Towards LLM-enhanced Compiler Optimization

Damian Garber[1,*], Tamim Burgstaller[1], Sebastian Lubos[1], Patrick Ratschiller[1] and Alexander Felfernig[1]

[1]Graz University of Technology, Inffeldgasse 16b, Graz, 8010, Austria

Abstract

Optimization has always been a central focus in computer science. There are various approaches to achieving this, from finding better algorithms to optimizing compiled code. One such approach is compiler optimization, where we can customize the compiler's configuration to optimize for runtime, energy consumption, or binary size, among other factors. However, these optimizations must be carefully selected for each program and typically require expert knowledge. We utilize compiler autotuning to address this, which automatically selects a program's optimization options. Most current solutions for this task require a significant amount of time. Driven by the growing popularity of AI-assisted coding, we have investigated the potential of Large Language Models (LLMs) as a tool for solving the task of compiler autotuning. We show that LLMs can produce well-performing optimization configurations within a reasonable timeframe acceptable for interactive settings.

Keywords

Compiler Autotuning, Optimization, Large Language Models

1. Introduction

Optimizing code can be achieved through various methods, with compiler optimization being one of the most straightforward approaches. Compilers like the GNU Compiler Collection (GCC)[1] offer over 200 optimization options that can be enabled or disabled. The selection of appropriate optimization options often requires an expert-level understanding. In order to allow non-expert users to use compiler optimization, GCC provides sets of recommended default optimizations depending on the optimization goal. For example, the `-Os` flag contains the recommended set of optimization options to minimize the binary size of the compiled executable. In the following, the most important of the default optimization sets is the `-O3` flag, which optimizes the runtime of the compiled executable. However, these default options may lead to suboptimal results [1]. Compiler autotuning solves this problem by selecting optimization options individually for a given program. The state-of-the-art modern compiler autotuning consists primarily of iterative approaches that consume a significant amount of time due to the need for repeated compilations to generate compiler optimizations, making them not scalable for larger projects. This paper investigates the applicability of using Large Language Models (LLMs) for compiler autotuning. To this end, we use ChatGPT-4o[2] to generate optimized GCC commands and compare their performance with state-of-the-art compiler autotuning approaches.

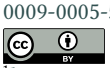
The remainder of this paper is organized as follows. Section 2 discusses related works on compiler autotuning and LLMs. Section 3 outlines the experimental setup, and Section 4 presents the findings. We address potential threats to validity in Section 5 and explore potential extensions of this work in Section 6. Finally, we present our conclusions in Section 7.

ConfWS'25: 27th International Workshop on Configuration, Oct 25–26, 2025, Bologna, Italy

*Corresponding author.

damian.garber@tugraz.at (D. Garber); tamim.burgstaller@ist.tugraz.at (T. Burgstaller); sebastian.lubos@tugraz.at (S. Lubos); patrick.ratschiller@ist.tugraz.at (P. Ratschiller); alexander.felfernig@tugraz.at (A. Felfernig)

0009-0005-0993-0911 (D. Garber); 0009-0007-4522-8497 (T. Burgstaller); 0000-0002-5024-3786 (S. Lubos); 0009-0005-5065-5149 (P. Ratschiller); 0000-0003-0108-3146 (A. Felfernig)



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]https://gcc.gnu.org/

[2]https://openai.com/index/gpt-4o-and-more-tools-to-chatgpt-free/

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-01-15

CEUR-WS.org/Vol-4149/paper5.pdf

2. Related Work

The field of compiler autotuning addresses two key challenges: the phase selection problem and the phase ordering problem, both aimed at optimizing program performance [2]. The phase selection problem identifies which optimizations to apply, while the phase ordering problem determines the sequence of these optimizations. This work focuses solely on phase selection. In the modern state-of-the-art, iterative solutions have become the standard approach [3, 4, 5, 6, 7]. Bodin et al. [8] propose one of the earliest iterative approaches. Their approach starts with an initial set of optimization options activated, compiles the program, evaluates its performance, and refines the configuration in a loop until satisfactory results are achieved. Newer approaches focus primarily on increasing the efficiency of iterative approaches. For example, *COBAYN* [9] uses Bayesian Networks to narrow the search space to the most promising configurations. The current state-of-the-art method, *BOCA* [10], employs Bayesian Optimization to identify key optimizations and streamline the search process. *CompTuner* [11] builds a prediction model for the runtime of different optimization options and uses a particle swarm optimization algorithm [12] to improve the search performance. *Cole* [5] can perform multi-target optimization (for example, runtime and energy consumption) by iteratively creating a Pareto front.

