ÜBERSETZERBAU

Manfred Brockhaus
Anton Ertl
Andreas Krall

Institut für Computersprachen
Abteilung für Übersetzerbau

Kapitel 1

Optimierungen

Optimierungen transformieren ein Programm in ein äquivalentes, effizienteres Programm. Äquivalent bedeutet, daß sich das optimierte Programm bei allen Eingabedaten genauso verhält, wie das nicht optimierte Programm. Um die Korrektheit der Transformationen zu garantieren und die dafür notwendigen Informationen zur Verfügung zu stellen, müssen semantische Programmanalysen durchgeführt werden.

Bei Optimierungen unterscheidet man weiters zwischen maschinenabhängigen und maschinenunabhängigen Optimierungen. Die meisten Optimierungen bringen auf allen Architekturen einen Effizienzgewinn. Diese werden auf der Zwischendarstellung (abstrakter Syntaxbaum oder Quadrupelcode) durchgeführt. Maschinenabhängige Optimierungen werden erst nach oder während der Codeerzeugung durchgeführt und arbeiten meist auf einer abstrakten Darstellung des Maschinencodes (Quadrupelcode). Viele Optimierungen kann man parameterisieren, um Maschinenabhängigkeiten portabel zu implementieren (z.B. Cachegröße oder Registeranzahl). Zum besseren Verständnis sind die Beispiel in C angegeben, manche Optimierungen lassen sich aber nur auf Maschinenbefehlen durchführen.

Optimierungen können entweder lokal (innerhalb eines Grundblocks), global (innerhalb einer Funktion oder Prozedur) oder interprozedural (über das ganz Programm) wirken. Interprozedurale Optimierungen sind wesentlich aufwendiger als lokale Optimierungen, liefern aber auch bessere Ergebnisse. Skalare Optimierungen arbeiten auf einzelnen Werten, Schleifenoptimierungen arbeiten auf Arrays und Codeoptimierungen verbessern die Befehlsabarbeitung.

1.1 Analysen

Die Kontrollflußanalyse bestimmt die Ausführungsreihenfolge der Befehle. Die Datenflußanalyse analysiert den Wertefluß in einem Program. Die Abhängigkeitsanalyse stellt die Abhängigkeiten zwischen den Befehlen fest. Die Aliasanalyse überprüft, ob zwei Zeiger auf die selbe Speicherzelle zeigen können.

Während der Kontrollflußanalyse wird das Programm in Grundblöcke zerlegt und der Kontrollflußgraph aufgebaut (siehe Abschnitt **??**). Eine Schleifenanalyse (*loop analysis*) erkennt Schleifen und die Schachtelungstiefe von Schleifen. Weiters wird noch Dominanzinformation (*dominator tree*) berechnet, die angibt, welche beliebigen Grundblöcke immer vor bzw. nach einem bestimmten Grundblock ausgeführt werden. Die Schleifeninformationen werden z.B. dazu genutzt, Schätzwerte für die Ausführungshäufigkeit von Grundblöcken zu berechnen. Alternativ dazu können reale Werte gespeichert werden, die durch Testläufe des Programms berechnet werden (*feedback directed compilation*).

Viele Optimierungen benötigen Informationen, wo Werte definiert und wo sie verwendet werden (*def-use information, reaching definitions*). Diese und ähnliche Informationen lassen sich mittels Datenflußanalyse berechnen. Die iterative Datenflußanalyse durchwandert entlang des Kontrolflußgraphen mehrmals die einzelnen Grundblöcke und Befehle. Dabei werden die Informationen mittels Ein- und Ausgangsgleichungen in Mengen gesammelt, bis sich die Inhalte in diesen Mengen nicht mehr ändern. Eine weitere Analysetechnik ist die abstrakte Interpretation. Anstelle der echten Programmbefehle werden abstrakte Operationen ausgeführt, die die benötigten Informationen berechnen. Die gesammelten Informationen müssen endlich sein, um eine Terminierung der Analyse sicherzustellen.

