View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidade Tecnológica Federal do Paraná (UTFPR)

Evaluating Knowledge Representations for Program Characterization

João Fabrício Filho[1,2], Luis Gustavo Araujo Rodriguez[1] and Anderson Faustino da Silva[1]
[1]*Universidade Estadual de Maringá, Maringá, PR, Brazil*
[2]*Universidade Tecnológica Federal do Paraná, Campo Mourão, PR, Brazil*

Keywords: Knowledge Representation, Program Representation, Reasoning System, Compiler, Code Generation.

Abstract: Knowledge representation attempts to organize the knowledge of a context in order for automated systems to utilize it to solve complex problems. Among several difficult problems, one worth mentioning is called *code-generation*, which is undecidable due to its complexity. A technique to mitigate this problem is to represent the knowledge and use an automatic reasoning system to infer an acceptable solution. This article evaluates knowledge representations for program characterization for the context of code-generation systems. The experimental results prove that program Numerical Features as knowledge representation can achieve 85% near to the best possible results. Furthermore, such results demonstrate that an automatic code-generating system, which uses this knowledge representation is capable to obtain performance better than others code-generating systems.

1 INTRODUCTION

Knowledge representation is a field of artificial intelligence dedicated to represent the knowledge of a specific context in order for it to be utilized to create formalisms and, thus, solve complex problems.

A complex problem, worth mentioning in the computer science field, is to generate good target code. This is due primarily for the characteristics of the source program (Aho et al., 2006).

A technique to mitigate the said problem is to represent the knowledge of the source program, and design a formalism that can be utilized by automatic reasoning systems to infer an acceptable solution for the code-generation problem. However, automated systems that attempt to mitigate this problem (de Lima et al., 2013; Tartara and Reghizzi, 2013; Queiroz Junior and da Silva, 2015) do not validate the utilized formalism.

This article aims to find a good knowledge representation that can be used to mitigate the code-generation problem.

The main contributions of this article are: (1) the description of knowledge representations to characterize programs; (2) the presentation of a technique to evaluate such representations; and (3) the demonstration of the use of a good representation.

The results prove that the Numerical Features representation is capable to obtain results 85% near to the best possible results of a knowledge base. Furthermore, when this representation is utilized by a code-generating system, it is able to obtain a target code with better performance than those acquired by `Best10` and `GA10` techniques.

2 CODE-GENERATING SYSTEM

The purpose of a code-generating system is to produce target code for a specific hardware architecture, through a source code (Aho et al., 2006). This process, which is divided into several phases, applies diverse transformations to the source code in order to improve the quality of the target code. However, finding a good transformation sequence for a particular program is a complex task, especially due to the size of the search space.

A way to mitigate this problem is to extract knowledge from the code-generating system and build a formalism capable of supporting the selection of the transformation sequence for the respective program.

A simple technique to extract knowledge is to apply a training phase for the system, in which various programs are compiled with different transformation sequences. After this process, it is possible to extract good sequences and their respective programs.

Based on this formalism (good transformation se-

Filho, J., Rodriguez, L. and Silva, A.
Evaluating Knowledge Representations for Program Characterization.
DOI: 10.5220/0006333605820590
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 1*, pages 582-590
ISBN: 978-989-758-247-9
Copyright © 2017 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

quences and their respective programs), to implement an efficient code-generation turns into a problem of identifying similar programs.

3 PROGRAM CHARACTERIZATION

Computer programs can be represented by dynamic or static characteristics, which assist in parameterizing the code-generating system. Dynamic characteristics describe the program behavior in regards to its execution. On the other hand, static characteristics describe the algorithmic structures of the program.

The appeal of dynamic characteristics is that it considers both the program and hardware characteristics. However, this provides a disadvantage due to being platform-dependent and, thus, incurring the need for program execution.

Alternatively, static characteristics are platform-independent and do not require program execution. However, such representation does not consider the program-input data, which is an element that can alter the behavior and consequently cause parameter alterations of the code-generating systems.

