YaCoS: a Complete Infrastructure to the Design and Exploration of Code Optimization Sequences

André Felipe Zanella
DIN – UEM
Maringá, Paraná, Brazil
aft.zanella@gmail.com

Anderson Faustino da Silva
DIN – UEM
Maringá, Paraná, Brazil
anderson@din.uem.br

Fernando Magno Quintão Pereira
DCC – UFMG
Belo Horizonte, MG, Brazil
fernando@dcc.ufmg.br

Abstract

The growing popularity of machine learning frameworks and algorithms has greatly contributed to the design and exploration of good code optimization sequences. Yet, in spite of this progress, mainstream compilers still provide users with only a handful of fixed optimization sequences. Finding optimization sequences that are good in general is challenging because the universe of possible sequences is potentially infinite. This paper describes a infrastructure that provides developers with the means to explore this space. Said infrastructure, henceforth called YaCoS, consists of benchmarks, search algorithms, metrics to estimate the distance between programs, and compilation strategies. YaCoS's features let users build learning models that predict, for unknown programs, optimization sequences that are likely to yield good results for them. In this paper, as a case study, we have used YaCoS to find good optimization sequences for LLVM, using code size as the objective function. Such study lets us evaluate three feature sets: two variations of the feature vectors proposed by Namolaru at al in 2010, plus the optimization statistics produced by LLVM. Our results show that YaCoS is able to find sequences that improve onto clang -Oz by 3.75% on average. Our experiments do not indicate a dominant feature set out of the three approaches that we have investigated—it is possible to find programs in which one of them is strictly better than the others.

CCS Concepts • **Software and its engineering → Runtime environments**; **Compilers**; *Software libraries and repositories.*

Keywords Compilers, Optimizations, Exploration

SBLP 2019, September 23–27, 2019, Salvador, Brazil

ACM ISBN 978-1-4503-7638-9/19/09…$15.00
https://doi.org/10.1145/3355378.3355383

ACM Reference Format:
André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão Pereira. 2019. YaCoS: a Complete Infrastructure to the Design and Exploration of Code Optimization Sequences. In *XXIII Brazilian Symposium on Programming Languages (SBLP 2019), September 23–27, 2019, Salvador, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3355378.3355383

1 Introduction

Recent years have seen a surge in the popularity of machine learning techniques as a way to improve the quality of the code produced by optimizing compilers [3, 35]. In this context, machine learning works as a means to adapt compilers to programs. In other words, a compiler is trained onto a universe of known programs P_t, and, once given an unknown program $p \notin P_t$, it treats it like a program $p' \in P_t$ that is similar to p. This modus operandi has been shown to be effective along different dimensions of code efficiency, such as runtime [14, 21], energy consumption [24, 32], code size [4, 8], hardware usage [10, 27, 28], and the size-speed relation [30, 37], for instance.

This form of adaptive compilation relies on two assumptions. First, there exists a universe of programs P_t that is sufficiently large to cover most of the codes that a compiler is likely to find during its lifetime. Second, there exists a metric that measures the distance between programs, such that optimizations tend to behave in similar ways among codes that are close. Supporting such assumptions requires a comprehensive infrastructure, formed by a collection of training programs, plus a way to measure the distance between them. We believe that the best such infrastructure is the project MilePost-GCC [15, 16, 21].

However, in our opinion, this project has two shortcomings. First, it is restricted to the GCC compiler. Although a fundamental tool, today LLVM seems to enjoy more the support of the research community, and several adaptive techniques are now implemented in this compiler [3]. To support this statement, we notice that in PLDI'19, one of the main programming languages conference, there were five papers using LLVM, wheres no paper was using GCC. Second, we believe that MilePost suffers from a shortage of benchmarks. Currently, only cBench is distributed in this project[1].

[1]See https://ctuning.org/wiki/index.php/CTools:CBench

This collection of benchmarks comprises 32 programs, each one with 20 input sets. Individual research groups have been using MilePost with other benchmark suites, but these programs are not part of the project [7].

To address these shortcomings, this paper presents YaCoS, a infrastructure for the exploration of code optimization sequences built on top of the LLVM compiler. This infraestructure provides (i) a larger training set; (ii) a number of ways to characterize programs; (iii) different strategies to create training data; (iv) different metrics to measure distance between programs; and (v) mechanisms to implement new compilation strategies.

Case Study. To demonstrate how YaCoS is useful in the exploration of optimization space, in Section 4 we use it to test the following hypothesis: "*can we use the statistics produced by LLVM as an effective way to build feature vectors to measure the distance between programs?*" In this context, we define the distance between programs P_1 and P_2 as a metric of the effect of an optimization sequence S over them. If S leads to similar results in both programs, than we say that their distance is small; otherwise, we say that it is large. Such effects can be measured, for instance, as a percentage of code size reduction. LLVM provides developers with hundreds of statistics derived from the applications of optimizations. Said statistics include, for instance, the number of variables that have been transformed into constants, or hoisted outside loops. Feature vectors can be constructed out of such data, in a way that programs that yield similar vectors should be treated similarly by the compiler, i.e., being optimized by the same sequence of passes.