Manche Analysen lassen sich einfacher durchführen, wenn für jede Variable maximal eine einzige Zuweisung existiert. Dazu wird das Programm in *static single assigment* (SSA) Form gebracht. Bei jeder Zuweisung an eine Variable wird eine neue Variable durch Indizierung des Variablennames angelegt. Bei der Vereinigung von Programmpfaden ist es dann möglich, daß von der selben Variablen mehrere Kopien mit unterschiedlichem Index existieren. Diese Variablen werden nun durch einen sogenannten Phi-Knoten zusammengefaßt. Ein Programm in SSA-Form kann wieder in ein übliches Programm zurücktransformiert werden, indem diese Phi-Knoten durch Kopierbefehle ersetzt werden. Unnötige Kopierbefehle können in einer weiteren Optimierungsphase eliminiert werden.

`a =`	$\texttt{a}_0$ `=`
`if (cond)`	`if (cond)`
`   a =`	`   `$\texttt{a}_1$ `=`
`else`	`else`
`   a =`	`   `$\texttt{a}_2$ `=`
	$\texttt{a}_3 = \phi(\texttt{a}_1,\texttt{a}_2)$
`= a`	`=` $\texttt{a}_3$

Beispiel 1.1: *static single assignment*

Die Datenabhängigkeitsanalyse bestimmt die Datenabhängigkeiten der Befehle. Wird diese Analyse nur innerhalb eines Grundblocks durchgeführt, so ist der resultierende Datenabhängigkeitsgraph ein zyklenfreier gerichteter Graph (siehe Abschnitt **??**). Für globale Optimierungen müssen auch Schleifen berücksichtigt werden und der resultierende Datenabhängigkeitsgraph ist ein zyklischer Graph. Für manche Optimierungen wird zusätzlich die Länge der Abhängigkeit als Anzahl der Iterationen berechnet. Alle Kontroll- und Datenabhängigkeiten können im *program dependency graph* zusammengefaßt werden.

1.2 Skalare Optimierungen

Die einfachste Optimierung ist die Auswertung von konstanten Ausdrücken (*constant folding*). Diese wird meistens noch mit einer Vereinfachung der Berechnung basierend auf den algebraischen Gesetzen (*algebraic simplification*) kombiniert. Der Ausdruck `(4 - 3) * a` kann auf die Variable `a` reduziert werden, da der konstante Ausdruck Eins ergibt und die Multiplikation mit Eins eliminiert werden kann. *Constant propagation* (Konstantenverbreitung) verteilt Konstante über ein Unterprogramm oder über das ganze Programm. Das kann wieder zu neuen Möglichkeiten für *constant folding* führen. Daher werden diese beiden Optimierungen oft mehrmals angewendet. Bei der *copy propagation* werden Kopierbefehle eliminiert, wenn der Originalwert in einer anderen Variablen zur Verfügung steht.

Strength reduction reduziert die Komplexität von arithmetischen Operationen. So kann eine Multiplikation mit Zwei durch eine Addition, eine Multiplikation mit 16 durch eine vierfache Schiebeoperation und eine Division mit einer Konstanten durch eine Multiplikation mit dem Reziprokwert ersetzt werden. Manchmal wird auch die Vereinfachung von Addressausdrücken als *strength reduction* bezeichnet (siehe Abschnitt 1.4).

Eine wichtige Optimierung ist die Vermeidung von Doppelberechnungen. Die Elimination gemeinsamer Teilausdrücke (*common subexpression elimination*) entfernt solche Doppelberechnungen. Oft treten solche Doppelberechnungen nur auf manchen Pfaden durch das Programm auf. Eine Verallgemeinerung dieser Optimierung ist die *partial redundancy elimination*. Das Beispiel 1.2 zeigt, wie durch die Entfernung des Befehls im `else`-Zweig die Doppelberechnung vermieden werden kann.

