Optimizing LLVM Pass Sequences with Shackleton: A Linear Genetic Programming Framework

Hannah Peeler*
hpeeler@utexas.edu
Arm Ltd.
USA

Shuyue Stella Li
sli136@jhu.edu
Department of Computer Science, Johns Hopkins University, USA

Andrew N. Sloss†
andrew@sloss.net
Arm Ltd.
USA

Kenneth N. Reid
ken@kenreid.co.uk
Department of Animal Science, Michigan State University, USA

Yuan Yuan‡
yyuan@msu.edu
Department of CSE, Michigan State University, USA

Wolfgang Banzhaf
banzhafw@msu.edu
Department of CSE, Michigan State University, USA

ABSTRACT

In this paper we explore the novel application of a linear genetic programming framework, Shackleton, to optimizing sequences of LLVM optimization passes. The algorithm underpinning Shackleton is discussed, with an emphasis on the effects of different features unique to the framework when applied to LLVM pass sequences. Combined with analysis of different hyperparameter settings, we report the results on automatically optimizing pass sequences with Shackleton for two software applications at differing complexity levels. Finally, we reflect on the advantages and limitations of our current implementation and lay out a path for further improvements. These improvements aim to surpass hand-crafted solutions with an automatic discovery method for an optimal pass sequence.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; • **Software and its engineering** → *Compilers.*

KEYWORDS

Evolutionary Algorithms, Genetic Programming, Compiler Optimization, Parameter Tuning, Metaheuristics

ACM Reference Format:
Hannah Peeler, Shuyue Stella Li, Andrew N. Sloss, Kenneth N. Reid, Yuan Yuan, and Wolfgang Banzhaf. 2022. Optimizing LLVM Pass Sequences with Shackleton: A Linear Genetic Programming Framework. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3520304.3528945

*At time of publication, is no longer affiliated with Arm.
†Now at University of Washington, Seattle, WA, USA.
‡Now at Beihang University, Beijing, China. Reachable at yyxhdy@gmail.com

GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA

ACM ISBN 978-1-4503-9268-6/22/07.
https://doi.org/10.1145/3520304.3528945

1 INTRODUCTION

In this paper we introduce and utilize Shackleton, a generalized framework that allows for the exploration of applying Linear Genetic Programming (LGP) [5] – a subset of Genetic Programming techniques [4, 10] – to novel use cases with minimal background knowledge. The core of this work assesses the performance of Shackleton on the optimization of a practical and complex use-case: the optimization of LLVM compiler optimization pass sequences. Optimizing compiler optimization pass sequences is a rare target for Evolutionary Algorithms (EAs) despite its pervasive importance to programming as a whole. We provide an analysis of the optimization sequences that Shackleton's evolutionary scheme generates compared to default general solutions.

2 BACKGROUND AND RELATED WORK

The work presented in this paper is a combination of method, implementation, and target application, but is built upon structures created by other researchers and software developers. We utilize a LGPs framework of our own creation, Shackleton, to target LLVM optimization pass sequences. Our framework was inspired by similar efforts to make EAs methods more accessible like Python's DEAP package [8] or PushGP [16].

To understand LGP, it is important to recall that Genetic Programming (GP) is an "Evolutionary Computation (EC) technique that automatically solves problems without requiring the user to know or specify the form or structure of the solution in advance" [15]. LGP further specifies that programs are represented as linear sequences of instructions, leveraging the fact that computers often represent and run programs in a linear fashion. Shackleton leverages this quality to target optimizing compiler pass sequences.

A compiler translates computer code written in one source language to a target language in an executable form. An *optimizing* compiler tries to minimize or maximize attributes of the computer program output in the target language to achieve some benefit at execution time [1]. Recent research aims to improve on a priori optimization methods by targeting machine learning based compilation [20]. GP and GAs as methods are no exception to this. A genetic algorithm approach has been applied to GNU Compiler Collection (GCC), a compiler similar to LLVM but differentiated by its fixed number of supported languages and reuse constraints [3, 9]. Further,

GP was already applied to a compiler context to take advantage of GP's optimization potential for well-known compiler heuristics [17]. Here we specifically target the LLVM Project - a collection of modular and reusable compiler and tool-chain technologies [12] - and evaluate Shackleton's effectiveness at optimizing the source- and target-independent segment of LLVM's core framework.

