UDC 004.93

USE OF GENETIC ALGORITHMS IN ADAPTIVE COMPILERS FOR CROSS-PLATFORM OPTIMIZATION

Berdnyk M. G. – Dr. Sc., Associate Professor, Professor of the Department of Computer Systems Software, National Technical University "Dnipro Polytechnic", Dnipro, Ukraine.

Starodubskyi I. P. – Post-graduate student, Department of Computer Systems Software, National Technical University "Dnipro Polytechnic", Dnipro, Ukraine.

ABSTRACT

Context. Modern software is developed under conditions of continuously increasing complexity of computing systems. Today, developers must consider a vast diversity of platforms, ranging from resource-constrained mobile devices to servers with high-performance processors and specialized architectures such as GPUs, FPGAs and even quantum computers. This heterogeneity requires software to operate efficiently across various hardware platforms. However, portability remains one of the most challenging tasks, especially when high performance is required.

One of the promising directions to address this problem is the use of adaptive compilers that can automatically optimize code for different architectures. This approach allows developers to focus on the functional part of the software, minimizing the effort spent on optimization and configuration. Genetic algorithms (GAs) play a special role among the methods used to create adaptive compilers. These are powerful evolutionary techniques that allow finding optimal solutions in complex and multidimensional parameter spaces.

Objective. The objective of this research is to apply genetic algorithms in the process of adaptive compilation to enable automatic optimization of software across different hardware platforms.

Method. The approach is based on genetic algorithms to automate the compilation process. The key stages include: population initialization – creation of an initial set of compilation parameters; fitness function evaluation – assessment of the efficiency of each parameter combination; evolutionary operations – applying crossover, mutation and selection to generate the next generation of parameters; termination criteria – stopping the iterative process upon reaching an optimal result or stabilization of metrics.

Results. The developed algorithm was implemented in Python using the numpy, multiprocessing and subprocess libraries. Performance evaluation of the algorithm was carried out using execution time, energy consumption and memory usage metrics.

Conclusions. The scientific novelty of the study lies in the development of: an innovative approach to automatic compilation parameter optimization based on genetic algorithms; a method for dynamic selection of optimization strategies based on performance metrics for different architectures; integration of GAs with modern compilers such as LLVM for automatic analysis of intermediate representation and code optimization; and methods for applying adaptive compilers to solve cross-platform optimization problems. The practical significance is determined by the use of genetic algorithms in adaptive compilers, which significantly improves the efficiency of the compilation process by automating the selection of optimal parameters for various architectures. The proposed approach can be successfully applied in the fields of mobile computing, cloud technologies and high-performance systems.

KEYWORDS: genetic algorithms, adaptive compilers, cross-platform optimization, automation.

ABBREVIATIONS

GA – Genetic Algorithm;

IR – Intermediate Representation (a form of program representation used during compilation for analysis and optimization);

LLVM – Low Level Virtual Machine (a toolkit for compiler development that provides IR for program analysis and optimization, as well as machine code generation for various architectures);

Clang – a high-performance compiler for C, C++ and Objective-C that uses LLVM as a backend for code optimization and generation;

LLVM Pass Manager – a mechanism in LLVM for managing code optimizations and transformations, allowing step-by-step analysis and modification of IR;

SPEC CPU2017 – a standardized benchmark suite for evaluating processor and compiler performance, including real-world workloads from domains such as computational math, artificial intelligence and physics;

429.mcf – a benchmark from the SPEC CPU2017 suite that simulates a network flow problem, used to test system performance under heavy computational loads;

456.hmmer – a benchmark from the SPEC CPU2017 suite used to evaluate performance in analyzing biological sequences using Hidden Markov Model (HMM) algorithms.

NOMENCLATURE

C is a set of compilation parameters;

C_i is a i-th chromosome (a vector of compilation parameters) in the current population;

C_{best} is a the best solution that has the maximum value of the fitness function;

C_{new} is a new chromosome obtained after crossover or mutation operations;

E is a set of graph edges representing dependencies between instructions in the form of data or control flow relationships;

$F^{(t)}$ is a vector of fitness function values for the population at generation t;

G_{IR} is a directed graph of intermediate representation (Intermediate Representation, IR) used for analyzing code structure during compilation;

DOI 10.15588/1607-3274-2025-4-16

$inst_i$ is an i-th instruction in the intermediate representation;

N is a normal distribution with zero mean and variance σ^2;

N_{elite} is a number of elite chromosomes preserved for the next generation;

P is a population of chromosomes used in the genetic algorithm; each chromosome represents a possible configuration of compilation parameters;

P_{next} is a next generation population;

$P^{(0)}$ is an initial (zero-th) population;

$P^{(t)}$ is a population at iteration t;

$P^{(t)}_{crossed}$ is a set of chromosomes obtained through crossover at generation t;

$P^{(t)}_{elite}$ is an elite subset of the population at generation t;

$P_{mutated}$ is a subset of the population obtained after mutation;

$P^{(t)}_{mutated}$ is a set of chromosomes obtained through mutation at generation t;

p_i is a value of a compilation parameter that reflects whether a specific compilation optimization is applied;

$p_{new,i}$ is a new i-th chromosome after applying genetic operators;

V is a set of graph vertices, each representing a separate instruction of the intermediate representation;

w_i is a i-th weight coefficients that determine the priority of the corresponding metrics;

δ is a delta value sampled from a normal distribution with zero mean and variance σ^2;

σ is a standard deviation used in the mutation process to sample random variations.

