Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

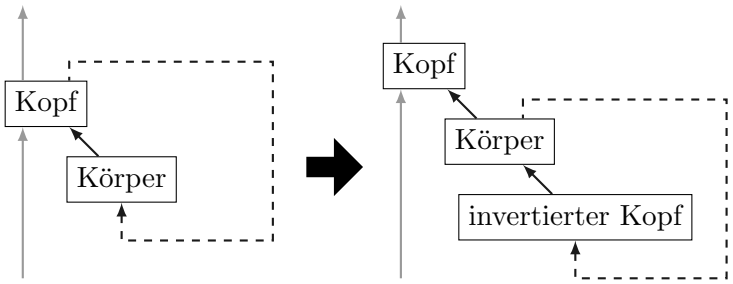
Lehrstuhl Prof. Dr.-Ing. Snelting

Entwicklung von Kriterien zur Anwendung von Schleifenoptimierung im Kontext SSA-basierter Zwischensprachen

Studienarbeit von

Christian Helmer

an der Fakultät für Informatik



Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter:	Dipl.-Inform. Sebastian Buchwald

Bearbeitungszeit: 19. April 2010 – 04. Oktober 2010

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Eine bekannte Regel besagt, dass 90% der Ausführungszeit in 10% des Maschinencodes verbracht wird – ein Großteil davon in Schleifen. Aus diesem Grund ist die Optimierung von Schleifen besonders lukrativ. Geringfügige Änderungen können durch die häufige Ausführung die Performanz stark beeinflussen.

Schleifenoptimierungen – wie die Inversion und das Ausrollen – können somit zu einer kürzeren Laufzeit beitragen. Voraussetzung für eine Beschleunigung ist jedoch die Wahl geeigneter Kriterien. Das Ziel ist es, Kriterien dafür zu finden, die angeben, ob eine bestimmte Optimierung ausgeführt werden soll. Zudem besitzen diese Optimierungen Parameter, die über die Ausführung der Optimierungen bestimmen. Diese Parameter gehen aus den Kriterien hervor. Die Kriterien sollen folglich dazu dienen, im Falle der Ausführung der Optimierung, in einer Beschleunigung des Zielprogramms zu resultieren. Die Verwendung ungeeigneter Parameter führt dazu, dass zu viele Instruktionen hinzugefügt werden und die positiven Auswirkungen durch Cache-Effekte revidiert werden.

Im Rahmen dieser Studienarbeit werden verschiedene Kriterien für die Schleifeninversion und das Schleifenausrollen auf ihren Einfluss auf die Laufzeit des Zielprogramms hin untersucht. Ziel ist, festzustellen, welche Parameter sich positiv auf die Ausführungszeit auswirken.

2 Grundlagen

Bei SSA[1]-basierten [RWZ88, CFR+91] Zwischensprachen wird jeder Variablen statisch genau einmal ein Wert zugewiesen. Dadurch werden die Datenabhängigkeiten zwischen Operationen explizit dargestellt. Da eine Vielzahl von Optimierungen auf der Datenflussanalyse basieren, bietet eine solche Darstellung besonders gute Voraussetzungen.

Firm [LBBG05, TLB99] ist eine SSA-basierte Zwischensprache für Compiler. Programme liegen in Firm als Graphen vor. Die Knoten der Graphen stehen für Werte. Mehrere Knoten können Teil eines Grundblocks sein. Datenabhängigkeiten und Steuerflussabhängigkeiten werden in Firm durch rückwärtsgerichtete[2] Kanten repräsentiert. Da bei den hier betrachteten Optimierungen zumeist Grundblöcke bearbeitet werden, werden Schleifen wie in Abbildung 2.1 dargestellt. Die Knoten in der Abbildung stehen hierbei für Grundblöcke und die Kanten sind somit Steuerflusskanten. Da es sich um eine Schleife handelt, ist eine Rückwärtskante vorhanden. Rückwärtskanten werden mit einer gestrichelten Linie dargestellt. Kanten, die für die Bearbeitung nicht zentral sind, werden in Grau dargestellt.

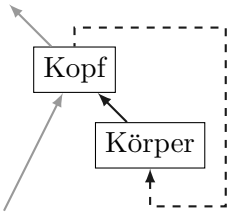


Abbildung 2.1: Grundblöcke einer einfachen Schleife in Firm

libFirm [Lin02] ist eine Bibliothek, die Analysen und Optimierungen auf Firm-Graphen bereitstellt. In dieser Arbeit wird libFirm mit dem Frontend cparser verwendet, um C-Programme zu übersetzen.

[1] engl.: static single assignment
[2] entgegen des Programmablaufs

3 Schleifeninversion

3.1 Ziele

Ziel der Schleifeninversion ist es unbedingte Sprungbefehle zu vermeiden. Diese entstehen in kopfgesteuerten Schleifen. Dazu zählen While-Schleifen, sowie For-Schleifen. Die Bezeichnung ergibt sich daraus, dass die Schleifenbedingung jeweils am Schleifeneintritt ausgewertet werden muss, und sich aus diesem Grund im Schleifenkopf befindet. Deshalb wird nach Abarbeitung des Schleifenkörpers, ein Sprung in die Schleifenbedingung nötig. Dies ist in Listing 3.1 in einer Pseudozwischensprache dargestellt. Betrachtet man dagegen Do-While-Schleifen – wie in Listing 3.2 dargestellt – entfällt dieser unbedingte Sprung und wird durch den bedingten Sprung der Schleifenbedingung ersetzt. Da hierbei die Schleifenbedingung nach Ablauf des Schleifenkörpers ausgewertet wird, werden Do-While-Schleifen als endgesteuert bezeichnet.

```
i = 0;
loop:
if (i >= 100) goto ende
//...
++i;
goto loop
ende:
```

Listing 3.1: Darstellung einer While-Schleife in einer Zwischensprache

```
i = 0;
loop:

//...
++i;
if (i < 100) goto loop
//ende:
```

Listing 3.2: Darstellung einer Do-While-Schleife in einer Zwischensprache

Bei der Verwendung einer While-Schleife, lässt sich folglich ein Befehl pro Schleifendurchlauf einsparen, wenn stattdessen eine Do-While-Schleife verwendet wird. Dies wirkt sich stärker auf die Laufzeit aus, wenn die Schleife häufig durchlaufen wird, und ansonsten einen nicht allzu großen Schleifenkörper besitzt.

3.2 Bedingungsketten

Bisher wurde davon ausgegangen, dass der Schleifenkopf im Wesentlichen aus einer Bedingung besteht. Oftmals besteht dieser allerdings aus mehreren Grundblöcken und enthält zahlreiche Operationen. Da häufig Werte berechnet werden, die sowohl für die Schleifenbedingung als auch für die Weiterverarbeitung innerhalb der Schleife benötigt werden, lassen sich zum Schleifenkopf gehörende Grundblöcke nicht immer

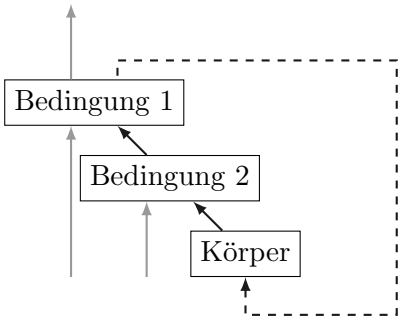


Abbildung 3.1: Schleife mit Bedingungskette

von den zu dem Schleifenkörper gehörenden unterscheiden. Der Grundblock mit der Bedingung 2 in Abbildung 3.1 kann entweder dem Schleifenkopf zugeordnet werden, oder dem Schleifenkörper. Für die Einsparung des unbedingten Sprungs ist dies unwichtig, da dieser mit der Inversion des Grundblocks, der die Bedingung 1 enthält, bereits überflüssig wird. Existieren am Anfang der Schleife zwei oder mehr aufeinander folgende Grundblöcke, wie in Abbildung 3.1, welche jeweils eine Bedingung enthalten, die die Schleife verlässt, so handelt es sich dabei um eine Bedingungskette[1].

