Computing and Informatics, Vol. 39, 2020, 1117–1147, doi: 10.31577/cai_2020_6_1117

FINDING EFFECTIVE COMPILER OPTIMIZATION SEQUENCES: A HYBRID APPROACH

Nilton Luiz Queiroz Junior, Anderson Faustino da Silva

Departament of Informatics
State University of Maringá
Maringá, Paraná, Brazil
e-mail: niltonlqjr@gmail.com, anderson@din.uem.br

Luis Gustavo Araujo Rodriguez

Institute of Mathematics and Statistics
University of São Paulo
São Paulo, São Paulo, Brazil
e-mail: luisgar1990@gmail.com

Abstract. The Optimization Selection Problem is widely known in computer science for its complexity and importance. Several approaches based on machine learning and iterative compilation have been proposed to mitigate this problem. Although these approaches provide several advantages, they have disadvantages that can hinder the performance. This paper proposes a hybrid approach that combines the best of machine learning and iterative compilation. Several experiments were performed using different strategies, metrics and hardware platforms. A thorough analysis of the results reveals that the hybrid approach is a considerable improvement over machine learning and iterative compilation. In addition, the hybrid approach outperforms the best compiler optimization level of LLVM.

Keywords: Compilers, optimization, optimization selection problem, iterative compilation, machine learning

Mathematics Subject Classification 2010: 68-N20

1 INTRODUCTION

Compilers are computer programs capable of transforming code written in a source language into a target language [1, 19, 20]. The final product is an equivalent program generated into an executable file.

This process is divided into several phases, and one of the most important is the *optimization phase*. This stage is fundamental because it improves the quality of the final executable program, reducing run-time, code size or even power consumption [1, 4, 10].

An important aspect of compilers is their ability to provide optimizations [13]. However, two optimizations (one after the other) can provide greater benefits. For example, *copy propagation* can generate *dead code*, which is *useless* code that will not be used in the future and does not affect program results. This type of code is removed by a compiler optimization called *Dead-code elimination*, and thus improves the performance.

The previous example provides an insight into how optimizations interact with each other. Based on these interactions, modern compilers (GCC [22], ICC [23], LLVM [24]) offer standard optimization levels (O1, O2, O3), which can be used to optimize the source code. However, the performance achieved by the aforementioned levels is different for each program. This is because optimization selection depends on program features. In addition, the most effective optimizations depend on the system architecture and input, however the latter is usually different and thus its effects are ignored.

The Optimization Selection Problem (OSP) consists in choosing the most effective optimizations for a given program. It is worth mentioning that this type of problem is classified as undecidable. This is because of the search-space size, which is related to the quantity of optimizations provided by the compiler and its possible combinations. Thus, mitigating this problem is highly desirable. There are several mitigating techniques for the OSP, and the most common are the following:

Selecting an optimization set: Optimizations are selected for a given program without considering their order of application. This approach is used frequently because compiler systems such as GCC and HotSpot VM cannot reorder optimizations based on the complexity of the intermediate code, which create dependencies between optimizations [6].

Selecting an optimization sequence: Optimization sequences are selected and their order of application is considered. Sequence selection considers whether or not to repeat optimizations within a sequence.

Parameterization of optimizations: Attempts to find the most effective combination of parameters for optimizations.

In general, researchers investigate only one approach to mitigate the OSP. This paper proposes to select an optimization sequence adopting automatic schemes. Our

system either creates sequences for programs using Iterative Compilation (IC) [26]; or selects sequences from a knowledge base using Machine Learning (ML) [10, 15].

IC consists in evaluating the quality of the target code generated by different sequences, and therefore returns the most effective target code. However, ML based approaches attempt to predict sequences, from previously-successful compilations, that will have good performance in new programs.

ML has higher usage than IC based approaches because of its lower response time, which is spent mostly on the training phase. In this stage, it is necessary to evaluate the performance of several different sequences with example programs. Thus, the exact program is compiled and executed several times.

In this context, it is vital to characterize programs. Thus, one of the most difficult tasks is choosing a set of features that can effectively represent a program. Certain studies in the literature have shown that extracting features through control or data flow graphs are strategies that achieve good results; consequently, surpassing features extracted directly from the source code [11, 15]. Research studies also indicate that applying IC for each program function in Just-in-Time (JIT) environments yields better results than characterizing the function and predicting which sequence to use [6].

This paper describes a hybrid approach, already implemented on [8], that combines the best of IC and ML in order to mitigate the OSP. The objective is to describe an approach that initially uses ML to select potential optimization sequences. This is done considering the features of an unknown program. Afterwards, it applies IC to adapt potential optimization sequences to the said program. Thus, it is expected that performance will improve by adapting the solution rather than only using potential sequences.

Furthermore, the hybrid approach applies a learning scheme for recently-compiled programs. This is done using a Genetic Algorithm (GA) that creates new sequences, and thus the explored portion of the search space will always have these types of sequences. Therefore, these new sequences can be used for recent programs either after: feeding the knowledge base; compiler processing, or finding sequences for batch programs. In addition, this paper also includes the analysis, and propose a new approach to select the initial solution for the GA. It is called Centralized Sequence Selection, that choses all sequence from the most similar program.

The main contributions of this paper are as follows:

- a comparison between two different strategies to select the initial solution;
- different approaches for feeding a knowledge database; and
- an analysis of the performance impact of the approach with different hardware platforms and input sets.

The results indicate that the proposed hybrid approach outperforms both IC and ML. Furthermore, the average speedup achieved by the hybrid approach is superior to the best compiler optimization level of LLVM.

The rest of the paper is organized as follows. Section 2 introduces the hybrid approach for mitigating the OSP. Section 3 presents the instantiation of the hybrid approach. Section 4 describes the experimental environment and setup. Section 5 presents a discussion of the results, comparing them with other approaches proposed in the literature. Section 6 presents related works. Finally, Section 7 provides the conclusions of this paper.

2 A HYBRID SOLUTION FOR MITIGATING THE OSP

IC is an appealing option because it achieves better results than ML. However, ML is also interesting because it applies strategies capable of accelerating the convergence to a good solution. Thus, this paper describes a hybrid approach for solving the OSP. The objective is to combine the best of both IC and ML. The proposed hybrid approach can be described as follows.

2.1 Overview of the Hybrid Approach

Suppose there exists a training set, containing S good optimization sequences for P programs with their features. First, the approach creates a model based on the knowledge database (KD), which is used to predict good optimization sequences for a particular test program. Second, the approach selects N potential optimization sequences, using the created model, for the test program. Afterwards, the N sequences will feed a solution adapter, which utilizes a strategy based on IC to adapt (improve) the solution found in the ML phase. Finally, the best target code, found by the adapter, is returned to the user and the KD is updated with recent knowledge.

Although the system has the capacity of providing itself with feedback, an initial KD is necessary. In addition, a database generator is used just once, and thus the initial knowledge is built with both sequences and performances for some programs. Therefore, the system has the capacity to generate new sequences and provide itself with a feedback.

The system increments its KD as new programs are compiled. However, not all sequences will be possible candidates for the compilation of new programs. This occurs because the database is filtered, and thus poor performing sequences are discarded. It is important to highlight that the architecture is flexible enough to allow for a change of focus in terms of performance improvement. Therefore, it is necessary to store/record values in the KD.

The proposed hybrid approach is shown in Figure 1, which will be described more specifically in the following subsections.