However, performance is the central problem for the computationally intensive iterative state-of-the-art approaches, requiring several compilations, which, with increasing project size, becomes a substantial problem. *Cole*, for example, needs to create a Pareto front, which takes 50 days on a single machine [5]. New lightweight approaches such as *Optimization Space Learning (OSL)* [13] try different strategies to achieve a responsive tool that provides optimization options faster, with the trade-off of lower prediction quality. *OSL* combines configuration space learning and collaborative filtering to achieve this. First, *OSL* generates a set of synthesized optimization configurations using a t-wise feature coverage heuristic and measures their performance for multiple benchmarks. *OSL* then recommends optimization configurations for new programs using collaborative filtering [14].

In this work, we explore the applicability of LLMs in the context of compiler autotuning. LLMs have already been used successfully in similar situations. For example, [15] uses a purpose-trained model to minimize the size of the compiled binary, achieving a 3% improvement over the default optimizations and outperforming several state-of-the-art iterative approaches. Another example is [16], which uses LLMs to generate hardware-optimized code, or [17], which proposes the Meta Large Language Model Compiler based on the CodeLLama model.

3. Experimental Setup

We used the following setup to evaluate the applicability of using LLMs in the context of compiler autotuning. We conducted all experiments on a machine running GCC version 11.4.0 on a Xubuntu-22.04 machine with an Intel i7 processor. No multithreading or multiprocessing was applied. We used the most recent release of OpenAI's ChatGPT-4o to generate the GCC command that would minimize the execution time of the resulting binary. To this end, we used the prompt visualized in Figure 1. We considered prompting techniques other than the zero-shot approach, such as few-shot or chain of thought, but they were ultimately disregarded. The few-shot approach is disregarded due to the lack of a dataset containing code and its optimal compiler optimization settings. At the same time, the chain of thought goes directly against the idea of automatization, without expert input, inherent to the concept of compiler autotuning.

We evaluate our results using the PolyBench[3] benchmarks, commonly employed in compiler autotuning evaluation. We run the prompt shown in Figure 1 for each of the 30 benchmarks, exchanging the "{Code}" with the full content of the respective C file for each benchmark. We use the framework

[3] https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1/tree/master

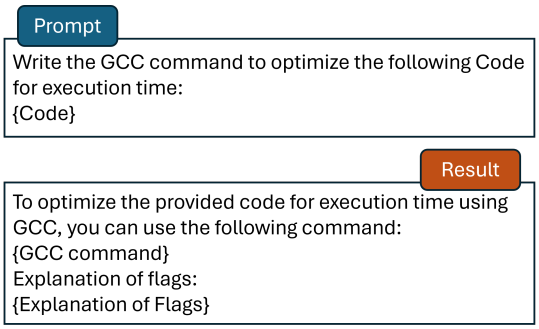


Figure 1: The prompt used to generate GCC commands and its result.

used by *OSL*[4], another compiler autotuning approach, to evaluate the performance of the generated GCC command [13]. The conversion to the *OSL* framework means that some optimization options, for example, hardware architecture-specific optimizations such as `-march=native`, are intentionally discarded. Discarding these options minimizes the influence of system-specific behavior and thus leads to more general results [13]. These results are then compared to the performance of the GCC command using `-O3` for the same program similarly converted to the *OSL* framework. The execution time of the binaries generated by both commands is measured using perf stat[5] and the speedup of the LLM generated GCC command (t_{LLM}) against the `-O3` GCC command (t_{O3}) is calculated using (1).

$$speedup = \frac{t_{O3}}{t_{LLM}} \tag{1}$$

4. Results

First, we investigate the LLM-generated optimization results on its own, and in the second step, we compare the results with other state-of-the-art alternatives.
We measured an average speedup of 1.020 when using the LLM-generated GCC command compared to the default optimization settings of the `-O3` GCC command over the 30 benchmarks tested. The median is marginally higher, with a speedup of 1.021 and a standard deviation of 0.046. We provide a histogram in Figure 2 to further visualize these results.

