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

1 Background

In this chapter we are introducing knowledge which is required to understand the content of this thesis. In section 1.1 we give an overview about how a compiler is typically implemented and which problems it has to solve that we want to tackle.

explain also the other sections

Motivation?

[10] state the important interdependence between instruction scheduling and register allocation.

1.1 Compilers

Making computer programs, written in high-level programming languages, 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 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.

The pure translation of the program is only one of the tasks a compiler has to fulfill.

What are the different tasks a compiler has?
How are compilers usually implemented? (Front-End, ...)
Front End
Optimization
Back End

Instruction Scheduling
Register Allocation

1.1.1 Instruction Scheduling

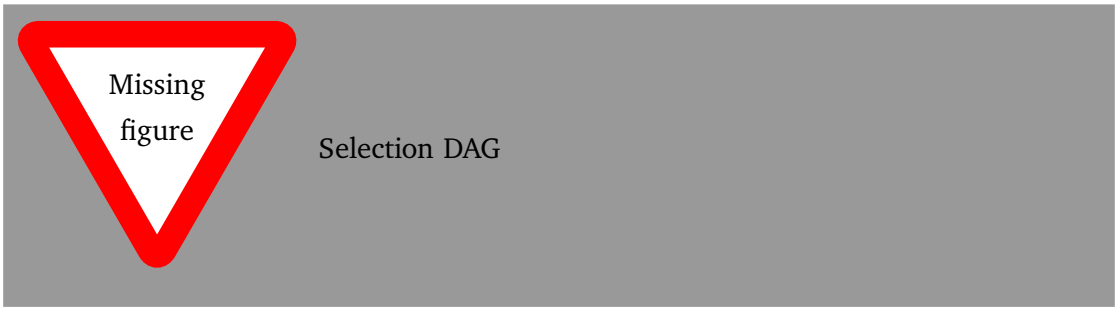
Add Example: e.g. see `https://youtu.be/brpomKUynEA?t=271`

1.1.2 Register Allocation

1.2 LLVM Compiler Infrastructure

1.2.1 Intermediate Representation

1.2.2 Instruction Selection DAG



Selection DAG

1.2.3 Pre-RA-Scheduling

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

1.2.4 Post-RA-Scheduling

1.3 Reinforcement Learning

2 Related Work

2.1 Instruction Scheduling

Code scheduling and register allocation in large basic blocks

[10]

Learning Instruction Scheduling Heuristics from Optimal Data

[23]

Learning to schedule straight-line code

[19]

Using Genetic Algorithms to Fine-Tune Instruction-Scheduling Heuristics

[3]

2.2 Register Allocation

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

[9]

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

[15]

Register Allocation for Intel Processor Graphics

[6]

2.3 Compiler Phase-Ordering

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

[12]

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 references used in https://chriscu thesis.pdf (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 [18]. 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 Intermediate Representation (IR) (e.g., LLVM-IR [14]). 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. [2] 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 [24, 1], with Support Vector Machines [21] or with Long short-term memory (LSTM) Networks for tree structures [8].

With Neural Code Comprehension (inst2vec) [4], 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 [17], 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

[7]

Write RW text for ProGraML

Compiler-based graph representations for deep learning models of code

[5]

Find Paper PDF and write text

IR2Vec [13] 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.

Write roughly how IR2Vec works

2.5 Applied Machine Learning on Code

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

[16]

NeuroVectorizer: End-to-End Vectorization with DeepReinforcement Learning

[11]

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

[22]

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

[20]

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

IR Intermediate Representation

XFG Contextual Flow Graph

LSTM Long short-term memory

Bibliography

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. “Learning to represent programs with graphs”. In: *arXiv preprint arXiv:1711.00740* (2017).

[2] Uri Alon et al. “code2vec: Learning distributed representations of code”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[3] Steven J Beaty, Scott Colcord, and Philip H Sweany. “Using genetic algorithms to fine-tune instruction-scheduling heuristics”. In: *Proceedings of the international conference on massively parallel computer systems*. 1996.

[4] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. “Neural code comprehension: A learnable representation of code semantics”. In: *Advances in Neural Information Processing Systems* 31 (2018), pp. 3585–3597.

[5] Alexander Brauckmann et al. “Compiler-based graph representations for deep learning models of code”. In: *Proceedings of the 29th International Conference on Compiler Construction*. 2020, pp. 201–211.

[6] Wei-Yu Chen et al. “Register allocation for intel processor graphics”. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pp. 352–364.

[7] Chris Cummins et al. “ProGraML: Graph-based Deep Learning for Program Optimization and Analysis”. In: *arXiv preprint arXiv:2003.10536* (2020).

[8] Hoa Khanh Dam et al. “A deep tree-based model for software defect prediction”. In: *arXiv preprint arXiv:1802.00921* (2018).

[9] Dibyendu Das, Shahid Asghar Ahmad, and Kumar Venkataramanan. “Deep Learning-based Hybrid Graph-Coloring Algorithm for Register Allocation”. In: *arXiv preprint arXiv:1912.03700* (2019).

[10] James R Goodman and W-C Hsu. “Code scheduling and register allocation in large basic blocks”. In: *ACM International Conference on Supercomputing 25th Anniversary Volume*. 1988, pp. 88–98.

[11] Ameer Haj-Ali et al. “NeuroVectorizer: end-to-end vectorization with deep reinforcement learning”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020, pp. 242–255.

[12] Qijing Huang et al. “Autophase: Compiler phase-ordering for hls with deep reinforcement learning”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 308–308.

[13] Venkata Keerthy S et al. “IR2Vec: A Flow Analysis based Scalable Infrastructure for Program Encodings”. In: *arXiv* (2019), arXiv–1909.

[14] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation”. In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[15] Henrique Lemos et al. “Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems”. In: *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2019, pp. 879–885.

[16] Charith Mendis et al. “Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 4505–4515.

[17] Tomas Mikolov et al. “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems* 26 (2013), pp. 3111–3119.

[18] Tomas Mikolov et al. “Efficient estimation of word representations in vector space”. In: *arXiv preprint arXiv:1301.3781* (2013).

[19] J Eliot B Moss et al. “Learning to schedule straight-line code”. In: *Advances in Neural Information Processing Systems*. 1998, pp. 929–935.

[20] Daniel Nemirovsky et al. “A machine learning approach for performance prediction and scheduling on heterogeneous CPUs”. In: *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE. 2017, pp. 121–128.

[21] Eunjung Park, John Cavazos, and Marco A Alvarez. “Using graph-based program characterization for predictive modeling”. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. 2012, pp. 196–206.

[22] Bo Qiao et al. “From loop fusion to kernel fusion: a domain-specific approach to locality optimization”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 242–253.

[23] Tyrel Russell. “Learning Instruction Scheduling Heuristics from Optimal Data”. MA thesis. University of Waterloo, 2006.

[24] Guixin Ye et al. “Deep Program Structure Modeling Through Multi-Relational Graph-based Learning”. In: *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 2020, pp. 111–123.