Aufgabenblatt zum Praktikum Optimierende Compiler

Prof. Dr. Andreas Koch
Jens Huthmann, Julian Oppermann

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2013
Aufgabenblatt 5 — Abgabedatum: 17.07.2013 23:59 CEST

Einleitung

In diesem Aufgabenblatt werden Sie einen bunte Mischung nützlicher Optimierungen implementieren. Dies ist das letzte Aufgabenblatt in diesem Semester. Im Anschluss werden Sie noch Zeit haben, ihre benotete Endabgabe vorzubereiten.

Aufgabe 5.1 Copy-Propagation

Implementieren Sie die Copy-Propagation aus der Vorlesung, und erweitern Sie Ihre Shell um ein Kommando `cp`.

Sehen wir uns das Beispiel in Listing 1 an.

Listing 1: Beispielprogramm vor Copy-Propagation

```
a = y + z;
x = y;
b = x + z;
```

Copy-Propagation erlaubt es uns in der dritten Anweisung `x` durch `y` zu ersetzen und die nun unnötige Kopieranweisung mit Dead-Code-Elimination zu entfernen (siehe Listing 2). Dies ermöglicht es uns zum Beispiel zu erkennen, dass die Werte `a` und `b` identisch sind.

Listing 2: Beispielprogramm nach Copy-Propagation und Dead-Code-Elimination

```
a = y + z;
b = y + z;
```

Hinweise zur Implementierung

Implementieren Sie das in Vorlesung vorgestellte Verfahren zur Copy-Propagation unter Verwendung Ihres Datenflussframeworks. Sie müssen bei der Copy-Propagation nur die Erkennung und Propagierung der Kopien implementieren. Zur Entfernung der nun unnötigen Kopieranweisungen können Sie die bereits im Compiler enthaltene Dead-Code-Elimination (DCE) verwenden.

Aufgabe 5.2 Dead-Code-Elimination

Verwenden Sie die bereitgestellte Dead-Code-Elimination mit dem in Listing 3 gezeigten Aufruf. Die Klasse `OptimizationFactory` finden Sie im Paket `opt` in unserer Bantam-Distribution. Machen Sie diese Optimierung als Kommando `dce` in Ihrer Shell verfügbar.

Listing 3: Aufruf der Dead-Code-Elimination

```
// perform dead code elemination
final Optimization dce = OptimizationFactory.createDCE(false);
optimizer.optimize(dce);
```

Aufgabe 5.3 Constant-Propagation

Als nächste Optimierung sollen Sie Propagierung und Auflösung von konstanten Ausdrücken implementieren. Nennen Sie das entsprechende Kommando in Ihrer Shell `csp`.

In Listing 4 welches Sie auch bereits aus der Vorlesung kennen, können mit Hilfe der Constant-Propagation einige Operation schon während der Übersetzung ausgerechnet werden. Dies führt zu dem Ergebnis in Listing 5

Listing 4: Beispiel vor Constant-Propagation

```
a = 5;
b = 3;
x = a + b;
p = q + a;
d = 0;
if(q > 0) {
  d = 1;
  c = 2 * a;
} else {
  c = 3 * b + 1;
  d = 2;
}
y = c + 7;
z = 4 * d;
```

Listing 5: Beispiel nach Constant-Propagation

```
a = 5;
b = 3;
x = 8;
p = q + a;
d = 0;
if(q > 0) {
  d = 1;
  c = 10;
} else {
  c = 10;
  d = 2;
}
y = 17;
z = 4 * d;
```

Hinweise zur Implementierung

Implementieren Sie das in der Vorlesung vorgestellte Verfahren zur Constant-Propagation unter Verwendung Ihres Datenflussframeworks. Denken Sie daran das Sie bei dem Auflösen der Operatoren auch Überläufe der Zahlenbereiche beachten müssen. Auch dürfen Sie nicht vergessen, dass man auch logische Operationen vorberechnen kann, soweit ihre Operatoren konstant sind.

Es ist nicht Aufgabe dieser Optimierung, Code zu entfernen, welcher durch potentiell konstante Bedingungen für einen Branch nicht mehr erreichbar und daher unnötig geworden ist.

Aufgabe 5.4 Reaching-Definitions Analyse

Implementieren Sie das in der Vorlesung vorgestellte Verfahren zur Reaching-Definitions Analyse unter Verwendung Ihres Datenflussframeworks. Nennen Sie das Kommando in Ihrer Shell `reach`.

Verwenden Sie zur Ausgabe des Analyseergebnisses mittels des Kommandos `dumpdataflow` wieder unsere Klasse `util.DataflowDumper`, diesmal parametrisiert mit der Klasse `util.Definition`, die Sie in der aktualisierten Version des Compilerframeworks auf der Veranstaltungswebsite finden.

