacher Algorithmus, um die Cacheausnutzung zu verbessern

Wie der Greedy-Algorithmus hat auch **ImprovePartition** einen linearen Aufwand.

4.5 Verbesserung der Vorbedingungen

Normalerweise könnten die Schleifen $L1$, $L4$ und $L5$ im Beispiel in Abbildung 1 nicht verschmolzen werden.

Nach dem Verschmelzen entstehen rückwärts-schleifengetragene Abhängigkeiten sowohl zwischen $a[i]$ und $a[i+1]$ als auch zwischen $e[i]$ und $e[i+2]$.

Um verschmelzungsverhindernde Abhängigkeit wieder loszuwerden, kann ein einfaches Verfahren angewandt werden, um ein Verschmelzen doch noch zu ermöglichen.

Ziel ist es, die Abhängigkeiten der Schleifen so umzubauen, daß keine rückwärts-schleifengetragene Abhängigkeit mehr entstehen [5]. Durch ein Verschieben des Iterationsraum der Schleifen gegeneinander kann dies erreicht werden. Einzige Voraussetzung ist eine konstante Abhängigkeitsdistanz.

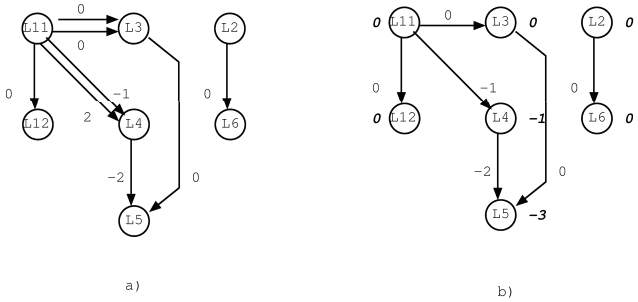


Abbildung 7: Berechnung der Verschiebung

Die Kanten im DAG werden nun mit den Abhängigkeitsdistanzen bewertet (Abb. 7 a)). Neu entstehende rückwärts-schleifengetragene Abhängigkeiten erhalten negative Werte. Gibt es zwischen zwei Schleifen mehrere Abhängigkeiten, werden die Kanten zu einer zusammengefaßt und das Minimum der Gewichte als Bewertung verwendet. Eine negative Kante gibt nun den Verschiebungsfaktor des Iterationsraums der Schleife an, die die Senke der Abhängigkeit enthält.

Nun kann es vorkommen, daß es mehrere negative Kanten im Graphen gibt. Um nun alle Verschiebungsfaktoren korrekt zu berechnen, werden sie mit Algorithmus **TraverseDependenceChainGraph** (Abbildung 8) relativ zur Wurzel ermittelt. Der Algorithmus weist jedem Knoten einen Wert zu, der dann den absoluten Verschiebungsfaktor angibt (Abb. 7 b)).

TraverseDependenceChainGraph (G)
EINGABE: $G(V, E)$ der ursprüngliche Graph
AUSGABE: G' mit gewichteten Knoten
ALGORITHMUS:
foreach $v \epsilon V$ **do**
    $gewicht(v) = 0$
end
foreach $v \epsilon V$ in topologischer Sortierung **do**
    foreach $e = (v, v_c) \epsilon E$ **do**
        if $gewicht(e) < 0$ **then**
            $gewicht(v_c) = \min(gewicht(v_c), gewicht(v) + gewicht(e))$
        end
    end
end

Abbildung 8: Algorithmus, um die Verschiebungsfaktoren zu berechnen

Hat man alle Verschiebungswerte, müssen die Schleifen transformiert werden, um die Verschmelzung legal zu machen.

Ein einfacher Ansatz ist es alle Ausdrücke, in denen Induktionsvariablen vorkommen, anzupassen, indem einfach der Wert der Variablen um den Verschiebungsfaktor geändert wird. Zum Schluß müssen noch die Schleifenränder angepaßt werden. Das Ergebnis sieht man in Abbildung 9 a).

Eine weitere Methode, um die Rümpfe zu verschieben, ist das *Streifenschneiden (strip-mine)*. Hierbei werden die Iterationsräume der ursprünglichen Schleifen in Streifen zu einer Länge von je s geschnitten

```
for i = 2 to 4 do
  a[i] = i
  b[i] = 0
end
for i = 2 to 3 do
  e[i] = a[i-2] + a[i+1]
end

...

for i = 5 to N do
  a[i]   = i
  b[i]   = 0
  e[i-1] = a[i-3] + a[i]
  f[i-3] = d[i-3] + e[i-1]
end

...

e[N] = a[N-2] + a[N+1]
for i = N-2 to N do
  f[i] = d[i] + e[i+2]
end
```

a) Einfaches Verschmelzen

```
for j = 1 to N in s do
  for i = max ( j,2 ) to min ( j + s - 1, N ) do
    a[i] = i
    b[i] = 0
  end
  for i = max ( j - 1, 2 ) to min ( j + s - 2, N ) do
    e[i] = a[i-2] + a[i+1]
  end
  for i = max ( j - 3, 2 ) to min ( j + s - 4, N ) do
    f[i] = d[i] + e[i+2]
  end
end

e[N] = a[N-2] + a[N+1]
for i = N-2 to N do
  f[i] = d[i] + e[i+2]
end
```

b) Verschmelzen mit Streifenschneiden

Abbildung 9: Verschmelzung mit Indexverschiebung

und in eine neue Schleife gepackt. Die neue äußere Schleife läuft dann mit einer Schrittweite von s. Die Iterationsräume der inneren Schleifen können nun um den Verschiebungsfaktor verschoben werden, indem einfach die Indexgrenzen angepaßt werden. Heißt die Laufvariable der äußeren Schleife i, so hat die neue Untergrenze einer inneren Schleife den Wert $\max(i - shift, alte\_untergrenze)$ und die Obergrenze den Wert $\min(i+s-1-shift, alte\_obergrenze)$. *Schleifenschälen* muß dann nur noch am Ende der Iterationen durchgeführt werden. Wie dies für das Beispiel aussieht, zeigt Abbildung 9 b).

Die durch Streifenschneiden erzeugten Schleifen sehen nicht viel besser als das Original aus. Doch mit Hilfe einer Cachepartitionierung wird die Datenlokalität erhöht.

4.6 Cachepartitionierung

Angenommen bei den Schleifen $L1.1$ und $L3$ wurde eine Schleifenverschmelzung durchgeführt.

```
for i = 1 to N do
  a[i] = i
  b[i] = 0
end                        ==>
                       Verschmelzen
for i = 1 to N do
  d[i] = a[i] + b[i]
end
```

```
for i = 1 to N do
  a[i] = i
  b[i] = 0
  d[i] = a[i] + b[i]
end
```

Abbildung 10: Beispiel zum Cachepartitionieren

In der neuen Schleife wird der Cache besser ausgenutzt als in den ursprünglichen zwei Schleifen wegen der sofortigen Wiederverwendung von $a[i]$; könnte man meinen. Was passiert aber bei einem nicht-assoziativen Cache, wenn zufällig die Adressen von $a[i]$ und $b[i]$ auf dieselbe Cacheline abgebildet werden? Dann hat das Verschmelzen in Bezug auf den Cache nichts gebracht, da in der Zuweisung an $b[i]$ die Werte der Cacheline, in der sich $a[i]$ im Cache befindet, überschrieben wurden. Auch die nachfolgenden Wert des Arrays a, die sich in der Cacheline befanden und in den nächsten Iterationen wiederverwendet werden könnten, sind weg.

Es kann also möglich sein, daß durch Cachekonflikte der verschiedenen Arrays die durch die Schleifenrestrukturierungen erzeugten Datenlokalitätsgewinne wieder zunichte gemacht werden. Diese Effekte treten auf, wenn Bereiche von verschiedenen Arrays auf die gleiche Cacheadresse abgebildet werden und somit alte Werte überschreiben.

Da innerhalb des Schleifenrumpfes und in den nächsten Iterationen einer Schleife meist nebeneinander liegende Arraywerte verwendet werden, scheint es sinnvoll zu sein, den Cache in einzelne zusam-

menhängende Stück zu zerteilen und jedem Array eines dieser Stücke zuzuordnen [5]. Während der Schleifendurchläufe kann dann dieser Bereich im Cache wandern, aber es kommt niemals zu Überlappungen (Abb.11).

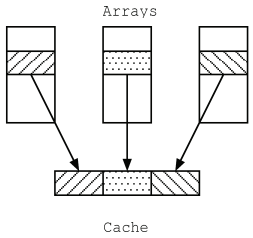


Abbildung 11: Cachepartitionierung

Damit jeder Array zu Beginn in seine Partition abgebildet wird, muß seine Startadresse im Hauptspeicher verschoben werden. Zwischen den einzelnen Arrays können Lücken freigelassen werden, um das richtige Mapping zu erreichen. Diese Speicherbereiche sind natürlich nutzloser Verschnitt, der möglichst klein gehalten werden sollte.

Der Algorithmus **GreedyMemoryLayout** (Abbildung 12) versucht nun diesen Versatz möglichst gering zu halten. Es wird von einem nicht-assoziativen Cache mit der Adressenabbildungsfunktion *CacheMap* ausgegangen. Der Einfachheit halber werden nur eindimensionale Arrays berücksichtigt. Eine Lösung für eine mehrdimensionale Einteilung des Caches wird in [5] beschrieben.

Zu Beginn wird der Cache in n_a gleichgroße Stücke der Länge s_p eingeteilt, wobei n_a der Anzahl der zu berücksichtigten Arrays entspricht. Jedem Array wird nun eine Partition zugewiesen, so daß die Lücke zwischen der Endadresse des zuletzt plazierten Arrays und der benötigten Hauptspeicheradresse für die nächste Partition möglichst klein ist. Jede so zugewiesene Partition wird danach aus der Liste der freien Partitionen P entfernt, damit es nicht vorkommt, daß zwei Arrays die gleiche Partition zugewiesen wird. Der Algorithmus hat einen Aufwand von $O(n_a^2)$.

Durch diese Aufteilung des Caches wird zugleich die Größe eines Streifens beim oben erwähnten Streifenschneiden festgelegt. Diese Größe muß so gewählt werden, daß die gesamten Arrayelement, die in einer inneren Schleife bei einem Durchlauf verwendet werden, in die Partition des Arrays im Cache passen. Wird die Streifenlänge zu groß gewählt, kommt es wieder zu Cachekonflikten, da die Partitionsgrenzen nicht eingehalten werden.

In der Seminarausarbeitung IV wird genauer auf das Abbilden eines Arrays im Speicher eingegangen.

4.7 Ergebnisse

Beispielschleifen. Abbildung 13 zeigt das Verschmelzen der Beispielschleifen. Zuerst wurde durch Schleifenschälen der Laufindex von $L3$ angepaßt und dann mit dem Greedy-Algorithmus der Abhängigkeitsgraph zerlegt. Zum Schluß wurden die verschmelzungsverhindernden Kanten mittels Schleifenschälens beseitigt. Die Datenlokalität wurde erhöht, da nun die meisten Variablen in der gleichen Iteration der Schleife wiederverwendet werden.

Erlebacher Das Schleifenverschmelzen wurde in [3] mit dem in Fortran geschriebenen Programm *Erlebacher* getestet. *Erlebacher* ist ein ADI Benchmark Programm mit 835 Codezeilen, das einen dreidimensionalen Array (50x50x50) benutzt, um ein Gleichungssystem zu lösen. Im Quellcode gibt es 26 zum Teil geschachtelte Schleifen, deren Rumpf meist nur aus einer Anweisung besteht. Insgesamt ergeben sich drei Abhängigkeitsgraphen, von denen zwei mit dem Greedy-Algorithmus optimal zerlegt wurden. Bei einem Graphen wurde zusätzlich der Algorithmus mit den gewichteten Kanten verwendet, um die Cacheausnutzung zu verbessern. Tabelle 2 zeigt die Geschwindigkeitsverbesserungen, die sich jeweils auf den verschiedenen Rechnern ergaben. Die Ausführungszeiten verringerten sich nach dem Schleifenverschmelzen um etwa 4 und 17 Prozent.

LL18 und calc. Um die Auswirkungen des Indexverschiebens und der Cachepartitionierung zu untersuchen, wurden beide Verfahren auf einem KSR2 Mehrprozessorsystem getestet [5]. Als Programm

GreedyMemoryLayout (A, c)
EINGABE: A Menge von Arrays
c Cachegröße
AUSGABE: $START(a)$ enthält die Startspeicheradresse des Arrays
ALGORITHMUS:
// Anzahl der Arrays
$n_a = \mid A \mid$
// Größe einer Partition
$s_p = \frac{c}{n_a}$
// Freie Partitionen
$P = \{0, 1, \cdots, n_a - 1\}$
// Startadresse im Hauptspeicher
$q = q_0$
do
Wähle $a \epsilon A$ beliebig
$mapped\_address = CacheMap\ (q)$
foreach $p \epsilon P$ **do**
$target\_address\ (p) = p * s_p$
$gap\ (p) = target\_address\ (p)$ - $mapped\_address$
if $target\_address\ (p) < mapped\_address$ **then**
$gap(p) = gap(p) + c$
end
end
Wähle $p_{opt} \epsilon P$ mit $gap(p_{opt}) = \min_{p \epsilon P}(gap(p))$
$P = P \backslash \{p_{opt}\}$
$START(a) = q + gap(p_{opt})$
$q = START(a) + SIZE(a)$
$A = A \backslash \{a\}$
while $A \neq \emptyset$

Abbildung 12: Layoutalgorithmus für das Cachepartitionieren

```
for i = 2 to N do
  g[i] = g[i-1] + r[i] + 1
end

for i = 0 to 1 do
  d[i] = a[i] + b[i]
end

for k = 2 to N in s do
  for i = max ( k, 2 ) to min ( k + s - 1, N ) do
    a[i] = i
    b[i] = 0
    for j = 1 to M do
      c[i] = c[i] + a[j]
    end
    g[i] = g[i] + r[i] + 1
    d[i] = a[i] + b[i]
    h[i] = g[i] * 2 + r[i]
  end
  for i = max ( k - 1, 2 ) to min ( k + s - 2, N ) do
    e[i-1] = a[i-3] + a[i]
  end
  for i = max ( k - 3, 2 ) to min ( k + s - 4, N ) do
    f[i-3] = d[i-3] + e[i-1]
  end
end

e[N] = a[N-2] + a[N+1]
for i = N-2 to N do
  f[i] = d[i] + e[i+2]
end
```

Abbildung 13: Verschmelzung aller Beispielschleifen

Erlebacher	Original	verschmolzen	Verbesserung
IBM RS/6000	0.813s	0.672s	17.34%
Intel i860	0.548s	0.518s	5.47%
Sun Sparc2	0.400s	0.383s	4.25%

Tabelle 2: Auswirkungen der Schleifenverschmelzung

wurden drei geschachtelte Schleifen aus den Livermore Loops der Kernel 18 (LLK18), einem Programm zur Berechnung von Hydrodynamikgleichungen, verwendet. Die Schleifen können nicht direkt verschmolzen werden, da es verschmelzungsverhindernde Abhängigkeiten gibt. Als zweites Programm wurden fünf geschachtelte Schleifen aus einem Ozeansimulationsprogramm verwendet (calc).

In beiden Fällen verringerten sich durch Cachepartitionierung die Cachekonflikte um gut 30%. Dabei entstand nur ein Speicherverschnitt von etwa 2%. Das Schleifenverschmelzen bracht beim LLK18 einen Geschwindigkeitszuwachs von 7% bis 15%, wenn weniger als 32 Prozessoren verwendet wurden. Bei calc waren es 11% bis 20% bei unter 24 Prozessoren. Bei mehr Prozessoren machte es sich bemerkbar, daß die Parallelität durch das Schleifenschälen, den Schleifenoverhead und das Cachepartitionieren verloren ging.

Die vorgestellten Verfahren sind nicht zu aufwendig, um in einen Übersetzer eingebaut zu werden. Wie die Beispiele zeigen, reduziert sich die Ausführungszeit der Programme in den meisten Fällen.

Bei Fortran-90 Programmen gibt es Anweisungen, die komplette Arrays auf einmal verarbeiten können. Während des Übersetzens werden dieses Anweisungen in Schleifen umgesetzt, deren Rumpf meist nur aus einer Zuweisung besteht. Es ergeben sich also geeignete Kandidaten für das Schleifenverschmelzen.

Beim zweiten Beispiel zeigt sich, daß ein Verschmelzen nicht immer ein Programm schneller macht. Je mehr Prozessoren zur Ausführung verwendet werden, desto eher bleiben Daten im Prozessorcache. Wird das Programm auf zu vielen Prozessoren ausgeführt, kann sich das Verschmelzen negativ bemerkbar machen. Auch das Cachepartitionieren funktioniert auf Mehrprozessorrechnern nicht so leicht, wie oben angegeben, da jeder Prozessor seinen eigenen Cache besitzt. Hier muß man dann die Daten geeignete Aufteilen, um den Datenaustausch zwischen den einzelnen Prozessoren zu minimieren. Voraussetzung sind natürlich Kenntnisse über den Hardwareaufbau.

Aber auch auf Einprozessorsystemen kann das Cachepartitionieren schiefgehen, wenn zum Beispiel innerhalb einer Schleifeniteration Werte eines Arrays verwendet werden, die zu weit auseinander liegen. Passen zum Beispiel in eine Cachepartition n Werte des Arrays a und werden in einer Iteration die Werte $a[i]$ bis $a[i + 2 * n]$ verwendet, gibt es wieder Cachekonflikte.

5 Unroll-and-jam

Unroll-and-jam läßt sich auf geschachtelte Schleifen anwenden. Wie der Name schon sagt, wird zuerst eine äußere Schleife ausgerollt und danach die entstehenden inneren Schleifen wieder verschmolzen. Das Verfahren kann auf die $L1.2$ Schleife im Beispiel angewandt werden.

```
for i = 1 to N do          Unroll+Jam     for i = 1 to N in 2 do
 for j = 1 to M do          =======>        for j = 1 to M do
  c[i] = c[i] + a[j]                          c[i]   = c[i]   + a[j]
 end                                          c[i+1] = c[i+1] + a[j]
end                                           end
                                             end
```

Hier wurde die äußere Schleife um den Faktor eins aufgerollt und dann die beiden entstehenden inneren Schleifen wieder zusammengefaßt.

Da es sich beim Ausrollen um eine Schleifentransformation handelt, ist sie immer legal. Beim Verschmelzen müssen dann die Schleifen die entsprechenden Vorbedingungen erfüllen.

Unroll-and-jam kann nun verwendet werden, um das Schleifengleichgewicht dem Rechnergleichgewicht anzupassen [6].

5.1 Rechnergleichgewicht

Heutige Prozessoren können meistens mehrere Maschinenbefehle parallel nebeneinander und somit gleichzeitig ausführen. Um eine Rechner gleichmäßig auszulasten und keine Rechenzeit zu verschwenden, sollten die verschiedenen Berechnungseinheit im Prozessor alle gleich ausgelastet sein. In diesem Abschnitt wird nun angenommen, daß es dem Prozessor möglich ist, Gleitkommabefehle und Speicherzugriffe gleichzeitig auszuführen, da beide Operationstypen zeitintensiv sind.

Ein Rechner ist im *Gleichgewicht*, wenn er nun jeweils Gleitkommaberechnungen und Speicherzugriffe nacheinander in Spitzengeschwindigkeit ausführen kann. Um das Ganze mathematisch auszudrücken wird der Wert β_M definiert als das Verhältnis der Datenrate M_M, mit der Daten aus dem Speicher geladen werden können, in Wörtern pro Takt, zur Ausführungsgeschwindigkeit von Gleitkommaoperationen F_M in FLOPS pro Takt. Beide Werte sollen jeweils die Spitzenwerte sein und die Größe eines Datenworts im Speicher der Größe eines Arguments bei der Gleitkommaberechnung entsprechen.

$$\beta_M = \frac{M_M}{F_M} \tag{1}$$

5.2 Schleifengleichgewicht

Jetzt liegt es nahe, für eine Schleife einen ähnlichen Wert zu definieren, der dann mit β_M verglichen werden kann und in derselben Größenordnung liegen sollte. Der Gleichgewichtswert einer Schleife ergibt sich mit

$$\beta_L = \frac{M_L}{F_L} \tag{2}$$

M_L gibt die Anzahl der Speicheroperationen im Schleifenrumpf und F_L die Anzahl der Gleitkommaberechnungen an. Speicherzugriffe werden alle gleich bewertet, egal ob der Wert im Cache oder im Hauptspeicher steht, da angenommen wird, daß der Übersetzer die Datenlokalität schon optimiert hat.

Ein Vergleich beider Maßzahlen β_M und β_L zeigt, wie gut die Schleife den Prozessor ausnutzt. Liegt der Wert für die Schleife über dem des Prozessors, kann der Prozessor die Daten nicht schnell genug heranschaffen. Es ist daher erstrebenswert den Abstand zwischen beiden möglichst gering zu halten. Unroll-and-jam kann dazu verwendet werden, beide Werte anzugleichen.

Im Beispiel hat die ursprüngliche innere Schleife in $L1.2$ einen Gleichgewichtsfaktor von 1. $c[i]$ kann in einem Register gehalten werden; $a[j]$ wird bei jeder Iteration geladen und es wird eine Gleitkommaoperation ausgeführt ($\beta_L = \frac{1}{1}$). Die neue innere Schleife besitzt einen Faktor von 0,5. $c[i]$ und $c[i+1]$ stehen in Registern; $a[j]$ wird einmal in ein Register geladen und es kommen zwei Gleitkommaoperationen vor. ($\beta_L = \frac{1}{2}$). Kann ein Prozessor zwei Gleitkommabefehle und einen Speicherzugriff gleichzeitig ausführen, wurde der Abstand von β_M und β_L verringert.

5.3 Zielsetzung beim Unroll-and-jam

Wie das Beispiel zeigt, kann Unroll-and-jam die beiden Gleichgewichtswerte einander anpassen. Unroll-and-jam kann mehr Gleitkommaoperationen in eine Schleife bringen, ohne die Anzahl der Speicherzugriffe proportional zu erhöhen.

Allgemein sollte Unroll-and-jam von folgender Regel geleitet werden:

$$\begin{aligned} Zielfunktion: &\quad \min \mid \beta_L - \beta_M \mid_0 \\ Nebenbedingung: &\quad R_L <= R_M \end{aligned}$$

Reihenfolge	Array Größe	Iterationen	Original	UJ	Verbesserung
JIK	50x50	500	4.53s	2.37s	1.91
	500x500	1	135.61s	44.16s	3.07
JKI	50x50	500	6.71s	3.3s	2.04
	500x500	1	15.49s	6.6s	2.35

Tabelle 3: Verbesserungen bei der Matrixmultiplikation

R_L gibt den Registerbedarf der Schleife an und R_M die Anzahl der tatsächlich vorhandenen Register.

Die Zielfunktion versucht nun den Abstand zwischen den beiden Gleichgewichten so gering wie möglich zu halten unter der Bedingung, daß die Register optimal ausgenutzt werden. Die Lösung des Problems ist dann rechnerunabhängig und kann bei Bedarf durch ändern der β_M und R_M Werte angepaßt werden. Die Werte von β_L und R_L können mit einem Aufwand von $O(n)$ errechnet werden, wobei n die Größe des Abhängigkeitsgraphen ist. Die Zielfunktion kann dann mit $O(\mid R_M \mid)$ gelöst werden. [6] zeigt ein Beispiel, wie Unroll-and-jam in einem Übersetzer implementiert werden kann.

5.4 Ergebnisse

Matrixmultiplikation. In [6] wurde anhand einer Matrixmultiplikation Unroll-and-jam getestet.

```
DO J = 1,N
  DO I = 1,N
    DO K = 1,N
     C[I,J] = C[I,J] + A[I,K] * B[K,J]
```

Das Beispiel ist in Fortran geschrieben und zeigt einerseits den Geschwindigkeitsgewinn durch Unroll-and-jam und andererseits wie wichtig eine optimale Cacheausnutzung sein kann (Tabelle 3). Da in Fortran mehrdimensionale Array spaltenweise gespeichert werden, kommt es in der ursprünglichen Schleifenanordnung (JIK) zu Cachekonflikten. Vertauscht man aber die I und die K Schleife, werden die Ausführungszeiten deutlich geringer. Dies zeigt sich besonders bei großen Matrizen.

Auch bei anderen Programmen zeigte sich ein Geschwindigkeitszuwachs um den Faktor 2 bis 5 durch den Einsatz von Unroll-and-jam. Zu den Testprogrammen gehörten Programme für die Lineare Algebra, der NAS Kernel aus dem SPEC-Benchmark und Fortran Anwendungen.

Die Experiment zu Unroll-and-jam wurden auf einer IBM RS/6000 mit einem Übersetzer compiliert, der den Programmcode schon von sich aus stark optimiert. Daher ist es zu erwarten, daß mit anderen Übersetzer noch weitere Geschwindigkeitssteigerungen zu erzielen wären.

Bei der Berechnung der Gleichgewichte wird angenommen, daß alle Prozessorbefehle eines Typs gleich viele Takte benötigen. In der Praxis können aber besonders bei den Gleitkommaoperationen gewaltige Unterschiede auftreten, wodurch der Gleichgewichtswert des Rechners nicht einheitlich ist. In den Experimenten mit Schleifen, bei denen Gleitkommadivisionen im Rumpf vorkamen, war der Geschwindigkeitsgewinn durch Unroll-and-jam nicht so groß, wie bei Experimenten mit “normalen” Schleifen, da Divisionsbefehle meistens viele Takte zur Ausführung benötigen.

6 Zusammenfassung

Die vorgestellten Verfahren und Algorithmen haben gezeigt, daß sich die Ausführungszeit eines Programmes erheblich verkürzen läßt, wenn man den Programmcode optimiert. Diese gilt besonders für Schleifen innerhalb eines Programmes, die durch Schleifentransformationen und -restrukturierungen der Rechnerarchitektur angepaßt werden können. Die einzelnen Verfahren dürfen dabei nicht isoliert betrachtet werden, da ein Verfahren meist die Voraussetzungen für ein anderes schafft.

Sowohl das Schleifenverschmelzen als auch Unroll-and-jam lassen sich in einen Übersetzer einbauen, da die verwendeten Algorithmen meist eine lineare Laufzeit besitzen. Durch das Nachschalten weiterer

Verfahren wie etwa dem Cachepartitionieren läßt sich die Cacheausnutzung und damit die Ausführungsgeschwindigkeit weiter erhöhen.

Allgemein stellt sich bei dem Versuch, die Geschwindigkeit eines Programmes zu erhöhen, die Frage, welche Restrukturierungen überhaupt eingesetzt werden können bzw. eingesetzt werden sollten. Alle Möglichkeiten durchzuprobieren ist sicherlich zu aufwendig. Bei größeren Programmen könnte man heuristische Verfahren verwenden, um zuerst einmal die geeigneten Schleifen für bestimmte Maßnahmen herauszusuchen und dann die Restrukturierungen durchzuführen.

Literatur

[1] A.V. Aho, R. Sethi und J.D. Ullman. *Compilers.* Addison-Wesley, 1996.

[2] Michael Philippsen. *Ausgewählte Kapitel aus dem Übersetzerbau.* Skript zur Vorlesung, Universität Karlsruhe, Sommersemester 1996

[3] K. Kennedy und K. McKinley. *Maximizing loop parallelism and improving data locality via loop fusion and distribution.* In *Proceeding of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Nummer 768 in LNCS, Seiten 301-320, Portland, OR, August 1993. Springer.

[4] S.K. Singhai und K.S. McKinley. *Loop fusion for parallelism and locality.* In *Mid-Atlantic States Student Workshop on Programming Languages and Systems (MASPLAS'96)*, April 1996.

[5] N.Mankjikian und T. Abdelrahman. *Fusion of loops for parallelism and locality.* In *Proceedings of the 24th International Conference on Parallel Processing*, Band II, Seiten 19-28, Oconomowoc, WI, August 1995.

[6] S. Carr. *Memory Hierarchy Management.* Dissertation, Department of Computer Science, Rice University, 1994.

[7] E. Dahlhaus, D.S. Johnson, C.H. Papadimitriou, P.D. Seymour, M.Yannakakis. *The complexity of multiway cuts.* In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, Mai 1992.

Seminar - Ausarbeitung IV

Thorwald C. Franke

Cache-Optimierung durch Speicherabbildung

Zusammenfassung

Die Diskrepanz zwischen der Leistungsfähigkeit heutiger Prozessoren und der Langsamkeit des Hauptspeichers macht den Cache-Speicher als Zwischenglied zu einem kritischen Faktor der Leistungsfähigkeit eines Gesamtsystems. Ein Weg, den Cache-Speicher besser auszunutzen, ist die Berücksichtigung der Abbildung der Daten des Hauptspeichers auf den Cache-Speicher. Durch die Auffächerung von Reihungen, die geeignete Anordnung frequentierter Prozeduren im Hauptspeicher und die Ausnutzung der Informationen über die Zugriffsmuster parallelisierter Programme kann das Leistungspotential von Cache-Speichern besser ausgeschöpft werden, wie Simulations- und Messergebnisse aufzeigen.

1 Einleitung

Die immer größer werdende Differenz zwischen der Geschwindigkeit des Prozessors und der des Hauptspeichers führt zu einer immer größer werdenden Bedeutung des Cache-Speichers. Seine bestmögliche Ausnutzung wird zu einem kritischen Faktor der Leistungsfähigkeit heutiger DV-Systeme. Um die Leistung des Cache-Speichers zu verbessern, lassen sich zwei Hauptwege unterscheiden: Die Verbesserung der Hardware oder die optimale Ausnutzung der Hardware durch die Software. Insbesondere auf dem Gebiet intelligenter Software-Konstruktion liegen noch brauchbare Verbesserungspotentiale brach. Hier gibt es wiederum grundsätzlich zwei Wege: Schleifentransformation oder die Berücksichtigung der Abbildung des Hauptspeichers auf den Cache-Speicher. In dieser Arbeit werden drei verschiedene Wege beschrieben, wie über Speicherabbildung eine bessere Ausnutzung des Cache-Speichers erreicht werden kann.

Das Grundproblem der Cache-Optimierung ist die Vermeidung von Fehlzugriffen auf den Cache-Speicher. Diese entstehen wesentlich dadurch, dass mehrere aus dem Hauptspeicher benötigte Datenblöcke auf dieselbe Stelle im Cache-Speicher abgebildet werden, und sich so gegenseitig verdrängen. Die grundsätzliche Lösungsidee, die allen drei Ansätzen gemein ist, besteht darin, die Abbildung der Daten aus dem Hauptspeicher in den Cache auf die verschiedensten Weisen zu berücksichtigen. Um Missverständnisse zu vermeiden, sei deutlich darauf hingewiesen, dass es *nicht* darum geht, die Abbildung vom Hauptspeicher auf den Cache-Speicher zu verändern. Diese bleibt starr bestehen, wird aber berücksichtigt, indem man die Daten im Hauptspeicher – unter dem starr bleibenden Abbildungsmuster – geschickt anordnet, um auf diese Weise deren Abbildung auf den Cache-Speicher zu beeinflussen.

Die drei zu beschreibenden Techniken lassen sich folgendermaßen charakterisieren: Die Cache-Optimierungs-Technik der Reihungsauffächerung versucht, die geeignet dimensionierten Blöcke geblockter Schleifen derart im Hauptspeicher anzuordnen, dass die Anzahl der Cache-Fehlzugriffe reduziert wird. Die farborientierte Priorisierung frequentierter Prozeduren führt zu einer Cache-optimierenden Anordnung von häufig aufgerufenen Prozeduren im Hauptspeicher. Die übersetzergesteuerte Seitenanordnungsstrategie für Multiprozessorsysteme beeinflusst die Strategie des Betriebssystems, den virtuellen Adressraum derart auf den Hauptspeicher abzubilden, dass die über den virtuellen Adressraum im Hauptspeicher lokalisierten Daten von dort geeignet auf den Cache-Speicher abgebildet werden, entsprechend dem Zugriffsmuster eines konkreten parallelisierten Programmes.

1.1 Konventionen und Definitionen

Im Mittelpunkt der Betrachtung stehen direktabgebildete Cache-Speicher, bei denen jede Stelle des Hauptspeichers fest einer Stelle im Cache-Speicher zugeordnet ist. Dies ist sinnvoll, da direktabgebildete bzw. gering assoziative Cache-Speicher wegen ihrer kürzeren Zugriffszeit häufig verwendet werden. Direktabgebildete Cache-Speicher finden sich z.B. in den Rechnern HP 735, DEC AXP 3000, SGI Indigo2 und SUN Sparc5.

Die Qualität der Cache-Optimierung erweist sich an der Behandlung rechenintensiver Programme. Hierzu zählen insbesondere numerische Anwendungen mit ihren umfangreichen Matrizenberechnungen. Deshalb dienen als beispeilhafte Datenstruktur stets zweidimensionale Reihungen, deren Zugriff Cache-optimal zu gestalten ist.

Cache-Zeile (*cache line*):
Der Cache-Speicher ist in mehrere unabhängig voneinander belegbare Stücke eingeteilt. Ein solches Stück wird Cache-Zeile genannt.

Kachel oder Block (*tile*):
Rechteckiger Ausschnitt einer zweidimensionalen Reihung. Bei der Blockung einer Schleife über einer Reihung wird diese in Kacheln eingeteilt.

Fehlzugriff (*cache miss*):
Erfolgt ein Zugriff auf ein Datum des Hauptspeichers, dessen Seite sich nicht im Cache-Speicher befindet, so muss diese Seite unter hohem Zeitaufwand in den Cache-Speicher geladen werden. Dieser Vorgang wird Fehlzugriff genannt. Fehlzugriffe unterteilen sich in folgende drei Arten:

- Primärer Fehlzugriff (*compulsory cache miss*):
  Ein primärer Fehlzugriff liegt dann vor, wenn eine Seite zum ersten mal seit Programmstart in den Cache-Speicher geladen werden muss. Er ist unvermeidbar.
- Cache-Konflikt (*cache conflict*):
  Ein Cache-Konflikt liegt dann vor, wenn zwei zeitlich lokal wiederholt benötigte Seiten des Hauptspeichers auf dieselbe Cache-Zeile abgebildet werden und sich so gegenseitig verdrängen. Diese Definition macht nur bei nicht voll-assoziativen Cache-Speichern Sinn.
  - Innen- und Aussenkonflikt (*self-/cross-interference*):
    Im Zusammenhang mit gekachelten Reihungen kann man zwei Arten von Cache-Konflikten unterscheiden: Innen-Konflikte sind Konflikte innerhalb derselben Kachel. Sie treten auf, wenn die Zugriffsmenge einer Kachel nicht vollständig in den Cache-Speicher passt. Aussenkonflikte sind Konflikte zwischen verschiedenen Kacheln derselben oder verschiedener Reihungen.
- Kapazitätsbedingter Fehlzugriff (*capacity cache miss*):
  Ein kapazitätsbedingter Fehlzugriff liegt dann vor, wenn eine bereits geladene Cache-Zeile vor einer wiederholten Anforderung aus Platzgründen verdrängt wurde. Eine solche Definition ist nur bei assoziativen Cache-Speichern sinnvoll. In einem direktabgebildeten Cache-Speicher dagegen treten nur Cache-Konflikte, aber keine kapazitätsbedingten Fehlzugriffe auf. Insofern eine Speicherabbildungsstrategie auch einen direktabgebildeten Cache-Speicher optimal auszunutzen sucht, kann man allerdings auch hier von kapazitätsbedingten Fehlzugriffen sprechen.

2 Datenauffächerung

Die folgenden Ausführungen referieren die Arbeit von P.R. Panda, H. Nakamura, N.D. Dutt und A. Nicolau aus dem Jahre 1997 ([1]).

2.1 Grundlagen und Idee

Die zwei wichtigsten Ursachen für Cache-Fehlzugriffe sind die begrenzte Kapazität eines assoziativen Caches und das Auftreten von Cache-Konflikten bei nicht voll-assoziativen Cache-Speichern.

Das Problem der begrenzten Kapazität lässt sich dadurch in den Griff bekommen, dass man Reihungen, die nicht vollständig in den Cache-Speicher passen, in kleinere Teile zerlegt. Dies führt zu einer Blockung der Schleifen über den Reihungen bzw. zu einer Kachelung der Reihungen selbst. Dieser Teil der Arbeit von P.R. Panda et al. soll an dieser Stelle nicht referiert werden, es wird aber eine Optimierung der Kachelgröße für das folgende vorausgesetzt.

Eine erprobte Methode zur Reduktion von Cache-Innen-Konflikten ist die Auffächerung von Daten im Hauptspeicher, die dafür sorgt, dass zugleich benötigte Daten nicht auf dieselbe Cache-Zeile abgebildet werden. Es wird eine Datenausrichtungstechnik (DAT) vorgestellt, die auf der Auffächerung der Daten im Hauptspeicher basiert.

Die beiden Schritte – Kachelgrößenbestimmung und Auffächerung – werden dabei getrennt durchgeführt, um ein höheres Maß an Flexibilität zu erzielen. Der Umgang mit mehreren Reihungen wird berücksichtigt.

2.2 Ein einführendes Beispiel

Um die Probleme und Ideen an einem Beispiel zu verdeutlichen, betrachte man den häufig angewandten Algorithmus der Schnellen Fourier-Transformation (*Fast Fourier Transform,* FFT). Es soll hier nicht um das Verständnis des FFT- Algorithmus gehen, sondern um seine Struktur, die sich aus mehreren ineinander geschachtelten Schleifen zusammensetzt. In der innersten Schleife wird zweimal auf jeweils zwei Stellen in derselben Reihung (hier: sigreal) in verschränkter Reihenfolge zugegriffen:

```
...
... := sigreal[i];
... := sigreal[i + 2^n];
sigreal[i]          := ...;
sigreal[i + 2^n]    := ...;
...
```

Veranschaulicht man sich das Zugriffsmuster anhand einer Matrix, so ergibt sich der bekannte "Schmetterling", das charakteristische Zugriffsmuster der Schnellen Fourier-Transformation.

Angenommen, die Reihung habe eine Länge von 2048 Worten, der Cache-Speicher habe eine Kapazität von 512 Worten und eine Cache-Zeile habe eine Länge von 4 Worten. Dies bedeutet, dass für n=9 ($2^9 = 512$) stets Cache-Konflikte auftreten. Anschaulich wird dies in Abb. 1 (oben).

Eine Lösung des Problems bietet die Auffächerung der sigreal-Reihung im Cache-Speicher: Indem man nach jeweils 512 Worten eine Verschiebung der folgenden Daten um 4 Worte vornimmt, werden die Zugriffsstellen auf den Hauptspeicher derart versetzt, dass es zu keinen Cache-Konflikten mehr kommt. Veranschaulicht wird dieser Vorgang in Abb. 1 (unten). Nach [1] ergibt sich dadurch eine Beschleunigung um 15 % auf einem SunSparc5-Rechner. Dies bedeutet zudem, dass der für die Auffächerung zusätzlich im Algorithmus benötigte Rechenaufwand klein ist gegenüber dem Gewinn an Rechenzeit.

2.3 Fächerungsgrößenberechnung

Wie schon gesagt wird im folgenden davon ausgegangen, dass eine geeignete Berechnung der Kachelgröße bereits durchgeführt wurde. Die Berechnung der Kachelgröße ignoriert bewusst die Möglichkeit von Innen-Konflikten innerhalb ein und derselben Kachel. Dieses Problem wird nun durch die Auffächerung der Reihung im Hauptspeicher behoben. Abb. 2 veranschaulicht die Situation an einem Beispiel.

Es wird von einem 1024elementigen Cache-Speicher ausgegangen, in den eine in 30 x 30 - Kacheln zerlegte 256 x 256 - Reihung abgebildet wird. Abb. 2 a) zeigt, dass die Zeilen 1 und 5 einer Kachel auf dieselbe Cache-Zeile abgebildet werden, sich also in einem Innen-Konflikt befinden. Fächert man die

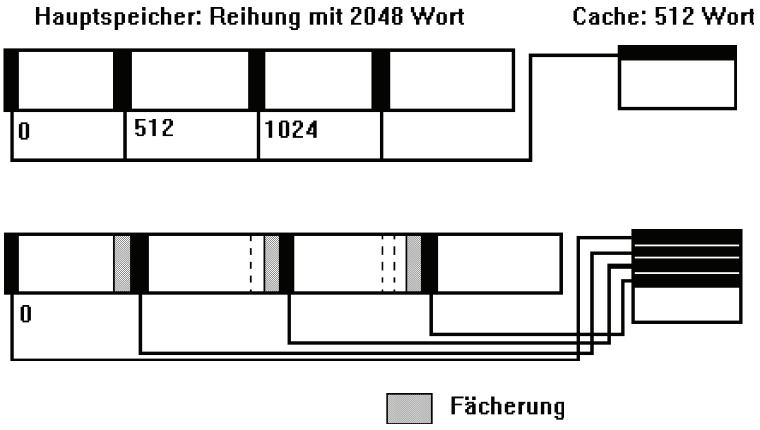


Abbildung 1: Oben: Konflikte im Original Unten: Konfliktfrei durch Auffächerung

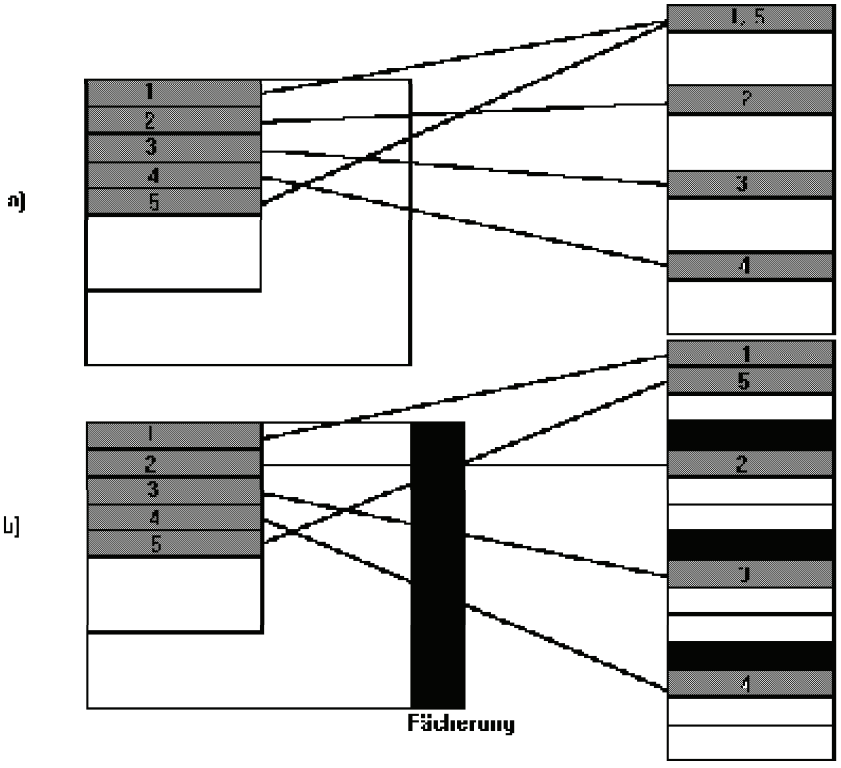


Abbildung 2: Innenkonflikt (a) und dessen Bereinigung (b)

Kachel um 8 Elemente nach jeder Zeile auf, so wird die fünfte Zeile gerade neben die erste Zeile abgebildet, der Konflikt ist behoben; siehe Abb. 2 b).