Among the program representations presented in the literature, this article evaluates the following:

- Dynamic
  1. **Performance Counters (PC):** are characteristics resulting from the program execution and consists in the hardware performance counters that are available. Various works utilized PC as a program-representation scheme (Cavazos et al., 2007; de Lima et al., 2013; Queiroz Junior and da Silva, 2015). This article specifically evaluates the characteristics described in (de Lima et al., 2013) and (Queiroz Junior and da Silva, 2015).
- Static
  1. **Compilation Data (CD):** are characteristics that describe the relationships between the program entities, defined by both the intermediate representation utilized by the code-generating system, as well as the respective hardware architecture. These characteristics was proposed by Queiroz and da Silva (Queiroz Junior and da Silva, 2015), and their usage is limit by the data provided by the compiler even though a direct relationship with the source code exists.
  2. **Numerical Features (NF):** are characteristics extracted from the relationships between program entities, which are defined by the specificities of the programming languages. They were proposed by Namolaru *et al.* (Namolaru et al., 2010) and were systematically produced experimentally. Namolaru *et al.* proved their influence in parameterizing code-generating systems.

In addition to these representations, this paper proposes a symbolic representation, similar to a DNA, in which each intermediate language instruction, used by the code-generating system, is represented by a gene. This is an extension of the representation proposed by Sanches and Cardoso (Sanches and Cardoso, 2010). The advantage of a representation similar to a DNA is that it captures all program structures while it encodes all program instructions.

PC are extracted with tools that analyze the program execution; however, CD, NF and DNA are extracted by the code-generating system.

4 REACTIONS

This article proposes the use of *reactions* to identify the similarity between two programs.

Hypothesis. *Two or more programs are similar if they react identically when applied the same transformation sequences.*

Validation. *It is possible to obtain performance curves with the same behavior, for programs P_x and P_y, applying the same transformation sequences. This indicates that both programs react identically, having a high degree of similarity. A simple method to prove this theorem is to: (1) compile both programs with the same sequences; (2) plot the performance graph for both programs; and (3) measure the curve behavior. As shown in Figure 1, the programs* ADPCM_C *and* N-BODY *are similar, because they possess a comparable behavior in regards to reactions, unlike the program* ACKERMANN*, which reacts differently. This pattern always occurs for programs that are similar or not.*

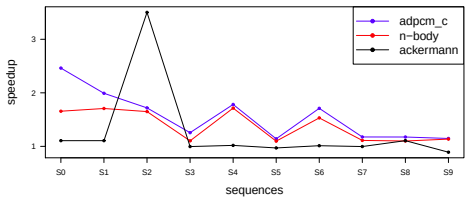

Figure 1: Performance over a compilation without using transformations.

Naturally, there is a need for a methodology to identify similar programs based on their characteris-

tics. Thus, based on the premise that *reactions* are a good strategy to identify similarities, a requirement emerges to specify a similarity coefficient that, given the characteristics of two programs, determines if they react identically.

4.1 Coefficients to Identify Programs with Similar Reactions

This article uses the coefficients applied in the works of Lima *et al.* (de Lima et al., 2013) and Queiroz and da Silva (Queiroz Junior and da Silva, 2015) to identify similar programs. These coefficients are the following:

Cosine. In this coefficient, the similarity coefficient between P_x and P_y is obtained by:

$$sim(P_x,P_y) = \frac{\sum_{w=1}^{M}(P_{xw} \times P_{yw})}{\sqrt{\sum_{w=1}^{M}(P_{xw})^2} \times \sqrt{\sum_{w=1}^{M}(P_{yw})^2}}$$

Euclidean. In this coefficient, the similarity coefficient between P_x and P_y is obtained by:

$$sim(P_x,P_y) = \frac{1}{\sqrt{\sum_{w=1}^{M}(P_{xw} - P_{yw})^2}}$$

Jaccard. In this coefficient, the similarity coefficient between P_x e P_y is obtained through:

$$sim(P_x,P_y) = \frac{1}{M}\sum_{w=1}^{M}\frac{min(P_{xw},P_{yw})}{max(P_{xw},P_{yw})}$$

In each coefficient, `M` is the number of characteristics.

In addition to these 3 coefficients proposed in the literature, this article evaluates two other coefficients:

SVM. Support Vector Machine.