As we explain in Section 4, the LLVM statistics do not lead to results that are better than those already made possible by Namularu's feature sets. We have used 1,300 programs to train the optimizer, and have tested it onto 200 different benchmarks. The results obtained with LLVM's vectors or Namularu's are very close: both improve on clang -Oz; the former by 3.76%, the latter by 3.54%, considering code size as the objective function. The largest improvement observed with Namularu's features was 35.19%; the largest observed with LLVM's was 50.0%, but this event refers to an outlier in our dataset. Nevertheless, the goal of this paper is not to provide an alternative to Namularu's vectors—said endeavor would require a serendipitous combination of intuition and much experimentation. Rather, we want to show that YaCoS can be effectively used as a vehicle to explore the vast space of code optimization sequences.

2 An Overview On Predictive Compilation

Compilers apply optimizations onto programs. These optimizations seek to improve the program along different dimensions of efficiency: runtime, energy consumption and code size being the most common. Optimizations are applied in sequence. As an example, at its highest optimization level (-O3), clang[2] applies 85 different optimizations onto the target program. These optimizations invoke other analyses to gather the information that they require. Moreover, they can be applied multiple times, because the transformations carried out by an optimization might further enable the effects of another one. Therefore, such sequences of analyses and transformations can become enormous. In total, clang -O3 applies 255 analyses and optimizations passes over the same program.

The Default Optimization Levels of a Compiler. Although compilers provide developers with a number of different optimization sequences, each of them is typically fixed. As an example, clang provide developers with three optimization levels that aim to reduce program runtime: -O1, -O2 and -O3 (in addition to the default -O0, which performs no optimization). This compiler also provides two optimization levels to reduce code size: -Os and -Oz. Each one of these optimization levels applies onto the target program a number of compilation passes—analyses or optimizations—always in a fixed order.

The Need for Customization. Since long, researchers and compiler engineers have already realized that it pays off to treat programs differently; hence, applying on them customized optimization sequences [19]. In this regard, Fursin and Temam provides striking evidence that it is possible to find optimization sequences that greatly improve on the default optimization levels provided by clang [16]. From such observations, appeared the notion of *Predictive Compilation*. A predictive compiler uses properties of a program to decide how to optimize it. In this scenario, the compiler is trained onto a collection of known programs to derive a model which determines its actions in face of unknown codes. The collection formed by all the known programs is called the *Training Set*. Predictions, in this context, consist in matching *program properties*, also called *features*, with *compilation actions*.

Program Characterization. Program features can be dynamic or static. Dynamic features are derived from observable effects of the runtime behavior of a program. Usually, hardware performance counters are used to produce dynamic program features [23, 24]. Static program features are derived from the program's syntax. Examples include countable data such as number of loops, number of constant variables, ratio between integer and floating point operations, etc. In this paper, we restrict our discussion to static program features; as they can be deterministically obtained from the syntactic analysis of a program's code. Static program features have been explored in several previous works [14, 21, 26]. Much of the challenge related to this kind of research consists in choosing a good set of static features. Unrepresentative features might complicate prediction of optimization sequences, inducing the construction of models that are not adequate representations of programs.

[2]Version 10.0.0

Program Distance. As mentioned before, a predictive compiler tries to match an unknown program with known codes; hence, treating that unknown program in the same way as it treats the known ones. Therefore, a good notion of program similarity is essential for predictive compilation. There are several ways to measure the distance between programs. A common technique is to group features into vectors. Said vectors contain one dimension for each available feature. The combination of program features used to characterize programs form the *program space*—a vector space formed by feature vectors. Once vectors are built, metrics such as Euclidean, Chebyshev, Manhattan or Hamming distance can be used to measure the closeness between programs.

Example 2.1 (Program Space)**.** Figure 1 (a) shows the program space formed by three static features: number of stores, number of loops in the most nested loop and number of instructions. Programs can be placed within this space based on the values of their features. The example highlights two programs, bound to vectors (88, 2, 191), and (93, 4, 234).

Example 2.2 (Optimization Sequence)**.** Figure 1 (a) shows two sequences of optimizations. The sequence `tc-ls-sz-lc-lr` has been found to be efficient to the program associated with feature vector (88, 2, 191), and the sequence `ls-lc-lr` has been found to be good for the program with feature vector (93, 4, 234).