Der Fächerungsgrößenbestimmungsalgorithmus ist denkbar einfach: Zunächst werden die Zeilen der Reihung an den Cache-Zeilen-Abbildungsgrenzen ausgerichtet. Dann wird für alle Fächerungsgrößen von 0 bis zur Cache-Kapazität (in Vielfachen der Cache-Zeilen-Größe) getestet, ob es innerhalb der so gefächerten Kachel zu Innen-Konfliken kommt, wobei die Kachelelemente in Schritten der Cache-Zeilen-Größe abgetestet werden. Die erste Fächerungsgröße, die keine Innen-Konflikte erzeugt, ist die Ausgabe.

```
Eingabe:  Reihungsgröße: rZeilen x rSpalten
          Kachelgröße : kZeilen x kSpalten
          Cache-Kapazität (in Worten): CKapaz
          Cache-Zeilen-Größe (in Worten): CZeilenGr
Ausgabe:  Fächerungsgröße: FächGr

if (rZeilen mod CZeilenGr == 0)
     then InitFächGr := 0
     else InitFächGr := CZeilenGr - (rZeilen mod CZeilenGr)
for  (FächGr := InitFächGr to CKapaz)
     status := OK
     zReihung[i] := 0 für alle i
     for (i := 0 to kZeilen)
          for (j := 0 to kSpalten step CZeilenGr)
               k := (i x (rZeilen + FächGr) + j) mod CKapaz
               if (zReihung[k/CZeilenGr] == 1)
                    then status := CONFLICT
                    else zReihung[k/CZeilenGr] := 1
     if (status == OK)
          then return FächGr
```

2.4 Anwendung auf mehrere Reihungen

Zur Vereinfachung seien zwei Reihungen betrachtet, die die gleiche Größe haben. Es wird die Annahme vorausgesetzt, dass sie, da der Zugriff auf sie in derselben geblockten Schleife erfolgt, die gleiche Kachelgröße X x Y haben.

Die Berechnung der Fächerungsgröße gestaltet sich nun folgendermaßen: Zunächst bildet man eine neue Kachel T, die das kleinste Rechteck repräsentiert, das die beiden Kacheln der zwei Reihungen umfasst. Für dieses T werden dann nach bekannter Methode die Innen-Konflikte bereinigt. Auf diese Weise werden nicht nur die Aussen-Konflikte zwischen den Kacheln bereinigt, sondern auch deren Innen-Konflikte. Veranschaulicht wird dieser Vorgang in Abb. 3.

Die Kachelgröße innerhalb der Reihungen wird jedoch nicht verändert. Schließlich werden die Anfangsstellen der Reihungen zueinander ausgerichtet.

2.5 Ergebnisse

Das vorgestellte Verfahren wurde auf Rechnern vom Typ SUNSparc5 und SUNSparc10 simuliert und getestet. Für die Simulation wurde der Simulator SHADE von SUN Microsystems verwendet. Testprogramme waren Matrixmultiplikationen, die Schnelle Fouriertransformation (FFT) u.a.

Wie sich ergab, ist der vorgestellte Ansatz zur Cache-Optimierung bisherigen Ansätzen der gleichen Art überlegen. Der Hauptvorteil von DAT ist die Entkopplung der Kachelgrößenbestimmung von der Auffächerung der Daten. Auf diese Weise kann die Größe der Kachel unabhängig von der Eliminierung von Innen-Konflikten optimiert werden, was zu größeren Kacheln führt. Damit wird die Cache-Kapazität besser ausgenutzt. Die Möglichkeit der benutzerdefinierten Kachelgröße führt durchweg zu einer stabilen Leistung des Cache-Speichers. Bisherige Ansätze stießen in ihrer Allgemeinheit bisher auf unlösbare

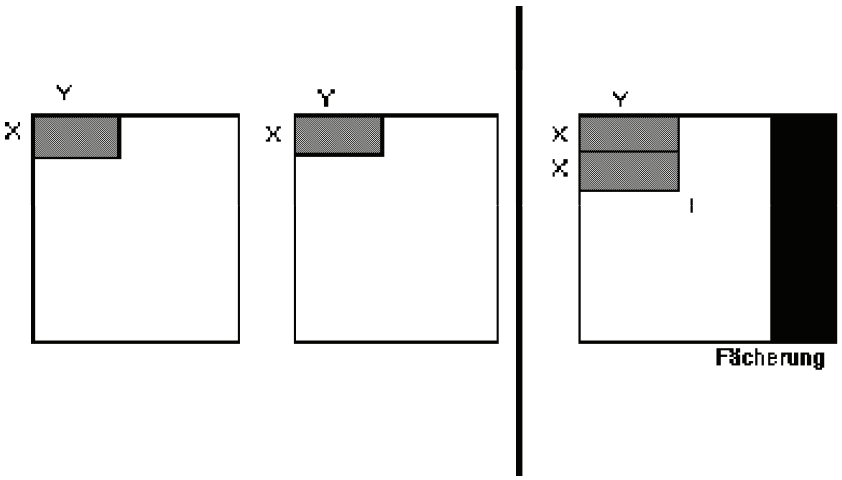


Abbildung 3: Auffächerung zweier Reihungen

Probleme. Eine Neuerung stellt ausserdem die Eliminierung von Aussen-Konflikten zwischen Reihungen dar. In Zahlen ausgedrückt lauten die Verbesserungen in der Ausführungszeit gegenüber dem besten bisherigen Ansatz auf der SunSparc5 12 %, auf der SunSparc10 4,8 %.

Ein bleibendes Problem ist der Konflikt von Fächerungsgrößen bei Reihungen, die in verschiedenen geblockten Schleifen benötigt werden. Solche Anwendungsfälle treten jedoch selten auf.

3 Farbabbildung frequentierter Prozeduren

Die folgenden Ausführungen referieren die Arbeit von A.H. Hashemi, D.R. Kaeli und B. Calder aus dem Jahre 1997 ([2]).

3.1 Grundlagen und Idee

Zu einem ganz anderen Ansatz der Cache-Optimierung kommt man, wenn man nicht Daten- sondern Programmstrukturen Cache-optimal im Hauptspeicher anordnen will. Indem man häufig genutzte Programmteile räumlich lokal anordnet, lassen sich Cache-Konflikte effektiv vermeiden. Die Granularität der betrachteten Programmteile kann dabei verschieden sein, z.B. auf Grundblock-, Schleifen- oder Prozedurebene. Für die Darstellung des Algorithmus wird hier die Prozedurebene gewählt. Als algorithmische Technik zur Anordnung der Prozeduren wird die Färbung derselben entsprechend ihrer Zuordnung zu Cache-Zeilen angewandt. Auf diese Weise lassen sich die Fehlzugriffe im Befehls-Cache durchschnittlich um über 40 % reduzieren.

3.2 Grundideen des Algorithmus

Zunächst wird der Hauptspeicher in Stücke von der Größe des Cache-Speichers zerlegt. In jedem solchen Stück wiederum werden jene Teil-Stücke, die einer Cache-Zeile entsprechen, in den Mittelpunkt der Betrachtung gestellt. Jeder Cache-Zeile entspricht eine Farbe.

Die wesentliche Struktur, aus der die Informationen zur Prozeduranordnung gewonnen werden, ist der Aufrufgraph des betrachteten Programms. In diesem Graphen entspricht jedem Knoten – und nur diesem Knoten – genau eine Prozedur des Programms. Zwei Knoten werden durch eine Kante verbunden, wenn die Prozeduren, die diesen Knoten entsprechen, durch einen Aufruf miteinander kommunizieren. Ein

aufrufender Knoten ist gegenüber dem aufgerufenen Knoten ein Elternknoten, der aufgerufene gegenüber dem aufrufenden ein Kindknoten. Die Kanten werden nach der Häufigkeit der Aufrufe gewichtet.

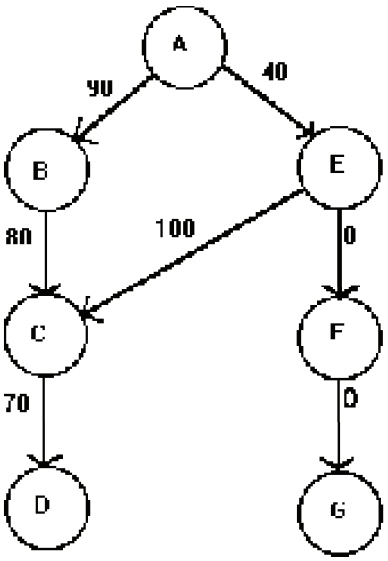
Für jede Prozedur wird eine Unverfügbarkeitsmenge von Farben, d.h. Cache-Zeilen, gebildet. In ihr sind alle Farben, d.h. Cache-Zeilen, enthalten, die einem direkten Eltern- oder Kindknoten zugeordnet sind. Cache-Konflikte werden vermieden, wenn den Eltern und Kindern eines Knoten nur solche Farben zugeordnet werden, die von der Farbe dieses Knotens verschieden sind. Auf diese Weise eliminiert der Algorithmus alle Ein-Generation-Cache-Konflikte.

Im Verlauf der farborientierten Anordnung der Prozeduren kommt es zu Lücken im Hauptspeicher. Um diese zu füllen bedient man sich folgender Erkenntnis: Statistische Untersuchungen haben gezeigt, dass 90 % der Laufzeit eines durchschnittlichen Programmes von 10 - 30 % seiner Prozeduren bestritten werden. Man teilt also die Menge aller Prozeduren eines Programmes in zwei Klassen: Frequentierte und nicht frequentierte Prozeduren. Mit den nicht frequentierten Prozeduren kann man die Lücken, die durch die Anordnung der frequentierten Prozeduren entstehen, ohne Rücksicht auf Cache-Optimalität füllen, da diese Programmteile keinen signifikanten Anteil an vermeidbaren Cache-Konflikten haben.

3.3 Der Algorithmus

Der Algorithmus wird nun anhand eines Beispiels aufgezeigt. Der Einfachheit halber wird von einem direktabgebildeten Cache-Speicher mit nur vier Cache-Zeilen ausgegangen. Jeder Cache-Zeile wird eine der vier Farben Rot, Grün, Blau und Gelb zugeordnet.

Der erste Schritt des Algorithmus besteht in der Ermittlung des Aufrufgraphen eines vorgegebenen Programms. Dies geschieht durch Protokollierung eines Programmlaufs (*tracing*). Ausserdem wird die Größe der Prozeduren, gemessen in Cache-Zeilen, ermittelt. Als Ergebnis für unser Beispiel nehmen wir den Aufrufgraphen und die Prozedurgrößen aus Abb. 4.



Prozedur	# Cache-Zeile
A	1
B	1
C	2
D	2
E	2
F	1
G	1

Abbildung 4: Aufrufgraph und Prozedurgrößen

Zur Klassifizierung der Prozeduren wird anhand des Aufrufgraphen die Frequentiertheit jeder Prozedur ermittelt. Diese ergibt sich aus der Summe der Gewichte aller Kanten, die mit dem Prozedurknoten verbunden sind. In unserem Beispiel ist {F, G} die Menge der nicht frequentierten Prozeduren. Man beachte, dass Frequentiertheit und Zeitaufwand einer Prozedur zwei völlig verschiedene Dinge sind. Eine Prozedur, die zwar viel Zeit in Anspruch nimmt, aber nur selten aufgerufen wird, trägt zu Cache-Konflikten auch nur wenig bei. Zusätzlich wird auch die Menge der Kanten in frequentierte und nicht

frequentierte Kanten partitioniert. Die Menge der frequentierten Kanten lautet hier {A → B, B → C, C → D, A → E, E → C}.

Im nächsten Schritt wird die Menge der frequentierten Kanten in absteigender Reihenfolge sortiert. Die Menge der nicht frequentierten Prozeduren dagegen wird in aufsteigender Reihenfolge sortiert.

Nach diesen Vorbereitungen beginnt der Hauptteil des Algorithmus. Der Ablauf wird in Abb. 5 anhand unseres Beispiels veranschaulicht.

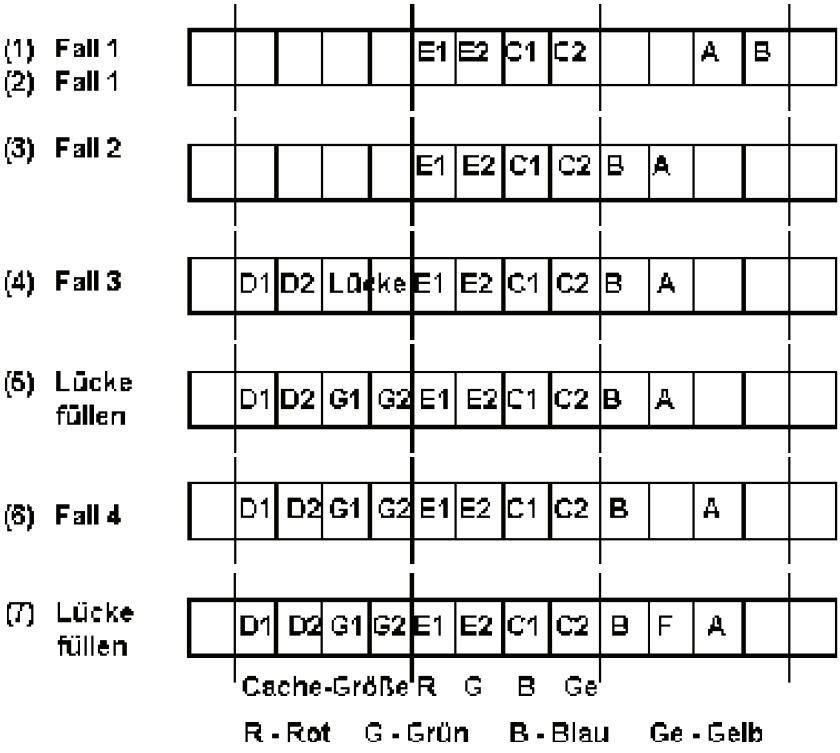


Abbildung 5: Prozedurabbildealgorithmus

Nacheinander werden die frequentierten Kanten betrachtet. Es ergibt sich jeweils einer der vier folgenden Fälle.

1. Fall: Die Kante verbindet zwei noch nicht bearbeitete Prozedurknoten.
Aus den beiden Knoten wird *ein zusammengesetzter Knoten* gebildet. Die beteiligten Knoten werden willkürlich einer Farbe, d.h. einer Cache-Zeile, zugeordnet. Dann werden die Unverfügbarkeitsmengen der zwei beteiligten Knoten aus den Farben des jeweils anderen Knoten gebildet, die zueinander im Verhältnis Elternknoten – Kindknoten stehen. Man beachte, dass Prozeduren, die länger als eine Cache-Zeile sind, in Teile von Cache-Zeilen-Größe aufgeteilt werden. Jeder Teil erhält eine eigene Farbe zugewiesen.

Abb. 5 zeigt unser Beispiel (1): Die gewichtigste Kante ist E → C mit dem Gewicht 100. E und C sind jeweils zwei Cache-Zeilen groß, werden also in E1, E2 und C1, C2 geteilt, denen die Farben Rot, Grün, Blau und Gelb ohne Cache-Konflikte zugeordnet werden. Die Unverfügbarkeitsmengen lauten demnach {Blau, Gelb} für E und {Rot, Grün} für C. Die nächstgewichtigste Kante A → B gehört auch zu Fall 1, und wird mit Blau und Gelb willkürlich zwei Cache-Zeilen zugeordnet (2). Die Unverfügbarkeitsmengen lauten also {Gelb} für A und {Blau} für B.

2. Fall: Die Kante verbindet zwei bereits bearbeitete Prozedurknoten in verschiedenen zusammengesetzten Knoten.
In diesem Fall wird der kürzere zusammengesetzte Knoten dem längeren hinzugefügt. Die Länge wird

dabei in der Anzahl der Prozedurknoten gemessen. Dabei sind zwei Entscheidungen zu fällen:

a) Soll der kürzere zusammengesetzte Knoten links oder rechts vom längeren Knoten angeordnet werden? Es wird folgende Heuristik angewandt: Der kürzere Knoten wird auf derjenigen Seite angeordnet, der derjenige Prozedurknoten der aktuellen Kante, der im längeren zusammengesetzten Knoten enthalten ist, am nächsten ist. Die Nähe wird dabei in Cache-Zeilen gemessen.

b) In welcher Reihenfolge sollen die Prozedurknoten des kürzeren zusammengesetzten Knoten angeordnet werden – in der vorhandenen, oder in der umgekehrten Reihenfolge? Diese Frage wird dahingehend entschieden, dass diejenige Reihenfolge zu wählen ist, bei der die beiden Knoten, die die betrachtete Kante bilden, einander am nächsten sind.

Hierauf wird überprüft, ob die beiden Knoten der aktuellen Kante in einer Farbe übereinstimmen, also einen Cache-Konflikt erzeugen. Falls ja, wird der kürzere zusammengesetzte Knoten so weit vom längeren weggeschoben, bis kein Konflikt mehr auftritt. Die entstehende Lücke wird mit nicht frequentierten Prozeduren gefüllt. Ist Konfliktfreiheit nicht erzielbar, so wird der Konflikt belassen.

Schließlich werden alle Knoten des kürzeren zusammengesetzten Knoten mit ihrer neuen Cache-Zeilen-Farbe versehen, wodurch sich auch Unverfügbarkeitsmengen ändern können.

In unserem Beispiel lautet die betrachtete Kante B → C. Beim Übergang von (2) nach (3) wird der kürzere zusammengesetzte Knoten AB rechts an den längeren zusammengesetzten Knoten EC angehängt, da C in EC dem rechten Ende von EC näher ist. AB wird ausserdem in der Reihenfolge BA angeordnet, da B als ein Knoten der aktuellen Kante so dem längeren zusammengesetzten Knoten näher ist. Ein Konflikt innerhalb des neuen zusammengesetzen Knotens wird (noch) nicht registriert, da die Kante A → E noch nicht in Betracht gezogen wurde. Also können A und B neu gefärbt und mit aktualisierten Unverfügbarkeitsmengen versehen werden.

3. Fall: Die Kante verbindet einen bereits bearbeiteten und einen noch nicht bearbeiteten Prozedurknoten. Diese Situation wird völlig analog zum 2. Fall abgehandelt. Dem kürzeren zusammengesetzten Knoten entspricht dabei stets der noch nicht bearbeitete Prozedurknoten.

In unserem Beispiel wird die Kante C → D betrachtet, die an den zusammengesetzten Knoten ECBA angehängt wird (4). D wird links angehängt, da C in ECBA dem linken Ende von ECBA näher ist. Da D mit C in Konflikt gerät, wird es nach links verschoben. Die entstehende Lücke wird mit der nicht frequentierten Prozedur G aufgefüllt (5).

4. Fall: Die Kante verbindet zwei Prozedurknoten innerhalb ein und desselben zusammengesetzten Knotens.
Falls die Farben, d.h. Cache-Zeilen, der zwei die aktuelle Kante bildenden Prozedurknoten nicht im Konflikt liegen, ist nichts zu tun. Ansonsten wird diejenige der beiden Prozeduren, die einem der beiden Enden des zusammengesetzten Knoten näher liegt, gewählt und ans Ende des zusammengesetzten Knotens und darüber hinaus verschoben, bis kein Konflikt mehr vorliegt. Die entstehende Lücke wird mit nicht frequentierten Prozeduren gefüllt. Ist Konfliktfreiheit nicht zu erreichen, unterbleibt ihre Eliminierung.

In unserem Beispiel kommt die Kante A → E in Bearbeitung (6). Da A und E miteinander im Konflikt liegen, wird A als die einem Ende des zusammengesetzten Knotens nähere Prozedur so weit verschoben, bis Konfliktfreiheit herrscht. Die entstehende Lücke wird mit der nicht frequentierten Prozedur F gefüllt (7).

Nach Abarbeitung aller Kanten nach den vier Fällen endet der Algorithmus mit folgenden Schritten: Noch verbleibende nicht frequentierte Prozeduren werden an freien Stellen im Hauptspeicher eingefügt. Verbleiben am Ende mehrere disjunkte zusammengesetzte Knoten, so werden sie in der Reihenfolge ihrer Frequentiertheit im Hauptspeicher angeordnet.

3.4 Vergleich mit früheren Arbeiten

Die vorgestellte Ausführung des beschriebenen Ansatzes baut auf etlichen Vorarbeiten auf. Stellvertretend für diese soll die Arbeit von Pettis und Hansen aus dem Jahre 1990 kurz vorgestellt werden ([4]).

Auch diese Arbeit wendet die Anordnung von Prozeduren und das Prinzip Je-näher-desto-besser an. Ausserdem wird auch hier schon die Betrachtung der Aufrufkanten nach ihrem Gewicht ins Zentrum

des Algorithmus gestellt. Die Gewichte werden ebenfalls durch einen protokollierten Programmlauf ermittelt. Zwei durch eine Aufrufkante verbundene Prozedurknoten werden hier allerdings nicht in einem zusammengesetzten Knoten sondern in einer sogenannten Kette zusammengefasst.

Der Algorithmus soll wieder anhand unseres in Abb. 4 vorgestellten Beispiels demonstriert werden. Zu Beginn ergibt sich das gleiche Bild wie oben: Die Betrachtung der Kante E → C führt zu Bildung der Kette EC, die Kante A → B zur Kette AB. Auch die Kante B → C führt noch zum gleichen Ergebnis, da B → C nach der Je-näher-desto-besser-Heuristik rechts und in umgekehrter Reihenfolge an die vorhandene Kette ECBA angefügt wird. Das Ergebnis ist in Abb. 6 (1) festgehalten.

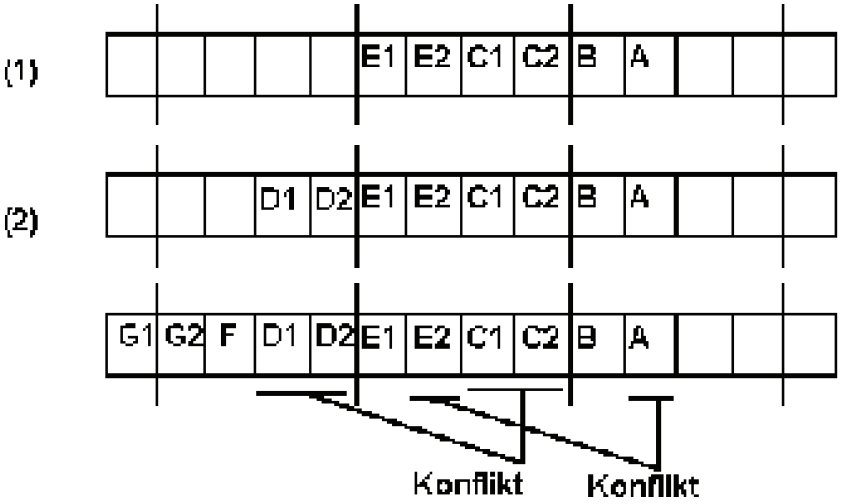


Abbildung 6: Der Algorithmus nach Pettis und Hansen 1990

Die Bearbeitung der Kante C → D führt jedoch zu einer Abweichung vom Ergebnis des oben vorgestellten Algorithmus. Abb. 6 (2) zeigt deutlich: Egal auf welcher Seite man D an die Kette ECBA anhängt, es tritt immer ein Cache-Konflikt mit C auf: Sobald eine Kette länger als die Größe des Cache-Speichers wird, verliert das Verfahren wesentlich an Wirkung. Die endgültige Abbildung der Prozeduren zeigt schließlich zwei Erste-Generation-Cache-Konflikte zwischen D und C und zwischen E und A.

Im Gegensatz dazu kann der hier neu vorgestellte Algorithmus durch die Abbildung der Prozeduren auf Cache-Zeilen (Farben) bei einer Überschreitung der Cache-Speichergröße entsprechend reagieren. Die Berücksichtigung der Cache-Maße und die Möglichkeit, durch Verschieben Lücken zu erzeugen und sie geeignet zu füllen, sind weitere Vorteile des neuen Algorithmus.

3.5 Ausblick

Für die Zukunft lassen sich einige Ansätze zur Verbesserung des vorliegenden Algorithmus nennen.

Um das Verfahren auch für assoziative Cache-Speicher optimal einsetzen zu können, könnte man den Adressraum des Hauptspeichers in einer anderen Weise einteilen: Nicht mehr jeder Cache-Zeile, sondern jedem Satz von Cache-Zeilen wird eine Farbe zugeteilt. Die Schwierigkeit dabei besteht wesentlich darin, dass dann in den Unverfügbarkeitsmengen eine Farbe mehrfach auftreten kann, aus der Menge also eine Mehrfachmenge wird.

Eine weitere Verfeinerung könnte durch Grundblockumordnung und Prozeduraufspaltung erzielt werden. Indem man zuerst die Grundblöcke jeder Prozedur in frequentierte und nicht frequentierte aufspaltet und dann durch die Abbildung auf Farben geeignet umordnet, erreicht man eine höhere Granularität. Danach kann man solche Prozeduren spalten, von denen aus mehrere Aufrufe an andere Prozeduren ausgehen. Auf diese Weise sind die Beschränkungen bei der Farbwahl nicht so eng, d.h., es entsteht mehr Flexibilität bei der Zuordnung zu Cache-Zeilen, was letztlich der Effizienz des Verfahrens zuträglich ist.

U.U. wäre es sogar sinnvoll, ganz von der Betrachtung von Prozeduren Abstand zu nehmen, und sich nur noch auf die Grundblöcke zu konzentrieren.

Eine weitere Verbesserung ließe sich aus der Information gewinnen, ob die Anzahl der Aufrufe von einer Prozedur A an eine Prozedur B zeitlich dicht oder verteilt stattfinden. Je weniger dicht die Aufrufe erfolgen, desto weniger wichtig ist es, Cache-Konflikte zu vermeiden. Darüberhinaus wäre die Betrachtung von mehr als nur von Ein-Generation-Cache-Konflikten ein wichtiger Verbesserungsschritt. Dazu müsste die Struktur des Aufrufgraphen grundsätzlich geändert werden.

3.6 Ergebnisse

Zu Testzwecken wurde der vorgestellte Algorithmus, der in der Bindephase des Übersetzens anzuwenden ist, in den gcc2.7.2-Übersetzer eingebaut. Die Testprogramme wurden dem SPEC95-Benchmark entnommen. Für die Protokollierung von Programmläufen und simulative Tests wurde das Simulationswerkzeug ATOM von DEC verwendet.

Wie sich ergab, kann der vorgestellte Algorithmus tatsächlich erhebliche Verbesserungen der Cache-Ausnutzung bewirken. Besonders effizient wirkt er sich dort aus, wo die Größe des Aufrufgraphen der frequentierten Prozeduren größer als der Cache-Speicher ist und eine gewisse Komplexität erreicht.

Im Ganzen werden die Cache-Fehlzugriffe damit um durchschnittlich 40 % reduziert. Verglichen mit früheren Arbeiten wird eine durchschnittliche Reduktion der Fehlzugriffsrate um 17 % erzielt.

4 Seitenanordnung in Multiprozessorsystemen

Die folgenden Ausführungen referieren die Arbeit von E. Bugnion, J.M. Anderson, T.C. Mowry, M. Rosenblum und M.S. Lam aus dem Jahre 1996 ([3]).

4.1 Grundlagen und Idee

Parallelisierung bietet neue Möglichkeiten, die Leistung von Rechensystemen zu verbessern. Die Entwicklung hat heute dazu geführt, dass man Programme automatisch parallelisieren kann. Dieser Fortschritt kann aber nur dann wirklich ausgenutzt werden, wenn das Speicherverwaltungssystem den Anforderungen parallelisierter Programme gewachsen ist. Die Seitenabbildungsstrategien heutiger Betriebssysteme wurden für Ein-Prozessor-Systeme entworfen, bei denen ein Prozessor auf alle Daten zugreift. In einem Multiprozessorsystem jedoch greift jeder Prozessor oft nur auf eine Teilmenge von Daten zu. Auf diese Weise kommt es zu einer sehr einseitigen Nutzung des Cache-Speichers, bei der manche Teile desselben überlastet und manche kaum genutzt werden.

Abhilfe soll die hier vorgestellte übersetzergesteuerte Seitenanordnungsstrategie für Multiprozessorsysteme schaffen (*Compiler-Directed Page Coloring for Multiprocessors*, CDPC). Dieses Verfahren macht zur Übersetzungszeit Vorhersagen über die Zugriffsmuster eines konkreten parallelisierten Programmes. Diese Information wird an das Betriebssystem weitergegeben, das dann die Daten im Hauptspeicher Cache-optimal anordnen kann. Dies geschieht über eine geeignete Abbildung des virtuellen Adressraumes auf den physikalischen Adressraum des Hauptspeichers.

4.2 Betriebssysteme: Grundlagen

Herkömmliche Betriebssysteme ordnen den einzelnen Seiten des physikalischen Hauptspeicheradressraumes Farben zu, wobei jede Farbe einer Cache-Zeile entspricht. Haben zwei Seiten dieselbe Farbe, so werden sie auf dieselbe Cache-Zeile abgebildet, sind also zwei potentielle Cache-Konflikt-Partner. Tritt ein Seitenfehler auf – kein Cache-Fehlzugriff! – so muss das Betriebssystem die Seite in den Hauptspeicher nachladen. Dabei ist es frei darin, zu entscheiden, an welche Stelle die Seite geladen wird. Um eine Cache-optimale Entscheidung zu treffen, gibt es grundsätzlich zwei Verfahren, diejenige Farbe zu bestimmen, die die zu allokierende Seite bekommen sollte: Seitenfärbung (*page coloring*, angewandt in SGI-Unix IRIX und Windows NT) und *bin hopping* (wörtlich: “Eimerspringen”, angewandt in Digital Unix). Seitenfärbung versucht, lokal zusammengehörige Seiten benachbarten Farben zuzuordnen, um so

durch räumliche Lokalität der Daten Cache-Konflikte zu vermeiden. *bin hopping* dagegen geht die Farben in der Reihenfolge durch, in der Cache-Fehlzugriffe auftraten, und versucht so, durch zeitliche Lokalität Cache-Konflikte zu vermeiden. Dabei entsteht in einem Multiprozessorsystem das Problem, dass bei konkurrierenden Seitenfehlern im Kern des Betriebssystems ein Wettrennen um die Seiten beginnt, das zu unvorhersehbaren Leistungseinbußen führen kann.

4.3 Der Algorithmus

Der Algorithmus besteht im wesentlichen aus zwei Teilen. Zunächst werden zur Übersetzungszeit Informationen über die Zugriffsmuster des parallelisierten Programmes gewonnen, worauf aus diesen zur Laufzeit Hinweise zur Färbung der Seiten für das Betriebssystem generiert werden.

Als erstes werden diejenigen Daten, die von einem Prozessor zugegriffen werden, im virtuellen Adressraum zusammenhängend angeordnet (soweit möglich). Dann werden die einzelnen parallel ausführbaren Programmteile den jeweiligen Prozessoren statisch zugeordnet. Dadurch werden die Zugriffsmuster der Prozessoren vorhersagbar. Folgende Informationen werden gewonnen:

- Reihungspartitionierung. Allgemeine Informationen über die Reihungen sowie die Information, wie die Bearbeitung einer Reihung in einer parallelen Schleife auf die einzelnen Prozessoren verteilt wird.
- Kommunikationsmuster. Kommunikation wird hier verstanden als die Zugriffe zweier Prozessoren auf dasselbe Datum, von denen der erste schreibend ist.
- Gruppenzugriffsinformation. Alle Paare von Reihungen werden registriert, die zusammen in einer Schleife bearbeitet werden.

Zur Laufzeit werden nun die Hinweise zur Färbung der Seiten generiert. Die Färbungsstrategie verfolgt zwei Ziele:

- Die Daten, die von einem Prozessor zugegriffen werden, sollten möglichst zusammenhängend im physikalischen Adressraum angeordnet werden. Dies führt offensichtlich zur Vermeidung von Cache-Konflikten.
- Die Anfangsstellen von Reihungen, die zeitlich dicht von ein und demselben Prozessor zugegriffen werden, sollten verschieden gefärbt sein. Der Grund dafür ist, dass beim Zugriff auf zwei Reihungen oft der Fall vorliegt, dass für den Zugriff auf die Stelle i der einen Reihung ein Zugriff auf die Stelle i+c der anderen Reihung erfolgt, wobei c eine kleine Konstante ist.

Zum weiteren Verständnis seien folgende zwei Definitionen gegeben: Ein *einheitliches Zugriffssegment* ist eine Menge von Seiten innerhalb derselben Reihung, die von einer konstanten Menge von Prozessoren zugegriffen werden. Jedes Datum wird somit eindeutig einem einheitlichen Zugriffssegment zugeordnet. *Einheitliche Zugriffsmengen* dagegen bestehen aus der Vereinigung von einheitlichen Zugriffssegmenten, die von ein und derselben Prozessormenge zugegriffen werden. Auch hier herrscht Eindeutigkeit in der Zuordnung. Der virtuelle Adressraum jeder Reihung wird nun in maximale einheitliche Zugriffssegmente aufgeteilt.

Der nun folgende Hauptteil des Algorithmus besteht aus fünf Schritten, die anhand von Abb. 7 veranschaulicht werden:

1. Bildung der einheitlichen Zugriffsmengen (unzusammenhängend).

Dies erfolgt durch ein simples Verfahren: Der Adressraum, der zunächst als eine einzige einheitliche Zugriffsmenge betrachtet wird, wird an den Grenzen der Reihungen und bei den Wechseln von Zugriffsmustern innerhalb einer Reihung geteilt, entsprechend der vorberechneten Information. Am Beispiel: In (a) bilden die Speicherzellen 0-7 und 8-13 jeweils eine Reihung. Also wird zwischen diesen eine Grenze gesetzt. Ausserdem wechselt an den farblich unterschiedenen Stellen das Zugriffsmuster: Auch hier werden die Grenzen entsprechend zwischen den Zellen 3 und 4, 5 und 6, 10 und 11 und 11 und 12 gesetzt.

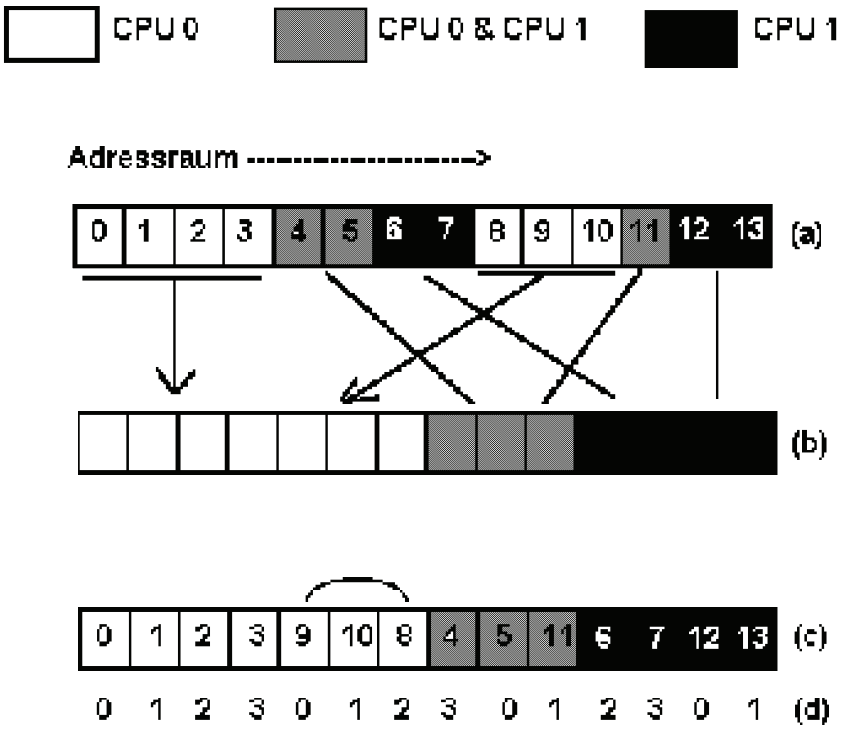


Abbildung 7: Der CDPC-Algorithmus

Die einheitlichen Zugriffsmengen werden nun von aussen nach innen sortiert:

2. Zusammenhängende Anordnung der einheitlichen Zugriffsmengen.
Mithilfe eines Graphalgorithmus werden nun die voneinander noch getrennten Teile der einzelnen Zugriffsmengen zusammenhängend angeordnet. Das Beispiel veranschaulicht dies in (b).

3. Einheitliche Zugriffssegmente innerhalb der einheitlichen Zugriffsmengen zusammenhängend ordnen.
Wiederum mithilfe eines Graphalgorithmus werden die voneinander getrennten Teile der einzelnen Zugriffssegmente zusammenhängend angeordnet, so dass zusammengehörige Reihungsteile innerhalb einer einheitliche Zugriffsmenge wieder nebeneinander im Zusammenhang stehen. Ausserdem werden Reihungen, die nach der vorberechneten Gruppenzugriffsinformation (s.o.) zusammen bearbeitet werden, nebeneinander angeordnet. Im Beispiel entfällt der Schritt 3, da diese Ordnung schon besteht.

4. Ordnen der Seiten innerhalb eines einheitliche Zugriffssegmentes.
Ziel ist es, für die Anfänge der Reihungen möglichst verschiedene Farben zutreffen zu lassen. Dies wird dadurch erzielt, dass man die Reihungen bzw. Reihungsteile zunächst innerlich aufsteigend sortiert, sie dann aber nicht mit ihrem ersten Element in der ersten Speicherzelle abspeichert. Vielmehr wird die Sequenz der Reihungselemente als ein geschlossenes Band aufgefasst, bei dem der Nachfolger des letzten Elementes wieder das erste Element ist. Bei jeder Reihung, die zu allokieren ist, wird also nicht das erste Element an die erste Stelle gesetzt, sondern bei jeder nächsten zu allokierenden Reihung das jeweils nächste Element. An unserem Beispiel: Die Reihung 0-1-2-3 wird mit ihrem ersten Element 0 an der ersten Stelle allokiert (c). Da sich nun in dieser einheitlichen Zugriffsmenge noch ein weiterer Reihungsanfang befindet, wird diese Reihung nicht mit seinem ersten, sondern mit seinem zweiten Element an erster Stelle allokiert: 9-10-8.

5. Belegung der Seiten mit Farben.

Reihum werden nun die Farben an die virtuellen Seiten verteilt. Im Beispiel: (d).

4.4 Ergebnisse

Die Testreihen wurden mit dem Simulationswerkzeug SimOS vorbereitet, auf dem IRIX 5.3 und SGI-Unix ausgeführt wurden. Modelliert wurde ein 400-MHz-R4400-Prozessor mit geteiltem 2-fach satzassoziativem Cache-Speicher von 32 KB bzw. einem direktabgebildeten Cache-Speicher von 1 MB. Getestet wurde auf einem 8-CPU DEC AlphaServer 8400 mit 350 MHz 21164-Prozessoren. Als Testprogramme dienten Teile des SPEC95fp-Benchmark.

Die Implementierung in die jeweiligen Betriebssysteme gestaltete sich denkbar einfach. Zusätzlich zum vorgestellten Algorithmus wurde ausserdem eine Ausrichtung der Daten an den Cache-Zeilen-Abbildungsgrenzen sowie eine Auffächerung der Datenstrukturen vorgenommen.

Die Testergebnisse zeigen, dass CDPC sowohl die Seitenfärbungs- als auch die *bin hopping*-Strategie übertrifft. Bei Ausführung von SPEC95fp-Programmen auf einem AlphaServer mit 8 Prozessoren konnte eine Beschleunigung von 8 % gegenüber *bin hopping* und eine Beschleunigung von 20 % gegenüber der Seitenfärbungsstrategie erzielt werden.

Seine besonderen Stärken hat das Verfahren bei der Reduzierung von Ersetzungsfehlzugriffen, also bei der Verhinderung einer vorzeitigen Verdrängung von Daten aus dem Cache-Speicher.

CDPC arbeitet vollautomatisch, ist leicht zu implementieren und kann ohne Probleme in kommerzielle Betriebssysteme integriert werden. Für den SPEC95fp-Benchmark konnte das bisher beste Ergebnis von 57,4 erzielt werden.

4.5 Vorladen als ergänzende Technik

Für kapazitäts- oder kommunikationsbedingte Fehlzugriffe ist CDPC nicht geeignet. Das Verfahrens des Vorladens von Daten bietet sich zur Ergänzung an. Dieses versteckt die Zeit, die beim Nachladen nach einem Fehlzugriff benötigt wird, indem zu einer speicherzugriffslosen Zeit ein Vorladen von später verwendeten Daten erfolgt. Vor allem bei wenigen Prozessoren bringt diese Kombination Erfolge, weil hier die Anzahl der kapazitätsbedingten Fehlzugriffe viel höher ist als die Anzahl der Cache-Konflikte, also der Fehlzugriffe aufgrund vorzeitiger Verdrängung. Besonders interessant ist, dass CDPC und Vorladen sich gegenseitig verstärken: Vorladen versteckt die Ladezeit bei Fehlzugriffen, die CDPC nicht verhindern kann, während CDPC den Datenbus frei hält, was wiederum das Vorladen begünstigt. In einem Beispiel ergab sich aus diesem Synergie-Effekt, dass eine Beschleunigung durch CDPC allein von 29 % und durch Vorladen allein von 24 % von einer Leistungssteigerung durch CDPC und Vorladen zusammen von 88 % weit übertroffen wurde.

5 Zusammenfassung

Es wurde gezeigt, dass mithilfe der Berücksichtigung der Abbildung vom Hauptspeicher auf den Cache-Speicher eine erheblich bessere Nutzung des Cache-Speichers erzielt werden kann. Dabei kamen verschiedenste Techniken zum Einsatz:

- Die Auffächerung von Reihungen
- Die Konzentration auf frequentierte Daten / Programmteile
- Die Unterscheidung von Daten und Programmen
- Das Ausnutzen von Vorwissen über das Programmverhalten
  - durch die Protokollierung des Programmlaufs
  - durch die Bestimmung der Zugriffsmuster von Prozessoren
- Die Zuordnungsabbildung Hauptspeicher → Cache durch Färbung

- Synergieeffekte
  - mit der Optimierung der Kachelgröße
  - mit Vorladen (*prefetching*)

Durch einfaches Verschieben und Anordnen von Daten nach ebenso einfachen Algorithmen kann der "Flaschenhals Speicherzugriff" offensiv angegangen werden.

Literatur

[1] P.R. Panda, H. Nakamura, N.D. Dutt, A. Nicolau. *Improving Cache performance through tiling and data alignment.* In Solving Irregularly Structured Problems in Parallel (Proceedings IRREGULAR '97), number 1253 in LNCS, pages 167-185, Paderborn, Germany, June 1997, Springer.

[2] A.H. Hashemi, D.R. Kaeli, B. Calder. *Efficient procedure mapping using cache line coloring.* In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 171-182, June 1997.

[3] E. Bugnion, J.M. Anderson, T.C. Mowry, M. Rosenblum, M.S. Lam. *Compiler-directed page coloring for multiprocessors.* In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), ACM SIGPLAN Notices 31(9), pages 244-255, Cambridge, MA, October 1996.