Eine `Definition` besteht aus dem Namen der definierten Variablen und der Position der Zuweisung. Die Position wird durch die Basicblock-ID und den Index der Instruktion in der Anweisungsliste des zugehörigen Basicblocks dargstellt ("`x` wird in der dritten Anweisung in BB 17 definiert").

Ein Beispiel zur Instantiierung für Reaching-Definitions sehen Sie in Listing 6.

Listing 6: Instantiierung des DataflowDumper für Reaching-Definitions

```
DataflowDumper<Definition> reachingDefinitionsDumper =
        new DataflowDumper<Definition>(methodName, "REACH");
```

Aufgabe 5.5 Loop-Invariant-Code Motion

(Teil)Ausdrücke einer Schleife, die innerhalb der Schleife nur Variablen verwenden, die außerhalb der Schleife definiert worden sind, können aus der Schleife herausbewegt werden, ohne die Korrektheit des Programms zu beeinflussen. Um diese Ausdrücke zu erkennen verwenden Sie das Ergebnis ihrer Reaching-Definitions Analyse.

Ihre Aufgabe ist es Loop-Invariant-Code Motion (LICM) zu implementieren. Sie sollen sich bei den bearbeiteten Ausdrücken auf primitive Integer Operationen beschränken. Erweitern Sie ihre Shell um ein Kommando `licm`.

Listing 7: Beispiel vor LICM

```
int foo(int x) {
  sum = 0;
  for(i=0;i<n;i++) {
    for(j=0;j<n;j++) {
      sum = sum + x + i;
    }
  }
  return sum;
}
```

Listing 8: Beispiel nach LICM

```
int foo(int x) {
  sum = 0;
  for(i=0;i<n;i++) {
    tmp1 = x + i;
    for(j=0;j<n;j++) {
      sum = sum + tmp1;
    }
  }
  return sum;
}
```

Programmierstil

Die von Ihnen erstellten Programme werden in der Endfassung erfahrungsgemäß einige Tausend Zeilen Java umfassen. Um dem Betreuer das Verständnis und Ihnen die Wartung zu erleichtern, sollen Sie von Anfang an einen sauberen und disziplinierten Programmierstil praktizieren.

Bei der Implementierung sind die Konventionen aus Writing Robust Java Code weitgehend einzuhalten. Dieses Dokument liegt als PDF auch auf der Web-Seite der Vorlesung. Ergänzend soll folgendes beachtet werden:

- Achten Sie darauf, dass Klassen nicht zu komplex werden (zu viele Instanzvariablen, zu viele Methoden). Bei deutlich mehr als 20 dieser Konstrukte sollten Sie die Klasse aufteilen.
- Analoges gilt für die Komplexität von einzelnen Methoden. Auch hier sollten Sie bei mehr als 100 Programmzeilen Länge die Methode aufteilen.
- Verwenden Sie statt Abfragen von `instanceof` echte objekt-orientierte Konstrukte (z.B. polymorphe Methoden).

Der Test und die Abnahme Ihrer Programme wird vom Betreuer auf Linux mit dem SUN Java Development Kit (JDK) Version 1.7 erfolgen.

Dokumentation

Die Lösungen werden nur durch das unten beschriebene `README` und die in das Java-Programm eingebetteten JavaDoc-Direktiven und Kommentare dokumentiert. Achten Sie daher darauf, dass Sie von diesen beiden Möglichkeiten ausreichend und aussagekräftig Gebrauch machen!

Kommentare sollen am Anfang jeder von Ihnen modifizierten oder neu erstellten Quelldatei, pro Klasse und pro Instanzvariable und Methode verfasst werden. Bei Verwendung relativ kurzer Methoden und aussagekräftiger Bezeichner können sich Kommentare innerhalb von Methoden auf wenige wirklich wichtige Stellen beschränken. Bei komplizierteren Methoden soll der Ablauf aber durch eine größere Anzahl an aussagekräftige Kommentaren im Methodenrumpf verdeutlicht werden.

Der Dateikopfkommentar muss neben einer allgemeinen Beschreibung auch eine Historie von Änderungen enthalten. Jeder Eintrag in dieser Historie beschreibt unter Angabe von Datum/Uhrzeit und Namen des Autors auf 1-2 Textzeilen die Natur der Änderungen. Alternativ kann hier auch das Log Ihres Versionskontrollsystems (z.B. CVS oder besser SVN) verwendet werden.

Diese Angaben sind für den Betreuer wichtig, damit im Kolloquium die für ein Thema passenden Ansprechpartner gefunden werden!