Example 2.3 (Prediction)**.** Figure 1 (b) shows how four known programs have contributed to determine an optimization sequence for an unknown program. The unknown program has feature vector (25, 1, 75). The Euclidean distance is used to find the K nearest neighbors of this unknown program. We consider $K = 4$. Once these similar programs are found, we have selected the most common subsequence within the four known sequences to associate `tc-ls-lr` with the unknown program.

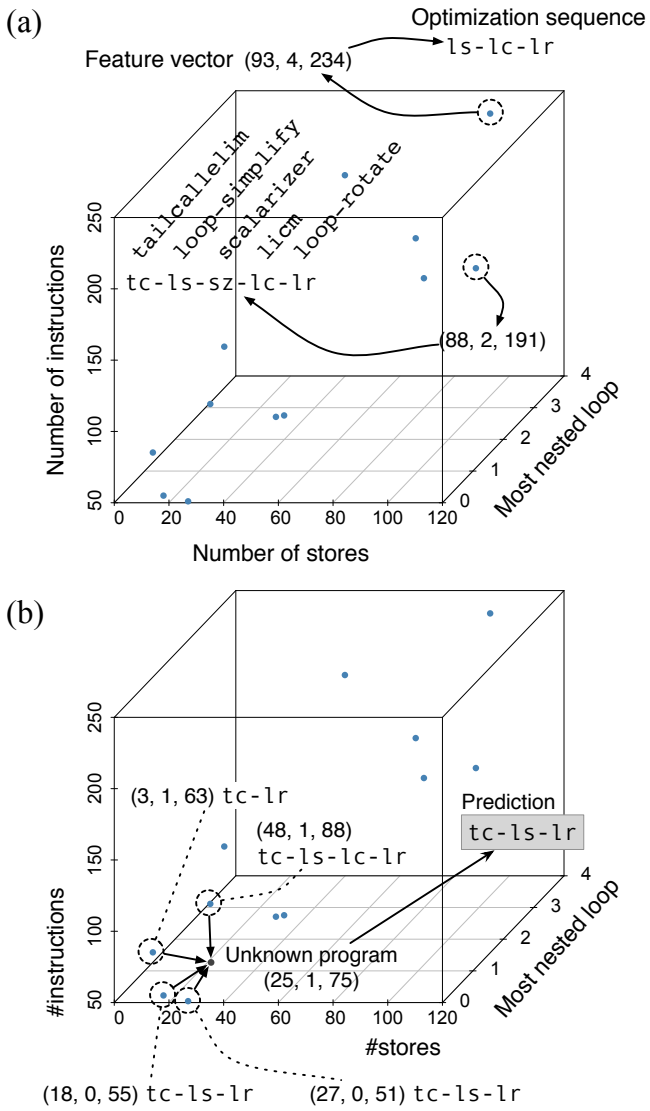

Figure 1. (a) Optimization space made of three static program features, plus two feature vectors and one optimization vector. (b) Known programs being used to determine the best optimization vector for an unknown program.

3 Exploration Infrastructure

This section describes YaCoS, the infrastructure that we have engineered to allow the exploration of code optimization sequences. Such infrastructure consists of five parts: a suite of program features that can be collected via syntactic analyzers and dynamic tools (Section 3.1); an ensemble of distance metrics (Section 3.2); a search engine parameterized by different algorithms (Section 3.3); a number of compilation strategies that retrieves an optimization sequence to an unknown program based on its similarity to known programs (Section 3.5); and a number of benchmarks that can be used as training and test sets (Section 3.6).

3.1 Program Features

YaCoS provides users with the necessary equipment to extract feature vectors from programs. This infrastructure builds static and dynamic feature vectors. In the rest of this section, we explain which vectors are currently available in YaCoS. Notice that in addition to the necessary infrastructure to extract features, YaCoS provides some machinery to manipulate feature vectors. Possible manipulations include concatenating vectors, finding the most relevant component, applying PCA, and such.

3.1.1 Static Features

To map programs into static feature vectors, YaCoS provides users with a collection of LLVM passes. The following feature sets are currently available.

MSF: 56 features proposed by Fursin *et al.* [15]. These features encompass characteristics of a program's CFG, such as number of nodes and edges, and of the program's instructions, such as number of arithmetic and logic instructions.

LOOP: 17 features proposed by Park *at al* [26]. Feature vectors are composed of data related to loops and control dependencies. Examples include loop carried dependencies (read-write, read-read, etc.) and loops with data dependencies like read-after-write, write-after-read and write-after-write.

LLB: 70 features derived from the LLVM bitcode analyzer. We call an LLVM bitcode an encoding of the LLVM IR in a stream of bits. This encoding's structure includes elements such as blocks and data records.

LLS: 52 features derived from LLVM statistics. Each analysis and optimization used in LLVM provides developers with statistical information, which can be optionally printed as the result of the execution of that pass. As an example, the constant propagation pass lets developers know how many instructions have been eliminated due to folding.

