979-8-3503-8713-1/24/$31.00 ©2024 IEEE

2024 IEEE High Performance Extreme Computing Conference (HPEC) | 979-8-3503-8713-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/HPEC62836.2024.10938458

A Graph-Based Algorithm for Optimizing GCC Compiler Flag Settings

Reza Sajjadinasab*‡, Sanjay Arora†, Ulrich Drepper†, Ahmed Sanaullah†, and Martin Herbordt*‡
*ECE Department, Boston University †Red Hat Inc. ‡{sajjadi, herbordt}@bu.edu

***Abstract*—Compiler tuning through external mechanisms — such as source code modifications, e.g., with pragmas, and adjusting compiler flags — is well-explored. Many researchers have shown significant performance improvement through different approaches, including heuristics and machine learning. Most of these approaches, however, require a few hundred iterations to converge towards an optimal answer. A number of studies have addressed this problem by reducing the number of iterations required, but we find that further improvements are still possible. In this work, we explore the optimization of GCC compiler flag settings with the goal of faster convergence. We find as an ancillary result that the effectiveness of the compilation itself is sometimes improved with respect to both code size reduction and application program execution time. The proposed graph-based approach can reduce the code size to 90% of the convergence point with 15 compilations fewer on average, i.e., the solution found after running Opentuner for many compilations. It also can reduce the execution time over the -O3 level for different versions of the Smith-Waterman and Bubble Sort by 1.2× and 5×, respectively.**

***Index Terms*—Compiler Tuning, Source Code Transformation, Source Code Optimization**

I. Introduction

A fundamental problem of computing is the automatic optimization of source code. A number of methods have been adopted; these sometimes vary depending on the optimization goal – e.g., latency, power, and/or code size – and the target hardware. Often automatic optimization is done in conjunction with optimizations applied by the programmer; usually iteratively, and sometimes following documented best practices. The goal of these processes is to produce code that is at least as good as what an expert alone can do. It is common, however, for programmers to only optimize for certain performance goals. For example, optimizing a source code to reduce the binary size, a basic concern of programmers some decades ago, nowadays, when done at all, generally depends solely on the compiler's capabilities.

In this work, the effectiveness of automatic optimization with respect to both performance, i.e. execution time and binary code size, and time to convergence are explored. The approach investigated seeks to find the best compiler flag settings to optimize a source code for a given optimization goal and may be referred to as a type of *external* compiler tuning. This type of optimization is in contrast to *internal* compiler mechanisms such as implementing a new transformation. Our motivation for this approach is as follows. While it is likely that new internal mechanisms are waiting to be found, it is also true that current compilers already have a sophisticated set of optimizations (based on decades of prior research). For example, GCC applies over 200 different optimization passes, many repeatedly, and in various sequences.

In fact, this leads to the problem of *pass selection*: i.e., that the overall benefit of these optimizations depends on what set of passes has been selected, but also that the number of possible selections is large. Compilers such as LLVM offer the flexibility of reordering the passes without internally changing the compiler. This is known as *phase reordering*. While this approach has proven effective (as shown in [1]–[3]), it also depends on the compiler internals for GCC, may be difficult to implement and maintain, and may be slow.

In practice, default optimization levels can optimize programs for either code size or execution time. Many studies, however, have shown that there are more effective ways to use compiler optimization passes than default configurations (e.g., [4]). The naive way of finding the best set of optimization flags is by brute force, but finding the best configuration in this way is obviously impracticable. Thus, many studies have tried to find a fast method using both iterative [5] and non-iterative techniques.

We propose a graph-based algorithm for optimizing GCC compiler flag settings. We address both time to convergence and improving the effectiveness with respect to the various performance types. This work has the following contributions and innovations.

A fast and effective compiler tuning algorithm for GCC for execution time improvement and code size reduction. This approach takes into account the single-state intrinsic of GCC pass optimization.

Analysis of the effectiveness of the algorithm: The approach is compared to other automatic and manual methods to understand better how much further the proposed and related compiler tuning algorithms can go.

Comparison with other possible code transformations and optimizations: A comparison has been made on different approaches used to transform and optimize a source code. Included are Large Language Model (LLM) methods and other optimizers.

II. Approaches to Optimizing Source Code

Optimization of a source code can be done for different performance goals. In this work, we optimize for binary size reduction and execution time improvement. Some of the methods typically used are as follows.

A. *Hand-tuning*

Hand tuning can be extremely effective, but requires significant expertise and effort. The products may also be brittle with respect to changes in code and target hardware.

B. *Creating New Compiler Optimizations*

Given a fixed hardware architecture and programming language, the lowest level of optimization is to implement a new optimization from scratch. Even so, however, current compilers suffer from the problem of knowing where and what optimization to apply to get the best performance. In other words, there is a need to make the existing compilers more intelligent. As described in the introduction, for a given source code this can be done in various ways: by directly managing compiler optimization passes; by inserting directives and pragmas into the source code; or by finding the best set of optimization flags and parameters.