INTRODUCTION

Modern adaptive compilers aim to solve the problem of automatic software optimization for various computing platforms. This is achieved through algorithms capable of analyzing the characteristics of target platforms, adapting compilation parameters and ensuring maximum performance. One of the most promising approaches is the use of genetic algorithms (GA), which provide powerful tools for searching optimal solutions in complex multidimensional parameter spaces.

Genetic algorithms are based on the principles of biological evolution. Their core idea lies in using processes similar to natural selection, mutation and crossover to improve solutions to complex problems. In the context of compilation, each possible solution, represented as a "chromosome", contains a set of parameters that define the optimization strategy. These parameters include elements such as loop unrolling, instruction selection, register allocation and memory management.

The population of solutions goes through iterations called "generations". At each step, the algorithm evaluates the effectiveness of the solutions based on a fitness function calculated using given performance metrics. The most successful solutions are preserved and used to create new, improved generations. This process continues until an optimal solution is found or until predefined termination criteria, such as a set number of iterations, are met.

To apply genetic algorithms in adaptive compilers, it is necessary to integrate them with existing tools for code analysis and generation.

The object of this research is the process of integrating genetic algorithms into adaptive compilation for various hardware platforms.

The subject of this research is the models and methods for developing and selecting genetic algorithms to build adaptive compilers.

The purpose of this research is to develop a model for using genetic algorithms in the process of adaptive compilation to ensure automatic optimization of software for different platforms.

1 PROBLEM STATEMENT

To apply genetic algorithms in adaptive compilers, it is necessary to integrate them with the existing LLVM code analysis and generation tool, which provides powerful capabilities for working with intermediate representation (IR). Genetic algorithms should utilize metrics collected during the IR analysis stage, such as the number of instructions, loop depth, register usage, memory access frequency, vectorization level and the size of the IR to evaluate the efficiency of each solution.

For the formalization of the problem, we assume that a set of compilation parameters $P = \{p_1, p_2, ..., p_k\}$ is given, where each parameter p_i takes a value of 0 or 1, corresponding to its disabled or enabled state. The compilation parameters may include: -O0, -O1, -O2, -O3, -Ofast, -Og, -funroll-loops, -fno-unroll-loops, -ftree-vectorize, -fno-tree-vectorize, -fomit-frame-pointer, -fno-omit-frame-pointer, -march=native, -march=armv8-a, -mtune=cortex-a73, -mtune=generic, -flto, -fno-lto, and others.

Additionally, a set of possible compilation configurations C is given, where each configuration $C \subseteq P$ is a feature vector describing the selected optimization parameters.

The performance metrics for compilation with parameters C include the program execution time $T(C)$, energy consumption $E(C)$ and memory usage $M(C)$.

It is required to find such a configuration of compilation parameters C^* that minimizes the objective function:

$$f(C) = w_1 \cdot T(C) + w_2 \cdot E(C) + w_3 \cdot M(C). \qquad (1)$$

DOI 10.15588/1607-3274-2025-4-16

The integration process should include the following stages:

1. Initial population generation. Initial compilation parameters are formed based on standard strategies or generated randomly. These may include loop unrolling, optimization flags and other parameters.

2. Fitness evaluation. Each combination of compilation parameters is tested on the target platform, and performance data is collected. This includes program execution time, energy consumption or memory usage.

3. Application of GA operators. The mutation operator randomly alters compilation parameters, adding diversity to the population. The crossover operator combines parameters from two successful solutions to produce a potentially more effective one.

4. Selection of the best solutions. Solutions with the highest fitness scores are retained for the next generation.

It is necessary to design application scenarios such as:

1. Optimization for mobile devices. Genetic algorithms can select parameters that minimize energy consumption, such as reducing input-output operations or minimizing memory access [1].

2. Server applications. For servers, GAs may focus on optimizing instructions for parallel execution, which is critical for high-performance computing.

Heterogeneous systems. In systems that use both CPUs and GPUs, GAs can find a balance between tasks executed on each architecture to minimize latency and improve overall performance.

2 REVIEW OF THE LITERATURE

Genetic algorithms (GAs) have long been used to search for optimal compiler parameters due to their ability to efficiently explore large solution spaces [2]. Modern compilers contain dozens or even hundreds of options and optimization phases, and a fixed optimization sequence (e.g., standard levels -O1 to -O3) is not universally optimal for all programs and platforms. One basic solution to this issue is iterative compilation, where the code is compiled repeatedly with different combinations of options, and the results are evaluated to select the best configuration. In this context, GAs have proven to be an effective global search method: evolutionary operators such as selection, crossover and mutation gradually improve generations of solutions and discover combinations of flags or compiler pass sequences that yield better program performance.

For example, a 2022 study proposed a parallel GA for selecting optimization sequences in LLVM [3]. The authors clustered programs into subsets and ran three GAs in parallel (with solution sharing between them), allowing exploration of different regions of the solution space. As a result, this approach outperformed the standard -O2 level by an average of 87% in execution speed and exceeded the performance of a sequential (single-threaded) GA by 72–75% [3].

