This is the post peer-review accepted manuscript of:

Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, Cristina Silvano
COBAYN: Compiler Autotuning Framework using Bayesian Networks
ACM Transactions on Architecture and Code Optimization

The published version is available online at: http://dx.doi.org/10.1145/2928270

©**2018 ACM**. Personal use of this material is permitted. Permission from the editor must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

COBAYN: Compiler Autotuning Framework using Bayesian Networks

AMIR HOSSEIN ASHOURI, Politecnico di Milano, ITALY
GIOVANNI MARIANI, IBM, The NETHERLANDS
GIANLUCA PALERMO, Politecnico di Milano, ITALY
EUNJUNG PARK, Los Alamos National Laboratory, USA
JOHN CAVAZOS, University of Delaware, USA
CRISTINA SILVANO, Politecnico di Milano, ITALY

The variety of today's architectures forces programmers to spend efforts for porting and tuning application codes across different platforms. Compilers themselves need additional tuning which has considerable complexity as the standard optimization levels, usually designed for the average case and the specific target architecture, quite often fail to bring the best results.

This paper proposes *COBAYN*: COmpiler autotuning framework using BAYesian Networks, an approach for a compiler autotuning methodology using machine learning to speed up application performance and to reduce the cost of the compiler optimization phases. The proposed framework is based on the application characterization done dynamically by using independent micro-architecture features and Bayesian networks. The paper also presents an evaluation based on using static analysis and hybrid feature collection approaches. In addition, the paper compares Bayesian networks with respect to several state-of-the-art machine-learning models.

Experiments were carried out on an ARM embedded platform and GCC compiler by considering two benchmark suites with 39 applications. The set of compiler configurations selected by the model (less than 7% of the search space), demonstrated an application performance speedup of up to 4.6× on Polybench (1.85× on average) and 3.1× on cBench (1.54× on average) with respect to standard optimization levels. Moreover, the comparison of the proposed technique with (i) random iterative compilation, (ii) machine learning-based iterative compilation and (iii) non-iterative predictive modeling techniques, shows on average, 1.2×, 1.37× and 1.48× speedup, respectively. Finally, the proposed method demonstrates 4× and 3× speedup, respectively on cBench and Polybench, in terms of exploration efficiency given the same quality of the solutions generated by the random iterative compilation model.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

CCS Concepts: •**Software and its engineering** → **Compilers;** •**Computing methodologies** → **Supervised learning;**•**Mathematics of computing** → **Bayesian networks;**

General Terms: Autotuning, Compilers, Machine Learning, Performance Evaluation

Additional Key Words and Phrases: Bayesian Networks, Statistical Inference, Design Space Exploration

ACM Reference Format:
Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos and Cristina Silvano, 2016. COBAYN: Compiler Autotuning using Bayesian Networks *ACM Trans. Architec. Code Optim.* , , Article ( ), 25 pages.
DOI: http://dx.doi.org/10.1145/2928270

1. INTRODUCTION

Usually, software applications are developed in a high-level programming language (e.g. C, C++) and then passed through the compilation phase to get the executable binary. Optimizing the second phase (*compiler optimization*) plays an important role for the performance metrics.

Author's addresses: Amir H. Ashouri, Via Giuseppe Ponzio, 34/5, DEIB, Politecnico di Milano, ITALY
ACM acknowledges that this contribution was authored or co-authored by an employee, or contractor of the national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Permission to make digital or hard copies for personal or classroom use is granted. Copies must bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. To copy otherwise, distribute, republish, or post, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© ACM. 1544-3566//-ART $15.00
DOI: http://dx.doi.org/10.1145/2928270

In other words, enabling compiler optimization parameters (e.g. loop unrolling, register allocation, etc.) might lead to substantial benefits in several performance metrics. Depending on the strategy, these performance metrics could be *execution time, code size* or *power consumption*. A holistic exploration approach to trade-off these metrics also represents a challenging problem [Palermo et al. 2005].