C. *Tuning An Existing Compiler, Why GCC?*

An alternative is tuning the compiler without further modification. This can be done by finding the best compiler flags and parameters for a given source code (the approach taken here). Success is measured by whether the selection outperforms the default optimization flags, e.g., -O3. The process of finding the best set is different for different compilers. In compilers such as LLVM, the problem has a more extensive solution space because of the ease of implementing pass reordering. On the other hand, in a compiler such as GCC, optimization flags can only be selected once and the compiler decides on the order they should be applied. It *is* possible to write a plug-in for GCC that internally changes the pass ordering. However, this requires a significant understanding of the GCC compiler, is hard to maintain, and can also result in incorrect compilation.

Although LLVM has the flexibility of pass reordering (without breaking into the compiler as is necessary with GCC), it doesn't have the number or sophistication of the optimization passes in GCC [6]. Moreover, in contrast to the limitations imposed by a compiler, arbitrary phase reordering can cause code to be incorrectly compiled. Tuning the GCC thus has the potential to produce a better source code transformation.

D. *AI Transformation-Based*

The emergence of LLMs introduces new ways of transforming source code. LLMs can, potentially, "understand" what a source code is doing, which is a big step in knowing what source of optimization is needed. However, no massive dataset has the best optimization configuration for them. Thus, LLMs can only identify some possible optimizations but can find the best practice for writing source code quickly and effectively. The problem with this approach is that it is not guaranteed to produce a valid source code, and often, the resultant code is not doing what it should.

III. GCC Compiler Tuning

In this work, tuning a compiler means finding the best set of actions as input to a compiler, which will result in a better performance than the default configuration. This process can be done in various ways depending on our compiler. In this work, we are tuning the optimization flags for GCC.

A. *Problem Statement*

The problem we are trying to solve is finding the best set of optimizations from a set of possible actions that optimize a source code for a desired performance. This can be either real-time execution time or binary size. More specifically, Given a set S of possible actions, we are trying to find the best subset P^* that minimizes the loss function $\mathcal{L}$. Mathematically, this can be written as (1).

$$P^* = \arg\min_{P \subseteq S} \mathcal{L}(P) \tag{1}$$

The set S is our solution space, and it contains all the possible optimization flags. A few optimization flags need an integer or choice value, and their space is not binary. For these flags we treat each different value as a different flag.

B. *Solution Space Characteristics*

The solution space contains roughly 280 flags resulting in 280! possible solutions. With brute force obviously being impractical a more intelligent approach is needed with time to convergence being a primary concern. Another concern is that flags are not independent of one another. Some flags disable the others, and some flags are only effective if they are used alongside the others. Also, some of the flags may not affect a given source code at all.

IV. A Graph-Based Solution For Compiler Tuning

Due to GCC characteristics, where flags are not all independent, we can redefine our problem statement as the finding of the best subgraph of a fully connected graph where the graph's nodes are all possible flags. The Graph-based Flags Tuner (GBFT) considers the dependencies of flags on each other. The new problem is how to learn these dependencies.

In other words, the problem is to find the best subset of edges of a fully connected graph with self-loops. The new problem space solution is transformed from all possible flags to all possible relations between two flags. Also, self-loops are allowed because a single flag can be effective individually. A relation between two flags is selected, and both of these flags are chosen as a part of the solution. Fig. 1 shows how the selection of edges helps in choosing the subgraph.

A. *A Cyclic Epsilon Greedy Algorithm*

To find the final version of the graph that has learned the relations of the flags, some experiments on the same source code should be run and evaluated. Each experiment should be done with respect to a particular set of flags because there are no apparent (*a priori*) relations between flags; an epsilon greedy algorithm is used. This algorithm selects an ϵ between