Another 2021 study focused on autotuning GCC compiler flags using a GA. The proposed framework FOGA (Flag Optimization with Genetic Algorithm) implemented dynamic tuning of the GA's own hyperparameters and a stopping criterion when no improvements were observed, resulting in a significant speedup of C++ program execution compared to the then-leading OpenTuner toolkit [4].

These studies confirm that evolutionary algorithms effectively solve both the problem of selecting compiler parameters and the phase-ordering problem in cases where exhaustive search is practically infeasible.

Adaptive compilation approaches can generally be divided into two major categories: search-based methods (such as GAs and other metaheuristics) and learning-based methods (machine learning). Iterative search methods (including GAs, evolutionary strategies, simulated annealing and others) perform black-box optimization by directly measuring the performance of the generated code [5]. Their advantage lies in discovering high-quality solutions without requiring an a priori model, but their main drawback is high computational cost – a compiler may need to run hundreds or thousands of times for a single program. This makes such approaches difficult to apply in real-world conditions, especially when just-in-time (JIT) optimization is needed.

An alternative is to train a model capable of predicting optimization decisions. In this case, the cost is shifted to the training stage (offline), after which the model quickly produces optimization sets for new programs. In recent years [6], there has been growing interest in integrating machine learning into compilers instead of relying on fixed heuristics. For example, in the MLGO (Machine Learning Guided Optimization) system, Google researchers replaced the heuristic function inlining algorithm in LLVM with a model trained using reinforcement learning [2].

This approach reduced the size of the executable code by up to 7% compared to the standard -Oz mode in LLVM, and the same model worked successfully across various program sets and hardware platforms [2].

The combination of machine learning (ML) and evolutionary algorithms opens new possibilities for code optimization. Evolutionary algorithms, including GAs, are good at exploring extreme solutions but can be complemented by ML models to accelerate convergence [6].

One of the emerging directions is predictive compilation optimization, where an ML model is trained on a dataset of programs and corresponding optimization settings to predict which optimizations will be most effective for a new program. By applying a k-nearest neighbors (KNN) algorithm along with a filtering stage to remove non-beneficial transformations, the system automatically selects an individualized sequence of passes for each program instead of using the standard -O2/-O3 levels [7].

Experimental results show that this method outperforms LLVM's fixed -O2 level, achieving an average of ~21% speedup, and with filtering included, the improvement reaches up to 23% [7].

DOI 10.15588/1607-3274-2025-4-16

Despite success in specific scenarios, deploying these approaches in production compilers remains challenging. First, the computational cost of iterative methods is still high. Second, machine learning models risk losing effectiveness on programs significantly different from the training set – the generalization problem remains an active area of research.

Another challenge is integrating these methods into existing compilation pipelines. Compatibility with existing optimization passes and programming languages must be ensured, along with the predictability of results.

Future prospects for adaptive compilers are linked to the continued growth of hardware capabilities and advances in artificial intelligence algorithms. It is expected that compilers will be able to dynamically adapt to the execution environment – for example, performing on-the-fly recompilation based on actual program profiling data.

Recent studies clearly demonstrate the advantages of adaptive compilation, particularly involving genetic algorithms and machine learning methods, for cross-platform optimization. The main challenges today include improving the efficiency and speed of such approaches, ensuring their reliability under various conditions and integrating them into real-world development tools.

3 MATERIALS AND METHODS

Adaptive compilers that utilize genetic algorithms (GAs) are complex modular software systems that provide automatic code optimization for various computing platforms. The architecture of such a compiler includes several interrelated components that sequentially perform analysis, optimization and code generation, enabling dynamic adjustment of compilation parameters based on target platform data.

The architecture of a GA-based compiler consists of the following core modules [8]:

1. Input code analysis module: performs syntactic and semantic analysis, transforming the source code into intermediate representation (IR);
2. Performance metrics collection module: generates numerical data such as instruction count, loops, register usage and memory footprint to evaluate the efficiency of different compilation strategies;
3. Genetic optimizer: responsible for evolutionary improvement of compilation parameters, including mutation, crossover and selection operators;
4. LLVM Pass Manager control module: applies the generated optimization parameters to the IR to enhance performance;
5. Machine code generator: creates executable code adapted to the target platform.

At the first stage, the source code is compiled into IR using LLVM tools such as Clang. IR is a low-level abstraction of the program that includes detailed information about instructions and control flow. Formally, IR can be represented as a dependency graph where vertices V correspond to instructions and edges E represent dependencies between them [3]:

$$G_{IR} = (V, E), V = \{inst_1, inst_2, ..., inst_n\}, E \subseteq V \times V.$$

At this stage, basic performance indicators are also collected. Total number of instructions, number of loops in the program, register usage.

Each chromosome represents a vector of compilation parameters:

$$C = \{p_1, p_2, ..., p_k\}, p_i \in \{0,1\},$$

where $p_i = 1$ indicates the activation of a specific optimization pass (such as loop unrolling, function inlining, or instruction combination), and $p_i = 0$ indicates its deactivation.

