Fabrício Filho J. *et al.* Yet Another Intelligent Code-generation System: A Flexible and Low-cost Solution.. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY : 1– 2018. DOI 10.1007/s11390-018-1867-7

Yet Another Intelligent Code-generating System: A Flexible and Low-cost Solution

João Fabrício Filho [12*], Luis Gustavo Araujo Rodriguez [1], and Anderson Faustino da Silva [1]

[1] *Department of Informatics, State University of Maringá, Maringá-PR, Brazil*
[2] *Federal University of Technology - Campus Campo Mourão, Campo Mourão-PR, Brazil*

E-mail: joaof@utfpr.edu.br, luisgar1990@gmail.com, anderson@din.uem.br

Abstract Modern compilers apply various code transformation algorithms to improve the quality of the target code. However, a complex problem is to determine which transformation algorithms must be utilized. This is difficult because of three reasons: number of transformation algorithms, various combination possibilities, and several configuration possibilities. Over the last few years, various intelligent systems were presented in the literature. The goal of these systems is to search for transformation algorithms and thus, apply them to a certain program. This paper proposes a flexible, low-cost and intelligent system capable of identifying transformation algorithms for an input program, considering the program's specific features. This system is flexible for parameterization selection and has a low-computational cost. In addition, it has the capability to maximize the exploration of available computational resources. The system was implemented under the Low Level Virtual Machine infrastructure and the results indicate that it is capable of exceeding, up to 21.36%, performance reached by other systems. In addition, it achieved an average improvement of up to 17.72% over the most aggressive compiler optimization level of the Low Level Virtual Machine infrastructure.

Keywords compilers, code transformations, iterative compilation, knowledge representation, machine learning

1 Introduction

Intelligent systems are systems that can interact and learn about particular contexts. These systems reduce the cost to solve complex problems because of their ability to extract, classify and accumulate knowledge. In recent years, several papers have applied intelligent systems to different contexts, for example, diagnosis of vitamins and mineral deficiency on the human body [1], databases [2], architectural anomalies on software reuse [3], or code generation [4].

Code generation is a process performed by compilers [5, 6]. It is considered to be a complex problem because of the difficulty in choosing (1) which code transformation algorithms (`TAs`) [7] must be utilized, (2) the order of such `TAs`, and (3) their parametrizations. In this context, intelligent systems can be used to select `TAs` and, consequently, apply them to an input program.

The first generation of intelligent systems, in the context of code generation, is based on iterative compilation techniques [8, 9, 10]. Although these systems are able to extract and classify knowledge in order to select `TAs` for code generation, they do not store knowledge. In addition, they require a high response time to provide a good solution. Therefore, the second generation of intelligent systems emerges and applies machine learning techniques in order to address these issues [11, 12, 13].

This paper proposes a flexible, low-cost and intelligent system called Yet Another Code-generating System (`YaCoS`), which is capable of efficiently selecting `TAs` to be used by the compiler. The objective is to present a system that provides various forms of knowledge representation and extraction. In addition, the proposed system offers continuous learning and exploits the available hardware resources in order to improve the knowledge database.

The main contributions of this paper are as follows.

1) We propose an intelligent system for selecting `TAs` to be used by the compiler, whose characteristics are:

*Corresponding author

(a) knowledge extraction, classification, and accumulation;

(b) configuration of knowledge representation;

(c) configuration of the knowledge extraction form;

(d) continuous learning; and

(e) exploitation of available computing resources.

2) We provide a real implementation of the proposed system as a tool for the Low Level Virtual Machine (`LLVM`) compiler infrastructure [14].

3) We provide a detailed analysis and evaluation of the proposed system.

In addition to the aforementioned contributions, we attempt to tackle the following questions.

1) What is the best accumulated knowledge? The one from the most similar past experience even if such experience does not perform well, or the accumulated knowledge that has the best performance even if such knowledge is not derived from the most similar past experience?

2) What is the best approach for characterizing programs? A dynamic approach or a static approach?

3) Which entities should be used for characterizing programs? All the structural entities of the program (i.e., the entire program), or only entities from the hot function?

4) What is the cost-benefit between performance (quality of results) and response time?

The results indicate that the proposed intelligent system is able to surpass the performance of other systems, in terms of quality of the results and response time. In addition, the system achieves an average improvement of up to 17.72% over the most aggressive compiler optimization level of the `LLVM` infrastructure.

2 Flexible, Low-Cost and Intelligent Code-Generating System

During code generation, modern compilers apply various `TAs` to improve the quality of the target code. However, some `TAs` may be beneficial to a particular program but not to others. Thus, the appropriate strategy is to select `TAs` considering the problem to be program-dependent. In order to solve this issue, this section presents Yet Another Intelligent Code-Generating System (`YaCoS`), which is flexible, low-cost and capable of selecting `TAs` for code generation.

2.1 Overview

Fig. 1 presents the code generation flow used by `YaCoS`. The input and the output of `YaCoS` are the source code and the target code respectively, and possess the same semantics.

The code generation flow is comprised of the following steps:

1) Characterizing. First, the system extracts features, which will represent the program.

2) Comparing. The knowledge (or feature vector) extracted from the test program is compared with vectors stored in the knowledge database. This is done in order to obtain solutions applied to similar programs, that were previously compiled by the system.

3) Retrieving. After identifying similar compiled programs, the system retrieves effective `TA` sequences from these programs. This process establishes a recovery model, which defines how and in what order the recovered solutions will be applied. The similarity between the programs stored in the knowledge database is considered on retrievement of the solutions.

4) Compiling. When the order of prior solutions is obtained, the test program is compiled with each retrieved `TA` sequence and, consequently, generates several target codes.

5) Evaluating. Each target code is evaluated in order to find the best solution. This end result is comprised of the best `TA` sequence that generates the best target code.

6) Returning. The best generated code is returned to the end user.

7) Learning. The knowledge database is fed with new knowledge.

Based on the aforementioned steps, `YaCoS` requires a knowledge database containing relationships between programs and `TAs`. An interesting feature of `YaCoS` is its ability to improve knowledge quality and, consequently, improve the generated code for future requirements.

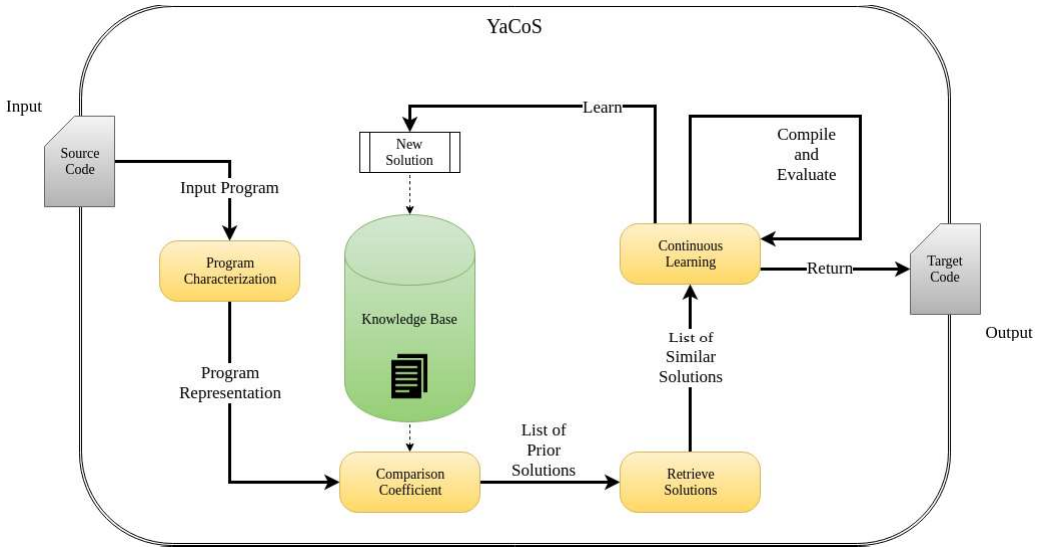


Fig. 1. Code generation flow used by `YaCoS`.

`YaCoS` is meant to be a system that assists the end user to generate code; however, generating a knowledge database is not an easy task. Thus, the knowledge database should be generated and maintained by compiler experts and not end users. To minimize this problem, the end user can use `YaCoS` with its standard knowledge database, which is described in Subsection 2.2.

Although there exists a necessity to select, order and parameterize `TAs`, `YaCoS` does not attempt to solve all these problems. In fact, `YaCoS` uses a mechanism to choose `TAs`. Furthermore, the order of application of each algorithm is not fully addressed. Finally, choosing parameters for each optimization is not addressed in this paper, although `TAs` requires such process. Thus, we use default parameters provided by the compiler.

2.2 Knowledge Base

The knowledge database stores relationships between program features and transformation algorithm (`TA`) sequences.

It is important to highlight that a concise knowledge database is required in order to simplify the search process for solutions. Thus, representing `TA` sequences should be amplified, containing good solutions for several classes of programs. Therefore, creating the database has the following steps.

1) Form a set with good `TA` sequences. This set consists of:
   - the best `TA` sequence, found by a genetic algorithm (`GA`), for each training program;
   - the 10 good `TA` sequences found by Purini and Jain's approach [10];
   - the three `LLVM` optimization levels (`-O1`, `-O2` and `-O3`).
2) Evaluate each training program using the entire set of `TA` sequences.
3) Store an input, in the knowledge database for each training program, containing all `TA` sequences and their respective performance (runtime), and program features.

Knowledge stored in the database must be able to cover different classes of problems. Thus, our selection strategy for creating the knowledge base focuses on this premise. In fact, our strategy is based on Purini and Jain's approach [10].

A genetic algorithm (`GA`) is used to create a good `TA` sequence for each training program. This approach can be described as follows. The initial population is random and each individual consists of a `TA` sequence, which evolves with each generation. Each individual is represented by a vector, where each position contains a `TA` sequence. The size of each individual can vary from 1 to $|TA\text{-}Space|$ (size of the search space, more precisely the number of `TAs`).

Two individuals are chosen at each iteration using a tournament strategy and, thus, generate new individuals through a crossover operator, which has a 60%

probability of occurrence. Furthermore, this operator can be applied to individuals of different sizes. Consecutively, the mutation operator can alter or modify new individuals. This operator has a 40% probability of occurrence.

The mutation process modifies an individual by (1) inserting a new TA sequence arbitrarily; (2) removing a TA at a random point; (3) exchanging two TAs that are in a sequence of arbitrary points; and (4) altering a TA for another. One of the aforementioned modifications is performed and chosen randomly.

Crossover and mutation, apart from modifying or creating new individuals, modify the order of TAs.

The execution of GA is set to 100 generations and a population of 50 individuals. The best performing individual always remains for the next generation. GA ends its execution when one of the following conditions occurs: (1) convergence does not exist for three consecutive generations; or (2) population diversity is less than 0.01.

It is worth mentioning that Purini and Jain [10] and Martins *et al.* [15] proposed a similar strategy.

Thus, the knowledge base stores relationships between program features and TA. Furthermore, it stores the order of application of each TA. It is important to note that the 10 good TA sequences found by Purini and Jain, and the three compiler optimization levels are sequences that indicate the order of application of each algorithm.

2.2.1 TA *Space*

Transformation algorithms are comprised of a TA sequence. In addition, the search space is comprised of 131 TAs[1]. Thus, the size of an individual can vary between 1 and 131 (algorithms).

2.2.2 Training Programs

The selection of training programs has to be based on knowledge capable of covering different classes of problems. Purini and Jain [10] proved that this issue can be addressed using simple programs with low runtime.

Thus, the knowledge base was created using Purini and Jain's programs [10], which were taken from the LLVM[2] test suite. An addition of six programs from The Computer Language Benchmarks Game[3] was also used. These programs allow establishing relationships of <features, TAs>, which will cover diverse classes of programs.

Table 1 presents the training programs.

