Computing and Informatics, Vol. 37, 2018, 165–185, doi: 10.4149/cai_2018_1_165

EXPLORATION OF COMPILER OPTIMIZATION SEQUENCES USING A HYBRID APPROACH

Tiago Cariolano de Souza Xavier, Anderson Faustino da Silva

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

Abstract. Finding a program-specific compiler optimization sequence is a challenge, due to the large number of optimizations provided by optimizing compilers. As a result, researchers have proposed design-space exploration schemes. This paper also presents a design-space exploration scheme, which aims to search for a compiler optimization sequence. Our hybrid approach relies on sequences previously generated for a set of training programs, with the purpose of finding optimizations and their order of application. In the first step, a clustering algorithm chooses optimizations, and in the second step, a metaheuristic algorithm discovers the sequence, in which the compiler will apply each optimization. We evaluate our approach using the `LLVM` compiler, and an `I7` processor, respectively. The results show that we can find optimization sequences that result in target codes that, when executed on the `I7` processor, outperform the standard optimization level `O3`, by an average improvement of 8.01 % and 6.07 %, on Polybench and cBench benchmark suites, respectively. In addition, our approach outperforms the method proposed by Purini and Jain, `Best10`, by an average improvement of 24.22 % and 38.81 %, considering the two benchmarks suites.

Keywords: Compilers, optimizations, sequence, performance

Mathematics Subject Classification 2010: 68-N20

1 INTRODUCTION

Optimizing compilers provide a large number of transformations, known as optimizations, which are applied during the compilation process [25]. The aim is to create a target code semantically equal to the source code, but with good performance. Due to the large number of optimizations and the fact that each optimization interacts with each other in complex ways, it is a challenge, even for an expert programmer, to find good optimization sequences. To minimize this challenge, optimizing compilers offer optimization levels (e.g. `O0`, `O1`, `O2` and `O3` in the case of LLVM), which consist of specific sequences.

The choice of optimizations and their order of application has a significant impact on performance [19]. In addition, it is program-specific dependent [4, 6, 10, 11, 21, 22].

Exhaustive design-space exploration, although possible, takes a long time to make it suitable for use in typical iterative compilers. Therefore, researchers engage in proposing design-space exploration schemes to find a program-specific optimization sequence using few evaluations.

In this paper, we propose a hybrid approach to search for good optimization sequences aiming at performance improvements. First of all, a training stage tries to choose good sequences. After the deployment stage, which relies on previously generated sequences, it discovers a program-specific optimization sequence. In our approach, we employ several strategies: random sampling, genetic algorithm, clustering, metaheuristic, and a reduction scheme.

On one hand, the choice of optimizations is based on the premise that similar programs react approximately the same way, when they are compiled using the same sequence. In such manner, a new program can be improved by the optimizations used on a similar program. However, discovering the order of application is based on the premise that this problem is similar to the Traveling Salesman Problem. Therefore, it is possible to develop a reduction algorithm to transform a problem into another, in order to solve it using an existing solution, and then utilizing such result as a solution for the previous conflict.

The experimental results show that our approach finds an optimization sequence that outperforms the standard optimization level `O3`, besides the approach proposed by Purini and Jain [24], `Best10`, considering Polybench and cBench as benchmarks suites.

2 OUR APPROACH

During compilation, the compiler applies several optimizations, in order to improve the target code. However, some optimizations can be useful to a specific program, but not to another. Thus, the most appropriate approach is to choose optimizations and their order of application, considering that it is program-specific dependent.

In this paper we present a design-space exploration scheme that chooses and orders optimizations.

2.1 Overview

It is possible to choose and order optimizations easily, based on the assumption that two *similar* programs react in the same manner, when they are compiled using the same optimizations. Thus, we can compile a new program by applying the optimizations utilized on a *similar* and previously-compiled program. In addition, based on the assumption that a problem can be transformed into another, we can convert the problem of discovering the order of application into the well-known Traveling Salesman Problem (`TSP`) [2], and use a well-known solution to solve the `TSP`.

Our approach can be outlined as:

1. Training stage
   (a) Generate the training data
      i Extract the feature vector f, using compiler level `O0`
      ii Record the feature vector f
      iii Record the benchmark running time, using compiler level `O3`
   (b) Generate the training data for `O3`
      i Instrument each training program
      ii For each training program
         A Select a set of optimizations and apply them
         B Record the running time
2. Deployment stage
   (a) Choose the optimizations
      i Collect feature vector $f1$ from each training program, using compiler level `O0`
      ii Extract feature vector $f2$ from the test program, using compiler level `O0`
      iii Reduce $f1$ and $f2$ to the most significant components using the Principal Component Analysis `PCA`
      iv Cluster the new feature vectors $f1'$ and $f2'$ into N clusters
      v Extract the best set of optimizations from each training program, which belongs to the same cluster of the test program
   (b) Discover the order of application
      i For each set of optimizations selected
         A Reduce the problem of discovering the order of application into the `TSP`
         B Solve the `TSP`
         C Transform the solution into an optimization sequence
   (c) Return the optimizations and their order of application

2.2 Training Stage

The training stage aims to collect pieces of information about several training programs. As a result, this stage provides a small knowledge base (`KB`).