To evaluate the effectiveness of each chromosome, a fitness function $F(C)$ is used, which is calculated based on performance metrics:

$$F(C) = w_1 \cdot T(C)^{-1} + w_2 \cdot E(C)^{-1} + w_3 \cdot M(C)^{-1}.$$

During the training phase, $F(C)$ is optimized using the collected performance data. The mutation operator is defined as follows, where a random value δ is sampled from a normal distribution with zero mean and variance σ^2:

$$p_i^{'} = p_i + \delta, \delta \sim N(0, \sigma^2).$$

Let us define the crossover for two chromosomes C_1 and C_2 as follows:

$$C_{new} = \alpha \cdot C_1 + (1 - \alpha) \cdot C_2, \alpha \in \{0,1\}.$$

For selection, an elitist approach is used, in which the best chromosomes of the current generation are preserved for the next one:

$$P_{next} = Elite(P) \cup Crossover(P) \cup Mutation(P),$$

where, $Elite(P)$ is the set of the best (elite) individuals from the current population P, which have the highest fitness and are passed unchanged to the next generation, preserving the quality of already found solutions and preventing the loss of top-performing chromosomes; $Crossover(P)$ is the set of new individuals obtained as a result of crossover between pairs of parent chromosomes from the population P, allowing the combination of successful traits from different solutions and the exploration of new areas in the parameter space; $Mutation(P)$ is the set of individuals created through mutation of existing chromosomes from the population

DOI 10.15588/1607-3274-2025-4-16

P, introducing random variations that provide genetic diversity and help avoid local maxima.

At the first stage, the initial population $P^{(0)}$ is generated, consisting of N chromosomes. Each chromosome is represented as a parameter vector C, which encodes a possible solution to the optimization problem.

The population can be initialized randomly:

$$P^{(0)} = \{C_1, C_2, ..., C_N\}, C_j = \{p_{j1}, p_{j2}, ..., p_{jk}\},$$
$$p_{ji} \in \{0,1\}.$$

Each chromosome is evaluated using the fitness function $F(C)$, which reflects the quality of the solution. The task is to minimize a certain objective function $F(C)$, and the fitness function can be defined as [3]:

$$F(C) = \frac{1}{1 + f(C)}.$$

$f(C)$ is calculated using a standard benchmark suite such as SPEC CPU2017 [9].

For maximization problems, the fitness function can be written as:

$$F(C) = f(C).$$

The objective function for compilation optimization is as follows: (1).

For all chromosomes in the current population $P^{(t)}$, a vector of fitness function values is computed:

$$F^{(t)} = \{F(C_1), F(C_2), ..., F(C_N)\}.$$

Based on the fitness function values, chromosomes are selected to participate in the next generation. One of the most popular methods is tournament selection. K random chromosomes are chosen from the current population, and the winner of the tournament is the chromosome with the highest fitness value.:

$$C_{best} = \arg\max F(C)_{C \in K}.$$

Thus, an intermediate population $P^{(t)}_{selected}$ is formed. For each pair of chromosomes C_1 and C_2 from $P^{(t)}_{selected}$, a crossover operation is performed. Single-point crossover is defined as follows:

$$C_{new} = \{p_{11}, p_{12}, ..., p_{1m}, p_{2(m+1)}, ..., p_{2k}\},$$

where the crossover point m is chosen randomly, $m \in [1, k-1]$. This can also be written as:

$$p_{new,i} = \begin{cases} p_{1,i}, i \le m, \\ p_{2,i}, i > m. \end{cases}$$

The mutation operation modifies the values of individual genes in a chromosome with a certain probability $P_{mutated}$.

The new generation $P^{(t+1)}$ is formed by adding elite chromosomes (elitism):

$$P^{(t+1)}_{elite} = \arg\max F(C)_{C \in P^{(t)}}, | P^{(t+1)}_{elite} | = N_{elite}.$$

Crossover and mutation for the remaining chromosomes are performed as follows:

$$P^{(t+1)} = P^{(t+1)}_{elite} \cup P^{(t+1)}_{crossed} \cup P^{t+1}_{mutated}.$$

The iterative process terminates when one of the following criteria is met:

1. The maximum number of generations has been reached.

2. The fitness function value does not change over a stable period of time.

3. The target fitness value has been achieved.

The program represents an implementation of a system for automatic optimization of compilation parameters using a genetic algorithm, developed in the Python programming language. The system consists of four interconnected modules: genetic_algorithm.py, compiler_wrapper.py, fitness_evaluator.py and main.py, each performing a distinct functional role within the overall architecture. Python 3 was chosen as the primary development language due to its expressiveness and extensive standard library.

The following libraries are used across the modules: numpy, which is employed for generating the initial population, performing array operations, and implementing tournament selection; random, used for the stochastic components of the algorithm, including mutation and crossover; logging, which serves to systematically log the execution process and diagnostic messages; concurrent.futures.ThreadPoolExecutor, which enables multithreaded evaluation of population fitness in order to improve performance; subprocess, which provides an interface for invoking the external gcc compiler and executing compiled binaries; and time, which is used to measure the execution time of program components and the target executables.