Abgabe

Grundsätzlich schickt jede Gruppe spätestens zum Abgabetermin in einem `.jar`- oder `.zip`-Archiv die Quelldateien Ihrer Version des Bantam-Compilers an

`oc@esa.informatik.tu-darmstadt.de`

mit dem Subject `Abgabe 5 Gruppe` N, wobei Ihnen N bereits in der Vorlesung mitgeteilt wurde. In dem Archiv sollen nicht nur die eigenen, sondern alle (auch unmodifizierten) Quellen des Bantam-Compilers enthalten sein. Ebenso legen Sie eventuell verwendete zusätzliche externe Bibliotheken in Form ihrer jeweiligen `.jar`-Dateien bei (aber siehe Abschnitt Plagiarismus).

Am Lehrstuhl wird zur Zeit ein automatisches Testsystem entwickelt. Bitte geben Sie daher zusätzlich auch ein mit `java -jar` ausführbares `.jar`-Archiv ihres Compilers mit ab. Sie müssen also eine Manifest-Datei erstellen, die als Main-Class die Hauptklasse Ihrer Shell spezifiziert, und externe Bibliotheken auf dem Classpath ausweist.

Wichtig: Um unseren Testaufwand überschaubar zu halten, lesen Sie bitte die letzten drei Absätze nocheinmal. Sie sollen abgeben:

- Die vollständigen Quellen (also die `.java`-Dateien!)
- Aber nur diese, nicht noch irgendwelche Altdaten Ihrer eigenen Testläufe (soweit nicht explizit in der Aufgabenstellung angefordert). Räumen Sie also Ihre Verzeichnisse auf, bevor Sie sie in ein Archiv packen!
- Die eventuell benötigten Fremdbibliotheken
- Ein ausführbares `.jar`-Archiv
- Alles zusammen in einem `.jar` oder `.zip`-Archiv. Also kein `.7z`, `.rar`, `.tar.gz` oder `.tbz`

Daneben enthält das Abgabearchiv eine Datei `README.txt`, die enthält

- die Namen der Gruppenmitglieder und die jeweils bearbeiteten Themen
- eine Übersicht über die neuen und geänderten Dateien mit jeweils einer kurzen (eine Zeile reicht) Beschreibung ihrer Funktion.
- Hinweise zur Compilierung der Quellen. Geben Sie eine `javac`-Kommandozeile an bzw. verweisen Sie auf mitgelieferte Makefiles oder ANT Build-Dateien. Nicht ausreichend ist ein Hinweis auf eine von Ihnen verwendete IDE (wie Eclipse, NetBeans etc.).
- Angaben über weitere Bibliotheken (beispielsweise JSAP, log4j, JUnit etc.), die Sie eventuell verwendet haben. Diese Bibliotheken legen Sie bitte dann auch als `.jar` Dateien in das abgegebene Archiv.

Gruppenarbeit

Bei den Tests von Optimierungen ist es häufig so, dass auf den ersten Blick alles funktioniert. Aber dann in einem Quelltext ein "Sonderfall" auftaucht, der vom bisherigen Optimierungscode noch nicht oder nur fehlerhaft bearbeitet wird. Unterschätzen Sie also den Testaufwand nicht!

Falls in Ihrer Gruppe eine Situation entstehen sollte, in der einzelne Mitglieder deutlich zu wenig (oder zu viel!) der anfallenden Arbeitslast bewältigen, sprechen Sie den Betreuer bitte frühzeitig auf die Problematik an. Nur so kann durch geeignete Maßnahmen in Ihrem Interesse gegengesteuert werden. Nach der Abgabe ist es dafür zu spät und Sie tragen die Konsequenzen selber (z.B. wenn sich eines Ihrer Team-Mitglieder wegen seiner Verpflichtungen beim Wasser-Polo nur stark eingeschränkt den Mühen der Programmierung widmen konnte, und Sie daher eine unvollständige Lösung abgeben mussten).

Plagiarismus

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe einer Lösung zu den Programmierprojekten bestätigen Sie, dass Ihre Gruppe die alleinigen Autoren des neuen Materials bzw. der Änderungen des zur Verfügung gestellten Codes sind. Im Rahmen dieser Veranstaltung dürfen Sie den Code des Bantam-Compilers von der FG ESA Web-Seite zu Optimierende Compiler sowie Code-Bibliotheken für nebensächliche Programmfunktionen (Beispiele siehe oben) frei verwenden. Mit anderen

Gruppen dürfen Sie sich über grundlegenden Fragen zur Aufgabenstellung austauschen. Detaillierte Lösungsideen dürfen dagegen nicht vor Abgabe, Artefakte wie Programm-Code oder Dokumentationsteile überhaupt nicht ausgetauscht werden. Bei Unklarheiten zu diesem Thema (z.B. der Verwendung weiterer Software-Tools oder Bibliotheken) sprechen Sie bitte Ihren Betreuer gezielt an.

Weitere Infos unter `www.informatik.tu-darmstadt.de/plagiarism`