electronics

MDPI

Article

Compiler Optimization Parameter Selection Method Based on Ensemble Learning

Hui Liu [1,2], Jinlong Xu [3,*], Sen Chen [1,2] and Te Guo [1,2]

[1] College of Computer and Information Engineering, Henan Normal University, Xinxiang 453007, China
[2] Key Laboratory of Artificial Intelligence and Personalized Learning in Education of Henan Province, Xinxiang 453007, China
[3] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China
* Correspondence: xujinlong_pla@126.com

Abstract: Iterative compilation based on machine learning can effectively predict a program's compiler optimization parameters. Although having some limits, such as the low efficiency of optimization parameter search and prediction accuracy, machine learning-based solutions have been a frontier research field in the field of iterative compilation and have gained increasing attention. The research challenges are focused on learning algorithm selection, optimal parameter search, and program feature representation. For the existing problems, we propose an ensemble learning-based optimization parameter selection (ELOPS) method for the compiler. First, in order to further improve the optimization parameter search efficiency and accuracy, we proposed a multi-objective particle swarm optimization (PSO) algorithm to determine the optimal compiler parameters of the program. Second, we extracted the mixed features of the program through the feature-class relevance method, rather than using static or dynamic features alone. Finally, as the existing research usually uses a separate machine learning algorithm to build prediction models, an ensemble learning model using program features and optimization parameters was constructed to effectively predict compiler optimization parameters of the new program. Using standard performance evaluation corporation 2006 (SPEC2006) and NAS parallel benchmark (NPB) benchmarks as well as some typical scientific computing programs, we compared ELOPS with the existing methods. The experimental results showed that we can respectively achieve 1.29× and 1.26× speedup when using our method on two platforms, which are better results than those of existing methods.

Keywords: compile optimization; ensemble learning; optimized space search; feature extraction

Citation: Liu, H.; Xu, J.; Chen, S.; Guo, T. Compiler Optimization Parameter Selection Method Based on Ensemble Learning. *Electronics* **2022**, *11*, 2452. https://doi.org/10.3390/electronics11152452

Academic Editor: Miin-shen Yang

Received: 22 June 2022
Accepted: 1 August 2022
Published: 6 August 2022

1. Introduction

Atmospheric climate [1], inorganic chemistry [2], biopharmaceuticals [3], aerodynamics [4], market economics [5], quality engineering [6], and many other applications require the use of high-performance computing [7–15]. The continuous innovation and development of high-performance hardware architecture give high-performance computing more opportunities, and the challenges that come with them. With the failure of Moore's Law and Dennard Scaling, software performance limitations have severely impacted the performance of high-performance hardware. Program execution efficiency is more dependent on software than ever, especially for compiler optimization techniques.

Program performance optimization involves a variety of program transformations, and every transformation needs to use appropriate optimization parameters to improve the efficiency of program execution. Appropriate optimization parameters can significantly reduce the run time of programs. Therefore, optimization parameter selection is a key point in the field of program performance optimization. Most compiler designers typically use their experience to make source code modifications manually and try multiple optimization parameter options to find the optimal performance optimization parameters for a

particular application or function. Currently, there are two methods to solve optimization parameter selection: iterative compilation [16,17] and machine learning-based iterative compilation [18–20].

Iterative compilation technology uses search algorithms to explore compiler optimization phases, and different optimization phases can generate different versions of the program. Then, the version with the maximum performance improvement after running different versions on target platforms can be selected. Normally, iterative compilation can bring a better performance improvement compared with static compilation methods. However, the choice of compiler transformation parameters, transformation sequences, and numbers in iterative compilation all require a very large search space. More importantly, iterative compilation is a completely nonautomatic search, without using the effective experience gained before. We need to use manual optimization to reduce the searching blindness most of the time, which is impractical for programmers when the data scale is very large.

Machine learning-based iterative compilation optimization takes advantage of the experience obtained by the previous iteration compilation and constructs the optimization parameter prediction model using machine learning algorithms. The inputs of the prediction model are the features of a program and the outputs are the optimization parameters. For a new program, optimization parameters can be directly predicted by the prediction model. A mechanical search is unnecessary for the optimization parameter space. Most of the existing machine learning-based iterative compilation optimization methods use a genetic algorithm to determine the optimization parameter space, resulting in a long search time and a low efficiency. In addition, the existing methods usually use the static feature representation method to extract the program features. However, dynamic feature representation usually works better to achieve better predictive performance, while we need to run a program at least once to find the dynamic information. There are complex dependencies between the program features and optimization parameters, or between the optimization parameters themselves. Existing methods usually use a separate machine learning algorithm to build prediction models, such as K-nearest neighbor, decision tree, logistic regression, support vector machine, and Bayesian network [21–23]. Many theoretical and experimental studies have proven that building a unified classifier through the integration of multiple learners together will make the ensemble classifier prediction more accurate [24].

In view of the above problems, we propose an ensemble learning-based optimization parameter selection (ELOPS) model to select the optimal parameters. ELOPS is a machine learning-based iterative compilation optimization method, aiming at improving the efficiency of choosing the parameters of optimizations that are not Boolean and maximizing the performance of compiler optimization. Ensemble classifier technology is a method of constructing classification models by using multiple classifiers. The prediction performance will be improved effectively by utilizing a variety of different types of learning classifiers [25]. ELOPS analyzes the complex dependencies of compiler optimization parameters through an ensemble learning model, and implements the optimization parameter decision at the function level.

As the existing research usually uses a separate machine learning algorithm to build prediction models, in order to further improve the prediction accuracy, we built a statistical learning classifier based on stacking ensemble learning techniques to predict the program transformation optimization parameters. As most of the existing methods use a genetic algorithm to search the optimization parameter space, resulting in a long search time and a low efficiency, in order to further improve the search efficiency and accuracy, an improved adaptive multi-objective particle swarm optimization (PSO) algorithm was proposed to resolve the constrained optimization parameter problem, and the search efficiency of the optimization parameters was improved in the search space. Furthermore, we used a feature-class relevance (FCR) to find the best feature subset of static and dynamic features of the program, rather than using static or dynamic features alone. The FCR can maintain

the main information of the program, and minimize the training time overhead. The experimental results of various benchmarks verify that the ELOPS method can bring significant performance improvement compared with the baseline methods using only one statistical classifier. The main contributions of this paper can be summarized as follows:

(1) For program optimization parameter search problems, an improved adaptive multiobjective PSO was proposed to explore the parameter optimization search space. Compared with the default standard optimization level -O3 of GCC, we obtained 1.39× and 1.36× speedup on two platforms at the training phase of the model.
(2) We proposed a program feature representation method FCR, which uses the mixed static and dynamic features to represent the programs. Based on FCR, the optimal feature subset of the original feature was applied to the ensemble learning process to minimize the training time cost.
(3) Program features and the optimal optimization parameters constitute the sample data of the ELOPS model. We used the sample data to establish a statistical model, and then integrated the model into the compiler framework. The ELOPS model can execute the heuristic selection of optimization parameters and compile the process drive. Given the new target program, the ELOPS model can extract the program features as input, and predict the optimal compiler optimization parameter effectively. We achieved 1.29× and 1.26× speedup on two platforms at the prediction phase of the model. At the same time, we saved considerable search time.

Section 2 introduces the related work. Section 3 introduces the ELOPS method based on ensemble learning. Section 4 elaborates the optimization space search algorithm of choosing optimization parameters, including the GA and an improved PSO. Section 5 explains the selection of compiler parameters, including the framework and program feature representations. Section 6 presents the experimental results and analysis, and Section 7 is the conclusion.

2. Related Work

All kinds of optimization transformations can be embedded in the iterative compilation process, and this process can find program optimization that has superior performance relative to those in commercial compilers [26]. However, iterative compilation relies on multiple execution of the object code to find the optimal program version, which is usually time-consuming and costly. For the time being, the research focus on iterative compilation can be categorized as follows: (*i*) The selections of compiler optimization. (*ii*) Iterative compilation overhead. The selection of program optimization transformation includes parametric and nonparametric. Parameterization means the selection of transformations that can be parameterized, and the choices between possible values of parameters such as loop tiling and array padding [20,27,28]. Nonparametric mainly studies the optimization pass (phase) selection and phase order implementation problems. The selection and implementation of these optimization passes have impacts and rely on each other. It is a rather sophisticated matter to decide the kind, order, and number of transformations [29–33]. We concentrate on the prediction of program optimization parameters and mainly introduce the relevant work of compiler optimization transformation as a consequence. We can classify the optimization transformation prediction based on iterative compilation into two categories: the analytic model method and the predictive model method.

2.1. Analytic Models

Program optimization parameter selection based on the analytic method constructs analytic models for target programs and traverses the optimization space on the basis of the analytic model. The analytic models use the cost assessment model to obtain the benefits or costs of each optimization configuration in the optimization space and select optimization parameters with the largest income and the minimum cost.

Programs, computationally, take a large proportion of the running time for the nested loops. The polyhedral model brings about a very useful abstract description to explain

transformations in these loop nests through a dynamic iteration of the respective statement as an integer set in a clearly defined space, and the space is named the polyhedron of the statement. Pouchet et al. [34] classified the program transformation into iterative fields, iterative scheduling, and access functions modification and used the polyhedron framework to represent these three types of operations, which have been implemented in the Open64 compiler.

Nobre et al. [35] employed the compiler optimization selections generated beforehand for iterative compilation. Purini et al. [29] utilized lower sampling methods to make the optimization search space tend to decrease, and then recognize compiler optimization for application programs. Fang et al. [36] put forward an iterative compilation technology through the data warehouse and implemented compilation optimization combination and cost-benefit analysis by marking performance on the master server.

These methods based on the analysis model can effectively select the program transformation optimization parameters, but it is very complicated to construct an analysis model that can fully reflect the complexity of the modern architecture and the compiler. When the target platform architecture changes, the original analysis model must be adjusted. This makes the analytical model method inefficient, and the model generalization is poor. This makes the method of analytical modeling inefficient and poor in model generalization.

2.2. Predictive Models

The predictive model of program optimization transformations is based on machine learning theory. The model transforms the application program through the patterns that machine learning can be used as its input and makes the prediction model learn by means of artificial intelligence. In principle, the prediction does not rely on either platform or program and is able to provide evaluations for various program versions, drastically lowering the overhead of the compiling process.

Milepost is a compiler using the machine learning on different target platforms; the compiler can automatically adjust optimization parameters for the optimum compiling and running time, program length, and so on [37,38]. Ashouri et al. [39] first clustered the good optimizations and then used the results on a predictive modeling to find good ordering of phases. The research put forward MiCOMP, which makes use of subsequences generated from LLVM and a full-sequence speedup predictor. Agakov et al. [19] concentrated on constructing a machine learning algorithm to predict the optimum compiler optimization parameters, and the learning algorithm depends on features gathered offline or online. After model training, it is able to generate a series of compiler optimization selections for elevating program performance if the target application has been assigned. Wang et al. [40] used profile-driven parallelism detection to overcome the drawback of static code analysis, and the recognition of further parallel applications became possible. They employed a prediction method based on machine learning to enable a better suited mapping mechanism automatically when using various target architectures. This method counted on the ultimate agreement from the user, so it was a semiautomatic approach, while our approach can automatically predict the value of the optimization parameters.

Park et al. [21] adopted a polyhedral compiler structure that which can predict the speedup for an application that has not been seen before. Loop-level optimization in the optimization space is employed, and the average speedup is obtained by utilizing various machine-learning algorithms in the WEKA environment. Their prediction model is constructed through the program features collected from performance counters. Our method uses mixed static and dynamic features, which can select the most useful features at the compiling time and the running time for each program. Furthermore, we propose a new optimization parameter search algorithm, and we use the stacking ensemble learning prediction model.

