MIMD Processing Unit Compiler Dokumentation

abaxor engineering GmbH

August 2, 2024

Modell Diagramm

Das folgende Diagramm veranschaulicht die gesamte Architektur und den Ablauf des Projekts:

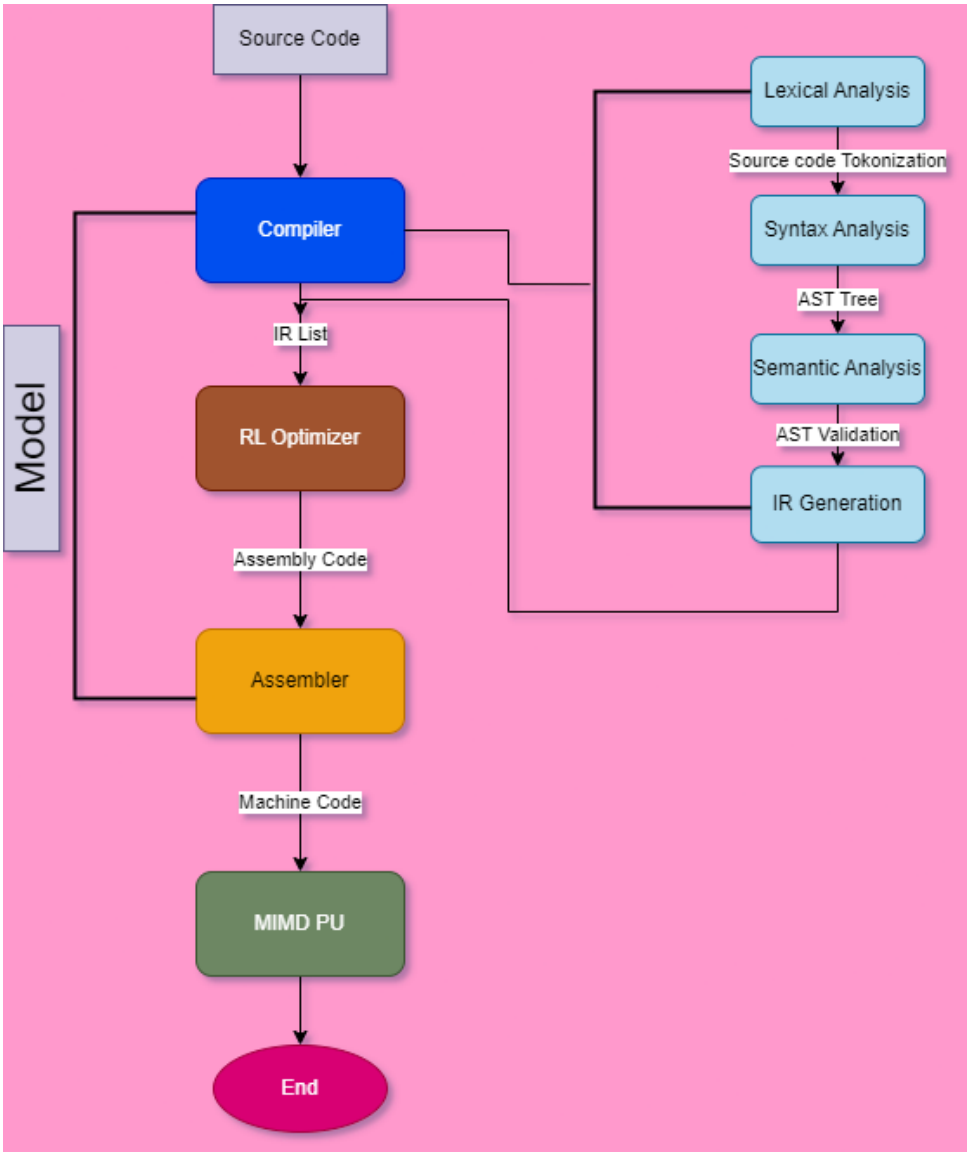

Figure 1: Beschreibung des Diagramms.