DNA: 63 features following the representation proposed by Filho *et al.* [14]. Each instruction of the LLVM IR is encoded as a gene with a unique symbolic representation.

3.1.2 Dynamic Features

To extract dynamic feature vectors from programs, YaCoS provides users with a number of PinTools. A PinTool is a plugin that can be loaded by the Pin binary instrumentation framework of Intel [20].

CAT: 23 numerical features, each representing the instruction types from the hardware instruction set.

JUMP: 6 features, each representing a type of jump instruction.

LDST: 5 features representing different kinds of load and store memory instructions.

MICA: a 116-dimensional microarchitecture-independent feature vector, which represent the hardware behaviour, as proposed by Hoste *et al.* [18].

PEV: 45 features (events) obtained by the Linux profiling tool *perf*.

PCS: 50 features based on performance counters obtained via *papi* [13].

3.2 Distance Metrics

The distance between two programs is a measure of similarity among them. Programs that are close should be treated in a comparable manner by the predictive compiler. There exist different distance metrics between programs, and YaCoS presently recognizes three of them.

MCoeff: this metric, proposed by Filho *et al* [14], measures the distance between two programs as the effect of a sequence of optimizations onto these programs. The behavior is understood as the performance speedup or slowdown provided by the optimizations, in case runtime is the objective function, or code size reduction or increase, in case code size is the goal.

DNA alignment: this metric, also proposed in the work of Filho *et al* [14], treats feature vectors as DNA sequences. It then uses the *Needleman-Wunsch* algorithm [22] to find an alignment of these sequences. The alignment determines a score that indicates the similarity between the two programs.

Distance (Cosine, Euclidean, Manhattan): These are standard geometric notions of distance, as computed over two feature vectors. The *Manhattan* distance or L_1 norm is the preferable distance metric when the feature vectors have high-dimensionality [2].

Notice that each one of these three categories of distance uses different inputs in the distance function. MCoeff compares the effect of a given sequence of compiler optimizations onto two programs. The DNA function receives two programs represented as a sequence of genes. These genes are determined by the instructions found after aligning each program's CFG. The other distance functions are calculated from the feature vectors that represent each program.

3.3 Search Engine

The search engine is used to associate good optimization sequences with programs in the training set. In other words, this is the algorithm that builds a database mapping known programs to optimization sequences that are good for those programs. The following search algorithms are presently supported:

- Pygmo *et al*'s particle swarm optimization and genetic algorithm [6];
- Lima *et al*'s improved batch elimination [11].
- Xavier *et al*'s phase ordering approach [12];
- Wang *et al*'s active learning approach [25];
- Lakshya's bestK and sequence reduction algorithms [29];
- Tonetti *et al*'s case-based reasoning [31];
- Filho *et al*'s case-based reasoning [14];
- Pan *et al*'s batch, iterative and combined elimination [38].

YaCoS provides a collection of adjustment parameters for most of the search algorithms. For instance, the genetic search algorithm is parameterized by its definition of population, generation, dimension and training data.

3.4 Objective Functions

The different search strategies can be guided by either static of dynamic objective functions. The static functions are: binary size (in bytes of objective file), code size (in number of LLVM instructions), and compilation time. The dynamic

functions are computed by running the program and profiling its behavior with *perf*, *llvm-mca* or *hyperfine*. These functions include: number of cycles, number of instructions, `llvm-cycles-mca`, `llvm-instructions-mca`, `llvm-nof-inst`, `llvm-runtime-mca` and `runtime`.

Additionally, the user can assign a weight value in the interval $[0, 1]$, to each goal, to create a multi-objective goal. For example, $goal = (runtime = 0.6, compile\_time = 0.4)$. In terms of engineering, we have strived to allow easy integration of new user-defined algorithms. Our intention is to allow users to add either new search algorithms or new objetive functions, without having to modify the other parts of YaCoS.

3.5 Compilation Strategy

A compilation strategy determines which optimization sequences are applied onto the unknown program P_u. These sequences are drawn from the set of sequences associated with the known program (or programs) that are the closest to P_u according to some distance metric (see Section 3.2). To this effect, we let $S_e \subseteq S_k$ be a subset of S_k ($|S_k| \in \mathbb{Z}$) formed by the lists that yield better code than the baseline when applied onto the known program P_k. The notion of better code is parameterized by an objective function. In this paper, we focus on size, for instance.

- **Elite:** We apply on P_u every "good" sequence in S_e, and keep the best result[3].
- **J*X*,** $X \in \mathbb{Z}, X > 0$**:** we apply on P_u **J**ust the X best sequences in S_k, and keep the best result.
- **G*X*,** $X \in \mathbb{Z}, X > 0$**:** we apply on P_u only the best X "**G**ood" sequences in S_k, and keep the best result. When not enough sequences are present in S_e, we look for sequences in P' that is the program the closest to P_u, after P_k.