The `KB` can be viewed as a table composed by several entries, where each entry consists in four fields, namely:

1. Program name;
2. Runtime for the program, when it is compiled using optimization level `O3`;
3. Feature vector, when the program is compiled using optimization level `O0`; and
4. Compiler optimization sequences and their runtime.

2.2.1 Generating the Feature Vector

The feature vector is composed of dynamic information, which is collected during program execution. This means that such vector characterizes the dynamic behavior of the program. We use performance counters as feature vectors.

Performance counters are dynamic information that consists of performance data such as the number of issued instructions, completed instructions, cache accesses, cache hits, cache misses, mispredicted branches, and others. They are traditionally used for hardware performance analysis [3, 8, 14, 16].

The work of Cavazos et al. [4] was the first to propose the use of performance counters to characterize programs and measure their similarities. A recent work [7] also demonstrated that performance counters is a good strategy to measure the similarity between two programs. In this paper, we characterize programs in the same manner.

The use of performance counters is attractive, because they do not limit the program class, which the system is able to handle. As a result, our system (strategy) can find a good compiler optimization sequence for any program.

Table 1 presents the features used in our approach.

Type	Features						
Cache	L1_ICM	L1_DCM	L1_STM	L1_TCM	L1_LDM	L2_DCR	L2_TCA
	L2_DCW	L2_STM	L2_TCM	L2_TCR	L2_DCA	L2_TCW	L2_ICR
	L2_DCH	L2_DCM	L2_ICA	L2_ICM	L2_ICH	L3_DCR	L3_TCA
	L3_DCW	L3_TCM	L3_TCR	L3_DCA	L3_TCW	L3_ICR	L3_ICA
Branch	BR_PRC	BR_UCN	BR_NTK	BR_INS	BR_MSP	BR_TKN	BR_CN
SIMD	VEC_SP	VEC_DP					
Floating Point	FDV_INS	FP_INS	DP_OPS	FP_OPS	SP_OPS		
TLB	TLB_DM	TLB_IM					
Cycles	REF_CYC	TOT_CYC	STL_ICY	STL_ICY			
Insts	TOT_INS						

Table 1. Features

In order to standardize the feature vectors of training and test programs, we normalize each feature by `TOT_INS`. To collect these features, we use the tools `PAPI` [18] and `PerfSuite` [12].

2.2.2 Generating Training Data for Optimization Level `O3`

The optimizations used to generate training data belong to the optimization level `O3`. They are presented in Table 2.

inline	prune-eh	scalar-evolution
argpromotion	inline-cost	indvars
gvn	functionattrs	loop-idiom
slp-vectorizer	sroa	loop-deletion
globaldce	domtree	loop-unroll
constmerge	early-cse	memdep
targetlibinfo	lazy-value-info	memcpyopt
no-aa	jump-threading	sccp
tbaa	loop-unswitch	dse
tailcallelim	adce	notti
ipsccp	loop-simplify	block-freq
instcombine	loop-rotate	loop-vectorize
verify	licm	simplifycfg
globalopt	loops	branch-prob
deadargelim	lcssa	basicaa
reassociate	barrier	basiccg
correlated-propagation	strip-dead-prototypes	

Table 2. Optimizations

The process of creating compiler optimization sequences is guided by the following criteria:

- Every optimization appears only once;
- Every optimization can appear in any position;
- Every optimization has to address the compilation infrastructure rules; and
- All sequences, in `KB`, have 40 optimizations.

The first criterion indicates that we do not explore the use of an optimization several times, even though this occurs in all optimization levels, in the case of `LLVM`. The second one indicates that there are no restrictions when a specific optimization should be applied. The third one indicates that a new sequence cannot violate the safety of the `LLVM`. In the fourth criterion, the creation process tries to give the same characteristic to every sequence.

Several strategies can be used to build a base of sequences. We use the strategy proposed by Purini and Jain [24], which consists in using random and genetic algorithms to create effective sets of optimizations. The random algorithm generates

sets utilizing a uniform and random sampling of the search space. While, the genetic algorithms use a sophisticated way to build sets, and explore the search space. The algorithms are described as follows.

Random Algorithm. This iterative algorithm randomly generates 500 sequences.

Genetic Algorithm with Rank Selector. This algorithm generates sets using a genetic process, such as crossover and mutation. A simple genetic algorithm consists in randomly generating an initial population, which will result in an iterative evolution process. Such procedure of evolving a population (or a generation) involves choosing the parents; applying genetic operators; evaluating new individuals; and finally a reinsertion operation deciding which individuals will compose the new generation. This iterative process is performed until a stopping criterion is reached. The first generation is composed of individuals that are generated by a uniform sampling of the optimization space. Evolving a population includes the application of two genetic operators: crossover, and mutation. The first operator has a probability of 90 % for creating a new individual. The second operator, mutation, has a probability of 2 % for transforming an individual. Two types of mutation procedures were proposed:

1. to exchange two optimizations from random points; and
2. to change one optimization in a random point.

Both operators have the same probability of occurrence, though only one mutation is applied over the individual selected to be transformed. This iterative process uses elitism, which maintains the best individual in the next generation. Furthermore, it runs over 100 generations and 60 individuals, and finishes whether the standard deviation of the current fitness score is less than 0.01, or the best fitness score does not change in three consecutive generations.