Ziel

Das Hauptziel dieses Projekts ist die Entwicklung eines Compilers, der mit einem Assembler für eine MIMD (Multiple Instruction, Multiple Data) Processing Unit Architektur integriert ist. Das Projekt zielt darauf ab, Techniken des Reinforcement Learning zu nutzen, um den Kompilierungsprozess zu optimieren. Aufgrund der aktuellen Unverfügbarkeit einer MIMD PU-Architektur richtet sich die anfängliche Implementierung an die x86-Architektur, mit dem Ziel, diese in Zukunft für die MIMD-Architektur anzupassen.

Aktueller Entwicklungsstand

Die aktuelle Implementierung ist ein grundlegendes Compiler-Framework, das die folgenden Komponenten umfasst:

Lexikalische Analyse

- **Zweck**: Zerlegt den Quellcode in eine Folge von Token.
- **Details**: Der lexikalische Analysator scannt den Quellcode und wandelt ihn mithilfe regulärer Ausdrücke in Token um. Diese Token umfassen Bezeichner, Zahlen, Operatoren, Zuweisungssymbole und Anweisungsbegrenzer.

Syntaxanalyse

- **Zweck**: Erstellt einen Abstrakten Syntaxbaum (AST) aus dem Tokenstrom.
- **Details**: Der Syntaxanalysator verwendet einen Parser, um den Tokenstrom zu verarbeiten und einen AST zu erstellen. Er erkennt verschiedene Arten von Ausdrücken und Anweisungen und stellt sicher, dass die syntaktische Struktur des Quellcodes korrekt ist.

Semantische Analyse

- **Zweck**: Führt grundlegende Typüberprüfungen und Validierungen der Variablendeklaration durch.
- **Details**: Der semantische Analysator durchläuft den AST, um die Typen und Bereiche der Variablen zu validieren. Er überprüft auf Typinkonsistenzen und stellt sicher, dass alle Variablen vor ihrer Verwendung deklariert sind.

Erzeugung der Zwischenrepräsentation (IR)

- **Zweck**: Erzeugt eine Zwischenrepräsentation des Codes.
- **Details**: Der IR-Generator wandelt den AST in eine niedrigere Zwischenrepräsentation um, die leichter zu optimieren und in Maschinencode zu übersetzen ist.

Codegenerierung (x86-Assembler)

- **Zweck**: Übersetzt die IR in x86-Assemblercode mit grundlegender Registerzuweisung.
- **Details**: Der Codegenerator übersetzt die IR in x86-Assembleranweisungen. Er umfasst grundlegende Registerzuweisungsstrategien, um Variablen und temporäre Werte den CPU-Registern zuzuordnen.

Übersetzung von Assembler in Maschinencode

- **Zweck**: Wandelt den generierten Assemblercode in eine vereinfachte Darstellung des Maschinencodes um.
- **Details**: Der Assembler nimmt den x86-Assemblercode und wandelt ihn in eine Folge von Maschinencode-Anweisungen um, die zur Ausführung auf einem x86-Prozessor bereit sind.

Implementierte Funktionen

- **Lexikalische Analyse**: Zerlegt den Quellcode in eine Folge von Token.
- **Syntaxanalyse**: Erstellt einen Abstrakten Syntaxbaum (AST) aus dem Tokenstrom.
- **Semantische Analyse**: Führt grundlegende Typüberprüfungen und Validierungen der Variablendeklaration durch.
- **IR-Erzeugung**: Erzeugt eine Zwischenrepräsentation des Codes.
- **Codegenerierung**: Übersetzt die IR in x86-Assemblercode mit grundlegender Registerzuweisung.
- **Assembler**: Wandelt den generierten Assemblercode in eine vereinfachte Darstellung des Maschinencodes um.

Code-Struktur

Die `BasicCompiler`-Klasse umfasst alle Kompilierungsstufen:

- `lexical_analysis`: Zerlegt den Quellcode in Token.
- `syntax_analysis`: Erstellt den AST.
- `semantic_analysis`: Validiert den AST auf semantische Korrektheit.
- `generate_ir`: Erzeugt eine Zwischenrepräsentation.
- `optimize_ir`: Platzhalter für zukünftige IR-Optimierung mit RL.
- `code_generation`: Erzeugt x86-Assemblercode.
- `assemble`: Wandelt Assemblercode in Maschinencode um.