3.3 Transformation

Die beiden Schleifen in Listing 3.1 und Listing 3.2 besitzen die semantisch gleiche Schleifenbedingung. Der einzige Unterschied besteht darin, dass der Schleifenkörper der Do-While-Schleife ohne Überprüfung der Bedingung betreten wird. Es muss nun sichergestellt werden, dass die Schleifenbedingung bei der invertierten Schleife vor dem Eintritt überprüft wird. Dafür existieren zwei Varianten.

```
i = 0;
goto :condition
:loop
//...
++i;
:condition
if (i < 100) goto :loop
```

Listing 3.3: Inversion 1

Die von Feign [Fei79] aufgezeigte Variante, besteht darin, die Schleifenbedingung an das Ende der Schleife zu verschieben, und vor die Schleife einen unbedingten Sprung in die Schleifenbedingung zu setzen. Das Ergebnis ist in Listing 3.3 dargestellt. In Abbildung 3.2 ist diese Methode auf einen Graphen angewandt. Allerdings ist dieser Graph identisch mit Listing 3.1[2].

[1] engl.: condition chain/CC

[2] Da Rückwärtskanten eine Information ist, die aus dem Graphen gewonnen wird, und nicht als Information für die Codeerzeugung festgelegt wird.

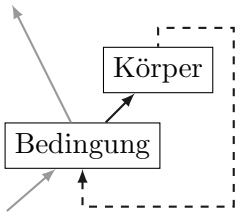


Abbildung 3.2: Inversion der Zwischensprache auf einen Graphen angewandt

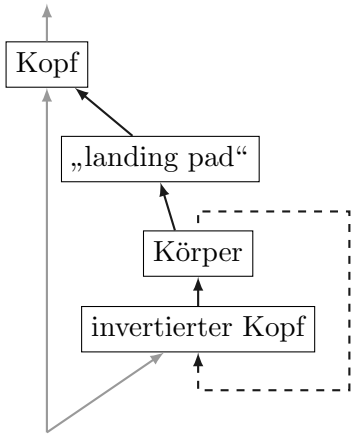


Abbildung 3.3: Schleifeninversion bei Graphen

Die zweite Variante kann jedoch auf Graphen angewendet werden. Hierbei wird ebenfalls der Schleifenkopf ans Schleifenende verschoben. Die daraus resultierende Do-While-Schleife würde auf jeden Fall einmal ausgeführt werden. Um dies zu verhindern, wird der Schleifenkopf kopiert und vor die Schleife gesetzt, wobei das Original, der invertierte Schleifenkopf, in der Schleife verbleibt. Abbildung 3.3 zeigt den Ergebnisgraph. Der als „landing pad“ bezeichnete Grundblock wird nach der Schleifenoptimierungsphase automatisch eingefügt, da es sich bei der Kante von *Kopf* zu *Körper* um eine kritische Kante handelt. Dies ermöglicht es nachfolgenden Optimierungen, schleifeninvarianten Code in das landing pad auszulagern. Im Falle einer Bedingungskette können möglicherweise Bedingungen nach einmaliger Auswertung aus der Schleife entfernt werden.

3.4 Kriterien für die Schleifeninversion

In [Fei79] wird die Schleifeninversion als Teil der Normalisierung von Schleifen angesehen. Dies kann allerdings nur in der ersten Variante als solche betrachtet werden, da in diesem Fall kein Maschinencode kopiert wird. Durch die Kopie des Schleifenkopfs erhöht sich die Anzahl der Instruktionen. Dies wirkt sich auf die betrachtete Schleife selbst nicht aus. Falls es sich jedoch um eine verschachtelte Schleife handelt, wird dadurch der Schleifenkörper der umgebenden Schleife vergrößert. Passt

die umgebende Schleife anschließend nicht mehr in den Befehlscache, ist ein Performanzverlust zu erwarten, da bei jeder Iteration Maschinencode nachgeladen werden muss. Da nicht bekannt ist, ob große Mengen an Instruktionen schleifeninvariant sind, sodass eine Auslagerung ermöglicht wird, wird die Größe der Schleife und der Bedingungskette limitiert. Um die Gesamtgröße der Bedingungskette zu limitieren, werden nicht alle Grundblöcke der Bedingungskette dem Schleifenkopf zugeordnet. Stattdessen wird vor dem Hinzufügen einer weiteren Bedingung sichergestellt, dass die Größe der Bedingungskette innerhalb eines Grenzwertes verbleibt.

Unter der 'Größe der Schleife' wurde bisher die Größe des Maschinencodes verstanden. Zum Zeitpunkt der Optimierung sind jedoch ausschließlich Knoten vorhanden, ohne zu wissen, welche Größe der Programmcode besitzen wird. Denkbar wäre hierbei die Verwendung des Backends, welches es ermöglicht, für einen Grundblock temporär Programmcode zu erzeugen, dessen Größe damit bekannt wäre. Allerdings ändern darauf folgende Optimierungen das Quellprogramm ab, wodurch auch diese Abschätzung ungenau wird. Da eine genauere Abschätzung nicht verfügbar ist, wird die Größe anhand der Knotenzahl abgeschätzt.

Zur Ausführung der Geschwindigkeitsmessungen, wurden folgende Werte als Parameter für die Schleifenoptimierung gewählt.

- Ein Schwellwert für die maximale Anzahl Knoten in der Schleife.
- Ein Schwellwert der den Anteil des Schleifenkopfs an der gesamten Schleife – und damit die Größe des kopierten Maschinencodes – limitiert.
- Das Zulassen von Funktionsaufrufen innerhalb der Schleife. Da die Laufzeit einer Funktion nicht bekannt ist, und der Aufruf selbst bei geringer Laufzeit einen zusätzlichen Aufwand verursacht, ist die Optimierung von Schleifen mit Funktionsaufrufen selten lohnend [Mor98].
- Ein positiver oder negativer Prozentsatz, der die erlaubte Gesamtgröße der Schleife pro Verschachtelungstiefe erhöht oder verringert. Damit soll festgestellt werden, wie sich die Schleifengröße in verschachtelten Schleifen auswirkt. Es ist möglicherweise vorteilhafter, dass mehrere verschachtelte Schleifen in den Befehlscache passen, statt die Abarbeitung der innersten zu optimieren.

3.5 Implementierung

Schleifeninversion und Schleifenausrollen wurden als Optimierungsphasen in libFirm implementiert. Diese Optimierungen werden auf die interne Graphrepräsentation jeder Prozedur des Eingabeprogramms angewendet. Durch libFirm wird bereits ein Mittel geboten, den Schleifenbaum aufzubauen. Dieser enthält sämtliche verschachtelte Schleifen einer Prozedur. In diesem Schleifenbaum kann somit bis zu den Blättern abgestiegen werden, um die am tiefsten verschachtelten Schleifen zu bearbeiten.

Die Zugehörigkeit eines Grundblocks zu der aktuell bearbeiteten Schleife lässt sich

ohne weiteres bestimmen, da alle Grundblöcke einen Zeiger auf die am tiefsten verschachtelte Schleife besitzen, zu der sie gehören.

Operationen auf dem Firm-Graphen lassen sich am besten mit sogenannten Walkern durchführen. Walker durchlaufen den Graph rekursiv entlang der rückwärtsgerichteten Kanten und führen bei jedem Knoten oder Grundblock einmalig die übergebene Funktion aus.

Das grundblockweise Bearbeiten, welches für die Schleifeninversion und das Schleifenausrollen intuitiv erscheint, hat sich schnell als sehr umständlich herausgestellt. Grund dafür ist, dass dynamisch erzeugte Informationen wie Rückwärtskanten oder inverse Kanten zu Steuer- und Datenflusskanten während der Transformation erhalten bleiben müssen. Unter Beachtung dieser Gegebenheiten, haben sich die im Folgenden beschriebenen Vorgehensweisen als vorteilhaft erwiesen.