The following compiler input parameters were used to implement the compilation optimization:

-O0, -O1, -O2, -O3, -Ofast, -Og – compiler optimization levels;

-funroll-loops – loop unrolling to reduce the number of iterations and improve performance;

DOI 10.15588/1607-3274-2025-4-16

-fno-unroll-loops – disables loop unrolling;

-ftree-vectorize – automatic loop vectorization for using processor SIMD instructions;

-fno-tree-vectorize – disables automatic vectorization;

-fomit-frame-pointer – removes the use of the frame pointer to improve performance, but may complicate debugging;

-fno-omit-frame-pointer – retains the frame pointer, which is useful for debugging;

-march=native – applies optimizations for the current processor architecture;

-march=armv8-a – compiles for the ARMv8-A architecture;

-mtune=cortex-a73 – code optimization for Cortex-A73 processors;

-mtune=generic – general optimizations without targeting a specific architecture;

-flto – Link-Time Optimization (applies optimizations during the linking stage);

-fno-lto – disables link-time optimizations;

-funsafe-math-optimizations – allows potentially unsafe math optimizations to improve performance (e.g., changing the order of floating-point operations);

-fstrict-aliasing – enables aggressive memory access optimizations based on assumptions about pointer aliasing (can improve performance but may cause unpredictable behavior);

-fno-strict-aliasing – disables these optimizations;

-fipa-pta – improved pointer analysis for whole-program optimization;

-fno-ipa-pta – disables this analysis;

-fexpensive-optimizations – enables complex and resource-intensive optimizations;

-fno-expensive-optimizations – disables such optimizations.

This expansion of the parameter space allows exploring a wide range of combinations, including architecture-specific optimizations (-march, -mtune), as well as parameters that affect memory management, loops and computations. Thus, the genetic algorithm minimizes the objective function (1).

The population initialization is performed as follows: a population of 100 chromosomes is created. Each chromosome encodes a combination of compilation parameters as a binary vector {0,1}, where 1 indicates that a parameter is enabled and 0 means it is disabled.

Selection is performed using the tournament method. Crossover is defined as two-point, with a probability of P_{cross} =0.8. The probability of gene mutation is P_{mut} =0.05. Termination criteria are defined as reaching 100 generations and no improvements for 10 consecutive generations.

The use of an extended parameter space makes it possible to account for the unique characteristics of the platform and application scenario, making the compilation process adaptive and highly efficient.

4 EXPERIMENTS

The experiment consists in validating the proposed algorithm for improving the efficiency of the compilation process by automating the selection of optimal parameters for different architectures.

To assess the effectiveness of the algorithm, the following metrics were used: execution time, energy consumption, and memory usage. Specialized hardware counters (Performance Monitoring Unit, PMU) were used to collect these metrics.

Testing was conducted on three distinct hardware platforms covering a wide range of usage scenarios:

– Mobile processor ARM Cortex-A73. Specifications: quad-core processor with a clock frequency of 2.2 GHz, manufactured using a 10 nm process technology. Environment: Android 11 with built-in energy sensors for measuring power consumption. The SPEC CPU2017 benchmarks (429.mcf, 456.hmmer) and mobile workload simulations were used for testing.

– Server-grade processor Intel Xeon Platinum. Specifications: 24-core processor with a clock frequency of 2.3 GHz, supporting 48 threads and a TDP of 165 W. Environment: Linux (kernel 5.15), GCC version 11.2 configured for multi-core optimization.

– Graphics processor NVIDIA RTX 3080 GPU. Specifications: Ampere architecture, 8,704 CUDA cores, 10 GB of GDDR6X memory. Environment: CUDA 11.5 with Python API used for running large-scale data processing tests.

The optimal compilation parameter configurations for the C program were identified for each of the three tested hardware platforms as follows:

– ARM Cortex-A73: -funroll-loops -ftree-vectorize -march=armv8-a -mtune=cortex-a73 -flto -fstrict-aliasing.

– Intel Xeon Platinum: -O3 -funroll-loops -ftree-vectorize -march=native -mtune=generic -flto -fstrict-aliasing -fipa-pta -fexpensive-optimizations.

– NVIDIA RTX 3080 GPU: -Ofast -funroll-loops -ftree-vectorize -flto -fomit-frame-pointer -funsafe-math-optimizations.

As a result of the optimization, the following improvements in three key metrics were achieved for each hardware platform:

– ARM Cortex-A73: execution time reduced by 35%, energy consumption by 50%, and memory usage by 25%;

– Intel Xeon Platinum: execution time reduced by 30%, energy consumption by 35%, and memory usage by 35%;

– NVIDIA RTX 3080: execution time reduced by 25%, energy consumption by 35%, and memory usage by 24%.

Figure 1 presents a diagram of the average performance across the three evaluated hardware platforms when applying the genetic algorithm.

DOI 10.15588/1607-3274-2025-4-16

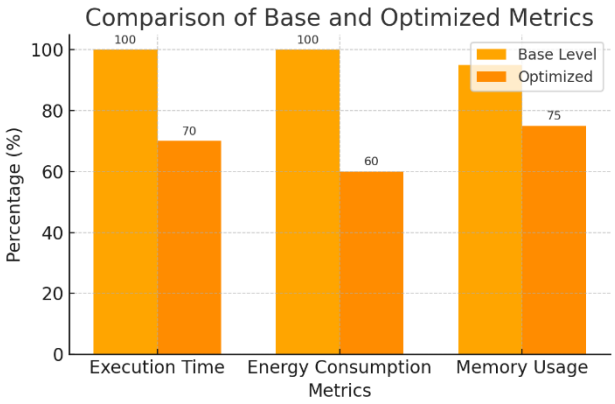

Figure 1 – Performance diagram when using the GA

5 RESULTS

In this work, alongside the theoretical presentation of applying a genetic algorithm to determine optimal compiler parameters, some practical testing results of the developed software were also partially presented for three different hardware platforms.