<pre>if (cond)
   a = b;
else
   x = a + c;
x = a + c;</pre>
	<pre>if (cond)
	   a = b;
	x = a + c;</pre>

Beispiel 1.2: *partial redundancy elimination*

1.3 Codeoptimierungen

Konstante Ausdrücke in Bedingungen können dazu führen, daß Teile des Programms nicht mehr ausgeführt werden. Diese Teile können dann entfernt werden (*dead code elimination*). Grundblöcke können in beliebiger Reihenfolge angeordnet werden. Daher werden sie üblicherweise so angeordnet, daß die Anzahl der Sprungbefehle minimiert wird. Dabei werden teilweise die Sprungbedingungen invertiert.

Ein Unterprogrammaufruf ist recht aufwendig. Kurze Unterprogramme können direkt an die Aufrufstelle kopiert werden, um die Effizienz zu steigern (*function oder *method inlining*). Bei objektorientierten Programmiersprachen muß bei virtuellen Methodenaufrufen sichergestellt sein, daß nur eine einzige Methode aufgerufen werden kann. In eingebetteten Systemen ist Programmspeicher sehr begrenzt. *Procedural abstraction* sucht idente Programmteile und ersetzt sie durch einen Unterprogrammaufruf. *Procedural abstraction* ist damit die inverse Optimierung zu *inlining*. Rekursive Unterprogrammaufrufe können durch Schleifen ersetzt werden, wenn der Unterprogrammaufruf am Ende des Unterprogramms steht (*tail recursion elimination*). Beispiel 1.3 zeigt wie die rekursive Funktion zur Fakultätsberechnung in eine Schleife transformiert werden kann.

```
int fak(int n) {
if (n == 1)
   return n;
return n * fak(n -1);
}
```

```
int fak(int n) {
int f;
for (f = 1; n > 1; n-- )
   f = f * n;
return f;
}
```

Beispiel 1.3: Rekursionseliminierung

Eine der einfachsten Optimierungen ist die *peephole* (Guckloch) Optimierung. Dabei werden mehrere aufeinanderfolgende Befehle auf Optimierungsmöglichkeiten untersucht. Z.B. erzeugt ein einfacher Codegenerator mit einer einfachen Registerzuteilung manchmal Befehlssequenzen, wo ein Register in den Speicher geschrieben wird und direkt danach der Wert wieder in das selbe Register geladen wird. In diesem Fall kann der Ladebefehl eliminiert werden.

1.4 Array- und Schleifenoptimierungen

Der überwiegende Teil der Rechenzeit wird normalerweise in Schleifen verbraucht. Daher zählen Schleifenoptimierungen zu den wichtigsten Optimierungen. Diese werden sowohl für klassische Architekturen wie Mikroprozessoren, als auch für Vektorrechner oder parallele Systeme benötigt. Das Ziel dabei ist, sowohl die Anzahl der ausgeführten Operationen zu reduzieren, als auch die Speicherzugriffe so zu gestalten, daß Caches oder Vektorregister effizient genützt werden können.

Ein typisches Beispiel für eine Schleifenoptimierung ist die *loop invariant code motion*, wo Berechnungen, die sich innerhalb einer Schleife nicht verändern, aus der Schleife hinaus verschoben werden. Besonders häufig treten solche Berechnungen implizit bei Adressberechnungen auf.

Weitere Optimierungen sind eine Reihe von Schleifentransformationen, die geschachtelte Schleifen aufspalten, einzelne Schleifen kombinieren, Schleifen vertauschen oder die Zugriffsreihenfolge auf einzelne Zellen von Arrays verändern. Diese Optimierungen haben teilweise inverse Transformationen und müssen daher abhängig vom Optimierungsziel eingesetzt werden. Alle diese Optimierungen sind nur anwendbar, wenn sie die meistens zyklischen Datenabhängigkeiten berücksichtigen.