As shown in Figure 1, the components of the hybrid architecture can be divided into 3 main groups:

1. Group of training components;
2. Group of prediction components; and

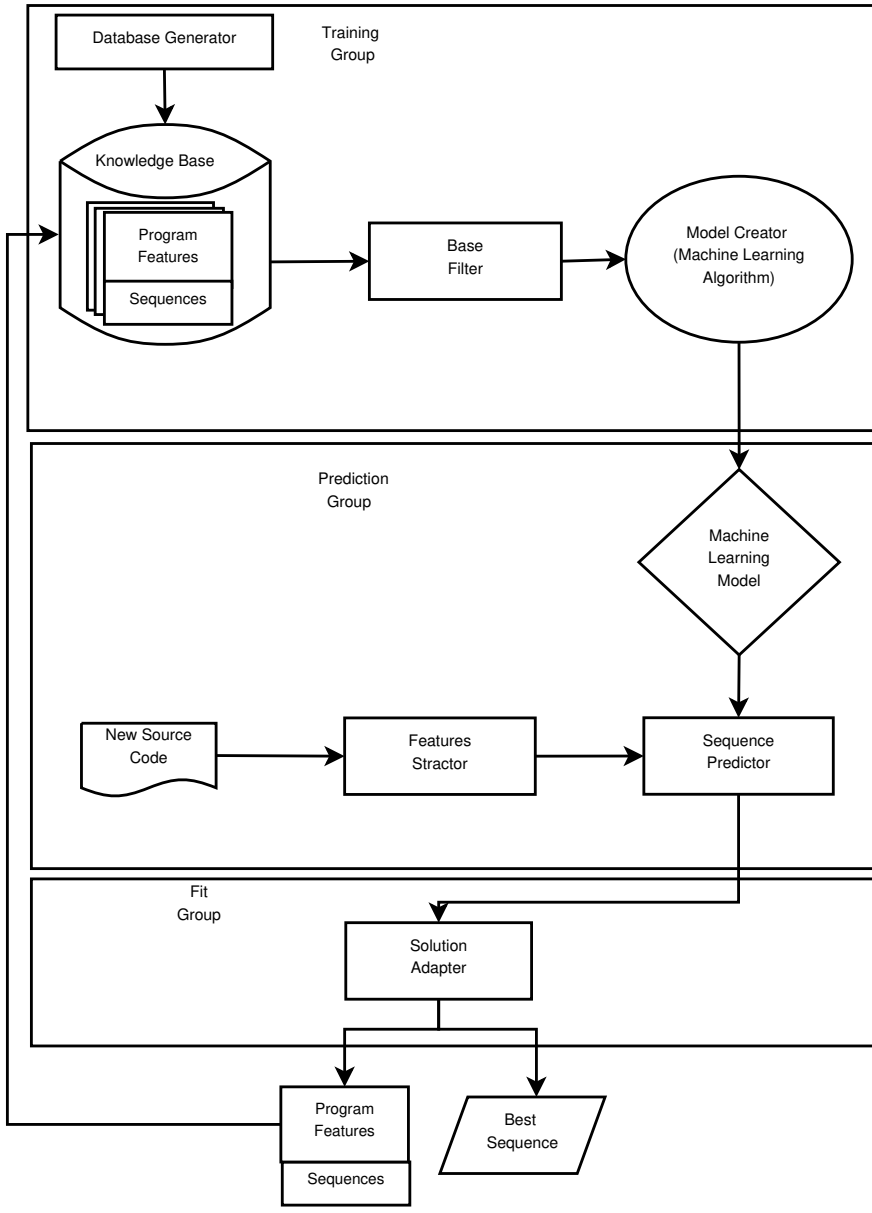

Figure 1. Architecture of our hybrid approach

3. Group of adaptive components.

These components will be described in Sections 2.2, 2.3 and 2.4. In addition, the approaches used to feed the knowledge database will be discussed in Section 2.5.

2.2 Group of Training Components

This group is comprised of components that handle the *background* process of the system. The aforementioned group has a component called the *database generator*, which is executed just once to create the KD. It is also comprised of a *database filter* and generative models in the context of ML, which the latter is used for prediction purposes.

Database Generator is an algorithm used for creating a knowledge database, which is usually created randomly [10, 2, 15]. However, there are several strategies for generating the knowledge database [7, 18]. The proposed hybrid approach uses a GA, which will be discussed in Section 3.1.1. The database generator will be used just once because of two reasons. First, it is a large IC process. Second, its objective is to generate a large database for initial programs, which will be compiled on the system.

Database Filter. The hybrid architecture considers similar programs, and thus chooses sequences that will compose the initial population of the GA. The low-performing sequences, in relation to the evaluation criteria, will be excluded from the model creation phase. However, these sequences will remain in the KD. The *database filter* is the component that executes these tasks. It discards bad sequences in terms of the performance goal, and associates the remaining sequences with the program features. It is consistently executed in the KD before the ML model is generated.

Generative Models in the Context of ML. This component collects program information given by the database filter. Afterwards, it creates the ML model using this information. This model will be utilized by the prediction components. Therefore, the parameters of the ML algorithm are defined in this phase. Subsequently, the model is created using sequences and information provided by the database filter. It is important to highlight that several ML algorithms can be used to create the model. The only restriction is that the algorithm must have a training and testing phase. In addition, the algorithm must be able to provide a classification based on a ranking scheme.

2.3 Group of Prediction Components

This group is comprised of components that predict sequences of the initial population. These components consist of a single feature extractor and sequence prediction scheme.

Feature Extractor. This component receives the source code, and afterwards extracts the features used in the prediction scheme. The extracted features are the same as those stored in the KD, and they will feed the KD using sequences generated in the solution adaptation stage. These features are also used for feeding purposes in the sequence prediction scheme.

Sequence Prediction Scheme. This module selects sequences that will compose the initial solution for the solution adapter. It receives a prediction model, created by the generative model, and features extracted from the test program. The selected sequences are chosen based on the similarity between program features in the KD and testing phase. The sequences can be chosen either from a single program or several programs.

2.4 Group of Adaptive Components

This group is comprised of just one component, which adapts the solution to the new program. This component is called the *solution adapter*.

Solution Adapter generates the final solution. Furthermore, this component creates solutions and feeds the knowledge database; in other words, it deeply explores the search space of the OSP. The initial solution of the algorithm is comprised of K sequences, which are selected by the sequence prediction scheme. Afterwards, the algorithm chosen to adapt the solution runs over these sequences. This algorithm must be capable of receiving and improving at least one sequence. The database generator stores all generated sequences in the KD. However, only the most effective sequence is considered the final solution, and thus it is given to the test program.

2.5 Approaches for Feeding the Knowledge Database

Although the hybrid approach is flexible enough to operate without feeding the database, a scheme for such a process generates knowledge for medium and long-term goals. Thus, the architecture of the hybrid approach allows the user to choose when the feedback will occur. Thus, two approaches are proposed for feeding the KD:

Constant Feeding: The hybrid approach stores new information in the KD (features and sequences) and recreates the model. This process is done for all compiled programs.

Batch Feeding: The hybrid approach stores new information in the KD, and recreates the model after K compiled programs. This is done for every compiled program.

These two approaches have different execution frequencies for creating the ML model.

The ML model based on the constant-feeding approach is created after a solution is found. Although, it has a higher cost than batch feeding, it has an intense feedback. This feeding mechanism produces a more non-deterministic approach than batch feeding. This result is produced because there are $N!$ forms of organizing N programs. Thus, altering the order of the programs can modify the initial population of the subsequent program; consequently, producing a different result.

However, batch feeding stores information and generated sequences in the database, and the model is recreated after K programs are compiled. The non-determinism of this feeding approach is less than constant feeding because altering the order of the programs that are in the same batch will not modify the initial population of the subsequent program. It is worth noting that constant feeding is basically a version of batch feeding with $K = 1$.

3 INSTANTIATION OF THE HYBRID APPROACH

The hybrid approach, described in the previous section, can be instantiated using different strategies. Thus, this section describes how it was implemented. The proposed strategy was implemented as a tool of LLVM [9], which was chosen because it allows full control over optimizations.

This means that it is possible to enable a list of optimizations through the command line. The position of each optimization indicates its order of application.

The infrastructure implemented can be viewed in Figure 2.

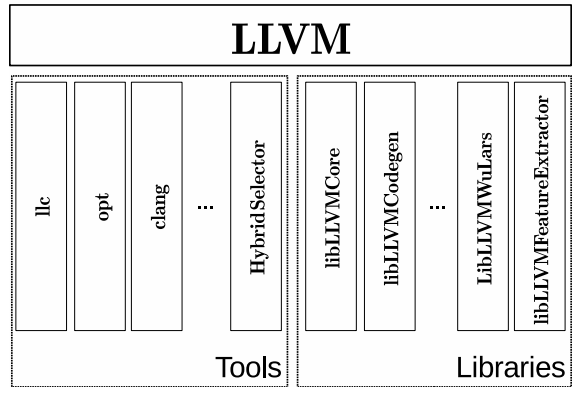

Figure 2. Infrastructure of the instantiations

Two plugins were implemented for LLVM:

1. libWuLars: used for extracting the hottest function of the program, as proposed by Wu and Larus (1994) [25]; and
2. libFeaturesExtractor: used for extracting features proposed by Namolaru et al. [14].

The remaining libraries (also known as *libs*) are standard for LLVM. In addition, they assist with developing plugins for LLVM.