Table 1. Training Programs

Purini and Jain's Training Programs	
ackermann (T00)	mandel (T28)
ary3 (T01)	mandel-2 (T29)
bubblesort (T02)	matrix (T30)
chomp (T03)	methcall (T31)
dry (T04)	misr (T32)
dt (T05)	n-body (T33)
fannkuch (T06)	nsieve-bits (T34)
fbench (T07)	oourafft (T35)
ffbench (T08)	oscar (T36)
fib2 (T09)	partialsums (T37)
fldry (T10)	perlin (T38)
flops (T11)	perm (T39)
flops-1 (T12)	pi (T40)
flops-2 (T13)	queens (T41)
flops-3 (T14)	queens-mcgill (T42)
flops-4 (T15)	quicksort (T43)
flops-5 (T16)	random (T44)
flops-6 (T17)	realmm (T45)
flops-7 (T18)	recursive (T46)
flops-8 (T19)	reedsolomon (T47)
fp-convert (T20)	richards_benchmark (T48)
hash (T21)	salsa20 (T49)
heapsort (T22)	sieve (T50)
himenobmtxpa (T23)	spectral-norm (T51)
huffbench (T24)	strcat (T52)
intmm (T25)	towers (T53)
lists (T26)	treesort (T54)
lpbench (T27)	whetstone (T55)
The Computer Language Benchmarks Game	
binary-trees (T56)	mandelbrot (T59)
fasta (T57)	pidigits (T60)
fasta-redux (T58)	regex-dna (T61)

2.2.3 Relationship of <features, TAs*>*

Characteristics extracted from training programs (or test programs) form a feature vector of size F, where F is the number of features used to represent a program. Thus, the relationship of <features, TAs> associates TA sequences with specific features. Therefore, YaCoS is capable of identifying which TA is adequate to use during code generation.

As mentioned before, the knowledge database stores for each training program: TA sequences and their respective performance (runtime), and the program features. This means that YaCoS is able to know which sequence was the best one during target code generation for the training program (analyzing the performance).

[1] http://sites.google.com/site/transformationspaceof131tas
[2] http://llvm.org
[3] http://benchmarksgame.alioth.debian.org/

Thus, YaCoS considers the order of performance during the evaluation of TA sequences during target code generation for the test program.

Subsection 2.3 explains in detail the structure of a feature vector.

2.3 Characterizing Programs

Since selecting TAs is a program-dependent problem, YaCoS can be parametrized to characterize programs by extracting features from: (1) their entire structure or (2) only their hottest function. In the first case, the process of selecting TAs is guided by features of the entire program. In the second case, the process is guided by features of the most critical part of the program.

Computer programs can be represented by dynamic or static features. Dynamic features describe program behaviors related to their execution. However, static features describe the program's algorithmic structures.

Dynamic features are appealing because they are related to both the hardware and program behavior during execution. However, these features possess disadvantages such as program execution and platform dependency.

Alternatively, static features are platform-independent and do not require program execution. However, these characterizations do not consider the input data, which can change the program behavior and, consequently, alter parameters of the code-generating system.

YaCoS can be parametrized to use one of the following program characterizations.

Performance Counters (PC) are the outcome of program execution. These features are related to the available hardware. Several research publications used PC as a program characterization scheme [11, 12, 13]. These features are collected with instrumentation tools during program execution. Table 2 presents PC used when YaCoS is executed on an I7 processor.

Table 2. Performance Counters

Class	Features		
Cache	PAPI_L1_TCM	PAPI_L1_LDM	PAPI_L1_ICM
	PAPI_L1_DCM	PAPI_L1_STM	PAPI_L2_STM
	PAPI_L2_TCA	PAPI_L2_DCW	PAPI_L2_TCM
	PAPI_L2_TCR	PAPI_L2_DCA	PAPI_L2_TCW
	PAPI_L2_DCH	PAPI_L2_ICR	PAPI_L2_DCM
	PAPI_L2_ICM	PAPI_L2_ICA	PAPI_L2_ICH
	PAPI_L2_DCR	PAPI_L3_TCM	PAPI_L3_DCW
	PAPI_L3_DCA	PAPI_L3_TCR	PAPI_L3_TCW
	PAPI_L3_ICR	PAPI_L3_ICA	PAPI_L3_TCA
	PAPI_L3_1DCR		
Branch	PAPI_BR_PRC	PAPI_BR_UCN	PAPI_BR_CN
	PAPI_BR_NTK	PAPI_BR_INS	PAPI_BR_MSP
	PAPI_BR_TKN		
SIMD	PAPI_VEC_SP	PAPI_VEC_DP	
Floating Point	PAPI_FDV_INS	PAPI_FP_INS	PAPI_DP_OPS
	PAPI_FP_OPS	PAPI_SP_OPS	
TLB	PAPI_TLB_DM	PAPI_TLB_IM	
Cycles	PAPI_REF_CYC	PAPI_TOT_CYC	PAPI_STL_ICY
	PAPI_STL_ICY		
Instructions	PAPI_TOT_INS*		

Compilation Data (CD) are features that describe the relationships between program entities. They are defined by the intermediate language used by the code-generating system, for which YaCoS is based on. Queiroz Junior and Da Silva proposed using these features [13]. However, their use is limited to data provided by the compiler, despite being directly related to the source code. These features are collected by the code-generating system, which YaCoS is based on, in other words LLVM. Such features are presented in Table 3.

Table 3. Compilation Data

Class	Feature
Binary Insts	Number of Add insts
	Number of Sub insts
Memory Insts	Number of Store insts
	Number of Load insts
	Number of memory instructions
	Number of GetElementPtr insts
	Number of Alloca insts
Terminator Insts	Number of Ret insts
	Number of Br insts
Other Insts	Number of ICmp insts
	Number of PHI insts
	Number of machine instrs printed
	Number of Call insts
Function	Number of non-external functions
Basic block	Number of basic blocks
Floating Point Insts	Number of floating point instructions
Total Insts	Number of instructions (of all types)*

Numerical Features (NF) are features extracted from relationships between the program entities, which are defined by specific features of the

programming languages. They were proposed by Namolaru *et al.* [16] and were systematically produced by experiments. Namolaru *et al.* proved their influence on parameterizing code-generating systems. Similar to CD, numerical features are also extracted from the intermediate language used by the code-generating system. These features are presented in Table 4.

Table 4. Numerical Features

Class	Features
Basic Block	Number of basic blocks in the method
	Number of basic blocks with a single predecessor
	Number of basic blocks with a single predecessor and a single successor
	Number of basic blocks with a single predecessor and two successors
	Number of basic blocks with a single successor
	Number of basic blocks with a two predecessors and one successor
	Number of basic blocks with more than two predecessors
	Number of basic blocks with more than two successors
	Number of basic blocks with more than two successors and more than two predecessors
	Number of basic blocks with number of instructions greater than 500
	Number of basic blocks with number of instructions in the interval [15, 500]
	Number of basic blocks with number of instructions less than 15
	Number of basic blocks with two predecessors
	Number of basic blocks with two successors
	Number of basic blocks with two successors and two predecessors
Binary operations	Number of binary bitwise operations in the method
	Number of binary floating point operations in the method
	Number of binary integer operations in the method
Call instructions	Number of calls that return a float
	Number of calls that return a pointer
	Number of calls that return an integer
	Number of calls with pointers as arguments
	Number of calls with the number of arguments is greater than 4
	Number of direct calls in the method
	Number of indirect calls (i.e. done via pointers) in the method
Control flow graph	Number of critical edges in the control flow graph
	Number of conditional branches in the method
	Number of edges in the control flow graph
	Number of unconditional branches in the method
	Number of phi-nodes in basic blocks
Conversion instructions	Number of floating point conversion instructions
	Number of integer conversion instructions
Functions	Number of functions
Memory instructions	Number of store instructions
	Number of load instructions
	Number of memory address instructions
Other instructions	Number of vector instructions
	Number of getElementPtr Instructions
	Number of instructions in the method*
	Number of switch instructions in the method
	Number of terminator instructions
	Number of aggregate instructions
	Number of assignment instructions in the method

Symbolic Representation (DNA) characterizes each instruction of the intermediate language as a gene. This representation is an extension of the proposal by Sanches and Cardozo [17]. The advantage of using DNA as a code representation is that it captures all of the program's structures and encodes all of its instructions simultaneously. The proposed symbolic representation is presented in Table 5.

Table 5. DNA Encoding

Transformation Rules			
Br	A	InsertValue	d
Switch	B	Load	e
IndirectBr	C	Store	f
Ret	D	Alloca	g
Invoke	E	Fence	h
Resume	F	AtomicRMW	i
Unreachable	G	AtomicCmpXchg	j
Add	H	GetElementPtr	k
Sub	I	Trunc	l
Mul	J	ZExt	m
UDiv	K	SExt	n
SDiv	L	UIToFP	o
URem	M	SIToFP	p
SRem	N	PtrToInt	q
FAdd	O	IntToPtr	r
FSub	P	BitCast	s
FMul	Q	AddrSpaceCast	t
FDiv	R	FPTrunc	u
FRem	S	FPExt	v
Shl	T	FPToUI	x
LShr	U	FPToSI	w
AShr	V	ICmp	y
And	X	FCmp	z
Or	W	Select	0
Xor	Y	VAArg	1
ExtractElement	Z	LandingPad	2
InsertElement	a	PHI	3
ShuffleVector	b	Call	4
ExtractValue	c	others	5

As shown in Table 5, a DNA encodes instructions into a string. These instructions, which compose a program function, are defined by the intermediate representation of the code-generating system.

It is worth mentioning that PC is a dynamic feature and, thus, the system must execute the program to extract such features. As mentioned earlier, this increases the system response time, consequently offering a disadvantage. However, it models the program behavior and, therefore, provides benefits.

In terms of different characterizations, it is important to note that:

- `PC` and `CD` characterize the entire structure of the program;
- `NF` characterizes the entire structure of the program, as well as its hottest function;
- `DNA` characterizes only the hottest function.

Because there are several different characterizations, `YaCoS` can be partially or fully based on the structure of the program for extracting knowledge and generating target code.

It is important to mention that similar static structures can generate different features. This is because of the differences in runtime, input and number of modules. In order to address this issue, `YaCoS` normalizes the collected features except for `DNA`. This is done because this issue does not occur when such a feature is utilized. Tables 2, 3 and 4 indicate the features used to normalize the feature vector. These features are highlighted with *.

2.3.1 Identifying Hot Functions

A hot function is related to the performance of the program and thus, it is considered to be the most representative function.

A program's real cost is related to user inputs and, therefore, calculated dynamically. However, program execution does not alter its static properties (`DNA` or `NF`). Thus, we have chosen a static profiling technique to estimate the cost of code functions and, therefore, identify their hot function.

Wu and Larus [18] proposed a static profiling technique that achieved significant results to estimate function costs. This was done by analyzing the control flow graph. They proposed a technique that integrated results from predictive heuristics. These results are based on a previous work by Ball and Larus [19]. The process of integrating the results is done by a mathematical technique [20] that incorporates evidence from different sources for reliable predictions. Therefore, the proposed system uses this static profiler to identify the hot function.

2.4 Comparing Program's Characteristics

`YaCoS` must identify similar programs in order to obtain prior solutions (`TA` sequences). More precisely, `YaCoS` must identify which programs in the knowledge base are similar to the test program.

This paper proposes the use of reactions as a mechanism to identify the similarity between two programs.

Hypothesis. Two or more programs are similar if they react identically when applying the same TA sequences.

Validation. It is possible to obtain similar performance curves for the programs P_x and P_y, applying the same `TA` sequences. This indicates that both programs react identically, thus having a high degree of similarity. A simple method to verify this hypothesis is to: (1) compile the programs with the same `TA` sequences; (2) plot the performance chart for both programs; and (3) compare the behavior of each curve. As shown in Fig. 2, the programs `adpcm_c` and `n-body` are similar because of their comparable reaction behavior. However, `ackermann` reacts differently, which means it is not similar.