Genetic Algorithm with Tournament Selector. It is similar to the previous strategy, but instead of using a rank selector it uses a tournament selector ($Tour = 5$).

Each strategy creates two sequences in each round. The first sequence is created utilizing the specific scheme (random or genetic), and the second one is the first sequence modified by human knowledge.

The `LLVM`'s manual suggests that some optimizations should precede and/or succeed a specific optimization for its effectiveness, so that the first sequence is updated to reflect this knowledge. This update follows the criteria:

- *loops* should appear before the first loop optimization;
- *inline-cost* should appear before *inline* and *always-inline*; and
- *verify* should be the last optimization.

After generating several sequences, we select the two best sequences for each training program; one is the best sequence generated by each algorithm, and the other is the best updated sequence.

Each sequence can contain optimizations that do not contribute to the program speedup or have a negative impact on the program. Therefore, the next step is to eliminate these unnecessary optimizations using the `Sequence Reduction Algorithm`, also proposed by Purini and Jain [24].

As the training stage uses 61 training programs, our `KB` has 366 sequences.

2.2.3 Training Benchmarks

The training programs are composed of microkernels, which were taken from `LLVM`'s test-suite. These are programs composed of a single source code, and have short running times. Table 3 shows the training programs.

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

Table 3. Microkernels

2.3 Deployment

The deployment stage performs seven steps in order to choose optimizations and their order of application, namely:

1. Extract the feature vector f from the test program, using compiler level `O0`;
2. Cluster the training and test programs, based on their feature vectors;
3. Extract from each training program, which belongs to the same cluster of the test program, their sequences;

4. Reduce the problem of choosing the order of applying compiler optimizations into the TSP;
5. Solve the TSP;
6. Transform the TSP's result into a solution to choose and order optimizations; and
7. Return the best target code.

2.3.1 Choosing Optimizations

The choice of optimizations is based on the premise that we can find similar patterns among programs, which give important insights for determining potential optimizations.

Based on the premise that similar programs react approximately the same way, when they are compiled using the same optimizations, we choose the optimizations that will be enabled during the compilation of the test program from a similar training program. In such manner, each program is represented by a feature vector forming points in a multidimensional space, and a clustering algorithm that operates in this space trying to group points that are proximate.

The task of the clustering algorithm is to group a set of programs, in such a way that programs in the same group (cluster) are more similar to each other than to those that belong to other clusters [28].

Finding similar programs is a task performed in two steps. First, we extract the feature vectors from the training and test programs. Second, the clustering algorithm reduces the feature vectors to the most significant components using PCA [28], and clusters the programs. In this moment, we know which programs are similar.

After clustering the programs, we extract from each training program, which belongs to the same cluster (C) of the test program, their sequences.

Even though the word *sequence* indicates order, in this point the extracted sequences only indicate the optimizations that will be enabled during the compilation of the test program. As ordering optimization is also a program-specific problem, we need to analyze the test program. In our strategy, ordering optimization is based on the insights given by the sequences, which achieve performance on training programs.

In a nutshell, each specific optimization that appears in the extracted sequences forms the set of optimizations that will be enabled by the compiler. In addition, these sequences give insights on when the compiler should apply each optimization.

2.3.2 Discovering the Order of Application

After choosing the optimizations, the next step is to discover the order of application. Such process is performed by extracting knowledge from KB, which is associated with the training programs (their sequences) that belong to C.

This knowledge is obtained by analyzing pairs of optimizations, in order to find patterns that are meaningful to the test program. In fact, these patterns determine

how we will join the pairs to form a new sequence. Therefore, based on these patterns we modify the order of application.

If we consider that optimizations are vertices, and that there is a cost of applying o_i before o_j, and vice-versa, the problem of choosing the order of application can be reduced into the well-known Asymmetric Traveling Salesman Problem (`ATSP`) [2].

It is important to note that the asymmetric version is the appropriate algorithm, because the performance of applying the optimization o_i before o_j can be different from applying o_j before o_i.

The performance of a pair of optimizations can be viewed as a cost. If after analyzing the previously-generated sequences, we consider that applying the optimization o_i before o_j will reduce the performance of the test program, therefore, we provide a high cost to the pair (o_i, o_j) in order to reflect this behavior.

The cost of a pair of optimizations is based on the frequency of all pairs. Given a set of optimizations `S`, we analyze the sequences previously generated in order to find how often all possible pairs formed with `S` optimizations occur in the sequences that belong to C. As the sequences previously generated can provide speedup to the training programs, our approach is guided by the assumption that pairs with high frequency are a potential order.

Reducing Our Problem into ATSP. To reduce the problem of discovering the order of application into `ATSP`, we perform three steps:

1. Create a complete digraph;
2. Map optimizations to vertices; and
3. Weigh the edges.

The first two steps are trivial. To perform the third, we need to infer the cost of the pair (o_i, o_j), which is based on the frequency that o_i appears before o_j in the sequences that belong to the cluster C and is defined as:

$$\mathit{Freq\_Prog}(p, o_i, o_j) = |\{s \in \mathit{Dom}(ES(p)) \mid (o_i \wedge o_j \in s) \wedge o_i \prec o_j\}|.$$

As a result of using $Freq\_Prog$, the function $Freq$ that returns how often o_i appears before o_j is defined as:

$$\mathit{Freq}(o_i, o_j) = \sum_{p \in C} \mathit{Freq\_Prog}(p, o_i, o_j). \tag{1}$$

If $\mathit{Freq}(o_i, o_j) < \mathit{Freq}(o_j, o_i)$, our approach considers that the application of (o_j, o_i) is the best choice. This means that the higher $\mathit{Freq}(o_j, o_i)$ is (implying in a low value of $\mathit{Freq}(o_i, o_j)$), the higher the cost of (o_i, o_j) will be. Therefore, the cost of applying the pair (o_i, o_j) is given by the inverse frequency of that pair, $\mathit{Cost}(o_i, o_j) = \mathit{Freq}(o_j, o_i)$.

