International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

A Survey of Adaptive Compiler Optimization Heuristics

Aamir Syed[1], Ashwin Harish[2], Keerthana Purushotham[3], Sini Anna Alex[4]
[1,2,3]*Student, Dept. of Computer Science and Engg., M. S. Ramaiah Institute of Technology, Bangalore, India*
[4]*Assistant Professor, Dept. of Computer Science and Engg., M. S. Ramaiah Inst. of Tech., Bangalore, India*

***Abstract*: This survey paper aims to illustrate the multiple problems and their innovative solutions in the fields of compiler optimization, performance and error removal and large scale database transaction abstractions common in Compiler Design heuristics. The performance of the code generated by a compiler depends on the order in which the optimization passes are applied. In the context of high-level synthesis, the quality of the generated circuit relates directly to the code generated by the front-end compiler. In simpler terms, choosing a good order of sequences is the phase ordering problem. The compiler can't switch to finding the most optimal solution, since it is an NP-Hard problem. However, sub optimal heuristics are employed to capitalize on this criteria. While this is the problem that is at the core of compiler designing, compilation errors set an equally challenging constraint to the programmer. To combat this, multiple heuristics have been employed with the help of Machine Intelligence, offering a reduction in resource consumption. Another chief area of interest is the exhibition of inefficient executions dominated by massive memory stalls in online transaction processing. To this end, profile-driven compiler optimizations to revamp the code layout in commercial workloads provide a massive improvement in instruction cache behaviours. This paper surveys novel methods using Machine Learning, Data Mining, Pattern Analysis and Natural Language Processing, to reduce the state-space for searching the most optimal sub-sequences.**

***Keywords*: Cache Behaviour, Data Mining, Machine Learning, Natural Language Processing, Optimization, Survey**

1. Introduction

It has always been a struggle for the programmers to find and code in the most efficient way possible, or to code in the most optimized of manners. This has a plethora of real life impacts, with multiple languages surfacing as functional programming takes over the design process. One such example is the use of databases in online platforms, with a powerful backend to crunch massive amounts of data. This has become even harder with the availability of multiple compilers, making the paradigm shift very imminent. An optimized machine outperforms others in any given setting, and this criteria extends to the compiler, whose code generation decides the working process. To this end, we identify three unique problems in efficient compiler designing:

1. Finding Effective Code Sequences.
2. Resolving and Correcting Errors
3. Efficient Code Layout Optimization in Web deployments

To this end, 3 papers in particular have been surveyed, where each paper deals with a specific part of the problem. This paper is broadly divided into 3 categories, where the topics are finding effective code sequences, resolution and correction of errors, and code layout optimization in web deployments. All of the subtopics of interest are explored in some detail.

2. Literature survey

A. Finding effective code sequences

The first paper by Ameer Haj-Ali et. al. deals with compiler optimization in sequence passes and high-level synthesis. Essentially, it is a phase ordering problem that can be solved using Machine Learning and Reinforcement Learning. Recent breakthroughs and advancements in the fields of Machine Intelligence has pushed the idea of Deep Reinforcement Learning for a solution over the phase ordering problem. As is a standard Reinforcement Learning approach, a learning agent takes an action on the observed the state of the environment. The ultimate goal for any such agent is to compute a mapping policy between state of the environment and viable actions for the maximal long term reward. The main two long term rewarding approaches used in the paper are Deep Q-Network (DQN)and Policy Gradient (PG). The paper delves into analysis passes to extract as many as 56 static features from an intermediate representation of the LLVM program. The features were defined as number of blocks, instructions, branches, etc. The RL states are represented as a series of penalties and rewards for either the features extracted or a graph of applied passes to a Reinforcement Learning Agent. A reward is added onto the agent's working everytime a sequence of passes performs better than -O3 within a reasonable amount of time. 12 HLS benchmarks were taken from multiple surveys and batches for high-level synthesis. The HLS metric is a product from the LegUp material referenced in the paper. Upon comparisons with state-of-the-art approaches like Greedy Algorithms, Genetic Algorithms, Random Search (of state space), -O3, DQN and PG both outperformed the former, with DQN reaching similar results for multiple sequence lengths. With Genetic Algorithms matching the same circuit speedup of 16%, RL methods proved to be almost 3x faster, making them more viable for the compiler sequences. Moreover, retraining

agents for multiple systems holds the ability to improve runtime even further.[1][4]