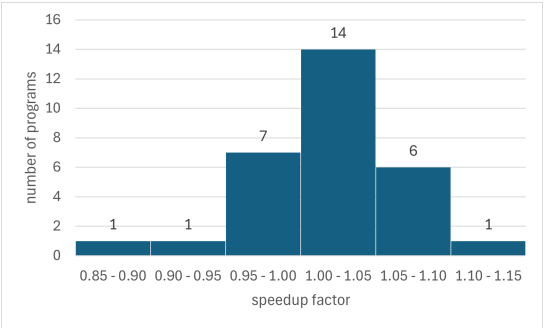


Figure 2: Histogram showing the performance speedups achieved by the LLM optimizations compared to `-O3`.

The time needed to generate the GCC commands is, on average, 8.96s. These results show the potential of an LLM-supported compiler autotuning approach, as it outperforms the default GCC optimization in 21 out of 30 tested benchmarks while needing a reasonable time.

[4]https://github.com/AIG-ist-tugraz/OptimizationSpaceLearning
[5]https://perf.wiki.kernel.org/index.php/Main_Page

We now compare our LLM-based approach with other state-of-the-art compiler autotuning approaches, more precisely eight other approaches, which are shown in Table 1.

Table 1
List of approaches used for comparison and their references

Approach	Reference
OSL	[13]
CompTuner	[11]
BOCA	[10]
TPE	[10]
Random Iterative Compilation (RIO)	[4]
Genetic Algorithms (GA)	[6]
OpenTuner	[18]
COBAYN	[9]

To allow for a direct comparison with the other approaches, we only visualize 10 of the 30 benchmarks provided by Polybench, as was done by [13, 10, 11]. The ten programs selected are listed in Table 2. We adapted Table 2 from a table provided by [13]. We compare the speedup of our results in Table 3 and the time to generate these results in Table 4 with the alternatives. The other results were taken from a table provided by [13] and extended with our results. We discuss the use of external data in Section 5.

We will first discuss the time needed to generate the results shown in Table 4. We can split the results into three categories. *OSL* provides the first and fastest in the single-digit millisecond range. Our approach provides the second fastest results in the single-digit second range. The remaining approaches operate in a range of several thousand seconds. Thus, we can conclude that *OSL* outperforms all other approaches in speed by an order of magnitude. However, while outperformed by *OSL*, our approach is still an order of magnitude faster than the other state-of-the-art approaches. It allows for a reasonably fast response for direct user interaction.

Regarding the speedup of the compiled code, we outperform the state of the art for the programs P4 and P8. We can only compare individual results for most alternatives since they usually calculate overall results using additional programs on top of the benchmark set used here or only use parts of it. *BOCA* [10], for example, calculates its overall performance using only 10 of the 30 programs from PolyBench, in addition to 10 programs from another benchmark, claiming that no significant speedup can be achieved for the remaining 20 programs. In our case, the average speedup increases from 1.020 to 1.026 when using only the 10 programs compared to the entire benchmark. The only directly comparable approach is *OSL*, which reports an average speedup of 0.994 over the entire benchmark. Our results outperform these results significantly, averaging a speedup of 1.020.

5. Threats to Validity

This work represents a proof of concept, exploring the potential use of LLMs in compiler autotuning. We demonstrated that the optimizations generated by LLMs could outperform default optimizations on average.

Several factors could have influenced the results of this work, but they were not within the scope of this study. Firstly, we utilized an externally hosted LLM, which could have affected result generation speed. We anticipate that using a locally hosted model would yield faster results. Secondly, we

Table 2
The list of programs from PolyBench used for the comparison with other approaches

ID	Program	#SLOC	Description
P1	correlation	248	Correlation computation
P2	covariance	218	Covariance computation
P3	symm	231	Symmetric matrix-multiply
P4	2mm	252	2 matrix multiplications
P5	3mm	267	3 matrix multiplications
P6	cholesky	212	Cholesky decomposition
P7	lu	210	LU decomposition
P8	nussinov	569	DP for sequence alignment
P9	heat-3d	211	Heat equation (3D data dom.)
P10	jacobi-2d	200	2-D Jacobi stencil comp.

Table 3
The speedup of the programs in Table 2 compared to -O3 as defined in (1). The best speedup is marked in bold font, while "-" denotes no speedup (...) denotes external data.