Testing demonstrated (Figure 1):

– a reduction in execution time by up to 30% on average across platforms, with the most significant improvement observed on the GPU due to parallelization;

– an average energy consumption decrease of 40% across platforms, especially noticeable on the ARM Cortex-A73 due to memory access optimization;

– an average memory usage improvement of 25%, achieved by eliminating redundant operations with memory and registers.

6 DISCUSSION

Genetic algorithms are particularly well suited for tasks related to adaptive compilation for several reasons:

1. They can efficiently explore large solution spaces, even when those spaces contain many local minima. This is especially important in compilation, where various parameters can interact in complex ways, affecting program performance.

2. GAs can be tuned to solve different optimization problems, including minimizing execution time, energy consumption or memory usage. In mobile devices, energy efficiency is the top priority, whereas in server systems, maximum performance takes precedence.

3. GAs can dynamically adjust compilation parameters based on an analysis of the target platform's characteristics. For programs running on GPUs [10], the algorithms can identify optimal loop unrolling and thread distribution parameters. For ARM processors, GAs may propose strategies that minimize energy consumption by reducing memory operations.

Despite their numerous advantages, using genetic algorithms in compilers also presents some limitations. First, the computational complexity of GAs can be significant, especially when the parameter space is large. This can slow down the compilation process, which is particularly critical in CI/CD pipelines. Second, the quality of the discovered solution heavily depends on the fitness function. If the performance metrics do not fully reflect the application's needs, GAs may lead to suboptimal outcomes.

CONCLUSIONS

he use of genetic algorithms in adaptive compilers significantly improves the efficiency of the compilation process by automating the selection of optimal parameters for different architectures. The study results demonstrated:

– a reduction in execution time by up to 30% on average across platforms, with the greatest improvement observed on GPU due to parallelism;

– a reduction in energy consumption by 40% on average across platforms, especially evident on ARM Cortex-A73 thanks to memory access optimization;

– energy consumption decreased by 18% on average for mobile platforms; – manual optimization was replaced with automatic tuning, reducing development time.

The scientific novelty lies in the development of a new approach to adaptive compilation based on the use of genetic algorithms to optimize compilation parameters depending on the architectural features of the hardware platform. Within the scope of this work, the following were proposed:

– an innovative method for dynamic tuning of compilation parameters using genetic algorithms that enables automatic adaptation to heterogeneous computing environments;

– a method for selecting optimization pass sequences for compilers based on evolutionary algorithms, which improves the performance of the generated code compared to static optimization levels (-O2, -O3);

– a mechanism for integrating genetic algorithms with modern compilers (LLVM, GCC) to analyze and adapt intermediate representation (IR) during compilation;

– a multi-objective optimization method for balancing trade-offs between performance, energy consumption and memory usage;

– an automated process for discovering optimal compilation configurations, reducing the need for manual tuning and accelerating software development.

The practical significance lies in implementing genetic algorithms in compiler technologies to increase the efficiency of cross-platform programming. Key aspects of practical application include:

– automation of the compilation optimization process, eliminating the need to manually select parameters for different architectures and reducing code tuning time;

– increased software performance, reducing code execution time by 30% compared to traditional optimization levels (-O2, -O3);

– 25% memory optimization achieved by eliminating redundant memory and register operations;

– reduced energy consumption of computing systems, which is critical for mobile devices and energy-efficient servers (up to 40% energy savings through memory access optimization);

DOI 10.15588/1607-3274-2025-4-16

– optimized use of hardware resources for both CPU and GPU, which is crucial in high-performance computing (HPC) and cloud technologies;

– flexibility and adaptability of the approach, making it suitable for a wide range of environments from embedded systems to large data centers.

Future research in this field may focus on:

– expanding algorithmic optimization methods: using hybrid approaches that combine genetic algorithms with machine learning to predict optimal compilation parameters;

– optimizing runtime performance: integrating approaches for Just-In-Time (JIT) compilation to adapt code execution to runtime environment characteristics;

– modeling adaptive strategies for heterogeneous systems: exploring effective mechanisms for distributing computing resources across CPU, GPU and FPGA;

– application in continuous integration and deployment (CI/CD): developing optimization methods that can be embedded into modern software development pipelines for automatic adaptation of code to new platforms;

– reducing the computational complexity of the optimization process: applying heuristic approaches to decrease the number of algorithm iterations without sacrificing optimization quality.

ACKNOWLEDGEMENTS

The research was conducted at the Department of Software Engineering of the National Technical University "Dnipro Polytechnic" as part of the state-funded scientific project: "Methods, Models and Technologies for Data Processing and Analysis in Computerized Systems" (State Registration No. 0124U004583, Project No. E-346).