[4] K. Pettis und R.C. Hansen. *Profile guided code positioning.*. Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pages 16 - 27, ACM, June 1990.

Seminar - Ausarbeitung V

Felix Schröter

Datenflußanalyse für Arrays

Zusammenfassung

Diese Ausarbeitung führt zuerst in die Kontroll- und Datenflußanalyse im allgemeinen ein. Mit Hilfe dieser Grundlagen werden dann drei Methoden der Datenflußanalyse für Arrayzugriffe dargestellt. Die erste Technik kann bei einer eingeschränkten Klasse von Programmen die genauen Datenabhängigkeiten und Zugriffsmuster ermitteln. Die zweite Technik konzentriert sich auf die Wiederverwendung von Variablen, Arrays oder Abschnitten daraus und ist in der Lage, die begrenzte Größe des Datencaches zu berücksichtigen. Die dritte Technik ist eine Analyse, die eigentlich für die Generierung von Code für effizienten Nachrichtenaustausch für Parallelrechner mit verteiltem Speicher gedacht ist. Abschließend werden Anwendungen der Informationen, die durch die beschriebenen Datenflußanalysemethoden gefunden werden, motiviert und Erfahrungen mit diesen Anwendungen dargestellt.

1 Einführung

Ein aktuelles Forschungsgebiet im Übersetzerbau ist die Verbesserung des Datenzugriffsverhaltens des generierten Codes, um durch eine höhere Lokalität und Verringerung von Konflikten die Nutzung von Caches zu verbessern. Diese Verbesserungen werden immer wichtiger, da die Diskrepanz zwischen der Geschwindigkeit der Prozessoren und des Hauptspeichers zunimmt.

Für viele Techniken in diesem Bereich sind Informationen über den zu optimierenden Code nötig oder nützlich. Ein Teil dieser Informationen kann durch eine statische Analyse des Programms[1] gewonnen werden.

Eine Analyse der Berechnung und Verwendung von Daten in einem Programm nennt man Datenflußanalyse. In gängigen Übersetzern wird diese meist durchgeführt, außer wenn alle Optimierungen ausgeschaltet sind. Allerdings betrachten die meisten Übersetzer Reihungen[2] als unteilbare Einheit, wodurch sie gerade im Bereich von Schleifen, aber auch der Wiederverwendung von Daten über Schleifengrenzen hinweg, wesentliche Information verlieren.

Die hier vorgestellten Methoden können verwendet werden, um genauere Informationen über Reihungszugriffe gewinnen und diese Informationen anschließend nutzbringend in der Optimierung einzusetzen.

2 Kontroll- und Datenflußanalyse

In diesem Abschnitt wird ein Überblick über die Datenflußanalyse gegeben. Einführend wird die Kontrollflußanalyse eingeführt, aber nur soweit sie für die spätere Datenflußanalyse relevant ist.

2.1 Kontrollflußanalyse

Die Datenflußanalyse setzt üblicherweise Informationen über den Kontrollfluß voraus. Dazu können wir einen Kontrollflußgraphen bilden und anschließend möglicherweise zusätzliche Informationen über den

[1] d.h. zur Übersetzungszeit
[2] deutsch für Arrays

Kontrollfluß sammeln. Diese Information wird verwendet, um den Kontrollflußgraphen weiter zu strukturieren und/oder ihn mit zusätzlichen Informationen „auszuschmücken“, z.B. in Form einer Markierungsfunktion für Ecken und Kanten.

2.1.1 Kontrollflußgraph

Ein Kontrollflußgraph ist ein gerichteter Graph (N, E), bei dem die Ecken die Grundblöcke einer Funktion[3] und die Kanten die möglichen Sprünge zwischen Grundblöcken repräsentieren. Zusätzlich wird eine Ecke als Anfangsecke ausgezeichnet. Dieser entspricht dem Grundblock, der beim Aufruf der betrachteten Funktion zuerst ausgeführt wird.

Ohne Beschränkung der Allgemeinheit wird angenommen, daß es genau eine Anfangsecke n_0 gibt und daß keine Kante n_0 als Ziel hat. Wenn eine Funktion mehr als einen Einsprungpunkt hat, kann eine künstliche Anfangsecke gebildet werden, von dem Kanten zu den Ecken, die die Einsprungpunkte repräsentieren, gehen. Wenn n_0 von einer Kante erreicht wird, wird ein leerer Grundblock n'_0 gebildet, mit Kante (n'_0, n_0). Dann ist n'_0 die neue Anfangsecke mit der gewünschten Eigenschaft. Außerdem wird angenommen, daß von der Anfangsecke alle Ecken des Graphen erreicht werden können. Wenn dies nicht der Fall ist, werden alle nicht erreichbaren Ecken entfernt – sie repräsentieren kontrollflußtoten Code.

2.1.2 Tarjan-Intervalle, Intervallflußgraph

Da Arrayzugriffe oft in Schleifen stattfinden, ist es nötig, im Kontrollflußgraph Informationen über Schleifen aufzubauen.

Eine Darstellung dieser Informationen ist der Intervallflußgraph. Dieser entsteht aus dem Kontrollflußgraph dadurch, daß die Ecken in Intervalle eingeordnet und die Kanten klassifiziert werden. Dabei können die Intervalle auch geschachtelt sein. Außerdem werden gegebenenfalls kleine Transformationen durchgeführt, um bestimmte Einschränkungen zu befriedigen.

Ein Tarjan-Intervall ist eine Menge von Ecken $T(h)$ mit einem eindeutig definierten Kopf $h \notin T(h)$, die einer Schleife im Programmtext entsprechen. Die Schleife wird dabei nur durch h betreten. Bei geschachtelten Schleifen enthält das Intervall für eine äußere Schleife nicht nur die Köpfe, sondern alle Ecken der inneren Schleifen.

In dem Intervallflußgraph gibt es eine Ecke ROOT, die als Kopf für die gesamte Funktion gilt. Jeder Ecke n im Graph ist seine Schachtelungstiefe LEVEL(n) zugeordnet. LEVEL(ROOT) ist dabei als 0 definiert. Mit CHILDREN(n) werden die Elemente m von $T(n)$ bezeichnet, für die LEVEL(m) = LEVEL$(n) + 1$ gilt.

Die Kanten $(x \to y)$ des Graphs werden jetzt wie folgt klassifiziert:

- ENTRY: Kante vom Kopf eines Intervalls in das Intervall hinein: $y \in T(x)$.
- CYCLE: Kante vom Inneren eines Intervalls auf den Kopf: $x \in T(y)$.
- JUMP: Sprung aus einer Schleife heraus: $\exists h : x \in T(h), y \notin T(h) \cup \{h\}$.
- FLOW: Kante zwischen verschiedenen Intervallen (also keine der obigen drei Klassen): $\forall h : x \in T(h) \Leftrightarrow y \in T(h)$.

Wir definieren E_{FLOW} als die Menge der FLOW-Ecken im Kontrollflußgraphen. Analoge Definitionen gelten für die anderen Klassen der Ecken.

Weiter definieren wir die Nachfolgermenge und Vorgängermenge einer Ecke wie folgt: SUCC$(n) := \{s|(n,s) \in E\}$, PRED$(n) := \{p|(p,n) \in E\}$. Mit einem Index können die betrachteten Kanten auf eine bestimmte Klasse eingeschränkt werden.

In dem Beispiel von Abbildung 1 sind die Ecken $(1 \to 2)$, $(2 \to 3)$ und $(6 \to 7)$ ENTRY-Ecken, die Ecken $(4 \to 1)$ und $(8 \to 6)$ CYCLE-Ecken. Die restlichen Ecken sind FLOW-Ecken. In diesem Beispiel kommen also keine JUMP-Ecken vor.

Wenn ein Intervallflußgraph durchlaufen wird, werden zwei Paare von Teilordnungen definiert:

[3] in dieser Ausarbeitung Synonym zu „Prozedur“

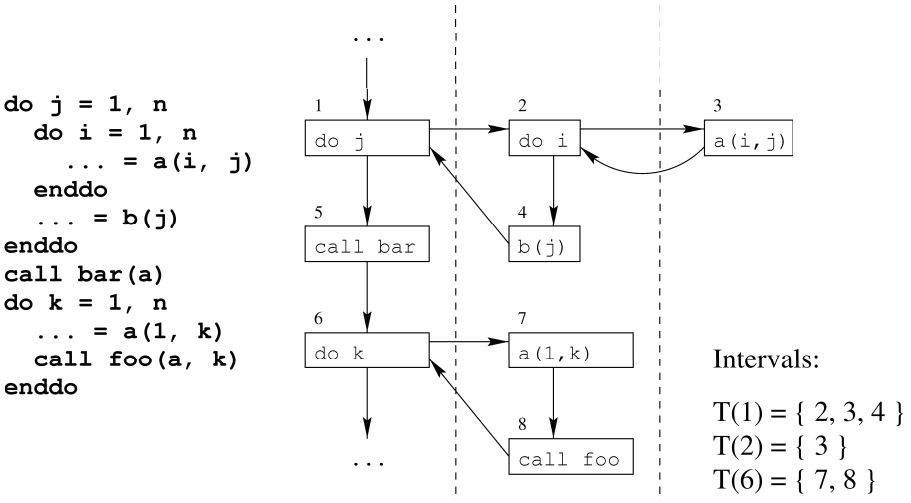


Abbildung 1: Beispiel für Intervallflußgraphen

- VORWÄRTS/RÜCKWÄRTS: Bei einer FLOW-/JUMP-Kante (n_1, n_2) wird bei der VORWÄRTS-Reihenfolge n_1 vor n_2 besucht, bei RÜCKWÄRTS umgekehrt.
- ABSTEIGEND/AUFSTEIGEND: Für $n_1, n_2 \in N$ mit $n_2 \in T(n_1)$ wird bei einer ABSTEIGENDen Reihenfolge n_1, bei einer AUFSTEIGENDen n_2 zuerst besucht.

Diese Teilordnungen sind orthogonal. Wenn also zwei Ecken n_1 und n_2 in der einen Ordnung (VORWÄRTS oder RÜCKWÄRTS) vergleichbar sind, sind sie in der anderen (ABSTEIGEND oder AUFSTEIGEND) ungeordnet und umgekehrt. Also können diese Ordnungen kombiniert werden.

2.2 Datenflußgleichungen

Nachdem die Kontrollflußanalyse abgeschlossen ist, kann die eigentliche Datenflußanalyse erfolgen.

Eine Methode ist, Datenflußgleichungen aufzustellen. Dabei wird eine Menge U vorgegeben, die das Universum des Datenflußproblems genannt wird. Außerdem gibt es eine Reihe von Abbildungen $f_i : N \to 2^U$. Diese Abbildungen werden dann durch ein Gleichungssystem definiert – die Datenflußgleichungen.

Normalerweise werden einige dieser Abbildungen durch bestimmte Ergebnisse lokaler Analyse (d.h. auf den einzelnen Grundblock beschränkt, ohne Betrachtung des Kontexts) bestimmt. Die Datenflußgleichungen stellen dann oft ein Fixpunktproblem dar. In manchen Fällen sind die Gleichungen jedoch durch eine geschickte Auswertungsreihenfolge ohne Fixpunktiteration in einem Durchlauf zu lösen.

Ein sehr einfaches Beispiel:

$$\begin{aligned} \text{USE}_{\text{IN}}(n_0) &= \emptyset && (3)\\ \text{USE}_{\text{IN}}(n) &= \bigcup_{(p,n)\in E} \text{USE}_{\text{OUT}}(p) && (4)\\ \text{USE}_{\text{OUT}}(n) &= \text{USE}_{\text{IN}}(n) \cup \text{USE}_{\text{LOCAL}}(n) && (5) \end{aligned}$$

Das Universum U ist die Menge aller Variablen, die in der untersuchten Funktion definiert sind. Die Abbildung $\text{USE}_{\text{LOCAL}}$ gibt an, welche Ecken in dem gegebenen Grundblock zugegriffen werden.

Dann gibt die Lösung $\text{USE}_{\text{IN}}(n)$ des Problems an, welche Variablen in der Funktion benutzt worden sein können, wenn der Anfang des Grundblocks n erreicht wird. Die Abbildung $\text{USE}_{\text{OUT}}(n)$ gibt dasselbe für das Ende des Grundblocks an.

Dieses Gleichungssystem kann so, wie es hier geschrieben ist, nur dann in einem Durchlauf gelöst werden, wenn keine Schleifen vorhanden sind.

Wenn Schleifen, also Zyklen im Kontrollflußgraphen, vorhanden sind, müssen wir jedoch eine Fixpunktiteration vornehmen. Dies wird in dem Beispiel aus Abbildung 2 gezeigt.

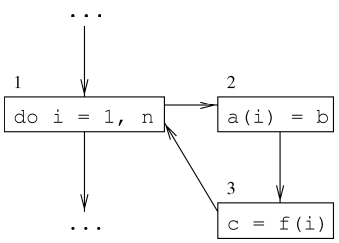


Abbildung 2: Beispiel für Datenflußgleichungen

In dem Beispiel ist $U = \{i, n, a, b, c\}$ (`f` sei eine Funktion). Die lokale Analyse ergibt: $\text{USE}_{\text{LOCAL}}(1) = \{i, n\}$, $\text{USE}_{\text{LOCAL}}(2) = \{a, b, i\}$ und $\text{USE}_{\text{LOCAL}}(3) = \{c, i\}$. Eine endgültige Berechnung für $\text{USE}_{\text{IN}}(1)$ ist nicht möglich, da $\text{USE}_{\text{OUT}}(3)$ noch nicht bekannt ist. Jedoch wird $\text{USE}_{\text{IN}}(1)$ benötigt, um $\text{USE}_{\text{OUT}}(1) = \text{USE}_{\text{IN}}(2)$, damit $\text{USE}_{\text{OUT}}(2) = \text{USE}_{\text{IN}}(3)$ und damit $\text{USE}_{\text{OUT}}(3)$ zu berechnen. Deshalb müssen wir eine Fixpunktiteration durchführen. Dabei nehmen wir zunächst an, daß die bisher nicht bekannten Werte die leere Menge sind.

Im Beispiel ergibt sich bei der ersten Iteration:

- $\text{USE}_{\text{IN}}(1) = \emptyset$
- $\text{USE}_{\text{OUT}}(1) = \{i, n\}$
- $\text{USE}_{\text{IN}}(2) = \{i, n\}$
- $\text{USE}_{\text{OUT}}(2) = \{a, b, i, n\}$
- $\text{USE}_{\text{IN}}(3) = \{a, b, i, n\}$
- $\text{USE}_{\text{OUT}}(3) = \{a, b, c, i, n\}$.

Die zweite Iteration ergibt dann die Lösung wie folgt:

- $\text{USE}_{\text{IN}}(1) = \{a, b, c, i, n\}$.

In diesem Beispiel sind dann alle anderen Lösungsmengen gleich, da $\text{USE}_{\text{IN}}(1)$ jetzt das ganze Universum umfaßt und in den Datenflußgleichungen nur Mengenvereinigungen auftreten.

2.3 Interprozedurale Analyse

Die intraprozedurale Analyse geht davon aus, daß die Grundblöcke leicht zu analysieren sind. Dies ist jedoch unter anderem für Funktionsaufrufe so nicht der Fall. Es gibt mehrere Ansätze, dieses Problem zu behandeln:

- Der pessimistische Ansatz: Bei einem Prozeduraufruf wird für den Grundblock, in dem die Prozedur aufgerufen wird, stets das „schlechteste“ angenommen. Wenn zum Beispiel untersucht wird, welche Definitionen eine Variablenreferenz erreichen können, wird davon ausgegangen, daß die Prozedur die Variable modifizieren und damit eine vorhergehende Definition ungültig machen kann – es sei denn, die Variable kann für keine aufgerufene Prozedur zugreifbar sein.
- Vollständige Expansion: An allen Stellen, wo eine Prozedur aufgerufen wird, wird deren Definition eingesetzt. Dies ist nur möglich, wenn die Prozedurdefinition dem Übersetzer bekannt ist und keine direkte oder indirekte Rekursion auftritt. Außerdem führt diese Methode zu einer starken Vergrößerung des Codes und zur wiederholten Analyse der selben Codestücke. Dafür ist diese Methode so exakt wie möglich.

- Ein Durchlauf: Jede Prozedur im Programm wird genau einmal analysiert. Dabei wird das Wissen verwertet, was über früher analysierte Prozeduren gewonnen wurde. Bei Aufrufen von noch nicht analysierten Prozeduren müssen wieder pessimistische Annahmen getroffen werden. Die Reihenfolge der Analyse bestimmt, wie präzise diese Methode ist. Wenn zum Beispiel keine Rekursion auftritt, kann man eine topologische Sortierung des Aufrufgraphen benutzen, um aufgerufene Prozeduren vor den aufrufenden zu analysieren.
- Iterative Analyse: Nach einem Durchlauf der Analyse werden die Prozeduren erneut analysiert, wobei über aufgerufene Prozeduren nun mehr Information vorliegt. Die Prozeduren werden so oft analysiert, bis sich die Ergebnisse der Analyse stabilisieren.
- Symbolische Analyse: Statt für Prozeduren bestimmte Fragen mit einem Bit zu beantworten (z.B. „Modifiziert die Prozedur die Variable `x`?“), wird die Antwort symbolisch als logischer Ausdruck gegeben. Die Variablen in diesem Ausdruck entsprechen den Informationen über den Kontext, in dem die Prozedur aufgerufen wird, und die entscheiden, ob die Frage mit ja oder nein zu beantworten ist.
- Gemischte Strategien: Dabei werden mehrere Ansätze kombiniert. Ein Beispiel ist, daß externe Prozeduraufrufe pessimistisch analysiert werden, während für Prozeduren, die in der selben Übersetzungseinheit definiert sind, eine der anderen Methoden oder eine Kombination eingesetzt wird, zum Beispiel Expansion für kurze Prozeduren und Ein-Durchlauf-Analyse für die anderen.

Bei manchen rekursiven Prozeduren kann die Analyse dadurch erleichtert werden, daß sie während der Analyse in ihr iteratives Äquivalent umgeschrieben werden.

In der Wiederverwendungsanalyse, die in Abschnitt 3.2 vorgestellt wird, wird auch eine Strategie für die interprozedurale Analyse angerissen.

3 Datenflußanalyse für Arrays

Nach der Einführung in Abschnitt 2 wird nun das eigentliche Thema dieser Ausarbeitung besprochen. Die meisten herkömmlichen Übersetzer betrachten Reihungen als unteilbare Einheiten. In dem Codestück

```
a(1) = ...
a(2) = ...
... = ... a(1) ...
```

wird daher zum Beispiel nicht unbedingt erkannt, daß die Definition für `a(1)` aus der ersten Zeile in der dritten noch gültig ist, da der Übersetzer annimmt, daß die Zuweisung von `a(2)` in der zweiten Zeile das ganze Array modifiziert. Auf diese Weise wird eine Optimierung behindert, die sonst den Code der dritten Zeile mit Hilfe der Kenntnis über die rechte Seite in der ersten Zeile möglicherweise verbessern könnte.

Auch für Zwecke der automatischen oder halbautomatischen Parallelisierung von Programmen sowie der Optimierung des Cache-Verhaltens eines Programms sind genauere Informationen über Arrayzugriffe und Zugriffsmuster, wenn die Zugriffe in Schleifen stattfinden und die Indizes von den Schleifenzählern abhängen, von großem Nutzen.

Im folgenden werden drei verschiedene Ansätze der Datenflußanalyse vorgestellt, die auch Arrayzugriffe analysieren.

3.1 Exakte Datenabhängigkeiten

Die Datenflußanalysetechnik, die in [Fea1] vorgestellt wird, kann im Vorhandensein von Schleifen die genauen Abhängigkeiten von Reihungszugriffen feststellen, allerdings unter deutlichen Einschränkungen in der Quellsprache.

Die Analysetechnik von Feautrier konzentriert sich auf die Datenflußabhängigkeiten[4], zumal die anderen Abhängigkeiten durch Variablenumbenennung oder -expansion eliminiert werden können.

3.1.1 Einschränkungen in der Quellsprache

Um eine exakte Analyse zu ermöglichen, wird die Quellsprache eingeschränkt: Als Datentypen sind nur ganze Zahlen, Gleitkommazahlen und n-dimensionale Reihungen von diesen zugelassen. Einfache Anweisungen sind die Zuweisungen von Skalaren und Reihungen. Die Kontrollstrukturen sind die Sequenz und Zählschleifen. Die Ausdruckssyntax wird um bedingte Ausdrücke der Form

```
if <boolean expression> then <expression> else <expression>
```

erweitert. Insbesondere fehlen `gotos`, bedingte Anweisungen, `while`-Schleifen und Prozeduren.

Ein Programm hat statische Kontrolle, wenn die Grenzen für Schleifenzähler nur von Strukturkonstanten[5], numerischen Konstanten oder äußeren Schleifenzählern abhängen.

Die hier vorgestellte Technik schränkt die Programme auf statische Kontrolle ein. Zusätzlich müssen die Schrittweite der Schleifen 1 sein, die Grenzen von Schleifen und Indizes für Reihungen affine Funktionen von den Schleifenzählern und Strukturkonstanten sein. Wenn bei Zählschleifen die untere Schleifengrenze größer als die obere ist, wird die Schleife nicht ausgeführt.

Es wird angenommen, daß das Programm korrekt ist, insbesondere daß die Indizes für Reihungen innerhalb der Indexgrenzen liegen. Damit haben alle Zugriffe auf ein Reihungselement den selben Indexvektor.

3.1.2 Sequenzprädikat

Jede Anweisung (einfache und zusammengesetzte) erhält einen eindeutigen Namen. Da die einzige Kontrollstruktur, die mehrfache Ausführung von Anweisungen erlaubt, die `for`-Schleife ist, kann eine Ausführung einer Anweisung durch eine Anweisungskoordinate, nämlich das Paar $(r, \vec{a})$ beschrieben werden, wobei r der Name der Anweisung und $\vec{a}$ der Iterationsvektor ist. Der Iterationsvektor ist dabei ein Vektor mit so vielen Elementen wie die Schachtelungstiefe, in der die Anweisung r auftritt. Die Komponenten entsprechen den Werten der Schleifenzähler der Schleifen, in denen r enthalten ist.

Eine Anweisungskoordinate $(r, \vec{a})$ ist dann gültig, wenn die Dimension von $\vec{a}$ gleich der Schachtelungstiefe der Schleifen, in denen r enthalten ist, ist und alle Koordinaten von $\vec{a}$ innerhalb den entsprechenden Schleifengrenzen liegen.

Diese Bedingungen lassen sich als Gleichung der Form $\vec{e}_r(\vec{a}) \geq 0$ schreiben, wobei $\vec{e}_r$ eine von der Anweisung r abhängige vektorwertige affine Funktion ist und $\geq$ bedeutet, daß alle Komponenten die Relation erfüllen müssen.

Man kann nun für alle Anweisungen den Iterationsbereich definieren. Der Iterationsbereich der Anweisung r ist die Menge der gültigen Iterationsvektoren für r. Dies ist eine Teilmenge eines n-dimensionalen Vektorraums, wenn n die Dimension des Iterationsvektors ist. Genauso kann man für Reihungszugriffe den Datenraum definieren, nämlich als die Menge der Indexvektoren, die der gegebene Zugriff durchläuft. Dies ist eine Teilmenge eines k-dimensionalen Vektorraums, wenn k die Dimension der Reihung ist.

Um das zeitliche Verhalten zu beschreiben, definieren wir das Sequenzprädikat: $(r, \vec{a}) \prec (s, \vec{b})$. Dies bedeutet, daß Anweisung r mit Iterationsvektor $\vec{a}$ vor Anweisung s mit Iterationsvektor $\vec{b}$ ausgeführt wird. Außerdem soll T_{rs} genau dann wahr sein, wenn r textuell vor s im Programm steht.

Wenn nun r und s nicht in Schleifen auftreten, dann gilt:

$$(r, []) \prec (s, []) \equiv T_{rs}\,.$$

Nun nehmen wir an, r und s seien die selbe Anweisung. Dann gilt, daß $(r, \vec{a}) \prec (r, \vec{b})$ genau dann, wenn $\vec{a}$ lexikographisch vor $\vec{b}$ kommt.

[4] Auch Producer-Consumer-Abhängigkeiten genannt

[5] Konstanten, die einmal definiert werden, zum Beispiel durch Eingabeanweisungen

Für den allgemeinen Fall definieren wir N_{rs} als die maximale Tiefe der Schleifenschachteln, in denen r und s gemeinsam enthalten sind. In dieser Schachtel gibt es Anweisungen r' und s', worin r bzw. s textuell enthalten sind. Offensichtlich gilt:

$$(r, \vec{a}) \prec (s, \vec{b}) \equiv (r', \vec{a}[1 \dots N_{rs}]) \prec (s', \vec{b}[1 \dots N_{rs}]) \,.$$

Wenn nun $\vec{a}[1 \dots N_{rs}]$ ungleich $\vec{b}[1 \dots N_{rs}]$ ist, gehören $(r, \vec{a})$ und $(s, \vec{b})$ zu verschiedenen Iterationen von t. In diesem Fall wird die Ordnung durch die lexikographische Ordnung von $\vec{a}[1 \dots N_{rs}]$ und $\vec{b}[1 \dots N_{rs}]$ gegeben. Wenn die Vektoren jedoch gleich sind, wird die Sequenzierung durch die textuelle Ordnung, also $T_{r's'} = T_{rs}$ festgelegt. Zusammen:

$$(r, \vec{a}) \prec (s, \vec{b}) \equiv \vec{a}[1 \dots N_{rs}] \ll \vec{b}[1 \dots N_{rs}] \vee (\vec{a}[1 \dots N_{rs}] = \vec{b}[1 \dots N_{rs}] \wedge T_{rs}) \,.^{6}$$

Also reicht die Kenntnis von N_{rs} und T_{rs} aus, um die Sequenz aller Anweisungen im Programm zu kennen.

3.1.3 Datenflußanalyse

Wenn in einer Anweisung $(t, \vec{b})$ eine Reihung M zugegriffen wird mit dem Indexvektor $\vec{g}(\vec{b})$, suchen wir dazu nun die Anweisung, die diesen Wert berechnet. Dazu betrachten wir alle Anweisungen s_i, die einen Wert für M produzieren, mit ihren Iterationsvektoren $\vec{a}_i$. s_i ist von der Form

$$M[\vec{f}_i(\vec{a}_i)] := \cdots \,.$$

Die Quelle ist eine Anweisungskoordinate, die in Abhängigkeit von $\vec{b}$ angibt, wo der zugegriffene Wert produziert wird. Die Abbilding von $\vec{b}$ auf diese Quelle nennen wir Quellfunktion von $M[\vec{g}(\vec{b})]$.

Wenn nun die Quelle von $M[\vec{g}(\vec{b})]$ eine Instanz von s_i ist, gibt es ein eindeutiges $\vec{a}_i$, so daß die Instanz $(s_i, \vec{a}_i)$ ist. Dieses $\vec{a}_i$ ist eine Funktion von $\vec{b}$, die wir $\vec{K}_{s_i t}$ nennen. Die tatsächliche Quelle ist dann die späteste Anweisung $(s_i, \vec{K}_{s_i t}(\vec{b}))$. Der Wert für i kann natürlich von $\vec{b}$ abhängen und $\vec{K}_{s_i t}$ kann für einige Werte von $\vec{b}$ undefiniert sein. Die undefinierte Anweisung schreiben wir $\perp$. Für alle $(r, \vec{c})$ gilt $\perp \prec (r, \vec{c})$.

Die Bedingungen für $\vec{K}_{s_i t}(\vec{b})$ sind:

- $(s_i, \vec{K}_{s_i t}(\vec{b}))$ muß einen Wert für $M[\vec{g}(\vec{b})]$ berechnen:

$$\vec{f}_i(\vec{K}_{s_i t}(\vec{b})) = \vec{g}(\vec{b})$$

- $(s_i, \vec{K}_{s_i t}(\vec{b}))$ muß vor $(t, \vec{b})$ kommen:

$$(s_i, \vec{K}_{s_i t}(\vec{b})) \prec (t, \vec{b})$$

- $\vec{K}_{s_i t}(\vec{b})$ muß eine gültige Koordinate sein:

$$\vec{e}_{s_i}(\vec{K}_{s_i t}(\vec{b})) \geq 0$$

- $(s_i, \vec{K}_{s_i t}(\vec{b}))$ muß die späteste Koordinate sein, die die obigen Bedingungen erfüllt:

$$\vec{f}_i(\vec{u}) = \vec{g}(\vec{b}) \wedge (s_i, \vec{i}) \prec (t, \vec{b}) \wedge \vec{e}_{s_i}(\vec{i}) \geq 0 \Rightarrow \vec{i} \ll \vec{K}_{s_i t}(\vec{b})$$

Wenn $\max_{\ll}$ das lexikographische Maximum bedeutet, ist

$$\vec{K}_{s_i t}(\vec{b}) = \max_{\ll} Q_{s_i t}(\vec{b}) \,,$$

[6] $\ll$ ist die lexikographische Ordnung von Vektoren

wobei $Q_{s_i t}$ die Menge

$$Q_{s_i t} := \{\vec{u} | \vec{f}_i(\vec{u}) = \vec{g}(\vec{b}),\ (s_i, \vec{u}) \prec (t, \vec{b}), \vec{e}_{s_i}(\vec{u}) \geq 0\}$$

ist und das lexikographische Maximum der leeren Menge die undefinierte Anweisung $\perp$ ist.

Da sich die lexikographische Ordnung und damit auch das Sequenzprädikat als logische Verknüpfung von linearen Gleichungen und Ungleichungen ausdrücken läßt, bilden die Anweisungen die den ersten drei der obigen Bedingungen entsprechen, eine Menge, die durch ein System von linearen Gleichungen und Ungleichungen beschrieben wird. Von dieser Lösungsmenge suchen wir das Maximum bezüglich des Sequenzprädikats. Damit wird letztendlich eine Instanz eines Parametric Integer Programs (PIP) beschrieben.

Ein Lösungsalgorithmus für diese Parametric Integer Programs wird im Anhang von [Fea1] vorgestellt. In [Fea2] wird Parametric Integer Programmming allgemein behandelt.

3.2 Wiederverwendung

Bei der Wiederverwendungsanalyse, wie sie in [Coo] beschrieben wird, wird mit Hilfe von Datenflußgleichungen die Wiederverwendung von Werten (Skalaren, Reihungen oder Teilen davon) analysiert. Ein besonderes Augenmerk wird auf die Wiederverwendung über Schleifengrenzen hinaus gelegt.

Die Analyse erfolgt mit Hilfe von Datenflußgleichungen in einem Tarjan-Intervallflußgraphen. Das Universum U ist die Menge aller Datenzugriffsdeskriptoren, hier kurz als Deskriptoren bezeichnet. Ein Deskriptor repräsentiert den Zugriff (schreibend oder lesend) auf einen Skalar, ein Reihungselement oder einen Abschnitt in einer Reihung.

Für die Datenflußanalyse brauchen wir bestimmte Abbildungen von der Menge der Grundblöcke auf die Potenzmenge des Universums U sowie Gleichungen, deren Lösung diese Abbildungen definiert.

Durch lokale Analyse wird jedem Grundblock n eine Menge von Deskriptoren $\textsc{Gen}_{\text{INIT}}(n)$ zugeordnet, die die Zugriffe in diesem Grundblock beschreiben. Der Kontrollflußgraph wird so transformiert, daß nur eine CYCLE-Kante einen Intervallkopf erreicht und daß von diesem nur eine ENTRY-Kante ausgeht.[7] Mit $\textsc{Lastchild}(n)$ ist für einen Intervallkopf n die Ecke bezeichnet, von der die CYCLE-Ecke ausgeht, die n erreicht. Mit $\textsc{Header}(n)$ bezeichnen wir Kopf des Tarjan-Intervalls, das n am engsten umfaßt.

$$\textsc{Gen}_{\text{IN}}(n) = \bigwedge_{p \in \textsc{Pred}_{\text{F}}(n)} \textsc{Gen}_{\text{OUT}}(p) \tag{6}$$

$$\textsc{Gen}_{\text{LOC}}(n) = \begin{cases} \textsc{Gen}_{\text{OUT}}(\textsc{Lastchild}(n)) & \text{wenn } n \text{ Intervallkopf} \\ \textsc{Gen}_{\text{INIT}}(n) & \text{sonst} \end{cases} \tag{7}$$

$$\textsc{Gen}_{\text{OUT}}(n) = \textsc{Gen}_{\text{IN}}(n) \vee \textsc{Gen}_{\text{LOC}}(n) \tag{8}$$

$$\textsc{Reach}_{\text{IN}}(n) = \begin{cases} \textsc{Reach}_{\text{IN}}(\textsc{Header}(n)) & \textsc{Pred}_{\text{E}}(n) \neq \emptyset \\ \bigwedge_{p \in \textsc{Pred}_{\text{F}}(n)} \textsc{Reach}_{\text{OUT}}(p) & \text{sonst} \end{cases} \tag{9}$$

$$\textsc{Reach}_{\text{OUT}}(n) = \textsc{Reach}_{\text{IN}}(n) \vee \textsc{Gen}_{\text{LOC}}(n) \tag{10}$$

Abbildung 3: Datenflußgleichungen für die Wiederverwendungsanalyse

Die Datenflußgleichungen sind in Abbildung 3 gezeigt. Dieses Gleichungssystem kann in einem Durchlauf gelöst werden, indem zuerst die Gleichungen (6), (7) und (8) in AUFWÄRTS-VORWÄRTS-Reihenfolge, dann die Gleichungen (9) und (10) in ABWÄRTS-VORWÄRTS-Reihenfolge ausgewertet werden.

Die letztendlich interessante Abbildung ist $\textsc{Reach}_{\text{IN}}(n)$. Diese Menge sagt aus, welche Reihungsabschnitte auf allen Pfaden von ROOT bis zum Beginn von n schon benutzt wurden. Die Hilfsabbildung $\textsc{Gen}_{\text{IN}}(n)$ gibt an, welche Abschnitte auf allen Pfaden vom Anfang des Intervalls, das n enthält, bis zum Anfang n verwendet werden. $\textsc{Gen}_{\text{LOC}}(n)$ gibt an, welche Abschnitte in n, oder wenn n Intervallheader

[7] Es werden künstliche Ecken im Intervallflußgraph entsprechend eingefügt.

ist, $\{n\} \cup T(n)$, verwendet werden. GEN_{OUT} ist die Kombination von GEN_{IN} und GEN_{LOC}. Ähnlich ist $\text{REACH}_{\text{OUT}}$ die Kombination von REACH_{IN} und GEN_{LOC}.

Wenn wir einfach beliebige Wiederverwendung analysieren wollen, können die Operatoren $\vee$ und $\wedge$ als Mengenvereinigung bzw. Durchschnitt implementiert werden.

Um jedoch Cache-Effekte zu berücksichtigen, müssen die Operatoren entsprechend modifiziert werden. Dazu wird das Universum für die Datenflußanalyse so modifiziert, daß die Elemente nicht mehr die Deskriptoren sind, sondern Paare aus Deskriptoren und einer Altersangabe. Das Alter eines Reihungsausschnitts soll der Anzahl der Cache-Misses entsprechen, die höchstens aufgetreten sind, seitdem das erste Element des Ausschnitts in den Cache gebracht wurde. Die Ausschnitte in den GEN_{INIT}-Mengen bekommen das „Volumen“ der Ausschnitte als Alter zugewiesen. Das Volumen ist z.B. die Anzahl der Cache-Zeilen, die für diesen Ausschnitt benötigt werden. Wenn sich ein Ausschnitt mittels der Datenflußgleichungen über das Programm ausbreitet, wird das Alter um das Volumen des durchlaufenen Grundblocks erhöht. Das Volumen des Grundblocks ist z.B. die Anzahl der Cache-Misses in dem durchlaufenen Codestück. Ab einer bestimmten Schwelle, z.B. der Cache-Größe, wird der Ausschnitt dann als „tot“ angesehen und aus den Mengen gestrichen.

In dieser Analyse werden die Effekte begrenzter Cache-Assoziativität nicht präzise analysiert. Stattdessen werden sie dadurch angenähert, daß mit einer kleineren effektiven Cache-Größe gerechnet wird. Die tatsächliche Cache-Größe wird mit $1 - \frac{1}{2^S}$ multipliziert, wobei S die Assoziativität des Caches angibt.

procedure *FiniteCache-*$\vee$
inputs: IN (GEN_{IN}/REACH_{IN}-Menge),
LOC (GEN_{LOC}-Menge)
outputs: OUT (GEN_{OUT}/$\text{REACH}_{\text{OUT}}$-Menge)
begin
  `volume` $\leftarrow 0$, $R \leftarrow \emptyset$
  forall `x` $\in LOC$:
    if $\exists$ `y` $\in IN$, `y` in `x` enthalten **then**
      $R \leftarrow R \cup \{$`y`$\}$
    else
      `volume` $\leftarrow$ `volume` + Volumen von `x`
    endif
  endfor
  $OUT \leftarrow LOC \cup (IN \setminus R)$
  forall `x` $\in OUT$:
    Alter von `x` um `volume` erhöhen
    `x` von OUT entfernen, wenn Alter die Grenze überschreitet
  endfor
end

Abbildung 4: Implementierung des Operators $\vee$

Ein Algorithmus zur Berechnung des Operators $\vee$ für endliche Caches ist in Abbildung 4 dargestellt.

Der Operator $\wedge$ ist weiterhin der Durchschnitt der Abschnitte in den einzelnen Mengen. Wenn ein Abschnitt in beiden Mengen mit unterschiedlichem Alter vorkommt, wird das höhere Alter übernommen.

In diesem Ansatz zur Wiederverwendungs-Analyse kann der interprozedurale Aspekt relativ einfach berücksichtigt werden. Wir benutzen die $\text{REACH}_{\text{OUT}}$-Menge der Prozedur[8] als GEN_{LOC}-Menge des Grundblocks, in dem sich der Prozeduraufruf befindet, wobei die Variablennamen entsprechend des Prozeduraufrufs übersetzt werden (z.B. Einsetzen der formalen Parameter). Die Begrenzungen dieses interprozeduralen Ansatzes sind, daß der Kontext, in dem die Prozedur aufgerufen wird, nicht für die Analyse dieser Prozedur verwendet wird (als REACH_{IN}-Menge) und daß diese Art der Analyse nur für begrenzte Caches durchgeführt werden kann, da sonst die Deskriptormengen schnell zu groß werden.

[8] Wenn die Prozedur von mehreren Grundblöcken aus zurückkehren kann, können wir die $\text{REACH}_{\text{OUT}}$-Mengen mit dem Operator $\wedge$ verknüpfen.

3.3 Give-N-Take

Die in diesem Abschnitt vorgestellte Technik ist wie die Technik von Feautrier aus dem Forschnungsgebiet der Übersetzung für Parallelrechner entstanden und wird in Kapitel 3 in [Han] beschrieben. Es wird mit einem System von Datenflußgleichungen ermittelt, wo für Architekturen mit verteiltem Speicher Anweisungen für den Nachrichtenaustausch generiert werden sollen, um nichtlokale Daten zu beschaffen bzw. lokal berechnete Daten dem Eigentümer der entsprechenden Variablen zurückzuübermitteln. Reihungszugriffe werden dabei analysiert und die Datenflußgleichungen sind darauf angelegt, auch mit unregelmäßigen Problemen zurechtzukommen.

3.3.1 Definitionen

Das Modell, das der Give-N-Take-Analyse unterliegt, geht von folgenden Voraussetzungen aus:

- Sowohl Absender als auch Empfänger von Daten müssen zueinander passende Kommunikationsanweisungen ausführen.
- Wenn ein Prozessor eine Stelle erreicht, in der Kommunikation nötig ist, müssen alle in die Kommunikation involvierten Prozessoren diese Stelle erreichen.

Es gibt zwei Charakterisierungen von Problemen, auf die die Give-N-Take-Analyse anwendbar ist:

- Vorher/Nachher: Bei Vorher-Problemen muß ein Datum vor seiner Verwendung produziert werden, z.B. Lesezugriff auf nicht lokale Daten. Bei Nachher-Problemen muß die Produktion nach dem Zugriff stattfinden, z.B. dem Zurückschreiben von veränderten Daten. Die nachfolgende Darstellung geht von einem Vorher-Problem aus.
- Faul/Fleissig: Bei Fleissig-Problemen wird die Produktion möglichst weit von der Verwendung entfernt generiert. Z.B. frühestmögliches Absenden der Anforderung nach nichtlokalen Daten für einen Lesezugriff, oder spätestmögliches Warten auf die Bestätigung beim Zurückschreiben von Daten. Bei Faul-Problemen wird die Produktion möglichst nahe bei der Verwendung generiert.

Bei Give-N-Take gibt es drei Arten der Manipulation von Daten:

- Eine Referenz zu Daten, die der betreffende Prozessor nicht selbst besitzt, *konsumiert* diese Daten.
- Jede Read-Operation (Anforderung von Daten vom besitzenden Prozessor) *produziert* diese Daten.
- Jede nicht-lokale Neudefinition von Daten, die der Prozessor auch nicht selbst besitzt, *zerstört* diese Daten.

Der Intervallflußgraph wird für die Give-N-Take-Analyse noch wie folgt modifiziert:

- Für jede Jump-Kante (n_1, n_2) werden synthetische Kanten erzeugt, die von den Intervallköpfen der Intervalle, in denen n_1, aber nicht n_2 enthalten ist, nach n_2 gehen. Die Anzahl dieser Kanten ist also $\text{Level}(n_1) - \text{Level}(n_2)$.
- Von jedem Intervallkopf geht genau eine Entry-Kante aus.
- Jeden Intervallkopf erreicht höchstens eine Cycle-Kante.
- Es gibt keine kritischen Kanten. Eine kritische Kante ist eine Kante, die von einer Ecke ausgeht, von der auch andere Kanten ausgehen, und eine Ecke erreicht, die auch von anderen Kanten erreicht wird.

Die letzten drei Eigenschaften können durch das Einfügen künstlicher Ecken erreicht werden.

3.3.2 Die Datenflußgleichungen

Das Universum für die Datenflußgleichungen enthält wieder Reihungsteile als Elemente. Diese Reihungsteile entsprechen den zugegriffenen (also benötigten) Teilen von Reihungen. Die genaue Darstellung ist für die Datenflußgleichungen nicht von Belang.

Durch lokale Analyse werden folgende Abbildungen definiert:

- $\textsc{Steal}_{\textsc{init}}(n)$: Alle Elemente, deren Produktion in n zunichte gemacht wird. Diese Menge kann auch benutzt werden, um bestimmte Verschiebungen von Produktionen zu verhindern. Dies beinhaltet Reihungsportionen, die in n teilweise modifiziert werden, oder auch Portionen, die selbst modifiziert werden, z.B. weil sie aus indirekten Referenzen resultieren.
- $\textsc{Give}_{\textsc{init}}(n)$: Elemente, die „kostenlos“ in n produziert werden.
- $\textsc{Take}_{\textsc{init}}(n)$: Die in n konsumierten Elemente. In unserem Fall sind das die Reihungsreferenzen, deren Daten der betreffende Prozessor nicht selbst besitzt.