3 METHODS

3.1 Problem Space

While Shackleton is designed to be applied to various use cases, here we focus on the LLVM use case as mentioned above. The framework takes in the source code of a target program, and outputs a near-optimal optimization pass sequence that the LLVM compiler can use to optimize this program. In our experiments, Ant Colony Optimization for the Traveling Salesman Problem (ACOTSP) [18] and the Backtrack Algorithm for the Subset Sum Problem (SSP) [13] are explored as the target programs. SSP is used for all the results shown in the following sections.

The Ant Colony Optimization (ACO) algorithm is inspired by foraging behavior of some ant species [7], in which paths with more visits would be reinforced after each iteration. In the traveling salesman problem, a graph of cities and the distance between each pair of cities are given, and the goal is to find a path in the topology that minimizes total distance travelled.

In the SSP, we are given a set of N numbers from 1 to 1,000,000, and we are asked to find a subset that sums up to a random number X (where $1,000,000 < X < N \times 1,000,000$) [2, 6, 11, 13]. Although it might run into halting states, the backtracking algorithm approximates the solution in a reasonable time. Both ACOTSP and SSP are NP-complete problems, as they do not have polynomial-time solutions. Therefore, optimizing the algorithms is non-trivial and worth inspection here.

3.2 The Shackleton Framework

Shackleton is a generic GP framework that aims to make GP easier for a myriad of uses. Currently, the main target of Shackleton is to use the framework for optimization of LLVM pass sequences that ultimately optimize executable code. The source code for Shackleton is publicly available on Arm's GitHub page.

3.2.1 Genetic Programming Design in Shackleton. In the Shackleton Framework of LGP for LLVM, see FIG 1, each chromosome consists of a sequence of optimization passes. First, a population of random individuals is generated. For each sequence, its fitness is calculated by compiling the program with that sequence of passes and averaging the program runtime over 40 runs. In each generation, a portion of the individuals with the best fitness scores are chosen as the elite group and copied over to the next generation unchanged. Then, in the special offspring selection process, a tournament-style selection approach is used to select parent individuals. Genetic operators including one-point crossover and one-point substitution mutation are applied to their genomes to produce a 'brood' of 4 offspring individuals; in each brood, the two best performing individuals out of the four offspring and the two parents are selected to join the next generation. This special offspring selection mechanism is inspired by Tackett's brood selection [19]. Finally, some new

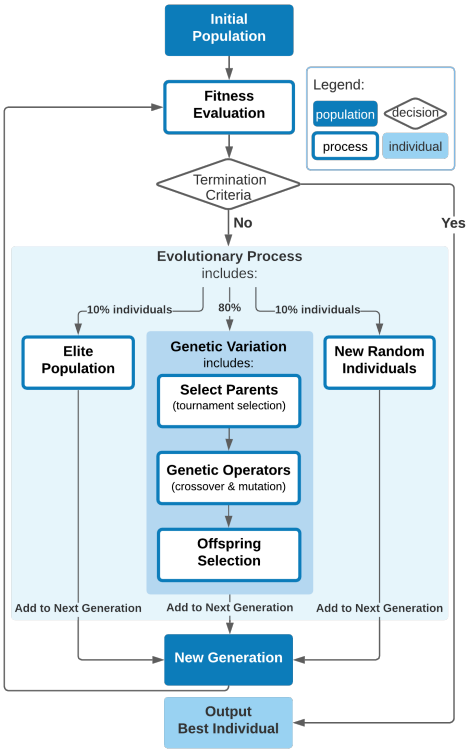

Figure 1: Shackleton LGP for LLVM

individuals are randomly generated to be included in the next generation in order to preserve diversity in the population. The fitness of the new generation is then evaluated and evolution continues. At termination, the program outputs individuals with the best fitness scores as the final solution. For the experiments in this paper, the termination criterion is when the number of generations reaches the set limit, but other implementations (e.g., converging fitness values) are also available and can be further explored. Shackleton's evolution hyperparameters (i.e., number of generations, population size, and tournament size) are set before program execution. Tuning of these hyperparameters is further explored in [14].