Not all pairs of optimizations appear in all C sequences, as a result a high variation occurrence between two different pairs is possible, on their frequencies. To solve this problem, the cost is normalized as follows:

$$Cost(o_i, o_j) = \frac{Freq(o_j, o_i)}{Freq(o_i, o_j) + Freq(o_j, o_i)}. \tag{2}$$

With this standardization, $Cost(o_i, o_j)$ will always range from 0 to 1. In addition, if there are only (o_j, o_i) occurrences, thus $Cost(o_i, o_j) = 1$, which is the highest possible cost.

Solving the ATSP. The algorithm that solves the `ATSP` is based on Ant Colony Optimization (`ACO`) [9].

`ACO` is a metaheuristic of combinatorial optimization, which is based on the behavior of real ants. A metaheuristic is "a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems" [9]. This metaheuristic was well exploited and firstly applied to the Traveling Salesman Problem (TSP) [2, 9].

The execution of an `ACO` algorithm is composed of cycles. Each ant is usually a constructive method and its behavior can be noted when, in order to choose the next vertex to where the ant must go, a probability is used which is calculated based on two factors: pheromone trail and heuristic information [27, 1]. Once the solutions are constructed by the ants, they are used to update the pheromone trail.

In our `ACO`-based algorithm, each ant constructs a solution S, choosing vertices to move to an iterative process. The choice of a vertex v, which was not visited, is based on the probability p, as follows:

$$p_{ij}^k = \begin{cases} \frac{[\tau_{ij}]^\alpha [\eta_i]^\beta}{\sum_{k \notin Visited_k} [\tau_{ij}]^\alpha [\eta_{ij}]^\beta} & \text{if } j \notin Visited_k, \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

where τ_{ij} is the pheromone on edge (i, j), $\eta_{ij} = 1/d_{ij}$ is the visibility of the vertex j by ant k positioned on i, d_{ij} is the distance between i and j, $Visited_k$ is the set of vertices visited by the ant k, α is the importance of the pheromone and β is the importance of the visibility (heuristic information).

After all ants construct their solutions, the algorithm updates the pheromone trails. This, stored on matrix $P_{|V|\times|V|}$, is initialized with 1 for each edge between non-adjacent vertices and with 0 for each edge between adjacent vertices. Updating the pheromone trail involves the persistence of the current trail by a τ factor, and the evaporation that is based on a ρ factor. The evaporation (Equation (4)), and the general form of depositing pheromone (Equation (5)) are as follows:

$$P_{ij} = \rho P_{ij}, \qquad \forall i, j \in V, \tag{4}$$

$$P_{ij} = P_{ij} + \Delta\tau_{ij}^k, \qquad \Delta\tau_{ij}^k = \frac{Q}{T_k} \tag{5}$$

where Q is an empirical parameter and T_k is the length of the *tour* found by the ant k.

These steps are repeated until the algorithm reaches 100 cycles.

2.3.3 Returning the Best Target Code

After choosing optimizations and their order of application, there are $L*2$ sequences; L sequences extracted from the `KB`, and new L sequences, in which the order of the optimizations was modified. To find the best optimizations and their order, we measure the performance of each target code using each sequence. After this evaluation, we return the best target code.

3 EXPERIMENTAL SETUP AND METHODOLOGY

This section describes the experimental setup and the steps taken to ensure measurement accuracy, besides outlining the methodology used in the experiments.

Platform. The experiments were conducted on a machine with an Intel processor Core I7-3779, 8 MB of cache, and 8 GB of RAM. The operating system is Ubuntu 14.04, with kernel 3.13.0-37-generic.

Compiler. Our technique was implemented on top of `LLVM` 3.5 [13, 15]. The choice of `LLVM` is based on the fact that it allows full control over the optimizations. This means that it is possible to enable a list of optimizations through the command line. In addition, the position of each optimization indicates its order. Neither `GCC` nor `ICC` provides these features, thus, we need to use `LLVM` to show the results of our strategy.

Benchmark Suites. The experiments use the POLYBENCH suite [23] with a large dataset, and the CBENCH suite [5] with dataset 1, as test programs.

Measurement. The results are based on the arithmetic average of five executions. In the experiments, the machine workload was as minimal as possible. In other words, each instance was executed sequentially. In addition, the machine did not have an external interference, and the running time variance was close to zero.

Baseline. The baseline is the `LLVM`'s highest compiler optimization level, `O3`. In terms of running time, the optimization levels `O2` and `O3` have similar performance, on several programs. Therefore, we choose the highest compiler optimization level, `O3`.

