Praktikum: Compilerbau - Bonusaufgabe

Betreuer: André-Marcel Hellmund
4. Semester - Sommersemester 2012

1 Organisatorisches

In diesem Semester gibt es die Möglichkeit durch das Compilerbau-Projekt Bonuspunkte für die Klausur bei Prof. Dr. Schwinn zu bekommen. Insgesamt kann jede Gruppe maximal 10 Prozentpunkte (entspricht ca. 8-9 Punkte in der Klausur) bekommen, die sich dabei folgendermaßen aufteilen:

- funktionale Anforderungen: **0 Prozentpunkte** - die funktionale Korrektheit ist Bedingung für das Bestehen des Praktikums. Bei den funktionalen Anforderungen wird auch die Fehlerfreiheit in Valgrind betrachtet.
- nicht-funktionale Anforderungen: **5 Prozentpunkte** - hierzu zählen unter anderem: Codequalität (inkl. Kommentare/Doxygen) und Qualität der Fehlermeldungen
- Bonusaufgabe: **5 Prozentpunkte**

2 Bonusaufgabe - Lokale Elimination von gemeinsamen Teilausdrücken (LCSE)

Die Bonusaufgabe besteht darin, lokale Elimination von gemeinsamen Teilausdrücken (im Folgenden als LCSE bezeichnet) auf Basis des vorgestellten Zwischencode zu implementieren. Die Aufgabenbeschreibung orientiert sich an [ALSU07] und den Folien zur Vorlesung Compilerbau. Als Grundlage der LCSE dienen zwei Datenstrukturen, die auch für andere Formen der Optimierung eine wichtige Rollen spielen: Basisblöcke (BB) und gerichtete, azyklische Graphen (im Folgenden DAG für Directed, Acyclic Graph).

2.1 Basisblock

Ein Basisblock ist eine Sequenz von Zwischencode-Instruktionen, die ohne Unterbrechung (z.B. Funktionsaufruf oder bedingter/unbedingter Sprung) ausgeführt werden können, d.h. Start der Sequenz ist immer die erste Instruktion des Blocks und Ende der Sequenz ist immer die letzte Instruktion des Blocks. In der Regel werden die einzelnen Blöcke dann noch zu einem sogenannten Kontrollflussgraphen (CFG) zusammengefügt, der eine sehr zentrale Bedeutung in jeder Optimierungseinheit eines Compilers hat. Aus Gründen der Komplexität verzichten wir aber auf die Erstellung eines Kontrollflussgraphen. An dieser Stelle ein kurzer Hinweis zur Unterscheidung von lokalen und globalen Optimierungen. Lokale Optimierungen beziehen sich i.d.R. nur auf Basisblöcke, während globale Optimierungen auch den Kontrollfluss, d.h. den Kontrollflussgraphen, miteinbeziehen und damit auf den Instruktionen einer kompletten Funktion arbeiten. Neben der lokalen Elimination von gemeinsamen Teilausdrücken gibt es auch noch eine global Elimination, die auch Teilausdrücke aus anderen Basisblöcken miteinbezieht. Der folgende Algorithmus zur Bestimmung der Basisblöcke aus einem gegebenen Zwischencodeprogramm ist aus [ALSU07] entnommen:

Eingabe: Zwischencodeprogramm in 3-Adress-Code
Ausgabe: Eine Liste von Basisblöcken für das Eingabeprogramm, wobei jede Instruktion genau einem

Basisblock zugewiesen wird. Ein Basisblock könnte z.B. minimal durch zwei Indizes im Zwischencodeprogramm repräsentiert werden.