The tools are provided by LLVM for compiling and optimizing code among other functions. Furthermore, a tool called HybridSelector is also used, which helps implement the proposed hybrid approach in Figure 1.

HybridSelector uses *Support Vector Machine* (SVM) as an ML algorithm. In addition, all IC phases are done using GAs. Finally, all features are extracted without the need to execute the program.

The following sections will present in detail the implementation of the HybridSelector tool. In addition, these sections will describe each component and the different parameterizations used for the experiments.

3.1 Iterative Compilation

IC is used in two phases of the hybrid approach. Its purpose is to create the KD and adapt the solution. Although the parameters are modified, the database generator and solution adapter use the same GA for each instantiation.

3.1.1 Database Generator

The algorithm 1 presents a pseudocode to the KD generation.

The GA presented in Algorithm 1 was used to create the database. This algorithm is executed with two distinct parametrization. Although these parameters are similar, they have two differences:

1. total number of individuals in a population;
2. and a stop criterion.

Both executions use *tournament selection*, and possess crossover and mutation operators. Although there are four mutation operators in the algorithm, only one can be applied to each individual. Thus, for every individual that mutation are applied, only one of four is chosen.

The mutation operators consist of:

1. substituting a randomly-positioned optimization for any other valid optimization;
2. permutation-based procedures for two optimizations that compose the sequence;
3. including a randomly-selected optimization into the sequence; and
4. excluding a randomly-positioned optimization.

Crossover takes a portion (specifically half) of each solution and concatenates them. The probability of applying mutation and crossover operators are 40 % and 60 %, respectively, as specified in [12]. In addition, this paper proposes an elitism-based algorithm, and thus the best/most effective solution is carried out to the

```
Data: TrainPrograms, MaxSeqSize, NumIndividuals, BaselineList
Result: KD
KD ← ∅;
foreach Program ∈ TrainPrograms do
    Generation ← ∅;
    NumGeneration ← 0;
    Sequences ← ∅;
    Size ← Random(1, MaxSeqSize);
    ProgramFeatures ← getFeaturesByFunction(Program);
    // Create a dictionary with functions names as keys and
       functions features as values
    Baselines ← GetBaseilnes(Program, BaselineList) // Compile with
       Compiler Baselines in BaselineList and get its execution
       time
    for i ← 1 to NumIndividuals do
        Individual ← CreateIndividual(Size); // Create an optimization
           sequence
        IndividualFitness ← Fitness(Individual, Program); // Compile and
           execute program with an individual
        Generation ← Generation ∪ (Individual, IndividualFitness) ;
    end
    Sequences[NumGeneration] ← Generation;
    while not reach at least one stop criteria do
        Generation ← evolve(Generation); // do Crossover, Mutation
           and get fitness of each individual
        NumGeneration ← NumGeneration + 1;
        Sequences[NumGeneration] ← Generation;
    end
    KD ← KD + (Program, Baselines, ProgramFeatures, Sequences)
end
```

Algorithm 1: Genetic algorithm to create KD

next generation. The fitness function used in this paper refers to the run-time of the program given in seconds. This function calculates the arithmetic mean of 5 executions for each sequence.

Both initial populations are randomly generated and comprised of either 10 or 50 individuals. Each chromosome of the individuals is a string that specifies one optimization to the compiler (those string are presented in Table 2). The number of individuals is given by a specific parameter. The size of each individual is randomly generated as well, varying between 1 and 61. This range was chosen because it relates to the number of different optimizations available (O1, O2, O3). The algorithm has 3 stop criteria:

1. The standard deviation of the fitness function is less than 0.01.
2. The total number of generations is either:
   - 100 with an initial population of 50 individuals; or
   - 20 with an initial population of 10 individuals.
3. The best fitness value does not improve after three consecutive generations.

After the GA is executed with the aforementioned parametrizations, all sequences are gathered to create the KD. Thus for every experiment, the first model is built with the KD created in the previous phase.

3.1.2 Solution Adapter

The GA was also used for adaptation purposes. Its parameterization is very similar to the GA presented in Section 3.1.1.

The mutation and crossover operators are identical, and thus have equal probability. Furthermore, the fitness function is identical as well. Thus, the main difference between the algorithms is their initial population, which is not comprised of randomly-selected individuals. Instead, the initial population is selected from the KD, and is comprised of 10 individuals and 20 generations, and every individual is a optimization sequence that is applied on the whole program.

3.2 Feature Extraction

The features used for all instantiations are shown in Table 1, and were proposed by Namolaru et al.

These features are provided by two different scopes, which are based on:

1. The entire program structure: this indicates that the extracted features describe the entities of the entire program;
2. Hot functions: this indicates that the program will only be represented by its hottest function, which is highly beneficial for the compiler to optimize. The algorithm used to search for the hottest function was proposed by Wu and Lars (1994) [25]. This strategy is justified by Amdahl's Law [16].

In both cases, the features were not submitted to a prior preprocessing.

3.3 Machine Learning

SVM is a popular and widely-acclaimed ML algorithm, and thus was chosen for this experiment. A machine learning library called *Scikit-learn* was selected for implementing SVM [17]. The configuration, parametrization and implementation of the algorithm are described in the following sections.

Number of Instructions
Number of assignment instructions
Number of integer binop instructions
Number of float binop instructions
Number of terminator instructions
Number of bitwise binop instructions
Number of vector instructions
Number of memory access and addressing instructions
Number of aggregate instructions
Number of integer conversion instructions
Number of float conversion instructions
Number of call instructions
Number of call instructions that has pointers as arguments
Number of call instructions that have more than 4 arguments
Number of call instructions that return an integer
Number of call instructions that return a float
Number of call instructions that return a pointer
Number of switch instructions
Number of indirect branches instructions
Number of conditional branches instructions
Number of unconditional branches instructions
Number of load instructions
Number of store instructions
Number of GetElemPtr instructions
Number of other instructions
Number of PHI nodes
Number of BBs with no PHI nodes
Number of BBs with up to 3 PHI nodes
Number of BBs with more than 3 PHI nodes
Number of Basic Blocks (BB)
Average number of instructions per BB
Number of edges in a Control Flow Graph (CFG)
Number of critical edges in a CFG
Average number of PHI nodes per BB
Number of BBs with 1-successor
Number of BBs with 2-successor
Number of BBs with more than 2-successor
Number of BBs with 1-predecessor
Number of BBs with 2-predecessor
Number of BBs with more than 2-predecessor
Number of BBs with 1-successor and 1-predecessor
Number of BBs with 2-successor and 1-predecessor
Number of BBs with 1-successor and 2-predecessor
Number of BBs with 2-successor and 2-predecessor
Number of BBs with more than 2-successor and 2-predecessor
Number of BBs with less than 15 instructions
Number of BBs with more than 15 instructions and less than 500 instructions
Number of BBs with more than 500 instructions

Table 1. Features

3.3.1 Parametrization of SVM

SVMs are effective tools for binary classification. One-Versus-All (commonly referred to as OVA) is a strategy used for these types of problems, and was implemented for this experiment. Thus, the decision function of our SVM algorithm is capable of ranking test programs. In addition, we also analyzed the possibility of using a statistical SVM, however the results did not match our predictions because it had a low sample rate for each class.

The SVM kernel function used were linear function, and all the parameters of the kernel were the default of *Scikit-learn* library.

The adopted approach considers every program of the KD as a class for the SVM. Thus, each class is comprised of only one example because its program characterization was static. If a dynamic characterization occurs, every program execution is collectively seen as an example.

The features used to classify the program are extracted by the Feature extractor, which was previously discussed in Section 3.2.

3.3.2 Model Creator

The algorithm 2 presents the pseudocode to the model creator.

```
Data: KD, Representation, SVMParameters
Result: Model
SVMFeatures ← ∅;
KD ← FilterBase(KD)// Removes sequences worst than the best
   baseline also remove programs that have one baseline with
   execution time equals 0 from base
if Representation = HotFunction then
    foreach (Program, Baselines, Features, Sequences) ∈ KD do
        hot ← findHotFunction(Program);
        Vector ← toVector(Features[hot]);
        SVMFeatures ← SVMFeatures ∪ (Vector, Program.Name);
        // Vector is a vector of features and Program.Name is the
           program name string, that will be used as label in SVM
    end
else
    foreach (Program, Features, Sequences) ∈ KD do
    end
    ProgramFeatures ← sumDictFields(Features);
    Vector ← toVector(ProgramFeatures);
    SVMFeatures ← SVMFeatures ∪ (Vector, Program.Name);
end
SVMModel ← SVMTrain(SVMFeatures, SVMParameters);
Model.SVM ← SVMModel;
Model.Representation ← Representation;
Model.KD ← KD;
```

Algorithm 2: Algorithm to create a SVM model

It is worth highlighting that training the SVM is done with data filtered from the KD. In addition, features that do not appear in at least one program from the KD are excluded from the training phase. The number of features can increase as new programs are added to the KD.