3.5.1 Implementierung der Schleifeninversion

Zunächst wird die Schleife auf Vorhandensein einer Bedingungskette untersucht. Dazu wird die Schleife – ausgehend vom Schleifenkopf – Grundblockweise entgegengesetzt der gerichteten Kanten durchlaufen. Dies ist möglich, da libFirm die Möglichkeit bietet, zu jeder Kante die ensprechend entgegengesetzte Kante zu erzeugen. Wird ein Block mit einer Steuerflusskante gefunden, die aus der Schleife heraus führt, so befindet sich in diesem Grundblock eine Bedingung, die zum Verlassen der Schleife führen kann. Verbleibt der Steuerfluss in der Schleife, so wird der folgende Grundblock untersucht. Die Grundblöcke der Bedingungskette werden als solche mit einer Markierung versehen. Die zuvor beschriebene Bedingung reicht noch nicht aus, sodass bisher nur Kandidaten für eine Bedingungskette gefunden wurden. Es muss anschließend sichergestellt werden, dass die Grundblöcke der Bedingungskette ausschließlich markierte Vorgänger besitzen. Im diesem Zug wird die Größe der Grundblöcke beachtet, und nur Kandidaten in die Bedingungskette übernommen, solange deren Gesamtgröße das Maximum nicht überschreiten. Die Markierung der nicht verwendeten Grundblöcke wird entfernt.

Um die Bedingungskette zu kopieren, werden alle Kanten benötigt, mit der die Bedingungskette betreten werden kann. Das sind alle Kanten, deren Ursprung sich außerhalb, deren Ziel sich jedoch innerhalb der Bedingungskette befindet. Diese, als Eintrittskanten bezeichneten Kanten, beinhalten die Information ihres Quell- und Zielknotens.

Indem diese Eintrittskanten von einem kopierenden Walker rekursiv verfolgt werden, kann die Bedingungskette kopiert werden. Dieser Walker läuft alle Knoten ab, deren Grundblock markiert ist und die sich somit in der Bedingungskette befinden. Dabei wird jeder Knoten kopiert und mit dem Originalknoten verknüpft, sodass aus einem Knoten die zugehörige Kopie erhalten werden kann. Alle Knoten werden mit ihren Eingangskanten kopiert, sodass die Kopie in sich schon korrekt vernetzt ist. Die Kopie der Bedingungskette befindet sich später vor der Schleife, das Original verbleibt in der Schleife und wird zu der invertierten Bedingungskette. In Abbildung 3.5 ist das Ergebnis angedeutet. Der dunkel hinterlegte Knoten ist die Kopie, welche

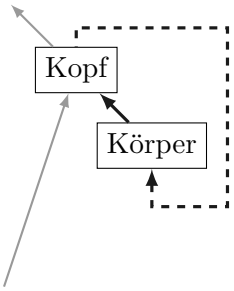


Abbildung 3.4: Ausgangspunkt für die Schleifeninversion

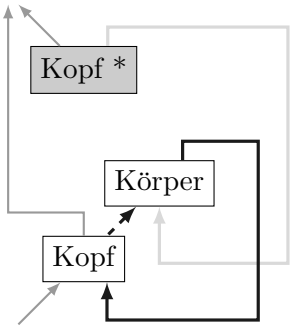


Abbildung 3.5: Schritt 1: Schleifenkopf kopiert

zunächst alle Eingänge behält. Mit dem Stern wird jeweils die aktuelle Veränderung gekennzeichnet.

Es folgt das Neuverbinden der bestehenden Knoten. Im Programmablauf folgt der Schleifenkörper auf den Schleifenkopf und den invertierten Schleifenkopf. Dazu werden zunächst alle Steuerfluss-Eintrittskanten des Schleifenkörper verdoppelt, wobei die duplizierten Kanten auf die Kopie ihres Zielgrundblocks zeigen. Das Ergebnis ist in Abbildung 3.6 zu sehen. Damit einher geht die Erweiterung der zu dem Schleifenkörper gehörenden ϕ-Knoten. Wurde die Steuerflusskante am Eingang i des Schleifenkörpers dupliziert, so muss auch jeder ϕ-Knoten des Schleifenkörpers um die Kopie des Eingangs an der Stelle i erweitert werden.

Damit sind jedoch noch nicht alle Datenflusskanten abgearbeitet. Da durch die Verdopplung des Schleifenkopfs dieser nun den Schleifenkörper nicht mehr dominiert, müssen ϕ-Knoten an den Dominanzgrenzen erzeugt werden. Dies geschieht mit der in libFirm bereits vorhandenen Funktion construct_ssa, die zu zwei Zuweisungen und deren Verwendern die nötigen ϕ-Knoten erzeugt.

Bezüglich der Datenflusskanten wurde bisher ein Sonderfall ignoriert. Es ist möglich, dass innerhalb der Bedingungskette Datenfluss-Schleifen vorhanden sind. Wird ein Wert ausschließlich in der Bedingungskette, nicht jedoch im Schleifenkörper zugewiesen, so erhält man einen Fall wie in Abbildung 3.7 dargestellt. Diese Datenfluss-Schleifen werden erkannt, indem alle Eingänge der ϕ-Knoten des ersten Grundblocks

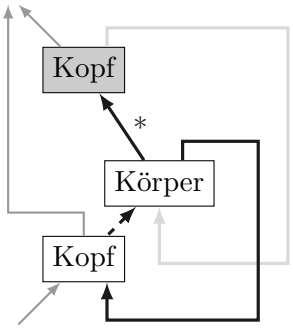


Abbildung 3.6: Schritt 2: Steuerflusskanten zwischen dem Schleifenkopf und dem Schleifenkörper erstellt

der Schleife darauf untersucht werden, ob sich die Berechnung innerhalb der Bedingungskette befindet. Die Werte von Add_1 und Add_2 werden werden durch Konstruktion von geeigneten ϕ-Knoten zusammengefasst.

Zuletzt werden überflüssige Steuerflusskanten der kopierten Bedingungskette und des invertierten Schleifenkopfs entfernt. Da die kopierte Bedingungskette nicht mehr Teil der Schleife ist, werden ebenfalls die Rückwärtskanten entfernt. Der invertierte Schleifenkopf dagegen behält ausschließlich die schleifeneigenen Rückwärtskanten. Abbildung 3.8 zeigt das Ergebnis.

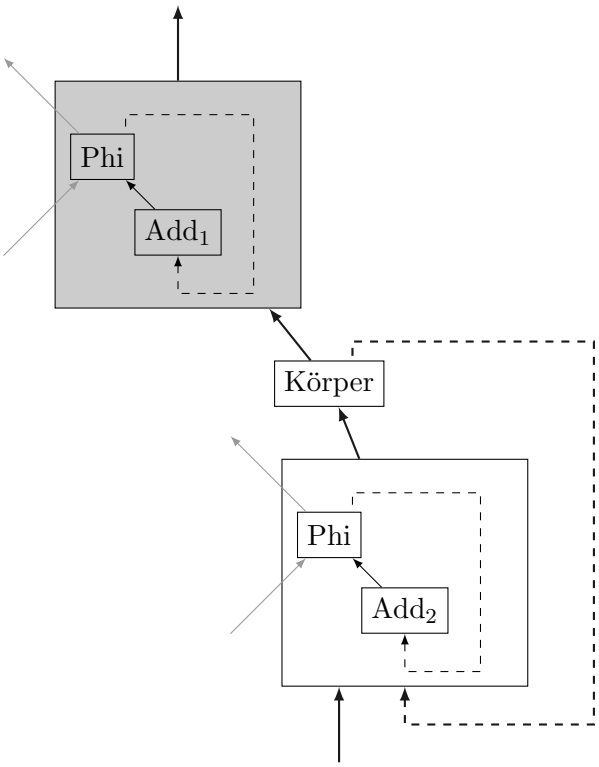


Abbildung 3.7: Schritt 4: Datenfluss-Schleife vor dem Neuverbinden

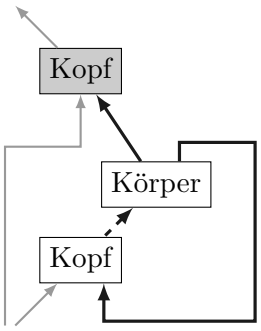


Abbildung 3.8: Schritt 5: Überflüssige Steuerflusskanten entfernt

4 Schleifenausrollen

4.1 Ziele

Betrachtet man eine Schleife, so besteht diese aus der Berechnung der Nutzdaten, und den Berechnungen für die Schleifenbedingung. Die Auswertung der Schleifenbedingung verursacht somit einen gewissen Overhead. Um diesen zu reduzieren, kann eine Schleife ausgerollt werden. Die Schleife aus Listing 4.1 wurde in Listing 4.2 4-fach ausgerollt. Dazu wird der Schleifenkörper vervielfacht, und die Schleifenbedingung derart angepasst, dass weiterhin die korrekte Anzahl von Schleifendurchläufen erreicht wird.

```
i=0
loop:
a[i] = b[i]
++i
if (i<100) goto loop
```

