A Three-Stage Genetic Algorithm for Compiler Flag and Library Version Selection to Minimize Execution Time

Chi Ho Chan*† and Spyro Nita†
* *Edinburgh Napier University*
chanchihoresearch@gmail.com
† *EPCC, The University of Edinburgh*
s.nita@epcc.ed.ac.uk

***Abstract*—Existing research in compiler autotuning mainly focuses on selecting optimization flags without configurable values. However, the potential of selecting optimization flags with configurable values, alongside using directory and link flags for library version selection to improve performance, remains largely unexplored. We propose a three-stage Genetic Algorithm (GA) that incrementally selects optimization flags without configurable values, then optimization flags with configurable values, and finally library versions, to minimize software execution time. We also discuss the implementation challenges of the proposed algorithm and outline potential future work.**

***Index Terms*—genetic algorithm, compiler optimization, compiler flag selection, library version selection**

I. Introduction

Manual flag selection is complex and time-consuming due to the large number of potential flag combinations. Although optimization levels, which are preset combinations of compiler flags, such as `-O1`, `-O2`, and `-O3` exist, they may not yield the best performance for all applications. This creates the need for automated flag selection methods for specific applications.

Existing automated methods for selecting optimization flags without configurable values mainly use Genetic Algorithm (GA) and Machine Learning (ML). In the GA-based methods, a binary genotype is employed, where each bit represents whether an optimization flag without configurable values is enabled [1], [2]. According to [1], the use of generated optimization flags without configurable values resulted in lower execution times compared to optimization levels across three test programs, each evaluated with varying problem sizes. In ML-based methods, approaches such as probabilistic and transductive models [3], as well as Bayesian Optimization [4], have been applied to predict optimization flags without configurable values. While some ML-based approaches [5], [6] predict unroll factors for individual loops instead of using an optimization flag to enable loop unrolling with a factor for all loops, the selection of broader optimization flags with configurable values remains overlooked.

Selecting the appropriate library versions has been shown to improve software performance, as newer library versions do not always yield the best results [7]. Nonetheless, directory flags, which can be used to specify directories for searching header files of libraries, and link flags, which instruct the compiler on how to link object files of libraries into an executable, are also overlooked despite their potential to select library versions for improving software performance.

We propose a three-stage GA that incrementally explores non-configurable optimization flags, configurable optimization flags, and library versions, beyond using only optimization levels or non-configurable optimization flags. A single algorithm is needed to optimize both optimization flags and library versions because they often work together to impact performance. Since some optimization flags may only provide benefits when used with specific library versions, separately selected library versions may not work well with separately selected optimization flags. Splitting the selection process into three stages reduces the likelihood of build failures and the vast search space. The staged approach also helps identify whether compilation or execution errors are related to specific flags or library version incompatibilities. While this staged approach may miss the global optimum, it may still uncover some high-performing configurations at a lower computational cost.

II. Proposed Algorithm

Before executing the GA, we conduct two preprocessing tasks: optimization flag collection and library compilation.

Optimization flags are collected from the compiler user manual or by using the compiler `help` command. After collecting all the flags, we iteratively compile and run the test program with non-configurable flags, filtering out those causing errors until no errors remain. The final collection consists of the filtered flags without configurable values, combined with all the flags with configurable values. Yet, enabling some of these flags may still cause compilation or execution errors due to incompatibility issues.

For each library version, a directory is created and named using the library's name and version number, separated by a dash, with dependencies separated by a specific number of underscores corresponding to the dependency level. For example, `netcdf-c-4.9.2_hdf5-1.14.1__zlib-1.3.1` represents `netcdf-c-4.9.2` compiled with `hdf5-1.14.1`, which is further compiled with `zlib-1.3.1`. This directory contains subdirectories such as `lib`, `include`, and `bin` for library version selection. All libraries are compiled and linked statically to ensure the correct versions are used.

After preprocessing, the GA begins by accepting files that specify the available optimization flags and the paths for each library version as input. The GA then proceeds through three stages. Each stage follows the generational GA workflow. The key difference in each stage lies in the genotype, as shown in Fig. 1, and the phenotype used during fitness evaluation.