4 EXPERIMENTS AND RESULTS

We conduct two primary experiments in this paper. We first test out extreme combinations of the number of generations and number of individuals to determine the effect of these parameters. We then explore the robustness of the framework's optimization benefits. The experiments were conducted on HPCC nodes running CentOS Linux version 7 and Clang version 8.0.0. We utilized the Backtrack Algorithm (from SSP), which contains recursive loops that are potential sites of optimization. Preliminary testing found this problem to be of adequate complexity for our experimentation purposes, considering both the runtime and human-verifiable testability. Expanded results and additional experiments can be found in [14].

Table 1: Number of Generations vs. Population Size Experiment Parameters

Experiment Nr.	Number of Generations	Population Size
0	50	40
1	250	8
2	200	10
3	10	200
4	8	250
5	4	500

The number of generations and population size directly determine the number of runtime evaluations. These two hyperparameters are roughly proportional to how many individuals will appear throughout the evolutionary process. Therefore, in order to reduce the overhead runtime of Shackleton, it is essential to determine how much these two factors influence the search process of the optimal solution. In our first experiment, we tracked the fitness of the best individual in each generation with extreme settings of number of generations and population size. This is to see whether more computing power and runtime should be allocated to a large number of generations or a larger population size, when their product is fixed. The values are set such that each parameter setting has a constant product, so the same number of total individuals are represented in the entire run of Shackleton.

Table 1 shows the combinations of parameters used for this experiment. Six trials were run for each setting. FIG 2 plots the control set up with 50 generations and a population size of 40; FIG 3 and FIG 4 plot two representative cases of a setting with a large *num_generation* value and a setting with a large *population_size* value. The y-axis is the runtime of the sample use case - the SSP with the Backtrack Algorithm - in seconds, which we use as the fitness of the individuals, and the x-axis is the generation number. The runtime of the program with no optimization and with default LLVM optimization passes (first 8 data points of each trend line) is plotted with the runtime of the individuals across generations. FIG 5 shows the percentage improvement of the parameter settings compared to each of the baseline optimization levels.

By exploring the extreme ends of the search space, this experiment shows that more computing power and runtime should be allocated to the population size rather than the number of generations. In the hyperparameter setting plotted in FIG 3, a small population size of 10 is allowed to undergo 200 generations in search for the optimal solution, but the fitness of the best individual is noisy and does not tend to converge, with a final percent improvement of 4.08%. On the other hand, a large population size of 200 shows converging fitness within just 10 generations and a final percent improvement of 5.25%. This result gives support to the claim that the greater diversity in a large population is crucial in the search for an optimal solution.

Even though the hyperparameter settings for Shackleton are problem-specific and can be set by the user, it is important to make sure that most (if not all) hyperparameter combinations will speed up the execution of the target source code. We tested a variety of combinations to examine the robustness of Shackleton, the details

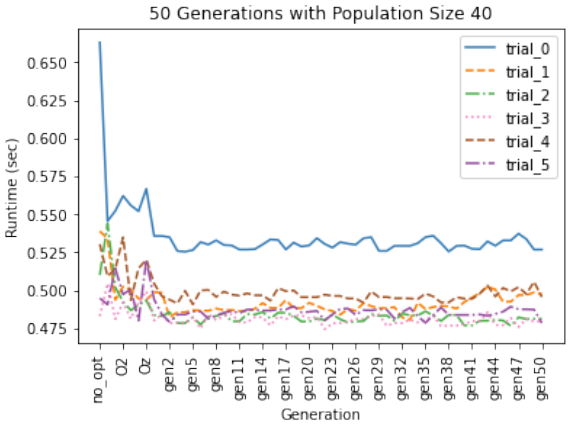

Figure 2: SSP Runtime - gen=50, pop=40. Fitness decreases then converges to a stable level.