In Abbildung 5 sind die Flußgleichungen gezeigt. Die Bedeutungen der Abbildungen werden im folgenden erklärt:

- $\textsc{Steal}(n)$: Alle Elemente, die von n oder einem Element von $T(n)$ vernichtet werden, ohne danach neu produziert zu werden.
- $\textsc{Give}(n)$: Alle Elemente, die von n oder einem Element von $T(n)$ produziert werden, ohne danach wieder vernichtet zu werden.
- $\textsc{Block}(n)$: Elemente, deren Produktion nicht über n hinweg verschoben werden kann, weil sie in n oder einem Element von $T(n)$ vernichtet oder schon produziert werden.
- $\textsc{Taken}_{\textsc{out}}(n)$: Elemente, die garantiert auf allen Pfaden, die von n ausgehen, konsumiert werden, bevor sie vernichtet werden. Konsumption in n selbst wird hier nicht berücksichtigt.
- $\textsc{Take}(n)$: Diese Elemente werden in n konsumiert, oder garantiert in $T(n)$, ohne in n vorher vernichtet zu werden, oder sie *können* in $T(n)$ konsumiert werden *und* werden garantiert nach n konsumiert.
- $\textsc{Taken}_{\textsc{in}}(n)$: Wie $\textsc{Taken}_{\textsc{out}}(n)$, aber n selbst wird berücksichtigt.
- $\textsc{Block}_{\textsc{loc}}(n)$: Elemente, die von n oder Nachfolgern von n im selben Intervall blockiert werden, ohne vorher konsumiert zu werden.
- $\textsc{Take}_{\textsc{loc}}(n)$: Elemente, die von n oder Nachfolgern im selben Intervall konsumiert werden.
- $\textsc{Give}_{\textsc{loc}}(n)$: Elemente, die von n oder Vorgängern im selben Intervall produziert werden. Elemente, die in n oder Vorgängern konsumiert werden, werden auch berücksichtigt, da diese Konsumption auf jeden Fall durch eine vorhergehende Produktion befriedigt wird.
- $\textsc{Steal}_{\textsc{loc}}(n)$: Elemente, die von n oder einem Vorgänger im selben Intervall vernichtet werden.
- $\textsc{Given}_{\textsc{in}}(n)$: Elemente, die auf jeden Fall am Anfang[9] von n verfügbar sind.
- $\textsc{Given}(n)$: Elemente, die bei n selbst verfügbar sind, entweder weil sie von allen Vorgängern kommen, oder weil sie in n selbst konsumiert werden, oder bei einem Fleissig-Problem von einem Nachfolger von n.
- $\textsc{Given}_{\textsc{out}}(n)$: Elemente, die am Ende von n verfügbar sind. Diese Menge enthält alles, was von n selbst kommt, schließt aber aus, was in n vernichtet wird.

[9] Für Nachher-Probleme am Ende

$$\textsc{Steal}(n) = \textsc{Steal}_{\textsc{init}}(n) \cup \textsc{Steal}_{\textsc{loc}}(\textsc{Lastchild}(n)) \quad (11)$$

$$\textsc{Give}(n) = \textsc{Give}_{\textsc{init}}(n) \cup \textsc{Give}_{\textsc{loc}}(\textsc{Lastchild}(n)) \quad (12)$$

$$\textsc{Block}(n) = \textsc{Steal}(n) \cup \textsc{Give}(n) \cup \bigcup_{s \in \textsc{Succ}_{\textsc{e}}(n)} \textsc{Block}_{\textsc{loc}}(s) \quad (13)$$

$$\textsc{Taken}_{\textsc{out}}(n) = \bigcap_{s \in \textsc{Succ}_{\textsc{fjs}}(n)} \textsc{Taken}_{\textsc{in}}(s) \quad (14)$$

$$\textsc{Take}(n) = \textsc{Take}_{\textsc{init}}(n) \cup (\textstyle\bigcup_{s \in \textsc{Succ}_{\textsc{e}}(n)} \textsc{Taken}_{\textsc{in}}(s) \setminus \textsc{Steal}(n)) \cup ((\textsc{Taken}_{\textsc{out}}(n) \cap \textstyle\bigcup_{s \in \textsc{Succ}_{\textsc{e}}(n)} \textsc{Take}_{\textsc{loc}}(s)) \setminus \textsc{Block}(n)) \quad (15)$$

$$\textsc{Taken}_{\textsc{in}}(n) = \textsc{Take}(n) \cup (\textsc{Taken}_{\textsc{out}}(n) \setminus \textsc{Block}(n)) \quad (16)$$

$$\textsc{Block}_{\textsc{loc}}(n) = (\textsc{Block}(n) \cup \textstyle\bigcup_{s \in \textsc{Succ}_{\textsc{f}}(n)} \textsc{Block}_{\textsc{loc}}(s)) \setminus \textsc{Take}(n) \quad (17)$$

$$\textsc{Take}_{\textsc{loc}}(n) = \textsc{Take}(n) \cup (\textstyle\bigcup_{s \in \textsc{Succ}_{\textsc{ef}}(n)} \textsc{Take}_{\textsc{loc}}(s) \setminus \textsc{Block}(n)) \quad (18)$$

$$\textsc{Give}_{\textsc{loc}}(n) = (\textsc{Give}(n) \cup \textsc{Take}(n) \cup \textstyle\bigcap_{p \in \textsc{Pred}_{\textsc{fj}}(n)} \textsc{Give}_{\textsc{loc}}(p)) \setminus \textsc{Steal}(n) \quad (19)$$

$$\textsc{Steal}_{\textsc{loc}}(n) = \textsc{Steal}(n) \cup \textstyle\bigcup_{p \in \textsc{Pred}_{\textsc{fj}}(n)} (\textsc{Steal}_{\textsc{loc}}(p) \setminus \textsc{Give}_{\textsc{loc}}(p)) \cup \bigcup_{p \in \textsc{Pred}_{\textsc{s}}(n)} \textsc{Steal}_{\textsc{loc}}(p) \quad (20)$$

$$\textsc{Given}_{\textsc{in}}(n) = \textsc{Given}(\textsc{Header}(n)) \cup \textstyle\bigcap_{p \in \textsc{Pred}_{\textsc{fj}}(n)} \textsc{Given}_{\textsc{out}}(p) \cup (\textsc{Taken}_{\textsc{in}}(n) \cap \textstyle\bigcup_{q \in \textsc{Pred}_{\textsc{fj}}(n)} \textsc{Given}_{\textsc{out}}(q)) \quad (21)$$

$$\textsc{Given}(n) = \textsc{Given}_{\textsc{in}}(n) \cup \begin{cases} \textsc{Taken}_{\textsc{in}}(n) & \textsc{Fleissig}\text{-Problem} \\ \textsc{Take}(n) & \textsc{Faul}\text{-Problem} \end{cases} \quad (22)$$

$$\textsc{Given}_{\textsc{out}}(n) = (\textsc{Give}(n) \cup \textsc{Given}(n)) \setminus \textsc{Steal}(n) \quad (23)$$

$$\textsc{Res}_{\textsc{in}}(n) = \textsc{Given}(n) \setminus \textsc{Given}_{\textsc{in}}(n) \quad (24)$$

$$\textsc{Res}_{\textsc{out}}(n) = \bigcup_{s \in \textsc{Succ}_{\textsc{fj}}(n)} \textsc{Given}_{\textsc{in}}(s) \setminus \textsc{Given}_{\textsc{out}}(n) \quad (25)$$

Abbildung 5: Datenflußgleichungen für Give-N-Take

- $\text{RES}_{\text{IN}}(n)$: Produktionen, die am Anfang von n generiert werden müssen. Dies enthält alles, was in n selbst, aber nicht am Anfang von n garantiert verfügbar ist.
- $\text{RES}_{\text{OUT}}(n)$: Produktionen am Ende von n. Dies enthält alles, dessen Verfügbarkeit Nachfolgern von n garantiert ist, aber nicht schon am Ende von n verfügbar ist.

3.3.3 Lösung der Gleichungen

procedure *GiveNTake*
inputs: $G = (N, E)$; $\forall n \in N$:
$\text{TAKE}_{\text{INIT}}(n)$, $\text{STEAL}_{\text{INIT}}(n)$, $\text{GIVE}_{\text{INIT}}(n)$
outputs: $\forall n \in N$: $\text{RES}_{\text{IN}}(n)$, $\text{RES}_{\text{OUT}}(n)$ für FLEISSIG und/oder FAUL
begin
  forall $n \in N$, in RÜCKWÄRTS-AUFSTEIGEND-Reihenfolge
    forall $c \in \text{CHILDREN}(n)$, in VORWÄRTS-Reihenfolge
      Berechne Gleichungen 19 und 20
    endfor
    Berechne Gleichungen 11 bis 18
  endfor
  forall $n \in N$, in VORWÄRTS-ABSTEIGEND-Reihenfolge
    Berechne Gleichungen 21 bis 23 für FLEISSIG bzw. FAUL
  endfor
  forall $n \in N$
    Berechne Gleichungen 24 und 25 für FLEISSIG bzw. FAUL
  endfor
end

Abbildung 6: Lösungsalgorithmus für GIVE-N-TAKE

Der Algorithmus in Abbildung 6 ist in der Lage, das Gleichungssystem so zu lösen, daß jede Gleichung nur einmal ausgewertet muß.

4 Anwendungen

Das Hauptaugenmerk dieser Ausarbeitung liegt auf den Datenflußanalysemethoden selbst. Allerdings will ich noch einige Anwendungen, die obigen Analysemethoden im Bereich der Cache-Optimierung haben, motivieren.

4.1 Codetransformationen

In [Coo] wird beschrieben, wie die Wiederverwendungsinformationen, die mit der Methode aus Abschnitt 3.2 gewonnen werden, dazu verwendet werden, abzuschätzen, ob bestimmte Codetransformationen gewinnbringend durchgeführt werden können. Zwei dieser Transformationen sind die Fusion von zwei Schleifen, sowie das Umkehren der Auswertungsreihenfolge von Schleifen.

Eine Fusion lohnt sich dann, wenn die GEN_{LOC}-Menge der zweiten Schleife sich genügend mit der REACH_{IN}-Menge des Kopfes der Schleife überschneidet. Um zu bestimmen, mit welcher Schleife eine Fusion durchgeführt werden kann, muß zu den Datenzugriffsdeskriptoren zusätzlich vermerkt werden, aus welcher Schleife die letzte Verwendung stammt.

Die Analyse für die Umkehrung der Auswertungsreihenfolge (*loop reversal*) benötigt in den Deskriptoren noch Information über die Zugriffsreihenfolge und die Schrittweite.

4.2 Prefetching

Prefetches sind Anweisungen, die den Prozessor dazu veranlassen, eine bestimmte Cache-Zeile zu laden, ohne daß sofort daraus Daten gelesen werden. Nach dem Prefetch-Befehl arbeitet der Prozessor also weitere Befehle ab, während im Hintergrund der vergleichsweise langsame Hauptspeicherzugriff erfolgt.

Wenn später nun ein tatsächlicher Zugriff auf die Speicherstelle erfolgt, ist sie bereits im Cache. So wird die Latenz verdeckt, die sich durch Cache-Misses ergibt.

Die Generierung von Prefetch-Anweisungen ist ein VORHER-FLEISSIG-Problem im Sinne der Definitionen in Abschnitt 3.3. Daher vermute ich, daß GIVE-N-TAKE auch dafür nutzbringend eingesetzt werden kann. Eventuell muß bei der Ausbreitung der Informationen eine Art Alterung erfolgen, ähnlich wie bei der Wiederverwendungsanalyse aus Abschnitt 3.2, um zu verhindern, daß ein Prefetch zu früh durchgeführt wird und die so geholten Daten wieder aus dem Cache verdrängt werden, bevor sie konsumiert werden. Dies kann zum Beispiel dadurch erfolgen, daß $\text{STEAL}(n)$ um diejenigen Elemente von $\text{TAKEN}_{\text{OUT}}(n)$ erweitert werden, deren Alter (bzw. Abstand zur Konsumption) nach dem Durchlaufen von n zu hoch wäre.

Alternativ könnte man dieses Problem auch als ein FAUL-Problem auffassen, wobei eine virtuelle Konsumption in einem Abstand vor der realen Konsumption stattfindet, der gerade die Speicherzugriffslatenz abdeckt.

4.3 Speicherabbildung

Das Thema der Speicherabbildung wird bereits in einer anderen Ausarbeitung behandelt. Daher wird im folgenden nur kurz eine mögliche Interaktion zwischen Datenflußanalyse und Speicherabbildung angerissen.

Die Informationen über Zugriffsmuster auf Reihungen können dazu verwendet werden, die Reihungen im Speicher so anzuordnen, daß möglichst wenige Konflikte bei einem Cache mit geringer Assoziativität auftreten. Solche Informationen werden durch die Analysetechnik von Feautrier (Abschnitt 3.1) und die Wiederverwendungsanalyse (Abschnitt 3.2) gewonnen. Diese Speicherabbildung kann für unterschiedliche Teile des Programms unterschiedlich ausfallen. Dies ist lohnend, wenn der zusätzliche Aufwand durch Umordnen oder Replikation von Reihungen oder Reihungsteilen durch besseres Cache-Verhalten mindestens ausgeglichen wird.

Wenn sich durch eine veränderte Speicherabbildung geänderte Volumina im Sinne der Wiederverwendungsanalyse ergeben, sollte dann eine erneute Wiederverwendungsanalyse durchgeführt werden. Eventuell ergeben sich dadurch andere Vorgaben für eine Speicherabbildung, sowie andere lohnende Codetransformationen (Abschnitt 4.1). In diesem Fall kann sich ein iteratives Verfahren lohnen. Dabei werden Wiederverwendungsanalyse und Speicherabbildung solange abwechselnd durchgeführt, bis sich deren Ergebnisse stabilisieren.

5 Erfahrungen

In [Coo] werden auch empirische Ergebnisse der Codetransformationen (Abschnitt 4.1) unter Zuhilfenahme der Ergebnisse einer Wiederverwendungsanalyse (Abschnitt 3.2) dargestellt.

Prg.	Befehle	L1-H.	L1-M.	L2-H.	L2-M.	P-Zyklen
hydro2d	5.543.080	1.186.888	218.461	16.091	202.371	23.917.370
tomcatv	6.398.033	1.647.655	172.041	26.139	145.901	19.790.523
turb3d	14.453.171	2.568.151	84.751	34.275	50.475	19.338.681
wave5	2.700.459	564.179	38.124	30.988	7.106	3.650.179

Abbildung 7: Simulationsergebnisse ohne Transformation [in 1000en]

Dazu wurden verschiedene Benchmark-Programme übersetzt und auf einem Simulator ausgeführt, der das Verhalten des Programmes bezüglich des Daten-Cache simuliert.

		Fusionierung		Umkehr	
Prg.	Schleifen	fusioniert	Kandidaten	umgekehrt	Kandidaten
hydro2d	163	3	14	5	136
tomcatv	16	0	2	0	8
turb3d	64	0	0	0	33
wave5	377	14	27	27	212

Abbildung 8: Transformationen

Prg.	Befehle	L1-H.	L1-M.	L2-H.	L2-M.	P-Zyklen
hydro2d	0	0,88	-0,09	15,08	-1,29	-0,88
tomcatv	-0,01	0	0	0,01	0	0
turb3d	1,48	-0,48	13,11	32,39	0,02	1,69
wave5	-0,50	-0,81	2,01	2,55	-0,35	-0,21

Abbildung 9: Prozentualer Unterschied nach den Transformationen

		Fusionierung		Umkehr	
Prg.	Schleifen	fusioniert	Kandidaten	umgekehrt	Kandidaten
hydro2d	163	5	14	20	136
tomcatv	16	1	2	0	8
turb3d	64	0	0	0	33
wave5	377	14	27	26	212

Abbildung 10: Transformationen (mit Profil)

Prg.	Befehle	L1-H.	L1-M.	L2-H.	L2-M.	P-Zyklen
hydro2d	0,69	0,74	-4,06	89,38	-11,49	-7,99
tomcatv	-0,01	0,99	-9,49	-11,64	-9,11	-6,20
turb3d	0	0	-0,05	-0,14	0	0
wave5	0,18	-0,81	2,05	2,61	-0,35	0,29

Abbildung 11: Prozentualer Unterschied nach den Transformationen (mit Profil)

Zur Angabe eines Maßes für die Gesamtlaufzeit des Codes werden P-Zyklen PC berechnet, die sich wie folgt zusammensetzen:

$$PC = IC + M_1 \cdot P_1 + M_2 \cdot P_2 \,.$$

IC ist die Anzahl der Maschinenbefehle, die für eine Ausführung des Programmes ausgeführt werden. M_1 ist die Anzahl der Level-1-Cache-Misses und P_1 die Anzahl der Befehlszyklen, die ein Level-1-Miss verbraucht.

Für die Ergebnisse wurden 10 Zyklen für Level-1-Misses und 80 Zyklen für Level-2-Misses angesetzt. Als Caches wird ein 64 KB großer, 4fach assoziativer L1-Cache und ein 1024 KB großer, auch 4fach assoziativer L2-Cache, beide mit 32 Byte langen Zeilen angenommen.

Abbildung 7 zeigt die Daten der untransformierten Programme, Abbildung 8 zeigt die Transformationen, Abbildung 9 zeigt die prozentualen Unterschiede zwischen den untransformierten und den transformierten Programmen.[10]

Da die Ergebnisse der Codetransformationen relativ schlecht waren, versuchten die Autoren von [Coo], mit Hilfe von Daten, die aus einem Ausführungsprofil gewonnen wurden, besser zu optimieren. Ein Mangel in der dargestellten Wiederverwendungsanalyse ist nämlich, daß sie zu pessimistisch analysiert, wenn Schleifengrenzen nicht durch statische Analyse herausgefunden werden. Die Daten aus dem Profil umfassen für alle Schleifen das Minimum, Maximum und den Durchschnitt für beide Schleifengrenzen, für die Schrittweite und die Anzahl der Durchläufe. Dadurch wurde die Einschätzung, ob bestimmte Transformationen lohnend sind, verändert.

Die Ergebnisse der Transformationen sind in Abbildungen 10 und 11 zu sehen. Bei zwei Programmen ergab sich durch die Verwendung eines Profils eine drastische Verbesserung, bei einem eine kleine Verschlechterung. Bei einem Programm wurde wenigstens eine Verschlechterung, die ohne Profil auftrat, vermieden.

6 Zusammenfassung

Es wurde eine Einführung in die Kontroll- und Datenflußanalyse gegeben, um den Hauptteil dieser Ausarbeitung vorzubereiten. In diesem Hauptteil wurden drei Techniken zur Datenflußanalyse für Reihungen vorgestellt. Anschließend wurden Anwendungen motiviert und Erfahrungen, die mit einer der angeführten Anwendungen gewonnen wurden, dargestellt.

Diese Erfahrungen, die in Abschnitt 5 vorgestellt wurden, zeigen, daß mit einer genaueren Datenflußanalyse, die Reihungen nicht mehr als unteilbare Objekte betrachtet, mäßige Verbesserungen im generierten Code möglich sind. Allerdings werden gute Ergebnisse erst dann erzielt, wenn die Informationen aus der Datenflußanalyse noch mit anderen Informationen verknüpft werden, zum Beispiel Ausführungsprofilen oder Annotationen des Programmieres im Quelltext.

Sicher ist das Gebiet der Cache-Optimierung immer noch ein Gebiet, in dem noch viel Forschungsbedarf besteht. So besteht die Hoffnung, daß neuere Techniken vielleicht manche der Informationen, die derzeit noch von außen (Profile, Annotationen des Programmierers) zugeführt werden, selbst berechnet werden können.

Ein interessantes Gebiet wäre, einen Rahmen für die Datenflußanalyse zu finden, der u.a. die drei vorgestellten Techniken umfaßt, und eventuell durch die Kombination dieser Informationen weiteren Nutzen möglich macht, den die einzelnen Techniken für sich nicht bieten können.

Literatur

[Coo] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh: Cross-Loop Reuse Analysis and Its Application to Cache Optimizations. Department of Computer Science, Rice University, Houston, Texas USA.

[10] In den Tabellen sind die Ergebnisse für einige Programme weggelassen. Siehe [Coo] für die vollständigen Daten.

[Fea1] Paul Feautrier: Dataflow Analysis of Array and Scalar References. International Journal of Parallel Programming, Vol. 20, Nr. 1. September 1991.

[Fea2] Paul Feautrier: Parametric Integer Programming. RAIRO Recherche Opérationelle, 22:243–268, September 1988.

[Han] Reinhard von Hanxleden: Compiler Support for Machine-Independent Parallelization of Irregular Problems. Technical Report CRPC-TR94494-S, Center for Research on Parallel Computation, Rice University, December 1994. Ph.D. Thesis.

Seminar - Ausarbeitung VI

Sven-Olaf Braun

Prefetching

Zusammenfassung

Prefetching bezeichnet eine Methode, Wartezeiten welche für die heutigen sehr schnellen Prozessoren durch Speicherlatenz entstehen, zu verringern. Dieses Dokument bezieht sich hierbei ausschließlich auf das sogenannte 'Software-Prefetching', welches im Gegensatz zum 'Hardware- Prefetching' steht. Diese Technik basiert auf einer Analyse des Programmcodes und dem anschließenden Einfügen von Prefetch-Operationen an kritischen Stellen. Durch diese Prefetch-Operationen soll die Wartezeit des Prozessors auf den Speicher eliminiert werden. Dieses Dokument zeigt einen systematischen Weg auf, um die richtigen Stellen zu finden, an denen diese Prefetch-Operationen eingefügt werden müssen, ohne 'zuviel des Guten' zu tun bzw. sogar kontraproduktiv zu handeln. Diese Methode eignet sich vor allem für wissenschaftliche Programme welche überwiegend auf dichtbesetzen Matrizen arbeiten. Programme mit unstrukturierten Speicherzugriffen gewinnen nicht so viel mit dieser Methode.

1 Einführung

Heutige Prozessoren werden immer schneller, zum einen durch höhere Taktraten, zum anderen durch verbesserte Architekturen. Eines der heutigen Hauptprobleme ist der Zugriff auf den Arbeitsspeicher welcher nicht in dem selben Maße schneller wurde, wie die Prozessoren. Gründe hierfür sind die höheren Prioritäten der Hersteller für die Kapazitäten der RAM-Bausteine, was Nachteile für deren Geschwindigkeit mit sich bringt. Es ist nicht abzusehen, daß die RAM-Bausteine in der nächsten Zeit in diesem Punkt nachziehen werden, sondern es ist eher das Gegenteil zu erwarten. Um dieses Problem zu mildern gibt es verschiedene Techniken. Eine davon ist das sogenannte 'Prefetching', was soviel bedeutet wie 'vorher holen' oder 'vorladen'.

Um das Potential der Verbesserungsmöglichkeiten zu verdeutlichen führe man sich vor Augen, daß speziell wissenschaftliche Programme oft mehr als die Hälfte ihrer Rechenzeit auf das Speichersystem warten.

2 Software-Prefetching im Verhältnis zu anderen Speicherlatenz-Techniken

Das in diesem Dokument behandelte Software-Prefetching ist nicht die einzige Technik, um das oben genannte Latenzproblem zwischen Prozessor und Speichersubsystem zu verringern. Die wohl bekannteste Technik ist das 'Caching'. Dieses 'Caching' ist Grundvorraussetzung für die Anwendung des Software-Prefetching, wird dadurch aber noch weiter optimiert. Andere Methoden sind generell das Schreiben von 'besserem' Code, d.h. Code der die Bandbreite des Speichersystems weniger beansprucht. Hierfür gibt es die verschiedensten Techniken, z.B. das 'blocking', was aber nicht Gegenstand unserer Betrachtung sein wird.

Das Multithreading bietet ebenfalls die Möglichkeit, das Latenzproblem zu mildern, indem bei einer Blockierung des Prozessors durch eine Leseoperation einfach ein anderer Thread ausgeführt wird, welcher nicht blockiert ist. Nachteil gegenüber dem Software-Prefetching ist, daß nicht gewährleistet ist, daß überhaupt ein anderer Prozeß vorhanden ist, der arbeiten kann, und daß die Kontextumschaltung oft den erzielten Gewinn wieder zunichte macht. Vorteile bietet es dadurch, daß man das eigentliche Programm

hierfür nicht ändern muß. Es funktioniert auch bei Zugriffsmustern auf den Speicher, welche sich mit der Prefetching-Technik nicht in den Griff bekommen lassen, z.B. wenn Speicherstellen nicht im Vorraus bekannt sind.

Speziell zur Milderung von Latenzproblemen beim Schreiben kommt die Pufferung zum Einsatz. Bei der Pufferung wird einfach der Schreibvorgang parallel zum Prozessor im Hintergrund durchgeführt und blockiert so den Prozessor ebenfalls nicht mehr.

3 Wie das Software-Prefetching funktioniert

Software-Prefetching basiert auf dem Prinzip 'Wenn Du dazu länger brauchst, dann fang früher an damit.'. D.h. Speicherinhalte, welche bald benötigt werden, werden schon früher beim Speichersubsystem angefordert, so daß sie rechtzeitig zur Verfügung stehen, ohne den Prozessor zu blockieren.

Hierbei nutzt das System die hierarchische Architektur eines typischen Speichersystems aus. Denn irgendwo müssen diese vom Speichersystem angeforderten Werte, welche ja eigentlich noch nicht benötigt werden, abgelegt werden. Dieser Ort ist der Cache-Speicher. Wegen dessen hoher Geschwindigkeit stehen die dort abgelegten Werte aber sehr schnell zur Verfügung, was einen enormen Geschwindigkeitsgewinn gegenüber dem Hauptspeicher ergibt.

Grundvorraussetzungen zur Implementierung dieses Software-Prefetching sind folgende :

- Der Prozessor muß einen speziellen Prefetch-Befehl bereitstellen, über den er dem Speichersystem mitteilt, daß er bald einen bestimmten Speicherinhalt benötigt. Solch ein Befehl kann oft auch durch einen üblichen Ladebefehl simuliert werden. Zweck des Befehls ist es, Daten vom Hauptspeicher in den Cache zu bringen. Üblicherweise sieht solch ein Befehl aus, wie ein normaler Ladebefehl, außer daß das Zielregister nicht angegeben wird. Optimalerweise hat ein solcher Befehl noch die Eigenschaft, daß er sich bei Fehlersituationen, wie z.B. bei der Dereferenzierung einer nichtinitialisierten Speicherstelle, neutral verhält. Das ermöglicht auch mehr spekulative Prefetch-Operationen, ohne daß das Programm dadurch Fehler erzeugt.
- Der Cache muß nicht-blockierend sein. Das bedeutet, daß das Cache-System mehrere Cache-misses verarbeiten kann, ohne den Prozessor anzuhalten, vorrausgesetzt natürlich, die gewünschte Speicherstelle wird nicht in den darauf folgenden Programmbefehlen verwendet.
- Die Speicherstellen, auf die in Kürze zugegriffen wird, lassen sich im Vorraus bestimmen. Dies trifft auf eine ganze Reihe von Programmen zu, z.B. auf viele rechenintensive Schleifen. Sind die Speicherstellen unbekannt, lassen sie sich natürlich auch nicht im Voraus holen.

4 Die Vorgehensweise zur Implementierung

4.1 Ein kleines Beispiel

Um das Software-Prefetching zu verstehen zuerst ein kleines Programm-Beispiel. Hierbei und im weiteren gehen wir davon aus, daß unser Computersystem einen Prozessor sowie ein Speichersubsystem mit einem 8 kB Primärcache und einer Cache-Blockgröße von 4 Worten hat, was realistisch in Bezug auf aktuelle Rechner ist.

4.2 Welche Probleme können hierbei auftreten ?

Die eingefügten Prefetch-Befehle könnten einfach überflüssig sein und nur unnötig Overhead produzieren. Gründe hierfür können sein :

- Der Speicherinhalt, welcher mit dem Prefetch in den Cache geholt wird, wird schon vor seiner Verwendung wieder aus dem Cache geworfen. Ist der Cache klein und der Abstand zwischen Prefetch und Verwendung des Datums groß, wird diese Situation begünstigt.

Beispielprogramm zuerst ohne Prefetching :

```
for (i=0;i < 3; i++)
   for (j=0; j<100; j++)
      A[i][j] = B[j][0] + B[j+1][0];
```

Jetzt mit Prefetching :

```
prefetch(A[0][0]);

for(j=0;j<6;j+=2) {
   prefetch(B[j+1][0]);
   prefetch(B[j+2][0]);
   prefetch(A[0][j+1]);
}

for(j=0;j<94;j+=2) {
   prefetch(B[j+7][0]);
   prefetch(B[j+8][0]);
   prefetch(A[0][j+7]);
   A[0][j] =B[j][0]+B[j+1][0];
   A[0][j+1] =B[j+1][0]+B[j+2][0];
}

for(j=94;j<100;j+=2) {
   A[0][j] =B[j][0]+B[j+1][0];
   A[0][j+1] =B[j+1][0]+B[j+2][0];
}

for(i=1;i<3;i++) {
   prefetch(A[i][0]);
   for(j=0;j<6;j+=2)
      prefetch(A[i][j+1]);
      for(j=0;j<94;j+=2) {
         prefetch(A[i][j+7]);
         A[i][j] =B[j][0]+B[j+1][0];
         A[i][j+1] =B[j+1][0]+B[j+2][0];
      }

      for(j=94;j<100;j+=2) {
         A[i][j] =B[j][0]+B[j+1][0];
         A[i][j+1] =B[j+1][0]+B[j+2][0];
  }
}
```

Abbildung 1: Code-Beispiel

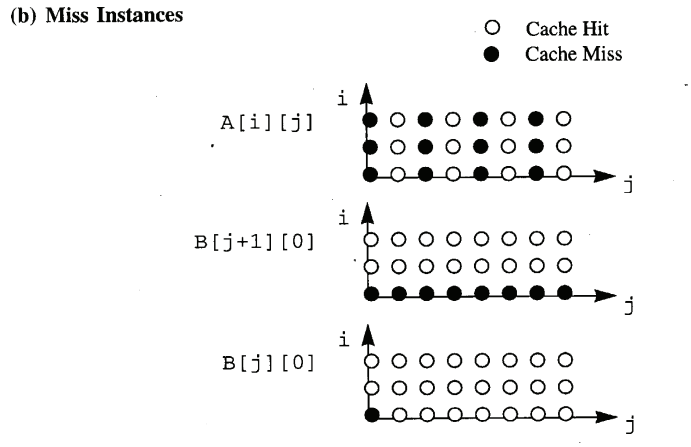


Abbildung 2: Cache Hits/Misses

- Die Prefetch-Operation wird zu kurz vor der Verwendung des Datums eingefügt. In diesem Fall wird die Speicherlatenz nur zum Teil verborgen, da dem Speichersubsystem einfach nicht genug bleibt, das Datum rechtzeitig in den Cache zu holen.
- Der Speicherinhalt ist bereits im Cache. Dann ist auch die kleinste Prefetch-Operation überflüssig. Tatsächlich sind heutige Cache-Systeme bereits derart effektiv, daß bei der raw-power-Methode (einfach überall alle benötigten Speicherwerte über einen Prefetch in den Speicher laden) in den meisten Fällen über die Hälfte aller Prefetches überflüssig wären.

Als Ergebnis wird klar, daß folgende Schritte notwendig sind, um eine optimale Prefetching-Methode umzusetzen :

1. Alle Stellen, an denen der Hauptspeicher im Programm referenziert wird müssen gefunden werden.
2. Diese Stellen müssen dahingehend analysiert werden, ob für sie ein Prefetching überflüssig oder notwendig ist.
3. Für alle notwendigen Prefetches muß der optimale Zeitpunkt ermittelt werden.
4. Die Prefetches müssen in das Programm eingebaut werden.

5 Die Lokalitätsanalyse

Der Schritt der Lokalitätsanalyse ermittelt, welche Speicherreferenzen überhaupt mit einem Prefetch optimiert werden müssen. Diese Analyse unterteilt sich in zwei Schritte.

5.1 Ermittlung von Wiederverwendung

Zuerst müssen die Speicherreferenzen ermittelt werden, die während des Programmablaufs mehrfach verwendet werden. Hierbei wird nicht ein Zugriff auf eine bestimmte Speicherstelle, sondern der Zugrif auf einen Cache-Block betrachtet. Dabei werden drei unterschiedliche Arten von Wiederverwendung unterschieden :

1. *Zeitliche Wiederverwendung :* Als solche bezeichnet man eine Situation, in der genau die selbe Stelle später im Programmlauf noch einmal verwendet wird, wobei das Datum aber noch im Cache steht und nicht erneut aus dem Hauptspeicher geladen werden muß.

2. *Räumliche Wiederverwendung :* Hierbei handelt es sich um den Fall, daß eine Referenz auf verschiedende Daten zugreift, die jedoch im selben Cache-Block stehen.

3. *Gruppen-Wiederverwendung :* Diese Art von Wiederverwendung beschreibt den Fall, daß verschiedene Referenzen auf den selben Cache-Block zugreifen.

Beispiel für :

- die Zeitliche Wiederverwendung : die Referenz B[j][0]
- die Räumliche Wiederverwendung : die Referenz A[i][j]
- die Gruppen-Wiederverwendung : die Referenzen B[j][0] und B[j+1][0]

Analog zu den Typen von Wiederverwendung unterscheiden wir ebenfalls drei Typen von Lokalität : die zeitliche Lokalität, die räumliche Lokalität und die Gruppen-Lokalität. Der Unterschied zwischen der Wiederverwendung und der Lokalität besteht darin, daß für die Lokalität mitentscheidend ist, wie der Cache aufgebaut ist. Lokalität ist Wiederverwendung eines Datums, wenn es tatsächlich noch im Cache war. Die Wiederverwendung hingegen ist eine inhärente Eigenschaft des Programmcodes. Wäre unser Cache unendlich groß, wären beide Begriffe deckungsgleich.

Zur Darstellung von Wiederverwendung bzw. Lokalität führen wir einen n-dimensionalen Vektorraum, den sogenannten Wiederverwendungs-Vektorraum, kurz WV ein. Dessen Dimension n entspricht der Schachtelungstiefe unserer Schleifen. Die erste Dimension des Vektorraums entspricht hierbei der äußersten Schleife. Zur weiteren Vereinfachung der Darstellung führen wir noch die Feldindexfunktion f ein, welche aus dem Raum der n Schleifenindizes in den Raum der d Matrizenindizes abbildet. n ist dabei wieder die Schachtelungstiefe der Schleife, d ist die Dimension des Feldes, auf welchem die Schleife arbeitet. Diese Funktion hat die Form $f(i) = Hi + c$.

H stellt hierbei eine lineare $d \times n$-Matrix dar, i ist ein Element des Schleifenindexraums und c ein konstanter Vektor bestehend aus d Elementen.

5.1.1 Zeitliche Wiederverwendung

Zur Ermittlung von zeitlicher Wiederverwendung bestimmen wir die Menge von Vektoren welche bei verschiedenen Schleifenindizes denselben Feldindex ergeben. Wir lösen also die Gleichung $Hi+c = Hj+c$ mit $i \neq j$. Das Problem läßt sich umformulieren in $H(x) = 0$. Als Ergebnis erhalten wir einen n-dimensionalen Vektorraum, welcher uns beschreibt, für welche Iterationen eine zeitliche Wiederverwendung auftritt.

5.1.2 Räumliche Wiederverwendung

Auf ähnliche Weise können wir den Vektorraum ermitteln, für den räumliche Wiederverwendung eintritt. Im Unterschied zur zeitlichen Wiederverwendung müssen wir nun aber beachten, daß unterschiedliche Feldindizes auf denselben Cache-Block verweisen können. Gehen wir o.B.d.A davon aus, daß unsere Daten zeilenweise im Hauptspeicher abgelegt sind (row major order), dann ergibt dies, daß Daten, welche sich im selben Cache-Block befinden können, sich nur im Zeilenindex unterscheiden. Dies berücksichtigen wir, indem wir die Transformationsmatrix H in H_S umformen, indem wir die Elemente der letzten Zeile auf Null setzen. Zur Ermittlung des gesuchten Vektorraums setzen wir nun $H_S(x) = 0$. Beachten müssen wir weiterhin, daß der Vektorraum $H_S(x) = 0$ den Vektorraum $H(x) = 0$ enthält, da zeitliche Wiederverwendung eigentlich eine degenerierte räumliche Wiederverwendung darstellt. Ist $H_S(x) = H(x)$ dann handelt es sich lediglich um zeitliche Wiederverwendung, nicht um räumliche.

5.1.3 Gruppen-Wiederverwendung

Die Gruppen-Wiederverwendung ist schwieriger zu analysieren. Dies gelingt nur dann, wenn die Referenzen *einheitlich erzeugt* wurden, d.h. wenn die Indizes der beiden Referenzen sich höchstens um einen konstanten Term unterscheiden. Ist dies der Fall dann suchen wir für die beiden Referenzen $A[Hi + c_1]$

und $A[Hi + c_2]$ eine bestimmte Lösung r der Gleichung $H_S(r) = c_{s1} - c_{s2}$. Die beiden Referenzen greifen nun auf die selben Daten zu, wenn eine ganzzahlige Lösung dieser Gleichung existiert. c_{s1} und c_{s2} gehen aus c_1 und c_2 hervor, indem wieder die letzte Zeile auf Null gesetzt wird. Dies geschieht, wie schon bei der räumlichen Wiederverwendung dazu, nicht nur die exakt selben Datenelemente, sondern auch Datenelemente desselben Cache-Blocks zu berücksichtigen. Hierbei wird der Fall, daß eine Cache-Block-Grenze überschritten wird nicht betrachtet.

5.2 Der Lokalitätsraum

5.2.1 Ein Beispiel

Aus den Vektorräumen der Wiederverwendung wird nun der Lokalitätsraum ermittelt. Wie bereits erwähnt unterscheidet sich vom dieser dadurch, daß er berücksichtigt, welche Datenelemente sich noch im Cache befinden, was außer vom eigentlichen Programm und dessen Datenvolumen zusätzlich von dessen Organisation und Größe abhängt. Anschaulich wird dies in folgendem Beispiel klar :

```
for(i=0;i<3;i++)
   for (j=0;j<3;j++)
      A[i][[j] = B[j][0] + B[j+1][0]
```

Abbildung 3: Daten alle im Cache

In diesem Beispiel werden in der Schleife 9 Elemente von Feld A und 4 Elemente von Feld B referenziert. So wenig Daten passen sicher in den Cache, ohne daß die ersten Daten von folgenden wieder verdrängt werden. Die Schleife ist also vollständig lokalisiert. Durch eine leichte Modifikation der Schleifenrandbedingungen ändert sich jedoch die Sachlage :

```
for(i=0;i<3;i++)
   for (j=0;j<10000;j++)
      A[i][[j] = B[j][0] + B[j+1][0]
```

Abbildung 4: Daten nicht alle im Cache

Da die innere Schleife jetzt bis 10000 läuft werden also insgesamt bereits mehr als 30000 Feldwerte von Feld A referenziert. Diese passen in unserer Beispielkonfiguration mit dem Cache von 8 kB nicht mehr gleichzeitig in den Cache. Die Referenz $B[j + 1][0]$, welche vorher noch in der äußeren Schleife zeitlich lokalisiert war ist es jetzt nicht mehr. Wichtig hierbei ist daß die zeitliche Wiederverwendung der Referenz $B[j + 1][0]$ weiterhin gilt, da diese eine Eigenschaft des Codes ist, unabhängig von der Beschaffenheit des Caches.

5.2.2 Die Ermittlung des Lokalisierten Raumes

Eine exakte Vorrausbestimmung des Cacheinhaltes ist für den Compiler undurchfürbar, da Schleifengrenzen oft symbolisch benannt sind und deren Wert erst zur Laufzeit ermittelt wird. Daher wird zur Ermittlung des sogenannten Lokalisierten Raumes, kurz LR nur dessen Dimension berücksichtigt. D.h. nur die Anzahl der Schleifen, deren Daten noch komplett in den Cache-Speicher passen, sind von Belang.

Diese Dimension der Lokalität von verschachtelten Schleifen folgt einigen einfachen Regeln :

- zuerst wird die Datenmenge, welche in der innersten Schleife durch den Cache geschleußt wird bestimmt. Dies geschieht einfach durch aufaddieren der Datengrößen der einzelnen Referenzen.
- dieser Vorgang wird rekursiv jeweils für die nächsthöhere Schleifenebene durchgeführt. Ist die Datenmenge einer Schleife zu groß für den Cache, sind es damit automatisch auch alle sie umfassenden.

Folgende Probleme machen diese simple Methode jedoch wieder kompliziert :

- Symbolische Schleifengrenzen lassen sich nicht im Vorraus bestimmen, daher müssen verschiedene Methoden verwendet werden, um möglichst viele symbolische Grenzen durch konstante Grenzen zu ersetzen. Solch eine Methode ist z.B. das 'interprocedural constant propagation'. Wo die symbolische Grenze schließlich nicht aufgelöst werden kann, wird sie entweder als 'klein' (paßt in den Cache) oder 'groß' (paßt nicht mehr in den Cache) angenommen.
- Cache-Abbildungs-Konflikte können dazu führen, daß der Cache zwar von seiner Größe her noch zusätzlich Daten aufnehmen könnte, er aber wegen ungünstiger Verteilung der zu cachenden Datenelemente im Speicher bereits vorzeitig wieder andere Werte hinauswerfen muß. Diesem Umstand wird Rechnung getragen, indem die tatsächliche Cachegröße mit einem Faktor α $(0 < \alpha < 1)$, multipliziert wird.

5.2.3 Die Ermittlung von Lokalität

Haben wir nun sowohl den Vektorraum der Wiederverwendung als auch den Vektorraum der Lokalisierung ermittelt ist sofort klar, welche Referenzen lokalisiert sind, nämlich genau diese, welche in beiden enthalten sind. Mathematisch ausgedrückt : $WV \cap LR \Rightarrow$ *Lokalitätsraum*

5.3 Das Prefetch-Prädikat

Obwohl der nun ermittelte Vektorraum recht bequem für den Computer ist, wird es dem Betrachter klarer, was zu tun ist, wenn das sogenannte Prefetch-Prädikat eingeführt wird. Dieses gibt an, ob für einen gewissen Schleifendurchlauf eine Referenz mit einem Prefetch versehen werden muß oder nicht. Die Ermittlung dieses Prädikats erfolgt unterschiedlich, je nach Art der

Lokalität der Referenz. Für unsere Ausführung gehen wir davon aus, daß die Schleifendurchläufe bei 0 beginnen und auf einer Cache-Blockgrenze ausgerichtet sind. Dies läßt sich auf den allgemeinen Fall durch einfache Transformationen übertragen, vereinfacht aber hier die Darstellung.