The database filter uses speedup as a threshold compared to the LLVM optimization levels.

The model creator has two types of instantiations, and the scope of features given by the feature extractor is different for each type. These two scopes were previously discussed in Section 3.2.

3.3.3 Sequence Predictor

The main objective of the SVM prediction phase is to predict the initial population, and consequently the GA will improve the sequences. This GA is described in Section 3.1.2.

Thus, a new program P is given to the feature extractor as an input. Afterwards, the collected features with an ML model are given to the sequence predictor. Therefore, the predictor will select sequences that will compose the initial population of the GA.

However, some sequences can cause errors to the LLVM optimizer, and thus it is vital to validate the selected sequences. Therefore, each sequence is validated, and their error prone counterparts are discarded.

The sequence prediction was instantiated in two different ways:

- Centralized: All sequences of the population are provided by the program with the highest similarity. Thus, only the aforementioned program is predicted, and consequently its sequences are extracted. However if the most similar program is not able to provide all the sequences, the second most similar program is chosen, and so forth. This process repeats until the initial population is completely built.
- Distributed: Programs provide a number of sequences (N_p), and it is proportional to the value of the decision function (Decision_val$_p$). This value is given by the prediction function of the SVM model. In this case, the programs provide their best sequences, and each program provides N_p, which is calculated by the equation 1. This prediction strategy is implemented to generate diversity between the initial sequences of the GA, considering that the sequences will originate from different programs. The sequences are extracted until the size of the initial population is reached, and are ordered from the most similar to least similar program.

$$N_p = \left\lceil \text{Population\_size} \times \frac{\text{Decision\_val}_P}{\sum_{x \in \text{Base}} \text{Decision\_val}_x} \right\rceil \quad (1)$$

In the experiments presented in this paper, the GA for adapting solutions has an initial population of 10 individuals. In the Distributed sequence prediction, each program contributes with only a single sequence. This occurs because the decision function of the SVM has small differences between two programs that occupy consecutive positions. The Algorithm 3 presents the pseudocode of this phase.

```
Data: Model, PredictionType, K, NewProgram
Result: Population
SVMModel ← Model.SVMModel;
Representation ← Model.Representation;
KD ← Model.KD;
NewProgramFeatures ← getFeaturesByFunction(NewProgram);
if Representation = HotFunction then
    hot ← findHotFunction(Program);
    NewFeatures ← NewProgramFeatures[hot];
else
    NewFeatures ← sumDictFields(NewProgramFeatures);
end
SimilartyRank ← Classify(NewFeatures, SVMModel);
// Creates an array where each element is a tuple composed by
   the programs in KD and its decision function value according
   to SVMModel.  This array is sorted byt decision function
   value.
Population ← ∅;
rank ← 0;
S ← size(Population);
if PredictionType = Centralized then
    while S < K do
        MostSimilar ← SimilarityRank[rank].Name;
        Population ←
            Population ∪ SelectSequences(MostSimilar, K-S, KD)// Select
            the K-S best Sequences form the MostSimilar
        S ← size(Population);
        rank ← rank + 1;
    end
else
    SumDV ← SumDecisionValues(SimilarityRank);
    while S < K do
        Program ← SimilarityRank[rank].Name;
        DV ← SimilarityRank[rank].DecisionValue;
        numProgSeq ← ⌈K × (DV/SumDV)⌉
        Population ← Population ∪ SelecteSequences(Program, K-S, KD);
        S ← size(Population);
        rank ← rank + 1;
    end
end
```

Algorithm 3: Algorithm to generate the starter population

3.4 Approaches for Feeding the Knowledge Database

Three experiments were conducted to evaluate the approaches for feeding the database. These experiments were handled for each instantiation of the hybrid approach. Two of these experiments were conducted to evaluate batch feeding, while the other evaluates constant feeding.

For batch feeding, the conducted experiments were the following:

- The first experiment consisted in predicting and adapting the solution for every program from one benchmark. This is done using a model generated by the KD of micro-benchmarks. Although the KD was fed during this procedure, the model was not recreated. Afterwards, an entirely new model was created, and the same procedure was made for every program from a different benchmark. However, the latter does not build new models.
- The second experiment is very similar to the first, however the benchmark order was inverted.

For constant feeding, the test programs were ordered alphabetically, and the model was recreated after every prediction and adaptation procedure.

4 EXPERIMENTAL ENVIRONMENT

The following subsections describe the hardware platform, strategies, benchmarks and metrics used for the experiments.

4.1 Experimental Architecture

The experiments were conducted in the following environment:

Hardware: Intel Core i7-3770 processor with a frequency of 3.40 GHz, 8 MB cache and 8 GB of RAM;

Operating System: Ubuntu 15.10 with kernel 4.2.0-35-generic.

4.2 Compilation System

The compilation system used was LLVM, which has difficulties with certain sequences, and thus the LLVM optimizer (opt) hangs or crashes; consequently having unresponsive behavior. Therefore, this problem was mitigated by reducing the number of optimizations. Thus, sequences were comprised of optimizations from O1, O2, and O3. These optimizations are shown in Table 2.

These optimizations do not guarantee that problem-less sequences will generate, however it does reduce unresponsive behaviors and crashes/hangs.

adce	alignment-from-assumptions	always-inline	argpromotion
assumption-cache-tracker	barrier	basicaa	basiccg
bdce	block-freq	branch-prob	constmerge
correlated-propagation	deadargelim	domtree	dse
early-cse	elim-avail-extern	float2int	functionattrs
globaldce	globalopt	gvn	indvars
inline	inline-cost	instcombine	ipsccp
jump-threading	lazy-value-info	lcssa	licm
loop-accesses	loop-deletion	loop-idiom	loop-rotate
loop-simplify	loop-unroll	loop-unswitch	loop-vectorize
loops	lower-expect	memcpyopt	memdep
mldst-motion	no-aa	prune-eh	reassociate
scalar-evolution	sccp	scoped-noalias	simplifycfg
slp-vectorizer	sroa	strip-dead-prototypes	tailcallelim
targetlibinfo	tbaa	tti	verify

Table 2. Optimizations

4.3 Benchmarks Used

We used three benchmarks: two to evaluate strategies and one to evaluate the KD generation.

KD Generation. This phase uses micro-kernel applications, which in this paper are referred to as *micro-benchmarks*. These applications are available on the LLVM test-suite, and were used for experiments conducted by Purini and Jain (2013) [18]. The complete list of these applications is shown in Table 3.

Test Programs. We used the Collective Benchmark (cBench, with the dataset configured to 1; and the Polyhedral Benchmark (PolyBench), with the dataset configured to extralarge. These benchmarks are shown in Table 4.

4.4 Evaluation Metrics

Four metrics were used for analyzing the results:

1. Speedup over O0;
2. NPS: number of programs that achieve higher speedup than the best compiler optimization level. This process is also called coverage;
3. NoS: number of evaluated sequences; and
4. ReT: response time.