Technique	ID	Speedup	ID	Speedup	ID	Speedup	ID	Speedup	ID	Speedup
LLM		1.019		-		-		**1.109**		1.021
OSL		1.000		1.043		-		-		-
CompTuner		**1.077** (...)		**1.080** (...)		1.042 (...)		1.071 (...)		1.041 (...)
BOCA		-		-		**1.075** (...)		1.071 (...)		**1.046** (...)
TPE	P1	-	P2	-	P3	1.046 (...)	P4	1.072 (...)	P5	-
RIO		-		-		1.042 (...)		-		-
GA		-		-		-		-		1.041 (...)
OpenTuner		-		-		-		1.075 (...)		-
COBAYN		-		**1.080** (...)		1.068 (...)		1.079 (...)		-
LLM		1.025		1.046		**1.057**		-		1.050
OSL		1.010		1.016		-		**1.109**		-
CompTuner		1.013 (...)		1.073 (...)		1.029 (...)		1.025 (...)		**1.055** (...)
BOCA		1.014 (...)		-		1.030 (...)		1.028 (...)		**1.055** (...)
TPE	P6	-	P7	-	P8	-	P9	1.027 (...)	P10	-
RIO		1.016 (...)		-		1.029 (...)		-		-
GA		1.013 (...)		-		-		1.025 (...)		-
OpenTuner		-		**1.075** (...)		1.033 (...)		-		-
COBAYN		**1.064** (...)		-		-		1.028 (...)		-

employed ChatGPT-4o, a general model. We expect a model trained explicitly for this purpose to yield superior results.

Furthermore, we only calculated the non-iterative approaches and sourced the results for the iterative approaches externally, recognizing that this may introduce distortions. This step was necessary because only around half of the approaches made their code publicly available, and the calculation of results would have taken several days per program per approach. The distortion is mitigated by comparing the relative speedup of two optimizations tested on the same machine rather than directly comparing the runtime of the selected benchmarks. Although comparing the time to calculate an optimization directly can lead to issues, in our case, the time differences are so significant that we consider any distortions negligible for the comparisons.

6. Future Work

We see future extensions of this work go in three principal directions. The first is increasing the prediction performance of the used LLM by creating a purpose-trained model dedicated to compiler

Table 4
The time needed to generate the GCC command of the programs in Table 2. The best speedup is marked in bold font, while "-" denotes no speedup and is thus disregarded (...) denotes external data.

Technique	ID	Time [s]	ID	Time [s]	ID	Time [s]	ID	Time [s]	ID	Time [s]
LLM		7.35		-		-		**7.78**		**7.96**
OSL		**0.0043**		**0.0039**		-		-		-
CompTuner		3107.00 (...)		4067.00 (...)		2573.00 (...)		3720.00 (...)		2976.00 (...)
BOCA		-		-		1923.00 (...)		3726.00 (...)		3639.00 (...)
TPE	P1	-	P2	-	P3	3775.00 (...)	P4	3112.00 (...)	P5	-
RIO		-		-		4172.00 (...)		-		-
GA		-		-		-		-		3160.00 (...)
OpenTuner		-		-		-		4691.00 (...)		-
COBAYN		-		4727.00 (...)		**1092.00** (...)		3102.00 (...)		-
LLM		11.76		5.64		**10.04**		-		**10.61**
OSL		**0.0039**		**0.0048**		-		**0.0040**		-
CompTuner		4726.00 (...)		5549.00 (...)		3661.00 (...)		2976.00 (...)		2192.00 (...)
BOCA		4971.00 (...)		-		4082.00 (...)		3420.00 (...)		3026.00 (...)
TPE	P6	-	P7	-	P8	-	P9	2637.00 (...)	P10	-
RIO		3018.00 (...)		-		3264.00 (...)		-		-
GA		3862.00 (...)		-		-		3684.00 (...)		-
OpenTuner		-		6792.00 (...)		4970.00 (...)		-		-
COBAYN		3109.00 (...)		-		-		4116.00 (...)		-

optimization. While cost-intensive in data and processing power, we expect such an endeavor to show significantly improved results, allowing a fast solution while still providing high-quality results. However, extending this approach to other models such as Gemini 2.5 Pro[6], Claude 4.0 Opus[7], or Codestral[8] is likely more cost-effective than training a completely new model and is very likely to yield improvements.

Another research direction would be to integrate this with the fast-emerging AI coding tools like GitHub's Copilot[9], JetBrains' AI Assistant[10], or CodeCompanion[11]. These tools are directly embedded into the Integrated Development Environment (IDE) and are already fully aware of the complete code base. Thus, they would be in a perfect environment to predict compiler optimizations. Additionally, this leads to the possible applicability of our approach to more extensive projects, for which most of the state-of-the-art is not suited.

