Robust Practical Binary Optimization at Run-time using LLVM

Seminar Supercomputer

Alexander Verdieck

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2021-01-07

Gliederung

Motivation

- Optimierung von Code für schnellere und effizentere Programme
- Verschiedene Arten der Optimierung
  - IO-Operationen
  - Parallelisierung von Code
  - Scheduling
  - Optimierung von CPU-Instruktionen
  - Bessere Effizienz von Code

Problem

- Viele Optimierungen brauchen Informationen über aktuelle Zustände im System
- Einige Informationen sind schon zur Compile-Zeit interessant
  - z.B. Data-Layout
- Es können auch nicht vorab beliebig viele Versionen erstellt werden

Einschub: LLVM

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies.

- Gestartet als Projekt an der University of Illinois
- Kann benutzt werden um Code zu optimieren
- Nutzt LLVM-IR als Zwischensprache
- Sehr großes und weit entwickeltes Projekt

BinOpt

- Entwickelt von an der Technischen Universität München

- Bibliothek zum optimieren während der Laufzeit
- Wird aus dem Code heraus aufgerufen
  - C-API
- Benutzt zusätzlich Rellume und LLVM
- Maschinencode wird zu LLVM-IR geliftet
- LLVM ermöglicht Optimierung

Hauptteil

```
1  int func(int a, int b) { return a - b; }
2  int main(void) {
3      // Create a new handle into the library
4      BinoptHandle h = binopt_init();
5      // Create a configuration for func
6      BinoptCfgRef bc = binopt_cfg_new(h, func);
7      // Specify signature, two parameters
8      binopt_cfg_type(bc, 2, BINOPT_TY_INT32,
9      BINOPT_TY_INT32, BINOPT_TY_INT32);
10     // Specify second parameter as constant 42
11     binopt_cfg_set_parami(bc, 1, 42);
12
13     int(*nfn)(int, int) = binopt_spec_create(bc);
14     // Call new version, uses 42 instead of 16
15     int res = nfn(48, 16);
16 }
```

Listing 1: Einfache Nutzung von BinOpt

Arbeitsweise

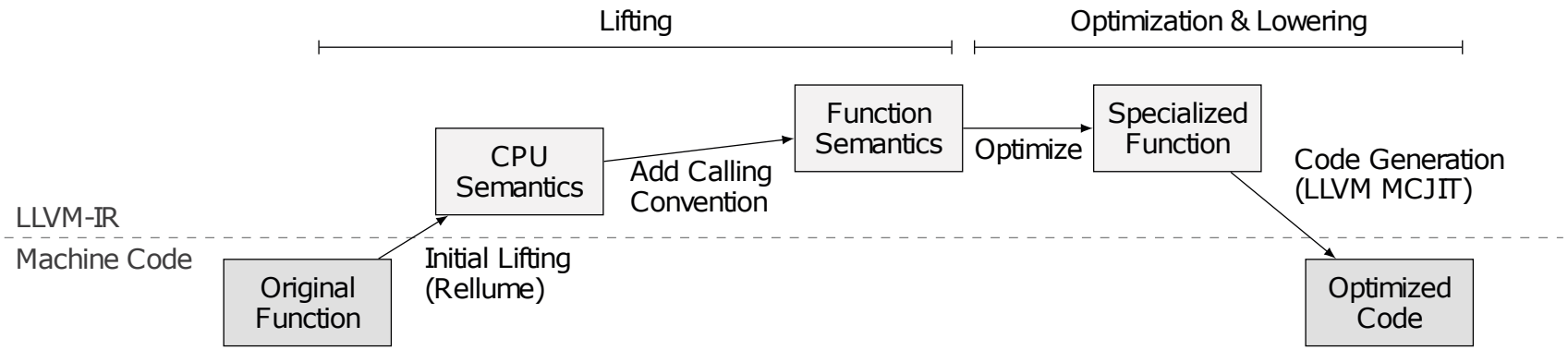


Abbildung: Arbeitsweise von BinOpt

Lifting

- Die Übersetzung wird von Rellume vorgenommen:
  1. Dekodierung der Befehle
  2. Befehle werden einzeln in die LLVM-IR gebracht
  3. LLVM-IR Blöcke werden verbunden
- Anschließend wird die Signatur genutzt um eine LLVM Funktion zu erstellen

Lifting

- Aufrufe und indirekte Sprünge können zum Abbruch der Funktion führen
- Einführen einer Hilfsfunktion die aufgerufen wird
  - Die zu optimierende Funktion terminiert korrekt
  - Sobald die Adresse bekannt wird, kann mehr Code gelifted werden.