- bei *Zeitlicher Lokalität* muß nur die Referenz des ersten Schleifendurchlaufs vorgeladen werden. Als Prädikat ergibt sich somit die Bedingung $i = 0$.
- bei *Räumlicher Lokalität* muß nur jede n. Referenz vorgeladen werden. n stellt hierbei die Anzahl der Datenelemente dar, welche in einen Cache-Block passen. Als Prädikat ergibt sich somit die Modulo-Bedingung $i \ mod \ n = 0$.
- bei *Gruppen-Lokalität* muß jeweils nur das erste Element der Gruppe vorgeladen werden. Daher ist die Bedingung für alle anderen Elemente automatisch 'false'.
- Referenzen ohne Lokalität haben als Bedingung immer 'true', da diese grundsätzlich vorgeladen werden müssen.
- Referenzen, welche mehrere verschiedene Arten von Lokalität geniessen, haben als Bedingung die Konjunktion ihrer einzelnen Prefetch-Prädikate.

6 Einfügen der Prefetches

Nachdem in der Analysephase ermittelt wurde, welche Referenzen durch einen Prefetch optimiert werden müssen, gehen wir nun daran, diese Prefetches systematisch im Programmcode einzubauen. Hierbei gilt es, wie bereits gesagt, darauf zu achten, sie zeitlich korrekt zu plazieren. Geschieht dies nicht, ist es möglich, daß solche Prefetches ineffektiv sind und nur Overhead produzieren.

6.1 Auswertung der Prefetch-Prädikate

Die Prefetch-Prädikate geben zwar direkt an, ob eine Referenz vorgeladen muß oder nicht, allerdings kostet es sehr viel Rechenzeit, diese Prädikate innerhalb einer Schleife jedesmal via 'if' auf deren Wert zu üeberprüfen. Daher beschreitet man einen anderen Weg, nämlich den der Schleifenauflösung.

Hierbei benötigt man je nach Art des Prefetch-Prädikats unterschiedliche Techniken. Normalerweise gelten hierfür folgende Richtlinien :

- Ist das Prädikat 'true' oder 'false' ist die Lage klar. Entweder es muß jedesmal ein Prefetch eingefügt werden, oder nie.
- Prädikate der Art 'i=0' werden durch das 'peeling' des ersten Schleifendurchlaufs modifiziert. Ein Beispiel für das 'peeling' ist z.B. folgendes :

```
Die Schleife vor dem Peeling-Vorgang~:

for (i=0;i<n;i++) {
      if (i==0)
      prefetch(i);
   restcode(i);
}

Die Schleife nach dem Peeling-Vorgang :

prefetch(0);
restcode(0);
for (i=1;i<n;i++) {
  restcode(i);
}
```

Abbildung 5: Peeling-Beispiel

- Prädikate der Art $(i \ mod \ l) \ = \ 0$ werden aufgelöst, indem die Schleife aufgerollt wird oder per 'strip-mining' in zwei verschachtelte Schleifen umgewandelt wird. Hierbei ist das aufrollen meist bei kleinen Werten von l zu bevorzugen, das 'strip-mining' jedoch bei großen l-Werten vorteilhafter. Auch hierzu jeweils ein Beispiel :

6.2 Plazieren der Prefetches

Wichtig für die Effektivität der Prefetch-Operationen ist deren korrekte Plazierung. Sind sie zu früh im Code plaziert, ist der Datenwert, welcher mit dem Prefetch geholt wurde eventuell bereits wieder aus dem Cache geworfen. Werden sie aber zu spät eingesetzt, hat das Speichersystem nicht genügend Zeit, den Datenwert in den Cache zu legen, die Speicherlatenz wird also nur ungenügend überbrückt.

Als Maß für den optimalen Abstand soll im folgenden die Anzahl der Schleifendurchläufe genommen werden. Als guten Abstand der Prefetch-Operation zur entsprechenden Speicherreferenz ergibt sich somit

$\lceil \frac{l}{s} \rceil$

wobei l die Speicherlatenzzeit ist und s die Länge des kürzesten Pfads durch den Schleifenkörper, welche vom Compiler näherungsweise bestimmt wird.

Angenommen, $n = \lceil \frac{l}{s} \rceil$ ergibt nun 5 Schleifendurchläufe, dann zerlegen wir die ursprüngliche Schleife für das Software-Pipelining nun in drei Teile :

- Den Prolog : hier werden die ersten n Speicherreferenzen vorgeladen.

```
Ausgangscode :

for (i=0;i<n;i++) {
   if ((i mod 8) == 0)
      prefetch(i);
      restcode(i);
}

Code mit ausgerollter Schleife :

for (i=0;i<n;i+=8) {
   prefetch(i)
   restcode(i);
   restcode(i+1);
   restcode(i+2);
   restcode(i+3);
   restcode(i+4);
   restcode(i+5);
   restcode(i+6);
   restcode(i+7);
}

Code nach dem 'strip-mining' :

for (j=0;j<n;j+=8) {
  prefetch(j);
  for(i=j;i<j+8;i++)
    restcode(i);
}
```

Abbildung 6: Schleifen ausrollen und 'strip-mining'

- Die Hauptschleife : hier läuft der mittlere Teil der Schleife, wie normal, allerdings ebenfalls mit Prefetches.
- Den Epilog : hier werden die letzten Schleifendurchläufe abgearbeitet, welche keine Prefetches mehr enthalten brauchen.

7 Ergebnisse

Schlußendlich stellt man sich natürlich nun die Frage : Was für einen Erfolg bringt die ganze Sache ?

Sicherlich sind die Ergebnisse stark vom jeweils zu optimierenden Code abhängig. Vorrangig für diese Art des Prefetching geeignet sind wissenschaftliche Programme, welche häufig mittels Schleifen auf Feldern arbeiten. Programme, welche in zufälliger Weise auf den Speicher zugreifen sind auf diese Weise schwieriger zu optimieren, da wichtig ist, daß sich die Speicheradressen im Vorraus ermitteln lassen. Aber gerade bei dieser Art von Programmen ist auch das Potential zur Optimierung gegeben, da diese oft lange Rechenzeiten aufweisen.

Bei einer Testimplementierung durch Todd C. Mowry [1] konnten gewisse Grundaussagen bezüglich der Optimierungs-Möglichkeiten, welche Prefetching bietet gefunden werden. Hierbei wurde eine Test-Suite von 13 Beispielprogrammen des oben genannten Typs einmal in der Grundform und das andere mal mittels Prefetching optimiert, mit einem Simulator analysiert.

Hierbei wurden 6 der 13 Programme um mehr als 45% schneller. Auch alle anderen Programme wurden insgesamt schneller, jedoch nur um 2 bis 45%.

Hauptsächliche Ursache war, wie beabsichtigt, die Reduzierung der Wartezeiten des Prozessors auf den Speicher um 50% bis 90%. Hierbei wurde der neu produzierte Overhead meist unter 15% gehalten (bei 5 von 13 Programmen) und ging nie über 50%, so daß sich in der Summe immer ein Geschwindigkeitszuwachs ergab.

Interessant war auch der Vergleich zwischen dem oben geschilderten Prefetching-System und einem, welches ohne die vorherige Analysephase einfach alle betroffenen Speicherreferenzen auf Felder innerhalb von Scheifen vorlädt. Diese Methode produziert bedeutend mehr Overhead, so daß sich manche Programme sogar verlangsamten.

In der detaillierten Anaylse der Ergebnisse ergab sich, daß bei 5 von 13 Programmen über 90% der notwendigen Prefetches gefunden wurden, bei 5 anderen jedoch nur 50%, was z.T. darauf zurückzuführen ist, daß nicht alle Speichereferenzen dem Muster des linearen Feldzugriffs folgen.

Die Technik der Schleifenauflösung ergab gute Resultate. Im direkten Vergleich mit der naiven Methode einer if-Abfrage für das Prefetch-Prädikat ergab sich ein Geschwindigkeitsvorteil von mehr als 25%.

Die Auswirkungen des Software-Pipelining-Schemas sind ebenfalls positiv. Von den 13 Programmen waren nur zwei Programme dabei, bei denen mehr als 20% der Prefetches ineffektiv waren. Der Rest lag meist weit unter 10%. Bei den beiden

Ausnahmen waren in der Regel Cache-Konflikte dafür verantwortlich, daß die referenzierten Daten bereits vor der Verwendung wieder aus dem Cache geworfen wurden.

8 Andere Anwendungsfelder

8.1 Rekursive Datenstrukturen

Für Schleifenorientierte Programme haben wir nun gesehen, daß sich doch eine recht ordentliche Steigerung der Geschwindigkeit ergibt, wenn Speicher-Wartezyklen über das Prefetching eliminiert werden. Es stellt sich die Frage, ob sich dieses System auch auf andere Gebiete erfolgreich anwenden läßt, da schließlich auch noch viele andere Programme existieren, welche grundsätzlich zu langsam laufen. Eine weitere wichtige Klasse von Programmen, welche in diese Kategorie fällt ist diese, welche mit rekursiven

Datenstrukturen arbeitet. Diese Klasse wird auch 'zeigerorientiert' genannt, da sie in der Regel die einzelnen Datenelemente über Zeiger verknüpfen. Hierzu gehören z.B. Programme, welche Binäre Bäume oder verkettete Listen verwenden.

Das Hauptproblem bei dieser Programmsorte ist das der rechtzeitigen Vorherbestimmung der Speicheradressen der Referenzen, welche über einen Prefetch optimiert werden sollen. Gründe hierfür sind, daß im Vorraus meist nicht klar ist, in welcher Weise die Datenstruktur später traversiert wird. Dies erschwert sowohl die Analysephase, als auch die Phase des Prefetch-Einbaus. Zur Veranschaulichung dieses Problems (Pointer-Chasing Problem genannt) stelle, man sich vor, man soll den 5. Knoten welcher nach dem aktuellen Knoten verarbeitet wird laden. Hierzu müssen über die Zeigerstruktur auch noch alle 4 dazwischenliegenden Knoten geladen werden und deren Verweisadressen decodiert werden.

8.2 Lösungsansätze für rekursive Datenstrukturen

Für die Lösung des Einbau-Problems sind drei Methoden bekannt :

- das *Greedy-Verfahren* : hierbei werden bei einem Knoten, welcher n weitere Verweise auf andere Knoten enthäelt alle diese n Knoten vorgeladen, in der Erwartung, daß diese bald ebenfalls verarbeitet werden. Höchstens der erste Knoten verursacht dann eine Wartezeit für den Prozessor, vorausgesetzt die Verarbeitungszeit des Knotens liegt über der Prefetch-Ladezeit eines Knoten). Die restlichen Knoten sind bereits im Cache, wenn sie 'zeitnah' weiterverarbeitet werden. Der Vorteil an diesem System ist, daß weder zusätzlicher Speicheraufwand, noch zusätzlicher Rechenaufwand betrieben werden muß.

- das *Historienzeiger-Prefetching* : hierbei wird für jeden Knoten ein Zeiger auf den Knoten eingeführt, welcher d Schritte zuvor besucht wurde, wobei d die Anzahl Knoten angibt, welche mindestens vorher geholt werden müssen, um die Speicherlatenz zu verbergen. Diese Zeiger werden bei der ersten Traversierung der Datenstruktur mittels eines ständig aktualisierten FIFO-buffers, in dem jeweils die zuletzt besuchten Knoten vermerkt werden aktualisiert. Am Ende der ersten Traversierung, sind diese Knotenzeiger dann in der Lage, jeweils anzuzeigen, welcher Knoten als nächstes üeber einen Prefetch optimiert werden muß. Nachteil an diesem System ist der zusätzliche Speicher- und Rechenaufwand für die Verwaltung der Historienzeiger-Struktur. Außerdem hat dieses System nur dann einen Sinn, wenn die Datenstruktur mindestens ein zweites mal in der selben Weise traversiert wird, da ansonsten natürlich die Historienzeiger ungültig sind und erneut initialisiert werden müssen. Ist dies jedoch der Fall, zeigt diese Methode gute Ergebnisse.

- die *Daten-Linearisierung* : Hierbei müssen größere Änderungen am Programmcode durchgeführt werden, welche allerdings nicht immer machbar sein werden. Grundprinzip hierbei ist, Strukturelemente welche nacheinander traversiert werden auch nacheinander abzuspeichern. Sozusagen wird die Baum- oder Listenstruktur der Datenstruktur auf eine Feldstruktur abgebildet. Dies ist meist nur dann möglich, wenn die Struktur bereits bei der Anlage entsprechend initialisiert wird. Eine nachträgliche Modifikation einer bestehenden Struktur wird nur schwer möglich sein. Grundsätzlich läuft dies auf eine eigene Implementierung der dynamischen Speicherzuordnung der Elemente hinaus, was recht aufwendig erscheint.

8.3 Ergebnisse für rekursive Datenstrukturen

Von den drei Methoden wurden bisher nur das Greedy-Prefetching in einem Compiler implementiert. Für die anderen beiden wurden von Hand jeweils 10 Beispiele, welche die entsprechenden Datenstrukturen verwenden, optimiert. Auf eine Lokalitätsanalyse wurde grundsätzlich verzichtet, was die Ergebnisse zwar etwas verbessern, aber wohl nicht von ihrer Grundtendenz her ändern könnte.

Grundsätzlich sind die Ergebnisse des Greedy-Prefetchings nur äußerst bescheiden. Teilweise werden die Beispielprogramme langsamer, größtenteils sind sie unmerklich schneller. Wirkliche Verbesserungen sind nur dort sichtbar, wo das Programm genau dem Zugriffsschema, welches Greedy optimiert anwendet. Allerdings sind diese eher erfolglosen Ergebnisse auch darauf zurückzuführen, daß die meisten dieser

Programme eher wenig zeit damit verbringen, auf den Speicher zu warten. Speicherlatenz ist also hier eigentlich gar nicht das große Problem. Obwohl dieses einigermaßen zufriedenstellend gelöst wird, ist das Gesamtergebnis keineswegs merklich besser.

Für das Historienzeiger-Prefetching ist nur ein Ergebnis verfügbar, da dessen Anwendungsmöglichkeiten aufgrund der wiederholten gleichartigen Traversierung etwas eingeschränkt ist. Allerdings ist diese Methode dann sogar noch sehr viel besser als das Greedy-Prefetching, da weniger unnötige Prefetches veranlaßt werden und außerdem eine bessere Deckung der benötigten Prefetches erreicht wird.

Ähnliche Ergebnisse bietet die Methode der Datenlinearisierung. Dort wo diese Methode überhaupt anwendbar ist, zeigt sie bessere Ergebnisse als Greedy, da einfach weniger unnötige Prefetches veranlaßt werden.

Allerdings hat Greedy-Prefetching gegenüber den beiden anderen Methoden den Vorteil, daß es bereits automatisierbar ist, was wohl Grundvorraussetzung für seine Verbreitung sein dürfte.

9 Zusammenfassung

Programme, welche über Schleifen auf Feldern arbeiten, was für viele wissenschaftliche Programme zutrifft, können effektiver mit dem Speichersystem umgehen, wenn sie bald benötigte Daten rechtzeitig dort anfordern. Dies geschieht über das sogenannte 'Prefetching'. Möglich ist das deshalb, weil das Speichersystem so gleichmäßiger ausgelastet wird. Programme, welche das Speichersystem bereits bis zum Limit ausreizen, haben mit dieser Methode allerdings keinen Erfolg. Die maximale Beschleunigung, welche über das Prefetching zu erzielen ist, entspricht demnach auch eben dieser Wartezeit des Prozessors auf einen Hauptspeicherzugriff. Das Potential ist jedoch nicht zu vernachlässigen, da gerade die oben erwähnte Sorte Programme oft beinahe die halbe Zeit damit verbringt auf den Hauptspeicher zu warten, daher also bis zu 50% Optimierungspotential bergen.

Literatur

[1] *Tolerating Latency Through Software Controlled Data Prefetching*, PhD thesis, dept. of Computer Science, Stanford University, march 1994

[2] T.C. Mowry, M.S. Lam, A. Gupta. *Design and evaluation of a compiler algorithm for prefetching*, In Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 62-73, October 1992

[3] T.C. Mowry and C.-K. Luk, *Compiler-based prefetching for recursive data structures*, In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), ACM SIGPLAN Notices 31(9), pages 222-233, Cambridge, MA, October 1996

Seminar - Ausarbeitung VII

Gerd Flaig

Konsistenzmodelle

Zusammenfassung

Dieser Artikel beschäftigt sich mit den gängigsten Methoden, mit DSM[11]-Systemen umzugehen. Man bedient sich dabei sogenannter Konsistenzmodelle, die im wesentlichen festlegen, unter welchen Voraussetzungen das Speichersystem ein definiertes Verhalten zeigt. Die wichtigsten dieser Modelle werden kurz vorgestellt und abschließend verglichen.

1 Einleitung

In Mehrprozessorsystemen stellt sich das Problem, die Sicht der einzelnen Knoten[12] auf einen gemeinsamen Speicher konsistent zu halten. Der gemeinsame Speicher kann dabei wie im in Abbildung 1 dargestelleten busbasierten Multiprozessorsystem als realer Speicher oder bei einem System wie in Abbildung 2 rein virtuell existieren. Greift ein Knoten auf eine Speicherseite[13] zu, die auf einem anderen Knoten gehalten wird, so ist der Zugriff im allgemeinen deutlich langsamer als auf eine lokal gehaltene Seite. Um dieses Problem zu lockern, werden Caches eingesetzt, womit man sich jedoch ein anderes Problem einhandelt, nämlich die auf unterschiedlichen Knoten gehaltenen Kopien der Seite konsistent zu halten. Hierzu gibt es mehrere Ansätze, die im Folgenden besprochen werden. Im Wesentlichen handelt es sich dabei um Konsistenzmodelle, die festlegen, was ein Programmierer vom Speicher erwarten darf. Ein Konsistenzmodell kann man sich dabei als eine Art Vertrag zwischen dem Programmierer und dem Speicher vorstellen, indem Bedingungen formuliert werden, unter denen der Speicher ein definiertes Verhalten aufweist.

Ein Beispiel: In Abbildung 1 greift CPU 1 auf den gemeinsamen Speicher zu, um auf Speicherseite x schreibend zuzugreifen. Die Seite wird in den lokalen Cache des Knotens übertragen, kurze Zeit später greift CPU 3 lesend auf dieselbe Seite zu. Wenn der Lesezugriff garantiert den letzte geschriebenen Inhalt liefern soll, fällt entweder beim Schreiben oder beim Lesen unweigerlich eine Übertragung über das langsame Verbindungsnetzwerk an. In der Praxis ist es jedoch nicht in allen Fällen nötig, jede kleine Änderung sofort auf allen Knoten sichtbar zu machen. Wenn man nicht darauf beharrt, auf jeden Fall den letzten geschriebenen Wert zu erhalten, so wird es z.B. möglich, auf CPU 1 mehrere Schreiboperationen auf Seite x durchzuführen, ohne bei jedem Zugriff darauf zu warten, daß die Änderung auf allen Knoten sichtbar wird. Dies kann zu für Programmierer, die Uniprozessormaschinen gewöhnt sind anfänglich verwirrend wirken, da unter Umständen auf unterschiedlichen Knoten Schreibzugriffe in unterschiedlicher Reihenfolge sichtbar werden.

An dieser Stelle möchte ich, um die Beispiele möglichst klar zu halten, eine Notation einführen, die in [5] eingesetzt wird, um die zeitliche Abfolge von Speicherzugriffen in einem verteilten System darzustellen. Dabei sind die Zeilen des Diagramms jeweils einem Prozeß zugeordnet, die Zeitachse verläuft horizontal mit nach rechts aufsteigender Zeit. Die Symbole R(x)a und W(y)b bedeuten jeweils einen Lesezugriff auf Speicherzelle x mit dem Ergebnis a sowie einen Schreibzugriff auf Speicherzelle y mit dem Wert b. Weiterhin wird angenommen, daß alle Speicherzellen mit dem Wert 0 vorbelegt sind.

Für die Definition der einzelnen Modelle werden noch weitere Begriffe benötigt:

- Das Prädikat $W(p, t, x, i)$ ist genau dann wahr, wenn Prozeß p zum Zeitpunkt t in Speicherstelle x den Wert i schreibt.

[11] Distributed Shared Memory
[12] bestehend aus einer CPU mit eigenem Speicher
[13] normalerweise wenige Kilobyte

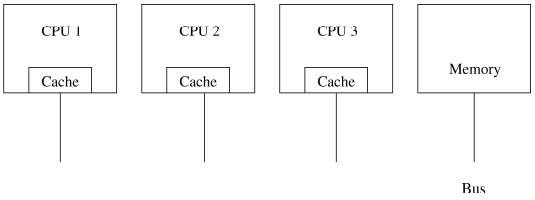


Abbildung 1: Busbasiertes Multiprozessorsystem

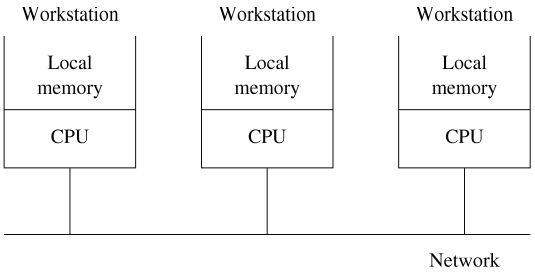


Abbildung 2: LAN-gekoppelter Multicomputer

- Die Gleichung $R(p,t,x) = i$ gibt an, daß Prozeß p zum Zeitpunkt t aus Speicherstelle x den Wert i liest.

2 Strikte Konsistenz

Das einfachste und und intuitiv einleuchtendste Konsistenzmodell ist die strikte Konsistenz. Sie ist wie folgt definiert: Ein Speichersystem ist *strikt konsistent*, wenn es folgende Bedingung erfüllt:

$$W(p,t_0,x,i) \Rightarrow (\exists t_1 > t_0 : \forall t : t_0 < t \leq t_1 \Rightarrow R(p',t,x) = i)$$

Das in Abbildung 3 aufgezeichnete Verhalten ist für in Speichersystem mit strikter Konsistenz akzeptabel. Im Gegensatz dazu verletzt das folgende Beispiel (Abbildung 4) die oben angegebene Bedingung für strikte Konsistenz, da der erste Lesezugriff einen anderen Wert liefert als denjenigen, der mit dem vorhergehenden Schreibzugriff geschrieben wurde.

Wie man leicht sieht, ist die Bedingung für strikte Konsistenz nicht erfüllt, wenn wir $p = P_1$, $t_0 = 0$, $i = 1$, $t_1 = 2$, $t = 1$ und $p' = P_2$ setzen.

3 Sequentielle Konsistenz

Definition: Ein Speichersystem ist *sequentiell konsistent*, wenn es nachfolgende Bedingung erfüllt:

P_1	W(x)1	
P_2		R(x)1

Abbildung 3: Beispieldiagramm

P_1	W(x)1			
P_2			R(x)0	R(x)1
t	0		1	2

Abbildung 4: Nicht strikt konsistenter Speicher

$$W(p,t_0,x,i) \wedge W(p,t_1,x,j) \wedge t_1 > t_0 \Rightarrow \exists t'_0, t'_1, t'_2 : t'_0 > t_0 \wedge t'_1 > t_1 \wedge t'_0 < t'_1 < t'_2 \\ \wedge \forall t : ((t'_0 \leq t < t'_1 \Rightarrow R(p',t,x) = i) \\ \wedge (t'_1 \leq t < t'_2 \Rightarrow R(p',t,x) = j))$$

Definition nach [4]: Ein Speichersystem ist sequentiell konsistent, wenn es die folgende Bedingung erfüllt:

> The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Damit ist unmittelbar einsichtig, daß aus strikter Konsistenz die sequentielle Konsistenz folgt, da erstere eine globale Totalordnung für die Zugriffe vorschreibt.

Aus obiger Definition folgt, daß ein Prozeß die Speicherzugriffe eines anderen Prozesses möglicherweise um eine unbestimmte Zeit versetzt sieht. Man kann sich nur darauf verlassen, daß die Speicherzugriffe eines Prozesses P von allen anderen Prozessen in genau der Reihenfolge gesehen werden, in der sie von P ausgeführt werden. Das Beispiel in Abbildung 4 erfüllt also die Bedingung für sequentielle Konsistenz.

Wir verdeutlichen uns den allgemeinen Fall anhand der Definition: Zu den Zeitpunkten t_0 und t_1 mit $t_0 < t_1$ finden nacheinander zwei Schreibzugriffe von Prozessor p statt. Daraus folgt für die möglichen Ergebnisse von Lesezugriffen eines beliebigen Prozessors p' zum Zeitpunkt t:

1. $\exists t'_0 > t_0 : R(p', t'_0, x) = i$
2. $\exists t'_1 > t'_0 > t_0 : R(p', t'_1, x) = j$

Das heißt, daß der Schreibzugriff $W(p,t_0,x,i)$ zeitlich um $\Delta t = t'_0 - t_0$ versetzt auf anderen Prozessoren sichtbar wird und daß der danach erfolgte Schreibzugriff $W(p,t_1,x,j)$ auf jeden Fall erst *nach* t'_0 sichtbar wird ($t'_1 > t'_0$).

Zur Verdeutlichung hier noch ein weiteres Beispiel: In Abbildung 5 sind drei parallele Prozesse abgebildet. Es gibt insgesamt 6! = 720 Möglichkeiten, die Anweisungen aller drei Programmfragmente in eine globale Reihenfolge einzuordnen. Bei sequentieller Konsistenz sind alle der in Abbildung 6 angegebenen Anordnungen erlaubt. Wie man sieht, können unterschiedliche Programmläufe unterschiedliche Ergebnisse haben. Im Beispiel sind die Ausgaben (001011, 101011, 111111) alle korrekt und es gibt noch weitere korrekte Ausgaben. Der Programmierer muß sich über diesen Indeterminismus im klaren sein und programmtechnisch entsprechend damit umgehen.

```
a = 1;          b = 1;          c = 1;
print (b, c);   print (a, c);   print (a, b);
```

Abbildung 5: Drei parallele Prozesse

4 Kausale Konsistenz

Definiton: Ein Speichersystem ist *kausal konsistent*, wenn es nachfolgende Bedingung erfüllt:

Code:	a = 1;	a = 1;	b = 1;
	print (b, c);	b = 1;	a = 1;
	b = 1;	print(a, c);	c = 1;
	print (a, c);	print(b, c);	print(a, c);
	c = 1;	c = 1;	print(b, c);
	print(a, b);	print(a, b);	print(a, b);
Ausgabe:	001011	101011	111111

Abbildung 6: Ausführungssequenzen

$$W(p,t_0,x,i) \wedge R(p',t_1,x) = i \wedge W(p',t_2,x,j) \wedge t_0 < t_1 < t_2$$
$$\Rightarrow \exists t_3, t_4 : t_2 < t_3 < t_4 \wedge R(p'',t_3,x) = i \wedge R(p'',t_4,x) = j$$

Das Modell für kausale Konsistenz[3] stellt eine Abschwächung der sequentiellen Konsistenz dar, indem es eine Unterscheidung zwischen möglicherweise kausal zusammenhängenden Speicherzugriffen und voneinander unabhängigen Zugriffen einführt.

Speichersysteme, für die die folgende Bedingung gilt, heißen kausal konsistent:

> Schreibzugriffe, die möglicherweise kausal zusammenhängen, müssen von allen Prozessen in derselben Reihenfolge gesehen werden. Nebenläufige Schreibzugriffe können auf unterschiedlichen Knoten in unterschiedlicher Reihenfolge gesehen werden.

Im Beispiel in Abbildung 7 hängt der Schreibzugriff W(x)2 in Prozess P_2 möglicherweise vom vorhergehenden Lesezugriff R(x)1 ab, z.B. weil der gelesene Wert 1 auf irgendeine Art in die Berechnung des später geschriebenen Werts 2 einfließt. Das Beispiel erfüllt die Bedingung für strikte Konsistenz nicht, weil P_3 die Schreibzugriffe W(x)2 und W(x)3 in umgekehrter Reihenfolge sieht. Es erfüllt auch die Bedingung für sequentielle Konsistenz nicht, da die Sequenzen R(x)2 R(x)3 und R(x)3 R(x)2 sich widersprechen.

P_1	W(x)1			W(x)3		
P_2		R(x)1	W(x)2			
P_3		R(x)1			R(x)3	R(x)2
P_4		R(x)1			R(x)2	R(x)3

Abbildung 7: Zugriffssequenz für kausal konsistenten Speicher

5 PRAM Konsistenz und Prozessor Konsistenz

Definition: Ein Speichersystem ist *PRAM konsistent*, wenn es nachfolgende Bedingung erfüllt:

$$W(p,t_0,x,i) \wedge W(p,t_1,y,j) \wedge t_0 < t_1$$
$$\Rightarrow \exists t_0', t_1' : t_0 < t_0' \wedge t_1 < t_1' \wedge t_0' < t_1' \wedge R(p',t_0',x) = i \wedge R(p',t_1',y) = j$$

PRAM (pipelined RAM) Konsistenz ist unter folgender Bedingung gegeben:

> Schreibzugriffe eines einzelnen Prozesses werden von allen anderen Prozessen in der Reihenfolge gesehen, in der sie ausgeführt wurden. Schreibzugriffe unterschiedlicher Prozesse können von verschiedenen Prozessen in unterschiedlicher Reihenfolge gesehen werden.

Mit anderen Worten: Es existiert u.U. keine globale Sequenz von Speicherzugriffen, die für alle Prozesse gilt. Die Zugriffe eines Prozesses bleiben in ihrer Reihenfolge unberührt, es gibt jedoch keine Garantien bezüglich der Struktur der Verschränkung von Zugriffen unterschiedlicher Prozesse. Der Unterschied zur kausalen Konsistenz besteht im wesentlichen darin, daß Schreibzugriffe eines Prozesses in eine Pipeline geschoben werden können, da auf den Abschluß der Zugriffe nicht gewartet werden muß.

Schreibzugriffe aller Prozesse sind untereinander nebenläufig.

P_1	W(x)1				
P_2		R(x)1	W(x)2		
P_3				R(x)1	R(x)2
P_4				R(x)2	R(x)1

Abbildung 8: Zugriffssequenz für PRAM Konsistenz

Abbildung 8 zeigt eine korrekte Zugriffssequenz für PRAM Konistenz. Das Beispiel erfüllt die Bedingung für strikte Konsistenz nicht, da P_4 die Schreibzugriffe W(x)1 und W(x)2 in umgekehrter Reihenfolge sieht. Sequentielle Konsistenz ist nicht gegeben, da sich die Sequenzen von P_3 und P_4 widersprechen. Kausale Konsistenz ist nicht gegeben, da die Zugriffe von P_2 möglicherweise kausal zusammenhängen und P_4 zuerst den Wert 2 und dann den Wert 1 liest. Daraus folgt, daß PRAM Konsistenz ein schwächeres Modell ist als die bisher vorgestellten.

PRAM Consistency und Processor Consistency unterscheiden sich nur in Details. [1] setzt sich mit diesen Unterschieden auseinander.

6 Schwache Konsistenz

Die bisherigen Modelle kamen ohne explizite Synchronisierungsoperationen aus. Dies gilt für die folgenden Modelle nicht mehr. Das Modell der schwachen Konsistenz führt Synchronisierungspunkte durch eine Trennung zwischen speziellen Variablen, den Synchronisationsvariablen, und sonstigen Variablen ein.

Das Modell wurde von [2] definiert als ein System, das folgenden Bedingungen genügt:

1. Zugriffe auf Synchronisationsvariable sind sequentiell konsistent.
2. Es ist kein Zugriff auf eine Synchronisationsvariable erlaubt, bis alle vorhergehenden Schreibzugriffe auf sie auf allen Knoten durchgeführt wurden.
3. Es ist kein Zugriff auf andere Variable erlaubt, bis alle vorhergehenden Zugriffe auf Synchronisationsvariable auf allen Knoten ausgeführt wurden.

Zugriffe auf Synchronisationsvariable erfüllen die Bedingung für sequentielle Konsistenz, außerdem wird garantiert, daß bei einem Zugriff auf eine Synchronisationsvariable alle vorhergehenden Schreibzugriffe ausgeführt wurden.

Eine der Grundannahmen bei schwacher Konsistenz ist, daß Zugriffe auf gemeinsame Variablen normalerweise in zusammenhängenden Blöcken, die durch längere Abschnitte, in denen kein Zugriff auf gemeinsame Variablen stattfindet, erfolgen.

7 Release Consistency

Ein Problem bei schwacher Konsistenz ist die fehlende Unterscheidung zwischen dem Betreten und Verlassen eines kritischen Abschnitts. Um die Grenzen von kritischen Abschnitten festzulegen, werden zwei zusätzliche Operationen eingeführt. Acq(L) kennzeichnet den Anfang, Rel(L) das Ende eines kritischen Abschnitts.

Abbildung 9 zeigt eine korrekte Zugriffssequenz für Release Consistency. P_2 erhält den zuletzt von P_1 in Speicherstelle x geschriebenen Wert, während P_3 ohne vohergehende Acq(L)-Operation liest und daher nicht garantiert den letzten Wert bekommt, der vor Rel(L) von P_1 geschrieben wurde.

P_1	Acq(L)	W(x)1	W(x)2	Rel(L)					
P_2					Acq(L)	R(x)2	Rel(L)		
P_3									R(x)1

Abbildung 9: Zugriffssequenz für Release Consistency

Im allgemeinen kann man ein Speichersystem als release consistent bezeichnen, wenn folgende Bedingungen erfüllt sind:

1. Bevor ein Zugriff auf eine gewöhnliche Variable ausgeführt wird, müssen alle Acquire-Operationen des Prozesses erfolgreich ausgeführt worden sein.
2. Bevor eine Release-Operation ausgeführt wird, müssen alle vorhergehenden Schreib- und Leseoperationen des Prozesses ausgeführt worden sein.
3. Die Acquire- und Release-Operationen müssen Processor Consistent sein (sequentielle Konsistenz ist nicht erforderlich).

8 Entry Consistency

Dieses Modell unterscheidet sich vom vorhergehenden Modell nur darin, daß die Synchronisation bei Betreten eines kritischen Abschnitts durchgeführt wird.

9 Implementierungsaspekte

Sequentielle Konsistenz ist am einfachsten zu erreichen. Der Trivialfall ist gegeben, wenn von jeder Speicherseite genau eine Kopie existiert, die nach Bedarf zwischen Knoten hin- und hertransferiert wird. Werden Seiten kopiert, auf die nur lesend zugegriffen werden kann, ist die sequentielle Konsistenz natürlicherweise weiterhin gegeben, da immer noch nur auf eine Kopie der Seite schreibend zugegriffen werden kann. Erst wenn mehrere schreibbare Kopien einer Seite existieren, müssen Maßnahmen zur Wahrung der Konsistenz ergriffen werden. Welche Maßnahmen das sind, hängt von der Art des Kommunikationsmediums zwischen den Knoten ab. Existiert ein zuverlässiger Broadcast-Mechanismus, so können Schreibzugriffe einfach über diesen Mechanismus allen Knoten bekannt gemacht werden. Hierbei kan entweder ein Tupel mit der geänderten Adresse und dem neuen Datum gesendet werden (Update) oder nur die geänderte Adresse (Invalidate).

Für Implementierungsaspekte bezüglich anderer Modelle sei an dieser Stelle auf die Literatur verwiesen, insbesondere auf [5].

10 Vergleich

Die besprochenen Konsistenzmodelle unterscheiden sich in ihren Einschränkungen, in der Komplexität der Implementierung, in ihren Ansprüchen an den Programmierer und in ihrer Performance. Sie lassen sich in zwei Kategorien einteilen: Modelle mit und ohne Synchronisierungsoperationen.

Der wesentliche Unterschied zwischen den einzelnen Konsistenzmodellen besteht darin, welche Umordnungen bei Speicherzugriffen erlaubt sind.

Die Modelle mit expliziten Synchronisierungsoperationen (Acquire/Release, Kennzeichnung von Variablen als Synchronisationsvariable) erlauben es dem Programmierer, innerhalb eines kritischen Abschnitts sequentielle Konsistenz anzunehmen.

11 Zusammenfassung

In der folgenden Tabelle werden die Schlüsseleigenschaften der vorgestellten Konsistenzmodelle kurz zusammengefasst:

Konsistenz	Beschreibung
strikt	Totale zeitliche Ordnung aller verteilten Zugriffe
sequentiell	Alle Prozesse sehen alle verteilten Zugriffe in derselben Reihenfolge
kausal	Alle Prozesse sehen alle kausal zusammenhängenden Zugriffe in derselben Reihenfolge
PRAM	Alle Prozesse sehen Schreibzugriffe eines Prozessors in der Reihenfolge, in der sie ausgeführt wurden
schwach	Gemeinsamer Speicher wird explizit synchronisiert
Release	Synchronisierung bei Beendigung eines kritischen Abschnitts
Entry	Synchronisierung bei Eintritt in einen kritischen Abschnitt

Es existieren eine Reihe von Konsistenzmodellen, die jeweils in verschiedenen Systemumgebungen sinnvoll einsetzbar sind. Anforderungen wie die verfügbare Kommunikationsbandbreite, Einfachheit der Programmierung, Struktur der Anwendung etc. beeinflussen die die Wahl entsprechender Systeme stark. Zum heutigen Zeitpunkt kann die Forschung auf dem Gebiet verteilter Speicher noch lange nicht als abgeschlossen betrachtet werden, und es ist sicherlich sinnvoll, sich mit den entsprechenden theoretischen Grundlagen vertraut zu machen.

Literatur

[1] M. Ahamad et al. The power of processor consistency. Technical report, College of Computing, Georgia Inst. of Technology, March 1993.

[2] M. Dubois et al. Memory access buffering in multiprocessors. In *Proc. 13th Ann. Int'l Symp. on Computer Architecture*, pages 434–442. ACM, 1986.

[3] P.W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proc. 10th Int'l Conf. on Distributed Computing Systems*, pages 302–311. IEEE, 1990.

[4] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28:690–691, Sept. 1979.

[5] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall, 1995.

Seminar - Ausarbeitung VIII

Florian Liekweg

TreadMarks — Ein Distributed-Shared-Memory-System für Workstation-Cluster

Zusammenfassung

TreadMarks ist ein in Software implementiertes DSM-System, das es erlaubt, einen Cluster von Workstations zu einem Parallelrechensytem zusammenzuschliessen. Mit der Flexibilität, die man daraus gewinnt, daß man die meist schon vorhandene Hardware nutzen kann, anstatt einen neuen GroSSrechner anzuschaffen, verbindet TreadMarks ein vergleichsweise komfortables Programmiermodell, das einerseits dem Benutzer die komplexe Kommunikation zwischen den einzelnen Workstations weitgehend verbirgt, andererseits jedoch weitergehende Optimierungen zulässt, um so die Leistung zu steigern. In Experimenten hat sich gezeigt, daß TreadMarks Leistungssteigerungen ermöglicht, die nahe am maximal Möglichen liegen.

Nach einem kurzen Überblick über andere Systeme gibt dieser Artikel eine Einführung in die Arbeitsweise mit TreadMarks und das zugrundeliegende Programmiermodell.

Den Hauptteil bildet die Vorstellung des inneren Aufbaus von TreadMarks. Hier werden die wichtigsten Mechanismen vorgestellt, aus denen TreadMarks seine Leistungsfähigkeit bezieht. Insbesondere sind dies die Release Consistency und das in TreadMarks implementierte Multiple-Writer-Protokoll.

Zum Abschluss werden einige Erweiterungen vorgestellt, mit denen die Leistung von TreadMarks weiter gesteigert werden kann. Die meist experimentell gewonnenen Ergebnisse zeigen, daß TreadMarks durchaus an die Geschwindigkeit anderer Systeme herankommt, ohne daß die in der Einleitung beschriebenen Vorteile aufgegeben werden müssen.

1 Einleitung

Trotz der großen Fortschritte in der Entwicklung von Prozessoren und Workstations gibt es nach wie vor Probleme, die selbst mit der heute vorhandenen Rechenleistung nicht in akzeptabler Zeit gelöst werden können. Dazu gehört zum Beispiel die Simulation von der Belastung und der davon verursachten Verformung von Materialien, wie sie in der Autoindustrie gefragt ist, um die Entwicklung sicherer Fahrzeuge auch ohne kostspielige Crash Tests zu ermöglichen. Ein weiteres Gebiet, das traditionell auf eine hohe Rechenleistung angewiesen ist, ist die Simulation von Strömungsvorgängen, wie sie bei der längerfristigen Wettervorhersage nötig ist. Weiterhin gibt es Gebiete, in denen zur Planung von Personal und Material Verfahren verwandt werden müssen, die nicht nur zu den „Musterbeispielen“ von NP-harten Problemen gehören, sondern bei denen in der Praxis genau die Instanzen zu lösen sind, die eine extrem hohe Rechenzeit provozieren.

Wegen dieser großen Nachfrage an „Rechenpower“ ist in den letzten Jahren viel Arbeit in die Entwicklung größerer, schnellerer Rechner geflossen. Bekannte Beispiele der letzten Zeit sind die Großrechner der Firma Cray und die neueren Modelle der Firma Sun Microsystems. Neben der Steigerung der Taktfrequenz der Prozessoren verspricht man sich auch eine Leistungssteigerung durch den Einsatz mehrerer Prozessoren in einem Rechner.

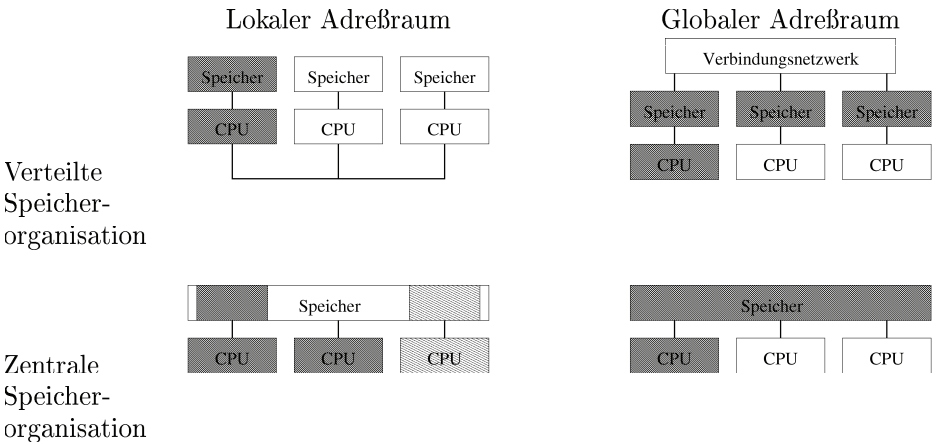


Abbildung 1: Konfigurationen von Multiprozessoren

1.1 Parallelrechner

In Abbildung 1 sind die heute üblichen Konfigurationen von Parallelrechensytemen nach [26] zusammengestellt. In dieser Abbildung wird durch Schattierungen angedeutet, in welchen Speicherbereichen eine CPU arbeitet. Diese konventionellen Multiprozessorsysteme unterscheiden sich nicht nur im Aufbau, sondern auch in der Arbeitsweise.

1.1.1 Multicomputer

Die linke Spalte von Abb. 1 zeigt die Konfigurationen für die sog. Multicomputer. Diese Maschinen verfügen über mehrere, voneinander unabhängig arbeitende Zentraleinheiten, von denen jede Einheit in ihrem lokalen Speicher arbeitet.

Im Vordergrund steht hier die Möglichkeit, mehrere unabhängige Prozesse auf dem gleichen Rechner ablaufen lassen zu können, wobei die Peripherie, die an das System gekoppelt ist, von allen Prozessen genutzt werden kann. Eine Kooperation der Prozessoren ist möglich, wenn die einzelnen Zentraleinheiten gekoppelt sind. Dies kann entweder dadurch geschehen, daß die Einheiten durch gemeinsame Speicher Daten austauschen, wie bei der IBM 3090/n00, oder durch Nachrichten, die zwischen den Einheiten durch ein Verbindungsnetzwerk ausgetauscht werden.

