Towards Inductive Synthesis of Compiler Heuristics: A Case Study with Register Allocation

Mohammad Ali
Texas State University
San Marcos, TX
mohammad.ali@txstate.edu

Apan Qasem
Texas State University
San Marcos, TX
apan@txstate.edu

ABSTRACT

There have been significant advances in machine learning-driven performance modeling in recent years. One key limitation of such approaches is that their success depends, to a large degree, on the formulation of the *outcome* or *objective*, which is typically done by human experts. In this paper, we propose a novel approach to automatically generating new optimization heuristics using inductive program synthesis. To explore the feasibility of this approach, we investigated the graph-coloring register allocation heuristic used in state-of-the-art compilers today. In particular, we focused on the task of live range splitting. The results show that when using a Genetic Algorithm, we can obtain splitting heuristics that are within 10% of the optimal split after 202 generations.

1 INTRODUCTION

Machine learning has emerged as an effective strategy for performance modeling, code optimization, and autotuning [2, 5, 9, 10, 17, 18]. In ML-driven performance modeling, a human expert first identifies a candidate optimization (e.g., vectorization) and a set of controlling parameters for the optimization (e.g., interleaving factor). A model, supervised or unsupervised RL, is then trained to learn the relationship between program features (e.g., dependencies), the execution environment (e.g., vector register size), and the optimization parameters. When a new program is fed into the model, it draws from its learning to predict the parameter values that will maximize the performance of the input program. At a high level, the model formulation can be generalized as

Given a representation of a program as input, predict the optimization parameter(s) that maximizes performance

Although this approach has been proven successful, it has one key limitation: the effectiveness of the model hinges, to a large degree, on the choice of the initial parameters. For optimizations like vectorization [8] or prefetching [11, 13, 15] the choice is straightforward. But for many critical low-level optimizations, such as register allocation and instruction scheduling, identifying suitable control parameters is extremely difficult and in some cases infeasible [4, 14, 16]. This is because the impact of such optimizations is determined by *many* small decisions embedded within the heuristic, which creates an exponential space of parameter configurations.

To address this limitation of ML-based models, we propose a new approach of automatically generating compiler optimization heuristics. First, we reformulate the predictive modeling task as follows

Given a program representation and a specification of an optimization task as input, generate an algorithm for that optimization that maximizes performance

Intuitively, instead of telling us **what** the best parameter values are, this new formulation tells us **how** to get to the best values. We attempt to build such a model using inductive program synthesis, in which a new program (or heuristic) is inferred from a given set of input-output examples [3]. Although technically, any optimization can be expressed as a transformation over a control-flow graph, we focus on data flow and optimization problems that are intractable and are solved using heuristics in practice.

In this poster, we investigate the register allocation problem, one of the most challenging optimization tasks in compilers [6, 12]. Register allocator in current compilers applies a complex set of heuristics for live range splitting and coalescing, eviction, spilling, and coloring [7, 16]. We developed a framework to synthesize register allocation heuristics comprising the above tasks [1]. Given an SSA-live range and an interference graph, the inductive model attempts to generate a graph transformation algorithm that is most likely to minimize the spill cost. In the interest of space, we limit the discussion to the task of live range splitting only.

2 METHODOLOGY

Fig. 1 gives an overview of our framework. We define the synthesis of a live range (LR) splitting algorithm as follows

Given a set of input-output samples, generate a program that transforms an input LR graph through a sequence of split operations, to produce an output LR graph, which is most likely to minimize spill cost

The input-output samples represent SSA graphs with embedded LR information. The input represents the LRs of the original program while the output represents the LRs after applying a sequence of split operations. To create a large set

