Maschinencodegenerierung

SWP Übersetzerbau SS12 (Gruppe F)

21. Juli 2012

Einleitung

Ziel unseres Teilprojektes war es, aus gegebenem optimierten LLVM-Code (32-bit) ausführbaren Maschinencode zu erzeugen. Hierbei konzentrierten wir uns zunächst auf die Erstellung von unter Linux-Systemen funktionierendem GNU-Assembler (32 Bit). Entstanden ist ein Programm namens Codegenerator welcher LLVM-Code in GNU und Intel Assembler übersetzen kann. Die Befehle um den Assemblercode zu übersetzen, können im Config-File entsprechend geändert werden.

Vorüberlegungen

Am Anfang des Projekts stellten wir uns die Frage was wir überhaupt schaffen wollen, dabei legten wir uns auf folgende Primär- und Sekundärziele fest:

Primärziele

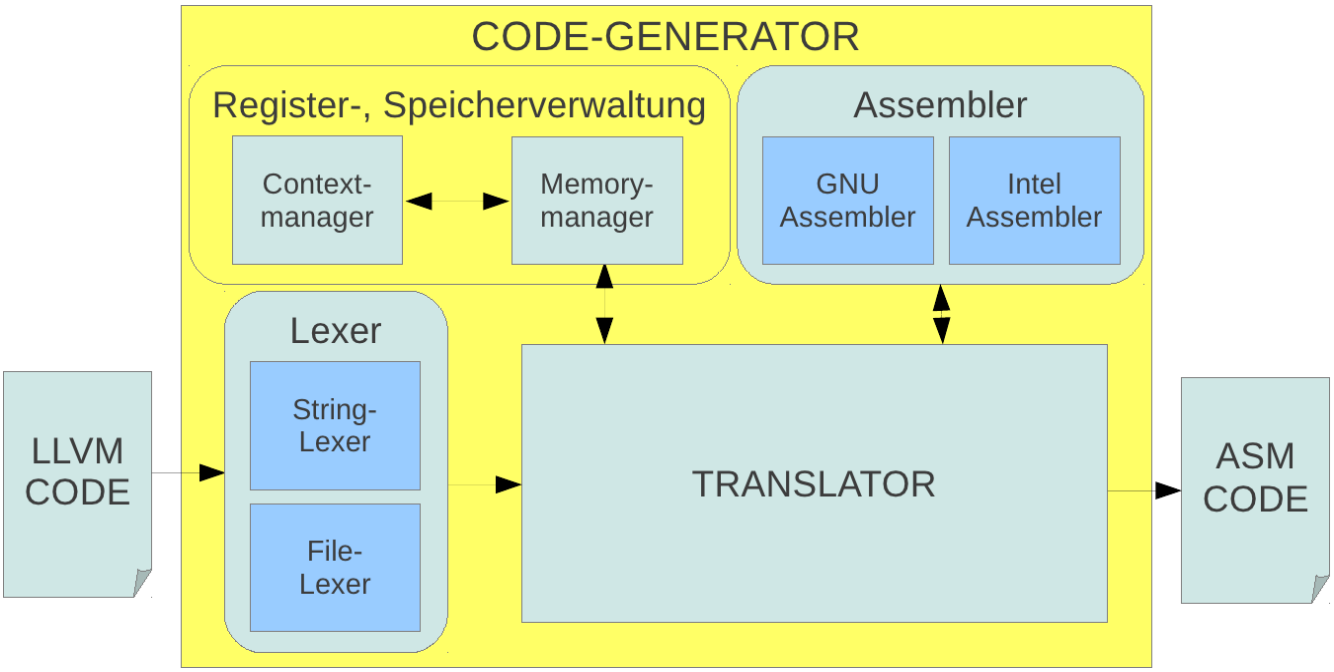
- Einlesen von LLVM-Code aus einer Datei oder Verwendung des Output der Optimierungsgruppe
- Erstellen einer effektiven Variablen- und Registerverwaltung
- Übersetzen des LLVM-Code in GNU-Assembler-Code mittels Assembler Templates
- Bedienung des Code Generators soll angelehnt sein an die Bedienung des C Compilers gcc. (Bsp.: CodeGenerator -o [Ausgabedatei] [Eingabedatei])

Sekundärziele

- Übersetzen in Intel-Assembler
- Optimierung des erzeugten Assembler-Codes

Als Implementierungssprache haben wir uns für Java entschieden, da diese Sprache im gesamten Übersetzerbau Projekt genutzt wird und weil jeder der Teammitglieder dieses Projekts Erfahrungen in Java hat.