Listing 4.1: Zählschleife

```
i=0
loop:
a[i] = b[i]
++i
a[i] = b[i]
++i
a[i] = b[i]
++i
a[i] = b[i]
++i
if (i<100) goto loop
```

Listing 4.2: Zählschleife, 4-fach ausgerollt

Ziel ist es, die Schleifenbedingung seltener zu überprüfen und somit die Laufzeit zu reduzieren. Zudem können darauffolgende Optimierungen auf den gesamten, ausgerollten Schleifenkörper angewandt werden, um statische Berechnungen zusammenzufassen. Das Ausrollen kann dazu führen, dass die Prozessorpipeline besser ausgenutzt wird, da nun größere Code-Blöcke entstehen, die nicht von der Auswertung der Schleifenbedingung unterbrochen werden [Mor98].

4.2 Verfahren

Die Schleife aus Listing 4.1 auszurollen ist möglich, da im Voraus bekannt ist, dass der Schleifenkörper 100 mal ausgeführt wird. Somit ist es ausreichend, die Bedingung jede 4. Iteration zu überprüfen. Die 4 ist somit der Ausrollfaktor F, und gibt die Anzahl der Schleifenkörper in der ausgerollten Schleife an. Aus diesem Grund sind Zählschleifen dazu geeignet ausgerollt zu werden, im Gegensatz zu Schleifen deren Bedingung von einer globalen Variable oder von einem komplexen Ausdruck abhängt.

4.2.1 Konstantes Ausrollen

Ist eine Schleife mit konstanten Werten für die Schleifenvariablen *Startwert*, *Schritt* und *Endwert* vorhanden, so kann zur Übersetzungszeit ein Ausrollfaktor bestimmt werden. Dazu wird die Anzahl der Schleifendurchläufe C ermittelt. Jeder Teiler von C ist ein möglicher Ausrollfaktor. Unterabschnitt 4.2.1 zeigt den Graph einer konstant ausgerollten Schleife.

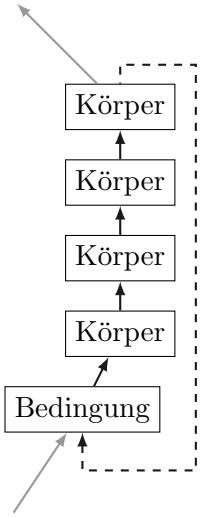


Abbildung 4.1: Graph einer ausgerollten Schleife

4.2.2 Invariantes Ausrollen

Neben Schleifen mit konstanten Schleifenvariablen, existieren Zählschleifen, deren *Startwert*, *Schritt* oder *Endwert* erst während der Laufzeit bekannt sind. Mit dem Verfahren für konstante Schleifenvariablen kann somit kein geeigneter Ausrollfaktor bestimmt werden, da erst zur Laufzeit bekannt ist, ob der Ausrollfaktor die Anzahl

der Iterationen ganzzahlig teilt. In diesem Fall kann auf Duff's Device[1] zurückgegriffen werden.

```
void copy(unsigned count) {
 do {
  *to++ = *from++;
 } while (--count > 0);
}
```

Listing 4.3: Einfache Schleife in C

```
void copy(unsigned count) {
  unsigned n = (count + 3) / 4;
    switch (count % 4) {
      case 0:
        do {
          *to++ = *from++;
      case 3:
          *to++ = *from++;
      case 2:
          *to++ = *from++;
      case 1:
          *to++ = *from++;
        } while (--n > 0);
  }
}
```

Listing 4.4: Duff's Device

Duff's Device geht auf die in C valide Prozedur in Listing 4.4 zurück, die einen Versuch einer Optimierung von Listing 4.3 darstellt[2]. Ausgegangen wird von einer bereits ausgerollten Schleife mit einem Ausrollfaktor $F = 4$. Die Bedingung soll folglich nur nach jedem 4. Schleifendurchlauf geprüft werden. Wobei 4 jedoch nicht nötigerweise ein Teiler von C ist.

Duff's Device löst dieses Problem, indem zunächst die ersten C *modulo* 4 Iterationen ausgeführt werden. Anschließend ist sichergestellt, dass die verbleibenden Schleifendurchläufe durch 4 teilbar sind. Einsprungpunkte vor jedem Schleifenkörper ermöglichen dies. Zur Verdeutlichung zeigt Abbildung 4.2 den daraus resultierenden Graphen.

Bedingungen für die Anwendung ist, dass es sich bei der Schleife um eine einfache Zählschleife handelt. Das heißt, in diesem Fall:

- Startwert, Schritt und Endwert sind schleifeninvariant. Damit ist die Anzahl

[1] Duff's Device ist eigentlich eine Umsetzung von – in Assembler problemlos umzusetzendem – Schleifenausrollen in C.

[2] Die tatsächliche Laufzeit hängt von dem verwendeten Compiler ab und ob dieser diese Struktur im Maschinencode wie erwartet abbildet.

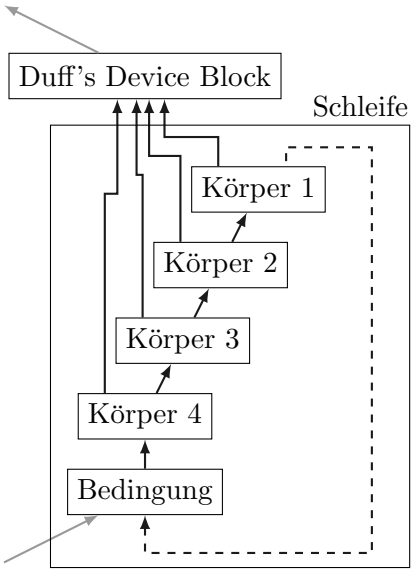


Abbildung 4.2: Schleife ausgerollt per Duff's Device

der Schleifendurchläufe zur Laufzeit berechenbar und ebenfalls schleifeninvariant.

- Es wird davon ausgegangen, dass nur eine Schleifenbedingung – abhängig von genau einer Induktionsvariablen – existiert.
- Wie in Listing 4.4 nachvollzogen werden kann, muss $C > 0$ sein.

4.3 Kriterien für das Schleifenausrollen

Der Ausrollfaktor wird so hoch gewählt, dass die Knotenanzahl der ausgerollten Schleife unter einem angegebenen Knotenlimit bleibt. Dies soll erreichen, dass die ausgerollte Schleife in den Befehlscache passt. Da durch Duff's Device zusätzlicher Aufwand beim Eintritt in die Schleife entsteht, wird im Fall von invariantem Ausrollen eine Mindestgröße vorausgesetzt.

4.4 Implementierung

Zunächst wird für eine gegebene Schleife geprüft, ob es sich dabei um eine einfache Zählschleife handelt. Dazu wird sichergestellt, dass nur eine Steuerflusskante existiert, die aus der Schleife heraus führt. Diese kann bis zu dem Knoten, der die Schleifenbedingung auswertet, zurückverfolgt werden.