REFERENCES

1. Tagtekin B., Höke B., Sezer M. K., Öztürk M. U.FOGA: Flag Optimization with Genetic Algorithm, *Proc. IEEE Int. Conf. on Innovations in Intelligent Systems and Applications (INISTA 2021), 29 June–1 July 2021*. Kocaeli, Turkey, IEEE, 2021, pp. 1–6. DOI: 10.1109/INISTA52462.2021.9456364.
2. Júnior N. L. Q. Rodriguez L. G. A., Silva A. F. da Combining Machine Learning with a Genetic Algorithm to Find Good Compiler Optimizations Sequences, *Proc. 19th Int. Conf. on Enterprise Information Systems (ICEIS 2017),* Vol. 1. Porto, Portugal, 26–29 Apr. 2017, pp. 397–404. DOI: 10.5220/0006270403970404.
3. Peeler H., Li S., Sloss A., Reid K., Yuan Y., Banzhaf W. Optimizing LLVM Pass Sequences with Shackleton: A Linear Genetic Programming Framework, *Companion Proc. Genetic and Evolutionary Computation Conf. (GECCO '22), 9–13 July 2022*. Boston, USA, ACM, 2022, pp. 578–581. DOI: 10.1145/3520304.3528945.
4. Ashouri A. H., Killian W., Cavazos J., Palermo G., Silvano C. A Survey on Compiler Autotuning Using Machine Learning, *ACM Computing Surveys*, 2018, Vol. 51, № 5, Art. 96, 42 p. DOI: 10.1145/3197978.
5. Rosario V. M. do, Borin E., Nacul A., Breternitz M., Collange S., Barrett E. Smart Selection of Optimizations in Dynamic Compilers, *Concurrency and Computation: Practice and Experience*, 2021, Vol. 33, № 10, e6089. DOI: 10.1002/cpe.6089.
6. Cummins C., Fisches Z. V., Ben-Nun T., Hoefler T., O'Boyle M. F. P., Leather H. ProGraML: A Graph-Based Program Representation for Data Flow Analysis and Compiler Optimizations, *Proc. 38th Int. Conf. on Machine Learning (ICML 2021), 18–24 July 2021, Virtual Event*. PMLR, 2021, Vol. 139, pp. 2244–2253.
7. Nugteren C., Codreanu V. CLTune: A Generic Auto-Tuner for OpenCL Kernels, *Proc. IEEE 9th Int. Symp. on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2015), 23–25 Sept. 2015*. Turin, Italy, IEEE, 2015, pp. 195–202. DOI: 10.1109/MCSoC.2015.33.
8. Ashouri A. H., Palermo G., Cavazos J., Silvano C. Automatic Tuning of Compilers Using Machine Learning. Cham, Springer, 2018, 78 p. (PoliMI SpringerBriefs in Applied Sciences). DOI: 10.1007/978-3-319-71489-9.
9. Martins L. G. A., Nobre R., Cardoso J. M. P., Delbem A. C. B., Marques E. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences, *ACM Transactions on Architecture and Code Optimization (TACO),* 2016, Vol. 13, № 1, Art. 14. DOI: 10.1145/2883614.
10. Liou J.-Y., Wang X., Forrest S., Wu C.-J. GEVO: GPU Code Optimization Using Evolutionary Computation, *ACM Trans. on Architecture and Code Optimization,* 2020, Vol. 17, № 4, Art. 33, 28 p. DOI: 10.1145/3418055.

Received 21.04.2025.
Accepted 09.09.2025.

DOI 10.15588/1607-3274-2025-4-16

УДК 004.93

ВИКОРИСТАННЯ ГЕНЕТИЧНИХ АЛГОРИТМІВ В АДАПТИВНИХ КОМПІЛЯТОРАХ ДЛЯ КРОСПЛАТФОРМЕНОЇ ОПТИМІЗАЦІЇ

Бердник М. Г. – д-р техн. наук, доцент, професор кафедри програмного забезпечення комп'ютерних систем, Національний технічний університет «Дніпровська політехніка», Дніпро, Україна.

Стародубський І. П. – аспірант кафедри програмного забезпечення комп'ютерних систем, Національний технічний університет «Дніпровська політехніка», Дніпро, Україна.

АНОТАЦІЯ

Актуальність. Сучасне програмне забезпечення розробляється в умовах постійно зростаючої складності обчислювальних систем. Сьогодні розробникам доводиться враховувати величезну різноманітність платформ: від мобільних пристроїв із обмеженими ресурсами до серверів із високопродуктивними процесорами та спеціалізованих архітектур, таких як GPU, FPGA та навіть квантові комп'ютери. Ця гетерогенність вимагає від програмного забезпечення здатності ефективно працювати на різних апаратних платформах. Однак переносимість програм залишається однією з найскладніших задач, особливо коли йдеться про досягнення високої продуктивності.

Одним із перспективних напрямів вирішення цієї проблеми є використання адаптивних компіляторів, які здатні автоматично оптимізувати програмний код для різних архітектур. Такий підхід дозволяє розробникам зосередитися на функціональній частині програм, мінімізуючи витрати на оптимізацію та налаштування. Особливе місце серед підходів до створення адаптивних компіляторів займають генетичні алгоритми (ГА). Це потужні еволюційні методи, які дозволяють знаходити оптимальні рішення у складних і багатовимірних параметрів.

Мета роботи. Метою дослідження є застосування генетичних алгоритмів у процесі адаптивної компіляції для забезпечення автоматичної оптимізації програмного забезпечення на різних апаратних платформах.