Cross-Validation, Clustering, and `ACO`. The experiments use two distinct groups of programs, separating one group for training and the other for testing. Therefore, the experiments perform a holdout cross-validation. The clustering algorithm used was Farthest First, which is implemented on `Weka` [28]. In fact, we evaluate Expectation Maximization, Kmeans, and Farthest First, and the

latter obtained the best results. We use ACO to solve ATSP because it was extensively studied on the TSP. In addition, its way of choosing the next vertex is helpful for our purpose.

Parameters. The parameters used are:

- Clustering: $[10, 15]$, and $[30, 35]$. It indicates that the clustering-based algorithm will try to find at least 10 centroids, and at most 15; or at least 30, and at most 35, respectively. The former tries to gather the training programs, while the latter tries to scatter the training programs.
- ACO: $\alpha = 1$; $\beta = 5$; $\rho = 0.99$; and $Q = 100$.

Metrics. The evaluation uses three metrics to analyze the results, namely:

1. Average Percentage Improvement (API): indicates how much our strategy outperforms the compiler optimization level O3;
2. Average Percentage Improvement Excluding (APIE) Programs: indicates how much our strategy outperforms the compiler optimization level O3, considering only the programs whose performance outperforms the compiler optimization level O3's performance; and,
3. Number of Programs Achieving Improvement (NBI): indicates the number of programs whose performance, obtained with our strategy, was better than using the compiler optimization level O3.

The improvement is calculated as follows:

$$Speedup = baseline\_running\_time / new\_running\_time,$$

$$Improvement = (Speedup - 1) * 100.$$

Training and Deployment Cost. The training, which builds sequences, is a high time-consuming phase. It took several days, which is a significant amount of time. However, it is important to note that it is performed only once, besides performed at the factory. The deployment cost is calculated as follows:

$$Deployment_{cost} = C_{time} + O_{time} + \sum_{S=0}^{Sequences*2} \left(Comp_{time} + \sum_{N=0}^{5} Runtime \right)$$

where C_{time} is the time spent to choose optimizations; O_{time} is the time spent to order optimizations; $Comp_{time}$ is the time spent to compile the program using a specific sequence; and $Runtime$ is the program running time.

Choosing and ordering optimizations takes only 20 % of the system response time in our experiments. It is directly proportional to the size of the ATSP, besides the size of the KB. The other portion of the system response time (80 %) is caused by the need of compiling and running a program several times, in

order to evaluate a sequence and ensure measurement accuracy. In our experiments, choosing and ordering optimizations took from 0.015 (ADI) to 6.755 (GRAMSCHMIDT) seconds, while evaluating sequences took from 0.06 to 9 460.5 seconds.

Best10. In order to evaluate the effectiveness of our approach, we compare it with the one proposed by Purini and Jain [24]. They proposed an approach that extracts the best 10 optimization sequences, from a small search space, and use these sets to compile all programs. They argue that it is possible to outperform the optimization levels using only 10 sets.

Briefly, the approach used to select the best 10 sequences can be summarized in five steps:

1. Generate training data to N programs;
2. Extract the best sequence from each training program;
3. Remove the duplicate sequences;
4. Remove from each sequence the optimizations that do not contribute to reduce the running time; and finally
5. Extract from the search space the best 10 sequences.

4 RESULTS

This section evaluates our hybrid approach that searches for good optimizations and their order, aiming at performance improvements. In other words, this section evaluates our approach that searches for sequences that outperform the optimization level `O3` in terms of running time.

Tables 4 and 5 present the results. In these tables `OT.15` means our approach created at most 15 centroids, `OT.35` our approach produced at most 35 centroids, `Best10` is the algorithm proposed by Purini and Jain, and `BestAll` means the maximum improvement available for compiling the test program with all sequences on `KB`.

Overview. Our approach is able to find sequences that outperform the well-engineered compiler optimization level `O3`, besides `Best10`. Only in five benchmarks (POLYBENCH.ADI, POLYBENCH.LUDCMP, CBENCH.LAME, CBENCH.PATRICIA, and CBENCH.SHA) `Best10` outperforms our approach. In some cases the two approaches have similar performance to `BestAll`. Our approach achieves the maximum available improvement on 26 programs, while `Best10` on 23 benchmarks. In addition, our hybrid approach outperforms `BestAll` on 5 programs: POLYBENCH.JACOBI-2D, POLYBENCH.LU, POLYBENCH.REG_DETECT, CBENCH.TIFF2BW, and CBENCH.PGP_E.