Cummins et al. [41] proposed CLgen; in order to create the performance autotuning, a benchmark synthesizer was put forward to form integrated programs. Ashouri et al. [42] proposed a Bayesian network model to derive the optimum compiler optimization parame-

ter for an embedded processor. They utilized the program's mixed features as the input of the trained model and showed that the method can surpass 50% of the GCC default optimization performance. Our work focused on choosing the optimization parameters that are not Boolean for the programs, such as the factors of unrolling loop, tiling loop, and array padding, while the existing work mainly focuses on finding good phase ordering. We believe our work is a useful supplement to existing work.

Leather et al. [43] proposed a method of learning static code features for compilers heuristically in the optimization process. Ding et al. [44] proposed an analytical approach, in which the method can automatically define which learning algorithms are suitable for the optimization process. Using a clustering algorithm, Martins et al. [45] proposed a frame closely related to the program. Kulkarni et al. [46] explored the optimization space exhaustively on function-level granularity, and solved the optimization selection problem through a search tree algorithm. Ballal and Kumar et al. [47,48] proposed compiler optimization based on genetic programming, and implemented compiler optimization selection through parallelism GA programming on the architecture with multiple cores.

Machine learning-based iterative compilation optimization methods use experience obtained by the previous iterative compilation and build the compiler optimization parameter prediction model through a machine learning algorithm. The prediction model is employed for predicting the optimization parameters of the fresh target program and avoids the repetitive test of the traditional iterative compilation method. However, when collecting the prediction sample collection, most of the existing methods use the static program feature representation and use the GA to explore the optimization parameter space of different compilers. In addition, the existing methods generally use a single machine learning algorithm when building models. Therefore, there are some problems, such as incomplete features, inefficient search algorithms, and low model prediction accuracy. In this paper, an FCR technique was used for program feature representation, and an improved adaptive multiobjective PSO algorithm was proposed to improve search efficiency. Based on the stacking ensemble learning technique, we built an effective prediction model for choosing program optimization parameters.

3. Modeling the Program Optimization Parameters Selection Problem

3.1. Definition of Optimization Parameters

To better describe the problem of parameter selection in program transformation optimization, we formulate the problem first.

Transformation indicates that a partial function F, which can take program P as its input, and a semantics equivalent program P' as its output, that is, F can change the order of a code segment while keeping the semantics.

Optimization parameters are the main part related to parameterization transformation. For example, the transformation parameters of loop unrolling are unrolling factors.

Assuming that an application program has s performance critical parameters $\{q_1, q_2, \cdots, q_s\}$, each parameter q_i is called an optimization parameter. The parameters in program performance optimization are usually integers, so $q_i \in Z, i = 1, 2, \cdots, n$. In addition, based on the knowledge of compiler optimization, each optimization parameter q_i has a lower and upper bound, i.e., $low \leq q_i \leq top$. The order of optimization parameters $\{q_1, q_2, \cdots, q_s\}$ is called an optimized parameter vector defined as $q = \{q_1, q_2, \cdots, q_s\}$.

The optimization parameter selection aims to find the optimal optimization parameter vector q to minimize the runtime of the target programs. Assuming T(x) is the execution time when using optimization parameter vector q, the optimization parameter selection problem is transformed into an equal problem: the vector $q = \{q_1, q_2, \cdots, q_s\}$ is chosen if

the execution time T(x) is the smallest, which is a constrained optimization problem with multiple objectives, defined as:

$$\begin{gathered} \min T(q) = (t_1(q), t_2(q), \cdots, t_k(q)) \\ \textbf{s.t.} \\ g_i(q) \leq 0, i = 1, 2, \cdots, p \\ h_j(q) = 0, j = 1, 2, \cdots, m \\ q = \{q_1, q_2, \cdots, q_s\}, q \in R^S \\ q_i^{\min} \leq q_i \leq q_i^{\max},\ i = 1, 2, \cdots, s \end{gathered} \tag{1}$$

where $T(q)$ is the objective function, and the vector $q = \{q_1, q_2, \cdots, q_s\}, q \in R^S$ is an S dimensional decision vector, called a feasible solution. $g_i(q)$ and $h_j(q)$ are the constraints. $q_i^{\min}$ and $q_i^{\max}$ are the upper and lower bounds of the ith decision variable. The space formed by all feasible solutions is called the solution space, denoted as Ω. The allowable error $\delta > 0$ can be set to transform the equality constraint $h_j(x) = 0$ into an inequality constraint, i.e., $\left|h_j(x)\right| \leq \delta$.

Based on the understanding of the program transformation optimization parameters, the most effective parameterization code transformations are loop unrolling, loop tiling, and array padding [20,27,49]. Therefore, this paper focused on the constrained multiobjective optimization parameters of these three typical transformations. For these three parameterized optimizations, we utilized ensemble learning for optimization parameter predictions. For other nonparametric optimizations we used the default optimization of GCC.

Loop unrolling copies the statements in the loop basic block many times and reduces the number of loop iterations to reduce loop branches and better prefetch data. Among them, the number of copies of the loop body is called the loop expansion factor. Loop unrolling is often used to reduce loop overhead, providing instruction-level parallelism for processors with multiple functional units. However, this transformation may also lead to negative effects on program performance. For example, excessive loop unrolling will cause register spilling, and the register resource will be wasted when the unrolling factor is too small. As a result, the key issue in loop unrolling is the choice of unrolling factor. As shown in Figure 1a, if the unrolling factor is set to 2, the unrolling of the loop is presented in Figure 1b. If the factor is set to 4, as shown in Figure 2, the unrolling technique improves the program performance by executing the same operations parallel. The constraint in the unrolling factor is:

$$\begin{gathered} c_1 U_1 * c_2 U_2 * \cdots * c_n U_n \leq N_R \\ 1 \leq U_n \leq N_R (i = 1, 2, \cdots, n),\ U_i \in N \end{gathered} \tag{2}$$

where c_i is the program-related constant that represents the number of registers in the body of the loop body corresponding to the i-th loop. U_i is the unrolling factor, and N_R is the number of registers in the processor.

Before loop unrolling	After loop unrolling
1 for(j=0;j<n;j++){	1 for(j=0;j<n;j++){
2 for(i=0;i<2;j++){	2 S1(0);
3 S1(i);	3 S2(0);
4 S2(i);}	4 S1(1);
5 }	5 S1(1)}
(a)	(b)

Figure 1. Loop unrolling transformation. (**a**) Before loop unrolling; (**b**) After loop unrolling.

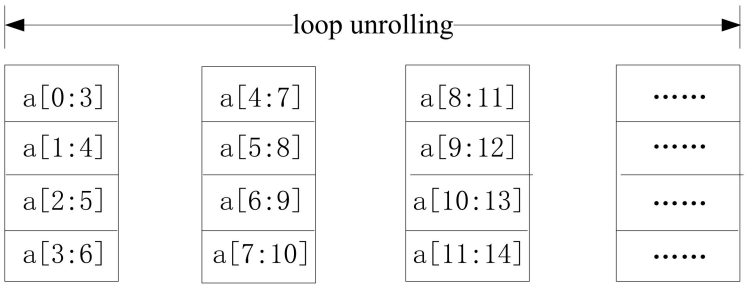

Figure 2. The unrolling factor is set to 4 for program transformation.

Loop tiling uses the affine transformation to change the access order of the loop iteration, improving the locality of the data. The approach can fully exploit tiling data reuse and reduce data transformation and cache misses. Using the appropriate loop transformation technique, loop tiling can realize data parallelism between tilings and within the tiling by eliminating the data dependency. The choice of the tiling factor determines the performance of the tiling code. The constraint in the tiling factor is shown in Equation (3), where T_i, S_k, M, and L_k are the tiling factor, size of the array element, memory capacity, and array dimension, respectively. The tiling factor of the loop nesting $L_i(i = 1, 2, \cdots, n)$ is b_i. If T_i is the jth dimension tiling factor of the array, $T_i = b_i$ when $b_i = 0$. Otherwise, this dimension is not to be tiled and $T_i = t_i$, where t_i represents the number of elements of the jth dimension. $x_{(k,j)} = 1$ means we can tile the jth dimension of the array while $x_{(k,j)} = 0$ means we cannot.

$$\begin{gathered} \sum_{k=1}^{m}\left(S_k * \prod_{i=1}^{l_k} T_i\right) \leq M \\ T_i = \begin{cases} b_i, & x_{(k,j)} = 1 \\ t_i, & x_{(k,j)} = 0 \end{cases} \end{gathered} \tag{3}$$

Array padding is typically for multidimensional array optimization by filling the minimum dimension length of the multidimensional array with a multiple of the vector registers. Each access is address-aligned and a nonaligned address should be avoided. Array padding requires additional space overhead and an excessive array filing factor increases the number of translation lookaside buffer (TLB) failures. In this paper, we limited the array padding factor within a range from 0 to 64. As shown in Figure 3a, assuming that the length of the vector register is 256 bits, each vector register can load four floating-point array elements. The minimum length of array a is 623, not an integer multiple of 4. It is difficult to determine whether the array reference a [*i*] [0] is aligned or not in each iteration. If the array dimension length is increased by 1 and becomes 624, which can be divisible by 4, we can use the aligned vector instructions for code generation, although we need slightly more storage space. The loop after array padding is shown in Figure 3b.

Before array padding

```
1 double a[623][623]
2 for(i=0;i<623;i++)
3   for(j=0;j<623;i++)
4       a[i][j]=a[i][j]*3.0;
```

(a)

After array padding

```
1 double a[623][624]
2 for(i=0;i<623;i++)
3   for(j=0;j<623;i++)
4       a[i][j]=a[i][j]*3.0;
```

(b)

Figure 3. Multidimensional array padding example. (**a**) Before array padding (**b**) After array padding.

3.2. Program Transformation Optimization Parameter Selection Model

The main goal of ELOPS is to determine the optimal compiler optimization parameters to be applied by the target program. The ELOPS model associates program features with

the optimization parameter of the compiler to gain the maximum execution efficiency. The structure of ELOPS is displayed in Figure 4, including the model training (learning phase) and prediction process (inferring phase). The benchmark program is divided into two sections in the phase of training the model, namely training set $P_1, P_2, \cdots, P_N$ and test set P_N.