Application developers usually rely on compiler intelligence for software optimization, but they are unaware of *how* the compiler itself does the job. Compiler interface usually has some standard optimization levels which enable the user to automatically include a set of predefined optimization sequences for the compilation process [Hoste and Eeckhout 2008]. These standard optimizations (e.g. -O1, -O2, -O3 or -Os) are known to be beneficial for performance (or code size) in most cases. In addition to the above-mentioned standard optimizations, there are other compiler optimizations which are not included in the predefined optimization levels. Their effects on the software are quite complex and mostly depend on the features of the target application. Therefore, it is rather hard to decide whether to enable specific compiler optimizations on the target code. Considering application-specific embedded systems, the compiler optimization task becomes even more crucial because the application is compiled once and then deployed on millions of devices on the market.
So far, researchers proposed two main approaches for tackling the problem of identifying the best compiler optimizations: i) *iterative compilation* [Chen et al. 2012] and ii) *machine-learning predictive modeling* [Agakov et al. 2006]. The former approach relies on several re-compilation phases and then selecting the best set of optimizations. Obviously this approach, although effective, has high overhead as it needs to be evaluated iteratively. The latter approach focuses on building machine-learning predictive models to predict the best set of compiler optimizations. It relies on software features that are collected either *offline* or *online*. Once the model has been trained, given a target application, it can predict a sequence of compiler optimization options to maximize performance. Machine learning approaches need fewer compilation try-outs, but the downside is typically represented by the performance of the final execution binary, which is worse than the one found with iterative compilation.

In this work[1], we propose an approach to tackle the problem of identifying the compiler optimizations that maximize the performance of a target application. Differently from previous approaches, the proposed work starts by applying a statistical methodology to infer the probability distribution of the compiler optimizations to be enabled. Then, we start to drive the iterative compilation process by sampling from this probability distribution. We use two major sets of training application suites to learn the statistical relations between application features and compiler optimizations. To the best of our knowledge, in this work, *Bayesian Networks* (BN) are used for the first time in this field to build the statistical model. Given a new application, its features are fed into the machine-learning algorithm as *evidence* on the distribution. This evidence imposes a bias on the distribution, and because compiler optimizations are correlated with the software features, we can iteratively sample the distribution obtaining the most promising compiler optimizations, by then exploiting an *iterative compilation* process.

The experiments carried out on an embedded ARM-based platform outperformed both standard optimization levels and the state-of-the-art iterative and not iterative (based on prediction models) compilation techniques, while using the same number of evaluations. Moreover, the proposed techniques demonstrated significant exploration efficiency improvement of up

[1]This article is an extended version of our previous work [Ashouri et al. 2014], providing additional details about **i)** the different feature selection techniques and introducing a new method by combining them as *hybrid*, **ii**) revising the machine-learning part, taking into account different aspects of statistical tuning and their performance comparisons, **iii**) adding more benchmarks and data-sets, and **iv**) introducing new chapter offering a holistic comparison with different state-of-the-arts machine-learning algorithms.

to 4× speedup compared with random iterative compilation when targeting the same performance. To summarize, our work contributes to the following:

— The introduction of a BN capable of capturing the correlation between the application features and the compiler optimizations. This enables us to represent the relation by an acyclic graph, which can be easily analyzed graphically.
— The integration of the BN model in a compiler optimization framework. Given a new program, the probability distribution of the best compiler optimizations can be inferred by means of BN to focus on the optimization itself.
— The integration of both dynamic and static analysis feature collections in the framework as hybrid features.

Furthermore, the experimental evaluation section reports the assessment of the proposed methodology on an embedded ARM-based platform and the comparison of the proposed methodology with several state-of-the-art machine learning algorithms on 39 different benchmark applications.

The remainder of the paper is organized as follows. Section 2 presents an review of recent related literature. Section 3 presents how the BN model can infer the probability of the distribution. Section 3.1 presents different techniques for collecting program features. Section 4 elaborates on the proposed framework. Sections 4.4 and 4.5 will introduce the results obtained on the application suites selected. Finally Section 4.6 presents the comparison of the proposed methodology with state-of-the-art models.

2. PREVIOUS WORK

Optimizations carried out at compilation have been broadly used, mainly in embedded computing applications. This makes such techniques especially interesting, and researchers are investigating more efficient techniques for identifying the best compiler optimizations to be applied given the target architecture. There are two major classes of optimization in the field of compiler: (i) The problem of *selecting the best compiler optimizations* and (ii) The *phase-ordering* problem of compiler optimizations. As the target of this work is in the scope of *selection*, here we mostly refer to these areas. However, there are notable works to be mentioned that support the seminal concepts of the current work.

The related work in this field can be categorized into two sub-classes: *(a) iterative compilation* [Bodin et al. 1998] and *(b) machine-learning* based approaches [Cooper et al. 1999; Kisuki et al. 2000]. Nonetheless, these two approaches have also been combined in many ways [Agakov et al. 2006] that they cannot be distinguished easily.