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

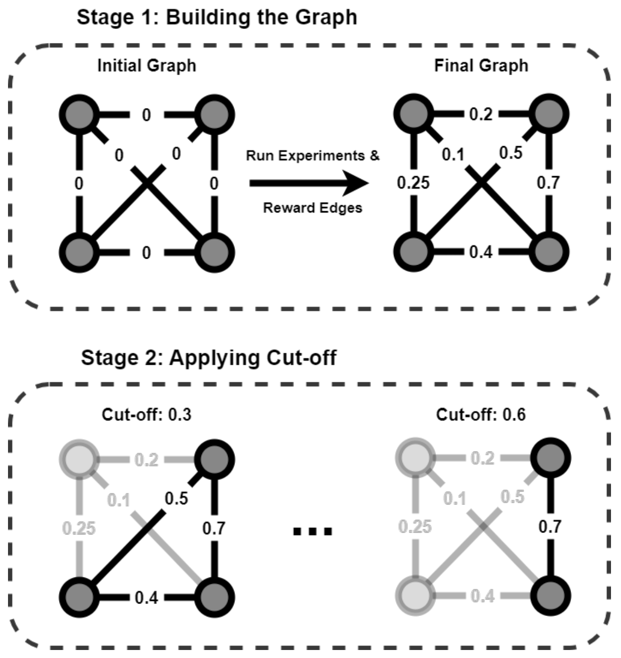

Fig. 1: Choosing the optimal subgraph that results in the optimal solution, nodes representing the compiler's flags.

```
Require: Fully connected graph $G$ with nodes representing flags
Ensure: Best set of edges $E$ representing dependencies between flags
 1: Initialize reward matrix $R$ with zeros
 2: Select $\epsilon \in [0, 1]$
 3: for each experiment do
 4:     Randomly select some flags and turn them off or on
 5:     Perform experiment and measure performance
 6:     for each edge $e_{ij}$ between selected flags $f_i$ and $f_j$ do
 7:         Update reward $R(e_{ij}) \leftarrow R(e_{ij}) + performance$
 8:     end for
 9:     for each edge $e_{kl}$ not between selected flags $f_k$ and $f_l$ do
10:         Update reward $R(e_{kl}) \leftarrow R(e_{kl}) - performance$
11:     end for
12: end for
13: for Some iterations do
14:     Pick a random number k and select kth largest edges
15:     Evaluate the performance
16: end for
```

Algorithm 1: Cyclic Epsilon Greedy Algorithm for Updating the Graph Edges

0 and 1. Then, with the probability of ϵ, a flag will be chosen to be selected or not. The rest of the flags can be set to be selected or not selected. In this work, in each iteration, a random epsilon is chosen.

After each experiment, the performance is evaluated, and a reward is given to each edge between two flags selected in the experiments. All the other edges will get a negative reward. This helps to provide a smaller reward to flags that are not effective. Algorithm 1 shows how to select the best flags using the graph-based approach suggested.

B. A Twin Graph

One problem with flag selection is the large space of possible solutions (input permutations). We have found that for an algorithm to be both effective and converge quickly, it should detect *ineffective* flags quickly. In this work, a pair of (twin) graphs are used to address these during the configuration. One is used for identifying flags set to **on** and the other is for the flags set to **off**. A negative of the reward is added to all the edges with nodes (flags) not selected in an experiment. In this way, if a flag, whether on or off, is not effective in improving or deteriorating the performance, it is expected to be neutralized through different experiments. Indeed, experimental results show that a bigger reverse reward is more effective in finding the optimal solution.

C. Finding The Cut-off

One challenge in the proposed algorithm is to find the cut-off to select the edges. In this work, within each experiment, two parameters are updated gradually, converging to a cut-off that can select highly condensed subgraphs with the highest possible number of edges. This indicates that most of the pairs in the subgraph were effective in improving the performance. Then, space exploration is done around that point to find the best cut-off and performance. The overall number of iterations reported is the sum of these two approaches' iterations.

V. LLM-Based Optimizations

Given the tremendous attention currently being paid to LLMs we decided to investigate using them as an alternate baseline. There are two different ways to use LLMs to optimize source code: prompt engineering and fine-tuning. The first approach is more straightforward because it does not require a considerable dataset or GPU capability to tune the model. Fine-tuning, however, does require a reasonably high-performance GPU for training and a labeled dataset containing the code's original version and the optimized version. Thus, it is not doable for small scales.

However, studies suggested prompt engineering can be more effective in some experiments [7]. In doing so, we tried to craft the best prompt that results in the best code in different ways:

- Giving the characteristics of a desired function and getting a well-written code;
- Giving the original code and asking the LLM to optimize it; and
- Giving the original code but changing the variable and function names to random ones.

The difference between options 2 and 3 is that it identifies whether the LLM optimizes based on the semantics of the code or just the finding of the best practice already somewhere in its database based on the function and variable names. One problem with this approach is that there is no clear way to

979-8-3503-8713-1/24/$31.00 ©2024 IEEE

ask LLM to produce code with a smaller binary size because it needs an end-to-end compilation.

VI. Methodology

One of the goals of this work is to determine whether compiler tuning is as effective as other approaches in improving performance. We are also interested in the general question of the limits of compiler tuning benefits, in particular, whether an algorithmic insight has been found. In previous optimization studies involving High Level Synthesis we have found that this can sometimes be achieved, e.g., with the FFT [8], where the butterfly circuit was generated, and Smith-Waterman [9], where a systolic array was generated.

In this work we examined the performance of two applications: four different versions of the Smith-Waterman algorithm and a bubble sort. We chose bubble sort because it is so simple and would be obvious if the code were transformed into an algorithm with $\Omega(n \log(n))$ complexity. We chose Smith-Waterman because we have recently taken a deep dive [10] into hand optimizing these codes using, e.g., a number of data reorganizations. Our result was an improvement of $6\times$ over the naive code and we believe currently the fastest existing version.

To compare the code size reduction, we used GNU core utilities. Reducing the binary size for this application is useful because it can reduce the size of the Linux kernel, which is currently of great interest.