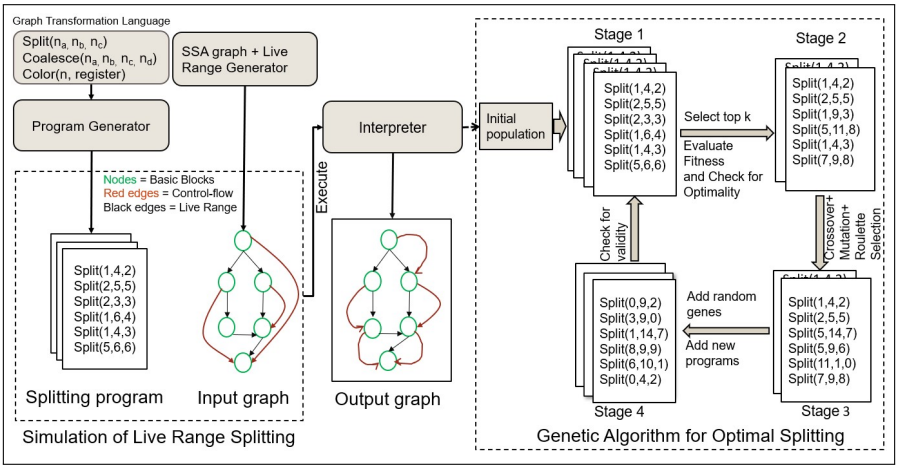


Figure 1: Overview of inductive synthesis framework

of input samples, we developed CGG, **a compiler graph generator** that can generate many of the graphs used in compiler analysis and optimization, including SSA, live range, and interference graphs. CCG is capable of generating arbitrary graphs for each type. Here, we generate LR graphs, varying the length, overlap, and number of live ranges.

Output samples are generated by applying a sequence of graph transformation operations on the input graph. To this end, we define GTL, a **Graph Transformation Language** to express the operations commonly found in compiler analysis and optimization including, meet and join, splitting, coalescing, and coloring. We develop GPG, a **Graph Program Generator**, that can generate arbitrary legal programs in GTL. We build an **interpreter** which, given an input graph generated by CGG and a program generated by GTL, applies the program to the input graph and produces the resulting output graph. A program consists of a sequence of $Split(n_1, n_2, n_3)$ operations where n_1 and n_2 represent nodes denoting the endpoints of the original LR and n_3 denotes the node around which the LR is going to be split.

To synthesize, we employ a Genetic Algorithm. In a GA formulation, a GTL program is an *individual* with a single *chromosome* and a split is a *gene*. The *fitness* score is calculated as a function of the number of LRs and the length of each LR. Intuitively, a *chromosome* with a higher fitness score is likely to result in fewer spills. The GA starts with an initial population of N individuals and *selects* k programs for the next stage based on their fitness score. In stage 2, we apply crossover, mutation, and roulette selection on the top k programs. New genes and chromosomes are randomly inserted in stage 3 and validated in stage 4. The search converges when a program is generated that has a fitness score that falls within a pre-specified threshold of the optimal split.

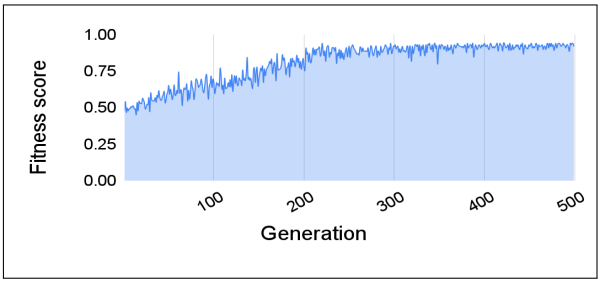


Figure 2: Average fitness score across GA generations

3 EXPERIMENTAL RESULTS

We conducted a preliminary evaluation of the proposed approach with a series of experiments in which we varied the characteristics of the LR graphs and the GTL programs, and the parameters of the GA. Fig 2 shows results from one set of experiments in which the number of overlapping LRs varied between 5-10 (avg. for SPEC17 [16]) and the number of GA generations was capped at 500. We observe that the GA results stabilize after about 210 generations and about 90% of the optimal split can be achieved after 202 generations.

4 CONCLUSIONS AND FUTURE WORK

This work provides initial evidence that inductive synthesis can be an effective tool for creating new heuristics for complex code optimizations. In the long run, the synthesis approach has the potential to create brand-new code transformations that human experts have yet to conceive. Our future plans include extending the framework to tackle other optimization and analysis tasks such as value numbering and instruction scheduling. We also plan to replace GA with more sophisticated techniques such as the Multi-Armed Bandit.