Für den Fall, bei dem alle Schleifenvariablen konstant sind, wird auf die in Abbildung 4.3 und Abbildung 4.4 dargestellten Muster überprüft. Hiermit wird gefordert,

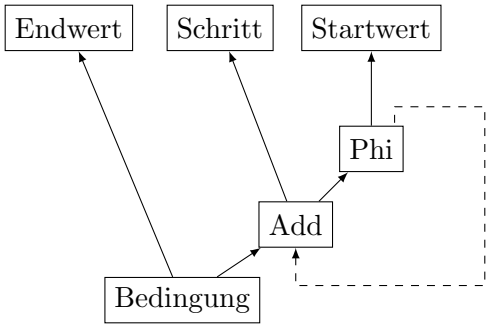


Abbildung 4.3: Fall 1: Die Bedingung verwendet den aktuellen Wert

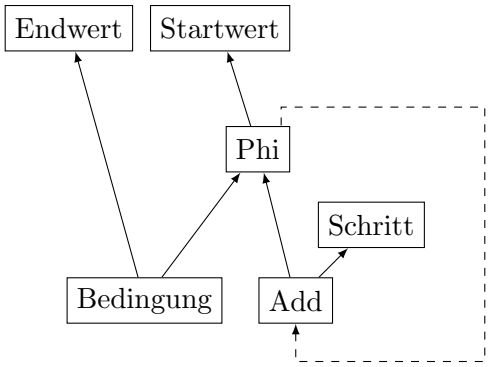


Abbildung 4.4: Fall 2: Die Bedingung verwendet den zuvor aktuellen Wert

dass zu der Induktionsvariable genau einmal ein Wert innerhalb der Schleife addiert[3] wird. Die beiden Muster unterscheiden sich darin, ob die Schleifenbedingung gegen den aktuellen Variablenwert oder gegen den zuvor aktuellen verglichen wird. Liegt der zweite Fall vor, so erfolgt eine Iteration mehr als im ersten Fall.

Zudem beeinflusst der Vergleichsoperator der Bedingung die Anzahl der Iterationen C. Die Bedingung wird für die Berechnung normalisiert, sodass bei der Auswertung zu wahr die Schleife nicht verlassen wird. Wird neben $<$ und $>$ auch auf Gleichheit geprüft, so steigt die Anzahl der Iterationen um eins. Damit kann nun die Gesamtzahl der Iterationen C bestimmt werden.

C wird auf Teilbarkeit durch den Ausrollfaktor F untersucht, mit der Bedingung, dass die F-fach ausgerollte Schleife unter dem Knotenlimit bleibt. Hierbei wird der größtmögliche Ausrollfaktor verwendet.

Der Schleifenkörper wird – analog zur Bedingungskette bei der Invertierung – komplett mehrfach kopiert. Dazu wird dieser von dem kopierenden Walker für jede Kopie einzeln durchlaufen. Dabei wird jede Kopie mit dem Originalknoten verknüpft, sodass unter Angabe von Originalknoten und Index die Kopie erhalten werden kann. Die einzigen Eingänge, die nun neu verbunden werden müssen, sind die der Schleifenköpfe. Die Steuerflusskanten des Schleifenkopfs der *Schleife 2* aus Abbildung 4.5

[3]Subtraktion, oder ein negativer Wert für Schritt, sind ebenfalls möglich. Im Folgenden werden diese Fälle nicht weiter erwähnt, benötigen jedoch zusätzliche Aufmerksamkeit in den Berechnungen.

werden mit dem Ziel der zugehörigen Kanten des Schleifenkopfs von Schleife 1[4] verbunden. Dabei wird die Schleifenbedingung durch einen Sprungbefehl ersetzt. Darauffolgende Optimierungen entfernen möglicherweise überflüssig gewordene Berechnungen und verschmelzen die Grundblöcke.

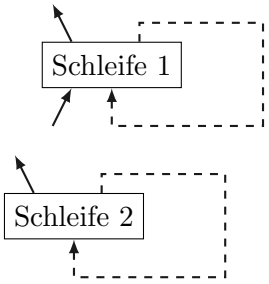


Abbildung 4.5: Kopien der Schleife vor dem Neuverbinden

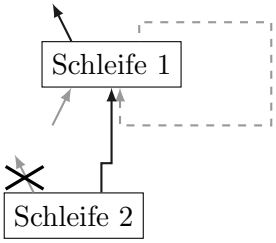


Abbildung 4.6: Neuverbinden der kopierten Schleifen am Beispiel von Schleife 2

Da alle Schleifenvarianten Werte einen ϕ-Knoten im Schleifenkopf benötigen, kann mit allen Eingängen der ϕ-Knoten der Schleifenköpfe analog zu den Steuerflusskanten verfahren werden. Das Ergebnis ist in Abbildung 4.6 dargestellt. Jede ausgerollte Schleife wird auf diese Weise mit ihrem Vorgänger verbunden. Ausnahmen sind die Schleife 1 und F. Schleife 1 behält die Kante für den Schleifeneintritt, und Schleife F behält die Schleifenbedingung. Abbildung 4.7 zeigt den letzten Schritt, der die Ausgangskante und die Rückwärtskante mit der Schleifenbedingung verbindet.

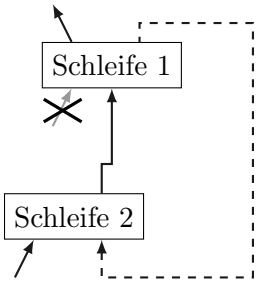


Abbildung 4.7: Verbinden der Schleife 1 und letzten Schleife

Wurde die Analyse der Schleife abgebrochen, da keine konstanten Schleifenvariablen gefunden wurden, wird mit demselben Muster auf schleifeninvariante Werte von Startwert und Endwert geprüft.

[4]Die Schleife 1 ist die Originalschleife, und besitzt deshalb eine Ausgangskante.

4.4.1 Invariantes Ausrollen

Die Schleifeninvarianz ist für einen Wert gegeben, wenn die Berechnung außerhalb der Schleife erfolgt. Falls es sich bei der Zuweisung um einen im Schleifenkopf befindlichen ϕ-Knoten handelt, müssen die zu den schleifeneigenen Rückwärtskanten gehören Eingänge des ϕ-Knotensreflexiv sein. Dies ist gleichbedeutend mit: Jede Schleifeniteration ändert den Wert nicht.

Für diesen Fall des Ausrollens wird eine zusätzliche Bedingung gefordert: Es wird eine endgesteuerte Schleife vorausgesetzt. Ziel ist es, die schon vorhandene Schleifenbedingung zu verwenden, da Duff's Device nicht prüft, ob die Schleife betreten wird. Zudem wäre für Duff's Device eine eigene Zählvariable nötig. Da die bestehende Zählvariable oftmals innerhalb der Schleife Verwendung findet, würde sie schließlich nicht wegfallen. Da die Schleifeninversion vor dem Ausrollen stattfindet, sind die Kandidaten für das Ausrollen bereits invertiert.

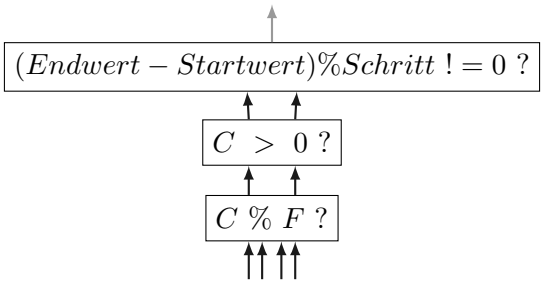


Abbildung 4.8: Grundblöcke für Duff's Device bei invariantem Ausrollen

Zur Laufzeit muss nun die Anzahl der Iterationen bestimmt werden. Da hierbei verschiedene Bedingungen überprüft werden müssen, unterteilt sich der Duff's Device Block in 3 Grundblöcke. Im 1. Grundblock wird – wie zuvor – die Anzahl der Iterationen $C_1 = (Endwert - Startwert)$ / $Schritt$ berechnet, nun jedoch zur Laufzeit. Um hierbei eine Division durch Null zu vermeiden, wird für $Schritt$ ein konstanter Wert gefordert. Da die Schleife endgesteuert ist, wird sie für den Fall, dass $B_1 : (Endwert - Startwert)\ \%Schritt \neq 0$ ist, einmal mehr durchlaufen, als ihre Bedingung zu „wahr“ auswertet. Dies wird im ersten Grundblock überprüft. Weiterhin ist zur Übersetzungszeit bekannt, ob gegen den aktuellen Wert der Zählvariablen verglichen wird (Abbildung 4.3 und Abbildung 4.4). Wird der aktuelle Wert verwendet (Fall 1), so wird – je nach Auswertung von B_1 – $C_2 = C_1 + [0/1]$ berechnet. Bei Fall 2 erfolgt eine Iteration mehr, sodass stattdessen $C_2 = C_1 + [1/2]$ verwendet wird.