Algorithmus: Zur Unterteilung des Programm in Basisblöcke müssen diejenigen Instruktionen identifiziert werden, die den Beginn einer nicht-unterbrechbaren Sequenz von Instruktionen darstellen. Folgende Regeln können zur Identifikation dieser Startinstruktionen angewandt werden:

1. die erste Instruktion eines jeden Zwischenprogramm ist eine BB-Startinstruktion
2. jede Instruktion, die von einem bedingten/unbedingten Sprung adressiert wird, stellt eine BB-Startinstruktion dar
3. jede Instruktion, die unmittelbar auf einen bedingten/unbedingten Sprung, einen Funktionsaufruf oder eine Rückgabeinstruktion (return) folgt, ist eine BB-Startinstruktion. Als Sonderregel führen wir hier noch eine kleine Optimierung ein. Der Zwischencode für if-then-else wird häufig durch zwei aufeinanderfolgende Sprungbefehle implementiert - in diesem Fall können wir beide Instruktionen als Ende eines Basisblocks betrachten und erst die darauffolgende Instruktion als BB-Startinstruktion betrachten.

Wenn alle BB-Startinstruktionen identifiziert worden sind, dann beginnt jeder Basisblock bei einer BB-Startinstruktion und endet mit der Instruktion vor der nächsten BB-Startinstruktion.

2.2 DAG

Wenn die Basisblöcke des Eingabeprogramms ermittelt worden sind, kann für weitere Analysen und Optimierungen ein gerichteter, azyklischer Graph aus den Instruktionen eines Basisblocks erstellt werden. Aus diesem Graphen lassen sich dann interessante Eigenschaften eines Basisblockes erkennen. Die Transformation von Basisblockinstruktionen zum DAG wird durch folgenden Algorithmus aus [ALSU07] beschrieben:

Eingabe: Basisblock mit Zwischencode-Instruktionen
Ausgabe: gerichteter, azyklischer Graph des Basisblocks

Algorithmus:

1. jede Variable (allerdings nicht temporäre Variablen), auf die als erstes lesend zugegriffen wird, wird als ein Blattknoten im DAG dargestellt.
2. jede Instruktion im Basisblock wird durch einen Knoten (das Label dieses Knoten ist die Operation) im DAG repräsentiert, wobei die Eingänge (= Kanten von anderen Knoten) die Operanden der Instruktion sind. Zusätzlich wird dieser Knoten noch durch Variablennamen (auch temporäre Variablen) annotiert, die durch diese Operation (neu) definiert (= zugewiesen) worden sind. Sollte diese Variable bereits in der Annotationsliste eines anderen Knoten stehen, wird die Variable aus dessen Liste entfernt.

2.3 Lokale Elimination im DAG

Im zweiten Schritt des obigen Algorithmus wird ein neuer Knoten pro Instruktion erzeugt. Um LCSE zu implementieren wird die zweite Regel wie folgt erweitert:

Bevor ein Knoten für die neue Instruktion erzeugt wird, wird überprüft ob es bereits einen Knoten n gibt, für den folgendes gilt:

- Das Label von Knoten n stimmt mit der Operation der Instruktion überein.

- Die Anzahl Kinder des Knotens n stimmt mit der Operandenanzahl der Instruktion überein.
- Der linke Operandenknoten vom Knoten n enthält den linken Operanden der Instruktion in der Annotationsliste.
- Sofern der rechte bzw. weitere Operanden existieren, enthält die Annotationsliste der jeweiligen Operandenknoten den jeweiligen Operanden der Instruktion.

Wenn ein solcher Knoten n gefunden werden kann, dann wird für die Instruktion kein neuer Knoten erzeugt, sondern lediglich die Zielvariable der Annotationsliste des gefundenen Knotens beigefügt.

Hinzu kommt eine weitere Einschränkung in Bezug auf Arrayzugriffe: da Arrayzugriffe auf den Speicher zugreifen, müssen wir die Zugriffsreihenfolge im Source- bzw. Zwischencode einhalten um die Semantik des Programms beizubehalten. Dies kann durch eine separate Abhängigkeit (= verbundene Kante) im Graph zwischen schreibenden und lesenden Arrayzugriffen (für ein Array) realisiert werden.

Wenn der DAG erzeugt wurde, dann können die gemeinsamen Teilausdrücke identifiziert werden, in dem nach Knoten gesucht wird, die mehr als eine Variable in der Annotationsliste enthalten. In diesem Fall muss der gemeinsame Teilausdruck nicht erneut berechnet werden, sondern lediglich durch eine ASSIGN-Operation der bereits berechnete Wert kopiert werden. Falls mehrere temporäre Variablen in der Annotationsliste vorkommen, können alle temporären Variablen bis auf eine entfernt werden.