Code size reduction measurement has no error, and we can report fixed numbers for improvement. In contrast, real-time performance measurement has errors. Thus, we report the statistical mean and distribution of evaluations. To do this, we used Hyperfine to report the improvement in performance and its variance [11].

To compare our approach with another existing approach, we used the Intel compiler, ChatGBT 4, and OpenTuner [12] as another tuning model to compare our results against it.

One consideration of this work is that the proposed algorithm has two stages. The first stage is used to build the graphs, and the next stage is to find the cut-off. The reported data for this algorithm starts from the iteration where the graph has been built and stopped with a given criteria, i.e., finding an approximate cut-off with a reasonable performance.

VII. Results

The result section is split into two major results: binary size improvement and execution time improvement. Size improvement is a better metric for comparing the convergence speed of the two non-deterministic algorithms with variations within each run. The execution time evaluation is done to find the answer to the question of whether compiler tuning can make a qualitative change or not. Thus, it is compared against other compilers and ChatGBT-generated codes.

A. Code Size Improvement

To report the result, we used five random applications from GNU core utilities within the Linux kernel. Improving their binary size will help the Linux kernel become smaller, so it is of significant importance.

The focus of this section is to show how fast our algorithm can find the optimal solution. Thus, we are looking into iterations before the 100th. More evaluations with longer iterations show all of the benchmarks have a peak answer between $8 - 13\%$ improvement over -Oz which is found in at least one of the experiments within 100 iterations. Fig. 2 shows what portion of the test can result in an improvement higher than certain thresholds. This is compared against OpenTuner. As shown in Fig. 2, GBFT in almost all of the iteration intervals has a higher chance of finding an answer with higher improvement than OpenTuner. In order to show the generalizability of the algorithm, the algorithm was run for all the GNU core utilities, and their performance was collected. Fig. 3 shows the distribution of tests reaching 90% of the convergence point. The convergence point is the point at which it converges after running the Opentuner for many iterations to tune the performance (code size). Not all of the tests could reach 90 % of the convergence point within the first 100 iterations. 63% of Opentuner tests failed to reach the threshold, whereas 50% of GBFT tests failed to reach the threshold. Also, for iterations finished before 100 iterations, the average iterations to convergence for GBFT is 16% better than Opentuner.

B. Execution Time Improvement

Execution time improvement is different than code size improvement in some sense. The improvement can be due to many factors depending on the application and hardware. One purpose of showing the results in this section for different applications is to investigate the effectiveness of compiler tuning in performance improvement. The question is whether a classic compiler like GCC can change the algorithm of an application. To this end, an in-depth comparison has been made between the different Smith-Waterman versions. To see how much improvement is achievable using other approaches. Four different versions of Smith-Waterman have been used as a benchmark. Each version is suitable for specific alignment score calculations and requires different heuristics to be improved further. Fig. 4 shows the results for Smith-Waterman versions against the expert hand-tuned version, Intel compiler, GCC, and ChatGBT 4o prompt-engineered (ChatGBT) version. As shown, the hand-tuned version has an improvement of $5.6\times$, while the compiler-tuned version has the least possible improvement of $1.2\times$ for three versions. Although this is not the best improvement that can be achieved, the interesting point is that compiler tuning can be done for any available source code, including the hand-tuned version, so it is of importance, whereas other approaches, such as the Intel compiler, can not result in a significant performance improvement on the hand-tuned version.

Also, the problem with the LLM approach is that it could only produce one correct version with no improvement. Any further changes resulted in a wrong code that was no longer doing the same thing. Also, it does not understand what the