B. Resolution and correction of errors

The paper by Khushali et. al. deals with the second problem in particular, taking care of the errors generated due to human error. The first of the two papers deals with correction and mitigation of errors using hashtags for comparison with a faulty / error ridden program. It delves into the use of programming practices in most academic institutions, which leads to discombobulation for newcomers into the field of programming. Correction of such errors is an issue of the highest importance, and a resource consuming process. The system proposed in the paper deals with correction of compile time errors using Data Mining and Machine Learning techniques. All programs are analyzed and broken into classes in a given database, where all logically incorrect programs are compared with to form pattern sequences using hashtags. The incorrect programs are then analyzed using Machine Learning and suggestions to correct them are given. A rule table comparison is mentioned as the inception for the pattern analysis approach used in the paper, while programming properties and pointer dereferences form the major part of suggestion building using Machine Learning. Control Flow Graphs (CFG), SVMs, Decision Trees, etc are used for the predicate analysis in program statements. The paper delves into heuristics for each of the building practices using Machine Learning. The latter three referenced developments include automated detection of logical errors using source code, logical error correction systems using Genetic Algorithms with statistical CFG techniques, and dependency analysis for precision increase. Fuzzy Logic with Apriori-All, Expression mining, analyzing data dependencies and merge rules form the majority of concepts in the respective developments. Logical errors are broached upon, with missing invariants forming the base for a solution using Data Mining techniques. Each program is visualized as a step of functions, and suggestions are given via a Machine Learning algorithm using profiling and previous data. Modification and code embeddings are suggested to the user, where code is embedded by replacing the whole code. The proposed system in the paper forms a 7-part sequence: Construction of the Compiler, Programming Construction, Comparing the programs, Deducing the errors, Classifying the errors, Recommending the right solution, and Embedding the correct solution. The first part of the sequence explains the architecture of the compiler, and software efficiency of it. The next part deals with comparison with the correct source program, focusing on modularity of code and logic comparison. The third part compiles the correct source program and the current incorrect one, upon differences of which takes 2 distinct operations. If the programs follow a different line of logical building, incorrect program is replaced by the source code without showing any errors. Else, the missing code is transferred to the data mining device (DMD) for base profiling. The errors are then deduced, missing part of the code is designated using data mining and profiling. This forms the 4th step of the series. The next step deals with classification of errors into logical, syntactical, and runtime errors. Machine Learning is used to get the syntactical and runtime error for suggestions, while logical errors require description of each element in addition to the Machine Learning program. Moreover, they need to be processed individually since each solution of a problem may consist of multiple correct logical iterations. Each new logic is stored in the database (db) using a hashtag, which serves as the reference for storing/retrieving them. They have 3 inherent properties, namely, uniqueness, individual definition for programs with unique logic, and case sensitivity. Depending on the type of error, the right solution is identified and suggested to the user. If the user makes changes and runs the program without the errors, the change is learnt by the machine for future suggestions. This marks the end of the 6th iteration of the sequence. The final iteration involves embedding of the correct solution in a program, which is done in the form of a macro or a new function, depending on the requirement of the user. The advantages and disadvantages of such a machine are later discussed in the paper [2], [3].

C. Code layout optimization in web deployments

Ramirez et. al. illustrates and develops an efficient approach to solving memory stalls in online database transactions. As the paper states, many architecture level heuristics have been employed to increase the efficiency of the operations, even with highly complex system designs. Large instruction and data footprints initiate such behaviours, and little work has been done to improve the underlying code layout for compiler-level optimizations. The paper concludes that code layout optimizations account for a marked improvement of instruction cache behaviour, providing a marked reduction of application misses, upto 55% - 65% in 64-128K caches. Another interesting point of note is that a significantly large number of instructions misses are because of self-interference. The overall performance with optimizations is improved by 1.33 times in the execution time of the workload. All the code layout algorithms mentioned are profile driven, the optimizations being implemented in the context of Spike. Pixie or DCPI [5] have been used to collect basic block execution counts. Spike builds the Control Flow Graphs (CFG) for all procedures, and the call graph for the program. Call graphs include edges or branches between procedures, where the edge weight of the branch is determined by number of executions of the basic block. For CFGs, the control flow edges are weighted, weights being estimated from the basic block counts. The three main layout algorithms discussed here are Basic Block Chaining, Fine-Grain Procedure Splitting, and Procedure Ordering. Basic Block chaining is the idea of a simple greedy algorithm, where all flow edges are sorted with respect to their weights in descending order. Each flow edge has a source and a destination block, which can be chained together if they don't have a successor and a predecessor block respectively. The algorithm