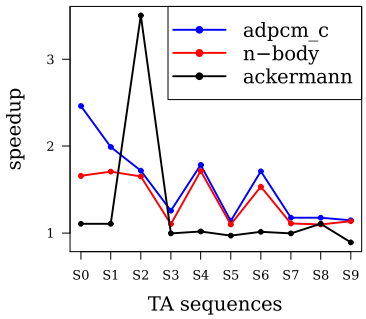


Fig. 2. Performance curves.

Based on the premise that reactions are a good strategy to identify similarities, it is necessary to specify the similarity coefficient that, given the feature vectors of two programs, determines whether they react similarly.

`YaCoS` can be parametrized to use one of the following coefficients:

Cosine (`CO`) the similarity between the programs P_x and P_y is obtained by:

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

where M is the size of the feature vector and, thus, P_{xw} is the feature W of program X (the same occurs for P_{yw}).

Euclidean (EU) the similarity between the programs P_x and P_y is obtained by:

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

where M is the size of the feature vector and, thus, P_{xw} is the feature W of program X (the same occurs for P_{yw}).

Jaccard (JA) the similarity between the programs P_x and P_y is obtained by:

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

where M is the size of the feature vector and, thus, P_{xw} is the feature W of program X (the same occurs for P_{yw}).

***Support Vector Machine* (SVM)** is a supervised learning model that analyzes data for classification purposes [21].

***Needleman-Wunsch* (NW)** is an algorithm [22] used to compare programs based on DNA. It is widely used in the literature to compare biological DNAs. Thus, a score is evaluated to determine similar reactions between two programs. This evaluation criterion is based on the alignment of DNAs.

The coefficients CO, EU and JA, initially applied by Lima *et al.* [12] and Queiroz Junior and Da Silva [13], return a similarity percentage based on two feature vectors. Furthermore, SVM classifies a training set and indicates which training element is more similar to the test element. Finally, NW returns a score between the alignment of two DNAs that indicates the similarity between two programs (feature vectors).

It is important to note that NW is utilized only when YaCoS is guided by the program's hottest function. However, other coefficients can be used to identify similar programs based on any other type of characterization.

2.5 Retrieving TA Sequences

It is necessary to develop a strategy to retrieve prior experiences (TA sequences). This strategy needs to be based on the premise that the knowledge database stores information about preceding compilations, which contain diverse TA sequences related to the program that was previously compiled and identified as similar.

YaCoS can be parametrized to retrieve prior experiences using the following models:

ELITE Retrieves all TA sequences that outperform the best compiler optimization level (with a lower runtime). This is done by analyzing only the training program that has the highest similarity.

JUST Retrieves from the knowledge database N TA sequences and analyzes only the training program with the highest similarity. This is done by considering the order of performance, in other words, the high-performing N TA sequences are chosen.

NEARLY Retrieves from the knowledge database N TA sequences. This is done by considering only TA sequences that outperform the best compiler optimization level. If the training program with the highest similarity does not have N better TA sequences, the TA sequences of the second highest training program are chosen, and so on.

JUST and NEARLY need the system parameter N, which must be selected by the user and indicates the number of prior experiences that the system must retrieve. In addition, such strategies consider the order of performance which means that the N selected sequences are in increasing order based on their performance (runtime). Consequently, this will recover the sequences that had the best performance.

The main objective is to propose strategies and, thus, analyze the best approach for obtaining performance. This can be done either by:

1) retrieving only good TA sequences from the training program with the highest similarity (ELITE); or

2) retrieving only the TA sequences that belong to the training program with the highest similarity and are ordered in terms of performance (JUST); or

3) retrieving good TA sequences from training programs that have a proximate similarity (NEARLY).

2.6 Compiling the Test Program

After selecting `N TA` sequences, `YaCoS` generates N target codes using a different `TA` sequence for each one. This task is performed by informing the `LLVM` compiler, which transformations should be enabled.

2.7 Evaluating the Target Codes

Afterwards, `YaCoS` evaluates and returns the N target codes to the user. This process is performed by executing each target code, analyzing its runtime and returning the less time-consuming code to the user.

2.8 Learning from New Compilations

It is worth highlighting that an intelligent system requires continuous learning over time. Thus, `YaCoS` uses a feedback mechanism to enable the said requirement.

Feedback can be done by:

1) storing only the knowledge from new evaluations;
2) adjusting familiar knowledge database solutions to the test program; or
3) creating new solutions and adjusting them to the test program.

The first feedback mechanism is the simplest because it stores (learns) considering the last compilation; in other words, it considers the target code that was returned to the user. This solution has a lower cost since there are no additional compilations and evaluations.

The second feedback mechanism requires finding the best possible solution of the knowledge database for such a test program. Thus, the cost increases because the system applies and evaluates all known solutions to the input program.

The third feedback mechanism requires a strategy to create new `TA` sequences. Iterative compilation techniques can be applied to the test program before it is stored in the knowledge database. Thus, new (good) solutions will be created, which are specifically designed for the new program. However, there is a high cost to execute such a strategy.

In order to offer a balance between costs and benefits, the second strategy will be utilized for `YaCoS`. The first strategy is the simplest because it does not analyze the entire database to find the best solution; however, this is the least efficient strategy. Although the third strategy is the most efficient in terms of performance (benefits), it has a high cost because it requires a thorough analysis of the solutions in the search space. Therefore, the second strategy is used because it provides the best existing knowledge, and it does not require considerable system cost to produce new knowledge.

System feedback provides continuous learning while simultaneously adding computational costs. However, computer architectures [23] have evolved over the last few years. Multi-core processors have emerged [24], thus allowing a concurrent use of resources. A solution to reduce the system cost is to perform background learning.

The proposed strategy is as follows. After the system provides a solution to the user, further attempts to improve it are made by searching for other possible knowledge database solutions. This process is executed in the background and updates the database after it is entirely analyzed. This strategy is similar to the proposal presented by Yang *et al* [25].

Background learning uses hardware resources transparently. This means that it does not block user response. Thus, it adjusts the best solutions without increasing the system response time.

In the aforementioned model, the background thread has the responsibility to learn about a new solution. This is done transparently, meaning that the use of resources is unnoticeable.

3 Finding the Default Configuration

Several experiments were performed in order to find the default configuration for `YaCoS`. These experiments are described in this section. It is worth highlighting that feedback and background learning were not considered for these experiments.

3.1 Hardware and Software Platform

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

Compiler The compiler infrastructure is `LLVM` 3.7.1 [14].

Feature extraction `PC` is extracted using `PAPI`[4] tools. `CD` is provided by the `LLVM` infrastructure.

[4] http://icl.cs.utk.edu/~mucci/papiex/papiex.html

Two extractor modules were implemented in order to extract NF and DNA during the compilation process of LLVM's intermediate language.

Representing programs This paper examines two different approaches to represent programs: (1) hot functions (HOT) for DNA and NF, or (2) the entire program (FULL) for NF, CD and PC.

SVM The SKLEARN library [26] was used to calculate this coefficient.

Programs The test phase was performed with programs from Collective Benchmark (cBench[5]) and Polyhedral Benchmark Suite (Polybench[6]. These programs are presented in Table 6. The inputs used for cBench and Polybench were 1 and large respectively.

Table 6. Test programs to evaluate YaCoS.

cBench Programs	
adpcm_c (C00)	patricia (C15)
adpcm_d (C01)	pgp_d (C16)
bitcount (C02)	pgp_e (C17)
blowfish_d (C03)	qsort1 (C18)
blowfish_e (C04)	rijndael_d (C19)
bzip2d (C05)	rijndael_e (C20)
bzip2e (C06)	rsynth (C21)
CRC32 (C07)	sha (C22)
dijkstra (C08)	susan_c (C23)
ghostscript (C09)	susan_e (C24)
gsm (C10)	susan_s (C25)
jpeg_c (C11)	tiff2bw (C26)
jpeg_d (C12)	tiff2rgba (C27)
lame (C13)	tiffdither (C28)
mad (C14)	tiffmedian (C29)
Polybench Programs	
2mm (P00)	gesummv (P14)
3mm (P01)	gramschmidt (P15)
adi (P02)	heat-3d (P16)
2mm (P03)	jacobi-2d (P17)
bicg (P04)	lu (P18)
cholesky (P05)	ludcmp (P19)
correlation (P06)	mvt (P20)
covariance (P07)	nussinov (P21)
deriche (P08)	seidel-2d (P22)
doitgen (P09)	symm (P23)
fdtd-2d (P10)	syr2k (P24)
floyd-warshall (P11)	syrk (P25)
gemm (P12)	trisolv (P26)
gemver (P13)	trmm (P27)

Runtime Each training program was executed 100 times to ensure accurate results. This was done during the process of creating the knowledge base. In addition, 20% of the results were discarded: the best 10% and the worst 10%. Thus, the average runtime is calculated based on 80% of the data. The test programs were executed 10 times with the same data analysis.

Parameters NEARLY and JUST were parameterized to retrieve N TA sequences. In fact, 1, 3, 5 and 10 TA sequences were evaluated. ELITE does not have a numeric parameter for retrieving TA sequences from the knowledge database.

Metrics The evaluation uses five metrics to analyze the results:

1) GMS: Geometric mean of speedups;
2) NPS: Number of programs that have a higher speedup than the most aggressive compiler optimization level (-O3);
3) IMP: Improvement related to -O3;
4) NS: Number of TA sequences evaluated; and
5) RT: The response time of the technique.

The speedup is calculated as follows:

$$Speedup = \frac{Runtime\_Level\_O0}{Runtime}.$$

The improvement is calculated as follows:

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

These metrics are also used for the final evaluation of YaCoS.

3.2 Methodology to Find the Default Similarity Coefficient

The default similarity coefficient will be the one that is capable of identifying if two curves are similar. This is done by considering the default knowledge database, behavior, and amplitude. The behavior refers to the performance about whether there was speedup or slowdown. The amplitude refers to how much gain or loss in performance there was.

It is possible to describe the behavior of program P_x using the Reaction Table (ReT) as shown in Table 7.

[5] http://ctuning.org/cbench
[6] http://www.cse.ohio-state.edu/~pouchet/software/polybench

Table 7. Behavior of P_x when compiled with `TA` sequences from S_0 to S_n.

	S_0	S_1	S_2	...	S_n
S_0	1	T_0/T_1	T_0/T_2	...	T_0/T_n
S_1	T_1/T_0	1	T_1/T_2	...	T_1/T_n
S_2	T_2/T_0	T_2/T_1	1	...	T_2/T_n
...	...	...	...	1	...
S_n	T_n/T_0	T_n/T_1	T_n/T_2	...	1

Note: yellow and white cells have the same amplitude and behavior, but in an inverse proportion. All gray cells has value 1.

In Table 7, the row i and column j represents the performance (E_{ji}) obtained by using the `TA` sequence S_j in relation to the `TA` sequence S_i, being T_k the runtime of the program when optimized by the `TA` sequence S_k.

There is a possibility to verify if there was a gain or loss in performance for each ij input. This can be seen in `ReTs`, corresponding to both P_x and P_y. The methodology consists in using the top of the diagonal to measure the behavior. Thus, for each pair $(E_{ji}(P_x),\ E_{ji}(P_y))$, the $Coeff(E_{ji}(P_x), E_{ji}(P_y))$ function measures the similarity between the behavior and amplitude of performance regarding S_x and S_y. This is calculated as follows:

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

in which $K_x = E_{ji}(P_x)$ and $K_y = E_{ji}(P_y)$.

Considering N `TA` sequences, the default coefficient obtains the highest value of *MCoeff*, which is calculated by:

$$MCoeff = \sum_{i=0}^{n} \sum_{j=i+1}^{n} \text{Coeff}(E_{ij}\ ,\ E'_{ij}).$$

where E_{ij} and E'_{ij} refer to the performance obtained by P_x and P_y respectively.