Elimination der Induktionsvariablen

Die Multiplikationen, die bei der Adressberechnung bei Zugriffen auf mehrdimensionale Arrays auftreten, sind aufwendige Operationen. Daher ist eine der wichtigsten Schleifenoptimierungen die Elimination der Induktionsvariablen (*induction variable elimination*) mit gleichzeitiger Vereinfachung der Adressberechnung (*strength reduction*). Im Bsp. 1.4 wird ein zweidimensionales Array mit 0 initialisiert.

```
for (i = 0; i < N; i++)
   for (j = 0; j < M; j++)
      a[i][j] = 0;
```

Beispiel 1.4: Beispielschleife

Für den doppelt indizierten Zugriff muß der erste Index mit der Größe der Dimension multipliziert werden, der zweite Index muß aufaddiert werden und (implizit im C Code) dieser Wert noch mit der Größe eines Arrayelements multipliziert werden, bevor er zur Anfangsadresse des Arrays aufaddiert werden kann (siehe Bsp. 1.5).

```
for (i = 0; i < N; i++)           | aptr = a; eptr = a + N * M;
   for (j = 0; j < M; j++)        | while (aptr < eptr)
      *(a + i * M + j) = 0;       |    *aptr++ = 0;
```

Beispiel 1.5: Elimination der Induktionsvariablen

Variablen, die in den einzelnen Schleifeniterationen mit gleichen arithmetischen Berechnungen fortschreitende Werte erhalten, heißen Induktionsvariablen. Im Beispiel 1.5 sind die Indexvariablen `i` und `j` die Induktionsvariablen. Diese werden nun entfernt und die komplexe Adressberechnung wird durch eine einfache Addition ersetzt. Die Abbruchbedingung für die Schleife wird so umgeformt, daß diese mit den Adressvariablen durchgeführt werden kann. In diesem Beispiel wird auch noch die doppelt geschachtelte Schleife auf eine einfache umgesetzt und die Matrix als linearer Speicher aufgefaßt.

Loop unrolling

Loop unrolling (Schleifenentrollen) vervielfacht den Schleifenrumpf. Dadurch wird der Schleifenverwaltungsaufwand reduziert. Im allgemeinen Fall muß vor oder nach der Schleife geprüft werden, ob die Iterationszahl durch den Vervielfachungsfaktor teilbar ist und die überzähligen Iterationen ausgeführt werden. Im Bsp. 1.6 wird angenommen, daß die Anzahl der Iterationen immer ein Vielfaches von 2 ist.

```
assert(N%2 == 0);             | for (i = 0; i < N; i+=2) {
for (i = 0; i < N; i++) {     |    a[i] = b[i];
   a[i] = b[i];               |    a[i+1] = b[i+1];
   }                          |    }
```

Beispiel 1.6: *loop unrolling*

Vektorisierung

Vektorisierung (*vectorization*) formt Schleifen mit Zugriffen auf die einzelnen Felder eines Arrays auf Vektoroperationen um. Im Bsp. 1.7 wird dabei ein Vektor durch das Paar [Untergrenze:Obergrenze] als Index eines Arrays dargestellt.

```
for (i = 0; i < N; i++)  | a[0:N-1] = b[0:N-1] + c[0:N-1];
   a[i] = b[i] + c[i];   |
```

Beispiel 1.7: *vectorization*

Parallelisierung

Parallelisierung (*parallelization, concurrentization*) teilt eine Schleife auf mehrere Prozessoren auf und führt Iterationen parallel durch. Das funktioniert allerdings nur dann, wenn keine Abhängigkeiten zwischen den Iterationen existieren. Bei verteiltem Speicher müssen auch die Daten auf die zu den Prozessoren gehörenden Speicher verteilt werden. Bsp. 1.8 zeigt die Aufteilung einer geschachtelten Schleife auf `N` Prozessoren, wobei jeder Prozessor eine Iteration der äußeren Schleife ausführt.

```
                             | for (j = 0; j < M; j++)  // CPU 0
for (i = 0; i < N; i++)      |    a[0][j] = b[0][j];
   for (j = 0; j < M; j++)   | ...
      a[i][j] = b[i][j];     | for (j = 0; j < M; j++)  // CPU N-1
                             |    a[N-1][j] = b[N-1][j];
```

Beispiel 1.8: *loop parallelization*

(*Loop Distribution, Loop Fission*)