The speedup is calculated as follows:

$$\text{Speedup} = \text{Runtime\_Level\_O0}/\text{Runtime}.$$

ackermann	flops-8	perm
ary3	fp-convert	pi
binary-trees	hash	pidigits
bubblesort	heapsort	puzzle
chomp	himenobmtxpa	puzzle-stanford
dry	huffbench	queens
dt	intmm	queens-mcgill
fannkuch	lists	quicksort
fasta	lpbench	random
fasta-redux	mandel	realmm
fbench	mandel-2	recursive
ffbench	mandelbrot	reedsolomon
fib2	matrix	regex-dna
fldry	methcall	richards_benchmark
flops	misr	salsa20
flops-1	n-body	sieve
flops-2	nsieve-bits	spectral-norm
flops-3	oourafft	strcat
flops-4	oscar	towers
flops-5	partialsums	treesort
flops-6	perlin	whetstone
flops-7		

Table 3. Micro-benchmarks

cBench					
automotive_bitcount	bzip2d	consumer_mad	network_dijkstra	security_blowfish_e	security_sha
automotive_qsort1	bzip2e	consumer_tiff2bw	network_patricia	security_pgp_d	telecom_adpcm_c
automotive_susan_c	consumer_jpeg_c	consumer_tiff2rgba	office_ghostscript	security_pgp_e	telecom_adpcm_d
automotive_susan_e	consumer_jpeg_d	consumer_tiffdither	office_synth	security_rijndael_d	telecom_CRC32
automotive_susan_s	consumer_lame	consumer_tiffmedian	security_blowfish_d	security_rijndael_e	telecom_gsm
Polybench					
2mm	cholesky	durbin	gesummv	lu	syr2k
3mm	correlation	fdtd-2d	gramschmidt	mvt	syrk
adi	covariance	floyd-warshall	heat-3d	nussinov	trisolv
atax	deriche	gemm	jacobi-2d	seidel-2d	trmm
bicg	doitgen	gemver	ludcmp	symm	

Table 4. Test programs

4.5 Strategies

Several strategies were evaluated. Table 5 presents the strategies for mitigating the OSP.

IC.GA.50 and IC.GA.10 are used by the GA to create the database. IC.GA.50 and IC.GA.10 have 50 and 10 individuals, respectively. IC.Best10 consists in applying 10 sequences found by Purini and Jain [18], and consequently returns the best target code.

Approach	Sequence Selection	Program Representation	Feeding Strategy	Compilation Order	Maximum NoS
The Proposed Hybrid Approaches					
H.DHB.PC	Distributed	Hot	Batch	Poly-cBench	200
H.DHB.CP	Distributed	Hot	Batch	cBench-Poly	200
H.DHC.A	Distributed	Hot	Constant	Alphabetical	200
H.DFB.PC	Distributed	Full	Batch	Poly-cBench	200
H.DFB.CP	Distributed	Full	Batch	cBench-Poly	200
H.DFC.A	Distributed	Full	Constant	Alphabetical	200
H.CHB.PC	Centralized	Hot	Batch	Poly-cBench	200
H.CHB.CP	Centralized	Hot	Batch	cBench-Poly	200
H.CHC.A	Centralized	Hot	Constant	Alphabetical	200
H.CFB.PC	Centralized	Full	Batch	Poly-cBench	200
H.CFB.CP	Centralized	Full	Batch	cBench-Poly	200
H.CFC.A	Centralized	Full	Constant	Alphabetical	200
Machine Learning Approaches					
ML.DH	Distributed	Hot	–	–	10
ML.DF	Distributed	Full	–	–	10
ML.CH	Centralized	Hot	–	–	10
ML.CF	Centralized	Full	–	–	10
Iterative Compilation Approaches					
IC.GA.50	–	–	–	–	5 000
IC.GA.10	–	–	–	–	200
IC.Best10	–	–	–	–	10

Table 5. Strategies

We also evaluated four machine learning methods: ML.DH that selects sequences using the distributed strategy for the hottest function features; ML.DF that selects sequences using the distributed strategy for the full program features; ML.CH that selects sequences using the centralized strategy for the hottest function features; and ML.CF that selects sequences using the centralized strategy for the full program features.

5 EXPERIMENTS

The following subsections describe in detail the experimental results.

5.1 Quality of the Knowledge Database

The KD was generated from two executions of our GA, and it is presented in Figure 3.

The aforementioned figure presents a *violin plot*, which shows sequences created for each *microbenchmark*. In addition, the *violin plot* represents each LLVM

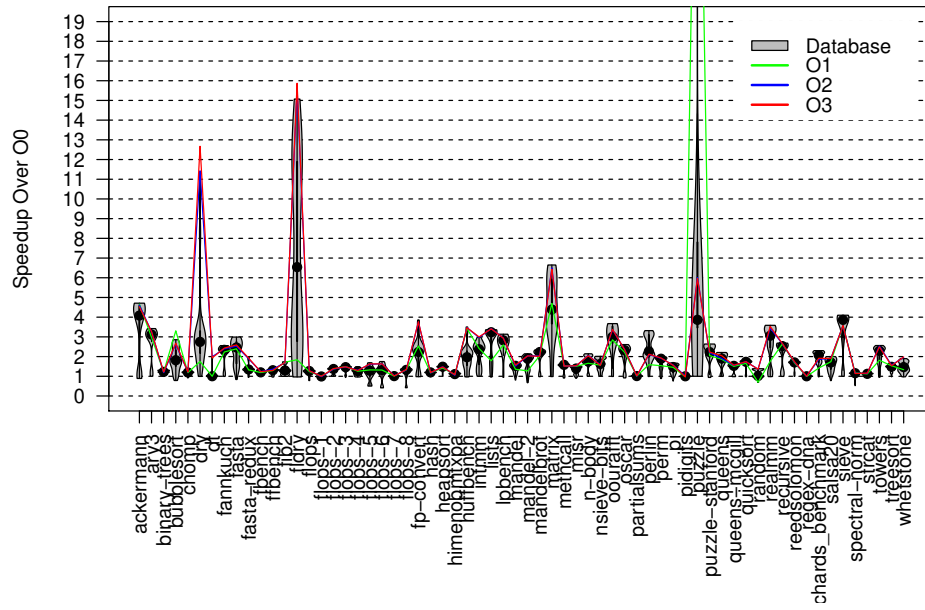

Figure 3. Knowledge Database

standard optimization level with lines. As shown in this figure, the best/most effective LLVM optimization level was superior to the GA in the following cases: *misr*, *bubblesort*, *dry*, *intmm*, *fldry*, *lists* and *whetstone*.

The genetic algorithm had a speedup rate slightly higher (not more than 3×) than the LLVM standard optimization levels. This indicates that although the base is not comprised of the best possible sequences, it can have sequences superior to the best/most effective LLVM optimization level (89.06 % of the programs) in terms of both quantity and quality.

In total, 23 410 sequences were generated. 25.18 % (5894) of these sequences were better, in terms of speedup over O0, than the most effective LLVM optimization level. The initial population can be comprised of these aforementioned sequences.

5.2 Performance

A summary of the results is presented in Table 6.

As shown above, "pure" or unaltered ML techniques had similar speedups in most cases, however their NPS is lower compared to distributed-selection-based hybrid approaches and IC. This does not apply for IC.Best10 approaches. ML.DH had the best/most effective speedups, however its NPS was worse compared to ML.DF.

A total of 59 programs were evaluated. ML.DH is superior to ML.DF in 35 programs, the latter overcame the former in just 24 programs. In addition, the best-case

Strategy	Speedup				NPS	NoS		
	Best	GMS	Worst	SDS		Max	AVG	Min
H.DHB.PC	4.47×	1.99×	1.06×	0.83×	46	110	51.09	10
H.DHB.CP	4.50×	1.99×	1.07×	0.79×	45	117	54.85	10
H.DHC.A	4.48×	1.97×	1,06×	0.83×	46	180	55.46	10
H.DFB.PC	6.00×	1.99×	1.06×	0.91×	45	140	50.14	10
H.DFB.CP	4.44×	1.96×	1.07×	0.75×	44	99	51.98	10
H.DFC.A	4.46×	1.95×	1.05×	0.78×	43	119	52.79	10
H.CHB.PC	4.42×	1.88×	1.00×	0.79×	38	150	50.53	10
H.CHB.CP	4.51×	1.85×	1.03×	0.70×	35	120	53.61	10
H.CHC.A	4.45×	1.87×	0.91×	0.77×	36	120	46.44	10
H.CFB.PC	4.20×	1.87×	1.02×	0.66×	36	149	52.98	10
H.CFB.CP	4.11×	1.85×	1.02×	0.66×	35	168	50.88	10
H.CFC.A	4.22×	1.88×	1.06×	0.66×	32	110	51.15	10
ML.DH	4.49×	1.91×	1.05×	0.77×	33	10	10	10
ML.DF	4.06×	1.90×	1.06×	0.68×	35	10	10	10
ML.CH	4.42×	1.84×	1.01×	0.72×	30	10	10	10
ML.CF	4.10×	1.82×	1.02×	0.63×	23	10	10	10
IC.GA.50	4.35×	2.083×	1.08×	0.83×	56	650	286.17	100
IC.GA.10	6.71×	1.930×	1.04×	0.92×	46	120	53.68	10
IC.Best10	3.75×	1.801×	1.05×	0.59×	24	10	10	10
O1	4.70×	1.69×	0.99×	0.63×	–	1	1	1
O2	4.37×	1.84×	1.03×	0.75×	–	1	1	1
O3	4.36×	1.84×	1.05×	0.76×	–	1	1	1

Best: the best result; GMS: geometric mean speedup; Worst: the worst result; SDS: standard deviation speedup; Max: maximun NoS; AVG: average NoS; Min: minimun NoS.