Thus, the default coefficient is found in two steps. First, we find the training program most similar to the test program and, consecutively, we calculate *MCoeff*. It is important to note that such steps are performed for each similarity coefficient defined in Subsection 2.4, in addition to several training programs.

3.3 Finding the Default Program Characterization and the Default Similarity Coefficient

Table 8 presents the most similar training program for each test program. This is done by considering the maximum value of *MCoeff* (*bMCoeff*).

Table 8. *bMCoeff*

cBench Programs					
bench	similar	value	bench	similar	value
C00	T43	973.81	C15	T03	1175.13
C01	T41	1016.72	C16	T32	981.34
C02	T41	1082.03	C17	T06	911.81
C03	T33	1087.25	C18	T08	1137.78
C04	T33	1090.80	C19	T59	660.21
C05	T56	919.08	C20	T07	552.13
C06	T48	891.61	C21	T48	1158.37
C07	T21	1118.84	C22	T06	1122.57
C08	T03	1188.44	C23	T33	906.77
C09	T22	419.72	C24	T06	927.34
C10	T36	633.14	C25	T50	1034.76
C11	T32	637.31	C26	T42	876.43
C12	T42	595.48	C27	T42	874.93
C13	T27	814.21	C28	T42	903.93
C14	T39	946.59	C29	T43	849.50
Polybench Programs					
bench	similar	value	bench	similar	value
P00	T25	1352.41	P14	T25	985.16
P01	T25	1380.98	P15	T28	913.13
P02	T07	1159.72	P16	T57	1192.77
P03	T57	1089.14	P17	T33	1083.28
P04	T29	1036.28	P18	T45	1156.71
P05	T45	1102.08	P19	T45	1071.26
P06	T07	1203.89	P20	T57	1109.52
P07	T57	1125.70	P21	T57	1066.27
P08	T45	1131.50	P22	T07	1125.84
P09	T47	1338.01	P23	T15	1204.63
P10	T07	1193.10	P24	T15	1111.42
P11	T25	1106.82	P25	T14	1227.14
P12	T20	1089.59	P26	T57	1030.43
P13	T57	1057.15	P27	T57	1205.69

The highest possible value for *bMCoeff* is 2775. This is because 75 transformation sequences (1 sequence generated for each training program, the 10 sequences found by Purini and Jain [10], and the 3 compiler optimization levels) were considered for the evaluation process. However, this value is not reached, as shown in Table 8. Thus, it is important to consider that the said metric attempts to model the behavior and amplitude reached by the performance.

It is important to identify the default (best) value even if the highest is not reached. In addition, it is indispensable to identify which similarity coefficient and characterization will produce a value that is proximate to the best one achieved.

Table 9 presents the results obtained by the strategies. The results are based on the distance to the best possible value ($\frac{MCoeff}{bMCoeff}$).

Table 9. Obtained Results

Strategy			WV	GM	BV	PR	BR
HOT	DNA	NW	0.63	0.80	1.00	1	9
		CO	0.46	0.78	1.00	2	6
	NF	EU	0.58	0.78	1.00	1	3
		JA	0.60	0.80	1.00	2	7
		SVM	0.58	0.78	1.00	1	3
FULL	PC	CO	0.49	0.76	1.00	1	6
		EU	0.55	0.73	1.00	1	1
		JA	0.58	0.77	1.00	1	4
		SVM	0.55	0.72	1.00	1	1
	CD	CO	0.46	0.77	1.00	1	6
		EU	0.59	0.79	1.00	1	5
		JA	0.50	0.76	0.98	0	13
		SVM	0.59	0.79	1.00	1	5
	NF	CO	0.48	0.81	1.00	3	13
		EU	0.58	0.78	1.00	1	8
		JA	0.59	0.79	1.00	1	10
		SVM	0.58	0.78	1.00	1	8

Note: WV: worst value; GM: geometric mean; BV: best value; PR number of perfect results; BR: number of best results

The best average value is obtained by FULL-NF with CO. However, other strategies has a performance loss of up to 10.78%.

It is worth highlighting the unexpected performance of PC, which is the only dynamic characteristic evaluated. In fact, PC has the lowest average value among the other strategies. The best value is up to 4.51%, which is worse than FULL-NF.

NF has consistent results, having the smallest variance: 0.58×10^{-4}, and 1.70×10^{-4}, for HOT and FULL respectively. In addition, other strategies has variances of 6.12×10^{-4}, and 2.77×10^{-4}, for PC and CD respectively.

The largest variance is obtained by PC. This means that although this representation achieves good performance with JA, the results with other coefficients are non-standard, reaching a difference of $\simeq 6.57\%$ between SVM and JA.

DNA has an average of 1.51% less than the best strategy. Overall, it is 30.77% worse than the best strategy. HOT, DNA and JA had satisfactory results and, thus, indicate the possibility of representing the program focusing on its hot function.

FULL-NF is the best program characterization scheme because of four reasons as follows.

1) FULL-NF has the best *MCoeff*.

2) FULL-NF has stability to achieve perfect results when the similarity coefficient is altered.

3) FULL-NF obtains one of the smallest variances among the best results.

4) The worst results of FULL-NF are not as low as the other characterizations.

CO is the best coefficient because of three reasons as follows.

1) CO has the best values of *MCoeff*.

2) CO is the coefficient that extensively presented the best results.

3) CO is among the coefficients that reach the highest number of programs with the best result.

These results indicate that FULL-NF and CO will be the default program characterizations and similarity coefficients respectively.

3.4 Finding Default Strategy to Retrieve Prior Experiences

After identifying the default program characterization and similarity coefficient, it is necessary to evaluate strategies to retrieve prior experiences.

Fig. 3 presents speedups for each recovery strategy of YaCoS, using NF-FULL and CO .

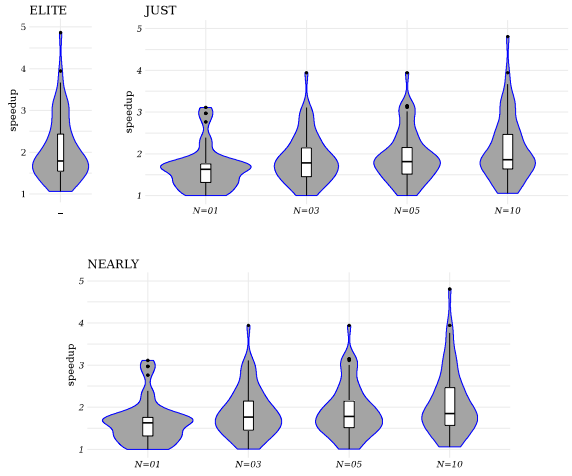


Fig. 3. Speedups for each recovery strategy.

GMS JUST.10 reaches 1.986, which is the highest average speedup and, thus, 1.05% higher than NEARLY.10 (1.976). Furthermore, it is 5.56% higher than ELITE, which obtains 1.930. JUST is

also higher than NEARLY, reaching 1.842 and 1.796 for N=5 and N=3 respectively. However, NEARLY obtains 1.829 and 1.794. NEARLY and JUST reach 1.456 and 1.453 for N=1 respectively, thus resulting in a difference of 0.3%. It is worth mentioning that JUST is able to surpass the performance of -O3 (1.838) for 5 TA sequences.

NPS JUST and NEARLY reach similar values considering the coverage metric. JUST reaches 25.86%, 50.00%, 65.52% and 87.93%, and NEARLY obtains 25.86%, 48.28%, 63.79% and 87.93% for 1, 3, 5 and 10 prior experiences respectively. However, ELITE covers 82.76% of the test programs.

IMP JUST.10 obtains the highest average improvement (18.55%) concerning -O3. NEARLY.10 and ELITE improve the programs with an average of 13.55% and 17.72%. For N=1, both JUST and NEARLY obtain an average improvement of 8.19%. However, they achieve 9.17% and 8.38% for N=3 respectively. Furthermore, NEARLY and JUST improve the test programs by 7.87% and 9.79% for N=5.

In terms of uncovered programs, JUST.10 has the smallest decline in performance (1.08%). ELITE and NEARLY.10 worsen the programs with an average of 1.90% and 1.98% respectively. The decline rates for JUST is 58.87%, 22.27% and 22.45% for 1, 3 and 5 prior experiences. NEARLY reaches 58.35%, 22.27% and 22.61% for N=1, 3, and 5 respectively.

NS ELITE is the only strategy that does not parameterize the number of TA sequences. The average value of NS for this strategy is 13.3. This means that it evaluates a larger number of TA sequences than JUST and NEARLY.

RT It is worth considering that ELITE evaluates a larger number of TA sequences than other strategies. However, it has a shorter response time (249.02s). JUST and NEARLY respond in 311.57s and 320.82s for N=10. JUST responds in 41.36s, 100.77s and 159.72s for N = 1, 3, and 5. NEARLY has a RT of 41.28s, 105.28s and 168.89s for N = 1, 3 and 5.

The highest performance for cBench programs is achieved by NEARLY.10, which reaches 1.919. JUST.10 and ELITE obtain 1.892 and 1.866 respectively. Other values worth mentioning for this set of programs are: 1.649 for NEARLY.1, 1.788 for NEARLY.3, 1.831 for NEARLY.5, 1.642 for JUST.1, 1.790 for JUST.3 and 1.833 for JUST.5. NEARLY.10 and JUST.10 have the highest coverage, reaching 83.33% for cBench programs. However, ELITE obtained 73.33%.

JUST.10 have the highest average number of speedups for Polybench programs, reaching 2.092. The highest value for NEARLY.10 and ELITE is 2.038 and 2.002 respectively. NEARLY have 1.274, 1.799 and 1.826 for 1, 3 and 5 retrieved experiences. JUST achieves 1.274, 1.804 and 1.851.

JUST has slightly better results than the other two strategies. This indicates that it is better to retrieve TA sequences from a program that has the highest similarity, but with less performance.

These results indicate that JUST has potential to be the default strategy to retrieve prior experiences.

4 Best Knowledge Database Solutions

In YaCoS, the results depend on the knowledge that is accumulated. In fact, this knowledge has to possess good solutions for input problems because it will not achieve good performance regardless of the number of TA sequences. Thus, the default YaCoS configuration (representation of the problem and coefficient) should retrieve the best possible TA sequence that exists in the knowledge database.

This section aims to evaluate the distance between the performance obtained by YaCoS and the performance obtained by compiling the test program using all TA sequences in the knowledge database (BestALL). These results may indicate the consistency of the configuration, as well as confirming JUST as the default configuration.

The results chosen for comparison in this section are the best achieved for each strategy. Table 10 presents speedups reached by NEARLY and JUST for N=10, ELITE, and BestALL.

In terms of cBench programs, the average differences are 9.91%, 7.04% and 4.59% for ELITE, JUST and NEARLY respectively. NEARLY achieves the best average proximity and the largest number of programs with the same performance for BestALL. NEARLY achieves this performance for 43.33% of cBench programs. However, ELITE and JUST obtain 33.33% and 40.00% respectively.