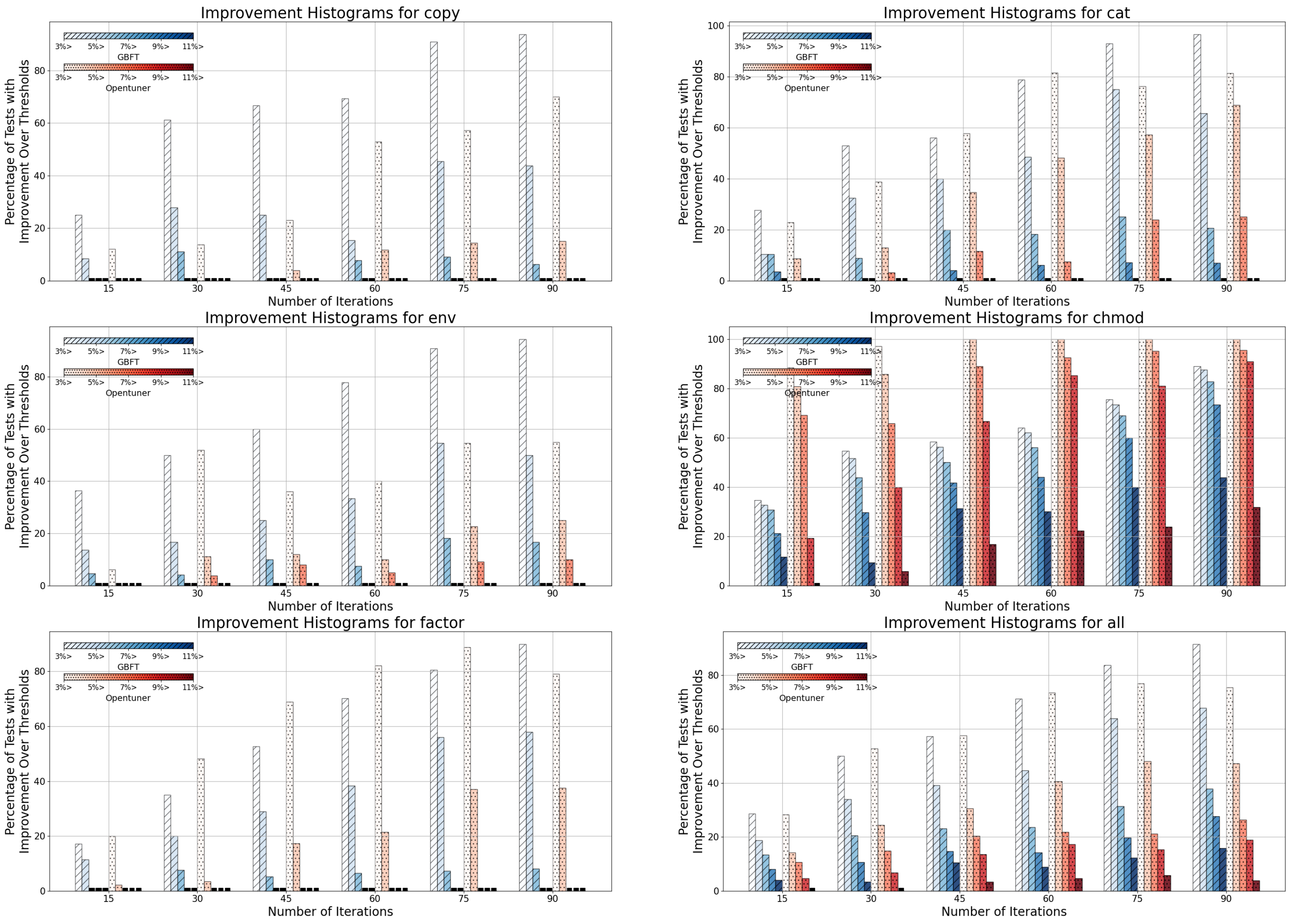

Fig. 2: Histogram of experiments with improvement of 3%, 5%, 7%, 9%, and 11% over different iterations for code size improvement.

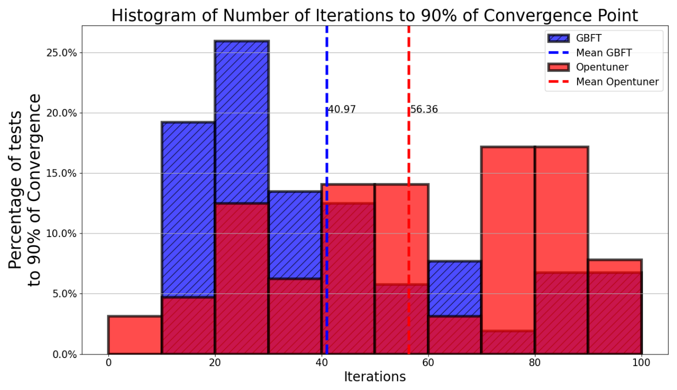

Fig. 3: Histogram of tests reaching 90% of a convergence point for all the GNU core utilities.

code is doing by changing the variable names. It seemed like it was replacing a code or writing something from scratch to do alignment rather than improvement on the given code. The important point was that it could only generate a correct optimized source code for one of the four examined Smith-Waterman versions. The performance improvement of this version was 2.7×, which is significant and shows the capability of these models for further research.

In addition, an investigation of tuning for sort algorithms has been done. The question that was investigated was whether compiler tuning can make a qualitative improvement or not. Fig. 5 shows the performance improvement for different input sizes and transformations in the assembly of the tuned version versus the original version. The tuned version could perform better, up to 5x over the -O3. However, the main changes that have been applied are to loop unrolling rather than a change in the algorithm, and the tuned version is still $O(n^2)$. The other interesting finding is that the tuning cannot improve the performance of the merge sort that has $O(n \log(n))$ time complexity. This is also aligned with the theory.