NW. This article utilizes the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970) to classify programs represented as `DNA`. Therefore, to determine a similar reaction between two programs, the score from the alignment between two `DNA`s is evaluated.

Thus, `Cosine` (`CO`), `Euclidean` (`EU`), `Jaccard` (`JA`) and `SVM` are coefficients that estimate the similarity between programs represented by `PC`, `CD` or `NF`. While, `NW` is the coefficient that estimate the similarity between programs represented by `DNA`.

4.2 A Methodology to Find the Best Coefficient

The similarity coefficient that is able to identify if two performance curves are similar, based on their behavior and amplitude, is considered to be the best. The behavior refers to the gain in program performance, in other words, if there was *speedup* or *slowdown*. The amplitude refers to how much gain or loss there was in performance.

It is possible to describe the behavior of the program P_x, as shown in Table 1, based on the generated-code analysis with `N` different transformation sequences.

Table 1: Behavior of P_x, when compiled with sequences from S_0 to S_9.

	S_0	S_1	S_2	...	S_9
S_0	1	T_0/T_1	T_0/T_2	...	T_0/T_9
S_1	T_1/T_0	1	T_1/T_2	...	T_1/T_9
S_2	T_2/T_0	T_2/T_1	1	...	T_2/T_9
...	...	...	...	1	...
S_9	T_9/T_0	T_9/T_1	T_9/T_2	...	1

In Table 1, the line i represents the performance of the sequences, where S_i is the baseline, and T_i/T_j is the (*speedup* or *slowdown*) performance.

There exists a possibility to verify if there was a gain or loss in performance for the inputs ij. This can be seen in the tables corresponding to P_x and P_y. The methodology consists in using the upper part of the diagonal to measure the behavior. Thus, for each pair (s_x,s_y) of performances compared, the function $Coeff(s_x,s_y)$ measures the resemblance between the behavior and amplitudes of *speedups* referring to s_x and s_y, as follows:

$$Coeff(s_x\ ,\ s_y) = \begin{cases} \frac{min(s_x,s_y)}{max(s_x,s_y)}, & \text{if } \neg(s_x > 1 \oplus s_y > 1) \\ 0, & \text{otherwise} \end{cases}$$

Since `N` transformation sequences are considered, the best coefficient is the one that obtains the highest value of `MCoeff`, which is given by:

$$MCoeff = \sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1} Coeff(s_{ij}\ ,\ s'_{ij})$$

where, s_{ij} and s'_{ij} are the *speedups* obtained by P_x and P_y, respectively.

5 A DATABASE OF TRANSFORMATION SEQUENCES

In order to evaluate *reactions* between two programs and also the performance of knowledge representations for program characterization, it is necessary to create a database of transformation sequences. Such process is performed as follows.

Code-generating System. The LLVM 3.7.1 compiler infrastructure (Lattner, 2017).

Transformations. The creation of a transformation sequence evaluates 133 transformations, which are available in LLVM.

Training Programs. The training programs are composed of programs taken from LLVM's test-suite (Lattner, 2017), and The Computer Language Benchmarks Game (Gouy, 2017). These programs are composed of a single source code, and have a short execution time. Such programs were used by Purini and Jain's work (Purini and Jain, 2013).

Reducing the Search Space. This article uses a five-step process:

1. Reduce the search space using a genetic algorithm (GA);
2. Extract the best transformation sequences from each training program;
3. Add to this transformation sequences, the 10 good sequences founded by Purini and Jain (Purini and Jain, 2013);
4. Evaluate each training program using the 62+10 sequences; and
5. Store into the database for each program the pair: <training program, good transformation sequences>. A good transformation sequence is that provides to the program a lower execution time than the best LLVM level of transformation (-O3).

The GA consists in randomly generating an initial population, which will be evolved in an iterative process. Such process involves choosing 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 transformation space. Evolving a population includes the application of two genetic operators: crossover, and mutation. The first operator has a probability of 60% for creating a new individual. In this case, a tournament strategy ($Tour = 5$) selects the parents. The second operator, mutation, has a probability of 40% for transforming an individual. In addition, each individual has an arbitrary initial length, which can ranges from 1 to $|Transformation\ Space|$. Thus, the crossover operator can be applied to individuals of different lengths. In this case, the length of the new individual is the average of its parents. Four types of mutation operations were used:

1. Insert a new transformation into a random point;
2. Remove an transformation from a random point;
3. Exchange two transformations from random points; and
4. Change one transformation in a random point.