In regards to Polybench programs, JUST approaches 9.40% of the average performance of BestALL, while ELITE and NEARLY obtain a difference of 17.21% and 13.74% respectively. In addition, JUST and ELITE

Table 10. Results of YaCoS compared to BestALL

cBench Programs									
bench	ELITE	JUST	NEARLY	BestALL	bench	ELITE	JUST	NEARLY	BestALL
C00	1.678	1.747	2.476	2.740	C15	1.408	1.448	1.457	1.457
C01	1.371	1.372	1.567	1.764	C16	1.680	1.686	1.680	1.686
C02	3.668	3.668	3.668	3.668	C17	3.260	3.260	3.260	3.260
C03	1.812	1.812	1.823	1.834	C18	1.481	1.481	1.481	1.481
C04	1.752	1.752	1.780	1.786	C19	1.373	1.406	1.432	1.432
C05	1.505	1.505	1.505	1.505	C20	1.328	1.328	1.328	1.328
C06	1.957	1.981	1.960	1.981	C21	1.767	1.767	1.767	1.767
C07	1.069	1.106	1.112	1.112	C22	2.440	2.364	2.364	2.440
C08	1.262	1.262	1.262	1.272	C23	2.764	2.764	2.880	3.041
C09	1.061	1.061	1.063	1.063	C24	3.147	3.147	3.182	3.182
C10	2.302	2.302	2.302	2.302	C25	3.009	3.311	3.030	3.311
C11	2.009	2.009	2.009	2.009	C26	1.824	1.867	1.855	1.922
C12	2.053	2.053	2.053	2.056	C27	1.714	1.789	1.751	1.827
C13	2.331	2.261	2.261	2.331	C28	1.686	1.704	1.705	1.753
C14	2.084	2.084	2.084	2.099	C29	2.080	2.438	2.372	2.438
Polybench Programs									
bench	ELITE	JUST	NEARLY	BestALL	bench	ELITE	JUST	NEARLY	BestALL
P00	2.674	2.665	2.665	2.798	P14	3.000	2.600	2.600	3.000
P01	2.445	2.481	2.476	2.630	P15	1.753	1.753	1.753	1.868
P02	1.147	1.137	1.137	1.147	P16	2.128	2.126	2.126	2.128
P03	2.423	2.423	2.423	3.000	P17	1.315	1.315	1.315	1.315
P04	1.543	1.862	1.543	1.862	P18	1.573	1.808	1.809	1.831
P05	1.559	1.852	1.846	1.852	P19	1.620	1.874	1.873	1.893
P06	1.754	1.754	1.754	1.754	P20	1.872	2.704	2.028	2.704
P07	1.660	1.628	1.628	1.660	P21	3.221	3.221	3.764	4.160
P08	1.843	1.796	1.796	1.843	P22	1.160	1.160	1.161	1.161
P09	2.972	3.039	2.990	3.039	P23	1.774	1.780	1.780	1.793
P10	1.105	1.105	1.105	1.107	P24	1.574	1.574	1.574	1.574
P11	3.943	3.943	3.943	4.057	P25	2.583	2.583	2.583	2.583
P12	4.867	4.808	4.808	4.867	P26	1.429	2.000	1.429	2.000
P13	3.172	3.172	3.172	3.172	P27	2.172	2.303	2.172	2.303

are able to match the best performance of the knowledge database for 39.29% of the TA sequences. However, NEARLY achieves 21.43%.

Overall, the results show that JUST has an average of 6.75% lower performance than BestALL. NEARLY and ELITE reach 7.80% and 12.31% respectively. Thus, the representation and coefficient that parameterize the retrieval of TA sequences, especially with JUST, yield results proximate to the best ones available.

ELITE and BestALL obtain the same performance, 36.21% of all evaluated programs. The other 31.03% of programs perform lower than 9.83%, while another 27.59% of programs range from 11.29% to 57.69%. The remaining 5.17% obtain a difference of 83.19% to 106.23%.

In terms of NEARLY, 32.76% of evaluated programs achieve the best possible performance, 41.38% have performance below 7.59% from BestALL, 15.52% have performance that ranged from 11.41% to 28,16%, and 10.34% have distances that are between 31.92% and 67.59%.

JUST matches the performance of BestALL in 39.66% of all programs. However, 36.21% of the programs have a difference lower than 5.00% from the best possible performance. Furthermore, 13.79% of the solutions have a performance lower than 15.00%. The largest difference is achieved by 6.90% of the programs, which vary from 27.74% to 57.69%. Finally, the remaining 3.45% (2 programs) are non-standard because they obtained a distance of 93.89% and 99.28%.

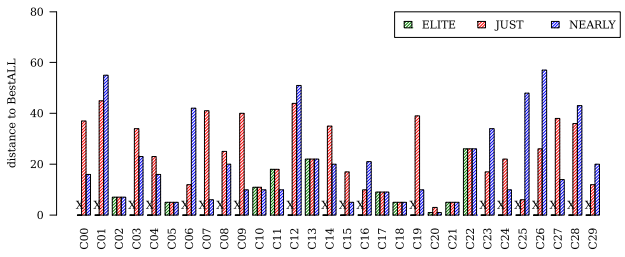
4.1 Number of TA Sequences

Ideally, the performance of BestAll should be achieved with few TA sequences; otherwise, additional code evaluations (compilation + execution) will be required. In this subsection, we analyze the number of evaluations necessary to achieve BestALL performance for each recovery strategy.

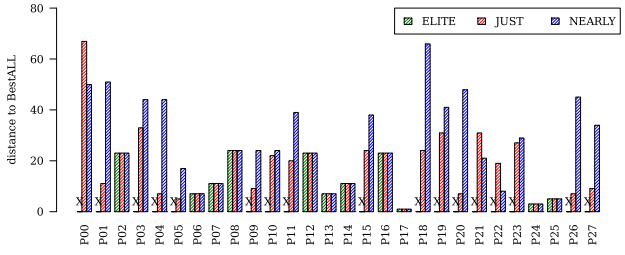
It is important to remember that ELITE is a strategy that does not parameterize the number of TA. This strategy retrieves the number of TA sequences considered good for the test program, from the training program with the highest similarity. Thus, there can be situations that it does not achieve BestALL performance because the best TA sequence may not be considered.

Fig. 4 presents the ideal values of N, for each strategy and test program, in order to obtain the performance value of BestALL. If ELITE does not reach the said value, the graph marks an X indicating where the

bar would be if ELITE achieved such a value.



(a)



(b)

Fig. 4. Number of prior experiences required in order to reach BestALL performance. (a) cBench programs. (b) Polybench programs.

In terms of cBench programs, the average TA sequences for achieving the best performance of the knowledge database are 10.90, 22.33, and 20.70 for ELITE, JUST and NEARLY respectively. However, ELITE have a coverage of 33.33%.

Considering the Polybench programs, the values of the average distance to achieve BestALL are 12.55, 17.54 and 27.18 for ELITE, JUST and NEARLY respectively. However, the coverage of ELITE is 39.29%.

The overall average number of TA sequences to reach BestALL confirms that the results of JUST are superior, achieving an average of 20.02. However, NEARLY obtains 23.83. ELITE has an average of 11.76 for TA sequences that reach BestALL. However, such range covers only 36.21% of the evaluated programs.

The standard deviation of the samples is 13.74 and 17.07 for JUST and NEARLY respectively. This means that although the values of the first strategy are proximate to the average, the results are highly non-standard. The Polybench.P00 program requires 67 evaluations to reach BestALL. However, Polybench.P17 achieves BestAll when retrieving the first TA sequence.

JUST achieves BestALL for 39.66% of the evaluated programs, retrieving 10 TA sequences (maximum situation evaluated in Subsection 2.5). However, NEARLY achieves 32.76%. NEARLY covers 46.55% of the best-performing programs, retrieving 20 TA sequences. However, JUST covers half of the programs (50.00%) with the said performance.

4.2 Average Distance

As discussed before, the number of TA sequences evaluated to reach BestALL can be high for some programs. However, there are TA sequences that have performance proximate to the best one possible.

The main question is how much does it compensate to carry out more evaluations in exchange for high performance.

The knowledge database will have better performance depending on how proximate it is to BestALL. However, a high number of evaluations means higher response time.

Fig. 5 presents the average speedup obtained after retrieving N TA sequences. The value of N for NEARLY and JUST ranged from 1 to 75, where 75 is the maximum number of TA sequences of the knowledge database. In terms of ELITE, the value depends on the number of TA sequences that have a performance superior to the most aggressive compiler optimization level, which in this case is 53.

ELITE has limitations related to the TA sequences, going up to 53. However, since there are 13.3 cases per program on average, the number of TA sequences is lower for most programs. Thus, this strategy maintains a distance of 12.31% for BestALL until 53 TA sequences are retrieved.

The distance for BestALL starts relatively high, obtaining a difference of 59.77% for $N = 1$. However, performance is becoming more proximate for every increment of N. The approximation is more significant for some TA sequences. In addition, it is important to highlight the final points of these approaches, which are small variations related to the number of evaluations that reach the best performance considerably.

Considering Δ_x as the distance for BestALL, with x retrieved TA sequences, the final points of significant approximations with N retrieved TA sequences are those that obtained the value of Δ_N more than 15% less than Δ_{N-1} and that $N + 1$ is not a significant approxima-

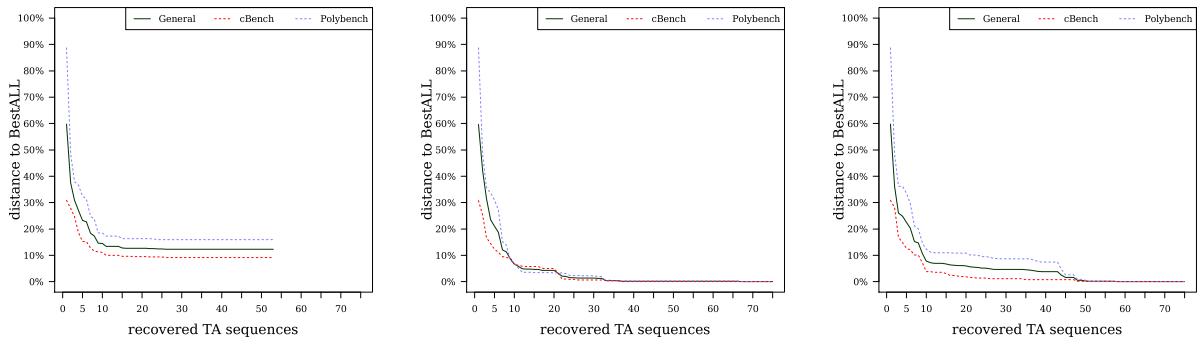


Fig. 5. Average distances to `BestALL` versus the number of past `TA` sequences retrieved. (a) `ELITE`. (b) `JUST`. (c) `NEARLY`.

tion point, being ($5 \leq N \leq 75$) and ($\Delta_{N-1} \geq 1$). It is important to not consider $N+1$ as a relevant point because of its descent, in which the lowest value of Δ is obtained. In addition, values of Δ_{N-1} greater than 1 are considered because a significant increase in speedup is obtained when a difference of 15% is reached.

The relevant points of `ELITE`, in terms of `cBench` programs, are with 5 and 7 `TA` sequences. The value of Δ drops 19.01%, obtaining an average speedup of 1.806, from 4 to 5 `TA` sequences. Furthermore, it drops to 15.55%, obtaining a speedup of 1.829, from 6 to 7 `TA` sequences.

Additionally, for `cBench` programs, `JUST` obtains significant approximation points for $N = 7$, 10, and 22. However, the values of Δ consequently decline to 15.78%, 27.33% and 67.68% respectively.

`NEARLY` obtains approximation points for 7, 10, 16, 26 and 35 prior `TA` sequences, considering `cBench` programs. The difference is 16.51% for 6 to 7 recoveries. However, the largest difference concerning Δ_{N-1} is 45.51% for Δ_{10}. The values of such a difference for $N = 16$, 26, and 35 were 23.31%, 15.16% and 27.15% respectively.

In terms of `Polybench` programs, the points considered are for $N = 7$ and 9 with `ELITE`. The decline of Δ_{N-1} is 20.89% and 22.20% for $N = 7$ and $N = 9$ respectively.

For `Polybench` programs, `JUST` has significant approximation points for 7, 9, 12, 24 and 33. These points decline 44.56%, 42.54%, 31.55%, 25.05% and 82.49% respectively.

`NEARLY` has relevant approximation points for 7, 10, 45 and 48. The differences are as follows: 28.78%, 15.68%, 42.86% and 66.49% for Δ_{N-1}.