Метод. Ґрунтується на генетичних алгоритмів для автоматизації процесу компіляції. Основніими етапами якого є: ініціалізація популяції – створення початкового набору параметрів компіляції; оцінка функції пристосованості – визначення ефективності кожної комбінації параметрів; еволюційні операції застосування схрещування, мутації та відбору для створення нового покоління параметрів; критерії зупинки – завершення ітераційного процесу при досягненні оптимального результату або стабілізації метрик.

Результати. Розроблений алгоритм був реалізований на Python із використанням бібліотек numpy, multiprocessing і subprocess. Для оцінки ефективності алгоритму використовувалися метрики часу виконання, споживання енергії та обсягу використаної пам'яті.

Висновки. Наукова новизна дослідження полягає в розробки: інноваційного підхіда до автоматичної оптимізації параметрів компіляції, заснованому на використанні генетичних алгоритмів; метода динамічного вибору стратегій оптимізації на основі метрик продуктивності для різних архітектур; інтеграції ГА із сучасними компіляторами, такими як LLVM, для автоматичного аналізу проміжного подання та оптимізації кодів; методів застосування адаптивних компіляторів для вирішення задач кросплатформної оптимізації. Практична значимість визначається у використання генетичних алгоритмів в адаптивних компіляторах, що дозволяє значно підвищити ефективність процесу компіляції, автоматизуючи вибір оптимальних параметрів для різних архітектур. Запропонований підхід може бути успішно застосований у галузях мобільних обчислень, хмарних технологій і високопродуктивних систем.

КЛЮЧОВІ СЛОВА: генетичні алгоритми, адаптивні компілятори, кросплатформна оптимізація, автоматизація.

ЛІТЕРАТУРА

1. FOGA: Flag Optimization with Genetic Algorithm / [B. Tagtekin, B. Höke, M. K. Sezer, M. U. Öztürk] // Proc. IEEE Int. Conf. on Innovations in Intelligent Systems and Applications (INISTA 2021), 29 June–1 July 2021, Kocaeli, Turkey. – IEEE, 2021. – P. 1–6. – DOI: 10.1109/INISTA52462.2021.9456364.
2. Júnior N. L. Q. Combining Machine Learning with a Genetic Algorithm to Find Good Compiler Optimizations Sequences / N. L. Q. Júnior, L. G. A. Rodriguez, A. F. da Silva // Proc. 19th Int. Conf. on Enterprise Information Systems (ICEIS 2017). – Vol. 1. – Porto, Portugal, 26–29 Apr. 2017. – P. 397–404. – DOI: 10.5220/0006270403970404.
3. Optimizing LLVM Pass Sequences with Shackleton: A Linear Genetic Programming Framework / [H. Peeler, S. Li, A. Sloss et al.] // Companion Proc. Genetic and Evolutionary Computation Conf. (GECCO '22), 9–13 July 2022, Boston, USA. – ACM, 2022. – P. 578–581. – DOI: 10.1145/3520304.3528945.
4. A Survey on Compiler Autotuning Using Machine Learning / [A. H. Ashouri, W. Killian, J. Cavazos et al.] // ACM Computing Surveys. – 2018. – Vol. 51, № 5. – Art. 96. – 42 p. – DOI: 10.1145/3197978.
5. Smart Selection of Optimizations in Dynamic Compilers / [V. M. do Rosario, E. Borin, A. Nacul et al.] // Concurrency and Computation: Practice and Experience. – 2021. – Vol. 33, № 10. – e6089. – DOI: 10.1002/cpe.6089.
6. Cummins C. ProGraML: A Graph-Based Program Representation for Data Flow Analysis and Compiler Optimizations / [C. Cummins, Z. V. Fisches, T. Ben-Nun et al.] // Proc. 38th Int. Conf. on Machine Learning (ICML 2021), 18–24 July 2021, Virtual Event. – PMLR, 2021. – Vol. 139. – P. 2244–2253.
7. Nugteren C. CLTune: A Generic Auto-Tuner for OpenCL Kernels / C. Nugteren, V. Codreanu // Proc. IEEE 9th Int. Symp. on Embedded Multicore/Many-core Systems-on-Chip (MCSoC 2015), 23–25 Sept. 2015, Turin, Italy. – IEEE, 2015. – P. 195–202. – DOI: 10.1109/MCSoC.2015.33.
8. Automatic Tuning of Compilers Using Machine Learning / [A. H. Ashouri, G. Palermo, J. Cavazos, C. Silvano]. – Cham : Springer, 2018. – 78 p. – (PoliMI SpringerBriefs in Applied Sciences). – DOI: 10.1007/978-3-319-71489-9.
9. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences / [L. G. A. Martins, R. Nobre, J. M. P. Cardoso et al.] // ACM Transactions on Architecture and Code Optimization (TACO). – 2016. – Vol. 13, № 1. – Art. 14. – DOI: 10.1145/2883614.
10. GEVO: GPU Code Optimization Using Evolutionary Computation / [J.-Y. Liou, X. Wang, S. Forrest, C.-J. Wu] // ACM Trans. on Architecture and Code Optimization. – 2020. – Vol. 17, № 4. – Art. 33. – 28 p. – DOI: 10.1145/3418055.

DOI 10.15588/1607-3274-2025-4-16