Both operators have the same probability of occurrence, besides 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 50 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.

The strategy used to reduce the search space is similar to the strategy proposed by Martins *et al.* (Martins et al., 2016) and Purini and Jain (Purini and Jain, 2013).

6 EVALUATING KNOWLEDGE REPRESENTATIONS

The following sections describe the evaluations performed in order to determine the best coefficient and strategy to characterize programs.

6.1 Experimental Setup

Architecture. Intel(R) Core(TM) i7-3770 CPU 3.4GHz with 8GB RAM running the Ubuntu 14.04 x64 operating system with kernel 4.2.0-41.

Compiler. The LLVM 3.7.1 compiler infrastructure (Lattner, 2017).

Feature Extraction. PC are extracted with the PAPI tool. CD are characteristics provided by the LLVM infrastructure. Two extractor modules, coupled

with LLVM, were implemented in order to extract NF and DNA during the compilation process, from LLVM intermediate representation.

Representing Programs. This article examines two different approaches to represent programs: (1) hot functions (HOT); and (2) full programs (FULL). Hot functions have a high execution cost, meaning they possess more time consumption in regards to the program execution. The similarity coefficient examines programs based on their hot functions and determines whether they are similar or not. The algorithm proposed by Wu and Larus (Wu and Larus, 1994) was performed to identify the hot functions.

SVM. The SKLEARN library was utilized for this coefficient (Pedregosa et al., 2011).

Programs. The test phase consists of programs that belong to the CBENCH benchmark (Fursin, 2017).

Transformation Sequences. This article utilizes 75 sequences to evaluate *reactions*: 62 founded by the GA during the process of reducing the search space; the 10 sequences founded by Purini and Jain (Purini and Jain, 2013); and the 3 sequences provided by LLVM (-O1, -O2, and -O3).

Runtime. Each program was executed 100 times in order to ensure accurate results. In addition, 20% of the results were discarded: 10% of the best and 10% of the worst. So, the geometric average runtime is calculated based on 80% of the data.

6.2 Results and Discussions

Analyzing the entire database, the value of $bMCoeff$ represent the best value that can be achieved by $MCoeff$ to available training programs. Table 2 presents the results obtained by the evaluated strategies, where WV, GM and BV refer to the distance for the best possible value ($\frac{MCoeff}{bMCoeff}$).

The best value of GM was obtained by FULL-NF with EU. Other strategies, lost up to 17.65% of performance.

It is worth highlighting the unexpected performance for PC, which was the only dynamic data evaluated. In fact, PC had the lowest average among all strategies. The best value was up to 12.94% worse than NF.

NF showed consistent results in regards to the four coefficients, having the smallest variance; 9.17×10^{-5}, and 10.00×10^{-5}, for HOT and FULL respectively. Alternatively, other strategies obtained a variance of 86.67×10^{-5}, and 342.50×10^{-5}, for PC and CD respectively.

Table 2: Obtained results (WV: worst value; GM: geometric mean; BV: best value; PR number of perfect results; BR: number of best results).

Strategy			WV	GM	BV	PR	BR
HOT	DNA	NW	0.63	0.82	1.00	1	5
	NF	CO	0.58	0.81	1.00	2	4
		EU	0.58	0.80	1.00	1	3
		JA	0.60	0.82	1.00	1	3
		SVM	0.58	0.80	1.00	1	3
FULL	PC	CO	0.60	0.79	1.00	1	3
		EU	0.60	0.75	1.00	1	1
		JA	0.62	0.80	1.00	1	3
		SVM	0.60	0.74	1.00	1	1
	CD	CO	0.52	0.81	1.00	1	5
		EU	0.63	0.82	1.00	5	13
		JA	0.50	0.70	0.98	0	3
		SVM	0.63	0.82	1.00	1	5
	NF	CO	0.66	0.85	1.00	3	10
		EU	0.63	0.83	1.00	1	6
		JA	0.63	0.83	1.00	1	8
		SVM	0.63	0.83	1.00	1	6

The highest variance was between the averages obtained by CD. This means that, although the representation achieved good performances with the EU and SVM coefficients, the results with other coefficients were discrepant, reaching a $\simeq$ 14.64% of difference between JA and EU.