3.6 Benchmarks

YaCoS presently contains two benchmark suites, which can be used either in combination or separately. The first suite consists of 5,000 C benchmarks mined from open-source repositories. These benchmarks are compilable, and yield an objective file when submitted to either clang or gcc; however, they are not executable. The second suite consists of 300 executable, non-parallel, benchmarks from the LLVM test-suit. Each benchmark in this group has at least one standard input; a few have more. Table 2 summarizes the different populations of programs found in these benchmarks.

4 Case Studies

This section illustrates how YaCoS can be used. To this end, we shall employ this infrastructure to setup three experiments. These experiments only differ with regards to the features used to represent programs: MSF, MSFL (MSF+LOOP) or LLS. For a description of these feature sets, see Section 3.1. Thus, except for the program features, the experiments share the following parameters:

- *Training Set*: 1,200 programs drawn from AnghaBench.
- *Test Set*: 300 programs programs drawn from AnghaBench.
- *Compilation Strategy*: J1, J10, J100, G1, G10, G100 and Elite.
- *Baseline*: clang -Oz, stripped from debugging information for maximum code compression.
- *Objective function*: size of the program's intermediate representation, measure in number of LLVM instructions.

	Angha		LLVM test-suite	
	#Insts	#Funcs	#Insts	#Funcs
max	2,218	1	213,354	9916
mean	401.49	1.0	7,859.78	99.52
median	342.0	1.0	1,072.0	13.0
min	256	1	42	1

Figure 2. Instructions and number of functions in the benchmarks currently distributed with YaCoS.

Experimental Setup. The evaluation reported in this section has been performed on a Intel Xeon E5-2650 (2.30 GHz, 32G RAM, 12 cores and hyper-threading) with Debian 10. Feature extraction and compilation has been carried in LLVM version 10.0.0. In total, the results reported in this section involved 288 hours of computational time.

4.1 Evaluation of the MSF Feature Set

Relative Numbers. Figure 3 shows results for this feature set. The table on the top (Full) summarizes results for all the 300 programs in the test set. The table on the bottom (Positive) shows results only for those programs in which YaCoS has been able to find a sequence of optimizations that improves on the baseline clang -Oz. The next figures (Figs. 5 and 6) also organize results in this way. In its best setup, G100, YaCoS parameterized with MSF has been able to find better optimization sequences than the baseline (clang -Oz) in 115 programs.

Absolute Numbers. The best observed improvement, in percentage, was 35.18%. To provide the reader with some perspective on this value, Figure 4 shows the absolute variation in the number of instructions observed in this experiment. The difference of 35.18% corresponds to a reduction of 254 instructions in one of our benchmarks.

Negative results and suggested modus operandi. Size variations are, on average, negative, as Figure 3 (top) shows. The baseline, clang -Oz, is a well-established optimization sequence, built throughout many years after the combined effort of many experts. Nevertheless, in several programs YaCoS has been able to outperform it. For instance, MSF +

[3] A good sequence is a sequence that improves on the baseline.

Full	J1	J10	J100	G1	G10	G100	Elite
Max	21.98	35.18	35.18	21.98	35.18	35.18	35.19
Mean	-7.19	-4.08	-3.25	-7.69	-4.4	-3.48	-4.99
Median	-1	0	0	-1.37	0	0	0
Min	-979.2	-979.2	-979.2	-979.2	-979.2	-979.2	-979.2
Positive	J1	J10	J100	G1	G10	G100	Elite
Progs	54	103	122	57	97	115	83
Max	21.98	35.19	35.19	21.98	35.19	35.19	30.19
Mean	3.03	3.29	3.58	3.14	3.62	3.75	3.76
Median	1.69	1.53	1.69	1.61	1.66	1.66	1.69
Min	0.25	0.25	0.21	0.25	0.25	0.21	0.21

Figure 3. Percentage of code size variation obtained after predictive compilation based on the MilePost GCC Static Feature Set (MSF).

Full	J1	J10	J100	G1	G10	G100	Elite
Max	91	125	254	91	125	254	125
Mean	-8.23	-1.8	0.59	-9.23	-2.26	-0.01	-0.21
Median	-2	0	0	-3	0	0	0
Min	-235	-235	-235	-235	-235	-235	-235
Positive	J1	J10	J100	G1	G10	G100	Elite
Progs	54	103	122	57	97	115	83
Max	91	125	254	91	125	254	125
Mean	8.37	8.36	10.52	8.07	9.52	10.87	10.03
Median	3.5	3	4	3	3	4	4
Min	1	1	1	1	1	1	1

Figure 4. Absolute code size variation obtained after predictive compilation based on the MilePost GCC Static Feature Set (MSF).