C. Discussion

The main challenges of this study were improving the speed of convergence in iterative compiler tuning and analyzing its effectiveness. Although iterative approaches can significantly optimize a code, they can not produce a highly optimal answer, and they still need a few iterations to optimize. The first problem is intrinsic, but the second one is an open research

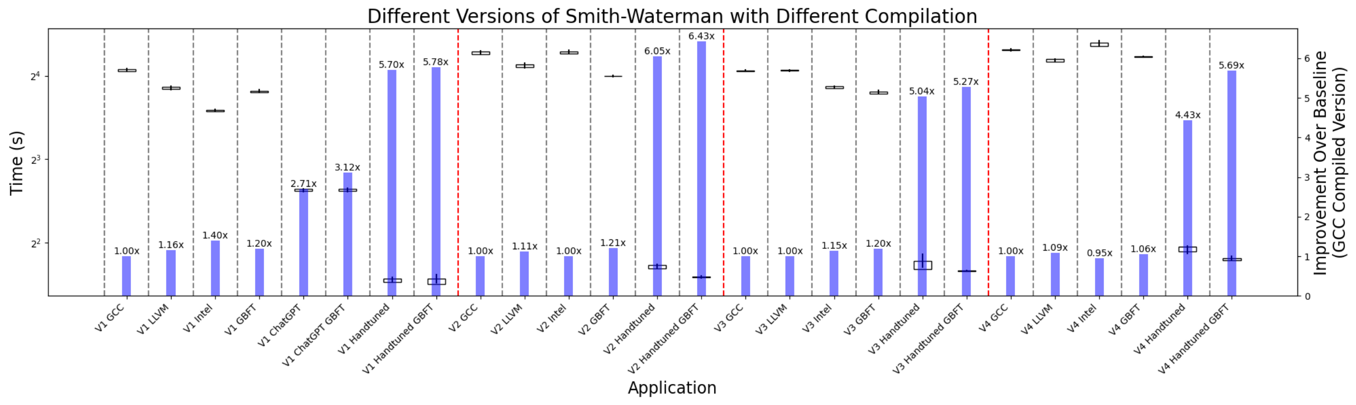

Fig. 4: Comparison of different approaches for optimizing source code for execution time.

direction. The question is whether it is possible to do the tuning even faster.

The proposed algorithm can be improved in two ways. The first is to update the graph more intelligently to find the optimal graph faster. The second is to come up with a non-iterative but effective way of finding the cut-off for the graph. A better but harder way of doing this is to use a fully non-iterative approach via new machine learning models, which is a direction that we are pursuing right now.

The other finding of this study was the effectiveness of LLMs in optimizations. The biggest challenge, though, is how to validate the resultant code. In our small benchmark, we have a chance to compare the outputs and see if they are fully correct or not manually, but it is harder to validate on a large scale automatically.

VIII. Related Work

Optimizing a source code has been done since the start of the first compilers, and much work has been done to improve it. The optimization target can be for code binary size [13], resource utilization [14], real-time performance [15], or a combination of them [16] etc. The challenge with the real-time performance evaluation is its variance, while code size

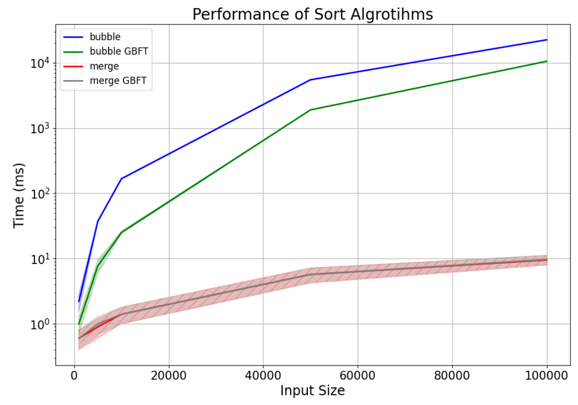

Fig. 5: Effects of compiler tuning for sort algorithms.

optimization doesn't have variations [17]. In this work, [11] is used to measure the run time.

Source code transformation: One way of optimizing a source code for real-time performance is by loop transformation. Polyhedral compilers [18], [19] transform loops to extract possible parallelism. Another way of transforming source code is by annotating it using pragmas and directives [3], [20]. As well as all the classical methods, new LLM methods are introduced for source code optimization, and they showed promising effects while not fully accurate responses [21]. This high rate of false positives for code generation and the non-deterministic behavior of LLMs make it hard to use them as the ultimate solution to code generation for different usages [22].

Compiler Tuning concept: [23] referred to tuning as finding the best tiling size, number of loop unrolling, etc. In this work, tuning refers to optimizing flag passes. Depending on what compiler is used, the optimization task for tuning varies [24]. The problem is either pass selections [25] or pass reordering [26]. This is based on the characteristics of the compiler.

Different Target Compilers: Different compilers require different techniques and heuristics to be tuned. Many works have been done on tuning only one compiler or a combination of them, such as Just-In-Time compilation [27], LLVM [15], [28], [29], GCC [30], [31].

Different Hardware Target: The source code optimization should be applied based on the hardware architecture. In FPGAs, one aim is extracting a systolic array, which requires changing the structure of loops [32] or adopting a combination of different optimization [33], [34]. Other tuning opportunities exist for GPUs [35], [36] or heterogeneous systems [37]. Another usage of compiler tuning is general CPUs, which is the focus of this work. It includes works that were mentioned already.