Analyzing the overall average among the evaluated programs, the significant approximation points for `ELITE` that prevail are 7 (19.12%) and 9 (15.19%). For `NEARLY`, the approximation points are 7 (25.11%), 10 (26.39%), 45 (36.27%) and 48 (69.48%). `JUST` has the highest number of points with significant deviations considering all evaluations, being 7 (35.49%), 10 (21.43%), 12 (16.75%), 22 (36.05%), 24 (21.71%) and 33 (62.41%).

It can been seen that `JUST`, with $N \geq 12$, has a difference of less than 5% to `BestALL`, which decreases to less than 2% for $N \geq 23$. This difference reaches approximately 1% or $N > 30$. However, the performance of `BestALL` for every program is achieved with only 67 `TA` sequences. This is because of `Polybench.POO`, which had the highest number of `TA` sequences to achieve the best performance.

Although `JUST` has a relatively early approximation to `BestALL`, `NEARLY` maintains a difference of more than 5% with up to 25 `TA` sequences. However, `ELITE` can not reach more than 12.31% regardless of the number of `TA` sequences.

This analysis continues to indicate that the best strategy to retrieve similar `TA` sequences is to only recover experiences of the program with the highest similarity, even if the performance is below the best compiler optimization level. Therefore, the results confirm that `JUST` should be considered the default recovery strategy for `YaCoS` because of three reasons as follows.

1) `JUST` obtains the best overall average speedup in less than 10 evaluations.

2) `JUST` obtains the largest overall number of signifi-

cant approximations for performance differences, which means that few evaluations affect the final performance.

3) `JUST` has an early approximation of average performance with `BestALL` for the `TA` sequences evaluated.

5 Performance of Default Configuration

Two issues still need to be addressed. First, performance-wise how is `YaCoS` compared with other techniques. Second, what is the performance of `YaCoS` when generating code for complex programs. All these issues are addressed in this section.

5.1 Performance of `YaCoS` versus Other Techniques

This subsection compares `YaCoS` with three techniques in order to evaluate its effectiveness. These techniques are:

1) **`GA` with tournament selection** (`GA50`): it is similar to the technique described in Subsection 2.2, which is an iterative compilation approach.

2) **`GA` with tournament selection** (`GA10`): it is similar to `GA50`; however, it executes over 10 generations and 20 individuals.

3) **10 good `TA` sequences** (`Best10`): it is a technique proposed by Purini and Jain [10]. The authors found 10 `TA` sequences that were considered good and capable of covering several classes of programs. Thus, the unseen programs are compiled with these 10 `TA` sequences and the best target code is returned.

The first two techniques (`GA50` and `GA10`) are chosen to evaluate the performance of `YaCoS` against more aggressive techniques, which belong to the first generation of intelligent systems. Iterative compilation techniques tend to provide good results because they evaluate a considerable amount of search points. However, these techniques possess a high response time. Thus, the objective is to analyze if the performance of `YaCoS`, in terms of quality of the results, is proximate to that of more aggressive techniques. It is important to mention that the response time of `YaCoS` is less than the aforementioned iterative compilation techniques. This is because `YaCoS` evaluates a smaller number of search points. Consequently, it is expected that the response quality of `YaCoS` is better than such techniques and thus, `YaCoS` will be superior in terms of result and response quality.

`Best10` is chosen because it has good performance in terms of result quality, evaluating only 10 points in the search space. Therefore, this technique has a low response time compared to `GA50` and `GA10`. Thus, the objective is to compare `YaCoS` with a technique that has a similar response time but produces good results. This technique is recent, demonstrating that it is possible to have good performance evaluating few search points.

The evaluation of `YaCoS` is done with its default configuration: `NF-FULL`, `CO`, and `JUST` with 10 evaluations from prior experiences.

Fig. 6 presents speedups for `YaCoS`, `Best10`, `GA10` and `GA50`.

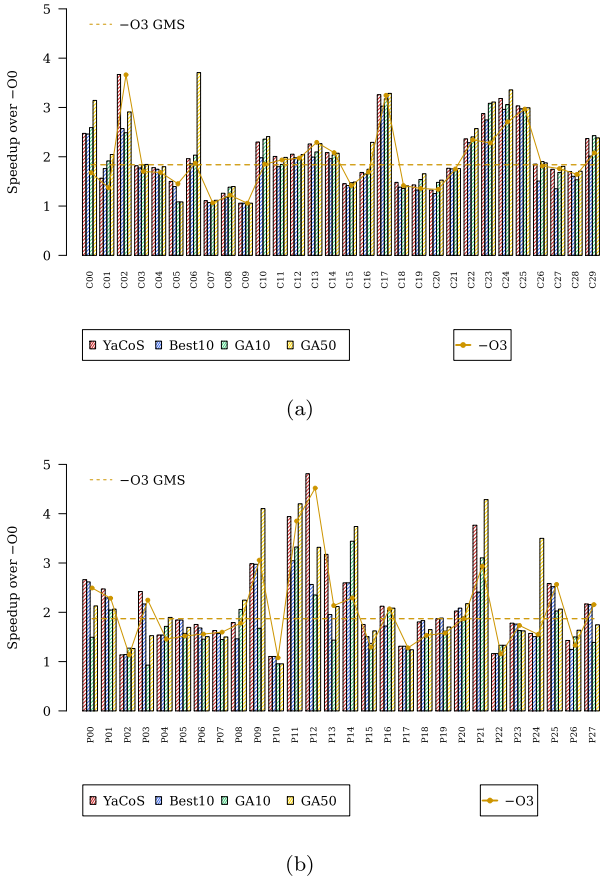


Fig. 6. Speedups for `YaCoS`, `Best10`, `GA10` and `GA50`. (a) `cBench` Programs. (b) `Polybench` Programs

`GMS` `YaCoS` reaches a `GMS` of 1.919, which is surpassed by only `GA50` (2.015). The values of `Best10` and

GA10 are 1.812 and 1.779 respectively. It is important to consider that YaCoS and GA possess different premises. The first one consists of a machine learning paradigm that returns a solution in few steps, while GA is an iterative compilation process that evaluates various TA sequences for the program. YaCoS achieves performance similar to GA50, with a difference of only 3.95% in a considerably shorter time (99%). In addition, YaCoS outperforms the other strategies.

NPS YaCoS do not reach the high speedups that GA50 obtains, however, the value for NPS is higher, covering 87.93% of the programs. GA50 has a coverage of 68.97%. Although, GA50 obtains high and non-standard values in isolated programs, YaCoS achieves good performance for more programs. Best10 and GA10 obtain 46.55% and 55.17% of coverage respectively.

IMP Considering the level -O3, YaCoS obtains an average improvement of 17.72% for covered programs. Best10, GA10 and GA50 have averages of 15.82% 22.15% and 43.20% respectively. The value that GA50 reaches is overshadowed when considering uncovered programs. Thus, there is an average decline of 27.88%. Best10 and GA10 have its programs decline by 25.11% and 43.93% respectively. YaCoS has a low average decline of 1.98%, compared to -O3, because of the high coverage.

NS Since GA10 and GA50 are iterative compilation techniques, they evaluate a high number of TA sequences to search for a result and, thus, have averages of 57.1 and 259.3 for NS, respectively. Best10 evaluates the same number of TA sequences as YaCoS (10 TA sequences). However, Best10 always evaluates the same 10 TA sequence and YaCoS recovers prior experiences based on program features.

RT GA50 is the most time-consuming strategy, taking over 69,018.0s on average to provide a result. GA10 spents an average of 13,693.8s for each input program. However, Best10 provides a response after 390.7s. YaCoS is 17.89% faster than Best10, evaluating the same number of TA sequences in an average time of 320.8s. Theoretically, the response time is essentially proportional to the number of TA sequences. However, Best10 evaluates the same 10 TA sequences for each test program. These sequences consume more time than good TA sequences retrieved by YaCoS.

For cBench programs, Best10, GA10 and GA50 obtain a GMS of 1.784, 1.882 and 2.022 respectively. YaCoS reaches 1.919, which is surpassed only by GA50; consequently, having a difference of 10.28%. In terms of the coverage, YaCoS obtains the highest NPS, covering 83.33% of the programs. Best10, GA10 and GA50 cover 40.00%, 66.67% and 80.00% of cBench programs, respectively.

Considering Polybench programs, YaCoS obtains the highest average performance among the evaluated strategies, with 2.038 for GMS. The results for Best10, GA10 and GA50 are 1.841, 1.675 and 2.008, respectively. The difference in the average performance for GA50 is 3.02%. However, YaCoS and GA50 obtain a coverage of 92.86% and 57.14%. Best10 and GA10 reach a coverage of 53.57% and 42.86% respectively.

The results indicate that iterative compilation strategies can achieve good speedups. However, it can consequently cause an increase in response time.

The metrics indicate that YaCoS is a good strategy to find good TA sequences for an input program. In addition, YaCoS surpasses other strategies. This is because as follows.

1) YaCoS provides good results in a low response time.

2) YaCoS founds solutions based on features of the input program.

3) YaCoS uses prior knowledge to return a TA sequence.

5.2 Performance of YaCoS using a Complex Benchmark

An evaluation using a complex benchmark is necessary in order to provide an effective evaluation of YaCoS. Thus, the chosen programs are part of the SPEC CPU2006 benchmark and possess more complexity than those of cBench and Polybench. The train input was used for these experiments. Table 11 presents the evaluated programs.

Table 11. SPEC CPU2006 Programs

SPEC CPU2006 Programs		
400.perlbench (S00)	445.gobmk (S06)	462.libquantum (S12)
401.bzip2 (S01)	447.dealII (S07)	464.h264ref (S13)
403.gcc (S02)	450.soplex (S08)	470.lbm (S14)
429.mcf (S03)	453.povray (S09)	471.omnetpp (S15)
433.milc (S04)	456.hmmer (S10)	473.astar (S16)
444.namd (S05)	458.sjeng (S11)	483.xalancbmk (S17)

The reasons for using the aforementioned programs are as follows.

1) It is necessary to evaluate the proposed strategy with more complex benchmarks in order to verify how much the results benefited from these experiments.

2) It is necessary to have input programs with higher runtime for evaluation purposes. These programs will increase the system response time because the solution executes target code.

3) It is necessary to evaluate different programs from which the proposed system was not calibrated. This needs to be done in order to demonstrate the results of YaCoS with various benchmarks.

The experimental environment is the same as the one utilized in the Subsection 5.1.

Fig. 7 presents the performance of Best10 and YaCoS for each SPEC CPU2006 program. This experiment does not consider continuous and background learning.

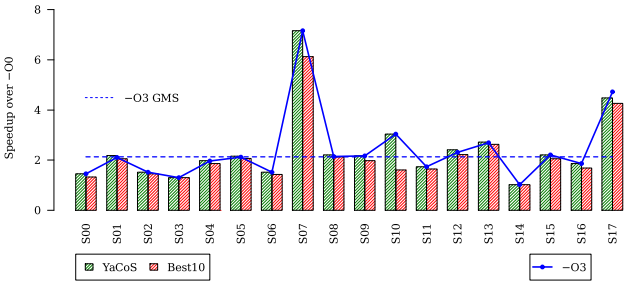


Fig. 7. Speedups obtained by applying the compared strategies.

GMS The performance that YaCoS obtains is superior to Best10, achieving a GMS of 2.144. Best10 has a performance increase of 18.88%. However, it is inferior to YaCoS, which obtained 1.944 for GMS.

NPS Best10 do not outperform -O3 in the evaluated programs. YaCoS obtains a coverage of 50.00%.

IMP YaCoS obtains an average improvement of 3.52% for programs that have performance gain related to -O3. There is only one case that there is performance loss (SPEC.S17), which has a difference of 24.98%. Best10 do not improve in regards to -O3 and declines by 25.55%.

NS The number of TA sequences recovered, for YaCoS, is parameterized with 10 evaluations. Best10 also evaluates 10 TA sequences.