1.1.2 Multiprozessorsysteme

Die Abarbeitung eines einzigen Algorithmus' durch mehrere Prozessoren erfordert eine globale Speicherstruktur, wie sie in der rechten Spalte von Abb. 1 gezeigt wird.

Die Organisation des Speichers als einheitlichen Zentralspeicher reduziert den Programmieraufwand für einen parallel auszuführenden Algorithmus auf die Synchronisation der auf den verschiedenen Prozessoren ausgeführten Programmen. Ist der Algorithmus jedoch nicht nur rechen-, sondern auch datenintensiv, so kann der Zentralspeicher leicht zu einem Engpass werden: Wollen mehrere Prozessoren gleichzeitig auf den Speicher zugreifen, so kann nur einer der Prozessoren den Zugriff durchführen, während die anderen Prozessoren warten müssen.

Ein weiteres Problem, das in erster Linie für eine zentrale Speicherorganisation gelöst werden muß, ist das der Cache-Kohärenz. Abbildung 2 zeigt noch einmal ein Multiprozessorsystem mit Zentralspeicher; im Gegensatz zu Abb. 1 sind diesmal die Cache-Speicher mit eingezeichnet. Die Cache-Speicher, über die die CPU auf den Hauptspeicher zugreift, speichern bei einem Ladezugriff der CPU zusammen mit der verwendeten Speicheradresse den Inhalt, der bei diesem Ladezugriff vom Speicher geliefert wird. Bei einem erneutem Zugriff auf dieselbe Adresse kann der Cache den Inhalt der angesprochenen Speicherzelle liefern, ohne daß ein erneuter, in der Regel sehr zeitaufwendiger Zugriff auf den Hauptspeicher erfolgen muß.

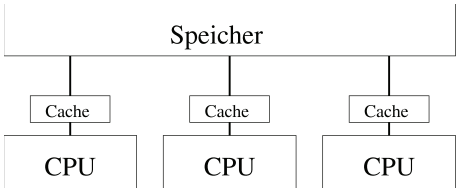


Abbildung 2: Zum Problem der Cache-Kohärenz: Da die Cache-Speicher wiederholte Lade- und Speicherzugriffe auf dieselbe Adresse bedienen, ohne jedesmal auf den Hauptspeicher zuzugreifen, können dem lesenden Prozessor Änderungen, die von anderen Prozessoren vorgenommen wurden, verborgen bleiben. Weiterhin führt die verzögerte Durchführung von schreibenden Zugriffen dazu, daß diese von anderen Prozessoren nicht wahrgenommen werden.

Wird allerdings dieselbe Speicherzelle von einem anderen Prozessor beschrieben, so wird diese Tatsache durch den Cache verdeckt. Erst ein Abgleich zwischen jedem einzelnen Cache und dem Zentralspeicher bewirkt, daß die Prozessoren mit aktuellen Daten arbeiten. Auch einen schreibenden Zugriff kann der Cache abfangen, um zu vermeiden, daß der schreibende Prozessor auf die Beendigung des Vorganges warten muß; auch hier wird der geschriebene Wert für die anderen Prozessoren erst nach einem Abgleich zwischen Cache und Zentralspeicher sichtbar.

Diesen beiden Aspekten der Problematik begegnet man mit Konsistenz-Protokollen, in denen festgelegt ist, wie die problematischen Situationen erkannt und behandelt werden. Bei älteren Rechnern findet sich noch eine Implementation in Software; inzwischen wird die Abarbeitung dieser Protokolle in die Hardware der Cache-Speichern integriert. Häufig anzutreffende Protokolle sind das MESI- und das etwas einfachere MSI-Protokoll. Weitere Informationen zu diesem Thema finden sich in [26]; eine ausführlichere Einführung bietet [25] und [7].

1.2 Ebenen der Parallelität

Die unterschiedlichen Organisationsformen der Rechner unterstützen verschiedene Arten der Parallelität. Ohne die sprachsyntaktischen Konstrukte zu berücksichtigen, lassen sich die Rechner und damit auch die Programme, die auf ihnen ablaufen, nach der Granularität klassifizieren. Sowohl die Einteilung der Granularität in verschiedene Stufen der Parallelität als auch die Einordnung eines konkreten Programms in eine wie auch immer gestaltete Einteilung ist nicht immer eindeutig vorgegeben. Im folgenden wird die fünfstufige Einteilung nach [26] verwandt:

Programmebene, Jobebene: Diese Ebene wird durch parallele Verarbeitung voneinander unabhängige Programme charakterisiert, wie sie für Multicomputer üblich ist. Die Parallelverarbeitung dieser Ebene wird durch das Betriebssystem organisiert. Ein Beispiel für ein Betriebssystem, das einen Rechner, der mit mehreren Prozessoren bestückt ist, so betreiben kann, ist LINUX/SMP.

Prozeß- oder Taskebene: Hier wird ein einziges Programm in eine Anzahl parallel auszuführender Prozesse zerlegt. Jeder der einzelnen Prozesse besteht aus einigen tausend sequentiell auszuführenden Befehlen und operiert auf eigenen Datenbereichen. Typische Beispiele sind UNIX-Prozesse und Tasks in Ada. Auch hier leistet das Betriebssystem Unterstützung, indem es Primitive zur Interprozeßkommunikation und -kooperation sowie zur Synchronisierung der einzelnen Tasks zur Verfügung stellt. Ein Beispiel für eine Workstation, auf der diese Art der Parallelarbeit implementiert ist, bieten die SPARC-basierten Mehrprozessorsysteme der Firma Sun Microsystems, bei denen das Betriebssystem Solaris die Zuordnung zwischen Prozessoren und Prozessen bzw. Jobs dynamisch vornimmt.

Blockebene: Im Gegensatz zur vorgenannten Prozeßebene arbeiten hier die parallel auszuführenden sog. Tread oder Light-Weighted Processes in einem gemeinsamen Adreßraum. Ein Beispiel hierfür bieten die Tread gemäß dem POSIX 1003.4a-Standard. Auch die Parallelisierung, die der weiter unten erwähnte APR-XHPF-Übersetzer der Firma IBM vornimmt, fällt in diese Klasse.

Anweisungsebene: Auf dieser Ebene werden mehrere aus Sicht der Maschine atomare Anweisungen gleichzeitig ausgeführt. Diese Art der Parallelität findet auf Hardwareebene statt. Allen Implementationen ist gemein, daß der Programmierer hier kaum mehr Eingriffsmöglichkeiten hat; die Parallelisierung erfolgt entweder in speziellen Übersetzern oder wird durch die Hardware selbst erledigt.

Ein Beispiel hierfür sind VLIW-Prozessoren, für die ein Übersetzer einen Befehlsstrom generiert, der in jedem der very long instruction words mehrere voneinander unabhängige Befehle enthält, die im Prozessor durch entsprechend viele Ausführungseinheiten bearbeitet werden. Ein weiteres Beispiel findet sich in den meisten der heute erhältlichen Mikroprozessoren, die vornehmlich superskalar arbeiten. Auf diesen Prozessoren sind die Ausführungseinheiten zum Teil mehrfach vorhanden; in jedem Falle können alle Ausführungseinheiten voneinander unabhängig arbeiten. Ein superskalarer Prozessor analysiert den eingehenden Befehlsstrom, und verteilt möglichst viele der eingehenden Befehle auf die Ausführungseinheiten.

Suboperationsebene: Hier werden Maschinenanweisung im Prozessor in Suboperationen aufgebrochen, die dann parallel ausgeführt werden. Der Einsatz dieser Prozessoren ist auf die wenigen Spezialfälle beschränkt, in denen sich der Programmcode zu dieser Art von Parallelisierung anbietet. Beispiele sind Vektor- und Feldrechner, die über langen Datenreihen arbeiten, wobei auf jedem einzelnen Wert die gleiche Operation ausgeführt wird; auch der XSC-Prozessor arbeitet auf diese Weise beliebig lange Skalarprodukte ab, wobei die Eingabe aus zwei Datenreihen besteht, von denen Wertpaare multipliziert und die Summe dieser Produkte akkumuliert wird. Näheres zu diesem Prozessor findet sich in [15] und [16].

1.3 Programmiermodelle

Neben den verschiedenen Möglichkeiten, Multiprozessorsysteme zu organisieren und zu implementieren interessieren den Anwender bzw. den Programmierer in erster Linie die Programmiermodelle, auf denen sie ihre Software erstellen. Die Programmiermodelle beschreiben eine abstrakte Maschine, die die erstellte Software ausführt.

Die Bewertung der unterschiedlichen Programmiermodelle kann in Hinblick auf zweierlei Aspekte erfolgen:

- Aus Sicht des Anwendungsprogrammierers steht der praktische Einsatz des Modells im Vordergrund. Er orientiert sich daran, wieviel Aufwand er auf den Betrieb des Modells verwenden muß, da dieser Aufwand von der Zeit abgeht, die ihm für die Entwicklung der eigentlichen Software bleibt. Weiterhin wird er beachten, ob und wie ihn das Programmiermodell bei seiner Arbeit einschränkt, indem es die Änderung, Erweiterung oder Wiederverwendung bereits erstellter Software erschwert.

- Aus Sicht derjenigen, die das Modell erstellen, ist es wichtig, einen gangbaren Kompromiß zu finden zwischen den Wünschen der Anwendungsprogrammierer und der zugrundeliegenden Hardwarearchitektur. Einerseits darf man, wie schon erwähnt, dem Anwender nicht zuviel Kleinarbeit aufbürden, andererseits sollte man jedoch der Endanwendung einen möglichst großen Anteil der vorhandenen Rechenkraft zur Verfügung stellen.

Im folgenden Abschnitt werden einige Alternativen vorgestellt, die zur Zeit Anwendung finden.

1.4 Programmierumgebungen

1.4.1 HPF — High Performance Fortran

Ein Übersetzer für High Performance Fortran bietet die Möglichkeit, ein `Fortran`-Programm so zu übersetzen, daß es auf einem Parallelrechner ausgeführt werden kann. Der Übersetzer versucht, in den Schleifen des Programms Muster zu erkennen, für die der Übersetzerbauer festgelegt hat, wie sie zu parallelisieren sind. Solange das Programm solche Muster enthält, die der Übersetzer kennt, kann er sehr effektiven Parallelcode erzeugen, ohne daß der Programmierer einzugreifen braucht. Diejenigen Konstrukte, die zwar

parallelisiert werden könnten, jedoch nicht vom Compiler erkannt werden, werden weiterhin sequentiell ausgeführt. Die Möglichkeiten des Programmierers, hier einzugreifen, sind sehr beschränkt.

1.4.2 PVM — Parallel Virtual Machine

PVM bietet dem Programmierer die Parallel Virtual Machine, ein DSM-System, bei dem die Prozessoren kommunizieren, indem sie sich gegenseitig Nachrichten zusenden.

Um von Prozessor p zu Prozessor q Nachrichten verschicken zu können, muß für p eine Datensenke deklariert werden, bei q die zugehörige Datenquelle. Die Daten, die p in die Datensenke gibt, werden durch die PVM-Software zu Prozessor q geleitet, dem sie an der Datenquelle in FIFO-Reihenfolge zur Verfügung stehen. PVM bietet sowohl blockierende als auch nicht-blockierende Sende- und Empfangsoperationen an.

Mit PVM läßt sich die Kommunikation zwischen den Prozessoren sehr präzise steuern. Ein gutes PVM-Programm kann dadurch ein DSM-System besonders gut ausnutzen, so daß solche Programme als Maßstab dafür dienen können, welche Leistung sich mit einer gegebenen Hardware überhaupt erreichen läßt.

Die Programmierung mit PVM ist jedoch sehr aufwendig. Der Programmierer muß nicht nur den Algorithmus parallelisieren, sondern zusätzlich nachvollziehen, *welcher* Prozessor *wann welche* Daten braucht, um die Kommunikation an den Datensenken und -quellen korrekt implementieren zu können. Diese ausprogrammierte Kommunikation kann einen erheblichen Anteil am Gesamtprogramm und damit an der Arbei des Programmierers ausmachen. Zudem läßt sich die Kommunikation selbst bei nur kleinen Änderungen am Algorithmus nur schlecht wiederverwenden, so daß auch die Überarbeitung eines PVM-Programms sehr aufwendig ist.

Eine Möglichkeit, diese komplexen Programmierarbeit zu reduzieren, besteht in der Verwendung eines parallelisierenden Übersetzers, wie dem Forge XHPF-Compiler [22], der als Präprozessor arbeitet, und ein annotiertes `Fortran 90`-Programm in ein parallelisiertes `Fortran 77`-Programm umschreibt. Allerdings sind solche Übersetzer bzw. Präprozessoren selbst wiederum kompliziert und schwer zu implementieren. Die Autoren von [6] weisen darauf hin, daß ein Präprozessor wie [21] für ein DSM-System wie wesentlich einfacher zu implementieren ist und auch effektiver arbeiten kann.

1.4.3 Das Inspector-Executor-Verfahren

Mit Inspector-Executor bezeichnet man ein Verfahren, das es ermöglichen soll, irreguläre Applikationen effizient auf einem DSM-System auszuführen. Bevor die eigentliche Berechnung durch den Executor beginnt, durchläuft der Inspector den Programm-Code, und erstellt unter Beachtung der Speicherzugriffe einen Plan, nach dem die Daten währen der nachfolgenden Executor-Phase zwischen den Workstations verteilt werden können.

In diesem Plan können Nachrichten identifiziert werden, die zusammengefaßt werden können, wodurch der Kommunikationsaufwand verringert wird. Einen weiteren Geschwindigkeitsvorteil erreicht man dadurch, daß der Inspector in manchen Fällen aus der Schleife „ausgelagert“ werden kann, d. h. nur ein einziges Mal über eine Schleife läuft, die dann in der Executor-Phase viele Male ausgeführt wird. Ein Nachteil besteht darin, daß die Analyse, die der Übersetzer bei der Übersetzung des Programms ausführen muß, um solche Fälle zu erkennen, äußerst kompliziert werden kann.

2 Synchronisation in Multiprozessorsystemen

2.1 Einleitung

In diesem Abschnitt werden Mechanismen vorgestellt, mit deren Hilfe man die Ausführung der zueinander parallel ablaufenden Kontrollfäden eines Parallelprogramms synchronisieren kann. Mit Synchronisation läßt sich erreichen, daß die verschiedenen Kontrollfäden des Programms ihren Ablauf aufeinander abstimmen, so daß die Fäden Operationen gleichzeitig ausführen, oder eben *nicht* gleichzeitig ausführen, sondern aufeinander warten.

Als zugrundeliegendes Modell kann man sich immer ein Mehrprozessorsystem vorstellen, bei dem also mehrere Prozessoren auf einem globalen Speicher arbeiten. Ob der Speicher als zusammenhängender Zentralspeicher oder als auf die lokalen Speicher verteilter virtueller Speicher organisiert ist, spielt für diesen Abschnitt keine Rolle. Der Befehlsstrom, den die Prozessoren ausführen, bezeichnet man oft als Kontrollfaden (Thread); mit Blick auf das TreadMarks-System, in dem immer eine Workstation mit einem Prozessor einen dieser Kontrollfäden ausführt, werden im folgenden Abschnitt die Begriffe „Prozessor“ und „Kontrollfaden“ synonym gebraucht.

In seinem Buch [27] identifiziert der Autor eine Vielzahl von Interaktionsaktionen und die dazu nötigen Kommunikations- und Kooperationsoperationen und beschreibt sie in allen Details. In der Praxis zeigt sich jedoch, daß man mit einer vergleichsweise geringen Menge an Synchronisationsoperationen auskommt, da ein großer Teil der Gesamtheit der in [27] beschriebenen Mechanismen eher akademischen Charakter hat, und man durch geringfügige Anpassungen des Programms weitere Mechanismen überflüssig machen kann.

Die in der Praxis wohl am häufigsten verwendeten Mechanismen sind die Locks und die Barriers.

2.2 Locks

Ein Lock ist ein Vorrecht, das höchstens ein Prozessor besitzen kann. Die Anforderung und Weitergabe dieses Vorrechts geschieht durch zwei Primitive, der `acquire`- und der `release`-Operation:

`acquire`: Mit diesem Primitiv kann ein Prozessor das Lock anfordern. Ist momentan ein anderer Prozessor im Besitz des Locks, so muß der Prozessor, der das `acquire` ausführt, warten, bis der derzeitige Besitzer des Locks dieses aufgibt. Argument zu dieser Operation ist das Lock, das angefordert wird.

`release`: Mit diesem Primitiv gibt ein Prozessor ein Lock, das er zuvor über ein `acquire` angefordert hat, auf, so daß ein anderer Prozessor, der eventuell infolge einer `acquire`-Operation darauf wartet, dieses in Besitz nehmen und seine Programmausführung fortsetzen kann. Auch hier wird als Argument das betreffende Lock übergeben.

Mit Locks kann man erreichen, das ein Programmabschnitt zu jedem Zeitpunkt von höchstens einem Prozessor durchlaufen wird, indem man für diesen Abschnitt ein Lock vorsieht, und vor Eintritt in diesen Abschnitt die Ausführung eines `acquire` an diesem Lock vorschreibt. Nachdem der Abschnitt beendet wurde, wird das Lock über die `release`-Operation wieder zur Verfügung gestellt.

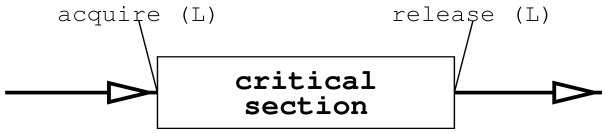


Abbildung 3: Gegenseitiger Ausschluß innerhalb eines kritischen Abschnittes: Vor Eintritt in den Abschnitt fordert der Prozessor das Lock an; nach Abarbeitung des Abschnittes stellt er es wieder zur Verfügung.

2.3 Barriers

Mittels Barriers kann man erreichen, daß die Prozessoren an einer Stelle im Programm aufeinander warten, um dann gemeinsam die Ausführung des Programmes fortzusetzen. Für eine Barrier ist nur ein einziges Primitiv nötig: Die `barrier`-Operation:

`barrier`: Ein Prozessor, der diese Operation ausführt, wird solange angehalten, bis alle anderen Prozessoren ebenfalls diese Operation ausgeführt haben. Argument zu dieser Operation ist die ID-Nummer der betreffenden Barrier.

In Abbildung 4 ist ein Programmablauf skizziert, der den Kern des in Abb. 18 beschriebenen SOR-Verfahrens umfasst. Das Verfahren arbeitet in einer zweistufigen Iteration: In `step 1` wird von jedem Prozessor im lokalen Speicher ein Teil der Matrix berechnet, in `step 2` werden die Teile in den globalen Speicher zurückkopiert. Dann wird nach Überprüfung einer Abbruchbedingung entschieden, ob ein weiterer Iterationsschritt nötig ist, oder ob die Iteration abgebrochen werden kann. Vor jedem Schritt warten die Prozessoren aufeinander, um zu verhindern, daß ein Prozessor einen neuen Teilschritt beginnt, bevor ein anderer Prozessor den vorhergehenden Teilschritt beendet hat.

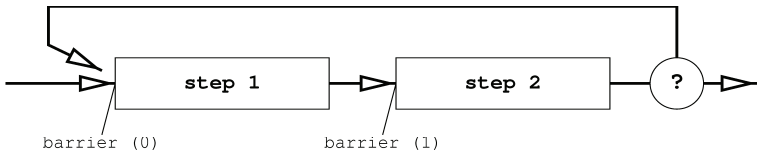


Abbildung 4: Synchronisation mittels Barriers: Am Ende jedes Teilschritts (`step 1` und `step 2`) warten die Prozessoren aufeinander, um zu verhindern, daß ein Prozessor einen neuen Teilschritt beginnt, bevor die anderen Prozessoren den letzten Teilschritt beendet haben.

Man kann sich leicht klarmachen, daß in einem Programm höchstens *eine* Sperre zu existieren braucht: Angenommen, zwei Prozessoren, p und q führen ein Programm aus, und p kommt an eine Sperre S an, während q an Sperre T wartet. Dann wird p die Sperre S nicht verlassen, ehe q ebenfalls bis zu Sperre S gekommen ist, während q nicht weiterlaufen kann, bevor p ebenfalls an T angekommen ist. Damit kann keiner der Prozessoren weiterarbeiten, bevor der jeweils andere weiterläuft; die beiden Prozessoren sind *verklemmt*.

Die Datenstrukturen, die ein DSM-System für die Implementation der Barriers vorhält, müssen also genau *einmal* vorhanden sein, da jede Situation, in der zwei verschiedene Sperren gleichzeitig benutzt werden, eine Verklemmung darstellt.

Für den Software-Entwickler stellt sich jedoch die Situation etwas anders dar. Angenommen, durch einen Programmierfehler entsteht ein Programmzustand, in dem ein Prozessor p an einer Stelle im Programm auf eine Sperre läuft, z. B. vor `step 1` im obigen Beispiel, während ein anderer Prozessor q fälschlicherweise das Warten an Barrier 0 überspringt, `step 1` ausführt, und erst am Anfang von step 2 wartet. Ist im DSM-System nur eine einzige Sperre implementiert, so wird dieser Fehler dazu führen, daß nun p den `step 1` ausführt, während q den `step 2` abarbeitet, worauf beide Prozessoren wiederum an den falschen Stellen aufeinander warten.

Das Programm wird also ablaufen, ohne den Fehler in irgendeiner Weise zu melden; die Ausgabe, die es liefert, ist jedoch fehlerhaft. Das dies jedoch erkannt wird, ist gerade bei den Problemen, für die man das TreadMarks-System einsetzen wird, sehr zweifelhaft. Außerdem tut das Vertrauen, daß man gerne fälschlicherweise in die moderne Computertechnologie setzt, ein Übriges.

Stellt das DSM-System jedoch *verschiedene* Sperren zur Verfügung, so kommt es bei dem Programm aufgrund des Programmierfehlers zu einer Verklemmung, die einen (scheinbar) normalen Programmablauf unmöglich macht. Zum einen wird der Fehler damit bemerkbar, zum anderen hat der Programmierer die Möglichkeit, mit einem Debugger das fehlerhafte Programm zu untersuchen, um so festzustellen, welcher Prozessor an welcher Stelle welche Sperre übersprungen hat.

3 Konsistenz-Modelle

Dieser Abschnitt leitet von den allgemeinen Problemen, die sich bei der Implementation eines DSM-Systems ergeben, über zu den möglichen Lösungen; näher beschrieben werden speziell die Lösungen, für die sich die Urheber von TreadMarks entschieden haben.

Nach dem ‘klassischen’ Konsistenzmodell, der Sequential Consistency werden abgeschwächte Konsistenzmodelle angesprochen, die Entry Consistency, das Structured Memory-Modell, und schliesslich das Modell, das in TreadMarks verwendet wird, die Release Consistency.

Ein weiteres Problem, der hier betrachtet wird, ist das des false sharing, das in DSM-Systemen auftreten kann. Anschliessend wird das Multiple-Writer-Protokoll erläutert, wie es in TreadMarks implementiert ist.

3.1 Vorüberlegungen

Bereits in Abschnitt 1.1.2 wurde unter dem Stichwort Cache Consistency ein Problem angesprochen, das für Multiprozessorsysteme mit zentral organisiertem Speicher gelöst werden muß. Es tritt allerdings nur deshalb auf, weil die meisten modernen Multiprozessorsysteme mit Cache-Speichern ausgerüstet sind, und dadurch für dieses Problem anfällig werden.

Für Multiprozessorsysteme, die auf einem verteilten Speicher arbeiten, ergibt sich ein analoges Problem: Wie sollen die Daten behandelt werden, die von mehr als nur einem Prozessor bearbeitet (d. h. gelesen und/oder geschrieben) werden?

Die Daten, die von mehr als einem Prozessor bearbeitet werden, können entweder migriert oder repliziert werden.

Migration der Daten bedeutet, das die betreffenden Daten immer genau *einmal* vorhanden sind, und dies vorteilhafterweise immer bei *dem* Prozessor, der sie gerade benötigt. Tritt im Ablauf des Programmes ein anderer Prozessor auf, der die Daten bearbeiten will, so werden sie dem 'alten' Prozessor entzogen und zu dem 'neuen' Prozessor transportiert.

Der Vorteil dieser Methode besteht darin, daß im Gegensatz zur Replikation die Daten genau einmal existieren, so daß jeder Prozessor, der die Daten anfordert, die Daten so erhält, wie sie zuletzt bearbeitet wurden. Es kann also nicht vorkommen, daß ein Prozessor eine Kopie der Daten verwendet, während ein anderer Prozessor an seiner privaten Kopie derselben Daten schon Änderungen vorgenommen hat.

Als nachteiliger Effekt ist die hohe Anzahl der Nachrichten zu bemerken, die zwischen den Prozessoren ausgetauscht werden. Auf einem in Hardware implementierten DSM-System mag dieser Effekt noch nicht zu einer merklichen Beeinträchtigung der Leistung führen; in der Hardwarekonfiguration jedoch, auf der TreadMarks läuft, geschieht der Nachrichtenaustausch über ein Netzwerk, das eine hohe Latenzzeit hat, und damit viel Zeit für die Nachrichtenübermittlung braucht. Dieser Nachteil fällt so schwer ins Gewicht, daß eine Strategie, die auf Migration von Daten beruht, für TreadMarks nicht in Frage kommt.

Replikation der Daten heisst, daß die Daten in mehreren Kopien existieren, wobei jedem Prozessor, der die Daten bearbeitet, eine Kopie zur Verfügung gestellt wird.

Der Vorteil liegt darin, daß einem Prozessor die Daten, die er gerade benötigt, zur Verfügung stehen, unabhängig davon, wieviele andere Prozessoren dieselben Daten gerade brauchen. Die Verbreitung von Änderungen, die ein Prozessor an seiner lokalen Kopie vorgenommen hat, kann hinausgezögert werden, bis die Änderungen für andere Prozessoren unbedingt sichtbar werden müssen; außerdem besteht die Möglichkeit, für den Austausch eine Menge von Daten zu aggregieren, die dann mit nur einer einzigen Nachricht verteilt werden. Wie schon erwähnt, 'leidet' das bei TreadMarks verwendete Netzwerk unter einer hohen Latenzzeit, die die Kosten pro Nachricht in die Höhe treibt; die Kosten pro übertragenem Byte jedoch sind auf niedrigem Niveau konstant.

Der Nachteil besteht offenbar darin, daß die Nachrichten zur *richtigen* Zeit an die *richtigen* Prozessoren verteilt werden müssen. Diese Aufgabe kann das DSM-System nicht allein erledigen.

Der in TreadMarks gewählte Ansatz folgt dem Replikations-Modell. Um das DSM-System zu unterstützen, wird der Programmierer in diese Aufgabe mit eingebunden, allerdings ohne das er dabei mit implementations- oder hardwarespezifischen Details überladen wird. Es bleibt ihm also nach wie vor möglich, sich auf die Entwicklung der Algorithmen zu konzentrieren, ohne daß er einen übermässig großen Teil seiner Arbeit auf die Dressur von TreadMarks verwenden muß.

Der Speicher in den meisten Rechnern ist auf Hardwareebene als (Bi-)CMOS-RAM implementiert, also als eine Liste aus einzeln adressierbaren Zellen; eine Zelle ist typischerweise ein Byte. Während dieses

lineare Speichermodell auch noch in der Speicherverwaltung mancher Programmiersprachen rudimentär vorhanden ist (vgl. `malloc` und die Semantik von Array-Variablen in `C`), stellen höhere Programmiersprachen heute durchgehend den Speicher in Form von typisierten Variablen (Fest- und Gleitkommazahlen, Buchstaben und -ketten) bzw. Objekten zur Verfügung. Für DSM-Systeme stellt sich die Frage, in welcher Form der Speicher gehandhabt werden soll:

Strukturierter Speicher: Hier betrachtet auch das DSM-System den Speicher als eine Menge von Tupeln oder Objekten, die zwischen den Prozessoren transportiert werden müssen. Die Granularität der Daten, die transportiert werden, steht hiermit fest, und kann nicht an andere 'Gegebenheiten' angepasst werden; insbesondere wird nicht berücksichtigt, wie mit dem Transportmechanismus, hier dem Netzwerk, umgegangen werden kann.

Linearer Speicher: Unabhängig von der Sichtweise des Programms betrachtet das DSM-System den Speicher als eine lineare Reihung von Speicherzellen. Vorteilhaft macht sich hier bemerkbar, das diese Betrachtungsweise die Möglichkeit eröffnet, die Größe der Konsistenzeinheiten, d. h. die Anzahl der Speicherzellen, die ggf. zwischen den Prozessoren ausgetauscht werden, so zu wählen, daß sie an die Arbeitsweise des Netzwerks angepasst ist. Das für TreadMarks verwendete Netzwerk verursacht (wie schon gesagt) hohe Kosten pro Nachricht, jedoch nur geringe Kosten pro übertragenem Byte einer Nachricht. Man wird also bestrebt sein, die Konsistenzeinheiten so groß wie möglich zu wählen, daß möglichst viele Daten pro Nachricht übertragen werden können.

Ein Nachteil ist, daß die Konsistenzeinheiten in der Regel nicht deckungsgleich mit den Daten sind, die zwischen den Prozessoren ausgetauscht werden müssen.

In TreadMarks wird die zweitgenannte Methode angewandt. Dadurch eröffnet sich nicht nur die Möglichkeit, die Größe der Konsistenzeinheiten an die charakteristischen Größen des Netzwerkes anzupassen; zusätzlich kommt der Speicherverwaltung von TreadMarks zugute, daß die Konsistenzeinheiten identisch mit der SeitenGröße der virtuellen Speicherverwaltung sind. Dadurch kann die MMU (Memory Management Unit) der Prozessoren zur Speicherverwaltung eingesetzt werden.

Beschreibt ein Prozessor einen Datenblock, der auch im lokalen Speicher anderer Prozessoren vorhanden ist, so muß sichergestellt werden, daß allen anderen Prozessoren diese Änderung mitgeteilt wird. Die einfachste Methode ist sicherlich, die Änderung sofort an alle betroffenen Prozessoren weiterzuleiten. Es genügt jedoch, wenn den anderen Prozessoren mitgeteilt wird, *daß* der Datenblock modifiziert wurde, und vor erneuter Benutzung vom letzten Schreiber angefordert werden sollte.

Update-Strategie nennt man diejenige Strategie, die Änderungen sofort propagiert. Dadurch stehen den beteiligten Prozessoren sofort die aktuellen Daten zur Verfügung. Die Tatsache, daß die Daten nicht von jedem Prozessor auch unbedingt gelesen werden, führt hier zu einem überflüssigen Datenverkehr zwischen den Prozessorknoten, wodurch diese Strategie der nachfolgend beschriebenen Methode des Write-Invalidate unterlegen ist.

Write-Invalidate bedeutet, daß die veralteten Daten, die sich im lokalen Speicher eines Prozessorknotens befinden, als ungültig markiert werden, wenn eine Kopie dieser Daten von einem anderen Prozessor modifiziert wird. Vor einem (lesenden) Zugriff auf solchermassen markierte Daten müssen die aktuellen Werte des letzten Schreibers angefordert werden. Der Vorteil dieser Methode liegt darin, daß nur die Daten angefordert und damit übertragen werden, die wirklich gelesen werden. Ein Nachteil liegt in der Logik, die implementiert werden muß. In einem in Hardware implementierten System kann dies Schwierigkeiten bereiten; in einem Software-System wie TreadMarks ist dies jedoch kein Problem.

In TreadMarks wird das Write-Invalidate-Verfahren angewandt. Ausschlaggebend ist ein weiteres Mal der hohe Zeitaufwand, der bei der Nachrichtenübertragung durch das Netzwerk entsteht. Im folgenden werden sich übrigens noch weitere Entscheidungen und Optimierungen an den Leistungsdaten des verwendeten Netzwerkes orientieren.

3.2 Sequential Consistency

Der Programmierer geht implizit davon aus, daß ein Lesezugriff auf den Speicher immer den Wert liefert, der zuletzt geschrieben wurde. In einem Einprozessorsystem ist dies das intuitive Speichermodell; wenn jedoch mehrere Prozessoren auf denselben Speicher zugreifen, ist der Begriff des „Wertes, der zuletzt geschrieben wurde“ nicht mehr eindeutig definiert. In [17] hat L. Lamport für dieses Problem mit der Definition der Sequential Consistency ein neues Konsistenzmodell vorgestellt. Der Originaltext lautet:

> Def. „Sequential Consistency“: A System is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of of each individual processor appear in this sequence in the order specified by its program.

In Abbildung 5 ist das Multiprozessorsystem dem hypothetischen Einprozessorsystem gegenübergestellt: Betrachtet man nur Instruktionen, die auf demselben Prozessor ausgeführt werden, so ist zwischen diesen Instruktionen selbstverständlich eine Ordnung definiert. Die Instruktionen werden nun alle auf

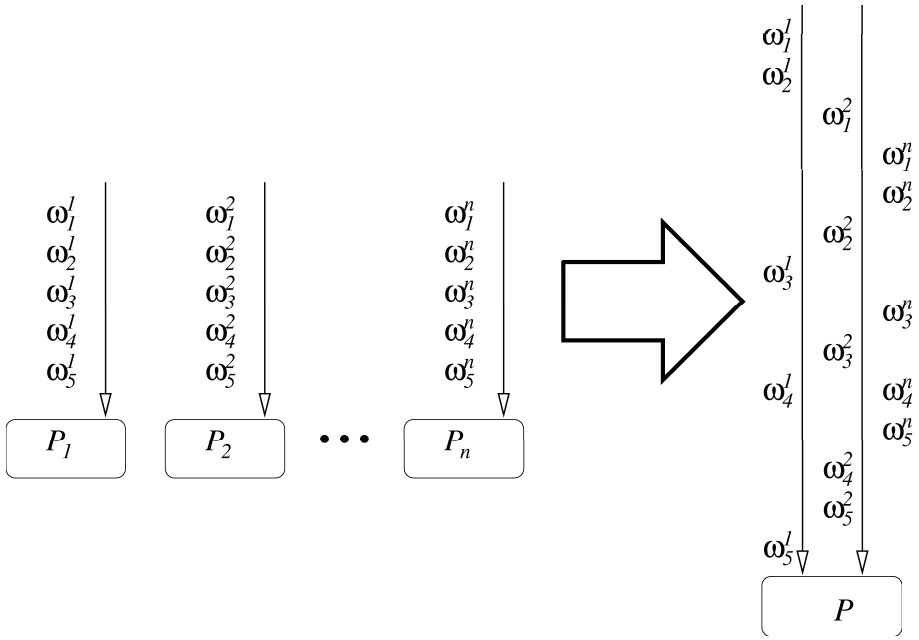


Abbildung 5: Zur Definition der Sequential Consistency. Links das (reale) Multiprozessorsystem, rechts das hypothetische Einprozessorsystem.

dem hypothetischen Einprozessorsystem ausgeführt, so daß diese Ordnung erhalten bleibt. Über eine zeitliche Beziehung zwischen Instruktionen, die auf verschiedenen Prozessoren ausgeführt wurden, wird weiterhin keine Aussage gemacht.

Ist nun das Ergebniss der Berechnung, die ein konkretes Programm anstellt, für *alle* Arten, in der die Instruktionen auf das Einprozessorsystem gebracht werden können, gleich, dann bietet das Multiprozessorsystem die Sequential Consistency.

Aber selbst in einem sequentiell konsistenten System ist noch nicht sichergestellt, daß ein Programm bei jedem Durchlauf das gleiche (und hoffentlich korrekte) Ergebniss liefert. In Abbildung 6 ist eine Situation dargestellt, in der das Ergebnis von Faktoren abhängen kann, die außerhalb der Kontrolle des Programmierers liegen. In [13] bezeichnen die Autoren eine solche Situation als einen Acces Conflict, einen Zugriffskonflikt:

> Def. Access Conflict: Ein Access Conflict liegt vor, wenn mindestens zwei verschiedene Prozessoren auf dieselbe Speicherstelle zugreifen, wobei mindestens einer der Prozessoren einen

schreibenden Zugriff ausführt.

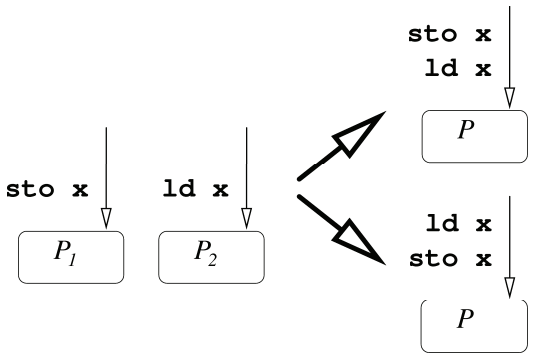


Abbildung 6: Data Race

Nicht jeder Zugriffskonflikt stellt unbedingt einen Programmierfehler dar.

Problematisch sind nur diejenigen Zugriffe, die in keiner Weise synchronisiert werden. Läßt sich ein Zugriffskonflikt im Sinne der Definition der sequentiellen Konsistenz so anordnen, das der schreibende Zugriff einmal vor einem Leser, und bei einem anderen Durchlauf nach dem Leser durchgeführt wird, so liegt ein Data Race vor:

> Def. Data Race: Ein Zugriffskonflikt, bei dem zwischen den beteiligten Prozessoren nicht durch synchronisierende Massnahmen eine eindeutige Zugriffsreihenfolge erzwungen wird, bezeichnet man als Data Race.

Es ist die Aufgabe des Programmierers, solche nicht-synchronisierten Zugriffe zu erkennen und dann zu beseitigen. Mit den in 2 vorgestellten Methoden läßt sich erreichen, daß Zugriffe auf eine Speicherzelle in einer definierten Reihenfolge erfolgen.

Umgangssprachlich läßt sich die Zusicherung, die ein sequentiell konsistentes System dem Programmierer macht, so formulieren: „Jeder synchronisierte Zugriffskonflikt wird entsprechend der Synchronisation korrekt ausgeführt.“ Für die in Abb. 6 dargestellte Situation heisst dies: Wenn der lesende und der schreibende Zugriff durch Synchronisationsmassnahmen in eine definierte Reihenfolge gebracht werden, z. B. so, daß der Schreiber vor dem Leser zugreifen darf, dann wird der Leser genau den Wert lesen, den der Schreiber geschrieben hat. Damit aber *jeder* Leser, der nach einem Schreiber auf ein Speicherzelle zugreift, den Wert liest, den der Schreiber geschrieben hat, muß der Schreibzugriff sofort allen anderen Prozessoren sichtbar gemacht werden. Hätte man Informationen darüber, *wann* der Leser kommt, oder ob überhaupt ein Leser auf die betreffende Speicherzelle zugreifen wird, so könnte die notwendige Kommunikation hinausgezögert oder gar ganz vermieden werden.

3.3 Release Consistency

Im folgenden Abschnitt wird die Release Consistency vorgestellt. Es wird gezeigt, wie man die Synchronisation, die in parallelen Programmen vorhanden ist, dazu ausnutzen kann, um den Kommunikationsaufwand wesentlich zu reduzieren.

In Abschnitt 2 wurden bereits einige Mechanismen vorgestellt, mit denen der Programmierer den Ablauf der einzelnen Kontrollfäden synchronisieren kann. Es wurde auch darauf hingewiesen, daß die Synchronisation über Locks als eine Art „Basis“ für eine große Klasse der verschieden Synchronisationsmechanismen angesehen werden kann. Deshalb wird im folgenden davon davon ausgegangen, daß die Synchronisation über Locks vorgenommen wird.

Betrachtet man einen kritischen Abschnitt, der durch ein Lock geschützt ist, so stellt man fest, daß sich immer nur höchstens ein Prozessor innerhalb des kritischen Abschnitts aufhält. Es kann also nicht vorkommen, das ein anderer Prozessor im gleichen Abschnitt Variablen ändert. Damit ist jedoch noch nicht

sichergestellt, daß diese Variablen nicht auch in einem anderen Abschnitt von einem anderen Prozessor geändert werden. Es empfiehlt sich also, die Locks mit einem Satz globaler Variablen zusammenzufassen, die gemeinsam in kritischen Abschnitten benutzt werden.

Um den Schutz von kritischen Abschnitten auf globale Variable zu verlagern, schlagen die Autoren von [13] folgendes Idiom vor:

```
typedef struct
{
  foo_t foo;
  lock_t lock;
}
shared_foo_t;

: : : :
shared_foo_t *shared_foo;

aquire (shared_foo->lock);
/*
 * access shared_foo->foo;
 */
release (shared_foo_lock);
: : : :
```

Für eine zu schützende Variable `foo` vom Typ `foo_t` wird ein Verbund `shared_foo` vereinbart, der zusammen mit der Variable `foo` auch ein Lock enthält. Dieses Lock kann jeder Prozessor anfordern, bevor er die Variable liest oder schreibt, um es nach dem Zugriff wieder zurückzugeben.

Damit ist sichergestellt, daß ein Prozessor, der sich in einem kritischen Abschnitt aufhält, auf andere Prozessoren keinerlei Rücksicht zu nehmen braucht. Insbesondere braucht er die Änderungen, die er am gemeinsam benutzten Speicher vornimmt, erst für die anderen Prozessoren sichtbar zu machen, wenn er den kritischen Abschnitt verlässt.

Bei Verwendung der Write-Invalidate-Methode muß der Daten-Abgleich also nur erfolgen, wenn ein Prozessor eine `release`-Operation ausführt.

Im Gegensatz zu sequentiell konsistenten Systemen müssen bei Systemen, die die Release Consistency bieten, nur folgende Bedingungen erfüllt sein [13]:

Def. Release Consistency:

- Sync-Operationen sind `acquire`- und `release`-Operationen.
- Bevor ein `load` oder `store` eines Prozessors für einen anderen Prozessor sichtbar wird, müssen alle vorangegangenen `acquire`-Operationen durchgeführt worden sein.
- Bevor ein `release` durchgeführt wird, müssen alle vorangegangenen `load`- und `store`-Operationen für alle anderen Prozessoren sichtbar gemacht worden sein.
- Die Sync-Operationen aller Prozessoren sind untereinander sequentiell konsistent.

3.3.1 Eager Release Consistency

Das Eager Release Consistency-Protokoll (ERC) nutzt die oben beschriebenen Vorteile, und verlangt, daß Änderungen an gemeinsam verwendeten Variablen immer dann an die restlichen Prozessoren gemeldet werden, wenn der Prozessor einen kritischen Abschnitt verlässt, d. h. die `release`-Operation ausführt. Im Gegensatz zum unten beschriebenen LRC-Protokoll ist es eine geradlinige, „naive“ Implementation des RC-Protokolls.