Im 2. Grundblock erfolgt die Überprüfung, auf $C > 0$. Falls der Wert von C nicht zulässig ist, wird $C = 1$ gesetzt. Somit wird der Schleifenkörper genau einmal ausgeführt, und die Schleifenbedingung wertet zu „falsch“ aus.

Die Auswertung, mit welchem Schleifenkörper der Schleifeneintritt erfolgt, befindet sich im 3. Grundblock.

Das Neuverbinden der Grundblöcke unterscheidet sich gegenüber dem konstanten Fall darin, dass die kopierten Schleifenköpfe einen Eingang aus dem Duff's Device

Grundblock erhält. Da der erste Duff's Device Grundblock die Funktion des ursprünglichen Schleifenkopfs übernimmt, wurden hierfür der Schleifenkopf und die ϕ-Knoten kopiert, und dabei die schleifeneigenen Rückwärtskanten entfernt. Dadurch besitzt der Duff's Device Block bereits die korrekten Eingänge, um als Ersatz für den Schleifenkopf zu dienen. Die ϕ-Knoten der Schleifenköpfe werden auf 2 Eingänge reduziert. Einer von ihnen ist der kopierte ϕ-Knoten im ersten Duff's Device Grundblock, der andere wird – analog zum konstanten Fall – mit der Vorgängerschleife verbunden.

5 Evaluierung

Der SPEC2000 Benchmark wurde mit verschiedenen Parametern der Schleifenoptimierung übersetzt. Anschließend wurde die Ausführungszeit auf einem Testrechner mit einem 2.6GHz Pentium Dual-Core Prozessor gemessen. Soweit nicht anders gekennzeichnet, wurden bei den Messungen ausschließlich Schleifen bearbeitet, die keinen Funktionsaufruf beinhalten.

5.1 Schleifeninversion

	Ausführungszeit							
		Maximale Anzahl Knoten der Schleife						
Testprogramm	–	25	50	100	250	500	750	1000
gzip	103.49	103.43	103.18	103.13	101.96	101.90	101.95	101.81
vpr	94.44	94.07	96.88	93.18	94.27	94.53	95.09	93.41
gcc	48.11	48.31	48.88	49.26	49.42	49.13	49.07	49.17
mcf	100.43	97.94	98.91	98.98	98.16	98.52	100.07	99.67
crafty	52.90	53.41	52.76	52.38	52.52	53.33	52.71	52.77
parser	124.12	124.16	123.84	124.99	124.83	125.01	124.75	124.69
perlbmk	83.90	83.92	82.69	82.50	84.00	84.30	84.87	84.39
gap	58.44	58.81	58.98	58.75	59.41	58.57	58.88	58.61
vortex	109.87	109.83	110.14	109.88	109.87	110.02	109.95	109.94
bzip2	96.42	95.83	96.78	97.32	98.72	96.96	96.92	97.84
twolf	109.06	112.24	113.03	109.90	116.20	112.66	113.95	112.16
Summe	981.18	981.95	986.07	980.27	989.36	984.93	988.21	984.46

Tabelle 5.1: Vergleich der Auswirkungen verschiedener Knotenlimits bei der Schleifeninversion. Bedingungskette maximal 20% des Knotenlimits

Wie Tabelle 5.1 zeigt, führt die Veränderung des Knotenlimits zu einer leichten Beschleunigung einzelner Testprogramme. In der Summe überwiegen jedoch die Programme deren Performanz gesunken ist.

In Tabelle 5.2, Tabelle 5.3 und Tabelle 5.4 sind die Ergebnisse verschiedener Adaptionsfaktoren in Kombination mit unterschiedlichen Werten für die maximale Größe der Bedingungskette dargestellt. Der Adaptionsfaktor ist ein Prozentsatz, der mit der Verschachtelungstiefe potenziert wird. Das Knotenlimit in verschachtelten Schleifen wird erhöht oder verringert, indem dieser Wert mit der maximalen Knotenanzahl der

	Ausführungszeit							
		Maximale Anzahl Knoten der Schleife						
Testprogramm	–	0	-50	-25	-10	10	25	50
gzip	103.49	103.01	103.33	102.97	102.94	103.63	103.01	103.30
vpr	94.44	95.75	96.43	93.38	94.15	92.95	93.79	92.88
gcc	48.11	49.37	49.01	49.27	48.79	49.36	49.16	48.91
mcf	100.43	99.68	97.54	99.30	99.45	99.28	97.69	97.53
crafty	52.90	52.42	52.86	52.59	52.27	52.48	52.35	52.40
parser	124.12	126.28	124.68	124.28	124.38	124.75	124.87	124.96
perlbmk	83.90	82.66	82.97	83.24	81.87	82.29	84.59	82.37
gap	58.44	58.82	58.82	58.89	58.71	60.02	59.97	60.18
vortex	109.87	109.95	109.91	110.02	109.90	110.04	109.99	110.45
bzip2	96.42	99.90	96.26	96.30	96.86	97.03	98.13	96.73
twolf	109.06	112.53	112.38	113.19	115.63	111.40	115.82	111.46
Summe	981.18	990.37	984.19	983.43	984.95	983.23	989.37	981.17

Tabelle 5.2: Vergleich der Auswirkungen verschiedener Adaptionsfaktoren für die Schleifeninversion. Bedingungskette max. 20% des Knotenlimits von 100 Knoten

	Ausführungszeit							
		Maximale Anzahl Knoten der Schleife						
Testprogramm	–	0	-50	-25	-10	10	25	50
gzip	103.49	102.93	103.37	102.93	102.94	103.01	103.03	103.04
vpr	94.44	94.27	93.53	95.07	93.99	93.20	93.55	92.99
gcc	48.11	49.05	48.88	48.47	48.97	48.96	49.07	48.99
mcf	100.43	97.87	98.94	98.02	97.09	97.70	99.70	98.84
crafty	52.90	52.36	52.61	52.74	52.57	52.29	52.33	52.34
parser	124.12	124.87	124.65	124.60	124.37	124.69	124.79	124.85
perlbmk	83.90	82.32	83.16	82.97	82.37	83.03	82.74	83.01
gap	58.44	58.45	58.51	58.72	58.34	58.31	58.29	58.63
vortex	109.87	110.19	110.11	110.08	110.12	109.88	109.97	110.53
bzip2	96.42	97.46	95.70	95.91	96.30	96.90	96.72	96.46
twolf	109.06	112.31	113.58	111.41	111.49	113.51	112.43	115.11
Summe	981.18	982.08	983.04	980.92	978.55	981.48	982.62	984.79

Tabelle 5.3: Vergleich der Auswirkungen verschiedener Adaptionsfaktoren für die Schleifeninversion. Bedingungskette max. 60% des Knotenlimits von 100 Knoten

Schleife multipliziert wird. Zumeist resultiert ein Performanzverlust, nur in wenigen Fällen werden einzelne Testprogramme leicht beschleunigt.