Code Discovery

- Einiger Code wird erst nach Optimierung erreichbar

- Mehrfache verkürzte Optimierungen
- Kein liften von Code im Procedure Linkage Table
  - Adressen können häufig nicht bestimmt werden
- Nicht alle Funktionen sollen optimiert werden
  - z.B. malloc oder memcopy

Indirect Jumps

- Indirekte Sprünge können nicht immer aufgelöst werden
- Drei mögliche Lösungsansätze:
  - Prozess abbrechen und Original zurückgeben
  - Eventuell neuer Prozess zu späteren Zeitpunkt
  - Weiterführung der Berechnung mit Originalcode
- Wiederherstellung der CPU-Register

Constant Propagation

- LLVM kann keine konstanten Adressen erkennen
- Zur Laufzeit sind solche Adressen bekannt

1. Erweiterung der Pipeline
2. Ausrechnen aller Adressen von Ladebefehlen
3. Ersetzen der Adresse falls Zugriff im konstanten Bereich

Benchmark

- Optimierung anhand von Programmen für Bildverarbeitung
  - Stencil
  - gblur-1d (GEGL Bibliothek)
  - bilateral-filter (GEGL Bibliothek)
- Alle Experimente wurden 5 mal wiederholt
  - LLVM 9.0
  - Fedora 31 (Linux kernel 5.7.15)
  - Intel Core i5-8250U CPU@1.6GHz