G100 found better optimization sequences for 115 programs, yielding an average improvement (restricted to these programs) of 3.75%. Thus, when used in tandem with YaCoS, a mainstream compiler, in this case clang, can opt for either using its default optimization sequence, or that suggested by our framework.

4.2 Evaluation of the MSFL Feature Set

Figure 5 shows compilation results obtained with the combination of the MilePost features plus the Loop Static Features mentioned in Section 3.1. Results are very similar to those observed in our first experiment. When we analyze the entire test set, we observe slightly worse averages than those seen in Section 4.1. However, when we analyze only the programs in which improvements have been observed over the baseline, MSFL yields slightly better results—at least when we look into the number of programs and medians. Nevertheless, this experiment tends to indicate that the extra features are not improving on the predictions that can already be performed via the standard feature set proposed by Namolaru [21].

Full	J1	J10	J100	G1	G10	G100	Elite
Max	21.98	35.18	35.19	21.98	30.19	31.02	30.19
Mean	-7.18	-4.08	-2.86	-7.06	-4.16	-2.95	-4.83
Median	-1	0	0	-0.84	0	0	0
Min	-979.2	-979.2	-979.2	-979.2	-979.2	-979.2	-979.2
Positive	J1	J10	J100	G1	G10	G100	Elite
Progs	63	111	131	67	106	125	83
Max	27.31	35.19	35.19	21.98	30;19	31.02	30.19
Mean	3.22	3.3	3.55	2.9	3.5	3.71	3.49
Median	1.61	1.53	1.68	1.46	1.67	1.7	1.91
Min	0.25	0.25	0.21	0.25	0.23	0.21	0.21

Figure 5. Percentage of code size variation obtained after predictive compilation based on the MilePost GCC Static Feature Set augmented with Loop features (MSFL).

4.3 Evaluation of the LLS Feature Set

Figure 6 shows the results of predictive compilation using the new feature set proposed in this paper: the statistics produced by LLVM optimizations. This experiment indicates that the LLS feature set does not improve on the other feature sets already evaluated in this paper, namely in Sections 4.1 and 4.2. When we analyze programs in which positive improvements have been found over the baseline, results are practically equivalent to those earlier reported. In all the cases, we have been able to achieve a code size improvement of about 3.5% over the baseline clang -Oz.

Full	J1	J10	J100	G1	G10	G100	Elite
Max	35.18	35.18	50	19.9	35.19	50	50
Mean	-6.78	-3.86	-2.94	-7.3	-4.2	-3.2	0.02
Median	-1.19	0	0	-1.34	0	0	0
Min	-979.2	-979.2	-979.2	-979.2	-979.2	-979.2	-38.14
Positive	J1	J10	J100	G1	G10	G100	Elite
Progs	61	106	122	61	100	122	82
Max	35.18	35.19	50	19.9	35.19	50	50
Mean	3.98	3.09	3.65	2.57	3.31	3.54	3.54
Median	1.24	1.29	1.58	1.27	1.23	1.51	1,44
Min	0.23	0.19	0.29	0.37	0.19	0.29	0.29

Figure 6. Percentage of code size variation obtained after predictive compilation based on the “LLVM-Stats" Static Feature Set (LLS).

5 Related Work

Space search exploration is an active topic in the programming language community. Much of this popularity is owned to the growing importance of machine learning within computer science. For an overview of the impact of machine learning onto compiler construction, we recommend surveys from Wang and O'Boyle [35] and Ashouri *et al* [3]. The *training phase* of a predictive compiler consists in a search, not necessarily exhaustive, for the most adequate compilation action for each program in the training set. The notion of "most adequate action" depends on two factors: (i) the objective function that guides the search; and (ii) the representation of the action. Typical objective functions include runtime, size and energy consumption. Common representations include tuples and lists of optimizations. In the former case, the order of application of an optimization is fixed—what varies is the occurrence or not of the optimization [14, 15]. In the latter, any permutation and/or combination of a known universe of optimizations is acceptable [1, 33]. YaCoS deals with the second representation, as it subsumes the first.

MilePost GCC. One of the most well-established efforts in the field of predictive compilation for the C programming language is the MilePost GCC project [15, 16]. Researchers involved in this project have assembled a training set of programs, and have extracted static features to represent each program. Training data is collected by compiling programs in the training set with varying sequences of optimizations, and recording how each sequence performs. Different machine learning models use the knowledge acquired during training to predict which sequence to use when optimizing an unseen program. Milepost GCC has been used to optimize for runtime and code size. In our opinion, a major shortcoming of the project is a perceived lack of benchmarks. The main benchmark suite used in MilePost contains 32 programs (the `cBench` suite). Another difference to this project is the underlying compilation infrastructure: whereas MilePost is built over GCC, we chose LLVM. Our choice is pragmatic: LLVM has a much lower entry curve than GCC.