Iterative compilation was introduced as a technique capable of outperforming static hand-crafted optimization sequences, those usually exposed by compiler interfaces as *optimization levels*. Since its introduction [Bodin et al. 1998; Kisuki et al. 1999], the goal of iterative compilation has been to identify the most appropriate compiler passes for a target application.

More recent literature discusses the use of down-sampling techniques to reduce the search space [Purini and Jain 2013] with the goal of identifying compiler optimization sequences for each application. Other authors exploit iterative compilation jointly with architectural design space exploration for VLIW architectures [Ashouri et al. 2013]. The intuition was that the performance of a computer architecture depends on the executable binary which in turn, depends on the optimizations applied at compilation time. Thus, by studying the two problems jointly, the final architecture is optimal in terms of the compilation technique in use and the effects of different compiler optimizations are identified at the early design stages. The authors of [Tang et al. 2015], proposed an autotuning framework targeting scale-free sparse matrix-vector multiplication by employing 2D jagged partitioning and tiling to achieve good cache efficiency and work balancing. Furthermore, [Mehta and Yew 2015] have addressed compiler scalability by reducing the effective number of statements and dependencies as seen

by the compiler through Integer Linear Programming (ILP). Petabricks [Ansel et al. 2009] has been introduced as a language for programmers to naturally express algorithmic choices explicitly so as to empower the compiler to perform deeper optimizations. They have developed an autotuner that is fed by a Choice Dependency Graph and interacts with the parallel runtime to optimize the binary code. On a High Performance Computing (HPC) level, [Schkufza et al. 2014] introduced an aggressive floating-point optimization using random search techniques to both eliminate the dependence on expert-written optimization rules and allow a user to customize the extent to which precision is sacrificed in favor of performance. OpenTuner [Ansel et al. 2014], is a framework for building domain-specific multi-objective program autotuners. It introduces the concept of ensembles of search techniques in autotuning, which allow many search techniques to work together to find an optimal solution and provide a more robust search than a single technique alone. Another recent iterative compilation is introduced by [Fang et al. 2015]. Authors proposed iterative optimization for the data center (IODC) by spawning different combinations across workers and recollect performance statistics at the master, which then evolves to the optimum combination of compiler optimizations and to manage the cost and benefits.

Given that compilation is a time-consuming task, several groups proposed techniques to predict the best compiler optimization sequences rather than applying a trial-and-error process, such as in iterative compilation. These prediction methodologies are generally based on *machine-learning* techniques [Cooper et al. 1999; Stephenson et al. 2003; Agakov et al. 2006; Cavazos et al. 2007]. Milepost [Fursin et al. 2011] is a machine-learning based compiler that automatically adjusts its optimization heuristics to improve the execution time, code size, or compilation time of specific programs on different architectures [Fursin et al. 2008]. Authors in [Leather et al. 2009], introduced a methodology for learning static code features to be used within compiler heuristics that drive the optimization process. [Ding et al. 2015] have proposed a two-level approach on providing insights into the analysis of variable input. Models are constructed as a general means of automatically determining what algorithmic optimization to use when different optimization strategies suit different inputs .[Martins et al. 2016] proposed good sequence of optimizations in application dependent mode by clustering Design Space Exploration (DSE) technique directed by genetic algorithms.

There are a few research works that tackled the ordering of the optimizations using machine-learning-based approaches [Kulkarni and Cavazos 2012; Ashouri et al. 2016]. These are namely, the non-iterative compilation approaches to predict the *immediate speedup* corresponding to the *next-best* optimization to be applied given the current state of the code. In particular, [Kulkarni and Cavazos 2012] used Neuro-Evolution for Augmenting Topologies (NEAT) on Jikes dynamic compiler and came up with sets of good optimization ordering. They proposed *immediate speedup prediction* given the current status of the source-code and define certain stop-condition rules to complete the final predicted sequence at each iteration. Other works [Kulkarni et al. 2009] have approached the problem by exhaustively exploring the ordering space at function granularity level and evaluate their methodology with search-tree algorithms. This exhaustive enumeration allowed them to construct probabilities of enabling/disabling interactions between different optimization passes in general rather than specific to any program.