DNA reached an average 3.53% lower than the best strategy. However, in global terms, it was 50% worse when compared to the best strategy. Among the HOT strategies, only DNA and JA achieved satisfactory results. These results indicate that the best strategy consists in representing the program completely and not focusing only on its hot function.

NF is the best program representation scheme, because of four reasons:

1. provided the best MCoeff;
2. has stability to achieve perfect results when the similarity coefficient is altered;
3. obtained one of the lowest variances between the best results; and
4. its worst results are not as low as those reached by other representations.

In regards to coefficients, CO is the best for three reasons:

1. provided the best values of MCoeff;
2. was the coefficient that presented, extensively, the best result; and
3. had the highest number of programs with the best result.

7 AN AUTOMATIC CODE-GENERATING SYSTEM

In order to evaluate the performance of the best strategy to characterize and find similar programs, this section describes an automatic code-generating system (ACGS) capable of inferring the best transformation sequence, based on previous experiences, for the test program.

The system employs a traditional machine-learning model, which is composed by offline and online phases. The former creates a database containing good transformation sequences for different programs. While the latter predicts a good sequence for the test program and generates the target code.

7.1 The Offline Phase

In a machine-learning model the offline phase consists on a training phase.

In this article, the offline phase is as described in Section 5. As a result, the training data is the same database that was generated to evaluate *reactions* and also the performance of knowledge representations for program characterization.

7.2 The Online Phase

The online phase is a case-based reasoning strategy (Richter and Weber, 2013), which performs the following steps:

1. **Retrieve Past Experiences.** This is accomplished by extracting the characteristics NF of the test program, and calculating the value of similarity for each program that belongs to the database with the CO coefficient. Finally, N sequences of the test program with the highest similarity are recovered.
2. **Reuse Past Experiences.** This is accomplished by generating code for the test program utilizing the transformation sequences recovered in the previous step.
3. **Review the Result, Evaluating the Success of the Solution.** This is performed by executing the generated code and measuring its execution time.

7.3 Methodology

The described system was evaluated utilizing a methodology described in Section 6.1. The evaluation considers 1, 3, 5 and 10 past experiences. In addition, to evaluate the effectiveness of the automatic code-generating system, this section compares it with three techniques:

1. **Genetic Algorithm with Tournament Selector.** (GA50) It is similar to the technique described on Section 7.1.
2. **Genetic Algorithm with Tournament Selector.** (GA10) It is also similar to the technique described on Section 7.1, except that it runs over 10 generations and 20 individuals.
3. **10 Good Sequences.** (Best10) It is a technique proposed by Purini and Jain (Purini and Jain, 2013). They founded 10 good sequences, which is able to cover several classes of programs. Thus, in this technique the unseen programs is compiled with all sequences, and the best target code is returned.

The evaluation uses four metrics to analyze the results, namely:

1. GMS: geometric mean speedup;
2. NPS: number of programs achieving speedup over the best transformation level (-O3);
3. NoS: number of sequences evaluated; and
4. ReT: the technique response time.

The speedup is calculated as follows:

$$Speedup = Running\_time\_Level\_O0 / Running\_time$$

7.4 Results and Comparison

This section compares the ACGS with other strategies. Figure 2 shows the speedups for ACGS, Best10, GA10 and GA50.

GMS. The GMS achieved by ACGS was 1.919, outperformed only by GA50 which achieved 1.980. The Best10 and GA10 reached 1.784 and 1.822, respectively. It is important to consider the different premises of ACGS and GA. The former consists on a machine learning paradigm which return a solution on a few steps. The latter is an iterative compilation technique, which evaluate several sequences over the program. ACGS achieves similar performance results for GA50, with a difference of only 6.1%, on a considerably lower time (99%). Futhermore, ACGS surpasses the other strategies.