Lastly, this work could be extended by including compiler optimization experts, both for creating datasets and prompts that could be used to enhance the approach directly, or to compare their recommended optimization options with the results produced by this and other compiler autotuning approaches.

7. Conclusion

This paper shows the applicability of using LLMs in compiler autotuning. The compiler optimizations generated using ChatGPT-4o for the GCC compiler improved the tested benchmark's runtime on average by a factor of 1.020 while taking an average of 8.96s to generate the optimizations. We outperform the state-of-the-art approaches in 2 out of 10 benchmarks while performing an order of magnitude faster. These results suggest that this approach is scalable also for large projects, a significant shortcoming of

[6]https://ai.google.dev/gemini-api/docs/models#gemini-2.5-pro
[7]https://www.anthropic.com/news/claude-4
[8]https://mistral.ai/news/codestral
[9]https://github.com/features/copilot
[10]https://www.jetbrains.com/ai/
[11]https://codecompanion.ai/

the existing iterative state-of-the-art approaches.

Acknowledgments

This study was funded by GENRE, Austrian Research Promotion Agency (Grant No. 915086).

Declaration on Generative AI

While preparing this work, the author(s) used ChatGPT-4 (GPT-4-turbo) and Grammarly to check grammar and spelling and improve formulations. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] J. Gong, T. Chen, Deep configuration performance learning: A systematic survey and taxonomy, arXiv preprint arXiv:2403.03322 (2024).

[2] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, C. Silvano, A survey on compiler autotuning using machine learning, ACM Computing Surveys (CSUR) 51 (2018) 1–42.

[3] S. Triantafyllis, M. Vachharajani, N. Vachharajani, D. I. August, Compiler optimization-space exploration, in: International Symposium on Code Generation and Optimization, 2003. CGO 2003., IEEE, 2003, pp. 204–215.

[4] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, C. Wu, Deconstructing iterative optimization, ACM Transactions on Architecture and Code Optimization (TACO) 9 (2012) 1–30.

[5] K. Hoste, L. Eeckhout, Cole: compiler optimization level exploration, in: Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization, 2008, pp. 165–174.

[6] U. Garciarena, 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.

[7] L. Pérez Cáceres, F. Pagnozzi, A. Franzin, T. Stützle, Automatic configuration of gcc using irace, in: Artificial Evolution: 13th International Conference, Évolution Artificielle, EA 2017, Paris, France, October 25–27, 2017, Revised Selected Papers 13, Springer, 2018, pp. 202–216.

[8] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, E. Rohou, Iterative compilation in a non-linear optimisation space, in: Workshop on profile and feedback-directed compilation, 1998.

[9] A. H. Ashouri, G. Mariani, G. Palermo, E. Park, J. Cavazos, C. Silvano, Cobayn: Compiler autotuning framework using bayesian networks, ACM Transactions on Architecture and Code Optimization (TACO) 13 (2016) 1–25.

[10] J. Chen, N. Xu, P. Chen, H. Zhang, Efficient compiler autotuning via bayesian optimization, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 1198–1209.

[11] M. Zhu, D. Hao, J. Chen, Compiler autotuning through multiple phase learning, ACM Trans. Softw. Eng. Methodol. (2024). URL: https://doi.org/10.1145/3640330. doi:10.1145/3640330, just Accepted.

[12] J. Kennedy, R. Eberhart, Particle swarm optimization, in: Proceedings of ICNN'95-international conference on neural networks, volume 4, IEEE, 1995, pp. 1942–1948.

[13] T. Burgstaller, D. Garber, V.-M. Le, A. Felfernig, Optimization space learning: A lightweight, noniterative technique for compiler autotuning, in: Proceedings of the 28th ACM International Systems and Software Product Line Conference, 2024, pp. 36–46.

[14] M. D. Ekstrand, J. T. Riedl, J. A. Konstan, et al., Collaborative filtering recommender systems, Foundations and Trends® in Human–Computer Interaction 4 (2011) 81–173.

[15] 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).

[16] C. Hong, S. Bhatia, A. Haan, S. K. Dong, D. Nikiforov, A. Cheung, Y. S. Shao, Llm-aided compilation for tensor accelerators, arXiv preprint arXiv:2408.03408 (2024).

[17] C. Cummins, V. Seeker, D. Grubisic, B. Roziere, J. Gehring, G. Synnaeve, H. Leather, Meta large language model compiler: Foundation models of compiler optimization, arXiv preprint arXiv:2407.02524 (2024).

[18] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, 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.