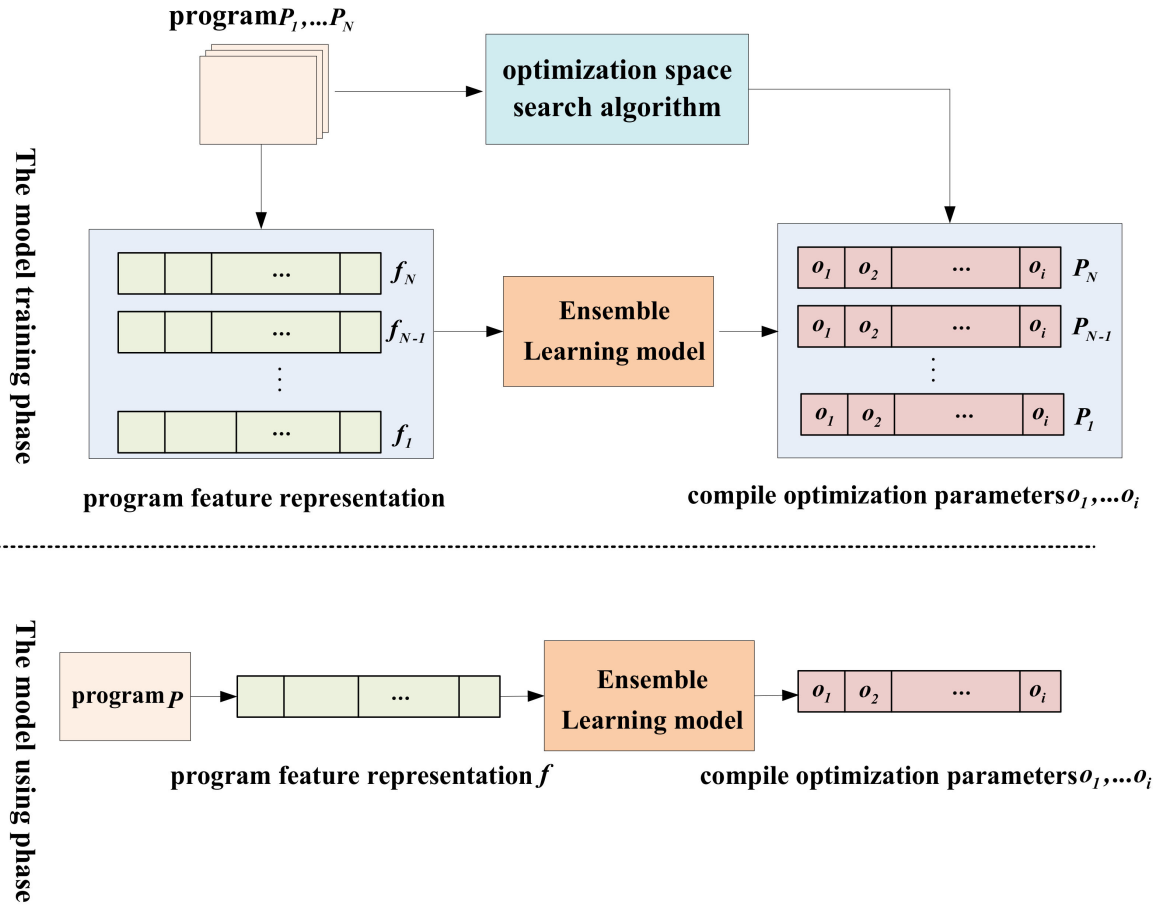

Figure 4. Compiler optimization parameter selection model framework.

Function features were extracted both statically and dynamically, and the best feature subset of the original set was selected through the feature-class relevance method. Using an optimization space search algorithm, an ensemble learning method was used to establish the statistical model framework. The model takes the function features as its input, and the approximate optimal optimization parameter of the function is the output. The details of this ensemble learning are described in Figure 5. The function feature values and the approximate optimal optimization parameters in the training set construct the training samples of the ELOPS model. For the sake of improving the generalization efficiency of the program and prediction accuracy, we chose the leave-one-out cross validation method for our model training implementation. In the model prediction phase, we extracted the function features of a new program first and then predicted the best compiler optimization parameters based on the knowledge stored in the ELOPS model.

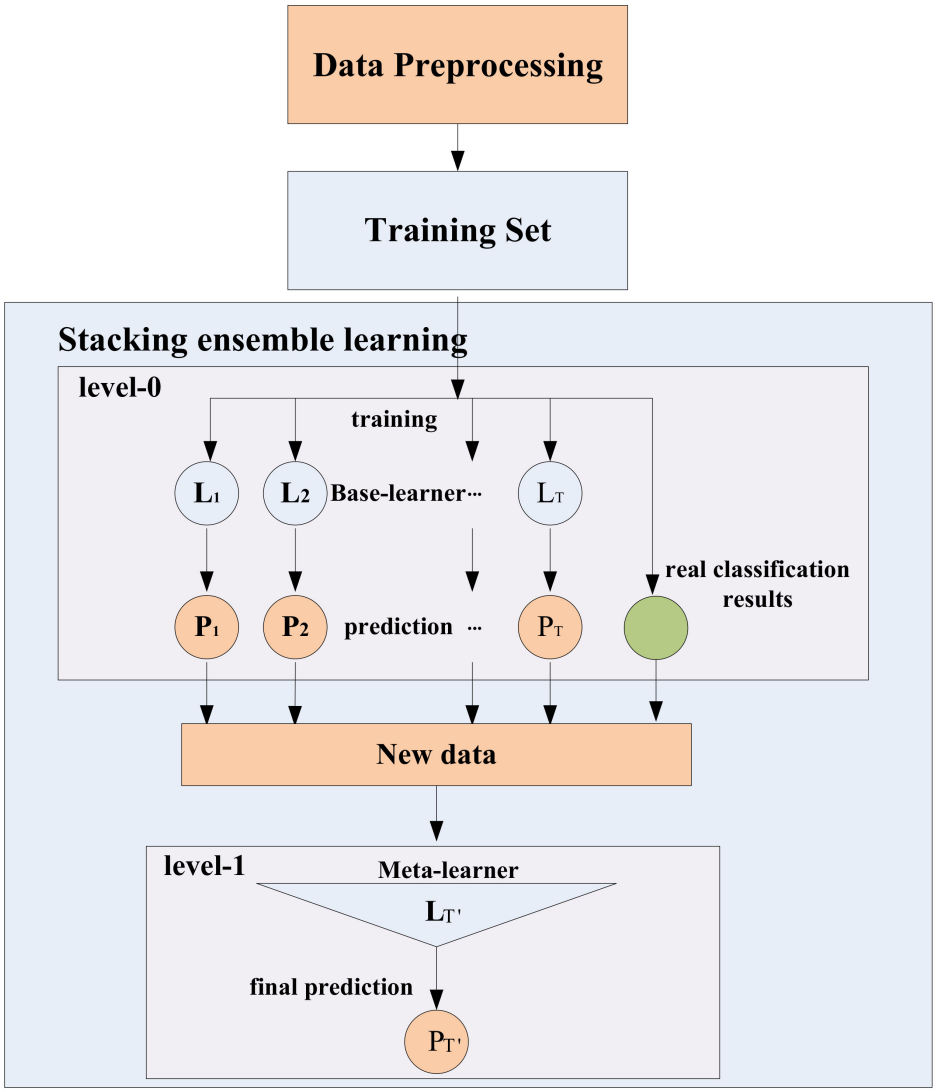

Figure 5. Stacking ensemble learning framework.

4. Optimization Space Search Algorithm for Optimized Parameters

The optimization space search is a type of algorithm with global optimum performance, which is very general and well suited for parallel computing. The method of space search requires a strict theoretical basis and searches solution spaces according to certain rules. Instead of the experiences from experts being reused, the search algorithm is capable of finding the solution that is optimum or being an approximation with a guaranteed period of time. The commonly used optimization space search algorithms include the genetic algorithm (GA), particle swarm optimization (PSO), simulated annealing, tabu search, etc. In this paper, we proposed an improved PSO algorithm to search the optimization space of compiler optimization parameters, and we used GA for comparison.

4.1. Genetic Algorithm

The genetic algorithm (GA) is a group search algorithm based on the natural selection of the biological evolution process. It has a strong search ability and is not limited to the specific model. In addition, GA can provide effective support for most kinds of complex decision optimization problems [50–52]. The GA can be understood as follows: given a map $f : S \to R^n$ and parameter optimization problem $\min_{x \in S} f(x)$, we need to determine the set $G = \{T | T = (a_1, a_2, \cdots, a_l), a_i \in A, i = 1, 2, \cdots, l\}$, in which A is a nonempty set and i is

a positive integer. We can construct mapping $g : G \to R^n$ and $d : G' \to S, G' \subset G$ defined on G and its subsets G', respectively. On this basis, we can obtain the following formula.

$$\forall T^* \in G', g(T^*) = \min_{T \in G} g(T) \Rightarrow f(d(T^*)) = \min_{x \in S} f(x) \tag{4}$$

We use various genetic operations to search the element $T^* \in G$, s.t. $g(T^*) = \min_{T \in G} g(T)$. As a result, the optimization problem can obtain the optimum solution $X^o = d(T^*)$. In practical applications, the coding space G is generally a finite set, which usually cannot guarantee that the original problem's optimum will exist in its representation space.

Consequently, we modify Formulas (4) to (5), in which ε is the error acceptable for the optimal solution which is also called the precision requirement, and $N_\varepsilon(X^o)$ is the ε neighborhood of X^o.

$$\begin{aligned} &\forall T^* \in G', g(T^*) = \min_{T \in G} g(T) \\ &\Rightarrow d(T^*) \in N_\varepsilon(X^o)(X^o \in S) = \min_{x \in S} f(x), \varepsilon > 0 \end{aligned} \tag{5}$$

In this paper, we used the same GA configuration for all hotspot functions in the test set to minimize the search space and selected the optimal solution from the search space. The optimal solution is the best compiler optimization parameter vector $q = \{q_1, q_2, \cdots, q_S\}$. We created an initial chromosome population, where each chromosome is represented by a certain optimization parameter vector, and similarly, each gene in that chromosome was represented by an optimization parameter. Each generation created 40 chromosomes and evolved for 50 generations. Algorithm 1 shows the details of the evolution.

Algorithm 1 Optimization parameter selection algorithm based on GA

1: **Input:** Hotspot function h, optimization parameter vector q
2: **Output**: The best optimization parameter vector q^* and its corresponding program running time $T(q^*)$
3: **Algorithm:**
4: Step 1: Initialization: Read the input parameters and set the population size N, the largest genetic generation M. Then, generate the initial population $Pop = [x_1, x_2, \cdots, x_N]$ with N individuals and each individual corresponds to an optimized parameter vector.
5: Step 2: For every chromosome in the current generation, calculate the fitness value. Ftness function *Fitness(T)* is represented as reciprocal form on the summed time for both compilation and execution of the function when the corresponding optimization parameter is adopted: $Fitness(T) = \frac{1}{T_{compilation} + T_{execution}}$
6: Step 3: Select: choose the individual to participate in the cross, select operator using a roulette selection method and elite strategy. First, choose a small proportion of the elite from the current generation Pop_j, directly into the next generation of population, and then select the remaining individuals based on roulette method.
7: Step 4: Cross: perform arithmetic cross operation, set the crossover probability $P_{crossover} = P_{mutation} = 0.6$.
8: Step 5: Variation: mutate the individual in the population, set the mutation probability to $P_{mutation} = 0.3$.
9: Step 6: algorithm termination condition judgment. When the genetic generation arrives at the peak genetic generation M or the number of optimal individuals' consecutive generations does not change, the algorithm terminates, go to step 7; otherwise $k = k + 1$, go to step 2.
10: Step 7: Select the optimization parameter vector q^* and the program run time $T(q^*)$ corresponding to the individual and the output will be the one that has the highest value of fitness in the current population.

4.2. Multiobjective PSO Based on Adaptive Constraints

4.2.1. Standard PSO Algorithm

Particle swarm optimization (PSO) is a multipoint search algorithm whose basic idea is that there are D-dimensional search spaces and some particles without weight [53,54]. The volumes are randomly distributed in the search space. Each particle has two values: position and speed vector. The particle's position represents a possible solution, and the particles move in the direction of the speed vector in the solution space. For each iteration, the particles will fly a distance to produce a new position. The individual and population-optimal positions of the particles are determined by the fitness function. On this basis, we can decide the new speed vector and continue to adjust the position until the best is found. The standard PSO evolutionary learning formula is as follows:

$$v_{id}(k+1) = v_{id}(k) + c_1 r_1 (P_{id}(k) - x_{id}(k)) + c_2 r_2 (P_{gd}(k) - x_{id}(k)) \tag{6}$$

$$x_{id}(k+1) = v_{id}(k+1) + x_{id}(k) \tag{7}$$

where $v_{id} = (v_{i1}, v_{i2}, \cdots, v_{iD}), 1 \le i \le N$ denotes the speed information of the ith particle, and $x_{id} = (x_{i1}, x_{i2}, \cdots, x_{iD}), 1 \le i \le N$ the location information. k stands for the number of iterations. c_1, along with c_2, acting as the learning factors that determine the value of the speed change vector. $r_1 \in [0, 1]$ and $r_2 \in [0, 1]$ represent two random coefficients used to prevent particles from flying out from the search space. The standard PSO algorithm can only solve the optimization problem with no constraint and only one objective being present. For the multiobjective optimization accompanied with constraints, we need to consider the constraint processing method and the boundary formed by the Pareto optimal solution, namely, the Pareto front end [55].

4.2.2. Adaptive Multiobjective Particle Swarm Optimization

When the constrained problem is optimized, the degree of particle constraint violation can be defined according to the constraints conditions, and the particles can be divided into feasible particles (FP) and infeasible particles (IFP). According to the standard PSO algorithm, if a particle falls into a feasible domain, the IFP near the constraint boundary will deviate from the boundary because of being fascinated by the global optimal solution in the feasible domain when the optimal solution exists in the feasible domain boundary. This phenomenon will result in missing the useful information around the bounds. Therefore, we propose a method to mitigate the speed of IFP flying near the constraint boundary through adaptive learning, namely, the adaptive multiobjective PSO algorithm (AMPSO).

In the AMPSO algorithm, we update v_{id} and x_{id} for FP according to the standard PSO algorithm. For the IFP generated in the population evolution process, the normalized processing method is used to obtain the constraint violation degree:

$$C_{nor}(x) = \frac{c(x) - \min_x C(x)}{\max_x C(x) - \min_x C(x)} \tag{8}$$

Using $C_{nor}(x)$ to influence the IFP learning factor and modifying $C_{nor}(x)$ and x_{id}, we can obtain the adaptive evolutionary learning formula:

$$v_{id}(k+1) = v_{id}(k) + c_1 r_1 (P_{id}(k) - x_{id}(k)) + c_2 C_{nor}(x) r_2 (P_{gd}(k) - x_{id}(k)) \tag{9}$$

$$x_{id}(k+1) = v_{id}(k+1) + x_{id}(k) \tag{10}$$

According to (9) and (10), IFP learns adaptively according to $C_{nor}(x)$. Particles with a large degree of constraint violation, $c_2 C_{nor}(x) \approx c_2$, maintain the ability to follow the global optimal solution, and can deviate from the existing location to implement an unknown feasible domain search. For the particles around the constraint boundary,$C_{nor}(x) \approx 0$ and $c_2 C_{nor}(x)$ is the minimum. The attraction of the global optimal solution to such particles is weakened, and the search for the useful information near the constraint boundary is

completed. For comparison, while the standard PSO algorithm only improves the constraint boundary search ability, the AMPSO algorithm also makes full use of the particles with different $c_2C_{nor}(x)$ to complete the global optimal search to ensure that the particles can converge quickly to the real Pareto front end.

When using the PSO algorithm, the use of crowding distance (CD) can guide the particles to move from dense places to loose places to improve the diversity of the Pareto solution and maintain the algorithm search coordination. The CD strategy is used to describe the degree of density of other solutions distributed around an optimal solution.

For each objective function to be optimized, the nondominated solution with the concentrated optimal solution is sorted by the objective function size. For each solution, the average side length of the cube surrounded by two adjacent solutions is calculated. When the CD of all the optimal particles are determined, the CD values are sorted in descending order. CD with a smaller solution has a larger probability of being removed. If there is more than one CD with the smallest solution, we will remove the CDs randomly. Then we determine whether the particles in the external file exceed the limit and remove them if beyond. The remaining are the individuals with the minimum CD in the optimal Pareto solution. This process is repeated until the size of the particles in the external file meets the set value. We present a new CD calculation method, and the formula is:

$$D_i = \frac{1}{m}\sum_{k=1}^{m}\left|f_k^{i+1} - f_k^{i-1}\right| \tag{11}$$

$$I_i^2 = \frac{\sum_{k=1}^{m}\left(\left|f_k^{i+1} - f_k^{i-1}\right| - D_i\right)^2}{m} \tag{12}$$

$$DD_i = \frac{D_i}{\ln\frac{1}{I_i^2}} \tag{13}$$

where D_i represents the CD with m objective functions and f_k^i is the object function value of the ith individual after being sorted from small to large on the kth target dimension. I_i^2 is the CD variance of adjacent individuals, which reflects the degree of CD difference on each dimension. The larger the difference degree is, the easier it is to keep the particle. DD is the CD of particle i. Delete only the smallest CDs at a time, and the optimal solution of the affected CD component on each target dimension is found by the position information of the deleted optimal solution. Let Formula (13) recalculate the CD component of the affected optimal solution, update the CD, and continue until the Pareto optimal solution scale reaches the given value. This method can make the Pareto optimal solution have the best distribution and avoid falsely deleting the Pareto optimal solution. Algorithm 2 is the optimization parameter selection algorithm based on AMPSO.

This AMPSO algorithm's time complexity is $O(n(N+L)^2)$ in which n, N, and L are the problem's objective function number, the population size, and the Pareto nondominated solution scale, respectively. That is, the time to finish computing the whole algorithm is mainly concentrated on the Pareto nondominated solution construction. Therefore, the AMPSO proposed in this paper does not raise the overall algorithm's time complexity.

Algorithm 2 Optimization parameter selection algorithm based on AMPSO

1: **Input**: Test program P, optimization parameter number n, evolutionary parameters (w, c_1, c_2), population size N, Pareto nondominated solution size S, maximum number of iterations M.
2: **Output:** The best optimization parameter vector q^* and its corresponding program running time $T(q^*)$.
3: **Algorithm:**
4: Step 1: The range of values Ω is defined for each optimization parameter, and initializes the population location x_{id} and speed v_{id} in the given search field.
5: Step 2: Calculate the target function value of each particle in the population, i.e., the program execution time $t_i(x)$ and the constraint violation degree $C_{nor}(x)$.

6: Step 3: Select all feasible solutions,
$x = (x_1, x_2, x \cdots, x_D), x \in R^D, x_i^{\min} \leq x_i \leq x_i^{\max}, i = 1, 2, \cdots, n$, and construct the current Pareto nondominated solution set based on Pareto domination relation. If the current nondominated solution is larger than the given scale S, the CD algorithm is used to maintain the scale of the nondominated solution set at a given scale.
7: Step 4: If the current Pareto nondominated solution set is empty, select the particle with the smallest $C_{nor}(x)$ as the global optimal position. Otherwise, the solution with the largest CD in the solution set of Pareto nondominated will be selected and be the global optimal position.
8: Step 5: Update the individual optimal position of the particles: if there are two particles, one is feasible, and the other is unfeasible, then we select the feasible particles as excellent. If the two particles are all unfeasible, then select particle with small $C_{nor}(x)$ is better. If both particles are feasible, the nondominated particle is chosen based on the Pareto domination relation.
9: Step 6: Update the position x_{id} and speed v_{id} of all particles according to AMPSO.
10: Step 7: Determine if the biggest number of iterations M is achieved; if not, return to Step 2; otherwise, the loop ends and output the best optimization parameter vector q^* and its corresponding minimum program running time $T(q^*)$ are selected.

5. Optimized Parameter Prediction Based on Ensemble Learning

The traditional machine learning classifier is simply a classifier, and there are some problems in model fitting together with the degree of accuracy to classify. There is one valid solution for both called ensemble learning, which would better address the issues on the capabilities of the classifier. The ensemble classifier technology is a method of integrating multiple classifiers to construct the classification model. By utilizing the diversity of different classifiers, the variance is reduced without increasing the deviation to effectively improve the prediction performance. In the process of classifier construction, to ensure that the combined classifier produces a better classification effect than a single classifier, two principles need to be followed: one is that the prediction errors generated by different classifiers are irrelevant, and the other is that the classification effect of each classifier is at least better than that of random prediction.

5.1. Data Preprocessing

During the time that the sample data are being gathered, problems such as data redundancy, data loss, and data errors may exist. Using such problematic materials, the model that predicts may cause some unreliable accuracy. Therefore, we need to perform preprocessing of the original data dataset. The preprocessed dataset is segmented into two parts: the set for training and the set for testing and validation. In addition, due to the difference between the different parts of the program, the preprocessing of the dataset may lead to unbalanced data. We used the resampling strategy to resample the unbalanced dataset and then constructed the model based on the balanced sample distribution.

5.2. Stacking Ensemble Prediction Model

At present, the boosting and bagging ensemble learning algorithm is more extensive. The algorithm can integrate some weak predictors into a new strong predictor to enhance prediction accuracy and generalization ability. We used a different approach from boosting and bagging, called stacked generalization or stacking ensemble learning technology. Stacking differs from boosting and bagging since it uses a different learning algorithm to construct a learner, not just a simple combination of the same types of basic learners. Although we did not improve the ensemble learning algorithm itself, to the best of our current knowledge, we are the first to use this algorithm for the compiler optimization parameter selection.

Figure 5 shows the stacking ensemble learning framework, which has level-0 (base-learner) and level-1 (meta-learner). Level-0 is formed with training data through bootstrapping, and the output of level-0 is taken as level-1 input. During the model training phase, we use diverse learning algorithms, such as logistic regression, to generate level-0 learners. Then, we can obtain the Meta-learner $L_{T'}$'s training data, which contains the predicted

results $\{P_1, P_2, \cdots, P_T\}$ of the Base-learner and the real classification results of the validation set. The goal of level-0 is to assemble output sets reasonably and amend predictive errors in the base learner. For fresh data, predictions of each base learner are taken as the input of level-0, while level-1 can predict the result simultaneously. Algorithm 3 shows the optimization parameter selection algorithm based on the Stacking ensemble model.

Algorithm 3 Optimization parameter selection based on the Stacking ensemble model

1: **Input:** Dataset $D(x, y) = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$; level-1 learning algorithm $L_1, L_2, \cdots, L_T$ and level-2 learning algorithm L.
2: **Output:** Program optimization parameter classification.
3: **Algorithm:**
4: ***for*** $t = 1, 2, \cdots, T$
5: $h_t = L_t(D)$
6: ***end for***
7: $D' = \varnothing$
8: ***for*** $i = 1, 2, \cdots, n$
9: ***for*** $t = 1, 2, \cdots, T$
10 $Z_{it} = h_t(x_i)$
11 ***end for***
12 $D' = D \cup \{((z_{i1}, z_{i2}, \cdots, z_{iT}), y_i)\}$
13 ***end for***
14 $h' = L(D')$

5.3. Program Feature Extraction

We can divide program features into two categories: static and dynamic. When extracting static features of a program, there is no need to execute the program, and it can be obtained during the process of compiling the program. Dynamic features need to be extracted during the running process of the program, and we can use tools such as performance counters to collect them. Since the use of dynamic features needs to actually run code, there is a very large execution time penalty. Our ELOPS model uses both dynamic and static features, which are extracted by the Gprof tool and our own compiler instruction script. Another study by Liu et al. [56] presented the specific static and dynamic features we want to use. The static features we use include the number of basic blocks in the function, number of basic blocks with a single successor, and so on. The dynamic features we use involve cache line access, level-1 cache, and so on.

Since the machine learning modeling process is time-intensive, we use the feature-class relevance (FCR) to find the optimum feature subset. This method can implement correlation statistics of the program features and the compiler parameters category, and improve model prediction accuracy effectively while using as few main features as possible. Different from the method in reference [30], this paper calculated the correlation between program features and program optimization categories to select the FCR method that is most relevant to the target problem for the construction of program feature subsets.

We took three FCRs as follows:

Information gain (IG) is calculated based on information entropy, which is the decreasing part from a priori entropy to a posteriori entropy and reflects the degree of information eliminating uncertainty. We can implement feature selection through the sort of feature f based on the size of information gain. The relation between the amount of information and probability is monotonically decreasing; the smaller the probability is, the greater the information that can be contained. When using $\{c_i\}_{i=1}^{m}$ as the set of optimization parameter classes, the IG of feature f is:

$$\begin{aligned} IG(f) = &-\sum_{i=1}^{m} p(c_i) \log_2 p(c_i) + p(f) \sum_{i=1}^{m} p(c_i|f) \log_2 p(c_i|f) \\ &+ p(\overline{f}) \sum_{i=1}^{m} p(c_i|\overline{f}) \log_2 p(c_i|\overline{f}) \end{aligned} \tag{14}$$

Chi-square examination is a nonparametric test, and the chi-square checking can determine whether the class distribution is closely related to the features of the program. The degree of deviation determines the value of the chi-square:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{n_c} \frac{\left(O_{i,j} - E_{i,j}\right)^2}{E_{i,j}} \tag{15}$$

In Formula (15), r expresses how many eigenvalues feature f has. n_c shows the number of classes. When the feature is i and the category is j, $O_{i,j}$ is the actual observation value, and $E_{i,j}$ is the theoretical expectation value.

ReliefF is a feature-relevance computational method that determines the features' capabilities for classification according to the significance of those features. Each time, the *ReliefF* algorithm collects a sample M from the training set and finds K neighboring samples with distances as short as possible from both identical and different samples; then, the weights of every feature are upgraded.

5.4. Evaluating Indicator

In terms of evaluating the model's prediction performance, our research takes the indicators including precision, recall, and F1 metrics. We also express the prediction results with a confusion matrix. In the process of program training, we find the finite optimization parameter combinations, and we treat each combination as one class. In other words, we have $class\ 1, \cdots, class\ n$ optimization parameter combinations. When predicting optimization parameters for the new function, if the best optimization parameters of the function belonging to *class i* can be classified into *class i* correctly, record as True Positive (TP). If the best optimization parameters that do not belong to *class i* are misclassified into *class i*, record as False Positive (FP). If the best optimization parameters belonging to *class i* are misclassified to other classes, record as True Negative (TN). If the best optimization parameter does not belong to *class i*, it is correctly classified into other classes and recorded as False Negative (FN). On this basis, we can obtain the following measurement values:

(1) Precision: $precision = \frac{TP}{TP+FP}$

(2) Recall: $recall = \frac{TP}{TP+FN}$

(3) F1 metrics: $F1 = \frac{2 \times precision \times recall}{precision + recall}$

(4) AUC: the area under the ROC curve. The ROC curve was originally used to describe the trade-off relationship between income and cost. We use the Y axis to represent the true rate, and the X axis to represent the true negative rate. The range of a AUC is [0, 1]. The larger the area, the better the model.

We take the program execution efficiency speedup to analyze and evaluate the performance of the program when using different optimization parameters.

(5) Speedup: $speedup = \frac{T_{default}}{T_{ELOPS}}$

where $T_{default}$ expresses the program running time using the compiler's standard optimization parameters. T_{ELOPS} denotes the running time through the ELOPS model proposed in this paper.

6. Experiment Analysis

6.1. Experiment Environment

We applied the training and testing of our model on two platforms. The inputs are the program features of the functions and the outputs are the optimal optimization parameters of the functions.

Platform I: Redhat Enterprise AS 5.0 and the Chinese Shenwei 26,010 were used as the operation system and processor, with 2.5 GHz CPU clock, 32 KB L1 data cache, 256 KB L2 cache, and 8 KB basic page. The compiler version is GCC 5.1.

Platform II: Redhat Enterprise AS 5.0 and Intel Xeon E5520 were used as the operation system and processor, with 2.26 GHz CPU, 32 KB L1 data cache, and 1 MB L2 cache. The compiler version is GCC 5.1.

Training set: We used the SPEC CPU2006 benchmark [57] as the training set, which is developed for estimating the general-purpose CPU performance. In our research, we took the reference scale for our experiments. We selected 2500 hotspot functions from the training set and chose the best optimization parameter according to the AMPSO algorithm. This is because in order to better optimize program performance, we must find the performance bottleneck, that is, the hot function. We used the Gprof tool to select the hotspot functions of different programs. Then, we used the method proposed to predict the best compilation optimization parameters for the new programs.

Testing set: We used the NPB benchmark [58] as the testing set, which was selected for extensive comparison for parallel architectures. We also tested our ELOPS model for large-scale scientific calculation programs as shown in Table 1. The program involves atmospheric diving waves, seismic wave simulations, hydrodynamics, and weather forecasts.

Table 1. Scientific computation programs with large scale.

Program	Description	Code Lines
SWE	Shallow water equation	12,000
OpenCFD	An Open CFD code with High Order Accuracy Method	13,000
FDM	Seismic 3D forward modeling computing program	2847
WRF	The weather research and forecasting model	860,000
GKUA	Algorithm has been applied to the gas flow simulations	14,000

6.2. Experiment Design

Based on the two different experimental platforms, the experiments were mainly designed from four aspects: (1) comparison of search performance between the genetic algorithm and AMPSO method; (2) comparison of program performance for different feature selection methods; (3) ELOPS method prediction performance analysis; and (4) ELOPS method offline learning and online prediction time analysis.

We first selected the 2500 hotspot functions of SPEC CPU2006 for training, comparing and analyzing the search performance between the traditional GA and our AMPSO method. Then, we applied the optimization parameters obtained by the two methods to the training set and record program performance speedup. Based on the ensemble learning model, we compared the speedup when using three different FCRs to obtain the program feature subset. The prediction performance of the ELOPS method was analyzed in two aspects: precision, recall, and F1 metrics analysis and speedup analysis when predicting optimization parameters for new programs (from NPB and large scientific calculation program). Finally, we analyzed the offline learning and online prediction time of our ELOPS method.

6.3. Experimental Results

6.3.1. Comparison of GA and AMPSO Search Performance

When the GA and AMPSO were used for the program transformation parameter search, we set the control parameters first. The GA's population size is 40, the crossover and mutation probability are 0.6 and 0.3, respectively, and the maximum genetic generation was set as 50; AMPSO's population size is 20, and the maximum evolutionary generation is 30. At this time, the optimal solution of the GA appears in the 45th generation, while AMPSO appears in the 16th generation. A GA and AMPSO performance comparison is shown in Figure 6.

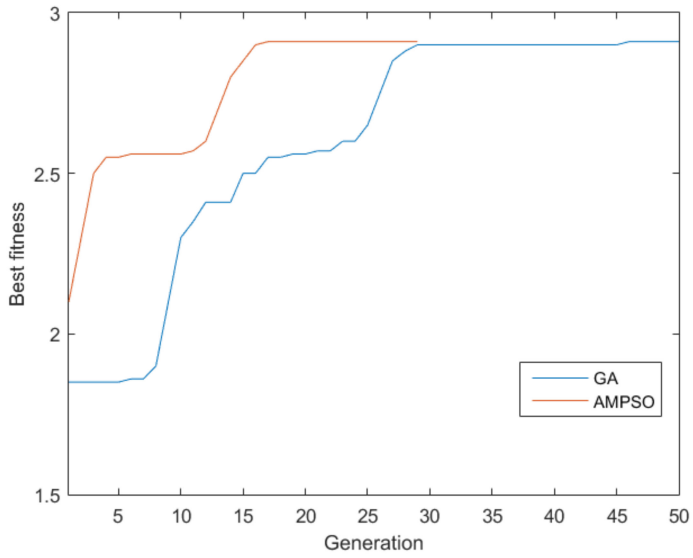

Figure 6. Overview of the stacking ensemble model.

GA and AMPSO have a lot in common: both start with random initialization of the population, using the evaluation function to measure the program optimization parameters and implementing a certain degree of random searching through the fitness value obtained by the evaluation function. However, we can see that the AMPSO algorithm searches faster and better than the GA in search performance. The following are the detailed reasons:

(1) The AMPSO algorithm does not include commonly used genetic operations such as crossover and mutation. AMPSO uses the speed information and position information of the individual in the solution space to make individual changes, which makes its computational complexity significantly smaller than that of GA.
(2) The AMPSO algorithm makes the particles have a "memory" function. The next generation of solutions can inherit more information from their ancestors through their own learning or learning from other particles. As a result, the AMPSO algorithm can select the optimum solution faster.
(3) The AMPSO algorithm has a special information sharing system. For the GA, chromosomes share information with each other, and all populations move to the optimal position at a more uniform speed. For AMPSO, the information moves in a one-way fashion, which means that only the best particles transmit messages to other particles.

When using GA and AMPSO to search the best optimization parameters of the training set, we obtain the speedup both on platform I and platform II relative to the GCC-O3 default optimization parameter settings, as shown in Table 2. The average speedup of the GA on the two platforms is 1.30 and 1.26, while the average speedup of AMPSO is 1.39 and 1.36, respectively.

We used (U, T, P) to indicate three factors, which correspond to loop unrolling, loop tiling factor and array padding. On Platform I, the default compiler optimization parameters of the *form* function in *416.gamess* were (1,0,0). When using the best optimization parameters searched by GA and AMPSO, they were (4,0,0) and (8,0,0). Consequently, the function running time was 0.60, 0.53, and 0.52, respectively. AMPSO can alter the control flow to obtain the fastest running time. When the innermost loop has function reuse, this variation can decrease the number of judgments and memory access.

Table 2. Speedup of the SPEC CPU2006 benchmark.

Benchmark	Platform I		Platform II	
	AMPSO	**GA**	**AMPSO**	**GA**
401.bzip2	1.51	1.42	1.49	1.38
429.mcf	1.48	1.35	1.49	1.34
445.gobmk	1.41	1.48	1.34	1.38
456.hmmer	1.38	1.25	1.36	1.27
462.libquantum	1.47	1.35	1.46	1.34
464.h264ref	1.38	1.21	1.36	1.17
473.astar	1.35	1.29	1.24	1.30
410.bwaves	1.46	1.37	1.35	1.29
416.gamess	1.37	1.25	1.35	1.26
433.milc	1.57	1.45	1.36	1.24
435.gromacs	1.46	1.37	1.29	1.28
437.leslie3d	1.31	1.24	1.34	1.24
444.namd	1.35	1.25	1.40	1.25
447.dealII	1.38	1.31	1.34	1.28
453.povray	1.45	1.29	1.44	1.23
454.calculix	1.47	1.35	1.48	1.26
459.GemsFDTD	1.36	1.37	1.30	1.25
470.lbm	1.08	1.11	1.12	1.13
481.wrf	1.23	1.17	1.34	1.19
482.sphinx3	1.41	1.05	1.29	1.11
Average	**1.39**	1.30	**1.36**	1.26

The experimental results showed that the GA achieved 1.30× and 1.26× speedup, while AMPSO achieved 1.39× and 1.36× speedup for the SPEC2006 benchmark on Platforms I and II. The reason why AMPSO performs better than GA has been analyzed in detail in the previous content. In some cases, AMPSO has a lower speedup than GA because it has a local rather than the global optimal solution. AMPSO's search ability is better than that of the GA in most cases, so we used the optimization parameters searched by the AMPSO algorithm as training samples for our model.

6.3.2. Comparison of Feature Selection

As mentioned earlier, we employed three approaches to obtain program feature subsets, and took the features selected as the input of the prediction part. In this situation, the prediction generated three optimization parameter categories when comparing the running time speedup of the program achieved by these three different parameter predictions with the default -O3 optimization of GCC. The experimental results are shown in Table 3, where IG, CS, and RF represent the information gain, the Chi-square value, and the ReliefF used in our framework, respectively. AMPSO means the results obtained by our optimization parameter search algorithm, which has the best performance.

As seen from Table 3, the stacking ensemble learning model has the fastest running time when using IG as an FCR to obtain the program features. The reason for the better performance of IG is that it is suitable for global feature selection and uses the statistical attributes of all samples to reduce the sensitivity to noise. Platform I and platform II can obtain the average speedups of 1.29× and 1.27×, respectively, which are 93% of the AMPSO method. The performance is close to that of AMPSO, while our prediction model can greatly save time overhead.

Table 3. Running time while using different features.

Benchmark	Platform I				Platform II			
	IG	**CS**	**RF**	**AMPSO**	**IG**	**CS**	**RF**	**AMPSO**
401.bzip2	1.31	1.19	1.15	1.51	1.35	1.34	1.21	1.49
429.mcf	1.33	1.13	1.10	1.48	1.33	1.26	1.24	1.49
445.gobmk	1.29	1.16	1.06	1.41	1.24	1.09	1.08	1.34
456.hmmer	1.29	1.25	1.27	1.38	1.18	1.16	1.17	1.29
462.libquantum	1.34	1.14	1.12	1.47	1.34	1.13	1.12	1.46
464.h264ref	1.23	1.05	1.03	1.38	1.29	1.17	1.18	1.36
473.astar	1.23	1.23	1.21	1.35	1.21	1.19	1.17	1.24
410.bwaves	1.33	1.22	1.21	1.46	1.23	1.21	1.22	1.35
416.gamess	1.25	1.14	1.14	1.37	1.26	1.14	1.12	1.35
433.milc	1.34	1.35	1.37	1.57	1.31	1.24	1.19	1.36
435.gromacs	1.34	1.23	1.22	1.46	1.29	1.27	1.27	1.36
437.leslie3d	1.23	1.19	1.17	1.31	1.26	1.16	1.14	1.34
444.namd	1.32	1.30	1.29	1.35	1.34	1.29	1.27	1.40
447.dealII	1.24	1.23	1.22	1.38	1.21	1.18	1.19	1.34
453.povray	1.36	1.23	1.21	1.45	1.29	1.23	1.21	1.44
454.calculix	1.32	1.25	1.24	1.47	1.32	1.21	1.19	1.48
459.GemsFDTD	1.31	1.29	1.31	1.36	1.27	1.25	1.23	1.30
470.lbm	1.03	1.01	1.01	1.08	1.09	1.07	1.06	1.12
481.wrf	1.35	1.29	1.18	1.23	1.28	1.09	1.10	1.34
482.sphinx3	1.33	1.09	1.08	1.41	1.21	1.15	1.13	1.29
Average	**1.29**	1.20	1.18	1.39	**1.27**	1.19	1.17	1.36

6.3.3. Prediction Performance Analysis

The performance of different machine learning models

The above experimental results indicate that the ELOPS model can effectively improve the performance of the testing programs. In the model training process, we take the program features obtained by information gain as input and the optimization parameters searched by the AMPSO algorithm as output to construct the training samples. We implement the prediction result for the SPEC CPU2006 training set with leave-one-out cross-validation (LOOCV), as shown in Table 4, in which DT, LR, BN, and NN represent decision tree, logical regression, Bayesian network, and neural network algorithms, respectively.

Table 4. Predictive results of different machine learning models.

Benchmark	Platform	Model	Precision	Recall	F1
SPEC 2006	Platform I	DT	0.482	0.257	0.335
		LR	0.549	0.209	0.303
		BN	0.528	0.259	0.359
		NN	0.546	0.269	**0.366**
	Platform II	DT	0.483	0.257	0.335
		LR	0.550	0.211	0.305
		BN	0.527	0.256	0.357
		NN	0.536	0.268	**0.365**

Table 4 shows that the LR algorithm obtained the optimum prediction accuracy, but the recall and F1 values were both the lowest. The BN and NN had relatively higher F1 values while the prediction time overhead was larger. For the DT and LR algorithms, experiments could be performed in a few minutes, while BN and NN needed several hours. The results indicate that the optimum machine learning algorithm is not exactly the same when using different evaluation indicators. For example, the precision of LR was higher than those of other algorithms, while the BN algorithm was better in terms of recall. When comparing the comprehensive evaluation indicator F1, BN and NN are better, but their

complexity is larger and the time overhead is too high, especially for programs with larger datasets and feature metrics.

Because BN and NN have the best F1, we separately added the results of these two to the training set for the following prediction. The experimental result is shown in Table 5. From the results, we can see that the prediction performance of the stacking model using the DT + NN results was similar to that of simply using the DT algorithm. The reason may be that the DT prediction accuracy is not high and some data are incorrectly classified into other classes, which influences the correct classification of the model when the wrong results are adjusted into the training data. While the BN results were added to the training data for model prediction, most of the evaluation indices showed quite good improvement. In particular, the F1 score was better than the result of the LR-only prediction, and the time cost was greatly reduced. The reason why adding BN can achieve better performance is that BN uses conditional probability to express the correlation between various information elements, and can learn and reason under limited, incomplete, and uncertain information conditions. Therefore, our ELOPS stacking ensemble prediction model uses the LR + BN model. The experimental results showed that ELOPS can improve the evaluation indicators to a certain degree and had a great performance improvement when compared with the single machine learning algorithm.

Table 5. Predictive Results of Different Machine Learning Models.

Benchmark	Platform	Model	Precision	Recall	F1
SPEC 2006	Platform I	DT + BN	0.483	0.257	0.335
		LR + BN	0.686	0.273	**0.391**
		DT + NN	0.482	0.256	0.334
		LR + NN	0.710	0.247	0.366
	Platform II	DT + BN	0.484	0.256	0.335
		LR + BN	0.691	0.272	**0.390**
		DT + NN	0.480	0.254	0.332
		LR + NN	0.711	0.245	0.364

Comparison with state-of-the-art prediction models

Further checking ELOPS performance, we contrast with the prediction accuracy of the work in [19,21]. The definition of prediction accuracy is as follows:

$$prediction\ accuracy = \frac{par_{predict}}{par_{AMPSO}}$$

where $par_{predict}$ indicates the optimal optimization parameters predicted by the different machine learning models and par_{AMPSO} represents the optimal optimization parameters searched by the AMPSO algorithm.

Because the prediction benchmarks in refs. [19] and [21] are different from ours, we used the methods in refs. [19] and [21] to optimize the parameters of the SPEC CPU2006 training set. The results are shown in Table 6, where KNN is the K nearest neighbor algorithm [19], SVM is the support vector machine method [21], ELOPS is the method proposed in this paper, and GCC represents the GCC default heuristics optimization for the optimal parameter selection.

The prediction accuracy of the ELOPS method for predicting the best parameters on both platforms was 0.71 and 0.72, respectively. The prediction accuracy corresponding to the best parameters was taken as the overall prediction accuracy of the model, because when using the model for prediction, we used the best parameters as the best compiler optimization parameter combination of the program. The prediction accuracy of SVM was 0.65 and 0.63, while the GCC default optimization parameters were only 0.15 and 0.16. The KNN prediction accuracy was slightly worse than that of SVM. The experimental results showed that ELOPS had a higher prediction accuracy. The reason ELOPS can obtain better prediction accuracy is that it uses mixed static and dynamic features to represent

the programs and an adaptive multiobjective PSO algorithm to improve optimization search efficiency. At the same time, based on the stacking ensemble learning technique, an effective prediction model for choosing program optimization parameters was constructed. In most cases, GCC default optimization cannot obtain the best optimization parameters. In addition, the experiment illustrates that the best parameter class is distributed in all parameter classes, that is, none of the parameter classes are optimal in all cases.

Table 6. Prediction accuracy on the different platforms.

	Platform I				**Platform II**			
	KNN	**SVM**	**ELOPS**	**GCC**	**KNN**	**SVM**	**ELOPS**	**GCC**
Best parameter	0.63	0.65	**0.71**	0.15	0.61	0.63	**0.72**	0.16
Second best parameter	0.63	0.64	0.65	0.15	0.59	0.63	0.65	0.26
Third best parameter	0.14	0.14	0.12	0.23	0.09	0.10	0.11	0.24
Fourth best parameter	0.08	0.04	0.01	0.23	0.07	0.06	0.08	0.17
Fifth best parameter	0.06	0.04	0.04	0.14	0.03	0.03	0.03	0.06
Sixth best parameter	0.03	0.01	0.01	0.05	0.02	0.02	0.01	0.03
Seventh best parameter	0.03	0.02	0.01	0.05	0.02	0.01	0.01	0.03

To test the predictive performance of ELOPS for new programs, we implementED the test using the NPB benchmark and five large scientific computational programs to compare the optimization performance with that of GCC default optimization. At the same time, we compareD the prediction results of the ELOPS with the methods of refs. [19,21]. The results are shown in Figures 7 and 8, where KNN stands for the method of ref. [19] and SVM represents the method of ref. [21]. ELOPS and Best represent the performance speedup that is achieved by using the methods presented in this paper and the best optimization parameters searched by the AMPSO method, respectively. In addition, we used Table 7 as a confusion matrix to represent the predicted results; some experimental results on platform I are listed, because there are many experimental results on the two platforms and the effects are similar.

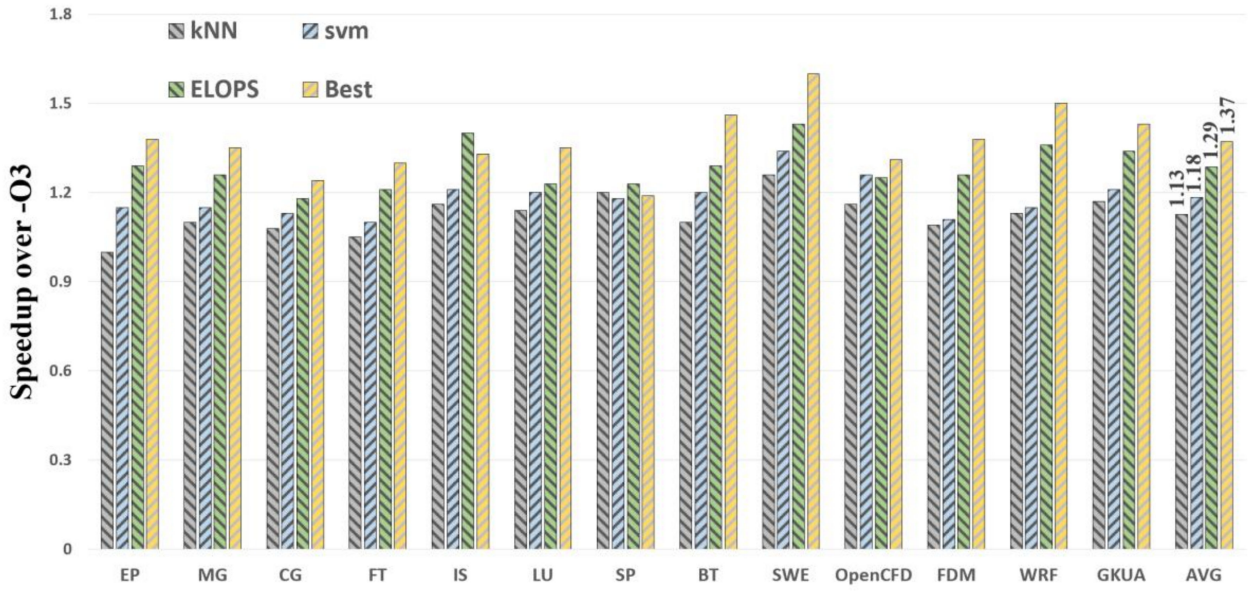

Figure 7. The speedup of the new programs when taking the ELOPS model on Platform I.

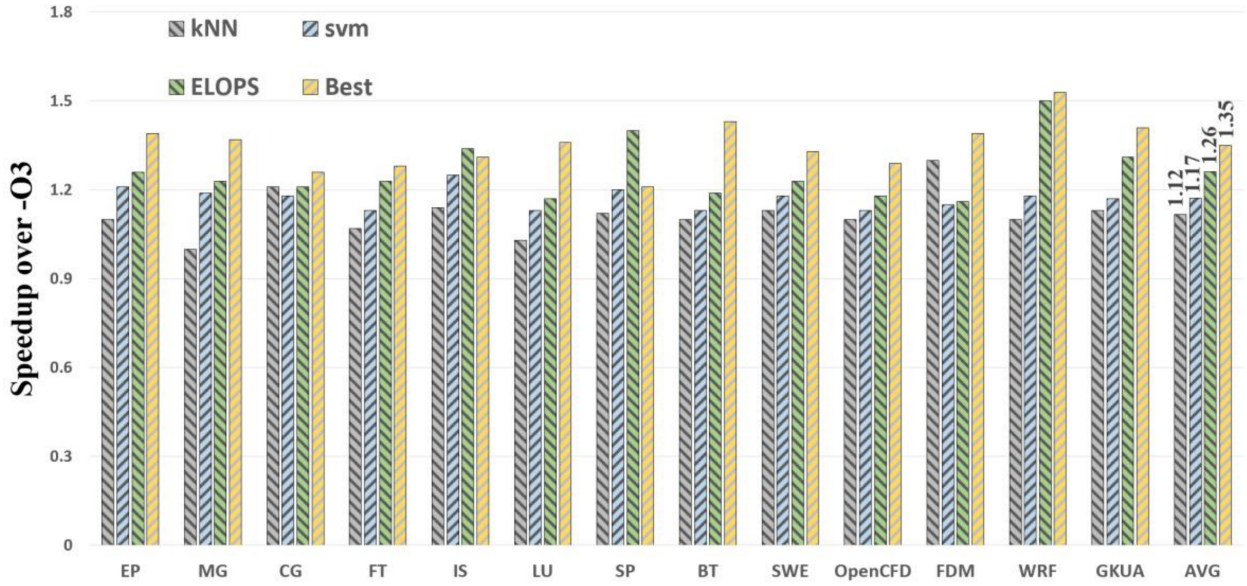

Figure 8. The speedup of the new programs when taking the ELOPS model on Platform II.

Table 7. Predicted results of time on the different platforms.

Benchmark	Model	Precision	Recall	F1	AUC
IS	KNN	0.682	0.267	0.329	0.732
	SVM	0.749	0.289	0.331	0.735
	ELOPS	0.828	0.359	0.378	0.885
SP	KNN	0.783	0.217	0.331	0.699
	SVM	0.789	0.299	0.327	0.721
	ELOPS	0.817	0.389	0.366	0.883
SWE	KNN	0.762	0.243	0.366	0.688
	SVM	0.749	0.278	0.334	0.721
	ELOPS	0.833	0.369	0.388	0.888
WRF	KNN	0.773	0.258	0.317	0.688
	SVM	0.819	0.277	0.316	0.722
	ELOPS	0.831	0.371	0.391	0.887

The experimental results showed that our ELOPS had an adequate prediction performance for the new programs. On platform I, the average speedup of the Best and ELOPS was 1.37 and 1.29, respectively, and our ELOPS was up to 94% of the Best. ELOPS had better performances of IS and SP than those of the Best, since one function's best parameter class does not depend on the others and the AMPSO search will sometimes fall into the local optimal solution. This may also be caused by some noise during data collection. The average speedup of Best on platform II was 1.35, and the average speedup of ELOPS on platform I was 1.26, which is up to 93% of Best. The reasons we performed well in the speedup and the prediction precision are our different feature selection, optimization parameter searching method, and stacking ensemble method. In the KNN model, the author uses static features and GA search algorithms to construct the perdition model. In SVM model, the author uses dynamic features. However, our method used mixed static and dynamic features, which can select the most useful features at the compiling time and the running time for each program. Furthermore, we proposed a new optimization parameter search algorithm better than GA, and we used the stacking ensemble learning prediction model, which are better than the single machine learning algorithms, such as KNN and SVM.

6.3.4. Offline Learning and Online Prediction Time

If the time is placed into different segments, the time consumed by ELOPS to obtain compiler optimization parameters for the fresh code can be partitioned into two sections: the stage of training, which is offline, and the stage of prediction, which is online. The training process should be completed once, and the time spent gathering the data for training is attributed to the program numbers. Let us take WRF as an example to illustrate; Table 8 displays the time needed for each exact stage of the two platforms.

Table 8. Prediction time on the different platforms.

	Phase	Platform I (Time)			Platform II (Time)		
		KNN	**SVM**	**ELOPS**	**KNN**	**SVM**	**ELOPS**
Offline training	Data collection	11 d	11 d	12 d	11 d	12 d	13 d
	Model construction	57 s	55 s	69 s	63 s	70 s	76 s
Online prediction	Feature extraction	11 s	15 s	12 s	12 s	16 s	14 s
	Model prediction	0.79 s	0.84 s	**0.36 s**	0.98 s	0.97 s	**0.45 s**

Experimental results demonstrated that ELOPS can finish the prediction for the fresh target programs in a rather low-time-cost manner. The reason why our model had a shorter prediction time is that the Stacking integrated learning model can overcome the defects of a single model, optimize the input of the meta learner, improve the prediction effect, and shorten prediction time. At the same time, it is obvious to see that the ELOPS works better than both the KNN and the SVM approaches from the aspects of the accuracy of prediction and program performance through the explanation in the previous paragraph. Therefore, the ELOPS approach can surpass some of the existing means with respect to the prediction accuracy, the time consumed on it, and the true performance of the program.

7. Conclusions

In this paper, we proposed a compiler optimization parameter selection model, ELOPS, which can automatically generate compiler optimization parameters for different programs. The model takes the program features as its input, and the approximate optimal compiler optimization parameter as its output. In the training data construction stage of the ELOPS model, an improved PSO algorithm for compiler optimization parameter space search was proposed, which can resolve the constrained multiobjective optimization more efficiently. Compared with the default standard optimization level -O3 of GCC, we obtained 1.39× and 1.36× speedup on two platforms. The improved PSO is a mechanical search of the compiler optimization space, which can obtain more accurate sample data in the model construction stage. We proposed a program feature representation method, FCR, which uses the mixed static and dynamic features to represent the programs. Through the search of compiler optimal parameters and the new method of program feature representation, we can provide more and better sample parameters for the construction of the compiler optimization prediction model. On this basis, we used Stacking ensemble learning to establish the statistical model and integrated into the compiler framework, then the optimization parameter selection decision and the compilation process were implemented. Given the new target program, the ELOPS model can automatically extract the features of the new program first and then predict the best compile optimization parameters based on the knowledge stored in the model. Relative to the compiler default standard optimization level -O3, the ELOPS method realized 1.29× and 1.26× program execution time averaging speedup on two platforms, respectively, which are better than results of the existing method.

When compiling the program, it needs to go through a large number of optimization phases to obtain more efficient object code. The current compiler implements relatively fixed optimization phases; however, the research shows that programs with different features should have various optimization phases. Our current work mainly solves the prediction of compilation optimization parameters. In future work, we will study applying

our method to other compilers for compiling and running programs, selecting of compiler optimization phases, and enlarging the benchmarks to more various application scopes.

Author Contributions: H.L. designed the analysis, designed the research experiment, wrote the original and revised manuscript, and conducted data analysis and details of the work. J.X. verified data and conducted statistical analysis. S.C. collected the data and conducted the analysis. T.G. verified image data analysis, and guided direction of the work. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by National key research and development plan "high performance computing" key project:" E-level computer key technology verification system" under Grant 2021C5406. This work was also supported by "The Doctoral Research Start-up Fee Support Project of Henan Normal University" under Grant 201911.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Benacchio, T.; Bonaventura, L.; Altenbernd, M.; Cantwell, C.D.; Düben, P.D.; Gillard, M.; Giraud, L.; Göddeke, D.; Raffin, E.; Teranishi, K.; et al. Resilience and fault tolerance in high-performance computing for numerical weather and climate prediction. *Int. J. High Perform. Comput. Appl.* **2021**, *35*, 285–311. [CrossRef]
2. Gómez-Gónzalez, E.; Núñez, N.O.; Caro, C.; García-Martín, M.L.; Fernández-Afonso, Y.; de la Fuente, J.M.; Balcerzyk, M.; Ocaña, M. Dysprosium and Holmium Vanadate Nanoprobes as High-Performance Contrast Agents for High-Field Magnetic Resonance and Computed Tomography Imaging. *Inorg. Chem.* **2021**, *60*, 152–160. [CrossRef] [PubMed]
3. Vermaas, J.V.; Sedova, A.; Baker, M.B.; Boehm, S.; Rogers, D.M.; Larkin, J.; Glaser, J.; Smith, M.D.; Hernandez, O.; Smith, J.C. Supercomputing pipelines search for therapeutics against COVID-19. *Comput. Sci. Eng.* **2021**, *23*, 7–16. [CrossRef]
4. Chetverushkin, B.N.; Olkhovskaya, O.G.; Tsigvintsev, I.P. Numerical solution of high-temperature gas dynamics problems on high-performance computing systems. *J. Comput. Appl. Math.* **2021**, *390*, 113374. [CrossRef]
5. Chen, Y.; Pan, F.; Holzer, J.; Rothberg, E.; Ma, Y.; Veeramany, A. A High Performance Computing Based Market Economics Driven Neighborhood Search and Polishing Algorithm for Security Constrained Unit Commitment. *IEEE Trans. Power Syst.* **2021**, *36*, 292–302. [CrossRef]
6. Xu, L.; Lux, T.; Chang, T.; Li, B.; Hong, Y.; Watson, L.; Butt, A.; Yao, D.; Cameron, K. Prediction of high-performance computing input/output variability and its application to optimization for system configurations. *Equal. Eng.* **2021**, *33*, 318–334. [CrossRef]
7. Asri, M.; Malhotra, D.; Wang, J.; Biros, G.; John, L.K.; Gerstlauer, A. Hardware accelerator integration tradeoffs for high-performance computing: A case study of GEMM acceleration in N-Body methods. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 2035–2048. [CrossRef]
8. Santana, A.; Freitas, V.; Castro, M.; Pilla, L.L.; Méhaut, J.F. ARTful: A model for user-defined schedulers targeting multiple high-performance computing runtime systems. *Softw. Pract. Exp.* **2021**, *51*, 1622–1638. [CrossRef]
9. Kumar, V.; Sharma, D.K.; Mishra, V.K. Cheval M: A GPU-based in-memory high-performance computing framework for accelerated processing of big-data streams. *J. Supercomput.* **2021**, *77*, 6936–6960. [CrossRef]
10. Gai, K.; Qin, X.; Zhu, L. An energy-aware high performance task allocation strategy in heterogeneous fog computing environments. *IEEE Trans. Comput.* **2021**, *70*, 626–639. [CrossRef]
11. Gill, A.; Lalith, M.; Poledna, S.; Hori, M.; Fujita, K.; Ichimura, T. High-Performance Computing Implementations of Agent-Based Economic Models for Realizing 1:1 Scale Simulations of Large Economies. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 2101–2114. [CrossRef]
12. Yang, Q.; Yang, H.; Lv, D.; Yu, R.; Li, E.; He, L.; Chen, Q.; Chen, H.; Guo, T. High-Performance Organic Synaptic Transistors with an Ultrathin Active Layer for Neuromorphic Computing. *ACS Appl. Mater. Interfaces* **2021**, *13*, 8672–8681. [CrossRef]
13. Li, J.; Zhang, X.; Han, L.; Ji, Z.; Dong, X.; Hu, C. OKCM: Improving parallel task scheduling in high-performance computing systems using online learning. *J. Supercomput.* **2021**, *77*, 5960–5983. [CrossRef]
14. Mohammed, A.; Eleliemy, A.; Ciorba, F.M.; Kasielke, F.; Banicescu, I. An approach for realistically simulating the performance of scientific applications on high performance computing systems. *Future Gener. Comput. Syst.* **2020**, *111*, 617–633. [CrossRef]
15. Morales-Hernández, M.; Sharif, M.B.; Gangrade, S.; Dullo, T.T.; Kao, S.-C.; Kalyanapu, A.; Ghafoor, S.K.; Evans, K.J.; Madadi-Kandjani, E.; Hodges, B.R. High-performance computing in water resources hydrodynamics. *J. Hydroinform.* **2020**, *22*, 1217–1235. [CrossRef]

16. Ogilvie, W.F.; Petoumenos, P.; Wang, Z.; Leather, H. Minimizing the cost of iterative compilation with active learning. In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization, Austin, TX, USA, 4–8 February 2017; pp. 245–256.
17. Zhao, J.; Li, Y.Y.; Zhao, R.C. "Black magic" of polyhedral compilation. *J. Softw.* **2018**, *29*, 2371–2396.
18. Wang, Z.; O'Boyle, M. Machine Learning in Compiler Optimization. *Proc. IEEE* **2018**, *106*, 1879–1901. [CrossRef]
19. Agakov, F.; Bonilla, E.; Cavazos, J.; Franke, B.; Fursin, G.; O'Boyle, M.F.P.; Thomson, J.; Toussaint, M.; Williams, C.K.I. Using Machine Learning to Focus Iterative Optimization. In Proceedings of the International Symposium on Code Generation and Optimization, New York, NY, USA, 26–29 March 2006; pp. 295–305.
20. Colucci, A.; Juhasz, D.; Mosbeck, M.; Marchisio, A.; Rehman, S.; Kreutzer, M.; Nadbath, G.; Jantsch, A.; Shafique, M. MLComp: A Methodology for Machine Learning-based Performance Estimation and Adaptive Selection of Pareto-Optimal Compiler Optimization Sequences. In Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference, Grenoble, France, 1–5 February 2021; pp. 108–113.
21. Park, E.; Cavazos, J.; Pouchet, L.-N.; Bastoul, C.; Cohen, A.; Sadayappan, P. Predictive Modeling in a Polyhedral Optimization Space. *Int. J. Parallel Program.* **2013**, *41*, 704–750. [CrossRef]
22. Pekhimenko, G.; Brown, A.D. Efficient program compilation through machine learning techniques. In *Software Automatic Tuning*; Springer: New York, NY, USA, 2011.
23. Kulkarni, S.; Cavazos, J. Mitigating the Compiler Optimization Phase-ordering Problem using Machine Learning. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, Tucson, AZ, USA, 19–26 October 2012; pp. 147–162.
24. Liu, H.; Zhao, R.; Nie, K. Using Ensemble Learning to Improve Automatic Vectorization of Tensor Contraction Program. *IEEE Access* **2018**, *6*, 47112–47124. [CrossRef]
25. Mohammed, M.; Mwambi, H.; Mboya, I.B.; Elbashir, M.K.; Omolo, B. A stacking ensemble deep learning approach to cancer type classification based on TCGA data. *Sci. Rep.* **2021**, *11*, 15626. [CrossRef]
26. Park, E.; Kulkarni, S.; Cavazos, J. An Evaluation of Different Modeling Techniques for Iterative Compilation. In Proceedings of the 14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, Taipei, Taiwan, 9–14 October 2011; pp. 65–74.
27. Yuki, T.; Renganarayanan, L.; Rajopadhye, S.; Anderson, C.; Eichenberger, A.E.; O'Brien, K. Automatic Creation of Tile Size Selection Models. In Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Toronto, ON, Canada, 24–28 April 2010; pp. 190–199.
28. Schkufza, E.; Sharma, R.; Aiken, A. Stochastic Optimization of Floating-point Programs with Tunable Precision. *ACM SIGPLAN Not.* **2014**, *49*, 53–64. [CrossRef]
29. Purini, S.; Jain, L. Finding Good Optimization Sequences Covering Program Space. *ACM Trans. Archit. Code Optim.* **2013**, *9*, 1–23. [CrossRef]
30. Nie, K.; Zhou, Q.; Qian, H.; Pang, J.; Xu, J.; Li, Y. Parallel Region Reconstruction Technique for Sunway High-Performance Multi-core Processors. In Proceedings of the 7th International Conference of Pioneering Computer Scientists, Engineers and Educators (ICPCSEE), Taiyuan, China, 17–20 September 2021; pp. 163–179.
31. Ashouri, A.H.; Killian, W.; Cavazos, J.; Palermo, G.; Silvano, C. A Survey on Compiler Autotuning using Machine Learning. *ACM Comput. Surv.* **2019**, *51*, 1–42. [CrossRef]
32. Ashouri, A.H.; Palermo, G.; Cavazos, J.; Silvano, C. Automatic Tuning of Compilers Using Machine Learning. In *Springer Briefs in Applied Sciences and Technology*; Springer: Cham, Switzerland, 2018.
33. Kong, J.; Nie, K.; Zhou, Q.; Xu, J.; Han, L. Thread Private Variable Access Optimization Technique for Sunway High-Performance Multi-core Processors. In Proceedings of the 7th International Conference of Pioneering Computer Scientists, Engineers and Educators (ICPCSEE), Taiyuan, China, 17–20 September 2021; pp. 180–189.
34. Pouchet, L.-N.; Bastoul, C.; Cohen, A.; Vasilache, N. Iterative Optimization in the Polyhedral Model: Part I, One-dimensional Time. In Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, USA, 11–14 March 2007; pp. 144–156.
35. Nobre, R.; Martins, L.G.A.; Cardoso, J.M.P. Use of Previously Acquired Positioning of Optimizations for Phase Ordering Exploration. In Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems, Sankt Goar, Germany, 1–3 June 2015; pp. 58–67.
36. Fang, S.; Xu, W.; Chen, Y.; Eeckhout, L.; Temam, O.; Chen, Y.; Wu, C.; Feng, X. Practical Iterative Optimization for the Data Center. *ACM Trans. Archit. Code Optim.* **2015**, *12*, 49–60. [CrossRef]
37. Fursin, G.; Kashnikov, Y.; Memon, A.W.; Chamski, Z.; Temam, O.; Namolaru, M.; Yom-Tov, E.; Mendelson, B.; Zaks, A.; Courtois, E.; et al. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *Int. J. Parallel Program.* **2011**, *39*, 296–327. [CrossRef]
38. Fursin, G.; Miranda, C.; Temam, O.; Namolaru, M.; Yom-Tov, E.; Zaks, A.; Mendelson, B.; Bonilla, E.; Thomson, J.; Leather, H.; et al. *Milepost GCC: Machine Learning Based Research Compiler*; GCC Summit: Ottawa, ON, Canada, 2008; pp. 1–13.
39. Ashouri, A.H.; Bignoli, A.; Palermo, G.; Silvano, C.; Kulkarni, S.; Cavazos, J. MiCOMP: Mitigating the Compiler Phase-Ordering Problem using Optimization Sub-Sequences and Machine Learning. *ACM Trans. Archit. Code Optim.* **2017**, *14*, 1–28. [CrossRef]
40. Wang, Z.; Tournavitis, G.; Franke, B.; O'Boyle, M.F.P. Integrating Profile-driven Parallelism Detection and Machine-learning-based Mapping. *ACM Trans. Archit. Code Optim.* **2014**, *11*, 1–26. [CrossRef]

41. Cummins, C.; Petoumenos, P.; Wang, Z.; Leather, H. Synthesizing Benchmarks for Predictive Modeling. In Proceedings of the International Symposium on Code Generation and Optimization, Austin, TX, USA, 4–8 February 2017; pp. 86–99.
42. Ashouri, A.H.; Mariani, G.; Palermo, G.; Park, E.; Cavazos, J.; Silvano, C. COBAYN: Compiler Autotuning Framework using Bayesian Networks. *ACM Trans. Archit. Code Optim.* **2016**, *13*, 1–25. [CrossRef]
43. Leather, H.; Bonilla, E.; O'boyle, M. Automatic Feature Generation for Machine Learning-based Optimizing Compilation. In Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Seattle, WA, USA, 22–25 March 2009; pp. 81–91.
44. Ding, Y.; Ansel, J.; Veeramachaneni, K.; Shen, X.; O'Reilly, U.-M.; Amarasinghe, S. Autotuning Algorithmic Choice for Input Sensitivity. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, 13–17 June 2015; pp. 379–390.
45. 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 Trans. Archit. Code Optim.* **2016**, *13*, 1–28. [CrossRef]
46. Kulkarni, P.A.; Whalley, D.B.; Tyson, G.S.; Davidson, J.W. Practical Exhaustive Optimization Phase Order Exploration and Evaluation. *ACM Trans. Archit. Code Optim.* **2009**, *6*, 1–36. [CrossRef]
47. Ballal, P.A.; Sarojadevi, H.; Harsha, P.S. Compiler Optimization: A Genetic Algorithm Approach. *Int. J. Comput. Appl.* **2015**, *112*, 9–13.
48. Kumar, T.S.; Sakthivel, S.; Kumar, S.S. Optimizing Code by Selecting Compiler Flags using Parallel Genetic Algorithm on Multicore CPUs. *Int. J. Eng. Technol.* **2014**, *6*, 544–551.
49. Li, Y.Y.; Zhao, J.; Pang, J.M. Split tiling design and implementation in the polyhedral model. *Chin. J. Comput.* **2020**, *43*, 1038–1051.
50. Torres-Madroñero, J.L.; Nieto-Londoño, C.; Sierra-Perez, J. Hybrid Energy Systems Sizing for the Colombian Context: A Genetic Algorithm and Particle Swarm Optimization Approach. *Energies* **2020**, *13*, 5648. [CrossRef]
51. Cao, Y.; Fan, X.; Guo, Y.; Li, S.; Huang, H. Multiobjective optimization of injection-molded plastic parts using entropy weight, random forest, and genetic algorithm methods. *J. Polym. Eng.* **2020**, *40*, 360–371. [CrossRef]
52. Nie, K.; Zhou, Q.; Qian, H.; Pang, J.; Xu, J.; Li, X. Loop selection for multilevel nested loops using a genetic algorithm. *Math. Probl. Eng.* **2021**, *2021*, 6643604. [CrossRef]
53. Al-Janabi, S.; Alkaim, A. A novel optimization algorithm (Lion-AYAD) to find optimal DNA protein synthesis. *Egypt. Inform. J.* **2022**, *23*, 271–290. [CrossRef]
54. Al-Janabi, S.; Mohammad, M.; Al-Sultan, A. A new method for prediction of air pollution based on intelligent computation. *Soft Comput.* **2020**, *24*, 661–680. [CrossRef]
55. Al-Janabi, S.; Alkaim, A.F.; Adel, Z. An Innovative synthesis of deep learning techniques (DCapsNet & DCOM) for generation electrical renewable energy from wind energy. *Soft Comput.* **2020**, *24*, 10943–10962.
56. Liu, H.; Zhao, R.; Wang, Q.; Li, Y. ALIC: A Low Overhead Compiler Optimization Prediction Model. *Wirel. Pers. Commun.* **2018**, *103*, 809–829. [CrossRef]
57. SPEC CPU2006: SPEC CPU2006 Benchmark Suite. Available online: http://www.spec.org/cpu/ (accessed on 12 October 2021).
58. Bailey, D.H.; Barszcz, E.; Barton, J.T.; Browning, D.S.; Carter, R.L.; Dagum, D.; Fatoohi, R.A.; Frederickson, P.O.; Lasinski, T.A.; Schreiber, R.S.; et al. The NAS Parallel Benchmarks. *Int. J. Supercomput. Appl.* **1991**, *5*, 63–73. [CrossRef]