Metrics. `API` means that the gap between our approach and `BestAll` is less than the gap between `Best10` and `BestAll`. This gap is 16.82 %, 41.33 %, and 37.38 %, respectively for `OT.15`, `OT.35` and `Best10`, on POLYBENCH; and 26.42 %, 39.03 %

Benchmark	OT.15	OT.35	Best10	BestAll
2mm	11.72	1.72	11.71	11.75
3mm	12.47	12.47	12.46	12.48
adi	0.002	−18.14	3.75	3.75
atax	6.72	2.58	6.36	7.44
bicg	0.97	0.97	0.97	0.97
cholesky	14.22	14.21	14.26	33.24
correlation	12.42	12.36	12.33	12.44
covariance	12.45	12.44	12.44	12.47
doitgen	12.27	12.27	11.67	12.55
durbin	0.75	0.00	−3.28	3.51
dynprog	12.28	0.00	11.99	12.28
fdtd-2d	6.29	−19.87	−4.89	6.30
fdtd-apml	3.42	0.00	3.58	8.00
floyd-warshall	0.00	0.00	0.00	0.01
gemm	11.04	11.07	0.01	11.09
gemver	1.59	−1.93	−3.76	3.02
gesummv	0.003	−4.74	0.00	1.26
gramschmidt	6.23	6.23	0.01	6.24
jacobi-1d	4.72	0.00	0.00	7.26
jacobi-2d	9.99	15.77	3.96	3.96
ludcmp	0.01	0.01	22.07	22.13
lu	22.13	22.12	0.03	7.03
mvt	1.90	1.90	0.00	1.90
reg_detect	12.35	28.25	12.33	12.44
seidel-2d	41.67	41.75	41.33	41.69
symm	7.14	7.12	7.12	15.3
syr2k	0.00	0.00	0.00	0.01
syrk	0.00	0.00	0.02	0.02
trisolv	4.39	1.59	4.40	7.34
trmm	11.09	0.01	0.01	11.12
API	8.01	5.65	6.03	9.63
APIE	8.29	11.31	7.14	9.63
NBI	30	26	27	30

Table 4. The improvements on POLYBENCH

and 55.27 %, on cBENCH. It means that in general, these gaps are 21.18 %, 40.36 % and 45.37 %, respectively. `APIE` also indicates that our approach outperforms `Best10`. However, `NBI` indicates that the two approaches and `Best10` have a similar performance.

Benchmarks. The general results, mainly `API`, show that our approach and `Best10` perform better on POLYBENCH. It can be explained by the fact that POLYBENCH is composed of kernels, while cBENCH of complete programs. However,

Benchmark	OT.15	OT.35	Best10	BestAll
bitcount	−46.45	−46.45	−45.75	−10.77
qsort1	10.45	10.45	6.71	10.45
susan_c	30.88	30.88	27.04	32.48
susan_e	5.75	6.29	1.86	6.88
susan_s	1.06	0.97	0.78	1.06
bzip2d	49.25	49.25	39.97	49.26
bzip2e	5.73	5.50	4.39	7.03
lame	3.22	6.16	8.75	8.75
mad	2.45	1.99	1.30	2.19
tiff2bw	17.04	14.94	7.45	13.32
tiff2rgba	15.45	15.07	7.49	15.11
tiffdither	2.06	−0.77	0.12	1.13
tiffmedian	25.89	22.14	23.05	26.02
dijkstra	0.59	0.14	0.43	0.64
patricia	0.00	−1.08	0.56	0.56
rsynth	0.46	−1.83	0.38	0.46
stringsearch1	−15.92	−28.91	−18.24	−0.36
blowfish_d	4.12	4.18	4.12	4.18
blowfish_e	4.10	4.10	3.94	4.16
pgp_d	5.13	4.97	1.91	6.51
pgp_e	4.72	0.96	0.59	3.49
rijndael_d	1.97	3.25	0.003	3.54
rijndael_e	−0.32	−2.24	−2.21	−0.32
sha	6.95	6.95	7.605	7.66
adpcm_c	13.12	12.98	5.83	15.05
adpcm_d	16.46	15.08	11.18	16.58
CRC32	2.13	2.13	2.13	2.13
gsm	3.85	3.63	1.97	3.88
API	6.07	5.03	3.69	8.25
APIE	9.70	10.09	6.78	9.70
NBI	25	22	25	25

Table 5. The improvements on cBench

the performance loss on cBench is due to the three programs that BestAll does not outperform with optimization level O3, namely: cBench.bitcount, cBench.stringsearch1 and cBench.rijndael_e. If we remove these three benchmarks, the scenario changes. In this case, API is 9.31 %, 8.73 %, 6.78 % and 9.70 %, respectively for OT.15, OT.35, Best10 and BestAll. The gap decreases from BestAll, and also reinforces that using a hybrid approach is the best choice.

Best10. It is important to remember that our approach and `Best10` have different premises. The former argues that it is necessary to handle individual programs, which means that the choice of optimizations and their order is program-specific dependent. The latter argues that it is possible to cover several sets of programs using the same sequences. The results indicate that handling individual programs tends to decrease the gap between the strategy and the maximum available improvement, consequently enhancing the performance.

Evaluations. Using a strategy that creates several centroids ($[30, 35]$) outperforms `Best10`, however it increases the distance from the maximum available improvement. Although, creating few centroids ($[10, 15]$) increases the performance, this strategy expands the response time. The problem is that this strategy creates less centroids, grouping more programs on the same cluster. As a result, more sequences will be evaluated. `Best10` needs to evaluate only 10 sequences. The strategy that creates about 35 centroids needs to evaluate at most $20 * 2$ sequences (20 programs in the same cluster plus 20 new sequences after changing their order), while that one that creates about 15 centroids needs to evaluate at most $30 * 2$. This means that in terms of evaluations, `Best10` is better than our hybrid approach.