On the Construction of Benchmark Suites. To circumvent the obstacle previously mentioned, posed by the lack of benchmarks, compiler researchers resort to program generation. With such purpose, automatically constructed programs have been used to tune compiler heuristics in specific scenarios [5, 32, 36]. However, these programs cannot be easily employed in general purpose compilers: they consist of micro-kernels that exercise particular aspects of the target hardware or of the target programming language. There exist tools that generate synthetic programs in different programming languages. One of the most recent elements in this family is DeepSmith [9], an evolution of Cummins *et al.*'s CLGen [10]. DeepSmith has been shown to be able to produce impressively realistic OpenCL programs. Nevertheless, in spite of early success, recently Goens *et al*, with collaboration of Cummins himself, have shown that CLGen still struggles to generate benchmarks that are good representatives of real-world programs [17].

6 Conclusion

This paper has presented YaCoS, an infrastructure designed to explore the space of code optimization sequences. This framework provides developers with benchmarks, functions to compute program distance, algorithms to associate optimization sequences with programs, and several search strategies, that let developers find local optima in the universe of optimization sequences that apply onto a program. To demonstrate how YaCoS is effective and useful to the tuning of predictive compilers, we have used it to test three different feature sets. These feature vectors are used to represent programs within prediction models. One of these feature sets, based on the statistics produced by LLVM optimizations, is an original contribution of this work. It does not outperform standard feature sets, such as those proposed by Namolaru in 2010; however, it demonstrates YaCoS's versatility, but letting us explore with ease new forms to represent programs.

Future directions. It is our intention to maintain YaCoS, and grow its community, in a way similar to what has happened to the MilePost project [15]. With this goal, we would like to use YaCoS to test new prediction models. A promising path along this direction seems to be the possibility of modeling programs as graphs—an approach recently proposed by Wang *et al.* [34]. Our intuition is that such representation might fit well the nature of code, given that control flow graphs are a ubiquitous program format within compilers.
Software: YaCoS is publicly available [4] [5]. The repository `YaCoS.Benchmarks` contains the benchmarks and datasets used in this paper.

Acknowlegment

This work is supported by different grants from CNPq, CAPES and FAPEMIG. We thank our colleagues from the Compilers Lab for reviewing an early version of this paper. We thank the SBLP referees for comments and suggestions that greatly improved the quality of this work.

References

[1] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. 2006. Using Machine Learning to Focus Iterative Optimization. In *CGO*. IEEE Computer Society, Washington, DC, USA, 295–305. https://doi.org/10.1109/CGO.2006.37

[2] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. 2001. On the Surprising Behavior of Distance Metrics in High Dimensional Spaces. In *ICDT*. Springer-Verlag, Berlin, Heidelberg, 420–434.

[4] https://github.com/AndersonFaustino/YaCoS
[5] https://github.com/AndersonFaustino/YaCoS.Benchmarks

[3] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning Using Machine Learning. *Comput. Surv.* 51, 5 (2018), 96:1–96:42. https://doi.org/10.1145/3197978

[4] Sorav Bansal and Alex Aiken. 2008. Binary Translation Using Peephole Superoptimizers. In *OSDI*. USENIX Association, Berkeley, CA, USA, 177–192.

[5] Rajkishore Barik, Naila Farooqui, Brian T. Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A Black-box Approach to Energy-aware Scheduling on Integrated CPU-GPU Systems. In *CGO*. ACM, New York, NY, USA, 70–81. https://doi.org/10.1145/2854038.2854052

[6] Francesco Biscani and Dario Izzo. 2020. esa/pagmo2: pagmo 2.15.0. https://doi.org/10.5281/zenodo.3738182

[7] Craig Blackmore, Oliver Ray, and Kerstin Eder. 2017. Automatically Tuning the GCC Compiler to Optimize the Performance of Applications Running on the ARM Cortex-M3. arXiv:1703.08228 http://arxiv.org/abs/1703.08228

[8] Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. 2017. Learning to superoptimize programs. In *ICLR*. OpenReview, Toulon, France, Article 1, 14 pages.

[9] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *ISSTA*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[10] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. IEEE, Piscataway, NJ, USA, 86–99.

[11] E. Daniel de Lima and A. Faustino da Silva. 2015. Improved batch elimination: A fast algorithm to identify and remove harmful compiler optimizations. In *2015 Latin American Computing Conference (CLEI)*. IEE, Arequipa, Peru, 1–8.

[12] Tiago de Souza Xavier and Anderson da Silva. 2018. Exploration of Compiler Optimization Sequences Using a Hybrid Approach. *COMPUTING AND INFORMATICS* 37, 1 (2018), 165–185. http://www.cai.sk/ojs/index.php/cai/article/view/2018_1_165

[13] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. 2001. Using PAPI for Hardware Performance Monitoring on Linux Systems. In *In Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute*. LCI, New Mexico, USA, 1–11.