Mit dem Ziel, die negativen Auswirkungen auf verschachtelte Schleifen zu reduzieren

	Ausführungszeit							
		Maximale Anzahl Knoten der Schleife						
Testprogramm	–	0	-50	-25	-10	10	25	50
gzip	103.49	102.97	103.31	102.94	102.95	102.96	102.99	103.01
vpr	94.44	93.65	96.30	93.69	93.18	91.58	95.34	94.69
gcc	48.11	49.04	49.01	48.51	48.85	48.96	49.08	49.20
mcf	100.43	99.11	99.13	98.49	98.73	98.00	96.81	101.18
crafty	52.90	52.64	52.59	52.98	52.40	53.02	52.38	52.48
parser	124.12	124.77	124.68	124.88	124.49	124.92	124.57	125.55
perlbmk	83.90	82.51	84.81	82.31	82.64	82.97	83.76	82.50
gap	58.44	58.22	58.44	58.82	58.44	59.07	58.23	58.38
vortex	109.87	110.20	109.97	109.59	110.23	110.35	110.14	110.19
bzip2	96.42	95.74	95.79	96.59	95.50	96.85	96.94	97.60
twolf	109.06	117.02	115.39	114.36	110.78	110.73	113.71	110.79
Summe	981.18	985.87	989.42	983.16	978.19	979.41	983.95	985.57

Tabelle 5.4: Vergleich der Auswirkungen verschiedener Adaptionsfaktoren für die Schleifeninversion. Bedingungskette maximal 80% des Knotenlimits von 100 Knoten

	Ausführungszeit							
		Maximale Anzahl Knoten der Schleife						
Testprogramm	–	50	100	200	400	600	800	1000
gzip	103.49	103.52	103.64	102.82	101.90	101.69	101.59	101.67
vpr	94.44	96.77	97.15	96.36	94.41	93.41	93.42	93.13
gcc	48.11	49.62	49.53	49.90	50.04	48.98	48.80	49.07
mcf	100.43	97.28	98.15	97.43	98.44	98.13	98.56	98.60
crafty	52.90	51.99	51.98	51.91	51.64	51.86	52.36	51.75
parser	124.12	112.87	113.08	112.83	112.52	112.49	112.59	112.54
perlbmk	83.90	85.49	85.19	82.62	83.84	83.27	82.60	82.87
gap	58.44	58.97	59.17	60.80	59.01	61.03	59.36	60.90
vortex	109.87	97.80	97.76	97.45	98.06	97.53	97.37	98.05
bzip2	96.42	95.06	95.45	94.88	96.96	97.35	97.98	97.26
twolf	109.06	108.38	113.22	113.84	111.78	114.49	110.38	112.15
Summe	981.18	957.75	964.32	960.84	958.6	960.23	955.01	957.99

Tabelle 5.5: Vergleich der Auswirkungen verschiedener Knotenlimits für die Schleifeninversion bei kleiner Bedingungskette von maximal 5% des Knotenlimits von 100 Knoten

und somit möglichst keines der Testprogramme zu verlangsamen, wurde für die folgende Messung nur eine kleine Bedingungskette zugelassen und ein Adaptionsfaktor von -50% gewählt, Wie die Ergebnisse in Tabelle 5.5 zeigen, führt dies zu einer

deutlichen Beschleunigung von *parser* und *vortex*.

5.2 Schleifenausrollen

Die Ausführungszeiten wurde für konstantes Ausrollen, sowie für die Kombination von konstantem mit invariantem Ausrollen gemessen.

	Ausführungszeit						
			Knotenlimit der ausgerollten Schleife				
Testprogramm	–	Inversion	100	200	400	600	1000
gzip	103.49	103.01	102.52	102.25	102.24	102.33	102.48
vpr	94.44	95.75	95.45	93.58	93.98	93.59	95.96
gcc	48.11	49.37	49.99	49.87	50.00	49.93	49.81
mcf	100.43	99.68	99.12	98.65	98.44	97.98	98.75
crafty	52.90	52.42	51.69	51.74	51.69	51.70	51.81
parser	124.12	126.28	112.76	112.62	112.82	112.68	112.60
perlbmk	83.90	82.66	83.99	82.62	83.40	83.13	83.69
gap	58.44	58.82	60.90	60.96	61.01	60.97	61.07
vortex	109.87	109.95	97.72	97.66	97.53	97.64	98.12
bzip2	96.42	99.90	93.73	93.74	93.62	93.63	93.86
twolf	109.06	112.53	114.09	110.66	116.70	112.56	109.88
Summe	981.18	990.37	961.96	954.35	961.43	956.14	958.03

Tabelle 5.6: Laufzeiten für verschiedene Knotenlimits der konstant ausgerollten Schleife mit vorgeschalteter Inversion. Bedingungskette max. 20%, Inversion mit max. 100 Knoten

In Tabelle 5.6 werden die Auswirkungen von konstantem Ausrollen untersucht. Da das Ausrollen auf invertierte Schleifen aufbaut, ist zum Vergleich die Laufzeit nach der Inversion ebenfalls in der Tabelle vorhanden. Obwohl die Inversion mit den gegebenen Parametern zu einer Verschlechterung führt, wird dies durch das Ausrollen kompensiert und resultiert – verglichen mit der ursprünglichen Laufzeit – in einer Laufzeitreduzierung. Im Fall von einem Knotenmaximum von 400 Knoten in der ausgerollten Schleife tritt nur bei 3 der Testprogramme eine geringer Verlust der Performanz ein.

Wird konstantes Ausrollen auf die Inversion mit den Parametern aus Tabelle 5.1 angewendet, so führt dies – wie in Tabelle 5.7 zu sehen ist – in der Summe zu einer Verschlechterung gegenüber der alleinigen Inversion. Dies legt nahe, Parameter für Inversion und Ausrollen einzeln, sowie für die Kombination aus beiden bereitzustellen.

Wird zusätzlich invariant ausgerollt, so ergeben sich die Laufzeiten aus Tabelle 5.8. Die Tabelle zeigt, dass dies ausschließlich zu Reduktion der Performanz führt. Möglicherweise müssen für die Schleifenvariablen strengere Voraussetzungen gefordert

	Ausführungszeit						
			Knotenlimit der ausgerollten Schleife				
Testprogramm	–	Inversion	200	400	600	800	1000
gzip	103.49	101.59	101.71	101.75	101.66	101.93	101.78
vpr	94.44	93.42	93.20	95.59	93.53	93.06	93.32
gcc	48.11	48.80	48.85	48.87	48.76	48.86	48.75
mcf	100.43	98.56	97.95	98.89	98.88	98.78	98.90
crafty	52.90	52.36	51.74	51.75	52.59	52.58	51.81
parser	124.12	112.59	113.07	113.05	112.95	112.99	112.93
perlbmk	83.90	82.60	83.30	83.26	82.74	82.78	82.66
gap	58.44	59.36	60.83	60.83	61.08	60.68	60.79
vortex	109.87	97.37	97.63	97.15	97.71	97.43	97.60
bzip2	96.42	97.98	96.49	96.38	96.26	96.42	96.40
twolf	109.06	110.38	112.37	110.09	114.08	113.08	113.13
Summe	981.18	955.01	957.14	957.61	960.24	958.59	958.07

Tabelle 5.7: Ergebnisse von konstantem Ausrollen mit vorheriger Inversion. Bedingungskette max. 5%, Inversion mit max. 800 Knoten, Adaptionswert -50%

	Ausführungszeit						
			Knotenlimit der ausgerollten Schleife				
Testprogramm	–	Inversion	200	400	600	800	1000
gzip	103.49	101.59	103.04	101.92	101.91	101.87	101.95
vpr	94.44	93.42	93.03	95.53	95.60	95.73	95.38
gcc	48.11	48.80	49.07	49.26	49.08	49.31	49.17
mcf	100.43	98.56	98.80	98.81	98.40	96.04	99.15
crafty	52.90	52.36	52.53	53.06	52.76	52.60	52.70
parser	124.12	112.59	124.94	125.00	124.96	125.55	125.80
perlbmk	83.90	82.60	82.31	84.48	84.94	85.36	84.30
gap	58.44	59.36	58.31	59.58	58.58	58.53	58.46
vortex	109.87	97.37	110.24	109.85	109.72	109.68	110.13
bzip2	96.42	97.98	96.58	97.17	96.62	98.18	97.22
twolf	109.06	110.38	112.66	113.58	114.20	110.95	113.04
Summe	981.18	955.01	981.51	988.24	986.77	983.8	987.3

Tabelle 5.8: Ergebnisse von konstantem und invariantem Ausrollen mit vorheriger Inversion. Mindestanzahl von 50 Knoten für auszurollende Schleifen, Bedingungskette max. 5%, Inversion mit max. 800 Knoten, Adaptionswert -50%

werden, sodass die zusätzlichen Bedingungen zur Laufzeit wegfallen. Es sind verschiedene Sonderfälle denkbar, bei denen beispielsweise ausschließlich der Endwert

invariant ist, oder durch die Analyse der Schleifenvariablen ausgeschlossen werden kann, dass die Iterationsanzahl negativ sein kann.