removes unconditional branches that are frequently executed, and the chaining completes every edge has been processed. This procedure yields one or more chains, which are sorted again by execution count on the first basic block. All the chains are placed in decreasing order, with the chain containing the procedure entry block being placed first. Fine-Grain Procedure Splitting deals with division of chains into multiple code-segments, where each segment is substituted as a discrete procedure in Spike. This leads to a program that consists of multiple segments, (where each segment has a few basic blocks) expected to execute sequentially. This adds another degree of flexibility for follow-on procedure ordering approaches. Procedure Ordering is a sorting attempt to place related procedures adjacently or close to one another. This is a simple and direct implementation of Pettis and Hansen [6]. The next order of business is defining a profiling scheme, on which the compiler optimization problem is built. A basic workload is set up and scaled after the TCB-B benchmark. To abstract latencies, OLTP runs were configured with multiple server processes per processor, 8 in this case. The OLTP profile data was collected using Pixie, with the original binary startup of database, cache in memory, etc. The server processes that are dedicated to client request executions are 'pixified' binary. This is done to focus only on the components of the given workload that deal with transaction processing. Kernel profiles are also collected, using tools that stem from PC sampling that uses Alpha performance counters. Profile data was hence derived. To evaluate performance measures, full system simulations and direct machine measurements are both involved for execution time, instruction cache misses, TLB performance, etc. An OS environment was set up for simulation of both user and system code. With analysis and subsequent enhancements, the code layout optimization improved workload efficiency, which is attributed to optimizations in the behaviour of the instruction cache of the application. It further delves into isolated database application environments to understand instruction cache behaviours. The paper also illustrates interaction between OS instruction streams and applications [7].

3. Discussions

In our survey, we have identified that reduction of state space, frequency patterns, etc. are involved in designing an efficient compiler. Deep Learning and Neural Networks can also be used to improve compiler performance, by treating the state space of sequences as input features, and by iterating through all of them till the most efficient of them is found. By exploiting multiple avenues for the same, it was apparent that efficient tools require an efficient programmer to work with, and to that end we have delved into programming practices and correction of common program errors. The Machine Learning and Data Mining approaches to the same are being used to improve overall efficiency and performance of the compilers. This type of work can be seen put together in practical examples using web services that employ huge backends, which has been abstracted as a series of code layout algorithms. Using instruction cache optimization techniques, code layout efficiency increased, which led to an overall efficient design.

4. Conclusion

A survey on ongoing research for compiler designs and optimization shows how various subsystems together contribute to an efficient system. This involves optimization of the compiler architecture and state space itself, along with making a programmer more efficient in dealing with errors. This can only be purposeful and practical if there are real time uses for these, which can be seen affecting performance on a large scale. Hence, we see a practical approach to web ontology of compilers, and how optimization can lead to efficiency in this respect. This entails the use of available stream of Machine Intelligence heuristics like Machine Learning, Data Mining, etc., in which respect the papers have been explored and surveyed. This is to understand and leverage modern technology to improve existing designs and impact them positively. The survey paper is meant to be an aggregation of research being conducted in the above mentioned fields. It is meant only to serve as a reference and as well as a benchmark for researchers working on innovative techniques to build better heuristic approaches to design and efficiency of compilers.

References

[1] Qijing Huang, Ameer Haj-Ali, William Moses, John Xiang, Ion Stoica, Krste Asanovic, John Wawrzynek, "AutoPhase: Compiler Phase-Ordering for High-Level Synthesis with Deep Reinforcement Learning", cs. LG, Cornell University, New York, January 2019.

[2] Khushali Deulkar, Jai Kapoor, Priya Gaud, Harshal Gala, "A Novel Approach to Error Detection and Correction of C Programs Using Machine Learning and Data Mining", International Journal on Cybernetics & Informatics, Vol. 5, No. 2, April 2016.

[3] K K Sharma, Kunal Banerjee, Indra Vikas, Chittaranjan Mandal, "Automated Checking of the Violation of Precedence of Conditions in else-if Constructs in Student's Programs", IEEE International Conference on MOOC, Innovation and Technology in Education (MITE), 2014.

[4] L. K. et al., "Reinforcement learning: A survey," Journal of Artificial Intelligence Research, vol. 4, 1996, pp. 237–285.

[5] J. A. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. T. Leung, R. L. Sites, M. T. Vandervoorde, C. A. Waldspurger, and W. E. Weihl. "Continuous profiling: Where have all the cycles gone?" in Proceedings of the 16th International Symposium on Operating Systems Principles, pages 1–14, October 1997.

[6] K. Pettis and R. C. Hansen. "Profile guided code positioning." Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pp. 16–27, June 1990.

[7] "Code layout optimizations for transaction processing workloads", ACM SIGARCH Computer Architecture News, 2001.