RT The programs in this experiment are more time-consuming than those in Section 4. This is because of the benchmark complexity. YaCoS gets an average of 1.963,02s for 10 evaluations, and Best10 has an average of 2.570,62s. Thus, Best10 has a response time greater than YaCoS for the same number of evaluations. This is because Best10 always evaluates the same 10 TA sequences. However, YaCoS evaluates 10 TA sequences that are most appropriate for the input program.

5.2.1 Comparison with the Best Database Solutions

Fig. 8 presents the performance difference obtained by YaCoS compared with the best possible solutions in database.

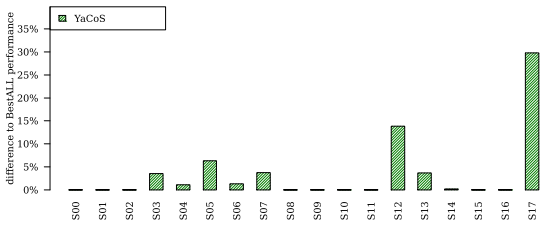


Fig. 8. Performance difference of YaCoS and BestALL.

The biggest difference for BestALL occurs at S17, which has a distance of 32.77% to reach the best performance found in the database.

The best possible solution is achieved for 50.00% of the programs. Considering other 33.33% programs, the performance difference for BestALL ranges from 0.20% to 3.74%. The remaining 16.67% obtain a performance difference ranging from 6.34% to 32.77%.

Furthermore, the average performance reaches a difference of 3.53% with the best solution.

Fig. 9 presents the average performance distance obtained for several number of `TA` sequences.

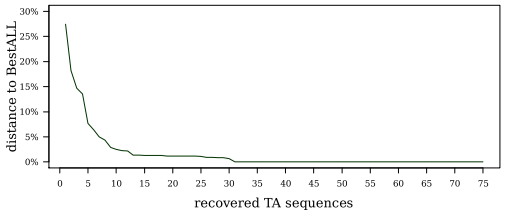


Fig. 9. Average distance to reach the performance of `BestALL`.

An average speedup of 1.894 is obtained when retrieving only 1 prior experience and, consequently, has a difference of 27.45% from the best possible result of the knowledge database. This difference declines to less than 7.68% when analyzing 5 experiences. In addition, it declines to 2.87% when analyzing 9 prior `TA` sequences.

It is possible to observe that from 12 to 13 retrieved `TA` sequences, the distance for `BestALL` is reduced by 36.90%, reaching an average speedup of 2.156%. This is because 13 is a sharp curve point for results of `SPEC` programs. Other curve points for the results are 7 (21.53%) and 9 (33.81%).

The distance of performance obtained by `YaCoS` and the best possible solution in database is only 0.96%. This is for an experiment that evaluates 26 prior experiences.

The complexity of `SPEC CPU2006` programs gives more complexity to obtain performance. Although in previous experiments `YaCoS` reached a coverage of 87.93% for `cBench` and `Polybench` programs with 10 evaluations, it reaches 66.67% with 23 evaluations for `SPEC CPU2006`. All speedups, including for `BestAll`, have an improvement of 3.57%, which is proximate to `-O3` for `SPEC` programs. Furthermore, these sequences have a 28.87% improvement for `Polybench` programs, which is also proximate to `-O3`.

Despite the complexity of the evaluated programs, `YaCoS` has been proved to be effective. This is because it offers improvements using a small number of evaluations. However, these evaluations require code execution. Consequently, this provides a cost that can be controlled only by user input. Thus, more complex programs demand a higher response time, which is the case with `SPEC` programs.

The strategy is adequate when a higher performance is desired. Thus, it can be applied to programs for embedded devices and network control. A balance must be found between immediate user requirements and necessary performance of the generated code. The types of programs mentioned tend to carry out executions with limited resources and thus, provides a long time period to obtain performance gain.

5.2.2 *Evaluating the Use of Continuous and Background Learning*

To evaluate the use of continuous and background learning, Δ_T time is considered between the compilation of two programs. This variable represents the time required to feed the database with knowledge about the first program. In addition, the compile order of programs can influence the performance because the knowledge database will be different for each new compilation.

It is not possible to determine the compile order of the programs. Thus, this experiment considers that the programs are compiled in alphabetical order.

Fig. 10 presents the results of `YaCoS`, using continuous and background learning.

`GMS` The average speedups are slightly higher (less than 1%) compared with the results without feedback, reaching 2.152 for `GMS`.

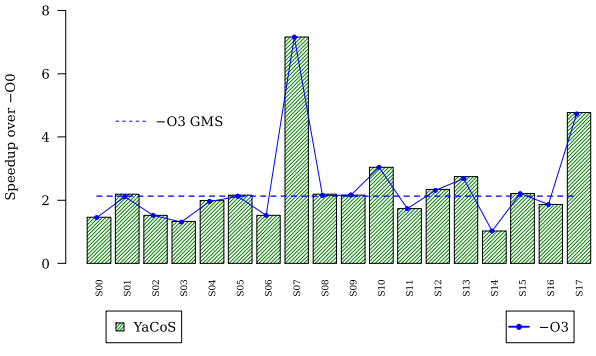


Fig. 10. Speedup obtained by `YaCoS`.

`NPS` The coverage is 11.11% higher than an execution whithout feedback, reaching 61.11% of the evaluated programs.

`IMP` Although improvement is achieved for a large number of programs, the average improvement related to `-O3` decreases to 3.22%. Consequently, this has a difference of 0.3% for results without feedback. In terms of the performance loss, concerning `-O3`, only `S10` has a loss of 0.15%. The system without feedback has a single loss as well. However, `S17` reaches 24.98%.

`NS` The number of `TA` sequences is parameterized and used as the system without feedback (10).

`RT` The system response time decreases by 2.82%, responding after 1,907.39 seconds for each program. The response time does not increase considerably because of the compilation and background process responsible for adjusting solutions of the knowledge database to the new program.

`YaCoS` has been proved to be effective in finding `TA` sequences that provide performance gain for programs. This can also be stated for programs unseen by the compiler during its training and use.

The use of continuous compilation is interesting because it allows for continuous learning of the compiler, associating features of new programs compiled with solutions in the knowledge database.

In addition, background compilation allows the response time of the strategy to remain the same and, consequently, does not block the user's response for acquiring new knowledge about the system.

6 Related Work

The first generation of intelligent code-generating systems employs iterative compilation techniques, which consist of programs compiled with several `TA` sequences until the best target code is found. Based on the search behavior, these systems can be classified as:

1) intelligent systems using partial search;

2) intelligent systems using random search;

3) intelligent systems using heuristic search.

Intelligent systems using partial search attempt to explore a portion of possible results [8, 27, 28]. Intelligent systems using random search aim to find a solution using statistical and/or random techniques, striving to reduce the number of evaluated `TA` sequences [29, 30, 31]. Intelligent systems using heuristic search use heuristics to reduce the quantity of evaluated `TA` sequences [32, 33, 34].

In terms of first generation systems, Cohen *et al.* [35] proposed a strategy to simplify the search through `TA` sequences. This strategy was based on polyhedral representations. This approach was implemented in a framework, which was done for simplifying comparisons between `TA`s. This technique depends on a unified polyhedral representation that contains loops and statements.

Based on polyhedral representations, Pouchet *et al.* [36] proposed an iterative compilation system focusing on loop transformations which reduces the search space. Therefore, it is possible to apply an extensive search and gain performance in small kernels. Although there is high difficulty in applying the proposed system to large programs, the implemented heuristic technique is capable of enumerating the initial search, consequently, mapping performance distribution by transformation spaces.

Pouchet *et al.* [37] proposed an iterative compilation system for finding `TA` sequences. The authors focused on nested loops, which are adapted for search space properties of the polyhedral model. The system is implemented with both heuristics and a genetic algorithm with specialized operators, which further improves the approaches. Furthermore, these strategies improved the previous system, having performance gains in small kernels and large benchmarks with either a small or big search space.

Another interesting work was proposed by Purini and Jain [10] in the context of first generation systems. Purini and Jain reduced the system response time by using `TA` sequences capable of covering multiple programs. The process of finding such `TA` sequences is as follows. First, the strategy applies heuristic and random searches to select the best `TA` sequence for each training program. Second, the `TA` sequences that do not contribute to performance gain (exceeding the most aggressive level of the compiler) are eliminated. Finally, a coverage algorithm analyzes and extracts the 10 best `TA` sequences. As a result, this strategy evaluates only 10 `TA` sequences to find the best one for a new program.

It is important to note that such studies focus on finding a good configuration for the compiler and, consequently, generating a high quality target code. Based on this premise, CUI *et al.* [38] developed a methodology in which the compiler is capable of identifying `TA` sequences for a specific algorithm. This work is interesting because the authors developed a methodology to

encapsulate TAs in optimization patters, which may be reused in occurring scenarios. However, this work differs from the aforementioned work, including YaCoS, because the programmer annotates the code as patterns, allowing the code-generating system to generate good-quality target code.

Although the first generation of intelligent systems performs well, it does not accumulate knowledge and thus, possess a high system response time. Because of this, it is decided that an intelligent system based on iterative compilation will not be implemented in this work.

The second generation of intelligent code-generating systems employs machine learning techniques. These systems are designed to be capable of not only extracting and classifying knowledge but also accumulating and reusing it. Therefore, a low system response time will be provided to the user.

The second generation systems create a prediction model in the training phase, based on the behavior of training programs. Afterwards, a prediction model in the test phase finds the best TA sequence for a program unseen by the system [39, 40, 12].

The prediction model creates a relationship between effective TA sequences and program features. The first step consists in creating these relationships in order to find good TA sequences for a set of training programs. Therefore, it is necessary to apply an iterative compilation technique. The second step consists in extracting and classifying features of the training programs. These features can be: performance counters [11, 12], control flow graphs [41], compilation data [13], features describing loops and vector structures of the program [39], numeric features [16, 4], or a symbolic representation similar to a DNA [17, 42]. After defining the representation, it is possible to relate it to the TA sequences and, thus, build a prediction model.

The test phase can be implemented using various strategies, such as instance-based learning [39], logistic regression [11] or case-based reasoning [12, 13]. These strategies provide the TA that should be activated [11] and utilized [12, 13].

An interesting work was proposed by Tartara and Reghizzi (2013) [4] based on second generation systems. They presented an intelligent system that applies a hybrid technique for long-term learning and, consequently, eliminates the training phase. Additionally, Tartara and Reghizzi proposed extracting the program features first, and then the evolutionary algorithm provides the TA sequence to generate the target code.

Table 12 summarizes related work in order to distinguish our contribution from them.

As shown in Table 12, YaCoS can be classified as a second generation system, which extracts, classifies and accumulates knowledge. YaCoS can be parameterizable and, thus, offers flexibility for different strategies. Another point worth highlighting is that YaCoS can employ background compilation, while its goal is to exploit the available computing resources in order to improve the knowledge database.

7 Conclusion and Future Work

This paper presented YaCoS, a flexible and low-cost intelligent code-generating system capable of retrieving good code transformation algorithms for test programs.

YaCoS is proved to be flexible to work with different program features, similarity coefficients and recovery strategies for prior experiences. In addition, YaCoS provides continuous and background learning.

7.1 Configuration

Knowledge extraction through NF, proposed by Namolaru *et al.* [16] and considering the entire program, has proved to be more efficient than other representations. NF obtains the best average for evaluating reactions and one of the smallest variances between the similarity coefficient values.

Among the possible similarity coefficients, the intelligent system can use four to obtain data numerically and one to compare character strings. In the evaluation, CO extensively presents the best results for a large number of programs, thus being the best coefficient.

In terms of the strategy to recover prior experiences, JUST presents the best results on two aspects as follows.

1) The performance achieved by JUST recoverig 10 prior experiences.

2) The difference between performance achieved by JUST and the best possible performance based on the knowledge database.