5 RELATED WORK

Cavazos et al. [4] proposed a machine learning strategy to find compiler optimizations for a specific program. This work was the first to use performance counters to measure the similarity between two programs. A machine learning strategy creates a prediction model in a training stage, based on the behavior of several training programs, and the prediction model, in a deployment (or test) stage, predicts the set of optimizations that will be enabled to compile the unseen program. In a training stage, Cavazos's strategy randomly creates several compiler optimization sets for a group of training programs. After the creation of several sets, their strategy collects the performance counters of each training programs. Based on these two pieces of information, a model based on a logistic regression scheme is created, which will predict the set of optimizations. The deployment stage collects the performance counters of the test program, invokes the prediction model, and finally returns the best target code.

They demonstrated that a machine learning strategy is able to outperform the compiler optimization levels. Furthermore, they also demonstrated that the use of performance counters is a good strategy to measure the similarity between two programs. Our strategy is similar to such method, because we also use a machine learning scheme and measure the similarity between two programs in the same manner. However, while Cavazos et al. tried to find optimizations, our work is one step further due to its searches and optimization orders.

De Lima et al. [7] proposed the use of a case-based reasoning strategy to find compiler optimizations for a specific program. They argue that it is possible to find good compiler optimizations, from previous compilations, for an unseen program. This strategy creates several compiler optimization sets in a training stage. Afterwards, in the deployment stage, the strategy infers a good compiler optimization set for a new program. This step is based on the similarity between two programs. De Lima et al. proposed several models to measure similarity, also based on feature vectors, which is composed by performance counters. They demonstrated that it is possible to infer a good compiler optimization set that achieves multiple goals; for example, runtime and energy. The limitation of this work is that it does not handle the problem of ordering optimizations.

Purini and Jain [24] proposed a strategy to find good compiler optimization sets, which are able to cover several programs. This means that they do not handle this problem as program-dependent. The strategy to find several sets consists in using random and genetic algorithms to create effective sets of optimizations. After creating several sets, they eliminate the optimizations, from each set, that does not contribute to the performance. Finally, they proposed an algorithm that analyzes all sets, and extracts the best 10 sets. As a result, each test program is compiled using 10 sets, and the best target code is returned. They demonstrated that it is possible to find a small group of sets that are able to cover several programs.

Our strategy uses Purini's and Jain's strategy to provide a knowledge base of good compiler optimizations sets; however, we handle the problem of finding good optimizations as a program-dependent problem.

Tartara and Crespi [26] proposed a long-term strategy, the goal of which is to eliminate the training stage. In their strategy, the compiler is able to learn, during every compilation, how to generate good target code. In fact, they proposed the use of a genetic algorithm that creates several heuristics based on the static characteristics of the test program [20]. Basically, this strategy performs two tasks. First, it extracts the characteristics of the test program. Secondly, the genetic algorithm creates heuristics inferring which optimizations should be enabled. They demonstrated that it is possible to eliminate the training stage, using long-term learning. While Tartara's and Crespi's work does not need a training stage, our work does; however, we handle two problems concerning compiler optimizations.

Martins et al. [17] proposed a clustering strategy in order to find good compiler optimizations sets. In fact, they proposed algorithms to find good optimizations, besides algorithms to order optimizations. The strategy used by Martins et al. is similar to Purini's and Jain's, both use random and genetic algorithms. This means that their strategy can be considered as an iterative compilation, where the test program is compiled with different sets of optimizations, and the best version is chosen. Our strategy is classified as a machine learning strategy, which tries to reduce the number of times that a test program needs to be evaluated.

6 CONCLUDING REMARKS

The selection of compiler optimizations and their order of application has a significant impact on performance. In addition, we need to remember that this problem is program-specific dependent. Therefore, a good approach is to propose a design-space exploration scheme to find the program-specific optimization sequence.

In this paper we proposed a design-space exploration scheme, which aims to find good compiler optimization sequences. Our approach employs several strategies, namely: random sampling, genetic algorithm, clustering, metaheuristic, and a reduction scheme.

We implemented a hybrid approach on top of the `LLVM` compiler, and the experiment results show that it finds optimization sequences that outperform the standard optimization level `O3`, besides the approach proposed by Purini and Jain, `Best10`.

The deficiency of our approach is the system response time, due to the number of evaluations. In a future work we will investigate a strategy to decrease the system response time. In addition, as programs are composed by several subroutines and each one will probably be best-optimized by a specific sequence, another future work will be to handle each subroutine.

REFERENCES

[1] Abdelbar, A. M.—Wunsch, D. C.: Improving the Performance of MAX-MIN Ant System on the TSP Using Stubborn Ants. Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO '12), New York, NY, USA, ACM, 2012, pp. 1395–1396, doi: 10.1145/2330784.2330949.

[2] Applegate, D. L.—Bixby, R. E.—Chvátal, V.—Cook, W. J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, 2007.

[3] Bertran, R.—Gonzalez, M.—Martorell, X.—Navarro, N.—Ayguade, E.: Decomposable and Responsive Power Models for Multicore Processors Using Performance Counters. Proceedings of the 24th ACM International Conference on Supercomputing (ICS '10), New York, NY, USA, ACM, 2010, pp. 147–158, doi: 10.1145/1810085.1810108.

[4] 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), Washington, DC, USA, IEEE Computer Society, 2007, pp. 185–197, doi: 10.1109/CGO.2007.32.

