Automatic compiler customization for novel hardware

Automatische Anpassung von Compilern für neuartige Hardwarearchitekturen
Master thesis by Janne Wulf
Date of submission: August 9, 2021

1. Review: Dr. rer. nat. Stefan Guthe
2. Review: Daniel Thuerck
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Janne Wulf, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, August 9, 2021

J. Wulf

Contents

Todo list

Check the font. The 't' looks very small

1 Background

In this chapter we are introducing knowledge which is required to understand the content of this thesis. In section 1.2 we give an overview about how a compiler is typically implemented and which problems it has to solve that we want to tackle. The sections 1.2.1 to 1.2.2 give a superficial overview of the first two phases of a typical compiler. Section 1.2.3 gives a more detailed introduction to compiler back ends, which we aim to optimize in this thesis.

explain also the other section

1.1 CPU Functionality

ISA
Memory and Registers
How do CPUs work in general? -> Fetch, Decode, etc.
In-order vs out-of-order architectures

1.2 Compilers

Making computer programs, that are written in high-level programming languages (e.g., C/C++, Java, Rust), executable on a specific machine is not a trivial task. Compilers are only one piece in the tool-chain required to make a program executable. The compiler translates the high-level language into assembly language, which is translated into object code by the assembler. Basic functionality like allocating memory or outputting strings

Start Cycle	Occupied Cycles	Instruction		
1	1-3	`load $a`	→	`a`
4	4-5	`mul a, a`	→	`a`
5	5-7	`load $b`	→	`b`
8	8-9	`mul b, b`	→	`b`
9	9-11	`load $c`	→	`c`
12	12-13	`mul c, c`	→	`c`
13	13	`add a, b`	→	`a`
14	14	`add a, c`	→	`a`
15	15-17	`store a`	→	`$a`

Table 1.1: Unscheduled example `a=a*a+b*b+c*c`

Start Cycle	Occupied Cycles	Instruction		
1	1-3	`load $a`	→	`a`
2	2-4	`load $b`	→	`b`
3	3-5	`load $c`	→	`c`
4	4-5	`mul a, a`	→	`a`
5	5-6	`mul b, b`	→	`b`
6	6-7	`mul c, c`	→	`c`
7	7	`add a, b`	→	`a`
8	8	`add a, c`	→	`a`
9	9-11	`store a`	→	`$a`

Table 1.2: Scheduled example a=a*a+b*b+c*c

on the screen is implemented in a standard library. The object code of the standard library and potentially other libraries are linked together with the translated program by the linker. There is more required to execute the code on a specific machine (e.g., a runtime library), but explaining this would go beyond the scope of this thesis.

The pure translation of the program is only one of the tasks a compiler has to fulfill. It also has to assure that the program is written in correct syntax of the high-level language. Most compilers will also optimize the given code and the translated code since a simple one-to-one translation would have a very poor performance. Eventually there has to be a mapping from variables and to the main memory and the processors registers. These are all by itself complex problems which are handled by a compiler.

Compilers are usually implemented in different phases to seperate the different tasks and have a structured approach. A common approach to structure a compiler is by having a front end (section 1.2.1), an optional optimization (section 1.2.2) and a back end phase (section 1.2.3). These phases are explained in more detail in the following sub sections.

1.2.1 Front End

The front end phase is the first step in the translation process. Its implementation is dependent on the source language that has to be translated. A typical front end includes a scanner, a syntax checker/parser, a context-sensitive analysis and translation into a Intermediate Representation (IR). The scanner translates a stream of characters into a stream of tokens that are classified as parts of the source language. These tokens are then taken by the parser and are checked against the grammer defined by the source language. Even with a syntactically correct program, there can still be errors in the code, e.g., assignments of incompatible types. These are checked during the context-sensitive analysis phase. Eventually, the source code is translated into some kind of a IR which will be used as input to the optimizer and the back end.

There might be additional steps required depending on the source language. C/C++ compilers, for example, use a preprocessor to replace macros like **`#include`** and **`#define`** with their actual values.

1.2.2 Optimizer

The usage of a IR, not only abstracts away the source language and the target hardware, but also permits to apply more passes in the compilation process. These addition passes transform IR to IR. Note that this step is optional and not required to produce correct translations. The purpose of this step is to optimize the code, in a source and target independent manner, for more efficient execution. Efficient can mean different things here. The transformed code might produce, e.g., a faster program, a program that is smaller in size, or a program with less power consumption. There exist a large amount of optimization passes in most compilers. They range from rather simple optimizations, like replacing constant variables with their actual value, to more advanced ones that might, for example, simpliy computations with the rules of algebra.

1.2.3 Back End

The compilers back end takes the IR as input and emits code for the target hardware and decides which variables will reside in registers only and which ones reside in memory. This last step consists of three main tasks.

The first step is the instruction selection. Instruction selection translates the operations of the IR to operations of the Instruction Set Architecture (ISA) of the target hardware. The next steps, instruction scheduling and register allocation are explained in a little more detail as they represent the main topic of this thesis.

Instruction Scheduling

Add Example: e.g. see `https://youtu.be/brpomKUynEA?t=271`
Probably best to introduce an example in the section 1.1 and reuse/extend it here.