NPS. The ACGS did not achieve the higher speedups reached by GA50; however the NPS was 38.9% larger, covering 25 programs while GA50 covers only 18. It means that GA50 achieves discrepant higher values on isolate programs, while ACGS achieves good performance for more programs. Best10 and GA10 had 12 and 17 NPS, respectively.

NoS. As iterative compilation techniques, GA10 and GA50 evaluate a high number of sequences to find

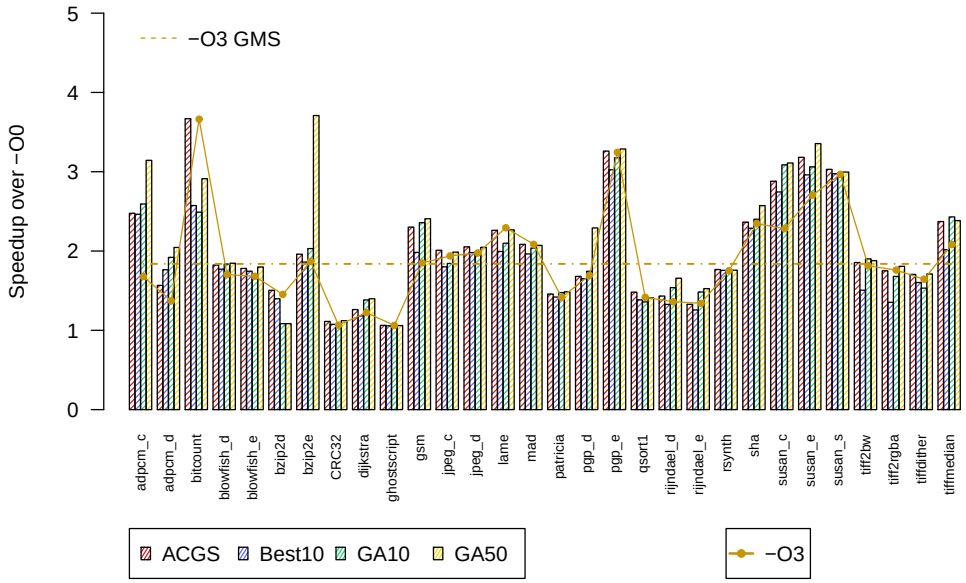

Figure 2: Speedups achieved compared with other techniques.

a result, having, on average, NoS of 57.1 and 259.3 sequences, respectively. The Best10 strategy evaluate the same number of sequences of ACGS.10 (10 sequences). However, Best10 evaluates always the same 10 sequences, while ACGS retrieves specific past experiences based on the program characteristics.

ReT. The GA50 was the most time-consuming technique, taking more than 26865 seconds to give a result, on average. GA10 spent a mean of 8378.5 seconds for each input program, while Best10 gave an answer after 424.2 seconds, on average too. ACGS was 62% faster than Best10 technique, evaluating the same number of sequences. Theoretically, the response time is essentially proportional to the number of sequences evaluated. However, Best10 evaluates the same 10 sequences to any test program, sequences which can be more time-consuming than the good sequences retrived by the ACGS.

These results indicates that iterative compilation achieves higher speedups, but with a high response time.

The considered metrics indicates that ACGS is a good strategy to find a good transformation sequence to a test program, surpassing other strategies. It is due to it is a technique which (1) gives an answer on a low response time, (2) finds a solution based on program characteristics, and (3) utilizes previous knowledge to return a sequence.

8 RELATED WORK

De Lima *et al.* (de Lima et al., 2013) proposed the use of a case-based reasoning strategy to find transformation sequences for a specific program. They argue that it is possible to find good sequences, from previous compilations, for an unseen program. This strategy creates several sequences in a training stage. Afterwards, in a deployment stage, the strategy infers a good sequence for a test 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 of performance counters. They demonstrated that it is possible to infer a sequence that achieves multiple goals; for example, runtime and energy efficency. De Lima's work has the same limitations than Cavazos' work.