Hinweis: im Zwischencode haben wir Funktionen auch durch einen 3-Adress-Code mit der Liste der Übergabeparameter als ein Operand implementiert. Für den DAG sollte aber zur Erkennung von gemeinsamen Teilausdrücken jeder Übergabeparameter als Operand eines Funktionsknoten betrachtet werden - das vereinfacht die Implementierung. Die Berücksichtigung von Funktionen im Allgemeinen ist aber nur optional!

2.4 Transformation von DAG nach 3-Adress-Code

Die im letzten Abschnitt beschriebenen Operationen - Verwendung von ASSIGN und entfernen von redundanten temporären Variablen - erfolgt während der Rücktransformation von DAG nach 3-Adress-Code. Dabei wird jeder vorhandene DAG eines Basisblocks von den Blattknoten beginnend Instruktion-für-Instruktion zurücktransformiert. Als weitere kleine Optimierung am Rande können wir Wurzelknoten - der DAG kann durchaus mehrere Wurzelknoten haben (siehe Beispiel) -, deren Annotationsliste leer ist, einfach verwerfen. Dadurch wird zusätzlich unnützer Code (dead code) entfernt.

3 Beispiel

Als Beispiel betrachten wir folgenden C-Code eines Basisblocks:

```
a = f[i] + 1;
b = a * c;
c = a - d;
b = a - d;
f[i] = b;
```

In Zwischencode entspricht dies folgenden 3-Adress-Code Instruktionen:

```
ARRAY_LD #t1, f, i
ADD a, #t1, 1
```

```
MUL b, a, c
SUB c, a, d
SUB b, a, d
ARRAY_ST, f, i, b
```

Der entsprechende DAG sieht dann wie folgt aus:

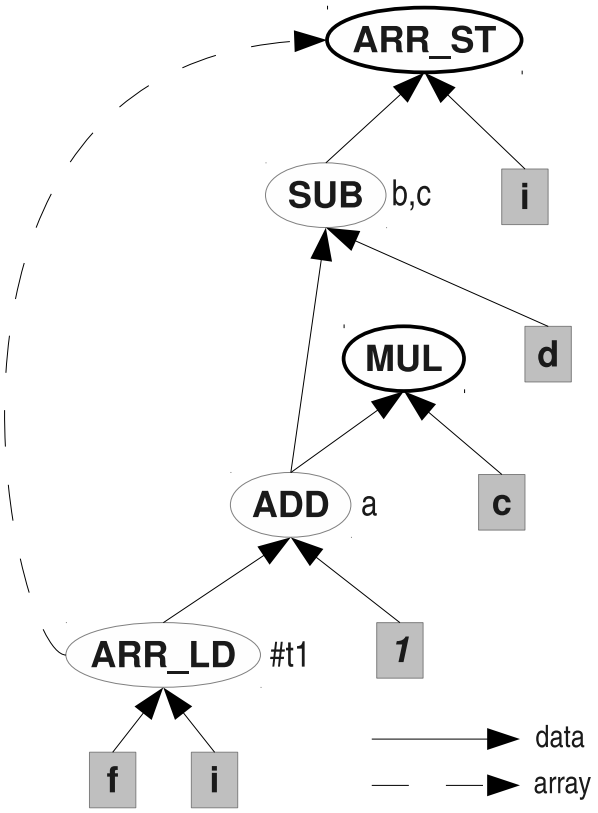

Abbildung 1: DAG

Und abschließend der zurücktransformierte Zwischencode:

```
ARRAY_LD #t1, f, i
ADD a, #t1, 1
SUB c, a, d
ASSIGN b, c
ARRAY_ST, f, i, b
```

Literatur

[ALSU07] Aho, Lam, Sethi, and Ullman. *Compilers - Principles, Techniques, & Tools.* Addison Wesley, 2007.