Stencil

$$\{(1, 0, 0.25), (0, -1, 0.25), (0, 1, 0.25), (-1, 0, 0.25)\}$$

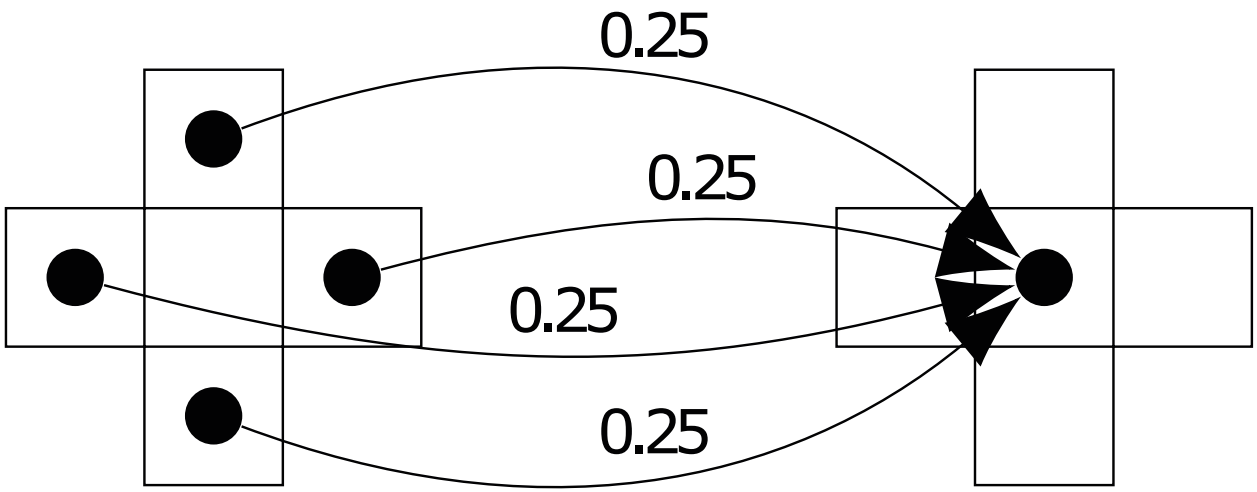


Abbildung: Beispiel Stencil

Konfigurationen

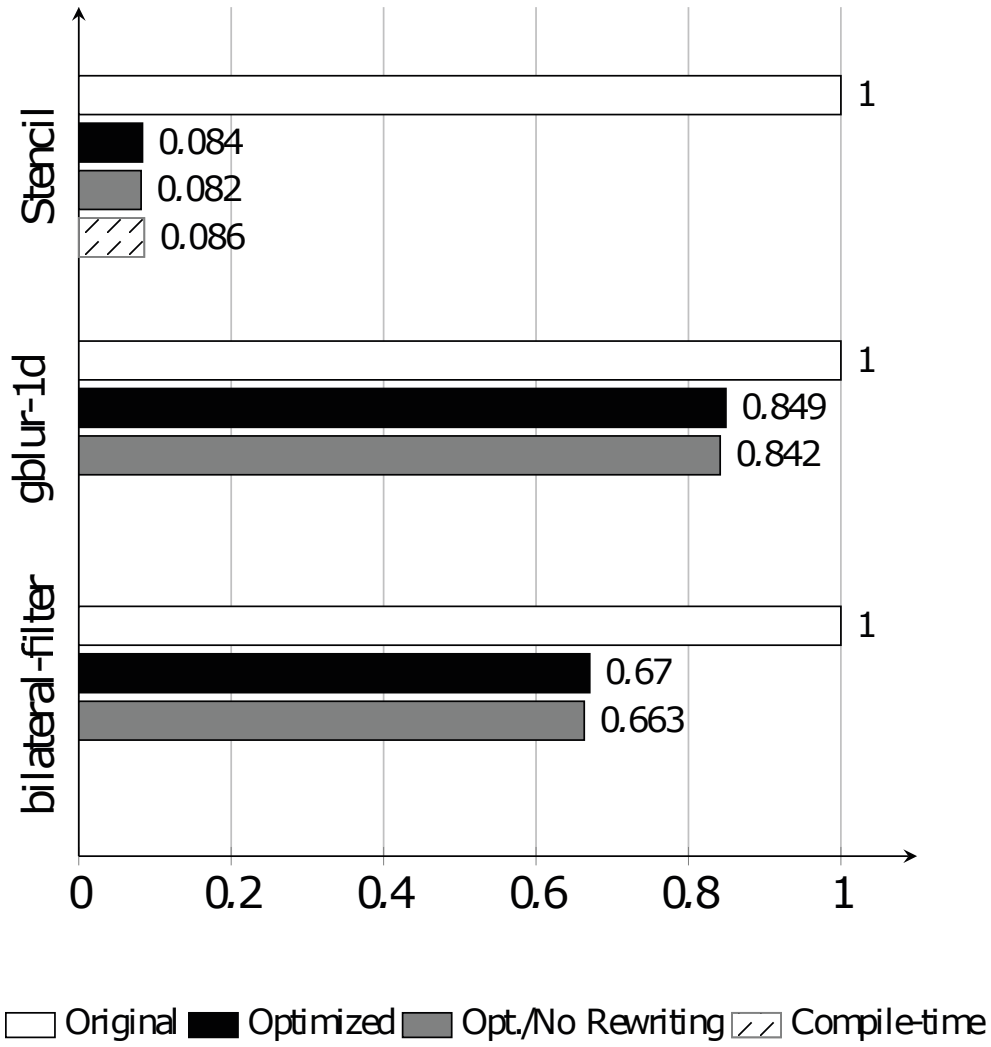
- Stencil
  - 649 · 649 Matrix
  - 10'000 Wiederholungen
- gblur-1d
  - 9000 · 4923 Farbbild
  - Stencilgröße von 11
- bilateral-filter
  - 3000 · 1641 Matrix
  - Stencilgröße von 9*9

Resultate

	Stencil	gblur-1d	bilateral-f.
Original run-time	17.712s	2.992s	5.461s
Rewriting time	0.031s	0.023s	0.039s
Opt. run-time (w/rew.)	1.481s	2.540s	3.662s
Overall improvement	−16.231s	−0.452s	−1.799s
Original code size	290 B	363 B	680 B
Rewritten code size	587 B	583 B	567 B
Nested loops	3	2	5
Loop unrolling	Yes	Yes	Yes
Constant propagation	Yes	Yes	Yes
Vectorization	Yes	Yes	No

Abbildung: Resultate

Laufzeit


Stencil
1
0.084
0.082
0.086
gblur-1d
1
0.849
0.842
bilateral-filter
1
0.67
0.663
0
0.2
0.4
0.6
0.8
1
Original
Optimized
Opt./No Rewriting
Compile-time

Zusammenfassung

- BinOpt ermöglicht die Optimierung zur Laufzeit
- LLVM für die Optimierung von Code
- Einfache benutzung über die C-API
- Deutliche Performance Verbesserungen möglich

- Gelernt, wie man Maschinencode zu LLVM überführen kann

Literatur

[Ale] Martin Schulz Alexis Engelke. Robust Practical Binary Optimization at Run-time using LLVM. *202 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC and Workshop on Hierachical Parrallelism for Exascale Computing.*

[Gita] BinOpt GitHub. `https://github.com/aengelke/binopt`.

[Gitb] Rellume GitHub. `https://github.com/aengelke/rellume`.

[Lin] Wikipedia Linker. `https://en.wikipedia.org/wiki/Linker_%28computing%29`.

[Pro] LLVM Projekt. `https://llvm.org/Features.html`.