Different Algorithm for compiler tuning: Countless heuristics and algorithms tune the compiler. There are two major optimization approaches: iterative and non-iterative. A general and effective iterative way of optimizing is through the evolutionary algorithms [10], [38], [39]. In recent years, more machine-learning models have been used for compiler tuning [40]. Reinforcement learning gives more flexibility in self-tuning the compilers and code transformations [41], [42] and is a well-known method for LLVM tuning since it requires phase reordering [43]. However, it is not fast enough for GCC since the problem is pass selection. Adapting the problem to RL can be done using a histogram of passes. This adds redundant complexity to the problem and the large state space can eventually reduce the performance [44].

One goal that matters in compiler tuning in iterative approaches is to reduce the number of iterations, which is addressed in [12]. This motivated this work to minimize the required iterations even further. On the other hand, the non-iterative approaches are those that try to predict a phase order or a pass selection that optimizes a code based on static analysis [31], [45].

Other heuristic ways are introduced to find the best set of optimizations that can optimize a source code and reduce the iteration numbers [46]. This work was inspired by [47] in adopting a graph-based tuning for GCC, from phase reordering problem to pass selection problem.

IX. Conclusion

One important aspect of compiler flags that should be considered is the relationship between flags. This study endorses the idea that flags are not independent, and for an algorithm to be effective, it needs to look into the relations between flags. The proposed graph-based algorithm uses this idea by learning the relations of each two pair of flags. It showed relatively faster convergence to the optimal answer for code size reduction only through the first 100 iterations.

In addition to this finding, we found that compiler tuning is effective in increasing the performance of almost any source code. It is a way that any programmer or tool can use to improve the performance of a source code further. However, it is not able to beat a hand-tuned version. In other words, it cannot change an algorithm's time complexity, but it is better adapted to the hardware.

Acknowledgement

This work was supported, in part, by grant 2024-01-RH01 from Red Hat.

References

[1] A. H. Ali, Q. Huang, W. S. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "Autophase: Compiler phase-ordering for high level synthesis with deep reinforcement learning," *CoRR*, vol. abs/1901.04615, 2019. [Online]. Available: http://arxiv.org/abs/1901.04615

[2] H. Shahzad, A. Sanaullah, S. Arora, R. Munafo, X. Yao, U. Drepper, and M. Herbordt, "Reinforcement Learning Strategies for Compiler Optimization in High Level Synthesis," in *The Eighth Workshop on the LLVM Compiler Infrastructure in HPC*, 2022, dOI: 10.1109/LLVM-HPC56686.2022.00007.

[3] R. Munafo, H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt, "Improved models for policy-agent learning of compiler directives in hls," in *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2023, pp. 1–8.

[4] M. Kovac, M. Brcic, A. Krajna, and D. Krleza, "Towards intelligent compiler optimization," in *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, 2022, pp. 948–953.

[5] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *International Symposium on Code Generation and Optimization (CGO'06)*. IEEE, 2006, pp. 12–pp.

[6] M. Poorhosseini, W. Nebel, and K. Grüttner, "A compiler comparison in the risc-v ecosystem," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2020, pp. 1–6.

[7] E. DHollander, E. Danneels, K.-B. Decorte, S. Loobuyck, A. Vanheule, I. V. Kets, and D. Stroobandt, "Exploring large language models for verilog hardware design generation," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, pp. 111–115.

[8] A. Sanaullah and M. Herbordt, "FPGA HPC using OpenCL: Case Study in 3D FFT," in *9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, 2018, p. 1–6, doi: 10.1145/3241793.3241800.

[9] ——, "An Empirically Guided Optimization Framework for FPGA OpenCL," in *2018 International Conference on Field Programmable Technology (FPT)*, 2018, pp. 46–53, doi: 10.1109/FPT.2018.00018.

[10] R. Sajjadinasab, H. Rastaghi, H. Shahzad, S. Arora, U. Drepper, and M. Herbordt, "Further optimizations and analysis of smith-waterman with vector extensions," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2024, pp. 561–570.

[11] D. Peter, "hyperfine," 2023. [Online]. Available: https://github.com/sharkdp/hyperfine

[12] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 303–316.

[13] A. F. d. Silva, B. N. De Lima, and F. M. Q. Pereira, "Exploring the space of optimization sequences for code-size reduction: insights and tools," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, 2021, pp. 47–58.

[14] A. S. Madhav, S. Singaravel, and A. Karmel, "Memory utilization and machine learning techniques for compiler optimization," in *ITM Web of Conferences*, vol. 37. EDP Sciences, 2021, p. 01021.

[15] S. Jain, Y. Andaluri, S. VenkataKeerthy, and R. Upadrasta, "Poset-rl: Phase ordering for optimizing size and execution time using reinforcement learning," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022, pp. 121–131.