This indicates that it is better to retrieve experiences applied to similar programs even if they are not good. This is a better alternative than either retrieving experiences considered good for a specific training program (ELITE); or training programs with proximate similarity (NEARLY). In addition, increasing the number of evaluations does not compensate for improving

Table 12. Comparison of intelligent systems that select `TA` sequences.

Authors	Year	Techniques	Architecture	Benchmark	Compiler
Cohen et al [35]	2005	Iterative Compilation, Polyhedral model, `TA`s downsampling	AMD Athlon XP 2800+ 2.08GHz 512MB RAM	SPECfp2000	GCC 3.4
Pouchet et al [36]	2007	Iterative Compilation, Polyhedral model, `TA`s downsampling	Intel Xeon 3.2GHz	UTDSP	GCC 3.4.2, GCC 4.1.1, ICC 9.0.1, EKOPath 2.5
Pouchet et al [37]	2008	Iterative Compilation, Polyhedral model	AMD Alchemy Au1500 500MHz	UTDSP	GCC 3.2.1, GCC 4.1.1, st200cc 1.9.0B
CUI et al [38]	2012	Pattern discovery, Pragmas	Intel Xeon, NVIDIA GTX285, Godson-T	BLAS, SPEC CPU2000, NPB-3.3, MGMRES, CUDA-SDK	Open64
Purini and Jain [10]	2013	Iterative Compilation, `TA`s downsampling	Intel Xeon W3550 3.06GHz 12GB RAM	Polybench, Mibench	LLVM 3.0
Lima et al [12]	2013	Machine Learning, Training Phase, Case-based Reasoning	Intel Core i7-3770 3.40GHz 4GB RAM	Polybench	LLVM 3.1
Tartara and Reghizzi [4]	2013	Long-term Learning, Continuous Learning	Intel Xeon X7550 2.00 GHz 128GB RAM	cBench, Mibench	GCC 4.6.3
Queiroz and Silva [13]	2015	Machine Learning, Training Phase, Case-based Reasoning	Intel Core i7-3779 3.40GHz 8GB RAM	cBench, SPEC CPU2006	LLVM 3.5
`YaCoS`	2017	Machine Learning, `TA`s downsampling, Training Phase, Case-based Reasoning, System configuration, Continuous Learning, Background Compilation	Intel Core i7-3770 3.40GHz 8GB RAM	Polybench, cBench, SPEC CPU2006	LLVM 3.7.1

the quality of the results. This is because the results will not considerably improve without a high response time and, consequently, making the use of the machine learning-based system impractical.

7.2 Results

`YaCoS` is able to surpass other strategies by 21.36%, and cover 50.00% of the programs using complex benchmarks. These results indicate an improvement up to 17.72% in terms of the most aggressive compiler optimization level of `LLVM` (`-O3`).

The use of continuous learning improved performance results (slightly more than 1% better) and coverage (superior by 11%).

The additional cost of continuous learning do not result in a long response time because of background learning. This strategy is capable of using more efficiently the resources available in the architecture, performing tasks transparently to the user. Thus, in the experiments with continuous and background learning, the response time is less than when the said strategy is disabled.

7.3 Future Work

Several issues can be improved in order to maximize the system efficiency and, consequently, its performance. It is also possible to emphasize points about the relationships between transformation algorithms and structures of the source program, which would be helpful in choosing the best transformation algorithm sequences. Thus, the issues to be explored are:

Construction of the knowledge database. The proposed knowledge database is built by reducing the search space that contained known transformation algorithm sequences and results of iterative compilation strategies. In addition to this strategy, others should be explored, namely: use of different benchmarks, heuristic, and metaheuristic algorithms.

Learning without an initial database. To avoid the cost of building a knowledge database, an alternative is to initialize the system without a database. In this case, the system must permit the creation of new solutions throughout its use, in addition to enabling continuous learning. Thus, the system would create new solutions on

demand.

Static evaluation. A static evaluation would avoid executing generated code to measure performance, which would lower the system cost. In this case, the cost of evaluating a solution would be proportional to the size of the source code, and not dependent on the program's runtime. One way to perform such an assessment is to generate profiles of static code structures and apply a static analysis.

Other strategies for continuous learning. The proposed continuous learning strategy consists in adjusting the solutions in the knowledge database for a new program. However, it is possible to use other forms of learning, such as applying iterative compilation techniques. Although iterative compilation techniques possess a high response time, background compilation allows the reduction of such a cost. In addition, the use of background compilation does not add a cost to the user response time.

Recovery of a single sequence. In order to avoid evaluating several target codes, it is necessary to recover only one transformation algorithm sequence (the best according to the `TA` sequences of the knowledge database). The retrieval of only one `TA` sequence requires a considerable maturity of the learning system. Thus, it is necessary to improve the learning techniques by providing additional details about the program structures and their relationships with code transformation algorithms.

Acknowledgments

We thank CNPq (National Council for Scientific and Technological Development) for the financial support towards this study. This is the author's version of this paper, the final publication is available at www.springerlink.com

References

[1] Ula M, Mursyidah, Hendriana Y, Hardi R. An expert system for early diagnose of vitamins and minerals deficiency on the body. In *Proc. International Conference on Information Technology Systems and Innovation*, Oct 2016, pp. 1–6.

[2] Muntean M V, Donea A. A hybrid intelligent agent based expert system for gps databases. In *Proc. International Conference on Electronics, Computers and Artificial Intelligence*, June 2016, pp. 1–4.

[3] Nascimento R J O, Fonseca C A G, Medeiros Neto F D. Using expert systems for investigating the impact of architectural anomalies on software reuse. *IEEE Latin America Transactions*, 2017, 15(2):374–379.

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

[5] Aho A V, S L M, Sethi R, Ullman J D. *Compilers: Principles, Techniques and tools*. Prentice Hall, 2006.

[6] Cooper K, Torczon L. *Engineering a Compiler*. Morgan Kaufmann, USA, 2nd edition, 2011.

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

[8] Pan Z, Eigenmann R. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proc. International Symposium on Code Generation and Optimization*, 2006, pp. 319–332.

[9] Park E, Kulkarni S, Cavazos J. An evaluation of different modeling techniques for iterative compilation. In *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2011, pp. 65–74.

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

[11] Cavazos J, Fursin G, Agakov F, Bonilla E, O'Boyle M F P, Temam O. Rapidly selecting good compiler optimizations using performance counters. In *Proc. International Symposium on Code Generation and Optimization*, 2007, pp. 185–197.

[12] Lima E D, Souza Xavier T C, Silva A F, Ruiz L B. Compiling for performance and power efficiency. In *Proc. International Workshop on Power and Timing Modeling, Optimization and Simulation.*, Sept 2013, pp. 142–149.

[13] Queiroz Junior N L, Silva A F. Finding good compiler optimization sets. In *Proc. International Conference on Enterprise Information Systems*, 2015, pp. 504–515.

[14] Lattner C, Adve V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. International Symposium on Code Generation and Optimization*, Mar 2004, pp. 75–86.

[15] 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*, 2016, 13(1):1–28.

[16] Namolaru M, Cohen A, Fursin G, Zaks A, Freund A. Practical aggregation of semantical program properties for machine learning based optimization. In *Proc. International Conference on Compilers Architectures and Synthesis for Embedded Systems*, 2010, pp. 197–206.

[17] Sanches A, Cardoso J M P. On identifying patterns in code repositories to assist the generation of hardware templates. In *Proc. International Conference on Field Programmable Logic and Applications*, 2010, pp. 267–270.

[18] Wu Y, Larus J R. Static branch frequency and program profile analysis. In *Proc. International Symposium on Microarchitecture*, 1994, pp. 1–11.

[19] Ball T, Larus J R. Branch prediction for free. *SIGPLAN Notices*, 1993, 28(6):300–313.

[20] Shafer G. *A Mathematical Theory of Evidence*. Princeton University press, 1976.

[21] Scholkopf B. *Learning With kernels - Support Vector Machines*. MIT Press, San Francisco, CA, USA, 2002.

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

[23] Tanenbaum A S, Goodman J R. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

[24] Patterson D A, Hennessy J L. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[25] Chen Y, Fang S, Eeckhout L, Temam O, Wu C. Iterative optimization for the data center. *SIGPLAN Notices*, March 2012, 47(4):49–60.

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

[27] Kulkarni P A, Whalley D B, Tyson G S, Davidson J W. Practical exhaustive optimization phase order exploration and evaluation. *ACM Transactions on Architecture and Code Optimization*, April 2009, 6(1):1–36.

[28] Foleiss J H, Silva A F, Ruiz L B. An experimental evaluation of compiler optimizations on code size. In *Proc. Brazilian Symposium on Programming Languages*, 2011, pp. 1–15.

[29] Haneda M, Knijnenburg P M W, Wijshoff H A G. Generating new general compiler optimization settings. In *Proc. Annual International Conference on Supercomputing*, 2005, pp. 161–168.

[30] Shun L, Fursin G. A heuristic search algorithm based on unified transformation framework. In *Proc. International Conference Workshops on Parallel Processing*, 2005, pp. 137–144.

[31] Cooper K D, Grosul A, Harvey T J, Reeves S, Subramanian D, Torczon L, Waterman T. Exploring the structure of the space of compilation sequences using randomized search algorithms. *Journal of Supercomputing*, 2006, 36(2):135–151.

[32] Kulkarni P A, Hines S R, Whalley D B, Hiser J D, Davidson J W, Jones D L. Fast and efficient searches for effective optimization-phase sequences. *ACM Transactions on Architecture and Code Optimization*, 2005, 2(2):165–198.

[33] Che Y, Wang Z. A lightweight iterative compilation approach for optimization parameter selection. In *International Multi-Symposiums on Com-*

puter and Computational Sciences, volume 1, 2006, pp. 318–325.

[34] Zhou Y Q, Lin N W. A study on optimizing execution time and code size in iterative compilation. In *Proc. International Conference on Innovations in Bio-Inspired Computing and Applications*, Sept 2012, pp. 104–109.

[35] Cohen A, Sigler M, Girbal S, Temam O, Parello D, Vasilache N. Facilitating the search for compositions of program transformations. In *Proc. International Conference on Supercomputing*, 2005, pp. 151–160.

[36] Pouchet L N, Bastoul C, Cohen A, Vasilache N. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Proc. International Symposium on Code Generation and Optimization*, March 2007, pp. 144–156.

[37] Pouchet L N, Bastoul C, Cohen A, Cavazos J. Iterative optimization in the polyhedral mode: Part ii, multidimensional time. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 90–100.

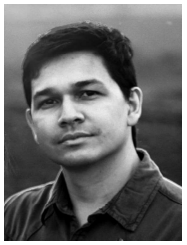
[38] Cui H, Xue J, Wang L, Yang Y, Feng X, Fan D. Extendable pattern-oriented optimization directives. *ACM Transaction on Architecture and Code Optimization*, 2012, 9(3):1–37.

[39] Long S, O'Boyle M. Adaptive java optimisation using instance-based learning. In *Proc. International Conference on Supercomputing*, 2004, pp. 237–246.

[40] Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle M F P, Thomson J, Toussaint M, Williams C K I. Using machine learning to focus iterative optimization. In *Proc. International Symposium on Code Generation and Optimization*, 2006, pp. 295–305.

[41] Park E, Cavazos J, Alvarez M A. Using graph-based program characterization for predictive modeling. In *Proc. International Symposium on Code Generation and Optimization*, 2012, pp. 196–206.

[42] Martins L G, Nobre R, Delbem A C, Marques E, Cardoso J a M. Exploration of compiler optimization sequences using clustering-based selection. *SIGPLAN Notices*, 2014, 49(5):63–72.



João Fabrício Filho is currently a PhD student in computer science at the University of Campinas in Brazil. He received his Bachelor's degree in informatics and a Master's degree in computer science from State University of Maringá, Brazil, in 2014 and 2016, respectively. His research interests includes high performance computing, approximate computing, parallel programming and compiler.

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

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