Table 6. Summary of experiments

scenario for ML.DH and ML.DF was 4.49× (gemm) and 4.06× (gemm), respectively. These results indicate that program characterization based on the hottest function provides a better initial solution to the solution adapter. In addition, another key element worth analyzing is ML. Distributed-selection-based approaches achieved a slightly higher NPS than Centralized-selection-based techniques. However, a thorough analysis revealed that ML strategies had a higher speedup than centralized-selection-based techniques in 9 programs (2mm, lame, covariance, doitgen, durbin, jacobi-2d, mvt, adpcm_d and CRC32). Finally, ML.DH had the best results in 2 cases.

Overall, IC.GA.50 had the best results. It had higher speedups than IC.GA.10 in 48 programs. Compared to other strategies, IC.GA.50 had higher speedups in 27 out of 59 programs. This result was highly anticipated because this strategy is the most aggressive; consequently, evaluating a high number of sequences. In addition, IC.GA.10 had higher speedups than IC.GA.50 in 11 programs (susan_e,

bzip2e, jpeg_c, lame, correlation, doitgen, durbin, gesummv, mvt, trmm, and seidel-2d). Thus, this confirms that less aggressive IC techniques have satisfying results in some cases, and consequently surpass more aggressive IC strategies. Additionally, IC.GA.10 and IC.GA.50 had maximal speedups of 6.71× (durbin) and 4.35× (gemm), respectively. Furthermore, IC.GA.10 had the best results, in 4 programs, among all the strategies.

IC.Best10 surpassed all the other strategies in just 2 programs (bicg and rijndael_d). This result was also anticipated because the aforementioned strategy evaluates the same number of sequences. However, this strategy excels over ML in only 14 programs (adi, atax, bicg, correlation, gemver, gesummv, lu, dikstra, ghostscript, pgp_d, rijndael_d, rijndael_e, seidel-2d and trisolv).

An individual analysis reveals that the success rate of IC.Best10 does not increase significantly compared to ML.DH and ML.DF, and achieves 15 and 19 than the aforementioned strategies, respectively. The highest speedup reached by IC.Best10 was 3.75× (doitgen). However, this is the lowest speedup compared to other strategies including the LLVM optimization levels. Thus, we conclude and confirm that this strategy is the worst compared to others.

The hybrid approach reached its highest speedup by using H.DFB.PC. The speedup rate is 6.00× (durbin). The other strategies do not possess significant differences in terms of speedups. H.CHB.CP reached approximately half of the performance of the other hybrid strategies in 1 case (nussinov). Furthermore, H.DHB.PC and H.DHC.A had the highest value in the same case (bitcount). The centralized-selection based hybrid approach had worse performance than its distributed-selection based counterpart. This indicates that looking for sequences from different sources to obtain a more diversified population is an appealing option. Overall, the best-performing hybrid strategy was H.DHC.A because it had the highest speedup in 6 programs.

All strategies, except for IC.Best10 and both MLS's, had higher speedups than the optimization levels of LLVM. Specifically, IC.GA.50 had the best performance. These results were as expected, since IC.GA.50 is the most aggressive strategy in these experiments.

The behavior of the hybrid approach was altered for every strategy. However, the most important factor in improving performance is the initial solution. It is widely known that the initial selection of sequences is highly influential to the end results, and thus centralizing the selection of an initial solution is not beneficial. This can be confirmed because its performance was lower than IC.GA.10 in the majority of the programs. However, decentralizing the selection of an initial solution was better than IC.GA.10.

IC.GA.10 is appealing because it had a high geometric mean. However, it is approximately 3.35 % lower than the geometric mean of the hybrid approach. Nevertheless, it surpassed every strategy of the hybrid approach with centralized-selection-based techniques.

It is very important to highlight that the both MLF's approaches had speedups of approximately 1.03 % and 1.04 % lower than IC.GA.10. This result indicates that these methods have low-cost benefits.

The strategies can be categorized as follows:

1. Strategies that evaluate an unfixed number of sequences; and
2. Strategies that evaluate a fixed number of sequences.

IC.GA.50 was the best strategy among those in the first category. In addition, it had the largest number of explored sequences. Statistically, it evaluated 5.5 times more than the hybrid approach.

However, the hybrid approach and IC.GA.10 evaluated almost the same number of sequences. This indicates that the hybrid approach can reach higher speedups than a "pure" or unaltered IC strategy. In addition, GAs with a well selected population is an improvement over GAs with initial random populations.

IC.Best10 and ML.CF were the worst-performing strategies among those in the second category. They had the worst coverage, and does not even reach speedups obtained by optimization levels of LLVM (O2 and O3). Considering the number of evaluated sequences, ML.DH and ML.DF had acceptable improvements, however its coverage was low. Both of these strategies are an improvement over optimization levels of LLVM (O1, O2 and O3).

IC.GA.50 is superior in terms of NPS, and covers 95 % of the programs. Furthermore, IC.GA.10 covered 78 % of the programs. The distributed-selection based hybrid approach covered 73 % (at its worst) and 78 % (at its best) of the programs. The centralized-selection based hybrid approach covered 54 % (at its worst) and 64 % (at its best). ML.DH and ML.DF covered 56 % and 59 % of the programs, respectively, and ML.CH covered 51 % of the programs. Finally, IC.Best10 and ML.CF were the two worst cases in covering, achieving 41 % and 38 %, respectively. This indicates that aggressive exploration strategies increase the coverage; consequently, covering the majority of all tested programs.

The response time is the total time spent in sequence-selection and improvement phases, however the time spent on creating the database is ignored because this process is executed only once and it will not be necessary for future compilations. ML and IC.Best10 had the lowest response times. In addition, they are also the worst-performing strategies, as discussed previously.

However, IC.GA.10 and IC.GA.50 spent an average time of 2 hours and 18 minutes, and 14 hours and 30 minutes, respectively. Finally, the hybrid approach spent 3 hours and 35 minutes finding a solution for each program.

The hybrid approach outperforms IC.GA.10 by 3.47 % (in terms of speedup); consequently, consuming 56 % more of the time spent by IC.GA.10. In addition, IC.GA.50 outperforms IC.GA.10 by up to 7.93 %, and thus consuming 530 % more time than IC.GA.10.

Another important fact to be observed is that there is a soft relation between Standard Deviation and GMS. The approaches that reach low GMS tend to provide

low Standard Deviations, and the approaches that reach high GMS tend to provide higher Standard deviation. This indicates that for some programs, the effort of iterative compilation, even with a selected start sequence set, may not achieve high improvements.

These results indicate that the most-time-consuming strategies reach the best speedups, however there are strategies (such as the hybrid approach) that increase speedups by slightly increasing the time consumption.

5.3 Different Hardware Platforms

Constant feeding experiments were executed on different hardware platforms as well. However, these experiments consisted of both cBench and Polybench benchmarks. Figure 4 presents the GMS of both the Core-i7 architecture (described in Section 4.1) and the following hardware platform: Intel Xeon E5504 processor with a frequency of 2.00 GHz, 4 MB of cache and 24 GB of RAM. The experiment performed on the Intel Xeon architecture considered an already-established KD. Thus, the results are based on the same database created on the Core-i7 architecture.