[16] S.-W. Lee, S.-M. Moon, W.-K. Jung, J.-S. Oh, and H.-S. Oh, "Code size and performance optimization for mobile javascript just-in-time compiler," in *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*, 2010, pp. 1–7.

[17] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-time systems*, vol. 1, no. 2, pp. 159–176, 1989.

[18] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, "Polygeist: Raising c to polyhedral mlir," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 45–59.

[19] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 193–205.

[20] H. Shahzad, A. Sanaullah, S. Arora, U. Drepper, and M. Herbordt, "Autoannotate: Reinforcement learning based code annotation for high level synthesis," in *2024 25th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2024, pp. 1–9.

[21] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve *et al.*, "Large language models for compiler optimization," *arXiv preprint arXiv:2309.07062*, 2023.

[22] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini, "Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks," in *IEEE symposium on security and privacy*, 2024.

[23] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[24] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, pp. 1–42, 2018.

[25] U. Garciarena and R. Santana, "Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, 2016, pp. 1159–1166.

[26] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "Micomp: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–28, 2017.

[27] K. Hoste, A. Georges, and L. Eeckhout, "Automated just-in-time compiler tuning," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 62–72.

[28] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "Mlgo: a machine learning guided compiler optimizations framework," *arXiv preprint arXiv:2101.04808*, 2021.

[29] R. Nobre, L. Reis, and J. M. Cardoso, "Impact of compiler phase ordering when targeting gpus," in *European Conference on Parallel Processing*. Springer, 2017, pp. 427–438.

[30] T. Sandran, M. N. B. Zakaria, and A. J. Pal, "A genetic algorithm approach towards compiler flag selection based on compilation and execution duration," in *2012 International Conference on Computer & Information Science (ICCIS)*, vol. 1. IEEE, 2012, pp. 270–274.

[31] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather *et al.*, "Milepost gcc: machine learning based research compiler," in *Proceedings of the GCC Developers' Summit*, 2008.

[32] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 93–104.

[33] H. Ye, H. Jun, H. Jeong, S. Neuendorffer, and D. Chen, "Scalehls: a scalable high-level synthesis framework with multi-level transformations and optimizations," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 1355–1358.

[34] H. Shahzad, A. Sanaullah, S. Arora, R. Munafo, X. Yao, U. Drepper, and M. Herbordt, "Reinforcement learning strategies for compiler optimization in high level synthesis," in *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 2022, pp. 13–22.

[35] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, pp. 1–23, 2021.

[36] C. Yu, S. Royuela, and E. Quiñones, "Openmp to cuda graphs: a compiler-based transformation to enhance the programmability of nvidia devices," in *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, 2020, pp. 42–47.

[37] M. Thavappiragasam and V. Kale, "Cpu-gpu tuning for modern scientific applications using node-level heterogeneity," in *2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2023, pp. 179–183.

[38] J. M. Aragón-Jurado, J. C. de la Torre, P. Ruiz, P. L. Galindo, A. Y. Zomaya, and B. Dorronsoro, "Automatic software tailoring for optimal performance," *IEEE Transactions on Sustainable Computing*, 2023.

[39] B. Tağtekin, B. Höke, M. K. Sezer, and M. U. Öztürk, "Foga: flag optimization with genetic algorithm," in *2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*. IEEE, 2021, pp. 1–6.

[40] P. M. Phothilimthana, A. Sabne, N. Sarda, K. S. Murthy, Y. Zhou, C. Angermueller, M. Burrows, S. Roy, K. Mandke, R. Farahani *et al.*, "A flexible approach to autotuning multi-pass machine learning compilers," in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2021, pp. 1–16.

[41] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "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.

[42] R. Mammadli, A. Jannesari, and F. Wolf, "Static neural compiler optimization via deep reinforcement learning," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2020, pp. 1–11.

[43] S. Jain, Y. Andaluri, S. VenkataKeerthy, and R. Upadrasta, "Poset-rl: Phase ordering for optimizing size and execution time using reinforcement learning," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2022, pp. 121–131.

[44] M. HajiKhodaverdian, H. Rastaghi, M. Saadat, and H. Shah-Mansouri, "Reinforcement learning-based task scheduling using dvfs techniques in mobile devices," in *2023 IEEE 34th Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2023, pp. 1–6.

[45] A. TehraniJamsaz, M. Popov, A. Dutta, E. Saillard, and A. Jannesari, "Learning intermediate representations using graph neural networks for numa and prefetchers optimization," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1206–1216.

[46] J. Thomson, M. O'Boyle, G. Fursin, and B. Franke, "Reducing training time in a one-shot machine learning-based compiler," in *International workshop on languages and compilers for parallel computing*. Springer, 2009, pp. 399–407.

[47] R. Nobre, L. G. Martins, and J. M. Cardoso, "A graph-based iterative compiler pass selection and phase ordering approach," *ACM SIGPLAN Notices*, vol. 51, no. 5, pp. 21–30, 2016.