5.3 Zusammenfassung und Ausblick

Das Invertieren von Schleifen kann auch auf modernen Prozessoren zu Geschwindigkeitsvorteilen führen, da nicht nur ein Sprungbefehl eingespart wird, sondern die neue Struktur der Schleife weitere Optimierungen erlaubt. Dabei ist zu beachten, dass bei verschachtelten Schleifen der Einfluss auf die sie umgebenden Schleifen möglichst gering ausfällt, um nicht zu Geschwindigkeitsnachteilen zu führen. Dies kann erreicht werden, indem der zu kopierende Anteil klein gehalten wird, und mit steigender Verschachtelungstiefe, die maximale Größe der zu behandelnden Schleifen reduziert wird.

Das Ausrollen von Schleifen zeigt sein Potential oftmals nur in Kombination mit anderen Optimierungen. Ein Ausrollen, ohne dass eine Optimierbarkeit des Schleifenkörpers besteht, sorgt in zu vielen Fällen zu Geschwindigkeitseinbußen.

Einige Programme, wie beispielswiese *twolf*, wurden fast immer verlangsamt. Das Knotenlimit ist in diesen Fällen zu statisch oder zu ungenau. Der Adaptionsfaktor passt das Knotenlimit and die Verschachtelungstiefe an, betrachtet dabei jedoch nicht die Knotenanzahl der umgebenden Schleifen. Da insbesondere bei *twolf* viele Berechnungen in inneren Schleifen durchgeführt werden, deutet dies darauf hin, dass die hier verwendeten Parameter auf diese Konfiguration einen zu großen negativen Einfluss haben. Eine Heuristik, die zusätzlich die Größe der umgebenden Schleife mit einbezieht, könnte dies verhindern.

Die Größenabschätzung mit Hilfe der Knotenzahl ist recht ungenau. Einerseits existieren Knoten, die letztlich in einer Verzweigung resultieren. Andererseits wird eine Verzweigung zur Berechnung des Absolutwertes einer Zahl vom Backend erkannt und, falls möglich, in eine Instruktion übersetzt. Die Genauigkeit der Größenabschätzung könnte somit durch das Backend unterstützt werden, indem jedem Knoten ein Gewicht zugeordnet wird.

Um die Meßergebnisse besser deuten zu können, wären Statistiken über die Schleifen innerhalb der Benchmarkprogramme hilfreich. Somit könnte das Vorhandensein und die Größe von verschachtelten Schleifen und deren Auswirkungen besser untersucht werden.

6 Verwandte Arbeiten

Weiss et al. [WS87] stellt in zahlreichen Messungen vor, wie sich Größe des Instruktionscaches, sowie die Anzahl der Register, zusammen mit dem Ausrollfaktor auf die Laufzeit auswirken. Unter bestimmten Bedingungen erhält er somit ein Speedup von 1.62 und demonstriert somit das Potential des Schleifenausrollens.

Davidson et al. [DJ95] evaluiert einen Ansatz, der die parallele Abarbeitung auf der Instruktionsebene optimiert. Indem Mehrdeutigkeiten bei Speicherzugriffen ausgeschlossen werden, können Schleifen aggressiv ausgerollt werden, sodass ein optimistisches Scheduling für Lade- und Speicher-Instruktionen ermöglicht wird. Dadurch entfallen Leerlaufzeiten auf Kosten der Größe des Maschinencodes da oftmals die Originalschleife als Rückfalllösung verwendet wird. Die Untersuchung des Parallelismus auf Instruktionsebene zeigt, dass das Ausrollen erst im Zusammenhang mit Registerumbenennung und aggressiv optimierten Schleifen deutliche Vorteile bietet.

Huang und Leng [HL97] stellen eine Möglichkeit vor, statt ausschließlich For-Schleifen, allgemeine Schleifen auszurollen, indem die Bedingung für das Ausführen der ausgerollten Schleife zur Kompilierzeit bestimmt wird. Wie bei Davidson et al. wird eine zweite Schleife als Rückfalllösung verwendet.

Statt Heuristiken für das Schleifenschälen zu verwenden, berechnen Song und Kavi [SK02] für die in der Schleife befindlichen Variablen, wie oft die Schleife geschält werden muss, damit eine bestimmte Variable invariant ist. Dies kann Verzweigungen innerhalb der Schleife reduzieren und somit die parallele Abarbeitung verbessern.

Keines der Verfahren aus den hier genannten Arbeiten wird auf Graphen angewandt. Oftmals wird eine hypothetische Maschine zur Evaluation verwendet oder gezielt die zu optimierenden Schleifen gewählt. Dies zeigt das Potential der jeweiligen Optimierungen auf und ermöglicht eine exakte Untersuchung, es erlaubt jedoch oftmals nicht das Ergebnis auf allgemeine Programme zu übertragen.

Literaturverzeichnis

[CFR+91] Cytron, Ron ; Ferrante, Jeanne ; Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Transactions on Programming Languages and Systems* 13 (1991), Oct, Nr. 4, 451–490. http://doi.acm.org/10.1145/115372.115320

[DJ95] Davidson, Jack W. ; Jinturkar, Sanjay: Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation. In: *In Proceedings of the 28th annual international symposium on Microarchitecture*, IEEE Computer Society, 1995, S. 125–132

[Fei79] Feign, David: A note on loop "optimization". In: *SIGPLAN Not.* 14 (1979), Nr. 11, S. 23–25. http://dx.doi.org/10.1145/988056.988060. – DOI 10.1145/988056.988060. – ISSN 0362–1340

[HL97] Huang, J. C. ; Leng, T.: Generalized Loop-Unrolling: a Method for Program Speed-Up. In: *in Proc. IEEE Symp. on Application-Specific Systems and Software Engineering and Technology*, 1997, S. 244–248

[LBBG05] Lindenmaier, Götz ; Beck, Michael ; Boesler, Boris ; Geiss, Rubino: Firm, an Intermediate Language for Compiler Research. Version: 3 2005. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000003172. University of Karlsruhe, 3 2005 (2005-8). – Forschungsbericht. – 19 S.

[Lin02] Lindenmaier, Götz: libFIRM – A Library for Compiler Optimization Research Implementing FIRM. Version: Sep 2002. http://www.info.uni-karlsruhe.de/papers/Lind_02-firm_tutorial.ps. Universität Karlsruhe, Fakultät für Informatik, Sep 2002 (2002-5). – Forschungsbericht. – 75 S.

[Mor98] Morgan, C. R.: *Building an Optimizing Compiler*. Butterworth, 1998. – 246–248 S. – ISBN 155558179X

[RWZ88] Rosen, B. K. ; Wegman, M. N. ; Zadeck, F. K.: Global value numbers and redundant computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1988, 12–27

[SK02] Song, Litong ; Kavi, Krishna M.: *A Technique for Variable Dependence Driven Loop Peeling.* 2002

[TLB99] Trapp, Martin ; Lindenmaier, Götz ; Boesler, Boris: Documentation of the Intermediate Representation FIRM / Universität Karls-

ruhe, Fakultät für Informatik. Version: Dec 1999. `http://www.info.uni-karlsruhe.de/papers/firmdoc.ps.gz`. Universität Karlsruhe, Fakultät für Informatik, Dec 1999 (1999-14). – Forschungsbericht. – 0–40 S.

[WS87] Weiss, Shlomo ; Smith, James E.: A study of scalar compilation techniques for pipelined supercomputers. In: *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1987. – ISBN 0818608056, S. 105–109