The phenotype is the command used to run the configuration script, which specifies the selected optimization flags along with directory and link flags that indicate paths to library versions. The fitness function runs the phenotype, compiles the test program, and executes it multiple times to ensure performance stability. If any compilation or execution errors occur, it returns a zero fitness value. Otherwise, it returns the inverse of the average execution time across all runs.

In the first stage, each optimization flag without configurable values is encoded as one bit in the genotype to indicate whether the flag is enabled. The phenotype specifies the selected optimization flags along with directory and link flags that indicate paths to fixed, compatible library versions.

In the second stage, each optimization flag with configurable values is encoded with multiple bits in the genotype: one bit indicates whether the flag is enabled, and additional bits represent the value(s) from a specified range. The value range is specified using mathematical notation enclosed in square brackets. If a flag has multiple configurable values, each value range is separated by a colon. To handle optimization flags with integer configurable values ranging from zero to infinity, infinity is replaced with a finite positive number. To handle GCC flags such as `-falign-functions`, `-falign-functions=n`, and `-falign-functions=n:m` that have the same name but require no, one, or multiple configurable value(s), we encode all of them in the genotype. When more than one of these flags is enabled in the genotype, only the one with the greatest number of configurable values is specified in the phenotype. Decoded values exceeding the allowed range are wrapped around. For example, if a flag allows values ranging from 0 to 9 and the decoded value is 10, it wraps to 0; 11 wraps to 1, and so on. The phenotype specifies the most effective optimization flags without configurable values, directory, and link flags from the first stage, and the selected optimization flags with configurable values.

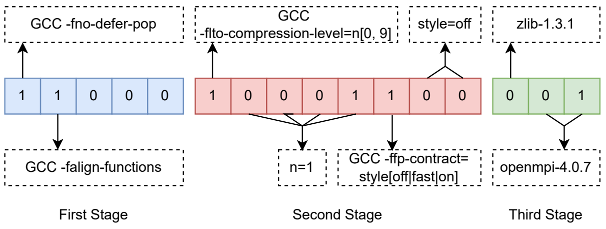

Fig. 1. The genotype for each stage of the GA: the genotype in the first stage encodes optimization flags without configurable values, the genotype in the second stage encodes optimization flags with configurable values, and the genotype in the third stage encodes library versions.

In the third stage, each library version is encoded in the genotype with bits allocated based on the number of available versions. The phenotype specifies the most effective optimization flags from previous stages, along with the directory and link flags that specify paths to the selected library versions.

III. Challenges and Future Work

One challenge in implementing the GA is determining a finite upper bound for optimization flags that theoretically accept infinity, which requires flag-specific knowledge. Another challenge is ensuring flag reliability. Initial exploration showed that even with identical optimization flags, compilation sometimes succeeded and sometimes failed. Despite repeated compilation and execution with error penalties in the fitness function, the non-deterministic behavior of some flags hinders consistent compilation and execution success across trials.

We aim to conduct empirical studies across diverse software applications to test if the flags generated by the GA can outperform optimization levels and optimization flags without configurable values generated by existing methods. We will explore the use of a variable-length genotype to dynamically omit optimization flags or library versions, or narrow the value range of optimization flags with configurable values if they lead to compilation or execution errors or performance declines. By applying a variable-length genotype, the GA is expected to achieve faster convergence and a shorter overall search time. Alternative optimization objectives beyond execution time, such as energy consumption and code size, will also be investigated. Additionally, the potential impact of selecting factors such as command-line interface (CLI) options, preprocessing directives, and environment variables on software performance will be examined.

References

[1] 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.

[2] T. S. Kumar, S. Sakthivel, and S. Kumar, "Optimizing code by selecting compiler flags using parallel genetic algorithm on multicore cpus," *Int. J. Eng. Technol.(IJET)*, vol. 6, pp. 544–555, 2014.

[3] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, pp. 296–327, 2011.

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

[5] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *International symposium on code generation and optimization*. IEEE, 2005, pp. 123–134.

[6] H. Leather, E. Bonilla, and M. O'boyle, "Automatic feature generation for machine learning–based optimising compilation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 1, pp. 1–32, 2014.

[7] S. Raemaekers, A. Van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE, 2012, pp. 378–387.