In Abbildung 7 ist ein Ablauf skizziert, in dem drei Prozessoren parallel laufen. Zuerst fordert Prozessor p_1 das Lock L an, modifiziert die Variable x und gibt das Lock wieder zurück. Später fordert Prozessor

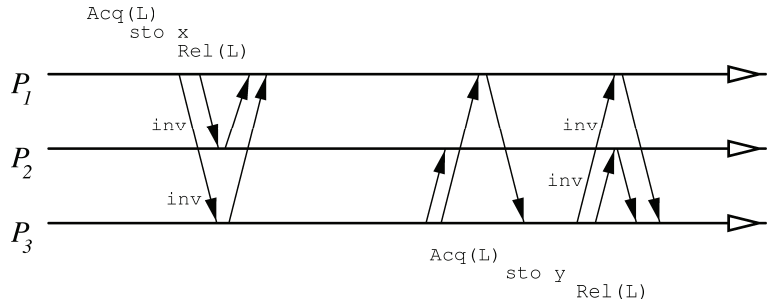


Abbildung 7: Zeitlicher Ablauf der Kommunikation beim ERC-Protokoll: Prozessor p_1 fordert das Lock L an und modifiziert die Variable x. Bei der Freigabe des Locks sendet er Write-Notices an die anderen Prozessoren. Analog geht Prozessor p_3 vor.

p_3 daßelbe Lock an, modifiziert die Variable y und stellt seinerseits das Lock zur Verfügung. Mit der Rückgabe des Locks muß der Prozessor die Seiten, die er modifiziert hat, während er das Lock besass, an die anderen Prozessoren melden.

Der folgende Abschnitt skizziert die Implementation des ERC-Protokolls in einem n-Prozessor-System.

Wenn ein Prozessor ein `acquire` ausführt, muß er sich darüber informieren, ob das Lock von einem anderen Prozessor gehalten wird. Dazu verschickt er einen Rundruf (Broadcast) an alle Prozessoren, in dem er den anderen Prozessoren mitteilt, welches Lock er anfordert. Bevor er das Lock nicht bekommen hat, kann er selbstverständlich nicht weiterarbeiten, sondern muß auf eine Antwort warten.

Ist zu diesem Zeitpunkt ein anderer Prozessor in Besitz des Locks, kann dieser entweder den das `acquire` ausführenden Prozessor davon informieren, das das Lock nicht verfügbar ist, oder er kann sich mit Eingang der Nachricht „merken“, wer das Lock anfordert, um ihn mit Ausführung des eigenen `release` vom Freiwerden des Locks zu benachrichtigen. Die Nachricht, mit der ein Prozessor ein Lock freigibt, bezeichnet man als die Lock Grant Message.

Hat jedoch der Prozessor, der das Lock als Letzter besaß, dieses bereits über eine `release`-Operation wieder zur Verfügung gestellt, so kann er auf den Broadcast sofort mit der Lock Grant Message reagieren.

Selbstverständlich könnte ein Prozessor bei Rückgabe eines Locks alle anderen Prozessoren von den Änderungen informieren, die er vorgenommen hat. In der Praxis zeigt sich allerdings, daß meistens nur wenige Prozessoren sich eine Seite „teilen“. Deshalb ist es lohnend, nur *die* Prozessoren zu informieren, die ebenfalls eine Kopie der modifizierten Seiten haben.

Dazu unterhält jeder Prozessoren für jede Seite des globalen Speichers, die er im lokalen Speicher hat, ein Copyset, ein Bitfeld mit (mindestens) n Bits, in dem festgehalten wird, welche andere Workstation ebenfalls eine lokale Kopie von dieser Seite hat. Führt die Workstation ein `release` aus, so sendet sie die Write-Notices an alle anderen Workstations, deren Bit im Copyset der modifizierten Seite gesetzt ist.

Da die Informationen im Copyset einer Workstation veraltet sein können, sendet jede Workstation, die eine Write-Notice erhält, ihr eigenes Copyset zurück, das vom Absender mit dem eigenen Copyset verglichen wird. Stellt der Sender fest, daß im Copyset des Empfängers Workstations vermerkt sind, die nicht im eigenen Copyset markiert sind, so werden diese in einem zweiten Durchlauf von den Modifikationen unterrichtet. Erfährt der Sender dabei von weiteren Workstations, so initiiert er noch einen weiteren Durchlauf, anderenfalls bricht das Verfahren ab. Laut [12] zeigt sich in der Praxis, das selten mehr als nur ein Durchlauf nötig ist.

3.3.2 Lazy Release Consistency

In Abb. 7 fällt auf, daß sowohl Prozessor p_1 als auch p_3 nach Freigabe des Locks an jeweils *alle* anderen Prozessoren Write-Notices verschicken, obwohl immer nur *ein* Prozessor als Nächster den kritischen Abschnitt betritt. Die zusätzlichen Nachrichten sind also eigentlich überflüssig. Um von dieser Beobachtung zu profitieren, wird die Lazy Release Consistency wie folgt definiert:

Def. „Lazy Release Consistency“:

- Bevor ein `load`- oder `store`-Befehl für *einen* anderen Prozessor sichtbar wird, müssen alle vorangegangenen `acquire`-Operationen *bezüglich dieses Prozessors* durchgeführt worden sein.
- Bevor ein `release` bezüglich irgeneines anderen Prozessors durchgeführt wird, müssen alle vorangegangenen `load`- und `store`-Operationen durchgeführt worden sein.
- Sync-Operationen sind untereinander sequentiell konsistent.

Das Lazy Release Consistency-Protokoll (LRC) nutzt diese Beobachtung, um die Zahl der versandten Nachrichten weiter zu reduzieren. Es verlangt nicht mehr, das ein Prozessor mit der Durchführung der `release`-Operation alle anderen Prozessoren benachrichtigt. Vielmehr muß ein Prozessor, der ein Lock anfordert, bei dieser `acquire`-Operation die Änderungen anfordern, die andere Prozessoren durchgeführt haben, während sie das gleiche Lock besassen. Die Kommunikation illustriert Abbildung 8.

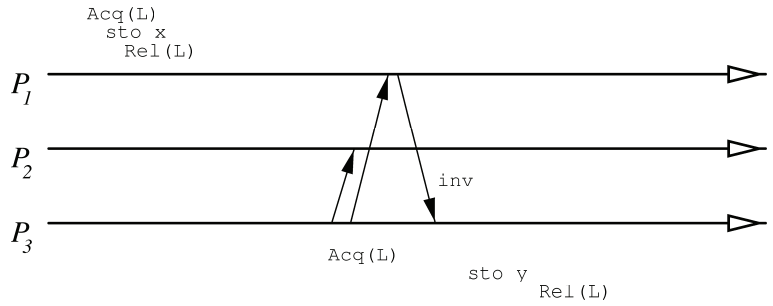


Abbildung 8: Zeitlicher Ablauf der Kommunikation beim Lazy Release Consistency-Protokoll. Im Gegensatz zu Abb. 7 reduziert sich die Anzahl der Nachrichten, die verschickt werden müssen, erheblich.

Verlässt ein Prozessor p einen kritischen Abschnitt, so speichert er nur ab, welche Seiten verändert wurden, während er den Abschnitt abgearbeitet hat. Zusätzlich vermerkt er, *welches* Lock zu dem kritischen Abschnitt gehört.

Wenn ein weiterer Prozessor q dieses Lock in einem Broadcast anfordert, so beantwortet p dies mit der Lock Grant Message, an die er gleich die Write-Notices für die modifizierten Seiten anhängt.

Nach dem Eintreffen dieser Nachricht schreibt Prozessor q die in den Write-Notices erwähnten Seiten ungültig, und beginnt mit der Abarbeitung des kritischen Abschnittes.

Die Vorteile, die das LRC-Protokoll hat, sieht man deutlich, wenn man Abb. 7 mit Abb. 8 vergleicht: Sind in Abb. 7 noch elf Nachrichten nötig, so kommt das LRC-Protokoll in Abb. 8 mit nur drei Nachrichten aus.

3.3.3 Lazy Release Hybrid

In [12] stellen die Autoren ein weiteres Protokoll vor, das im wesentlichen auf dem Lazy Release-Protokoll beruht, aber nicht ausschließlich mit Write-Invalidate arbeitet, sondern zusätzlich eine update-Strategie verwendet, um zeitliche Lokalität im Programmablauf auszunutzen. Dabei wird davon ausgegangen, daß ein Prozessor nach Eintritt in einen kritischen Abschnitt immer die gleichen Seiten anfordert.

Fordert ein Prozessor q ein Lock L an, das ein Prozessor p als Letzter freigegeben hat, werden von p mit der Lock Grant Message nicht einfach Write-Notices für *alle* Seiten verschickt; stattdessen verschickt p für die Seiten, die q das letzte Mal angefordert hat, während er den kritischen Abschnitt abgearbeitet hatte, statt Write-Notices gleich die Diffs.

Um die Seiten zu ermitteln, für die nicht erst eine Write-Notice, sondern sofort ein Diff verschickt wird, hat sich folgende Heuristik besonders bewährt: Für eine Seite wird ein Diffs genau dann verschickt, wenn der Prozessor, der das Lock anfordert, im Copyset des Prozessors vermerkt ist, der das Lock als letztes hielt.

Programme, bei denen die Menge der Seiten, die verschiedene Prozessoren über den Programmablauf benutzen, im wesentlichen konstant ist, profitieren von diesem Protokoll, während Programme, bei denen

sich diese Menge dynamisch ändert, keinen Nutzen daraus ziehen oder sogar schlechter laufen als mit dem LRC-Protokoll.

3.4 Bewertung

Jedes der drei vorgestellten Protokolle hat seine Vor- und Nachteile.

Das ERC-Protokoll zeichnet sich dadurch aus, das es sehr einfach zu implementieren ist. Es ist damit auch für eine Implementierung in Hardware geeignet. Da aber TreadMarks in Software implementiert ist, fällt dieser Vorteil nicht sehr ins Gewicht. Wichtiger ist, das im Gegensatz zu den anderen Protokollen bei jedem `acquire` bereits die Write-Notice vorliegen. Nachteilig macht sich allerdings bemerkbar, das sich viele Nachrichten, die verschickt werden, im Nachhinein als überflüssig herausstellen. Die Latenzzeit, die das Netzwerk beansprucht, um eine Nachricht zu verschicken, ist dann allerdings schon verbraucht, und für die eigentlichen Berechnungen verloren.

Das LRC-Protokoll vermeidet einen beachtlichen Teil der Nachrichten, die sich bei ERC als überflüssig herausstellen würden, ohne wesenliche Nachteile einzubringen, die sich in Bezug auf die Laufzeit des Programms negativ auswirken. Die aufwendigere Implementierung läßt sich in einem Software-System wie TreadMarks bewerkstelligen, ist jedoch für eine Implementierung in Hardware nicht geeignet. Das eine solche Implementierung durchaus Vorteile gegenüber bisher existierenden Hardware-DSM-Systemen bringen würde, zeigt eine Simulation, die in [14] beschrieben ist.

Es läßt sich nicht allgemein sagen, daß das LRH-Protokoll grundsätzlich besser als das LRC-Protokoll ist. Liegt in den Zugriffsmustern des Programmes eine große zeitliche Lokalität vor, wie LRH annimt, dann ergibt sich eine Beschleunigung; ist die Annahme einer zeitliche Lokalität nicht gerechtfertigt, so ist LRC schneller. Die Entscheidung zwischen LRC und LRH muß also von Anwendung zu Anwendung immer von neuem getroffen werden, um für jedes Programm das passende Protokoll auszuwählen.

In ihrer Untersuchung [12] geben die Autoren einen Überblick über das Verhalten verschiedener Benchmarks (siehe Abschnitt 3.5). Die Beschleunigungen, die auf einem 8-Prozessor-System erreicht werden konnten, zeigt Abbildung 9. Es wird deutlich, das für die meisten Applikationen das LRC-Protokoll bessere Laufzeiten als das ERC erzielt, während der Geschwindigkeitsgewinn, der aus der Verwendung von LRH erzielt wurde, zwar noch deutlich sichtbar, aber dennoch geringer ist. Am schlechtesten verhalten sich hier die Applikationen, für die sich keine an das Software-System angepasste Parallelisierung finden läßt, die FFT (Fast Fourier Transform) und Barnes, eine n-Körper-Simulation. Einen kurzen Überblick über die anderen Benchmarks, die in Abb. 9 aufgeführt sind, findet sich in Abschnitt 3.5.

Weiterhin wurde für fünf der Benchmarks der Programmablauf analysiert. Für jedes Programm wurden die Anteile an der Laufzeit ermittelt, die für TreadMarks aufgewendet wurden, und so dem Programm nicht mehr zur Berechnung zur Verfügung stehen.

Abbildung 10 zeigt die Anteile, die TreadMarks für die Verwaltung und Konsistenzerhaltung des verteilten Speichers aufwenden muß. Man erkennt, daß die Werte mit einer Ausnahme deutlich unter 3% der Gesamtzeit liegen, daß also TreadMarks an der gesamten Laufzeit nur einen geringen Anteil hat.

In Abbildung 11 werden die Anteile dargestellt, die TreadMarks mit Kommunikation und Speicherorganisation verbringt. Mit Ausnahme der Water-Simulation liegen die Werte mit $\approx 6\%$ auch hier in einem akzeptablen Bereich.

Aus Sicht der Anwendungsprogramme betrachtet Abbildung 12 die Ausführungszeit. Sie stellt dar, wieviel Zeit einem Kontrollfaden durchschnittlich durch das Betriebssystem verloren geht (Unix), welchen Anteil TreadMarks ausmacht (TreadMarks), und wieviel Zeit der Kontrollfaden infolge von Synchronisationsoperationen warten muß (Idle Time). Der „Rest“ (Computation) bleibt für die Berechnung übrig.

3.5 Benchmarks

Um die Leistungsfähigkeit von TreadMarks zu beurteilen, bedienen sich die Autoren von [12] einer Benchmark-Suite aus acht Benchmarks, die durchgehend Applikationen oder Kernels von Applikationen sind, die in der Praxis verwendet werden. Im folgenden werden die einzelnen Benchmarks kurz vorgestellt.

SOR: Dieser Successive Over-Relaxation-Algorithmus verwendet einen einfachen iterativen Relaxations-Algorithmus. Die Eingabe besteht aus einem zweidimensionalen Gitter $A \in \mathbb{R}^{n \times n}$. Die Randwerte

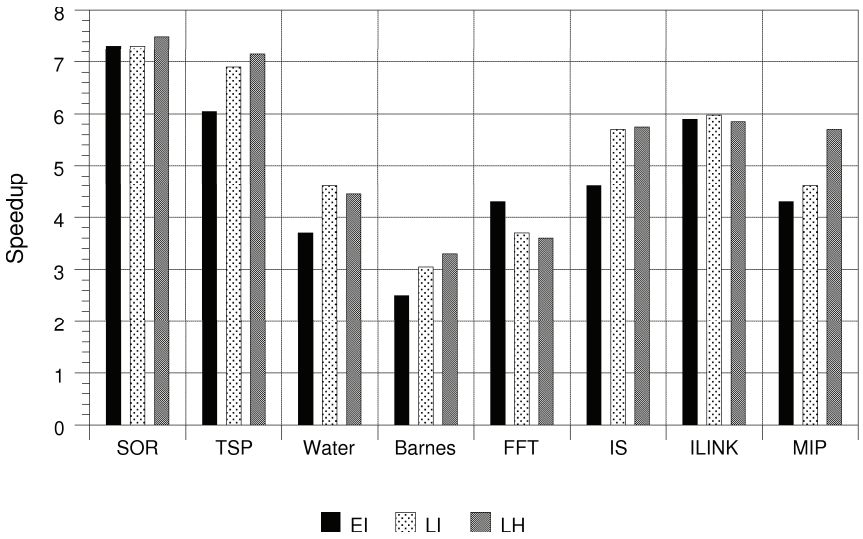


Abbildung 9: Vergleich ERC, LRC und LRH

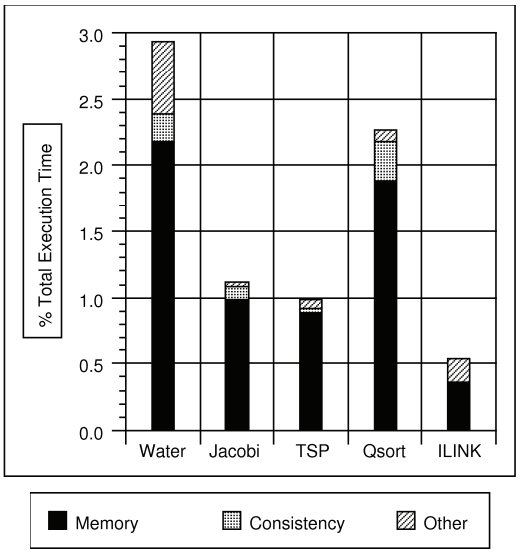


Abbildung 10: Hier wird demonstriert, wieviel Zeit TreadMarks mit der Speicherverwaltung (Memory), der Konsistenzerhaltung (Consistency) und anderen Aufgaben (Other) verbringt.

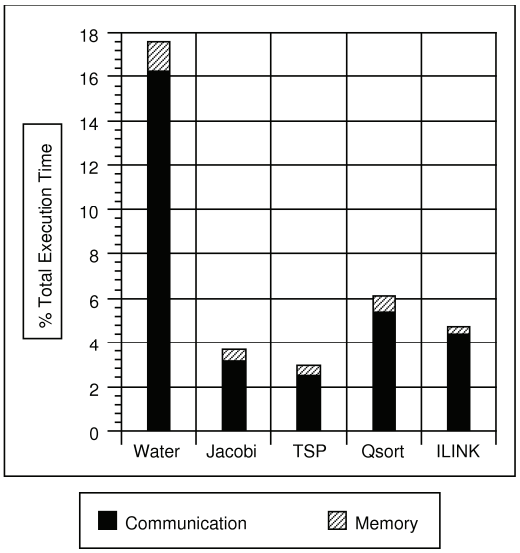


Abbildung 11: Die Abbildung zeigt, welche Anteile der Laufzeit durch Kommunikation (Communication) und Speicherverwaltung (Memory) verloren gehen. Die verbleibende Zeit steht den Anwendungen zur Verfügung.

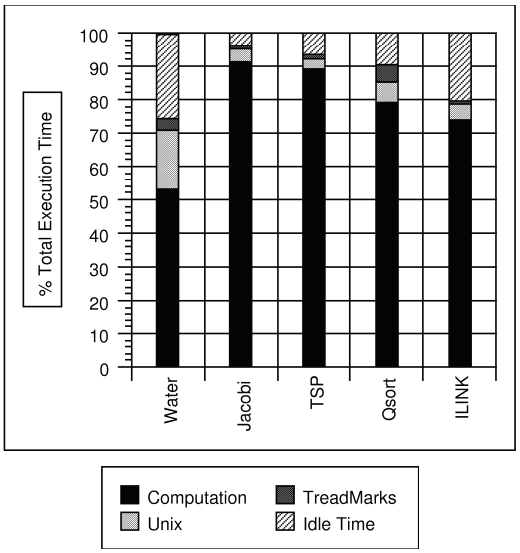


Abbildung 12: In diesem Gesamtüberblick sieht man, wie die verschiedenen Programme ihre Laufzeit ausnutzen können. Computation bezeichnet die Zeit, in der die Programme ihre eigentlichen Berechnungen ausführen, TreadMarks bezeichnet die Zeit, die von TreadMarks zur Kommunikation und Konsistenzerhaltung des globalen Speichers benötigt wird, Unix stellt den Anteil dar, den das Betriebssystem für das Programm arbeitet, und Idle ist der Anteil, den das Programm aufgrund von Synchronisationsoperationen warten muß.

der Matrix, $a_{1,\star}$, $a_{n,\star}$, $a_{\star,1}$ und $a_{\star,n}$ werden mit den Funktionswerten der Randwertfunktion vorbelegt. In jedem Iterationsschritt wird eine neue Martix berechnet, wobei sich mit Ausnahme der Randwerte jedes Element der neuen Matrix ergibt als gewichtetes Mittel seiner „alten“ Nachbarn (oben, unten, links und rechts).

Typische Werte für n sind 1000 oder 2000. Für eine 1024×3072-Matrix betrug die Laufzeit für 50 Iterationen 6.19 – 6.45 s.

TSP: Die Eingabe dieses Algorithmus' besteht aus n Städten, und Informationen darüber, wie weit die Städte voneinander entfernt sind. Der Algorithmus versucht, den kürzesten Pfad zu finden, der alle Städte genau einmal durchläuft und zum Startpunkt zurückkehrt.

In den Simulationen bestand die Eingabe aus $n = 19$ Städten, so das es insgesamt $n! \approx 1.216451 \cdot 10^{17}$ Pfade in Betracht kommen. Durch die Verwendung der Branch-and-Bound-Methode werden allerdings einige Pfade von der Suche ausgeschlossen, ohne das ihre Länge vollständig bekannt zu sein braucht. Außerdem liefert der Branch-and-Bound-Algorithmus nicht unbedingt die optimale, sondern eine (im Allgemeinen aber gute) suboptimale Lösung.

Die Laufzeit betrug 42.61–49.12 s.

Water ist eine Simulation der Dynamik von (Wasser-) Molekülen. In jedem Iterationsschritt werden die intra- und intermolekularen Kräfte benachbarter Moleküle in einer (flachen) Wanne berechnet. Um einen Aufwand der Größenordnung $O(\frac{n^2}{2})$ zu vermeiden, werden für ein Molekül nur solche Moleküle berücksichtigt, die weniger als die halbe Länge der Wanne davon entfernt sind.

Die Simulationen wurden mit $n = 343$ Molekülen über $k = 5$ Schritte durchgeführt. Die Laufzeit betrug 10.63–12.84 s.

Barnes simuliert das Verhalten von n Körpern im $\mathbb{R}^3$ unter dem Einfluss der wechselseitigen Gravitationskräfte der Körper. Durch die Organisation der Körper in einem Baum wird der Aufwand von $O(n^2)$ auf $O(n \log n)$ gedrückt.

Die Simulationen wurden mit $n = 4096$ Körpern durchgeführt. Die Laufzeit betrug 20.54–27.91 s.

FFT: Hier wird eine partielle Differentialgleichung über eine Vorwärts- und eine Rückwärts-Fourier-Transformation numerisch gelöst. Die Eingabe ist ein 3-dim. Gitter $A \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, das in row-major-Anordnung abgespeichert ist.

Die Elemente von A werden über die erste Dimension auf die Prozessoren verteilt, d. h., für ein $i \in [1\,..\,n_1]$ werden alle Elemente $A[i, \star, \star]$ demselben Prozessor zugeteilt.

Auf den $n_1 \cdot n_2$ Vektoren $\vec{u} \in \mathbb{R}^{n_3}$ wird dann eine 1-D-FFT durchgeführt, dann auf den $n_1 \cdot n_3$ Vektoren $\vec{v} \in \mathbb{R}^{n_2}$, wobei jeder Prozessor alleine arbeiten kann, ohne mit den anderen Prozessoren kommunizieren zu müssen. Erst wenn auf den $n_2 \cdot n_3$ Vektoren $\vec{w} \in \mathbb{R}^{n_1}$ gearbeitet werden muß, müssen Daten von anderen Prozessoren angefordert werden.

Die Simulationen wurden mit einem Gitter $A \in \mathbb{R}^{64 \times 64 \times 32}$ durchgeführt. Die Laufzeit betrug 10.10 – 11.95 s.

IS: Hier besteht die Aufgabe darin, eine unsortierte Sequenz S aus N Schlüsseln in einer Reihenfolge anzuordnen. Die Position in der Reihenfolge eines Schlüssels der Sequenz ist der Index, den der Schlüssel hätte, wenn die Sequenz sortiert wäre. Alle Schlüssel liegen in einem Bereich $[0\,..\,B_{\max}]$. Der verwendete Algorithmus ist als Bucket Sort bekannt.

Die Laufzeit des Algorithmus' ist linear zur Anzahl N der Schlüssel. Der Kommunikationsaufwand ist allerdings proportional zu $B_{\max}$, da für den Bucket Sort-Algorithmus ein Array der Länge $B_{\max}$ zwischen den Prozessoren 'transportiert' werden muß.

Die Simulationen wurden mit $N = 2^{20}$ und $B_{\max} = 2^7$ durchgeführt. Die Laufzeit betrug 1.73–2.19 s.

ILINK führt eine Genetic Linkage-Statistik durch, die über den Familienstammbaum eines Lebewesens versucht, die Gene den Vorfahren zuzuordnen. Insbesondere erhofft man sich hier Aufschluß über die Herkunft defekter Gene oder solcher Gene, die für Krankheiten, insb. Seuchen, verantwortlich sind.

Die Laufzeit betrug 1021.4–1030.8 s.

MIP: Die Technik des linear programming wird oft im Operational Resarch eingesetzt, um operationale, seltener taktische oder strategische Maßnahmen zu planen. Neben der 'klassischen' Aufgabe, der Produktionsplanung, finden sich Anwendungen in der Einsatzplanung von Flugpersonal oder in der Planung von Netzwerken.

Die Optimalität des erstellten Planes (Produktionsplanes) wird durch eine Zielfunktion bewertet; die Nebenbedingungen werden durch weitere lineare (sic!) Ungleichungen beschrieben.

Das entstehende Lineare (Un-) Gleichungssystem läßt sich recht leicht lösen, z. B. durch ein Austauschverfahren, jedoch ist die Lösung, die solch ein Verfahren liefert, nicht unbedingt ganzzahlig ist. Für die Anwendung in der Praxis repräsentieren die Komponenten des Lösungsvektors allerdings Größen, die nur ganzzahlig sein können (z. B. Stückzahlen). Dabei ergibt sich die beste ganzzahlig Lösung nicht unbedingt dadurch, daß man den Lösungsvektor komponentenweise rundet, sondern es kommt (im Prinzip) jeder Vektor in Frage, der durch Aufrunden einiger Komponenten und Abrunden anderer Komponenten entsteht.

Für einen Lösungsvektor $\vec{x} \in \mathbb{R}^n$ müssen also 2^n Kombinationen betrachtet werden, von

$$\begin{array}{ll} (\lfloor x_0 \rfloor, & \lfloor x_1 \rfloor, \ldots, \lfloor x_n \rfloor), \\ (\lceil x_0 \rceil, & \lfloor x_1 \rfloor, \ldots, \lfloor x_n \rfloor), \\ (\lfloor x_0 \rfloor & \lceil x_1 \rceil, \ldots, \lfloor x_n \rfloor), \end{array}$$

bis

$$\begin{array}{ll} (\lceil x_0 \rceil, & \lfloor x_1 \rfloor, \ldots, \lceil x_n \rceil), \\ (\lfloor x_0 \rfloor, & \lceil x_1 \rceil, \ldots, \lceil x_n \rceil), \\ (\lceil x_0 \rceil, & \lceil x_1 \rceil, \ldots, \lceil x_n \rceil). \end{array}$$

Wie bereits bei TSP kann man hier einen Branch-and-Bound-Algorithmus verwenden, der möglichst viele Kombinationen möglichst früh ausschliesst.

Die Laufzeit betrug 19.09–16.10 s.

4 Multiple-Writer-Protokolle

In den bisherigen Abschnitten wurde oftmals gefordert, daß die Workstations untereinander Speicherblöcke abgleichen, an denen eine Workstation Änderungen vorgenommen hatte. Dabei wurde — absichtlich — nicht genauer darauf eingegangen, *wie* diese Änderungen übermittelt werden sollten.

Greifen (mindestens) zwei Prozessoren auf dieselbe Speicherzelle(n) zu, spricht man von True Sharing. Solange zwischen einigen Prozessoren bezüglich eines Speicherbereiches True Sharing herrscht, muß das DSM-System die Daten zwischen den beteiligten Prozessoren abgleichen. Es liegt dann in der Aufgabe des Programmierers, zu untersuchen, ob der Algorithmus so abgeändert werden kann, daß der Anteil an True Sharing verringert wird.

Eine einfache Lösung ist es sicherlich, die Konsistenzeinheiten im Ganzen zu übermitteln. Im Vorgriff auf Abschnitt 5 sei erwähnt, daß die Konsistenzeinheiten in TreadMarks Virtuelle Seiten sind. Die typische Seitengrösse liegt bei 4 kB oder 8 kB.

Ebenfalls im Vorgriff auf Abschnitt 5 sei angesprochen, daß die Hardware-Plattform, auf der TreadMarks läuft, ein Netzwerk von Workstations ist, die durch ein 'normales' Netzwerk, wie Ethernet oder ATM verbunden sind. Auf diesem Netzwerk sind die Kosten pro Nachricht in Bezug auf Zeit- und Rechenaufwand sehr hoch, während die Kosten pro Byte in *einer* Nachricht nicht so sehr ins Gewicht fallen. Es ist also durchaus sinnvoll, wenige, aber dafür umso längere Nachrichten zu versenden.

Probleme bereitet also nicht die Übermittlung der Nachrichten, sondern die Tatsache, das die Konsistenzeinheiten auf die oben erwähnte Größe von 4–8 kB festgelegt sind.

Als Beispiel betrachte man die Paralellisierung des SOR-Verfahrens aus Abb. 18. Die Größe der Matrix dort ist vorgegeben, und damit auch die Größe der 'Streifen', die jeder Prozessor abzuarbeiten hat. Im Speicher könnten nun zwei dieser Streifen so zu liegen kommen, wie dies in Abbildung 13 dargestellt wird. Eine solche Situation, die dadurch charakterisiert wird, daß sich zwei Schreiber zwar nicht die Daten teilen, jedoch die Konsistenzeinheit, in der diese Daten (zufällig) zu liegen kommen, bezeichnet man als False Sharing.

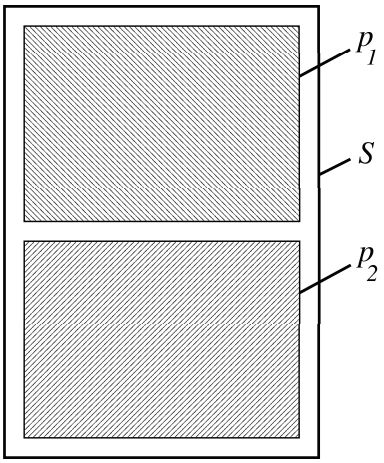


Abbildung 13: False Sharing am Beispiel des Jakobi-SOR-Verfahrens: Zwei der 'Streifen' liegen beide (zumindest teilweise) in der gleichen Konsistenzeinheit.

> Def. False Sharing: Greifen mindestens zwei Prozessoren auf disjunkte Speicherzellen zu, die jedoch in der gleichen Konsistenzeinheit liegen, so bezeichnet man diese Situation als False Sharing.

Wenn die Prozessoren den ihnen zugeteilten Streifen beschreiben, so schreiben sie nicht in die gleichen Regionen. Es liegt hier also keine Data Race-Situation vor, so daß die Zugriffe nicht synchronisiert zu werden brauchen. Nachdem beide Prozessoren den Zugriff durchgeführt haben, sind jedoch zwei 'Versionen' der Seite vorhanden, die nicht ohne weiteres zusammengefügt werden können, da keinerlei Informationen darüber vorliegen, welcher Prozessor welche Stelle der Seite geändert hat.

Im vorliegenden Beispiel könnte man argumentieren, es gebe nur zwei Bereiche, eine 'obere' und eine 'untere' Hälfte, und aus den Schleifengrenzen liesse sich leicht ablesen, wo die eine Hälfte ende und die andere beginne. Dies ist allerdings nur in diesem Beispiel so einfach möglich. Die Informationen darüber, wer welche Speicherzelle einer Seite verändert hat, hängen von einer Vielzahl von Faktoren ab, über die man nicht unbedingt Bescheid weiss; viellmals wäre es auch nicht wünschenswert, so detailliert über die zugrundeliegende Hardware-Plattform, das Betriebssystem und den verwendeten Compiler Bescheid wissen zu müssen.

Die einzige Lösung scheint hier zu sein, die betreffende Seite zwischen jedem Schreibzugriff des einen Benutzers zum 'anderen' Besitzer zu übertragen, so daß dieser in jedem Fall über eine aktuelle Version dieser Seite verfügt, was sicherlich eine sehr aufwendige Methode wäre.

Um dieses Problem zu lösen, wurde in TreadMarks ein Multi-Writer-Protokoll implementiert, mit dem mehrere Schreiber dieselbe Seite modifizieren können, solange sie nicht überlappende Speicherbereiche beschreiben. Es sei angemerkt, das dies einer Data-Race-Situation gleichkäme, wenn die Prozessoren zwischen den Schreibzugriffen synchronisieren.

Um mehreren Schreibern zu ermöglichen, dieselbe zu modifizieren, wird wie folgt vorgegangen: Vor Beginn des Schreibzugriffes wird von der Seite eine Kopie angefertigt, ein Twin. Für die Twins, die gerade

benötigt werden, hält TreadMarks einen eigenen Speicherbereich vor. Der Prozessor modifiziert nur das 'Original' der Seite.

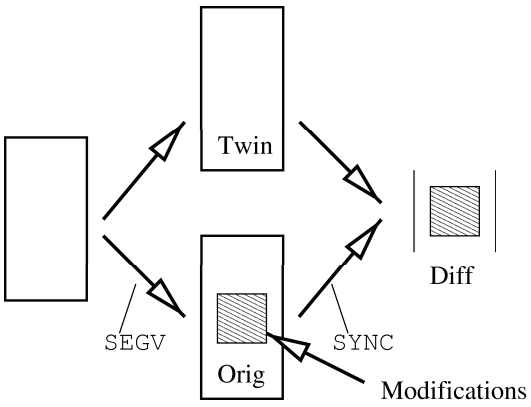


Abbildung 14: Erstellen der Twins und Diffs bei einer Seite, die im lokalen Speicher einer Workstation entweder noch nicht vorhanden war, oder bei der letzten Synchronisations-Operation als ungültig markiert wurde. Beim Schreibzugriff auf diese Seite wird zuerst das Twin angefertigt. Danach kann der Prozessor die Seite selbst modifizieren. Um die Änderungen (Modifications) an andere Prozessoren weiterzugeben, wird das Original mit dem Twin verglichen, und die Unterschiede in einem Diff zusammengefasst.

Um die Änderungen zu anderen Workstations weiterzugeben, werden Original und Twin Byte-weise verglichen, um festzustellen, *welche* Stellen der lokalen Kopie modifiziert wurden, und welche nicht. Diese Modifikationen werden in einem sog. Diff zusammengefasst, das mit einem RLE[14]-Verfahren komprimiert wird.

Das Aktualisieren der lokalen Kopien von ungültigen Seiten geschieht ausschließlich über diesen Mechanismus. Damit sind die zu übertragenden Daten in der Regel merklich kleiner als die Seiten der virtuellen Speicherverwaltung.

5 TreadMarks

TreadMarks bietet ein in Software implementiertes DSM-System. Die Programme, die für TreadMarks erstellt werden, fallen in die SIMD-Klasse, d. h., jeder Prozessor führt dasselbe Programm aus, während die verschiedenen Prozessoren auf unterschiedlichen Daten arbeiten. Dazu steht jedem Programm in einer Variable die Nummer des Prozessors zur Verfügung, aufgrund der sich der Prozessor entscheiden kann, *welche* Daten er zu bearbeiten hat.

Im Gegensatz zu den weiter oben angesprochenen Programmierumgebungen läßt sich TreadMarks auf Workstation-Clustern betreiben. Damit entfallen für den Anwender die Anschaffungskosten, der Aufwand und die Kosten für Wartung und Betrieb. Statt dessen kann die Hardware genutzt werden, die an vielen Stellen, in denen EDV eingesetzt wird, sowieso vorhanden ist. Es sind auch keine zusätzlichen Anschaffungen für Hardware nötig. Auch Eingriffe in die Softwarekonfiguration sind überflüssig, so das die Workstations weiterhin zum 'normalen' Gebrauch zur Verfügung stehen. Die Autoren von [11] haben ihre Resultate auf einem ATM-basierten Netzwerk aus acht DECstation-5000/240s gewonnen; in [23] werden mit unter anderem DEC AXPs, IBM RS/6000, Intel-n86 und Sun SPARC die wichtigsten auf dem Markt vertretenen Workstations als mögliche Plattformen für TreadMarks genannt. Weitere Arbeiten verwenden eine Portierung auf die IBM SP/2 mit acht Knoten.

[14] Run-Length Encoding

5.1 Hardwarekonfiguration

Wie schon erwähnt, läuft TreadMarks auf einem Netzwerk aus Workstations. Dieses Setup wird in Abbildung 15 skizziert. Das Netzwerk, das die Workstations verbindet, kann entweder Ethernet sein, oder

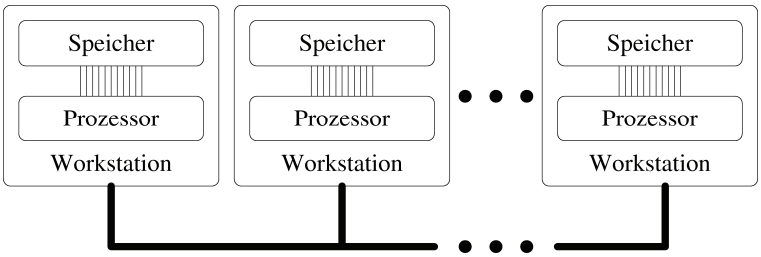


Abbildung 15: Hardware-Setup zur Verwendung von TreadMarks. Zur Verwendung kommen hier handelsübliche High-End-Workstations, wie DEC AXPs oder Sun SPARCs.

ein ATM-Netzwerk. In [13] werden Resultate aus Experimenten sowohl mit ATM- und 10Mbit-Ethernet- als auch mit 1Gbit-Ethernet-Netzwerken gezeigt, die den Einfluss auf die Rechenleistung zeigen; bis auf wenige Ausnahmen sind die Gewinne, die man durch Einsatz der schnelleren Netzwerke erzielen kann, allerdings gering, so daß man also mit den weit verbreiteten Ethernet-Netzwerken akzeptable Rechenzeiten erwarten kann.

5.2 Das Programmiermodell von TreadMarks

Wie schon mehrfach erwähnt, bietet TreadMarks dem Programmierer ein Distributed Shared Memory System, wie es in Abbildung 16 skizziert ist. Der globale Speicher besteht physikalisch aus mehreren

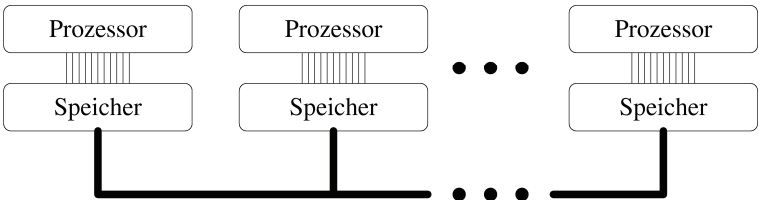


Abbildung 16: Das Programmiermodell eines Distributed Shared Memory Systems: Mehrere Prozessoren arbeiten auf einem globalen Speicher, der physikalisch aus mehreren lokalen Speichern besteht.

lokalen Speichern; typischerweise ist bei jedem Prozessor eine physikalische Speichereinheit vorhanden. Die Aufgabe der DSM-Umgebung besteht darin, dem Programmierer einen virtuellen, globalen Speicher zur Verfügung zu stellen, auf den jeder Prozessor beliebig zugreifen kann. Die Mechanismen, die diesen virtuellen globalen Speicher entstehen lassen, bleiben dem Programmierer dabei verborgen.

Nichtsdestotrotz muß sich der Programmierer darüber im klaren sein, in welchem Masse das System welche Ebenen der Parallelität unterstützt. Der Zugriff auf eine Speicherzelle, die sich im lokalen Speicher des zugreifenden Prozessors befindet, geschieht zwar genauso schnell wie bei einem Einprozeßorsystem; ist die Speicherzelle jedoch nur in einem entfernten lokalen Speicher vorhanden, so muß damit gerechnet werden, daß sie erst über das Netzwerk in den lokalen Speicher des betreffenden Prozessors transportiert werden muß. Die dafür benötigte Zeit wirkt sich zwar bei in Hardware implementierten Verbindungsnetzwerken nicht allzu tragisch aus, kann aber bei den hier verwendeten Netzwerken signifikant zur Zugriffszeit beitragen.

Unter Berücksichtigung dieser Tatsache kommen die unteren, feinkörnigen Ebenen der Parallelität nicht in Frage. Die ‘unterste’ Ebene, die hier verwendet werden kann, ist die Blockebene. Man versucht dabei, die gesamte Aufgabe des Programms in möglichst gleichgroße Teilaufgaben zu unterteilen, so daß

jeder Prozessor eine Teilaufgabe in seinem lokalen Speicher abarbeiten kann, ohne auf den lokalen Speicher der anderen Prozessoren zuzugreifen.

Die Verteilung der Arbeit geschieht in TreadMarks so, daß auf jeder Workstation ein Prozeß läuft, der einen Teil der gesamten Arbeit erledigt. Im Prinzip läuft jeder Prozeß nach dem gleichen Programmcode ab, wobei allerdings jedem Prozeß von TreadMarks eine eindeutige Prozeßnummer zugeordnet wird, durch die sich der Prozeß identifizieren kann.

Aus Sicht des Programmierers wird das gesamte Programm in einzelne Kontrollfäden (Treads) unterteilt; aus Sicht einer Workstation und ihrem Betriebssystem ist jeder dieser Kontrollfäden ein separater Prozeß der über das Netzwerk mit anderen Prozessen auf anderen Workstations kommuniziert. Die momentane Implementation von TreadMarks ist nur für den Einsatz auf Uniprozessor-Rechnern gedacht, so daß jeder Teilauftrag bzw. jeder Kontrollfaden von genau einem Prozessor bearbeitet wird. Im folgenden werden dementsprechend die Begriffe *Prozessor*, *Prozeß*, *Kontrollfaden* und *Workstation* synonym verwandt, während der Begriff *Programm* oder *Algorithmus* dem gesamten, parallel abzuarbeitenden Software-System vorbehalten sind.

5.3 Ansätze zur Parallelisierung

Wie eine solche Parallelisierung erreicht werden kann, ist von dem konkreten Algorithmus bzw. Programm abhängig. Es gibt jedoch zwei Betrachtungsweisen, anhand derer sich eine eventuell vorhandene inhärente Parallelität identifizieren läßt.

Eine große Gruppe von Programmen, die hauptsächlich im Gebiet der Numerik Anwendung finden, arbeitet auf einer großen, regelmässigen Datenmenge, z. B. auf Matritzen, und führt auf diesen Daten viele homogene Operationen aus, die jeweils nur einen kleinen Teil der Daten in Betracht ziehen. In diesen Fällen kann es möglich sein, die gesamte Datenmenge in gleich große 'Streifen' zu teilen, und jedem Prozessor einen solchen Streifen zuzuordnen, den er parallel zu den anderen Prozessoren abarbeitet. Diese inhärente Parallelität bezeichnet man als *Datenparallelität*. Ein Beispiel, in dem diese Art der Parallelisierung vorgenommen wird, ist das Beispielprogramm in Abb. 18.

In einer anderen Gruppe von Programmen läßt sich eine große Menge von vergleichsweise kleinen 'Jobs' identifizieren, die alle unabhängig voneinander bearbeitet werden können, ohne daß das Ergebniss eines Jobs für die Bearbeitung eines anderen Jobs von Bedeutung wäre. Ein gutes Beispiel sind solche Probleme, die sich nicht durch eine zielgerichtete Berechnung, sondern nur durch exhaustive search lösen lassen. Liegt eine solche *Kontrollflussparallelität* vor, so lassen sich die Daten, die einen Job beschreiben, in einem geeigneten 'Container', wie z. B. einer Warteschlange anordnen, von der sich jeder Prozessor einen Job abholt, ihn bearbeitet, und dann das Ergebniss speichert. Ein Beispiel, in dem diese Art der Parallelisierung vorgenommen wird, ist das Beispielprogramm in Abb. 20.