Programmaufbau



Programmablauf

Dem Codegenerator wird je nach Implementierung der zu übersetzende LLVM-Code als Datei oder als String übergeben. Der Lexer erstellt dann aus dem eingelesene LLVM-Code eine Tokenstream, wobei jeder Token die relevanten Informationen aus einer LLVM-Codezeile enthält, und übergibt diesen an den Translator. Dieser erstellt nun anhand des Tokenstreams das fertige Assemblerprogramm, welches extern in Maschinencode übersetzt werden kann. Beim Übersetzen behilft sich der Translator einer abstrakten Assemblerklasse, welche die Befehle in den gewünschten Assemblerdialekt umwandelt. Mit Hilfe der Register- und Speicherverwaltung baut sich der Translator eine abstrakte Darstellung der später erwarteten Prozessorvorgänge auf um die richtigen Entscheidungen für Register- und Speicherplatzvergaben zu treffen.

Benutzung

Die Benutzung des Code-Generators orientiert sich an anderen Standard Linux Compilern wie dem gcc. Um auf das Verhalten des Code-Generators Einfluss zu nehmen, wurden folgende Flags definiert:

- `-asmType gnu | intel`
  Wahl zwischen Gnu-Assembler und Intel-Assembler, wobei Gnu-Assembler der Standard ist
- `-o <Dateiname>`
  Mit dieser Option wird die Ausgabedatei `<Dateiname>` angegeben
- `-e`
  Der erzeugte Assemblercode wird im Nachhinein in eine ausführbare Datei compiliert

- `-C <Config datei>`
  Als Standard Config-Datei, wird die Datei `mc_config.cfg`aus dem Hauptverzeichnis des Code-Generators benutzt. Mit diesem Flag ist es möglich eine eigene Config-Datei zu wählen. Eine Config-Datei enthält Befehle, welche auszuführen sind um den erzeugten Assembler-Code in eine ausführbare Datei zu compilieren.

  folgende Optionen gibt es nur im separaten Projekt :

- `-v | --verbose`
  Ausgabe von debug Informationen

- `-g | --gui`
  Ausgabe von debug Informationen in einer GUI

  Es ist äußerst wichtig, dass es sich bei dem eingegebenen LLVM-Code um solchen für 32-Bit Systeme handelt. Ansonsten kann kein Maschinencode erzeugt werden, da im 64-Bit-LLVM unbekannte Befehle verwendet werden um z. B. mit den größeren Pointern umzugehen. Bei clang wird dies mit der Flag -m32 gesichert.

  Einige Beispiel-LLVM-Programme für Tests befinden sich im Branch der Gruppe F im input-Ordner unter llvm_examples

Hauptkomponenten

Lexer

Der Lexer ist ein Objekt, welcher den LLVM-Code Zeilenweise aus einer angegebenen Datei (FileLexer) oder aus einem übergebenen String (StringLexer) liest. Die Relevanten Informationen aus einer LLVM-Codezeile, die zur Übersetzung benötigt werden, werden dann in ein Tokenkonstrukt gepackt und zur weiteren Verarbeitung bereitgestellt. Da es sich bei LLVM um Drei-Adress-Code handelt lassen sich die relevanten Informationen der einzelnen LLVM-Befehle grob aufteilen in: Art des Befehls, Ziel und Operanden. Bei Methodendefinitionen werden zusätzlich die Parameter gespeichert. Des weiteren führt der Lexer, auf dem vorher erstellten Tokenstream, eine grobe Vorverarbeitung durch, bei dem inline String-Definitionen in globale überführt und implizite Label eingeführt werden.

Beim erstellen des Lexer's, muss die zu parsende Datei dem Konstruktor übergeben werden. Danach stehen einem die folgenden drei öffentlichen Methoden zur Verfügung:

```
void open(<Dateiname>);
ArrayList<Token>() getTokenStream();
void close();
```

Mithilfe der Methode `open(<Dateiname>)`, kann dem Lexer eine neue zu parsende Datei übergeben werden. `getTokenStream()`startet nun den Parse-Vorgang und liefert den generierten Tokenstream in Form einer ArrayList zur Verfügung. Sollte ein Fehler beim öffnen der zu parsenden Datei auftauchen, werden geeignete Fehlermeldungen auf der Fehler-Console ausgegeben. Um den Lexer wieder von der Datei zu trennen, muss die Methode `close()`aufgerufen

werden.