Einschränkungen des aktuellen Modells

- **Begrenzte Sprachunterstützung**:
  - Die aktuelle Implementierung unterstützt nur eine Teilmenge der C-ähnlichen Sprachmerkmale.
  - Keine Unterstützung für komplexe Konstrukte wie Schleifen, Bedingungen und Funktionen.
- **Einfaches Typsystem**:
  - Unterstützt nur ganzzahlige Typen.
  - Keine Unterstützung für Gleitkommazahlen, Arrays und benutzerdefinierte Typen.
- **Vereinfachte Assembler-Generierung**:
  - Erzeugt einfachen x86-Assemblercode ohne fortgeschrittene Optimierungen oder vollständige Unterstützung des Befehlssatzes.
- **Naive Registerzuweisung**:
  - Die aktuelle Registerzuweisungsstrategie ist einfach und nutzt die verfügbaren Register möglicherweise nicht effizient.
- **Keine Optimierung**:
  - Obwohl ein Platzhalter für die IR-Optimierung vorhanden ist, wurden noch keine tatsächlichen Optimierungen implementiert. wird in Zukunft RL-Optimierung umsetzen
- **Begrenzte Fehlerbehandlung**:

  - Die Fehlerberichts- und Wiederherstellungsmechanismen sind einfach und bieten möglicherweise keine detaillierten Diagnoseinformationen.

- **Fehlende MIMD-spezifische Funktionen**:
  - Da sich die aktuelle Implementierung auf x86 richtet, enthält sie keine MIMD-spezifischen Optimierungen oder Codegenerierungsstrategien.

- **Vereinfachter Assembler**:
  - Die Übersetzung von Assembler in Maschinencode ist stark vereinfacht und spiegelt nicht die Komplexität eines realen Assemblers wider.

- **Keine Reinforcement Learning-Integration**:
  - Der im Diagramm gezeigte RL-Optimierer ist im aktuellen Code noch nicht implementiert.

- **Begrenzter Umfang**:
  - Der Compiler verarbeitet derzeit nur einfache arithmetische Ausdrücke und Variablenzuweisungen.

Zukünftige Arbeiten

- **Implementierung des Reinforcement Learning Optimierers**:
  - Entwicklung und Integration von RL-Techniken zur Optimierung der IR.

- **Erweiterung der Sprachunterstützung**:
  - Einbeziehung komplexerer Konstrukte wie Schleifen, Bedingungen und Funktionen.

- **Verbesserung des Typsystems**:
  - Unterstützung einer breiteren Palette von Datentypen, einschließlich Gleitkommazahlen, Arrays und benutzerdefinierter Typen.

- **Fortgeschrittene Optimierungstechniken**:
  - Implementierung verschiedener Optimierungsstrategien sowohl in der IR- als auch in der Codegenerierungsphase zur Verbesserung der Effizienz.

- **MIMD-spezifische Codegenerierung**:
  - Entwicklung von Codegenerierungs- und Optimierungsstrategien, die speziell für die MIMD PU-Architektur angepasst sind.

- **Verbesserung der Fehlerbehandlung**:
  - Verbesserung der Fehlererkennung, -berichterstattung und -wiederherstellung, um detaillierte Diagnoseinformationen bereitzustellen.

- **Fortschrittliche Registerzuweisung**:
  - Implementierung fortgeschrittener Registerzuweisungsalgorithmen zur effizienten Nutzung der verfügbaren Register.

- **Erweiterung des Assemblers**:
  - Unterstützung eines umfassenderen Spektrums an x86-Befehlen und Adressierungsmodi, um den Assembler realistischer und vielseitiger zu gestalten.

- **Anpassung an die MIMD PU-Architektur**:
  - Beginn des Anpassungsprozesses des Compilers für die Zielarchitektur MIMD PU, sobald diese verfügbar ist.

Fazit

Diese Dokumentation bietet einen Überblick über den aktuellen Stand des MIMD Processing Unit Compiler-Projekts, indem sie die Ziele, die aktuelle Implementierung, die Einschränkungen und die zukünftigen Richtungen hervorhebt. Im Laufe der Entwicklung sollte diese Dokumentation aktualisiert werden, um neue Entwicklungen und erreichte Meilensteine zu reflektieren.