Loop distribution, loop fission (Schleifenaufteilung oder Schleifenaufspaltung) teilt eine einzige Schleife auf mehrere einzelne Schleifen auf. Dadurch können entweder Caches besser genutzt werden oder die Vektorisierung erleichtert werden. Bsp. 1.9 zeigt eine geschachtelte Schleife, die auf zwei Schleifen aufgeteilt wurde und leichter vektorisierbar ist.

```
for (i = 0; i < N; i++) {
    a[i] = 0;
    for (j = 0; j < N; j++)
        b[i][j] = a[i];
}
```

```
for (i = 0; i < N; i++)
    a[i] = 0;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        b[i][j] = a[i];
```

Beispiel 1.9: *loop distribution*

Schleifenaustausch

Schleifenaustausch (*loop interchange*) vertauscht in geschachtelten Schleifen die äussere mit der inneren Schleife. Dadurch werden die Arrays in verschiedenen Reihenfolgen abgearbeitet. Schleifenaustausch wird angewendet, um lineare Speicherzugriffe und damit eine bessere Nutzung von Caches zu ermöglichen. Bsp. 1.10 vertauscht die beiden Schleifen und ermöglicht einen linearen Speicherzugriff.

```
for (j = 0; j < N; j++)
    for (i = 1; i < N; i++)
        a[i][j] = a[i-1][j] + b[i];
```

```
for (i = 1; i < N; i++)
    for (j = 0; j < N; j++)
        a[i][j] = a[i-1][j] + b[i];
```

Beispiel 1.10: *loop interchange*

Schleifenvereinigung

Schleifenvereinigung (*loop fusion*) vereinigt mehrere Schleifen zu einer einzigen. Dabei werden die Schleifenmehrkosten (Zähler und Sprungbefehle) reduziert. Im Bsp. 1.11 kann weiters das zweifache Laden des Wertes `a[i]` eliminiert werden.

Strip Mining (Cache Blocking)

Strip mining (Abarbeitung in Streifen) formt eine einfach geschachtelte Schleife in eine doppelt geschachtelte Schleife um. Dabei werden immer kleine Teile des Arrays

```
for (i = 0; i < N; i++)          | for (i = 0; i < N; i++) {
    a[i] = b[i];                 |     a[i] = b[i];
for (i = 0; i < N; i++)          |     c[i] = a[i] + d[i];
    c[i] = a[i] + d[i];          |     }
```

Beispiel 1.11: *loop fusion*

in Blöcken abgearbeitet. Als Blockgröße (im Bsp. 1.12 ist sie 32) wird dabei die Vektorlänge oder die Cacheblockgröße gewählt.

```
for (i = 0; i < N*32; i++) {     | for (j = 0; j < N*32; j+=32)
    a[i] = b[i] + 1;             |     for (i = j; i < j+32; i++) {
    c[i] = b[i] - 1;             |         a[i] = b[i] + 1;
    }                            |         c[i] = b[i] - 1;
                                 |         }
```

Beispiel 1.12: *strip mining*

Loop collapsing

Loop collapsing (Schleifen zusammenklappen) transformiert eine doppelt geschachtelte Schleife in eine einfach geschachtelte Schleife. Diese Transformation wird verwendet, um die nutzbare Vektorlänge zu erhöhen. Im Bsp. 1.13 wird dabei ein zweidimensionales Array durch ein eindimensionales Array ersetzt.

```
for (i = 0; i < N; i++)          | for (i = 0; i < N*N; i++)
    for (j = 0; j < N; j++)      |     a[i] = b[i];
        a[i][j] = b[i][j];       |
```

Beispiel 1.13: *loop collapsing*

Loop Peeling

Loop peeling (Schleifen schälen) entfernt Iterationen am Anfang oder am Ende einer Schleife. Das ist dann sinnvoll, wenn diese Iterationen Optimierungen wie zum Beispiel eine Vektorisierung behindern. Bsp. 1.14 zeigt eine Schleife, bei der auf eine ringförmige Datenstruktur zugegriffen wird.

```
j = N - 1;
for (i = 0; i < N; i++) {
   a[i] = b[j];
   j = i;
   }
```

```
a[0] = b[N-1];
for (i = 1; i < N; i++)
   a[i] = b[i-1];
```