[14] João Fabrício Filho, Luis Gustavo Araujo Rodriguez, and Anderson Faustino da Silva. 2018. Yet Another Intelligent Code-Generating System: A Flexible and Low-Cost Solution. *J. Comput. Sci. Technol.* 33, 5 (2018), 940–965. https://doi.org/10.1007/s11390-018-1867-7

[15] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International journal of parallel programming* 39, 3 (2011), 296–327.

[16] Grigori Fursin and Olivier Temam. 2010. Collective Optimization: A Practical Collaborative Approach. *Trans. Archit. Code Optim.* 7, 4 (2010), 20:1–20:29. https://doi.org/10.1145/1880043.1880047

[17] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. 2019. A Case Study on Machine Learning for Synthesizing Benchmarks. In *MAPL*. ACM, New York, NY, USA, 38–46. https://doi.org/10.1145/3315508.3329976

[18] Kenneth Hoste and Lieven Eeckhout. 2007. Microarchitecture-independent workload characterization. *IEEE micro* 27, 3 (2007), 63–72.

[19] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. 2003. Finding Effective Optimization Phase Sequences. In *LCTES*. ACM, New York, NY, USA, 12–23. https://doi.org/10.1145/780732.780735

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, USA, 190–200.

[21] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. 2010. Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *CASES*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/1878921.1878951

[22] S. B. Needleman and C. D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48 (1970), 443–453.

[23] Rajiv Nishtala, Paul M. Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *HPCA*. IEEE, New York, NY, USA, 409–420.

[24] Marcelo Novaes, Vinicius Petrucci, Abdoulaye Gamatié, and Fernando Magno Quintão Pereira. 2019. Compiler-assisted adaptive program scheduling in big.LITTLE systems: poster. In *PPoPP*. ACM, New York, NY, USA, 429–430. https://doi.org/10.1145/3293883.3301493

[25] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. 2017. Minimizing the cost of iterative compilation with active learning. In *CGO*. IEE, Austin, TX, USA, 245–256.

[26] Eunjung Park, Christos Kartsaklis, and John Cavazos. 2014. HERCULES: Strong Patterns towards More Intelligent Predictive Modeling. In *ICPP*. IEEE, New York, NY, USA, 172–181. https://doi.org/10.1109/ICPP.2014.26

[27] Fernando Magno Quintão Pereira, Guilherme Vieira Leobas, and Abdoulaye Gamatié. 2018. Static Prediction of Silent Stores. *ACM Trans. Archit. Code Optim.* 15, 4, Article 44 (2018), 26 pages. https://doi.org/10.1145/3280848

[28] Gabriel Poesia, Breno Campos Ferreira Guimarães, Fabricio Ferracioli, and Fernando Magno Quintão Pereira. 2017. Static placement of computation on heterogeneous devices. *PACMPL* 1, OOPSLA (2017), 50:1–50:28.

[29] Suresh Purini and Lakshya Jain. 2013. Finding Good Optimization Sequences Covering Program Space. *ACM Trans. Archit. Code Optim.* 9, 4, Article 56 (Jan. 2013), 23 pages. https://doi.org/10.1145/2400682.2400715

[30] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In *CGO*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

[31] Marcos Yukio Siraichi, Caio Henrique Segawa Tonetti, and Anderson Faustino da Silva. 2019. Pinhão: An Auto-tunning System for Compiler Optimizations Guided by Hot Functions. *j-jucs* 25, 1 (2019), 42–72. http://www.jucs.org/jucs_25_1/pinhao_an_auto_tunning

[32] Jyothi Krishna Viswakaran Sreelatha, Shankar Balachandran, and Rupesh Nasre. 2018. CHOAMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors. *Trans. Multi-Scale Computing Systems* 4, 2 (2018), 163–176.

[33] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *PLDI*. ACM, New York, NY, USA, 77–90. https://doi.org/10.1145/781131.781141

[34] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *ArXiv* abs/1909.01315 (2019), 1–7.

[35] Zheng Wang and Michael F. P. O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901. https://doi.org/10.1109/JPROC.2018.2817118

[36] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. 2014. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous

platforms. In *HiPC*. IEEE, Los Alamitos, CA, USA, 1–10. https://doi.org/10.1109/HiPC.2014.7116910

[37] Peng Zhao and José Nelson Amaral. 2003. To Inline or Not to Inline? Enhanced Inlining Decisions. In *LCPC*. Springer, Heidelberg, Germany, 405–419.

[38] Zhelong Pan and R. Eigenmann. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *International Symposium on Code Generation and Optimization (CGO'06)*. IEE, New York, NY, USA, 12 pp.–332.