5.4 Das TreadMarks API

Um mit dem DSM-System zu kommunizieren, stehen dem Programmierer einige Dienstprimitive zur Verfügung. Diese Primitive dienen zum einen der Initialisierung und Finalisierung des DSM-Systems, zum anderen der Synchronisation der Prozessoren. Zur Initialisierung bzw. Finalisierung bietet TreadMarks die Aufrufe `Tmk_startup ()` bzw. `Tmk_exit ()`. Das Speichermanagement erfolgt über die Funktionen `Tmk_malloc ()` und `Tmk_distribute ()`. Zur Synchronisation bietet TreadMarks die Aufrufe `Tmk_lock_aquire ()`, `Tmk_lock_release ()` und `Tmk_barrier ()`.

Im Einzelnen bedeuten die Primitive folgendes:

`Tmk_startup`: Nach Aufruf der `main`-Funktion läuft zunächst nur ein Prozeßauf der Workstation, auf der das Programm gestartet wurde. Diese Funktion leitet die Phase ein, in der der Prozeßauf die anderen Workstations verteilt wird.

`Tmk_malloc`: Mit dieser Funktion wird verteilter Speicher angefordert, der von TreadMarks verwaltet wird. Änderungen, die ein Prozessor in solchermassen angeforderten Speicher vornimmt, werden 'automatisch' den anderen Prozessoren sichtbar gemacht. Das Programm kann zusätzlich Speicher mit der `malloc`-Funktion anfordern, um ihn lokal zu verwenden.

`Tmk_free` gibt den mittels `Tmk_malloc` angeforderten Speicher wieder frei, analog zur `free`-Funktion der `C`-Bibliothek. Es ist die Aufgabe des Programmierers, dafür zu sorgen, daß nur ein Prozeßden verteilten Speicher freigibt.

`Tmk_sbrk:` Diese Funktion ist das Analogon zur `sbrk`-Funktion. Sie ist nicht für den Gebrauch in Anwendungsprogrammen gedacht. Statt dessen soll es dadurch ermöglicht werden, eine eigene Speicherverwaltung zu implementieren, die auf einen einzelnen Anwendungsfall bzw. auf eine kleine Klasse von Anwendungsfällen spezialisiert sind.

`Tmk_distribute:` Mit dieser Funktion kann ein Prozeß seinen Speicherbereich, den er lokal alloziert hat, in verteilten Speicher umwandeln und ihn den anderen Prozessen zur Verfügung stellen. Diese Funktion löst ein Problem, das oft beim Programmstart auftritt: Wenn ein Prozessor verteilten Speicher anfordert, sorgt zwar TreadMarks dafür, das der Speicher auf *allen* Workstations verfügbar ist; jedoch muß den anderen Prozessoren mitgeteilt werden, wo sie diesen Speicher finden können. Die Lösung besteht darin, daß die Adresse des verteilten Speichers in einer globalen Variable abgelegt wird, deren Adresse wiederum bei allen Prozessoren gleich ist. Durch einen Aufruf der `Tmk_distribute`-Funktion mit der globalen Variable als Argument wird der Wert der Variable, d. h. die Adresse des verteilten Speichers an die anderen Prozessoren übermittelt.

`Tmk_barrier:` Eine Barrier ist eine Stelle im Programm, an der jeder Prozessor warten muß, bis alle anderen Prozessoren diese Stelle auch erreicht haben. Erst danach darf der wartende Prozessor (wie auch alle anderen wartenden Prozessoren) weiterarbeiten. Man macht sich leicht klar, daß es in einem Programm höchstens *eine* Sperre geben muß: Gäbe es auch nur zwei Sperren s_1 und s_2, so könnte ein Prozessor p_1 an s_1 und ein anderer Prozessor p_2 an s_2 warten. Prozessor p_1 kann seine Arbeit nicht fortsetzen, bevor p_2 an Sperre s_1 angekommen ist, während Prozessor p_2 darauf wartet, bis p_1 an s_2 ankommt. Die Prozessoren blockieren sich also gegenseitig. Warten andererseits beide Prozessoren immer an der *gleichen* Sperre, so besteht auch kein Grund, die 'beiden' Sperren voneinander zu unterscheiden.

Trotzdem bietet TreadMarks verschiedene Barriers an, um es dem Software-Entwickler einfacher zu machen, die Software zu testen und auf Fehler zu überprüfen. In einem fehlerhaften Programm kann es durchaus vorkommen, daß zwei Prozessoren zwar an verschiedenen Stellen auf eine Sperre auflaufen, so wie oben geschildert. Kann man in einer solchen Situation die Sperren, an denen die Prozessoren warten, unterscheiden, so kommt man in vielen Fällen dem Fehler einfacher auf die Spur, als wenn man den Programmablauf daraufhin überprüfen müsste, wo die fehlerhafte Synchronisation auftrat.

`Tmk_lock_aquire:` Ein Lock ist ein **??**, das höchstens *ein* Prozessor besitzen darf. Fordert ein Prozessor p_2 mit der Funktion `Tmk_lock_aquire` ein Lock an, das ein anderer Prozessor p_1 bereits angefordert und erhalten hat, so muß er warten, bis p_1 das Lock wieder freigibt. Im Gegensatz zu Barriers sind zur Synchronisation über Locks mehrere Locks nötig.

`Tmk_lock_release` ist das Gegenstück zu `Tmk_lock_aquire`: Damit wird das angeforderte Lock wieder zurückgegeben, so das es ein evtl. bereits wartender Prozessor erhalten kann.

5.5 Beispiele

Für die Erstellung eines Algorithmus', der parallel ausgeführt werden kann, gibt es zwar keine algorithmisch anwendbaren Methoden, die garantieren, das der Algorithmus effizient parallelisiert wird; jedoch kann man sich auf einige Prinzipien stützen, von denen in diesem Abschnitt die zwei wichtigsten anhand jeweils eines Beispiels vorgestellt werden.

In Abbildung 18 und Abbildung 20 sind die beiden Beispiele in einem an `C` angelehnten Pseudocode angegeben.

5.5.1 Datenparallelität

Das Jakobi SOR-Verfahren dient zur Lösung einer Differentialgleichung bzw. eines Systems aus Differentialgleichungen. In einer Iteration wird über eine Iterationsfunktion f eine Folge von Matritzen berechnet, bis die Iteration zum Stillstand kommt.

Hier wird ausgenutzt, das die Iterationsfunktion f nur auf einem kleinen Bereich der Matrix arbeitet. Typischerweise umfassen z. B. die Formeln zur numerischen Lösung selten mehr als einen Bereich der Größe 9×9, während diese Formel über Matritzen der Größeordung 1000×1000 angewandt wird. Die Daten, auf denen der Algorithmus arbeitet, (im Beispiel die Matrix) wird auf die Prozessoren aufgeteilt, so daß jeder Prozessor einen kleinen Bereich unabhängig von den anderen Prozessoren bearbeiten kann.

Das Jakobi-Verfahren iteriert über einem zweidimensionalen Gitter. In jedem Iterationsschritt wird das Gitter durch ein neues Gitter ersetzt, wobei sich jedes neue Element als Mittelwert der 'alten' Nachbarelemente ergibt. Nach Vorgabe einer Startmatrix $A^{(0)} \in \mathbb{R}^{N \times N}$ berechnet das Jakobi-SOR-Verfahren also eine Folge $(A^{(k)})_{k=1}^{\infty}$, wobei gilt:

$$a_{ij}^{(k+1)} = \begin{cases} \frac{1}{4}\left(a_{i+1,j}^{(k)} + a_{i-1,j}^{(k)} + a_{i,j-1}^{(k)} + a_{i,j+1}^{(k)}\right), & \text{für } i,j \in [2\,..\,N-1], \\ a_{ij}^{(0)} & \text{sonst.} \end{cases}$$

Zur Implementierung verwendet man zweckmässigerweise einen temporären Speicherbereich, in den man die neu berechnete Matrix einträgt; in einem zweiten Schritt werden die Daten in das ursprüngliche Gitter zurückkopiert.

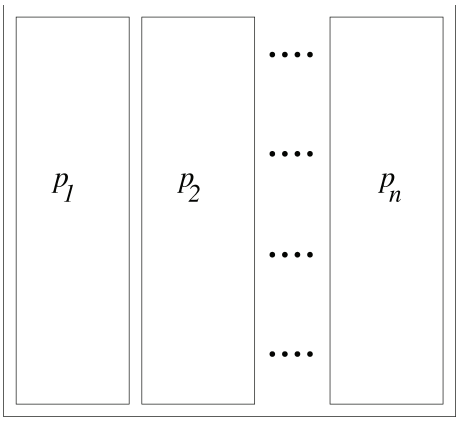


Abbildung 17: Matrix beim parallelisierten Jakobi-SOR-Verfahren

Die Parallelisierung erfolgt so, daß die Matrix in soviele etwa gleichgroße 'Streifen' geteilt wird, wie Prozessoren vorhanden sind. Jedem der Prozessoren wird einer der Streifen zugeteilt, wie es in Abbildung 17 verdeutlicht wird. Den temporären Speicherbereich, im Beispiel das `scratch`-Array, legt jeder Prozessor im lokalen, privaten Speicher an; die Matrix selber (`grid`) liegt in dem verteilten, globalen Speicher. Abbildung 19 zeigt den Speedup, den man auf diese Art erreichen kann, im Vergleich zu einer Implementation desselben Algorithmus' in PVM. Man erkennt, daß die TreadMarks-Version der PVM-Implementation knapp überlegen ist.

5.5.2 Kontrollflussparallelität

Ziel des Travelling-Salesman-Algorithmus ist es, den kürzesten Pfad zu finden, der durch eine vorgegebene Menge von Städten führt. Jede Stadt soll dabei genau einmal betreten werden, mit Ausnahme derjenigen Stadt, in der die Rundreise beginnt und auch endet. Im Beispiel wird ein einfacher Branch-and-Bound-Algorithmus benutzt.

```
#define M 1024
#define N 1024

float **grid; /* shared array */
float scratch [N][M]; /* private array */
int main (int argc, char **argv)
{
    Tmk_startup ();

    if (Tmk_proc_id == 0)
        {
            grid = Tmk_malloc (N * M * sizeof (float));
            /* initialize grid */
        }

    Tmk_barrier (0);

    int length = M / Tmk_nprocs;
    int begin = length * Tmk_proc_id;
    int end = length * (Tmk_proc_id+1);

    for (/* number of iterations */)
        {
            for (int i = begin; i < end; i++)
              for (int j = 0; j < N; j++)
                scratch [i][j] = 0.25 *
                     (grid[i+1][j] + grid[i-1][j]
                   + grid[i][j+1] + grid[i][j-1]);

            Tmk_barrier (0);

            for (int i = begin; i = end; i++)
              for (int j = 0; j < N; j++)
                grid [i][j] = scratch [i][j];

            Tmk_barrier (1);
        }
}
```

Abbildung 18: Jakobi SOR-Verfahren

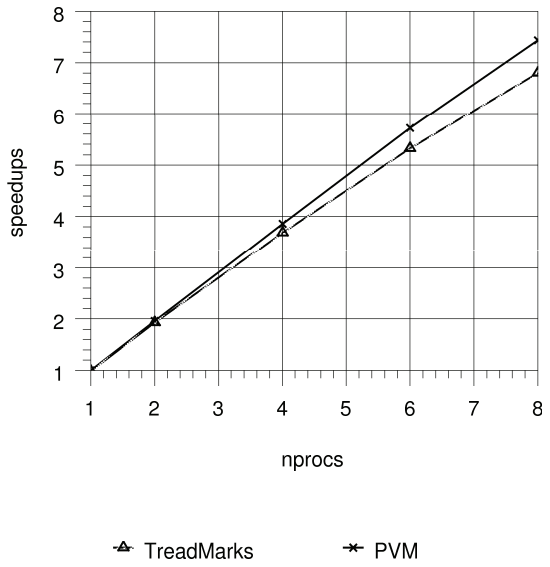


Abbildung 19: Speedup beim Jakobi SOR-Verfahren

Bei der Parallelisierung des Travelling-Salesman-Algorithmus' wird ausgenutzt, daß eine große Menge an vergleichsweise kleinen Jobs auszuwerten ist, bevor das Ergebnis feststeht. Für diese Jobs wird eine Warteschlange unterhalten, aus der sich jeder Prozessor einen Job abholt und auswertet. Danach wird das Ergebnis bekannt gemacht, indem zum Beispiel globale Variable neu gesetzt, oder vielleicht neue Jobs in die Warteschlange eingereiht werden.

Die Branch-and-Bound-Methode dient in der Kombinatorik, insbesondere auf dem Gebiet des Operational Resarch, der Lösung rechenaufwendiger Probleme. Es ähnelt dem Divide-and-Conquer-Verfahren; jedoch verursacht es in den meisten Fällen einen geringeren Rechenaufwand, so das ein Branch-and-Bound-Algorithmus noch praktikabel ist, selbst wenn ein Divide-and-Conquer-Algorithmus zu Speicher- oder rechenaufwendig ist.

Gegeben sei ein Problem P. Für dieses Problem existiert eine evtl. sehr große Menge $\mathbb{X} = \mathbb{X}(P)$ an sog. zulässigen Lösungen. Für jede zulässige Lösung $X \in \mathbb{X}$ ist eine Bewertungsfunktion $F : X \mapsto F(X)$ definiert. Ziel ist es, diejenige Lösung $X^\star$ zu finden, die $F(X^\star)$ maximiert. Diese Lösung $X^\star(P)$ nennt man die optimale Lösung.

Durch die große Anzahl an zulässigen Lösungen $X \in \mathbb{X}$ ist es meist nicht realistisch, die optimale Lösung durch eine erschöpfende Suche zu ermitteln. Man ist daher bestrebt, möglichst große Teilmengen $\mathbb{X}' \subset \mathbb{X}$ von der Suche auszuschliessen.

Wie bei den Divide-and-Conquer-Algorithmen unterteilt man zunächst das Problem P in Teilprobleme $P_1 \,..\, P_k$, so daß

$$X(P) = \bigcup_{i=1}^{k} X(P_i), \text{ und möglichst}$$

$$\bigcap_{i=1}^{k} X(P_i) = \emptyset,$$

daß also mit der Gesamtheit der Lösungen der Teilprobleme auch die Lösung des Ausgangsproblemes bekannt ist, wobei sich die Teilprobleme möglichst nicht oder nur wenig überschneiden. Diese zweite Bedingung stellt sicher, daß bei der Berechnung der Lösung nicht zuviel Arbeit mehrfach getan wird.

Im Gegensatz zu den Divide-and-Conquer-Algorithmen wird jetzt nicht jedes Teilproblem weiter zerteilt, bis die Teilprobleme zu Trivialfällen degeneriert sind. Wie oben schon erwähnt, würde bei einem solchen Vorgehen die Anzahl der Teilprobleme die Kapazität des Rechners schnell übersteigen.

Vielmehr wird durch das sog. Bounding versucht, möglichst viele Teilprobleme daraufhin zu untersuchen, ob sie überhaupt einen Anteil zu Gesamtlösung beitragen können, oder ob sie nicht weiter zu untersucht werden brauchen.

Es läßt sich stets eine untere Schranke $\underline{F}$ für den Zielfunktionswert des Ausgangsproblemes angeben, den die optimale Lösung nicht unterschreitet. Zu Beginn des Algorithmus' kann man entweder $\underline{F} := -\infty$ setzen, oder durch eine geeignete Heuristik einen sinnvollen Anfangswert finden. Für jedes Teilproblem $P_i,\ i \leftarrow [1\,..\,k]$ wiederum läßt sich eine Obergrenze $\overline{F_i}$ angeben.

Diese Schranken kann man in vielen Fällen über eine Relaxation der Bewertungsfunktion berechnen, indem man z. B. Nebenbedingungen wegläßt, oder sich von exakten Berechnungen auf Abschätzungen zurückzieht.

Ein Teilproblem wird nur dann weiter verzweigt, wenn es nicht ausgelotet werden kann. Ein Problem bezeichnet man als ausgelotet, wenn eine der folgenden drei Bedingungen gilt:

1. Es gilt $\overline{F_i} \leq \underline{F}$: Die optimale Lösung des Teilproblems P_i kann nicht besser als die beste bisher bekannte zulässige Lösung sein. Damit braucht man Problem P_i nicht weiter zu betrachten.

2. Es gilt $\overline{F_i} > \underline{F}$: Das Teilproblem P_i ist eine neue beste zulässige Lösung oder kann zu einer solchen beitragen. Deshalb wird $X(P_i)$ gespeichert, und man setzt $\underline{F} := \overline{F_i}$.

3. Es gilt $X(P_i) = \emptyset$, das heisst, P_i besitzt keine zulässigen Lösungen und braucht weiter nicht betrachtet zu werden.

Der Algorithmus verwendet eine anfangs leere Liste, in der bisher gefundene, unvollständige Reisen gespeichert sind. In der Variable `Shortest_length` wird die Länge der kürzesten der bisher gefundenen Rundreise gespeichert. Die unvollständigen Rundreisen werden schrittweise um einen Teilpfad zu einer weiteren Stadt erweitert. Zusätzlich wird durch einen konservativen Algorithmus die eine untere Grenze für die Länge der restlichen Reise berechnet. Dazu wird die Länge des minimal spanning tree betrachtet, der die letzte Stadt der unvollständigen Reise und alle noch nicht betretenen Städte umfasst. Liegt die Länge einer solchermassen erweiterten Reise zusammen mit der Untergrenze der restlichen Reise über dem Wert von `Shortest_length`, so wird diese Möglichkeit nicht weiter untersucht. Ein Pseudocode-Beispiel findet sich in Abbildung 20. In der parallelisierten Version wird das Berechnen neuer partieller Pfade auf die verschiedenen Prozessoren verteilt. Jeder Prozessor holt sich einen neuen partiellen Pfad ab, und versucht, diesen zu erweitern. Der beste Pfad wird in der globalen Variable `Path` abgelegt. Wenn alle partiellen Pfade aus der Schlange entfernt sind, bricht der Algorithmus ab. Abbildung 21 zeigt den Speedup, den man auf diese Art erreichen kann. Hier liegt die TreadMarks-Version knapp hinter der PVM-Version zurück.

6 Optimierungen und Erweiterungen von TreadMarks

6.1 Einleitung

Das TreadMarks-System enthält bereits in der Basisversion, wie sie in Abschnitt 5 beschrieben wurde, Mechanismen, die ausgewählt wurden, um auf der zugrundeliegenden Hardware akzeptable Ergebnisse zu erzielen. Im Vergleich zur Sequential Consistency wird mit der Release Consistency, die hier über das LRC-Protokoll implementiert wurde, wird die Kommunikation zwischen den Workstations stark reduziert; ohne das Multiple-Writer-Protokoll wäre es nicht sinnvoll, Algorithmen wie die in Abschnitt 4 unter dem Stichwort „False Sharing“ (FS) angesprochenen, mit TreadMarks auszuführen.

Auch wenn man bei diesen beiden Mechanismen stark darauf geachtet hat, daß sie nicht nur ein korrektes System implementieren, sondern auch darauf, das sie ihre Aufgaben möglichst schnell durchführen, bilden sie die Grundbausteine, die TreadMarks ausmachen.

```
queue_t *Queue;
int     *Shortest_length;
int     queue_lock;
int     min_lock_id;

int main (int argc, char **argv)
{
  Tmk_startup ();
  queue_lock_id = 0;
  min_lock_id = 1;
  if (Tmk_proc_id == 0) /* we're the leader: setup shared memory */
    {
      Quere = Tmk_malloc (sizeof (queue_t));
      Shortest_length = Tmk_malloc (sizeof (int));
      /* initialize Heap and Shortest_length */
    }
  Tmk_barrier (0);

  while (true)
    {
      Tmk_lock_aquire (queue_lock_id);
      if (/* queue is empty */)
        {
          Tmk_lock_release (queue_lock_id);
          Tmk_exit ();
        }

      /* keep adding to the queue, until a long, promising
       * tour appears at the head; */

      Path = /* Delete the tour from the head */;
      Tmk_lock_release (queue_lock_id);
    }

    length = /* recursively try all cities not yet on the Path;
              * find the shortest tour length */

    Tmk_lock_aquire (min_lock_id);
    if (length < *Shortest_length)
      *Shortest_length = length;
    Tmk_lock_release (min_lock_id);
}
```

Abbildung 20: TSP-Algorithmus

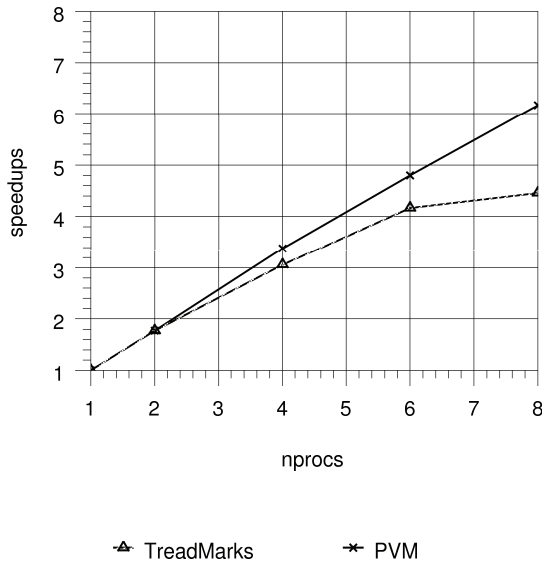


Abbildung 21: Speedup beim Travelling-Salesman-Algorithmus

In diesem Abschnitt werden weitere Arbeiten besprochen, in denen Optimierungen entwickelt wurden, die für solche Probleme eine Lösung oder Verbesserung bringen, die nicht unbedingt in *jedem* Programm zu finden sind, die jedoch in einigen Spezialfällen einen deutlichen Leistungsabfall provozieren.

In [4] diskutieren die Autoren die Vor- und Nachteile, die sich ergeben, wenn man die Größe der Konsistenzeinheiten erhöht. Sie stellen Ergebnisse vor, die die Auswirkung einer Erhöhung der Größe der Konsistenzeinheiten auf ein ganzzahliges Vielfache der Seiten der virtuellen Speicherverwaltung illustrieren. Weiterhin skizzieren sie einen Algorithmus, der Zugriffsmuster erkennt, und versucht, die Seiten, die zusammen angefordert werden, dynamisch zu gruppieren. Diese Ideen werden in 6.2 weiter ausgeführt.

Die Arbeit [18] zeigt einen Weg auf, über den der den Programm-Code mit zusätzlichen Anweisungen versehen kann, die es TreadMarks ermöglichen, Seiten zu identifizieren, die von den einzelnen Prozessoren in einem Code-Abschnitt angefordert werden, um diese Seiten aggregiert anzufordern. Sie wird in Abschnitt 6.3 näher beschrieben.

6.2 Aggregation

In [4] diskutieren die Autoren die Auswirkungen des False Sharing im Zusammenhang mit der Größe der Konsistenzeinheiten, und zeigen, wann es sich lohnen kann, die Konsistenzeinheiten auf ganzzahlige Vielfache der Seiten zu erhöhen.

Von der Zusammenfassung mehrerer Seiten zu einer Konsistenzeinheit verspricht man sich eine Reduktion der versandten Nachrichten. Andererseits können die negativen Folgen des False Sharing die Leistungssteigerung wieder zunichte machen.

Die Autoren untersuchen einerseits, wie sich die Leistungsfähigkeit ändert, wenn mehrere aufeinanderfolgende Seiten statisch zu einer einzigen Konsistenzeinheit zusammengefaßt werden, und diskutieren die vor- und nachteiligen Effekte. Weiterhin präsentieren sie ein Verfahren, das versucht, dynamisch diejenigen Seiten zu gruppieren, die im Ablauf des Algorithmus' auch zusammen übertragen werden.

6.2.1 Statische Aggegation

Die Aggegation hat einige Vorteile, die Nichts mit der Kommunikation zu tun haben: Werden die Seiten statisch aggregiert, so reduziert sich die Anzahl der Access Faults, und damit die Anzahl der Operationen,

die das TreadMarks-System ausführen muß. Negativ macht sich bemerkbar, daß die Twin- und Diff-Operationen auf größeren Datenmengen arbeiten und dafür mehr Zeit beanspruchen.

Im Hinblick auf die Kommunikation stellen vergrößerte Konsistenzeinheiten einen Kompromiss zwischen Aggegation und einem vergrösserten Potential zum FS dar. Hier darf man den FS-Effekt nicht isoliert betrachten, sondern immer das gleichzeitig auftretende TS im Auge behalten. Zur Illustration betrachte man die beiden in Abbildung 22 dargestellten Situationen:

- Angenommen, ein Prozessor p_1 beschreibt zwei aufeinanderfolgende Seiten, s_k und s_{k+1}, die dann nach einer Synchronisationsoperation ein weiterer Prozessor p_2 liest. Sind die Seiten selbst die Konsistenzeinheiten, so entstehen hierbei zwei Access Faults, zwei Diffs werden angefertigt und in zwei Nachrichten verschickt. Hier hätte die Vergrösserung der Konsistenzeinheiten auf zwei Seiten die Anzahl der Nachrichten halbiert. Während einige Daten in diesen Diffs sicherlich useless Data gewesen sein könnten, hat sich an der Gesamtmenge der übertragenen Daten nichts geändert. Hier hat die Vergrösserung der Konsistenzeinheiten eine Reduktion des Kommunikationsaufwandes bewirkt.
- Liest hingegen p_2 nur eine der beiden Seiten, sagen wir, Seite s_k, dann hätte der Kommunikationsaufwand mit den vergrösserten Konsistenzeinheiten zwar gleich viele Nachrichten und Diffs umfasst, während die Diffs jedoch zur Hälfte aus useless data bestanden hätten. In diesem Fall hätte die Vergrösserung der Konsistenzeinheiten eine Verschlechterung gebracht.

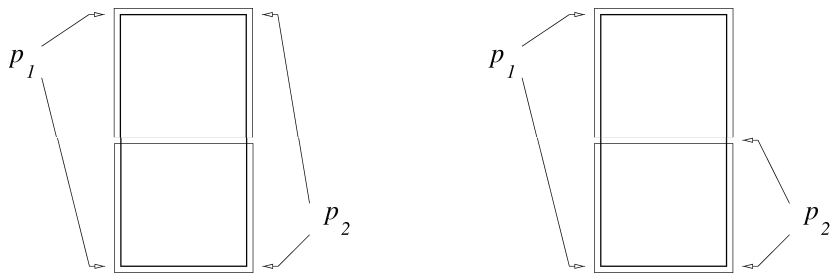


Abbildung 22: Situationen, in denen sich die Verdoppelung der Konsistenzeinheiten (a) positiv und (b) negativ auswirken können.

False Sharing kann zweierlei negative Folgen haben: Einerseits kann es dazu führen, daß (eigentlich) überflüssige Write-Invalidate-Nachrichten verschickt werden; diese Nachrichten bezeichnet man als useless Messages. Andererseits kann es vorkommen, daß beim Datenabgleich mehr Daten verschickt werden, als unbedingt nötig gewesen wären. Diese Daten nennt man useless Data. Beide Effekte werden an kurzen Beispielen erläutert:

1. Useless Messages werden bei sog. write-write FS verschickt.

   Angenommen, zwei Prozessoren, p_1 und p_2 beschreiben dieselbe Seite, wobei p_1 auf die obere Hälfte zugreift, während p_2 die untere Hälfte benutzt. Nachdem alle Prozessoren an einer Barrier synchronisiert haben, liest ein weiterer Prozessor p_3 die obere Hälfte der Seite. Natürlich benötigt p_3 nur die Daten, die von p_1 geschrieben wurden; bei einem (lesenden) Zugriff auf diese Seite erfährt p_3 jedoch einen Access Fault, und muß die Write-Invalidate-Nachrichten sowohl von p_1 als auch von p_2 anfordern.

   Diese Nachrichten sind die useless Messages, die durch write-write FS verusacht werden.

2. Useless Data tritt zwar im Zusammenhang mit True Sharing (TS) auf; jedoch wird dieser Effekt *auch* als FS klassifiziert, da er Kommunikation verursacht, die eigentlich nicht unbedingt nötig gewesen wäre.

Angenommen, Prozessor p_1 beschreibt eine ganze Seite, synchronisiert dann mit Prozessor p_2, der dann die obere Hälfte der Seite liest. Dazu muß ein Diff übertragen werden, daß die Änderungen der *ganzen* Seite enthält, während ein Diff über die obere Hälfte der Seite ausgereicht hätte.

Die Daten, die vom Empfänger eigentlich nicht benötigt werden, sind useless Data.

Die zentrale Aussage ist, daß die Effekte des FS nicht isoliert betrachtet werden sollten, sondern nur im Zusammenhang mit TS:

- Wenn in Bezug auf eine Seite sowohl TS als auch FS existiert, so verursacht die Aggregation mehrerer Seiten einen Anstieg von useless Data.
- Existiert ausschließlich FS, so werden durch die Aggregation useless Messages verursacht.

Wie bereits in früheren Abschnitten erwähnt, sind auf dem Hardware-Setup, auf dem TreadMarks läuft, die Kosten pro Nachrichten viel höher als die Kosten pro Byte innerhalb einer Nachricht. Von den negativen Effekten des FS fallen also in erster Linie diejenigen ins Gewicht, die useless Messages verursachen, während useless Data kein besonders schwerwiegendes Problem darstellt.

Die Benchmark-Applikationen, anhand derer die Autoren von [4] die Auswirkungen ihrer Optimierungen demonstrieren, lassen sich in zwei Klassen einteilen:

1. Bei der einen Klasse findet sich wenig FS, was sich durch kleine Werte sowohl für useless Messages als auch für useless Data ausdrückt.
2. Die andere Klasse zeigt wenige useless Messages, aber eine merkliche Menge an useless Data. Bei diesen Anwendungen tritt zwar FS auf, jedoch nur auf Seiten, auf denen auch TS vorhanden ist.

Für die zweitgenannte Klasse von Applikationen erscheint es lohnend, die Konsistenzeinheiten zu vergrössern: Wenn für die meisten Seiten von vorneherein TS vorliegt, wird durch die Aggregation mehrerer Seiten die Anzahl der zu versendenden Diffs reduziert, ohne daß man mit einer Erhöhung des Anteils an FS rechnen muß.

Bei der erstgenannten Klasse ist nicht offensichtlich, ob eine Aggegation von Seiten von Vorteil ist. Die Aggegation könnte zusätzliches FS an Stellen verursachen, die bisher noch frei davon waren.

6.2.2 Dynamische Aggegation

In der Diskusion der statischen Aggegation hat sich gezeigt, daß einige Programme von dieser Aggegation profitieren werden, während für andere ein Rückgang der Leistungsfähigkeit zu beobachten sein wird. Die Lösung, die in [4] vorgeschlagen wird, besteht darin, die Seiten dynamisch zu sog. Page Groups zusammenzufassen. Diese Zusammenfassung basiert auf den Zugriffsmustern, die das Programm aufzeigt.

Der Gruppierungs-Algorithmus gruppiert Seiten nach folgendem Schema: Bei einer Synchronisationsoperation werden alle Seiten, die zu diesem Zeitpunkt beschrieben worden sind, zu einer Gruppe zusammengefasst.

Wenn im folgenden Ablauf des Programms für eine Seite Diff angefordert wird, werden zusätzlich für *alle* Seiten, die in ihrer Gruppe sind, die Diffs übertragen.

Damit versucht der Algorithmus die Vorteile zu bieten, die sich bei der statischen Aggegation zeigen, und zu verhindern, daß dabei die Nachteile zum Tragen kommen.

Wenn mehrere Seiten von einem Prozessor zusammen angefordert werden, „merkt“ sich der Algorithmus diese Tatsache, und fasst sie in Zukunft zu einer einzigen Nachricht zusammen. Dies ist gerade die Arbeitsweise der statischen Aggregation in den Programmen, für die sie Vorteile bietet.

Wenn der Prozessor jedoch Seiten nicht oder nicht mehr zusammen anfordert, „merkt“ dies der Algorithmus ebenfalls, und fordert die Seiten wieder einzeln an. Damit vermeidet er genau das Verhalten, das statische Aggregation für einige Anwendungen unbrauchbar macht.

Die Nachteile dieses Verfahrens bestehen in dem erhöhten Aufwand, der zur Erkennung der Zugriffsmuster nötig ist. Die Zeit, die dafür benötigt wird, wird allerdings von den Vorteilen mehr als nur wettgemacht.

6.3 Prefetching

Die Basisversion von TreadMarks fordert Daten ausschließlich auf on-demand-Basis an: Genau dann, wenn feststeht, daß eine Seite gebraucht wird, wird sie (genauer gesagt, das entsprechende Diff) übertragen.

Bereits in Abschnitt 5.2 wurde betont, daß die Parallelisierung von Algorithmen für TreadMarks nur von sehr grober Granularität sein kann, da sonst der Kommunikationsaufwand zu groß würde. Umgangssprachlich ausgedrückt, sollte ein Algorithmus so gestaltet werden, „daß jeder Prozessor möglichst lange möglichst alleine läuft, ohne mit anderen Prozessoren kommunizieren zu müssen". In den Phasen, in denen die Prozessoren nun „alleine laufen", kann es vorkommen, daß sie mehr Daten anfordern, als in einer einzigen Konsistenzeinheit enthalten sind. Die in [18] vorgeschlagene Methode ermöglicht es, einen großen Teil dieser Seiten zu identifizieren, und bei Schleifeneintritt in eine einzigen Nachricht zusammenzufassen.

Um festzustellen, *welche* Seiten ein Prozessor in einer Schleifeniteration anfordern wird, wird das sog. Indirection Array betrachtet. Ein Beispiel für solch ein Array findet sich in dem SOR-Algorithmus in Abschnitt 5.5: Dort werden für jeden Prozessor lokal in den Variablen `begin` und `end` die Grenzen der 'Streifen' berechnet, die der einzelne Prozessor zu bearbeiten hat. Speichert man diese Grenzen in einer

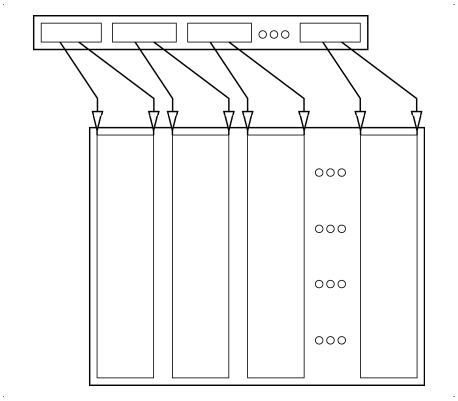


Abbildung 23: Im Indirection Array (oben) ist festgehalten, welcher Prozessor welchen Bereich der Matrix (unten) bearbeitet. Mit diesen Bereichen kennt man auch die Seiten, die der einzelne Prozessor in der nächsten Iteration laden wird. Diese Seiten können dann zu Beginn der Iteration in einer einzigen Nachricht übertragen werden.

globalen Datenstruktur, wie in Abbildung 23 angedeutet, so lassen sich mit den 'Streifen' auch die Seiten identifizieren, die ein Prozessor im nächsten Iterationsschritt laden wird. Diese Seiten können dann in einer einzigen Nachricht übertragen werden.

Das Inspector-Executor-Modell erfordert, wie oben angedeutet, zwei Durchläufe durch den Algorithmus: Der Inspector identifiziert Zugriffe, für die Daten von anderen Prozessorknoten angefordert werden muß, während der Executor mit dem vom Inspector erstellten Kommunikationsplan den Code 'richtig' ausführt. Ein Nachteil dieser Methode ist, daß die Analysearbeit, die der Inspector durchführt, sehr komplex ist und vom Compiler ein großes Mass an 'Intelligenz' erfordert. Auch das Auslagern des Inspectors aus der innersten Schleife ist sehr aufwendig. Die hier vorgestellte Methode hingegen betrachtet lediglich die Indirection Arrays, die gewöhnlich eine regulärer Abschnitt sind oder als ein solcher implementiert werden können.

Die Autoren von [18] illustrieren die Folgen dieser Optimierung an mehreren Benchmarks. Als Massstab dient das CHAOS-System, das auf Basis des Inspector-Executor-Modells arbeitet.

Sie präsentieren Ergebnisse, nach denen das so optimierte TreadMarks-System für ein Problem bis zu 23% schneller als CHAOS ist; für ein anderes Problem ist TreadMarks nie mehr als 14% langsamer. Bezieht man die Laufzeit, die der Inspector benötigt, mit in die Zeitmessungen ein, so ist das optimierte TreadMarks-System *immer* schneller als das Inspector-Executor-System.

Insgesamt ist die optimierte Version von TreadMarks bis zu 38% schneller als die Basisversion.

7 Zusammenfassung

In diesem Artikel wurde das TreadMarks-System beschrieben, das es ermöglicht, einen Cluster aus Standard-Workstations als Parallelrechensystem zu benutzen.

Damit wird es einerseits möglich, eine „GroSSrechner“ zu benutzen, der in vielen Institutionen und Firmen schon vorhanden ist, wodurch die problematische und kostenintensive Anschaffung eines GroSSrechners vermieden werden kann. Andererseits müssen die Workstations, auf denen TreadMarks läuft, nicht für diesen Zweck ‘beiseitegestellt’ werden, sondern sie stehen weiterhin für den ‘normalen’ Gebrauch zur Verfügung.

Die Probleme, die die Autoren von TreadMarks gelösst haben, liegen zum einen in einer einfachen, übersichtlichen und vor allem portablen Programmierung des DSM-Systems; so kann sich der Programmierer ganz auf die Erstellung eines korrekten, effizienten Algorithmus' konzentrieren, ohne zuviel Zeit auf die Programmierung komplizierter Kommunikationsoperationen ver(sch)wenden zu müssen.

Zum Anderen erweist sich bei einem Workstation-Cluster das Netzwerk, mit dem die Workstations verbunden sind, oft als Flaschenhals. Auch hier haben die Autoren dafür gesorgt, daß dieses Problem sich kaum mehr negativ auf die Leistung auswirkt.

Nach einer Einführung über die Parallelprogrammierung und einer Übersicht über die Synchronisations-Mechanismen wurden die beiden Mechanismen vorgestellt, die die Leistungsfähigkeit von TreadMarks ausmachen, die Lazy Release Consistency und das Multiple-Writer-Protokoll. Das Lazy Release Consistency ist eine Weiterentwicklung der Sequential Consistency, das jedoch die Kommunikation zwischen den Workstations entscheidend reduziert.

Daran anschliessend erfolgte ein Überblick über das API von TreadMarks. Durch Zeitmessungen von Benchmarks konnte deutlich demonstriert, das mit TreadMarks realisierte parallele Programme nur einen gerinen Anteil ihrer Zeit mit Konsistenzerhaltung, Speicherverwaltung und Kommunikation verbringen, und den grössten Teil iherer Zeit für der eigentlichen Berechnung zur Verfügung haben.

Den Abschluss bildeten zwei Möglichkeiten, durch Erweiterungen von TreadMarks eine weitere Leistungssteigerung zu erzielen. Zum einen handelte es sich hier um die Aggregation von Konsistenzeinheiten, zum anderen wurde gezeigt, wie sich Konsistenzeinheiten durch Prefetching angefordert werden können, bevor ein Kontrollfaden darauf warten muß.

Literatur

[1] S. Adve, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the Second High Performance Computer Architecture Conference*, pages 26–37, February 1996.

[2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proceedings of the Third High Performance Computer Architecture Conference*, pages 261–271, February 1997.

[4] Cristiana Amza, Alan L. Cox, Karthick Rajamani, and Willy Zwaenepoel. Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proceedings of the Sixth Conference on Principles and Practice of Parallel Programming*, pages 90–99, June 1997.

[5] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, and Willy Zwaenepoel. An Integrated Approach to Distributed Shared Memory. In *First International Workshop on Parallel Computing*, December 1994.

[6] Alan L. Cox, Sandhya Dwarkadas, Honghui Lu, and Willy Zwaenepoel. Evaluating the Performance of Software Distributed Shared Memory as a Target for Parallelizing Compilers. In *Proceedings of the International Parallel Processing Symposium*, pages 474–482, April 1997.

[7] K. Diefendorff and M. Allen. Organisation of the Motorola 88110 Superscalar RISC Microprocessor. *IEEE Micro*, pages 40–63, April 1992.

[8] Wolfgang Domschke and Andreas Drexel. *Einführung in Operations Resarch.* Springer-Verlag, 1990.

[9] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[10] Sandhya Dwarkadas, Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. Technical report, Rice University, 1994.

[11] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter '94 Usenix Conference*, pages 115–131, January 1994.

[12] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. An Evaluation of Software-Based Release Consistent Protocols. *Journal of Parallel and Distributed Computing, Special Issue on Distributed Shared Memory*, 29:126–141, October 1995.

[13] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, Houston, Texas, January 1995.

[14] Leonidas I. Kontothanassis, Michael L. Scott, and Ricardo Bianchini. Lazy Release Consistency for Hardware-Coherent Multiprocessors. Technical report, Department of Computer Science, University of Rochester, December 1994.

[15] Ulrich Kulisch. Memorandum über Computer, Arithmetik und Numerik. Institut für Angewandte Mathematik, Universität Karlsruhe (TH), 1997.

[16] Ulrich Kulisch, Thomas Teufel, and Bernd Höfflinger. Genauer und trotzdem schneller — Ein neuer Coprozessor für hochgenaue Matrix- und Vektoroperationen. *Elektronik — Fachzeitschrift für industrielle Anwender und Entwickler*, 26, 27. Dezember 1994.

[17] Leslie Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):241–248, September 1979.

[18] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and Software Distributed Shared Memory Support for Irregular Applications. In *Proceedings of the Sixth Conference on Principles and Practice of Parallel Programming*, pages 48–56, June 1997.

[19] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Proceedings on Supercomputing '95*, December 1995.

[20] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Quantifying the Performance Differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computation*, 43(2):65–78, June 1997.

[21] Parallel Resarch, Inc. *FORGE High Performance Fortran User's Guide, version 2.0.*

[22] Parallel Resarch, Inc. *FORGE Shared Memory Parallelizer User's Guide, version 2.0.*

[23] L.L.C. ParallelTools. *Concurrent Programming with TreadMarks*, 1996.

[24] M. Ronsse and Willy Zwaenepoel. Execution Replay for TreadMarks. In *Proceedings of the Fifth EUROMICRO Workshop on Parallel and Distributed Processing*, pages 343–350, January 1997.

[25] Theo Ungerer. *Mikroprozessortechnik.* Thomson-Verlag, 1995.

[26] Theo Ungerer, Winfried Görke, and Detlev Schmid. Rechnerstrukturen. Institut für Rechnerentwurf und Fehlertoleranz, Fakultät für Informatik, Universität Karlsruhe (TH), 1996. Skripten an der TH Karlsruhe.

[27] Horst Wettstein. *Systemarchitektur.* Hanser Verlag, 1988.

[28] Weimin Yu and Alan L. Cox. Java/DSM: a Platform for Heterogeneous Computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.