Register Allocation

The usage of an infinite amount of virtual registers in the IR was ignored in the instruction selection step.

1.3 LLVM Compiler Infrastructure

1.3.1 Intermediate Representation

1.3.2 Instruction Selection DAG

Selection DAG

1.3.3 Pre-RA-Scheduling

Welche gibt es?
Wie funktionieren sie?
Welche Infos nutzen sie?

1.3.4 Post-RA-Scheduling

1.4 Reinforcement Learning

2 Related Work

2.1 Instruction Scheduling

Code scheduling and register allocation in large basic blocks

[goodman1988code]

Learning Instruction Scheduling Heuristics from Optimal Data

[russell2006learning]

Learning to schedule straight-line code

[moss1998learning]

Using Genetic Algorithms to Fine-Tune Instruction-Scheduling Heuristics

[beaty1996using]

2.2 Register Allocation

Deep Learning-based Hybrid Graph-Coloring Algorithm for Register Allocation

[das2019deep]

Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems

[lemos2019graph]

Register Allocation for Intel Processor Graphics

[**chen2018register**]

2.3 Compiler Phase-Ordering

Autophase: Compiler phase-ordering for hls with deep reinforcement learning

[**huang2019autophase**]

2.4 Code Representation

For making use of data driven techniques in the area of compiler optimization, it is required to somehow extract features from the code to make it accessible for data driven algorithms. Older works usually made use of approaches that used hand-tuned features.

Maybe add referer used in https:/ thesis. (3.3.2.1

Recent works are inspired by the advances in the the field of Natural Language Processing (NLP), which are caused by neural networks and continuous distributed vectors (referred to as embeddings) e.g., , word2vec [**mikolov2013efficient**]. Although, human language is different from codes of programming languages in many aspects, embeddings prove to be useful in code related tasks, too.

Code inputs may be used directly in a high-level programming language or in an IR (e.g., , LLVM-IR [**LLVM:CGO04**]). The advantage of using an IR is that it is independent of the source programming language and the target architecture.

Most approaches for representing high-level language code use some sort of the Abstract Syntax Tree (AST) in combination with various learning mechanisms. Alon et al. [**alon2019code2vec**] used paths of the AST in combination with a Attention Neural Network model. Others have used the AST in combination with Gated Graph Neural Networks [**ye2020deep**, **allamanis2017learning**], with Support Vector Machines [**park2012using**] or with Long short-term memory (LSTM) Networks for tree structures [**dam2018deep**].

With Neural Code Comprehension (inst2vec) [**ben2018neural**], Ben-Nun et al. defined an embedding space for the LLVM-IR. Relevant information to discover code semantics are data and control flow. To emphasize the semantics, the data and control flow are represented in a novel graph structure, called Contextual Flow Graphs (XFGs). The context of an individual statement, with size N, is defined as the statement and its graph neighbors that are connected by a path of length N. This statement is then mapped to its embedding by using the skip-gram model [**mikolov2013distributed**], which are known to work good in NLP tasks. The XFG captures features like data and control dependence's, instructions and data types, which are important for our task.

ProGraML: Graph-based Deep Learning for Program Optimization and Analysis

[**cummins2020programl**]

Write RW text for ProGra

Compiler-based graph representations for deep learning models of code

[**brauckmann2020compiler**]

IR2Vec [**keerthy2019ir2vec**] is another approach that maps an IR to a embedding space. [...] However, the datatype size, which is important for code optimizations, is abstracted away during the embedding process.

Find Paper PDF and write text

2.5 Applied Machine Learning on Code

Write roughl how IR2Vec works

Ithemal Accurate, portable and fast basic block throughput estimation using deep neural networks

[**mendis2019ithemal**]

NeuroVectorizer: End-to-End Vectorization with DeepReinforcement Learning

[**haj2020neurovectorizer**]

From Loop Fusion to Kernel Fusion: A Domain-Specific Approach to Locality Optimization

[**qiao2019loop**]

A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs

[**nemirovsky2017machine**]

3 Approach

3.1 Breaking down the problem (maybe just chapter introduction)

- Schedulers run on basic blocks -> Select the most executed (hottest) BB's

3.2 Experimentation Pipeline

Explain and illustrate the pipeline incl. injection of timers and counters

3.3 Benchmarking

- Benchmarking methods: Instrumentation, sampling -> only instrumentation makes sense here
- Where to inject timer in BB DAG
  - We only want to measure optimized code, but not too kleinteilig because of lacking accuracy. Problematic code is IO code
  - Time whole function?
  - Time only relevant BB's with surrounding ones (e.g., loop headers) -> Where exactly place the the timer
  - Provide data and examples to demonstrate decision making

3.3.1 Implementation

Injection of the `std::chrono::high_resolution_clock`

Potential exploration -> random scheduling

Compare speedup with complexity of the problem (number of possible schedulings) vs speedup

Compare CPU Architectures, In-Order vs Out-Of-Order (`https://en.wikipedia.org/wiki/Out-of-order_execution`)

Acronyms

AST Abstract Syntax Tree

NLP Natural Language Processing

ISA Instruction Set Architecture

IR Intermediate Representation

XFG Contextual Flow Graph

LSTM Long short-term memory