There are works using similar predictive modeling technique as proposed in [Park et al. 2013] with static program features instead of hardware-dependent features. [Park et al. 2012] used *Control Flow Graph* (CFG) with graph kernel learning to construct a machine learning model. First, they construct CFGs by using the LLVM compiler and convert the CFGs to *Shortest Path Graphs* (SPGs) by using the Floyd-Warshall algorithm. Then, they apply the shortest graph kernel method [Borgwardt and Kriegel 2005] to compare each one of the possible pairs of the SPGs and calculate a similarity score of two graphs. The calculated similarity scores for all pairs are saved into a matrix and directly fed into the selected machine-learning algorithm,

specifically SVMs in their work. In [Park et al. 2014], they use user-defined patterns as program features. They use a pattern-driven system named HERCULES [Kartsaklis et al. 2012] to derive arbitrary patterns coming from users. They focused on defining patterns related to loops, for example, the number of loops having memory accesses, having loop-carried dependencies, or certain types of data dependencies. Both works use static program features mainly focusing on loop and instruction mixes. Although our static features do not include direct loop information, we use other types of program features, such as memory footprint, memory reuse distances, and ranch predictability. Our work also differs from previous literature in forms of the machine-learning algorithm used and the target compiler framework. They use SVMs and the models are targeted to polyhedral optimization space, whereas we use statistical analysis and BN, and focus on the GCC optimization space.

Our approach is significantly different from the previous ones given that it applies a statistical methodology to learn the relationships between application features and compiler optimizations as well as between different compiler optimizations where *machine-learning* techniques are used to capture the probability distribution of different compiler transformations. In this work, we propose the use of *BN* as a framework enabling statistical inference on the probability distribution given the evidence of application features. Given a target application, its features are fed to *Bayesian Networks* to induce an application-specific bias on the probability distribution of compiler optimizations.

Most recent machine-learning works aim at the generation of prediction models that, given a target application, predict the performance of the application for any set of compiler transformations applied to it. In contrast, in our work the machine-learning methodology aims directly at predicting the best compiler optimizations to be applied for a target application without going through the predictions of the resulting application performance.

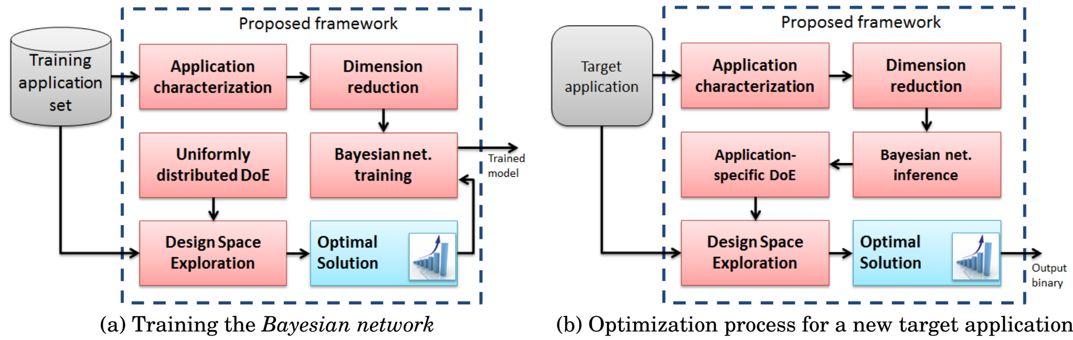
Additionally, in our approach, program features are dynamic and obtained through micro-architecture-independent characterization [Hoste and Eeckhout 2007] and compared with the results using the static profiling [Fursin et al. 2008]. The adoption of dynamic profiling provides insight into the actual program execution with the purpose of giving more weight to the code segments executed more often (i.e. code segments whose optimization would lead to higher benefits according to Amdahl's law).

3. PROPOSED METHODOLOGY

The main goal of the proposed approach is to identify the best compiler optimizations to be applied to a target application. Each application is passed through a characterization phase that generates a parametric representation of the application under analysis in terms of its main features. These features are pre-processed by means of statistical *dimension reduction* techniques to identify a more compact representation, while not loosing important information. A statistical model based on *BN* correlates these reduced representations to the compiler optimizations to maximize application performance.

The optimization flow is shown in Figure 1 and consists of two main phases. During the initial *training phase*, the *Bayesian network* is learned on the base of a set of training applications (see Figure 1a). During the *exploitation phase*, new applications are optimized by exploiting the knowledge stored in the *Bayesian Network* (see an example of a BN topology in Figure 3).