Folgendes Beispiel erläutert dieses Vorgehen anhand des FileLexer's:

```
ArrayList<Token> tokenStream;
Lexer lex = new FileLexer("llvmFile.llvm");
tokenStream = lex.getTokenStream();
if(tokenStream == null) {
    System.out.println("Error");
}
lex.close();
```

Token

Wie im Lexer bereits erwähnt, enthält ein Token alle, für die Übersetzung, relevanten Informationen die in einer LLVM-Code Zeile enthalten sind. Die folgenden Informationen werden gespeichert:

- Ziel

  Der Variablenname des Ziels in dem ein Wert gespeichert werden soll

- Typ des Ziels

  Welchen Typ das Ziel hat z.B. i32 oder double

- Operator 1

  Der Operand 1 des drei Address-Codes

- Typ des Operator 1

  Welchen Typ der Operand 1 hat z.B. i32 oder double

- Operator 2

  Der Operand 2 des drei Address-Codes

- Typ des Operator 2

  Welchen Typ der Operand 2 hat z.B. i32 oder double

- Typ des Tokens

  Der Typ des Tokens z.B. Branch, Assignement, Call, Store etc...

- Parameter

  Eine Liste von Parametern, wie zum Beispiel bei Funktionsaufrufen. Ein Parameter enthält dabei den Namen der Variablen und den Typ der Variablen.

Translator

Der Translator greift auf eine Umsetzung der abstrakten Klasse Assembler zu, z.B. GNUAssembler. Mit den dort zur Verfügung stehenden Funktionen wird zunächst die Grundstruktur für ein ausführbares Programm in der jeweiligen Architektur erstellt. Dann werden die vom Lexer erstellten Token durchlaufen, übersetzt und in das Grundgerüst eingefügt. Für die Übersetzung von Variablennamen zu Speicheradressen greift der Translator auf die Funktionen der Speicherverwaltung (MemoryManager) zu.

Als Beispiel die Übersetzung eines Tokens tok mit den Eigenschaften
type = Allocation, target = % i, typeTarget = i32.
Es soll also Speicher für eine 32-Bit-Integer-Variable namens i alloziert werden:

Anhand des Tokentypes Allocation werden die entsprechenden Befehle im Translator aufgerufen. Dann wird anhand des Typs der Variablen festgestellt, für welche Art von Variable Speicher angelegt werden soll. In diesem Falle wird durch den MemoryManager eine neue Variable auf dem Stack angelegt, anschließend wird der Stackpointer entsprechend der Größe der neuen Variable bewegt.

```
case Allocation:
    String tT = tok.getTypeTarget();
    // Array
    if  (tT.startsWith("[")){
        ...
    }
    // Record
    else if(tT.startsWith("%")) {
        ...
    }
    // Variable
    else{
        // Neue Variable anlegen
        Variable newVar = mem.newStackVar(tok.getTarget(), tT);
        // Stackpointer verschieben
        asm.sub(String.valueOf(newVar.getSize()), "esp", "Allocation " + tok.getTa
    }
    break;
```

Register- und Speicherverwaltung

Der MemoryManager verwaltet den für ein Programm zur Verfügung stehenden Speicher. In den einfachsten Fällen gibt er also bei Anfrage freie Speicherbereiche in Form von Registern oder Stack-Adressen zurück und ordnet bereits existierenden Variablen ihre aktuellen Adressen zu. Natürlich muss dies je nach Art der Variable unterschiedlich behandelt werden.

Hierbei wird zwischen dem „globalen“ Speicher und den einzelnen LLVM-Namespaces, also den Zuordnungen innerhalb der einzelnen Methoden (MemoryContext) unterschieden.

Darüber hinaus gibt es noch die Möglichkeit Variablen von Registern auf den Stack zu verschieben, um sie z. B. vor einem Funktionsaufruf zu sichern.

Assembler

Die Klasse Assembler an sich enthält lediglich die abstrakten Methoden, die in den Klassen GNUAssembler und IntelAssembler überschrieben werden. Der abstrakte Aufbau der Assemblerklasse macht es sehr leicht, neue Assemblerdialekte hinzuzufügen.