Beispiel 1.14: *loop peeling*

Software Pipelining

Aktuelle Prozessoren können mehrere Befehle parallel ausführen. Manche Befehle (z.B. Ladebefehle) benötigen mehrere Zyklen. Kurze Schleifen können dann diese Prozessoren nicht ausnützen, da zu wenig Möglichkeiten für eine Befehlsumordnung vorhanden sind. Beim *software pipelining* werden mehrere Iterationen einer Schleife überlappend ausgeführt. Der Prolog startet die erste Schleifeniteration, der Epilog schließt die letzte Iteration ab. Im Beispiel 1.15 gehen wir davon aus, daß die Schleife mindestens einmal durchlaufen wird. Im Prolog wird die Variable `t` gelesen, im Epilog wird `t` gespeichert. In der Schleife kann das Lesen und Speichern gleichzeitig durchgeführt werden.

```
for (i = 0; i < N; i++) {
   t = a[i];
   b[i] = t;
   }
```

```
t = a[0];
for (i = 1; i < N; i++)
   b[i-1] = t; t = a[i];
   }
b[N-1] = t;
```

Beispiel 1.15: *software pipelining*

Literaturverzeichnis

[App98] A.W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

Das Buch beschreibt umfassend und klar die aktuellen Konzepte und Verfahren. Es werden auch objektorientierte Sprachen behandelt.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, 1986.

Dieses internationale Standardwerk bietet eine umfassende Einführung in den Übersetzerbau und eine Vertiefung vieler Themenbereiche.

[FH95] Ch. Fraser and D. Hanson. *A Retargetable C compiler: Design and Implementation.* Benjamin/Cummings Publishing, 1995.

Das Buch enthält einen kompletten ANSI C Compiler mit Codegeneratoren für Intel, SPARC und MIPS Prozessoren

[Kan89] Gerry Kane. *MIPS RISC Architecture.* Prentice Hall, 1989.

[Kas90] U. Kastens. *Übersetzerbau.* R. Oldenbourg Verlag, 1990.

Das Buch ist vergleichsweise kurz und vollständig

[Mot84] Motorola. *M68000, Programmer's Reference Manual.* Prentice Hall, 1984.

[Par92] Th.W. Parsons. *Introduction to Compiler Construction, Computer Science Press.* W.H.Freeman, 1992.

Der Verfasser legt besonderen Wert auf Klarheit und weniger auf Tiefe. Das Buch behandelt auch theoretische Grundlagen sowie die Generatoren Lex und Yacc.

[SF85] A.T. Schreiner and G. Friedman. *Compiler bauen mit UNIX - Eine Einführung.* Carl Hanser Verlag, 1985.

Mithilfe der Übersetzer-Generatoren Lex und Yacc wird ein Compiler für ein C-Subset erstellt.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1994.

Das Buch enthält einen kleinen in Prolog geschriebenen Compiler (Kapitel 24).

[Sta89] R.M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, 1989.

Das Manual beschreibt einen der besten portablen C-Compiler. Er ist unter UNIX und MS-DOS als freie Software verfügbar.

[WG92] N. Wirth and J. Gutknecht. *Project Oberon, The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

Der Oberon-Compiler und die gesamte interaktive Systemumgebung werden beschrieben. Das Buch enthält die vollständigen Programmlisten.

[Wir86] N. Wirth. *Compilerbau - Eine Einführung.* Teubner, 1986.

Es wird ein kleiner Compiler-Interpreter für ein Pascal-Subset entwickelt.

[WM92] R. Wilhelm and D. Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung.* Springer Verlag, 1992.

Das Buch enthält theoretische Grundlagen und praktische Verfahren nach dem aktuellen Stand der Technik. Auch funktionale und logische Programmiersprachen werden berücksichtigt.