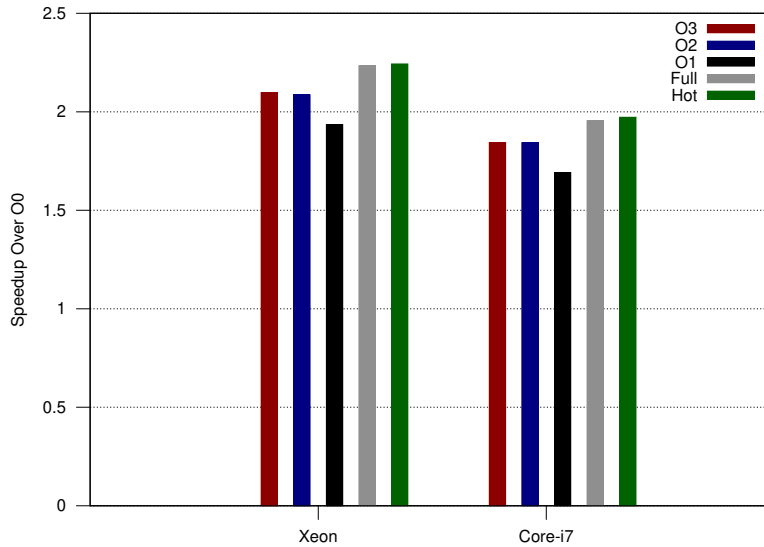

Figure 4. Results on different hardware platforms

Intel's Xeon processor had better results than the Core-i7. Compared to the best compiler optimization level (03), the hybrid approach gained performance by up to 6.49 % and 6.47 % on the Core-i7 and Xeon architecture, respectively. The performance gain at 02 and 01 optimization levels were very similar, varying no more than 1 %. The results indicate that, for both architectures, the performance gain of the hybrid approach was approximately the same proportion.

A thorough analysis of the results reveals the following conclusions.

- It is possible to obtain effective speedups using a database created on a different hardware platform.
- Representing programs based on their hot functions is an efficient strategy to reduce the performance loss when the data-set is altered, as well as the hardware platform.
- Using a hybrid approach is a smart and effective strategy to mitigate the OSP, regardless of the data-set or hardware platform.

5.4 Different Input Sets

An additional experiment was conducted with different input sets. These experiments were performed only with constant feeding and cBench because of the limited availability of several datasets. In addition, they are based only on distributed-selection based strategies. Figure 5 shows the results for these experiments. In addition, Table 7 presents speedups for each different input.

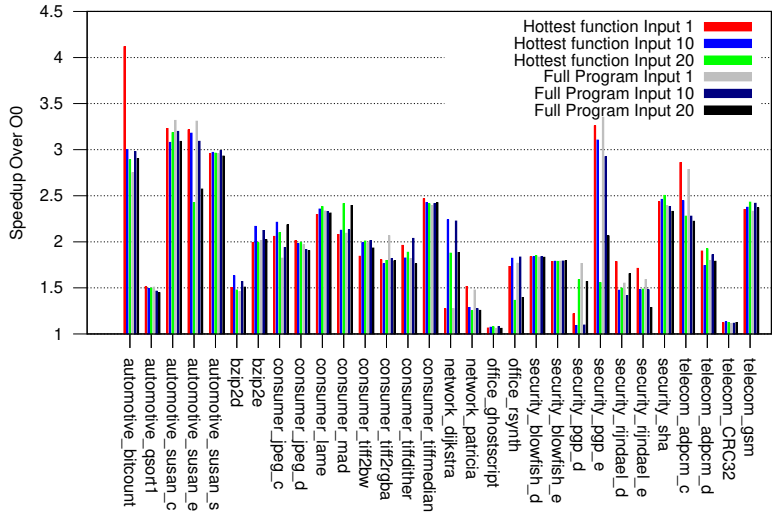

Figure 5. Different input sizes

Based on the aforementioned results, the performance was influenced by altering the data-sets, as reported in the literature [3]. The performance loss was only up to 4.86 %, regardless of the program characterization.

It is important to specify that only static features are considered when extracting good/effective sequences from the database. This process does not consider the

Input	Hot Function			Full Program		
	Best	GMS	Worst	Best	GMS	Worst
1	4.12×	1.99×	1.06×	3.35×	1.97×	1.06×
10	3.18×	1.96×	1.07×	3.20×	1.95×	1.08×
20	3.18×	1.90×	1.08×	3.09×	1.89×	1.06×

Best: the best result; GMS: geometric mean speedup; Worst: the worst result

Table 7. Speedups for each different input

program behavior when data-sets are exchanged. However, this process does not require the program execution to extract features, which will affect the system response time. Therefore, a performance loss of up to 4.86 % is appealing compared to the cost of executing a program.

6 RELATED WORKS

Zhou and Lin [26] used a genetic algorithm called NSGA-II to investigate multi-objective compilations, and thus compared randomly-selected sequences. The compiler used for this research study was GCC. The optimization levels -O1, -O2, -O3, -Os were selected for the experiments, thus totaling 54 optimizations and a search space size of 2^{54}. A set of 10 cBench programs were used, and the goal was to optimize the code size and run-time. The hyper-volumes created by NSGA-II had the best results, and random sequences had better performance than GCC optimization levels. A thorough analysis revealed that NSGA-II had the best results in terms of run-time and code size.

Jantz and Kulkarni [5] proposed an approach reducing the search-space of the optimization sequences. This is done by exploring dependencies between the optimizations. Applying cleanup phases such as dead-code elimination and dead-assignment elimination, which does not have much interaction with other optimizations, some optimizations can be removed from the search space and applied after each optimization.

The works presented by [5] and [26] focus on a pure IC strategy. They does not use a combination between ML and IC strategies as our work.

Malik [11] uses the concept of an histogram based on a data flow graph to establish the similarity between programs. This histogram was built considering the distance to a *sink* node (node without successor nodes) and a *root* node (node without predecessor nodes). Thus, Malik showed how data flow graphs can be beneficial to characterize programs using static information and SVM. However, these sets are not adapted to recently-established programs, which could provide greater benefits.

Park et al. [15] introduced a model to characterize programs based on *graph-model representation*, collecting instruction information for each node. In addition to the proposed model, the authors also implemented other techniques to characterize

programs, which can be categorized into dynamic or static representations. The former requires program execution for classifications and the latter only needs to analyze the source code. The results showed that control flow graphs had better performance among other strategies.

Jantz and Kulkarni [6] addressed the Optimization Selection Problem in just-in-time compilers using Java Virtual Machine (JVM). This paper does not consider the order of optimizations because HotSpot (in JVM) does not require it. First, the analysis was done by recompiling the methods, removing only one optimization at a time from the standard sequence of JVM (used as *baseline*). Thus, it was possible to make the following observations: most optimizations do not have negative impacts for several methods of the program. However, some methods can hinder the performance more frequently; and most optimizations have a small individual influence on the performance. In addition, they also used a logistic regression thechnique to predict sequences and then compare it to an GA implementation.

Lima et al. [10] used machine learning to improve power and performance efficiency during compilation. They showed that sacrificing performance was unnecessary to reduce power consumption.

Junior and da Silva [7] evaluated the performance of different case-based reasoning configurations, which use dynamic and static features to find good/effective optimization sequences. The authors presented a measure of similarity among other strategies to build past experiences. The proposed algorithm is divided into two phases: an *offline* phase; and an *online phase*. The former creates sequences for prior experiences of the knowledge database. This phase can be done randomly or using a meta-heuristic. The *online* phase is based on case-reasoning, and thus all prior program experiences are analyzed, and its sequences are filtered. Afterwards, the input sequence is compiled with the given number of analogies. These sequences are extracted from the most similar program. The results show that the approach used to create the knowledge database influences the results.

The works presented by [7, 10, 11, 15, 6] uses IC to generate a knowledge database and also use ML to select a number of sequences and then pick up the best between them. Instead of the instruction: select the best one, our work adapts the sequence after ML phase.

Martins et al. [12] implemented a clustering approach, transforming code from program functions to symbolic representations, to mitigate the OSP. These representations refer to the DNA of the program, where *tokens* of the source code are transformed into characters. The clustering approach starts by extracting DNA from the program function. Thus, the distance matrix for programs is calculated using the normalized compression distance algorithm. This matrix will serve as a basis to build a tree topology in which the clustering algorithm will be capable of finding possible clusters. Finally, all optimizations are included in the reduced search space. The results showed that reducing the search space is significant and highly beneficial to the performance. This work differs from our because the initial population of GA is generated by the optimization of the reduced search space,

while our work select the initial population to the program based on SVM prediction.

Purini and Jain [18] proposed a search strategy slightly different than machine learning and iterative compilation. The objective of this strategy was to build a set of good optimization sequences, where for each class of programs there exists a sequence of optimizations that reach a satisfying performance. While in this work the goal is to search a generic optimization sequence that fits well in different programs, our work focus on search specific optimization sequences.

Tartara and Crespi Reghizzi [21] proposed a continuous learning approach that does not require prior knowledge to create optimization sequences. These sequences are represented as mathematical formulas. Every formula can be represented using a grammar based on certain rules of binary operations, Boolean and numeric values, if-expressions, comparison operators, arithmetic and Boolean operators, and program features. This approach uses a knowledge database for storing and extracting sequences, however this information is randomly-generated if the number of sequences is insufficient. The results showed that converging for good performance does not require several executions, and in some cases it has better values than the highest optimization level in both compilers. This work is different from our work, because it does not have an *offline* phase. In our work we used the training group component that does the offline phase.

7 CONCLUSION

The selection of optimization sequences can greatly impact the run-time of a program. In addition, the OSP depends heavily on the hardware architecture.

This paper proposes a hybrid approach for mitigating the OSP. Since this problem is complex, this approach uses an ML approach (SVM) to feed the initial solution of a GA and then search for good/effective solutions. The results showed that the hybrid approach achieved significant improvements over "pure" or unaltered ML and IC strategies, achieving speed-ups over 2.00×, 1.93× and 2.08× with H.DHB.PC, IC.GA.10 and IC.GA.50, respectively. Machine learning strategies achieved the lowest speed-ups among the aforementioned approaches, 1.91.

It is extremely important to highlight that the strategy used for the initial sequences had the highest overall influence, and we conclude that distributed-selection is the best approach for selecting the initial solution. In addition, the experiments showed that the approach is portable, because it can be transferred from one architecture to another without a performance loss. Furthermore, the hybrid approach is beneficial because its solutions improve over time, whereas ML and IC do not provide this advantage. IC strategies need to restart the process on every occasion and ML cannot create new sequences.

Future works include investigating the influence of different schemes to generate the KD, and additional ML approaches to predict the initial population. In addition,

instantiating the hybrid approach with more aggressive solution adapters can be proposed in the future.

REFERENCES

[1] Aho, A. V.—Lam, M. S.—Sethi, R.—Ullman, J. D.: Compilers: Principles, Techniques and Tools. Prentice Hall, 2006.

[2] Cavazos, J.—Fursin, G.—Agakov, F.—Bonilla, E.—O'Boyle, M. F. P.—Temam, O.: Rapidly Selecting Good Compiler Optimizations Using Performance Counters. Proceedings of the International Symposium on Code Generation and Optimization (CGO '07), IEEE, 2007, pp. 185–197, doi: 10.1109/cgo.2007.32.

[3] Chen, Y.—Huang, Y.—Eeckhout, L.—Fursin, G.—Peng, L.—Temam, O.—Wu, C.: Evaluating Iterative Optimization Across 1000 Datasets. SIGPLAN Notices, Vol. 45, 2010, No. 6, pp. 448–459, doi: 10.1145/1809028.1806647.

[4] Daud, S.—Ahmad, R. B.—Murthy, N. S.: The Effects of Compiler Optimisations on Embedded System Power Consumption. International Journal of Information and Communication Technology (IJICT), Vol. 2, 2009, No. 1-2, pp. 73–82, doi: 10.1504/ijict.2009.026431.

[5] Jantz, M. R.—Kulkarni, P. A.: Exploiting Phase Inter-Dependencies for Faster Iterative Compiler Optimization Phase Order Searches. 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013, pp. 1–10, doi: 10.1109/cases.2013.6662511.

[6] Jantz, M. R.—Kulkarni, P. A.: Performance Potential of Optimization Phase Selection During Dynamic JIT Compilation. SIGPLAN Notices, Vol. 48, 2013, No. 7, pp. 131–142, doi: 10.1145/2517326.2451539.

[7] Queiroz Junior, N. L.—da Silva, A. F.: Finding Good Compiler Optimization Sets – A Case-Based Reasoning Approach. Proceedings of the 17th International Conference on Enterprise Information Systems, 2015, Vol. 2, pp. 504–515, doi: 10.5220/0005380605040515.

[8] Queiroz Junior, N. L.—Rodriguez, L. G. A.—da Silva, A. F.: Combining Machine Learning with a Genetic Algorithm to Find Good Compiler Optimizations Sequences. Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS), 2017, Vol. 3, pp. 397–404, doi: 10.5220/0006270403970404.

[9] Lattner, C.—Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04), 2004, pp. 75–86, doi: 10.1109/cgo.2004.1281665.

[10] de Lima, E. D.—de Souza Xavier, T. C.—da Silva, A. F.—Ruiz, L. B.: Compiling for Performance and Power Efficiency. Proceedings of the 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013, pp. 142–149, doi: 10.1109/patmos.2013.6662167.

[11] Malik, A. M.: Spatial Based Feature Generation for Machine Learning Based Optimization Compilation. 2010 Ninth International Conference on Machine Learning and Applications (ICMLA), 2010, pp. 925–930, doi: 10.1109/icmla.2010.147.

[12] 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, Vol. 13, 2016, No. 1, Art. No. 8, 28 pp., doi: 10.1145/2883614.

[13] Muchnick, S. S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[14] Namolaru, M.—Cohen, A.—Fursin, G.—Zaks, A.—Freund, A.: Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, ACM, 2010, pp. 197–206, doi: 10.1145/1878921.1878951.

[15] Park, E.—Cavazos, J.—Alvarez, M. A.: Using Graph-Based Program Characterization for Predictive Modeling. Proceedings of the Tenth International Symposium on Code Generation and Optimization, ACM, 2012, pp. 196–206, doi: 10.1145/2259016.2259042.

[16] Patterson, D. A.—Hennessy, J. L.: Computer Organization and Design: The Hardware/Software Interface. Fourth Edition. The Morgan Kaufmann Series in Computer Architecture and Design, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[17] Pedregosa, F.—Varoquaux, G.—Gramfort, A.—Michel, V.—Thirion, B.—Grisel, O.—Blondel, M.—Prettenhofer, P.—Weiss, R.—Dubourg, V.—Vanderplas, J.—Passos, A.—Cournapeau, D.—Brucher, M.—Perrot, M.—Duchesnay, E.: Scikit-Learn: Machine Learning in Python. The Journal of Machine Learning Research, Vol. 12, 2011, pp. 2825–2830.

[18] Purini, S.—Jain, L.: Finding Good Optimization Sequences Covering Program Space. ACM Transactions on Architecture and Code Optimization, Vol. 9, 2013, No. 4, Art. No. 56, 23 pp., doi: 10.1145/2400682.2400715.

[19] Scott, M. L.: Programming Languages Pragmatics. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2009.

[20] Sebesta, R. W.: Concepts of Programming Languages. Addison Wesley, San Francisco, CA, USA, 2009.

[21] Tartara, M.—Crespi Reghizzi, S.: Continuous Learning of Compiler Heuristics. ACM Transactions on Architecture Code Optimization, Vol. 9, 2013, No. 4, Art. No. 46, 25 pp., doi: 10.1145/2400682.2400705.

[22] GNU Compiler Collection. 2017, `http://gcc.gnu.org`.

[23] Intel C Compiler. 2017, `https://software.intel.com/en-us/c-compilers`, doi: 10.1145/62297.62412.

[24] The LLVM Compiler Infrastructure. 2017, `http://llvm.org`.

[25] Wu, Y.—Larus, J. R.: Static Branch Frequency and Program Profile Analysis. Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO 27), ACM, 1994, 11 pp., doi: 10.1145/192724.192725.

[26] Zhou, Y.—Lin, N.: A Study on Optimizing Execution Time and Code Size in Iterative Compilation. 2012 Third International Conference on Innovations in Bio-Inspired Computing and Applications (IBICA), 2012, pp. 104–109, doi: 10.1109/ibica.2012.46.

Nilton Luiz Queiroz Junior is Professor in the Department of Informatics of the State University of Maringá located in Brazil. He is lecturing undergraduate courses. He received his Bachelor and Master degrees in computer science from the State University of Maringá, Brazil in 2014 and 2016, respectively. His research interests include parallel programming and compile techniques.

Anderson Faustino da Silva is Professor in the Department of Informatics of the State University of Maringá located in Brazil, lecturing undergraduate and graduate courses. He received his Bachelor's degree in computer science from the State University of West Paraná, Brazil in 2000. He then received his Master and Ph.D. degrees in systems engineering and computer science from the Federal University of Rio de Janeiro, Brazil in 2003 and 2006, respectively. His research interests include parallel programming and compile techniques.

Luis Gustavo Araujo Rodriguez is currently Ph.D. student in computer science at the University of São Paulo in Brazil. He received his Bachelor's degree in computer science from the Catholic University of Honduras in 2013. He then received his Master's degree in computer science from the State University of Maringá, Brazil in 2016. His research interests include high performance computing, parallel programming, and compiler.