[5] The Collective Benchmarks, 2014, `http://ctuning.org/wiki/index.php/CTools:CBench`. Access: January 20, 2016.

[6] Chabbi, M. M.—Mellor-Crummey, J. M.—Cooper, K. D.: Efficiently Exploring Compiler Optimization Sequences with Pairwise Pruning. Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, New York, NY, USA, ACM, 2011, pp. 34–45, doi: 10.1145/2000417.2000421.

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

[8] Dongarra, J.—London, K.—Moore, S.—Mucci, P.—Terpstra, D.: Using PAPI for Hardware Performance Monitoring on Linux Systems. Proccedings of the Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute, 2001.

[9] Dorigo, M.—Stützle, T.: Ant Colony Optimization. Bradford Books, MIT Press, Cambridge, Massachusetts, 2004.

[10] Fang, S.—Xu, W.—Chen, Y.—Eeckhout, L.—Temam, O.—Chen, Y.—Wu, C.—Feng, X.: Practical Iterative Optimization for the Data Center. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 12, 2015, No. 2, pp. 15:1–15:26.

[11] Foleiss, J. H.—da Silva, A. F.—Ruiz, L. B.: An Experimental Evaluation of Compiler Optimizations on Code Size. Proceedings of the Brazilian Symposium on Programming Languages, São Paulo, Brazil, EACH USP, 2011, pp. 1–15.

[12] Kufrin, R.: PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux. Proceedings of the Linux Cluster Conference, Chapel, 2005.

[13] Lattner, C.—Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, California, March 2004, doi: 10.1109/CGO.2004.1281665.

[14] Lim, M. Y.—Porterfield, A.—Fowler, R.: SoftPower: Fine-Grain Power Estimations Using Performance Counters. Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10), New York, NY, USA, ACM, 2010, pp. 308–311, doi: 10.1145/1851476.1851517.

[15] LLVM Team. The LLVM Compiler Infrastructure, 2016, `http://llvm.org`. Access: January 20, 2016.

[16] Malone, C.—Zahran, M.—Karri, R.: Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs? Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing (STC '11), New York, NY, USA, ACM, 2011, pp. 71–76, doi: 10.1145/2046582.2046596.

[17] Martins, L. G. A.—Nobre, R.—Cardoso, J. A. M. P.—Delbem, A. C. B.—Marques, E.: Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 13, 2016, No. 1, pp. 8:1–8:28, doi: 10.1145/2883614.

[18] Mucci, P. J.—Browne, S.—Deane, C.—Ho, G.: PAPI: A Portable Interface to Hardware Performance Counters. Proceedings of the Department of Defense HPCMP Users Group Conference, 1999, pp. 7–10.

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

[20] Namolaru, M.—Cohen, A.—Fursin, G.—Zaks, A.—Freund, A.: Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. International Conference on Compilers Architectures and Synthe-

sis for Embedded Systems (CASES '10), Scottsdale, United States, 2010, doi: 10.1145/1878921.1878951.

[21] 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 (CGO '12), New York, NY, USA, ACM, 2012, pp. 196–206, doi: 10.1145/2259016.2259042.

[22] Park, E.—Kulkarni, S.—Cavazos, J.: An Evaluation of Different Modeling Techniques for Iterative Compilation. Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems, New York, NY, USA, ACM, 2011, pp. 65–74.

[23] Polybench. The Polyhedral Benchmark Suite. Access: March 2, 2014.

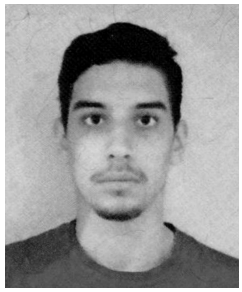
[24] Purini, S.—Jain, L.: Finding Good Optimization Sequences Covering Program Space. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 9, 2013, No. 4, pp. 56:1–56:23, doi: 10.1145/2400682.2400715.

[25] Srikant, Y. N.—Shankar, P.: The Compiler Design Handbook: Optimizations and Machine Code Generation. 2nd ed., CRC Press, Inc., Boca Raton, FL, USA, 2007.

[26] Tartara, M.—Crespi Reghizzi, S.: Continuous Learning of Compiler Heuristics. ACM Transactions on Architecture and Code Optimization (TACO), Vol. 9, 2013, No. 4, pp. 46:1–46:25, doi: 10.1145/2400682.2400705.

[27] Tavares, J.—Pereira, F. B.: Towards the Development of Self-Ant Systems. Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO '11), New York, NY, USA, ACM, 2011, pp. 1947–1954.

[28] Witten, I. H.—Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.



Tiago Cariolano de Souza Xavier got his B.Sc. and M.Sc. degrees in computer science from the State University of Maringá, Brazil, in 2012 and 2014, respectively. Nowadays, he is a Ph.D. student in computer science at the Federal University of Rio de Janeiro, Brazil. His research interest is in compilers and mobile systems.

Anderson Faustino da Silva is Professor in the Department of Informatics, State University of Maringá, Brazil, where he has been teaching since 2008. He received his B.Sc. degree in computer science from the State University of West Paraná, Brazil, in 2000, and his M.Sc. and Ph.D. degrees also in computer science from the Federal University of Rio de Janeiro, Brazil, in 2003 and 2006, respectively. His research interest is in compilers and parallel and distributed computing.