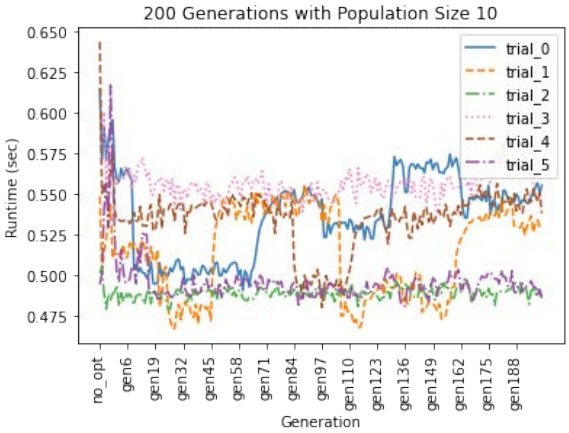

Figure 3: SSP Runtime - gen=200, pop=10. Fitness fluctuates significantly and does not converge.

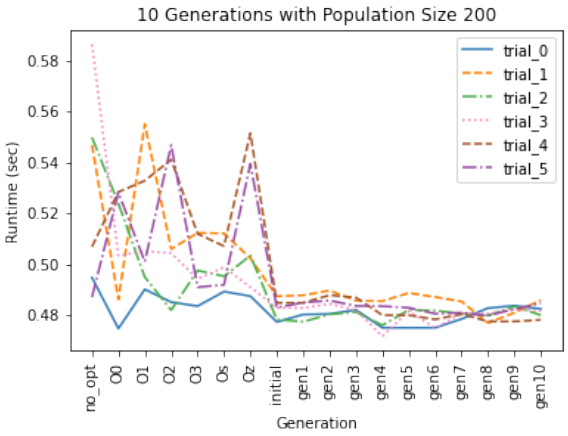

Figure 4: SSP Runtime - gen=10, pop=200. Fitness converges very quickly but longer overhead runtime (not shown in plot).

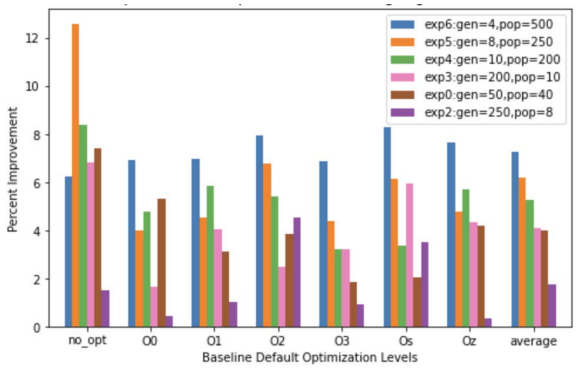

Figure 5: SSP Runtime - Improvement over Baseline. Percent improvement of top 6 hyperparameter settings when plotted against each baseline level. As shown in plot, all combinations result in reduced runtime compared to the default LLVM optimization levels.

Table 2: Percent Improvement

Comparison	Average % Improvement	Additional %
O2-raw	1.40	74.09
Shackleton-raw	6.20	93.88
Shackleton-O2	4.77	90.17

of which can be found in [14]. Two metrics are of importance in this evaluation - percent optimization from the raw source code and the percent improvement of the automatically-generated sequence compared to the default optimization. We first want to ensure that Shackleton will produce optimization sequences that speed up the execution of the target source code. Then, we want the automatically-generated sequences to perform better than the default LLVM optimization. 94% of these runs produced faster code compared to the unoptimized source code.

Shackleton's use of stochastic elements in its evolutionary scheme means that speed improvements over the raw source code after optimization are not always guaranteed. However, the experiments shown here indicate that Shackleton consistently produces optimized code that is faster and offers a larger average improvement than that produced by the default LLVM optimization sequences.

5 CONCLUSION AND FUTURE WORK

The existing predefined LLVM optimization levels of the -Ox type are used for general purpose program optimization and are not problem specific. Targeted runtime optimization of a program by a hand-crafted LLVM pass or a pass sequence would require expert knowledge in both LLVM and the program to be optimized. In this paper, we presented the Shackleton Framework, which is able to automatically generate near-optimal LLVM optimization pass sequences for specific programs. The Shackleton Framework, without any prior knowledge of the compiler, the optimization passes, or the program, consistently produces optimization pass sequences that achieve significant runtime improvement over default optimization options.