Tartara and Reghizzi (Tartara and Reghizzi, 2013) proposed a long-term strategy, which its goal 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 (Namolaru et al., 2010). Basically, this strategy performs two tasks. First, it extracts the characteristics of the test program. Second, a genetic algorithm creates heuristics inferring which optimizations should be enabled. Although Tartara's and Reghizzi's work uses a good static characteristic to represent programs (NF), they did not formalize the efficiency of

this representation.

Queiroz and Da Silva (Queiroz Junior and da Silva, 2015) evaluates different configurations of a case-based reasoning strategy, which aims to find transformation sequences for a specific program. The goal of their work was to evaluate the performance of such strategy using: (1) different databases; (2) different coefficients to identify programs with similar reactions; and (3) different program characterizations. Although, Queiroz's and Da Silva's work has the appeal of evaluating several configuration, it has three problems: (1) it does not describe a formalism to find an efficient representation; (2) it evaluates only two representations; and (3) the results obtained by the code-generating system does not use only one configuration.

9 CONCLUSIONS AND FUTURE WORK

Finding the best form of knowledge representation depends on a determined objective and requires detailed evaluations of the constructed formalism.

A complex problem, in the computer science field, is to generate good target code because it is program-dependent. This indicates that proposed strategies should consider the program during decision-making. In addition, they need to contemplate which transformations should be applied during the code-generation process.

Although the literature describes several strategies that attempt to mitigate the code-generation problem, there is no consensus on which knowledge representation should be utilized in these types of systems. Furthermore, various strategies do not consider the said problem as program-dependent, because the complexity to identify an efficient knowledge representation.

This article presented and validated an efficient knowledge representation to characterize programs, the `NF`. This representation is interesting because can be extracted statically and is not dependent of programming languages nor hardware architecture. Another contribution of this article is the identification of a coefficient that is able to identify programs that react similarly when compiled applying the same transformation sequence.

The results obtained by the code-generating system, that considers `NF` as program representation, is able to find good transformation sequences, as well as outperforms other code-generating systems.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Prentice Hall, Boston, MA, USA, 2 edition.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA. IEEE Computer Society.

de Lima, E. D., de Souza Xavier, T. C., da Silva, A. F., and Ruiz, L. B. (2013). Compiling for performance and power efficiency. In *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2013 23rd International Workshop on*, pages 142–149.

Fursin, G. (2017). Collective Benchmark - Enabling realistic benchmarking and optimization. http://ctuning.org/cbench. Access: January, 9 - 2017.

Gouy, I. (2017). The Computer Language Benchmarks Game. http://benchmarksgame.alioth.debian.org/. Access: January, 9 - 2017.

Lattner, C. (2017). The LLVM Compiler Infrastructure. http://llvm.org. Access: January, 9 - 2017.

Martins, L. G. A., Nobre, R., Cardoso, J. a. M. P., Delbem, A. C. B., and Marques, E. (2016). Clustering-based selection for the exploration of compiler optimization sequences. *ACM Trans. Archit. Code Optim.*, 13(1):8:1–8:28.

Namolaru, M., Cohen, A., Fursin, G., Zaks, A., and Freund, A. (2010). Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *International Conference on Compilers Architectures and Synthesis for Embedded Systems*, Scottsdale, United States.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453.

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., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Purini, S. and Jain, L. (2013). Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization*, 9(4):56:1–56:23.

Queiroz Junior, N. L. and da Silva, A. F. (2015). Finding good compiler optimization sets - a case-based reasoning approach. In *International Conference on Enterprise Information Systems*, pages 504–515.

Richter, M. M. and Weber, R. (2013). *Case-Based Reasoning: A Textbook*. Springer, USA.

Sanches, A. and Cardoso, J. M. P. (2010). On identifying patterns in code repositories to assist the generation of hardware templates. In *International Conference on*

Field Programmable Logic and Applications, pages 267–270, Washington, DC, USA. IEEE Computer Society.

Tartara, M. and Reghizzi, S. C. (2013). Continuous learning of compiler heuristics. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):46:1–46:25.

Wu, Y. and Larus, J. R. (1994). Static branch frequency and program profile analysis. In *Annual International Symposium on Microarchitecture*, pages 1–11, New York, NY, USA. ACM.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS