Producing Optimal Code Using the Unharnessed Power of Esoteric Language Compilers

Abstract. Few people think of esoteric programming languages - such as Brainfuck or INTERCAL - as practical solutions to real-world problems. Their unconventional nature often means that their compilers make use of design choices and optimizations that could potentially be used to improve mainstream languages. Writing optimal code has always been a burning question for programmers. In this paper, we explore the question of achieving comprehensive optimization across every aspect of code performance without degrading any of them. These compilers are usually overlooked and don't have enough research because they are not taken seriously. The main goal of this research is whether the unusual optimization strategies developed for esoteric language compilers can be formalized and applied to produce optimal code in more practical contexts. In our research we compare these unusual languages with more conventional ones, such as imperative and functional languages to further explore the difference in linguistic elements. After this we compared the conventional compilers with the esoteric ones. The structural possibilities, constraints and paradigms that esoteric languages provide make it possible to further expand optimization trough control flow optimization, compile time evaluation, branch, loop and redundancy elimination, partitioned parallelism, metaprogramming and the like. Following this research we invented a technique that allows compiling truly optimal code. According to the test results, the new technique performed well over the predicted results. These findings contribute to a more optimized way of coding.

1. Introduction

In the birth of programming languages, code optimization has been the focus and cause of headaches for many compiler programmers. With time, accepted structures and paradigms of programming languages arose, and many wanting to come up with compilers have chosen to walk the beaten path, tossing away esoteric languages as strange and absurd, something unable to be useful in the real world. While their denouncement of such languages as strange and absurd may hold some truth, their assumption of uselessness does not. As shown in Listing 1 and Listing 2, even the most absurd and unconventional languages such as Brainfuck and `><>` (fish) are capable of expressing meaningful logic and structure, challenging our assumptions about what counts as a "useful" language.

```
++++++++++[>++++++>+++++++++>++++++++>++++>+++>+<<<<<<-]>+++
+++.>++.+++++++..+++.>>.>-.<<-.<.+++.------.--------.>>>+.>-.
```

Listing 1. Example code of Hello World! written in Brainfuck

```
0voa                                ~/?=0:\
 voa             oooo'Buzz'~<        /
 >1+:aa*1+=?;::5%:{3%:@*?\?/'zziF'oooo/
 ^oa                    n:~~/
```

Listing 2. Example code of FizzBuzz written in `><>`

The focus on code optimization has been losing its relevancy over time. Over the years developers have learnt to rely on hardware optimization in order to focus more on code reusability and ease of understanding [1]. With improving ways of processing, only important code is deemed to be optimized as much as possible. But as the world is emerging into a state of absolute technological dependence, optimization once again became important. To write code that not only fulfills its job, consumes the least amount of resources and provides fast execution time is a challenge many software developers have to face. Most claim that optimization and esoteric languages are contrary to each other, and that programming languages are either industrialized efficiency machines or abstract forms of art [2].

In this paper we claim, that such languages intended for artistic expressions have come up with ideas that programming languages repeating and adhering to the norms are not capable of. To provide an example, Icon has proven to improve not only code quality, but also code efficiency, through the use of special values, avoiding branches and computation-heavy error checking [3]. Another example shows indirect improvement, Whitespace itself wasn't created with much thought about optimization, but it has been demonstrated, that with

optimization in mind, it can produce extraordinary results. As a result we can achieve a compiler with a heavily lowered stack-depth and -manipulation, an excellent way of code partitioning for parallelism.[4]. And as a not-much seen idea, metaprogramming, or even hyper-metaprogramming, is able to evaluate code at compile time that we only dreamed about beforehand [5].

In many esoteric programming languages, optimization ideas are either invented by accident, or they remain hidden until their potential is discovered. In this paper, we explore and analyze these unconventional concepts, introducing a new, better approach to compiler optimization that refines these overlooked techniques.

1.1. Related work

Esoteric programming languages, despite having little to no use in real-world projects, are nonetheless commonly discussed in research papers due to their unique nature. Singer and Draper[6] explore why esoteric programming languages fascinate people and how they can act as a form of artistic expression, while also serving as a useful tool for expanding programmers' knowledge.

Walker and Griswold[3] demonstrate a unique example of compiler optimization through the use of Icon, highlighting the difficulties that one might encounter when attempting to optimize with such narrow margins. The paper also explains the how the use of special values and their handling can help to achieve safe and secure code without computing heavy error checking.

Witmans and Zaytsev[4] take a different approach and illustrate an example of optimization related to esoteric languages by refactoring Whitespace. In their paper, they explore refactoring methods that are uniquely applicable to esoteric languages, and how they can be used to expand the idea of compiler optimization in general.

Sang-Woo [7] takes this one step further, and applies Branfuck's esoteric features into building architectures immensely capable of parallel processing. In his paper, he discovers not only the possibility of impressive performance improvement, but of greatly reduced energy consumption.

On the other hand, Teodorescu and Potolea[5] introduce the idea of hyper-metaprogramming, which is a new concept that expands on metaprogramming, the branch of programming in which programs manipulate other programs. Hyper-metaprogramming takes this even further, where programs are produced that operate on metaprograms. This is highly relevant to compiler optimization, since metaprogramming largely happens at compile-time.

2. Methodology

We checked multiple esoteric languages such as Icon, Whitespace, Brainfuck, Sparrow and Geometry Dash. Our goal was to create a technique that solves the optimal code problem. The question that interested us was if it is possible through the methods constructed by esoteric language compilers, to improve conventional compilers. This includes performance, storage and allocation optimization. We used a modified version of the Sparrow esoteric hyper-meta programming language for research to fit our Poisson-Steve technique and Linux operating systems. The reasons for these are that these tools are easy to use, relevant to our research and they have good performance. For comparing we also checked conventional programming languages, such as Java, C++, Python. We started building our technique with checking state of the art compiler optimization techniques.

Our technique consists of two phases. We introduce the name Poisson-Steve to this technique. An image of the technique can be seen on Figure 1.

Figure 1. The phases of Poisson-Steve technique

The first phase is a pre-processing one, where we heavily optimize an unoptimized code using both widely known and uniquely esoteric techniques. The esoteric optimization consists of three steps. These steps can be seen on Figure 2.

The first one is minimizing branching - as this is a heavy process for the CPU - by leveraging special inbuilt values. This is inspired by Icon compilers since it relies heavily on the use of special values explicitly understood by the compiler. This step also includes approaching error handling the same way, which could be processor heavy based on the amount of tasks it has to check. For this we also took inspiration from Haskell's Maybe type.

The second step is minimizing stack depth and stack manipulation. For this we used various form of unusual optimization techniques, inspired by the possibilities Whitespace can provide. This contributes to more efficient memory usage and faster execution time, particularly in computation-heavy or deeply nested programs.

The third step is maximizing bit-level and computing parallelism. This

was inspired by the possibilities of Whitespace and Brainfuck, as they can be structured for parallelism immensely. The unique structure of these languages provide a perfect ground for code partitioning and bit-level structures usage.

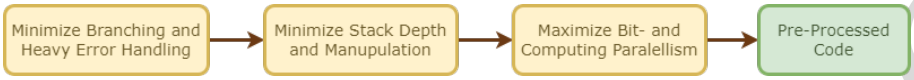

Figure 2. The steps of first phase

The second phase is hyper-metaprogramming, where we use the pre-processed code as an input to produce the optimized code as an output. These steps can be seen on Figure 3.

In our technique, hyper-metaprogramming is used as a form of optimization implemented by the compiler using the pre-processed program. This step creates and expands a separate compile- and run-time environment. The goal is to move as many parts of the program into the compile-time environment as possible.

The compile time environment includes every part of the program that the compiler was able to compute at compile time, while the run time environment includes those parts that the compile-time environment does not. The maximization of the compile-time environment reduces execution cost significantly.

The compiler mainly achieves this trough changing the program to acquire the ability to modify itself when needed. Through this, the program can alter certain parts of itself depending on its state. This results in algorithms and data structures computed in compile-time that could only be computed in run-time without such feature.

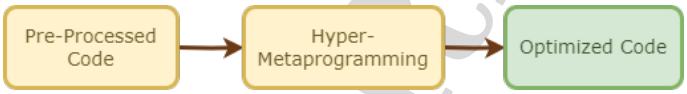

Figure 3. The steps of second phase

3. Results

For evaluating our proposed technique (Poisson-Steve) we conducted tests. The results are presented in three separate tables, each focusing on a key performance metric.

Table 1 summarizes the impact of our technique on runtime performance. Table 2 presents the memory usage improvements, while Table 3 highlights the reduction in peak stack depth.

Each table showcases three programs (Hello World, a 2000-line program, and a 2-million-line program). For each program, we compare the average performance of three compiler techniques: our proposed Poisson-Steve technique, Optimizer 1, and Optimizer 2.

Program	Poisson-Steve	Optimizer 1	Optimizer 2
Hello World	20 ms	20 ms	21 ms
2000 Line Program	483 ms	546 ms	573 ms
2 Million Line Program	3246 ms	4976 ms	5647 ms

Table 1. Runtime comparison across compiler techniques.

Program	Poisson-Steve	Optimizer 1	Optimizer 2
Hello World	0.6 MB	0.6 MB	0.6 MB
2000 Line Program	96.2 MB	102.4 MB	110.2 MB
2 Million Line Program	1978.2 MB	2034.8 MB	2125.4 MB

Table 2. Memory usage comparison across compiler techniques.

Program	Poisson-Steve	Optimizer 1	Optimizer 2
Hello World	4	4	4
2000 Line Program	96	162	174
2 Million Line Program	754	1226	1298

Table 3. Peak stack depth across compiler techniques.

Across all evaluated metrics, the Poisson-Steve technique showed consistent improvements over the other compiler optimizers. While the improvements in memory usage were modest, in runtime performance and stack depth were quite significant. These results indicate a strong possibility for our technique in advancing compiler optimization, particularly in scenarios where resource efficiency is critical.

4. Discussion

In this paper we proposed a new technique to solve the optimal code problem. We compared esoteric language compilers to conventional ones. It provided a deeper understanding for performance and memory usage of compilers. We evaluated this technique with other optimization techniques. In conclusion this shows significant findings in this field of research.

Acknowledgment

We would like to thank the Department of Computer Science at Eötvös Loránd University and our teacher Ábel Szauter for providing resources and support during this research.

References

[1] H. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," Nov. 2010, pp. 421–426. DOI: 10.1145/1882362.1882448.

[2] D. Temkin, "The less humble programmer," *DHQ: Digital Humanities Quarterly / DHQWords*, vol. 17, no. 2, 2023. [Online]. Available: https://rlskoeser.github.io/dhqwords/vol/17/2/000698/.

[3] K. Walker and R. E. Griswold, "An optimizing compiler for the icon programming language," *Software: Practice and Experience*, vol. 22, no. 8, pp. 637–657, 1992. DOI: https://doi.org/10.1002/spe.4380220803. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380220803. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380220803.

[4] R. Witmans and V. Zaytsev, "Perfecting nothingness by refactoring whitespace," English, in *SATToSE'23*, A. {De Lucia}, D. {Di Nucci}, V. Pontillo, and G. Recupito, Eds., ser. CEUR Workshop Proceedings, 15th Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2023,

SATToSE 2023 ; Conference date: 12-06-2023 Through 14-06-2023, Germany: CEUR, Sep. 2023, pp. 19–30.

[5] L. Teodorescu and R. Potolea, "Compiler design for hyper-metaprogramming," in *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2013, pp. 201–208. DOI: 10.1109/SYNASC.2013.34.

[6] J. Singer and S. Draper, "Let's take esoteric programming languages seriously," in *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! '25, ACM, Oct. 2025, 213–226. DOI: 10.1145/3759429.3762632. [Online]. Available: http://dx.doi.org/10.1145/3759429.3762632.

[7] S.-W. Jun, "50,000,000,000 instructions per second: Design and implementation of a 256-core brainfuck computer," in *Proceedings of the ACM SIGBED International Conference on Embedded and Real-Time Computing Systems and Applications (SIGBED '16)*, 2016. [Online]. Available: https://people.csail.mit.edu/wjun/papers/sigtbd16.pdf.

Boróka Anna Aradi
Department of Computer Science
Eötvös Loránd University
Pázmány Péter Sétány 1/C Budapest, Hungary
Budapest
Hungary
w5t3fd@inf.elte.hu

Levente Cantwell
Department of Computer Science
Eötvös Loránd University
Pázmány Péter Sétány 1/C Budapest, Hungary
Budapest
Hungary
ezn9nb@inf.elte.hu

András Szabó
Department of Computer Science
Eötvös Loránd University
Pázmány Péter Sétány 1/C Budapest, Hungary
Budapest
Hungary
gkvz6p@inf.elte.hu

Richárd Krisztián Szigeti
Department of Computer Science
Eötvös Loránd University
Pázmány Péter Sétány 1/C Budapest, Hungary
Budapest
Hungary
qm41ka@inf.elte.hu