There are numerous opportunities to further improve and analyze the Shackleton Framework. Widening the scope of LLVM passes considered (including custom passes) and allowing for a comparison against sequences fit for a specific use case could offer insights beyond what is shown here. In addition to benefits for LLVM, the generic Shackleton Framework could be improved by allowing for hyperparameter tuning beyond what is currently supported. With further improvements, the pass sequences automatically generated by the Shackleton Framework could potentially surpass hand-crafted pass sequences while simultaneously freeing developers' time to pursue other advancements for compilers.

ACKNOWLEDGMENTS

This work was generously funded by the EnSURE (Engineering Summer Undergraduate Research Experience) program at Michigan State University as well as the John R. Koza Endowment. We also gratefully acknowledge The Institute of Cyber-Enabled Research (ICER) at MSU for providing the hardware infrastructure that made the computation required to complete this work possible.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, Boston, MA.
[2] Michael Alekhnovich, Allan Borodin, Joshua Buresh-Oppenheim, Russell Impagliazzo, Avner Magen, and Toniann Pitassi. 2011. Toward a model for backtracking and dynamic programming. *Computational Complexity* 20, 4 (2011), 679–740.
[3] Prathibha A. Ballal, H. Sarojadevi, and Harsha P S. 2015. Compiler Optimization: A Genetic Algorithm Approach. *International Journal of Computer Applications* 112, 10 (2015), 9–13.
[4] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank Francone. 1998. *Genetic Programming – An Introduction.* Morgan Kaufmann Publishers, San Francisco, CA.
[5] Markus F. Brameier and Wolfgang Banzhaf. 2007. *Linear Genetic Programming.* Springer, New York, NY.
[6] Pinar Civicioglu. 2013. Backtracking search optimization algorithm for numerical optimization problems. *Appl. Math. Comput.* 219, 15 (2013), 8121–8144.
[7] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. 2006. Ant colony optimization. *IEEE Computational Intelligence Magazine* 1, 4 (2006), 28–39.
[8] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (2012), 2171–2175.
[9] William von Hagen. 2006. *The Definitive Guide to GCC, Second Edition.* Apress Media, Springer, New York, NY.
[10] John R. Koza. 1992. *Genetic Programming.* MIT Press, Cambridge, MA.
[11] Jeffrey C Lagarias and Andrew M Odlyzko. 1985. Solving low-density subset sum problems. *J. ACM* 32, 1 (1985), 229–246.
[12] llvm-admin team. 2021. The LLVM Compiler Infrastructure. https://llvm.org.
[13] Parth Shirish Nandedkar. 2019. SubsetSum-BacktrackAlgorithm. https://github.com/parthnan/SubsetSum-BacktrackAlgorithm.
[14] Hannah Peeler, Shuyue Stella Li, Andrew N. Sloss, Kenneth N. Reid, Yuan Yuan, and Wolfgang Banzhaf. 2022. Optimizing LLVM Pass Sequences with Shackleton: A Linear Genetic Programming Framework. *arXiv* 2201.13305 (2022). https://arxiv.org/abs/2201.13305
[15] Riccardo Poli, William Langdon, and Nicholas Mcphee. 2008. *A Field Guide to Genetic Programming.* Lulu Enterprises, UK Ltd, Egham, UK.
[16] Lee Spector. 2010, last accessed 2022. Evolutionary Computing with Push. http://faculty.hampshire.edu/lspector/push.html
[17] Mark Stephenson, Una-May O'Reilly, Martin C. Martin, and Saman Amarasinghe. 2003. Genetic Programming Applied to Compiler Heuristic Optimization. In *Genetic Programming*, Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa (Eds.). Springer, Berlin, Heidelberg, 238–253.
[18] Thomas Stützle. 2002. ACOTSP: A software package of various ant colony optimization algorithms applied to the symmetric traveling salesman problem. *URL http://www. aco-metaheuristic. org/aco-code* (2002).
[19] Walter A. Tackett and A. Carmi. 1994. The unique implications of brood selection for genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, Vol. 1. IEEE, Orlando, Florida, 160–165.
[20] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.