REFERENCES

[1] Mohammad Ali. 2023. Register Allocation Synthesis. https://git.txstate.edu/pum8/Register-Allocation-Synthesis.

[2] Amir H Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A survey on compiler autotuning using machine learning. *ACM Computing Surveys (CSUR)* 51, 5 (2018), 1–42.

[3] Kory Becker and Justin Gottschlich. 2021. AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Lille, France) *(GECCO '21)*. Association for Computing Machinery, New York, NY, USA, 1513–1521. https://doi.org/10.1145/3449726.3463125

[4] John Cavazos and J. Eliot B. Moss. 2004. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. 183–194. https://doi.org/10.1145/996841.996864

[5] Tiffany A. Connors and Apan Qasem. 2017. Automatically Selecting Profitable Thread Block Sizes for Accelerated Kernels. In *19th IEEE International Conference on High Performance Computing and Communications (HPCC17)*. 442–449. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.58 [Acceptance Rate: 36%].

[6] Keith D Cooper, Anshuman Dasgupta, and Jason Eckhardt. 2006. Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms. In *Languages and Compilers for Parallel Computing: 18th International Workshop, LCPC 2005, Hawthorne, NY, USA, October 20-22, 2005, Revised Selected Papers 18*. Springer, 1–16.

[7] LLVM Research Group. 2007. The LLVM Compiler Infrastructure Project. http://http://llvm.org/.

[8] Ameer Haj-Ali, Nesreen K. Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: End-to-End Vectorization with Deep Reinforcement Learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO 2020)*. Association for Computing Machinery, New York, NY, USA, 242–255. https://doi.org/10.1145/3368826.3377928

[9] Jacob M. Hope, Mikel Gjergji, Johana Di Girolamo, Marco A. Alvarez, and Apan Qasem. 2021. Characterizing Input-sensitivity in Tightly-Coupled Collaborative Graph Algorithms. In *21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, Melbourne, Australia, May 10-13, 2021*, Laurent Lefèvre, Stacy Patterson, Young Choon Lee, Haiying Shen, Shashikant Ilager, Mohammad Goudarzi, Adel Nadjaran Toosi, and Rajkumar Buyya (Eds.). IEEE, 287–296. https://doi.org/10.1109/CCGrid51090.2021.00038 [**Best Paper Award**].

[10] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[11] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. 2009. Machine Learning-Based Prefetch Optimization for Data Center Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. 56:1–56:10.

[12] Apan Qasem and Samuel Teich. 2017. Mitigating register pressure in GPU kernels for improved energy efficiency. In *Eighth International Green and Sustainable Computing Conference, IGSC 2017, Orlando, FL, USA, October 23-25, 2017*. IEEE Computer Society, 1–7. https://doi.org/10.1109/IGCC.2017.8323599

[13] Saami Rahman, Martin Burtscher, Ziliang Zong, and Apan Qasem. 2015. Maximizing Hardware Prefetch Effectiveness with Machine Learning. In *17th IEEE International Conference on High Performance Computing and Communications (HPCC15)*.

[14] Biplab Saha, Tiffany A. Connors, Saami Rahman, and Apan Qasem. 2017. A Machine Learning Approach to Automatic Creation of Architecture-sensitive Performance Heuristics. In *19th IEEE International Conference on High Performance Computing and Communications (HPCC17)*. 18–25. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.3 [**Best Paper (1/176)**].

[15] A. Sethia, G. Dasika, M. Samadi, and S. Mahlke. 2013. APOGEE: Adaptive Prefetching On GPUs for Energy Efficiency. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*. 145–156. https://doi.org/10.1109/PACT.2013.6618805

[16] S. VenkataKeerthy, Siddharth Jain, Anilava Kundu, Rohit Aggarwal, Albert Cohen, and Ramakrishna Upadrasta. 2023. RL4ReAl: Reinforcement Learning for Register Allocation. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/3578360.3580273

[17] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[18] Nan Wu and Yuan Xie. 2022. A survey of machine learning for computer architecture and systems. *ACM Computing Surveys (CSUR)* 55, 3 (2022), 1–39.