Die Klassen GNUAssembler und IntelAssembler implementieren in den einzelnen Methoden die Übersetzung der einzelnen Teile in entsprechenden Assemblercode als String. Für IntelAssembler gibt es eine automatische Prüfung auf das ausführende Betriebssystem um den korrekten Code für Windows oder Windows zu erstellen, der leicht voneinander abweichen kann. Zum Beispiel muss **extern exit** bei Linux unter Windows als **extern _exit** geschrieben werden. Der gesamte Assembler wird nach und nach Aufgebaut und schlussendlich als Ganzes vom Translator geholt.

Deployment

Da die Maschinencode-Erzeugung das letzte Glied in diesem Projekt darstellte, waren wir nicht sehr stark von anderen Gruppen abhängig. In welchem Format wir den LLVM der Optimierungsgruppe übergeben bekommen, haben wir uns schon früh geeinigt und so ging das Einpflegen unseres Teilprojekts in den Master ohne Probleme von statten. Lediglich das genaue Set der verwendeten LLVM-Befehle stand erst sehr spät fest, was keine Auswirkungen auf unser Deployment hatte, jedoch bei Tests von anderen Gruppen dazu führte, dass ein Durchlauf im Master in bestimmten Fällen nicht das Ende erreichte.

Arbeitsaufteilung

- Henry: Arrays, Funktionsaufrufe, Speicherverwaltung, Assembler, Dokumentation
- Michael: Registerverwaltung, Deployment
- Peter: Assembler, Records, Double's , Troubleshooting, Executables
- Regina: Intel Assembler, Test-LLVMs, Ansprechpartner, Recherche, Wiki
- Team Regina Peter: Lexer, restlicher Translator, Dokumentation

Probleme

Da wir mit den anderen Gruppen gleichzeitig begonnen haben und so für die meiste Zeit des Projektes keinen LLVM-Code zur Verfügung hatten, der aus der Quellsprache erzeugt wurde, haben wir uns hauptsächlich an Code orientieren müssen, der mit Standardtools (Clang) aus C-Code erzeugt wurde. Für die anderen Gruppen haben wir deshalb, wie abgesprochen, ein Set mit von uns unterstützen LLVM Befehlen zusammengestellt.

Leider wurde dies nicht weiter beachtet und so gab es trotzdem beim intern erzeugten LLVM-Code Unterschiede, auf die wir dann kurzfristig reagieren mussten, da wir diesen erst sehr spät erhielten. Dies war zum Beispiel besonders bei der Umsetzung von Strings problematisch und erforderte manchmal sogar komplettes ersetzen von bereits implementierten Methoden um sowohl

generierten LLVM als auch an uns übergebenen LLVM zu unterstützen.

Da wir also kaum intern erzeugten Code zur Verfügung hatten, waren wir auch beim Testen und Debuggen auf externen Code angewiesen, daher konnten wir Tests als Teil der gesamten Compilerkette nicht in dem Maße ausführen, wie wir es gerne getan hätten.

Weiterhin haben wir keine Übersetzung für Arrays of Records und Records of Arrays implementiert, da dies schon von Gruppe A nicht unterstützt wird und wir somit nicht wissen, wie der entsprechende LLVM aussehen würde und entsprechender Code somit auch gar nicht erst bei uns ankommen kann.

Fazit

Betrachtet man rückblickend unsere gestellten Ziele in diesem Projekt, haben wir bis auf das Sekundärziel der Optimierung von Maschinencode alle Punkte erfolgreich umgesetzt. Jedoch hätte eine nachträglich Optimierung auch vollkommen den Rahmen gesprengt. Auf die Übersetzung von Arrays of Records und Records of Arrays haben wir verzichtet, da diese Fälle von anderen Gruppen nicht vorgesehen sind und somit nicht bei uns ankommen können. Auch in unserem separaten Teilprojekt, was auf generiertem bzw manuell erstelltem LLVM-Code arbeitet, haben wir uns zugunsten von ausgiebigeren Tests auf dem Master dagegen entschieden.

Alle anderen Punkte werden für Linux zuverlässig in korrekt ausführbaren Intel-, oder Gnu-Assembler übersetzt. Die automatische Betriebssystemerkennung erstellt auch für Windows angepassten Maschinencode, der mit entsprechender Änderung der Config-Datei auch direkt als ausführbare Datei erzeugt werden kann. Leider konnten wir unter Windows keine Tests durchführen.