During both phases, an optimization process is necessary to identify the best compiler optimizations to achieve the best performance. This is done for learning purposes during the *training phase* and for optimization purposes during the *exploitation phase*. To implement the optimization process, a Design Space Exploration (DSE) engine has been used. The DSE engine automatically compiles, executes and measures application performance by enabling/disabling different compiler optimizations. Which compiler optimizations will be enabled is decided in the Design of Experiments (DoE) phase. In our approach, the DoE is obtained by sampling



(a) Training the *Bayesian network* (b) Optimization process for a new target application

Fig. 1: Overview of the proposed methodology.

from a given probability distribution that is either a *uniform distribution* (during the training phase as in Figure 1a) or an *application-specific distribution* inferred through the BN (during the *exploitation phase* as in Figure 1b).

The *uniform distribution* adopted during the *training phase* allows us to explore the compiler optimization space $\mathcal{O}$ uniformly to learn what the most promising regions of this space are. The *application-specific distribution* used during the *exploitation phase* allows us to speed up the optimization by focusing on the most promising region of the compiler optimization space $\mathcal{O}$.

3.1. Applying Program Characterization

The classic *supervised* Machine Learning (ML) approach deals with fitting a model exploiting a function f of program characterization. Function f might use a variety of comparison/similarity functions, such as *nearest-neighbor* and *graph-kernels*. To obtain a more accurate fitting, compiler researchers have been trying to understand the behavior of programs/kernels better and derive a *feature vector* that represents pair functionality efficiently. As a rule of thumb, the derived feature vector must be i) representative enough of its program/kernel, and ii) different programs/kernels must not have the same feature vectors as this will confuse the subsequent machine-learning process. Thus, building a huge non-efficient feature vector slows down the ML process and obtain less-precision.

Another goal of this work is to exploit the efficient use of different program characterization techniques and demonstrate their performance and effectiveness. Three characterization techniques have been selected among state-of-the-art works, namely, i) *dynamic feature selection* using *MICA* [Hoste and Eeckhout 2007], ii) *static analysis* using *MilePost* [Fursin et al. 2008] framework, and iii) our handcrafted combination of those two as hybrid analysis.

MICA. *Microarchitecture-independent workload characterization* represents a recent work on dynamic workload characterization [Hoste and Eeckhout 2007]. It is a plugin for the Linux-PIN tool [Luk et al. 2005] and is capable of characterizing the fed kernels *independently* from its running architecture as it monitors the *non-hardware* features of the kernels. This feature is of interest for targeting embedded domain as one might not be able to exploit PIN tools on the board. The main categories of MICA include *Instruction-Level-Parallelism (ILP), Instruction Mix (ITypes), Branch Predictability (PPM), Register Traffic (REG), Data Stream Stride (Stride), Instruction and Data Memory Footprint (MEMFootprint) and Memory Reuse Distances (MEMReusedist)*.

MilePost. This recent tool [Fursin et al. 2011; Fursin et al. 2008] was built as a plugin on top of *GCC* to capture static features of the programs. One advantage of *static analysis* is that the compiler researchers do not have to run the actual binary just like what they do in a

dynamic feature technique. On the other hand, *static-analysis* techniques fail to capture any correlations when different data streams are involved as input dataset.

Hybrid. The third characterization technique consists of the combination of the two previous ones. We believe that, in some cases, hybrid feature selection can capture the kernel behaviors better as it takes into account both feature-selection methods.

3.2. Dimension-Reduction Techniques

In the proposed approach, the *dimension-reduction* process is important for two main reasons: *a)* it eliminates the noise that might perturb further analyses, and *b)* it significantly reduces the training time of the BN. The techniques used are *Principal Component Analysis* (PCA) and *Exploratory Factor Analysis* (EFA). The experimental results show that the selection of a good dimension-reduction technique has a significant impact on the final model quality. In the original work proposed in [Ashouri et al. 2014], PCA was used. In this work, we changed the model by exploiting *Exploratory Factor Analysis* (EFA) as explained in the following paragraphs. Experimental results will show the benefits of using EFA with respect to PCA for the specific problem addressed herein. For a quantitative comparison the readers is referred to Section 4.4 Table V.

Let γ be a characterization vector storing all data of an application run. This vector stores l variables to account for either the static, dynamic or both analyses. Let us consider a set of known application profiles A consisting of m vectors γ. The application profiles can be organized in a matrix P with m rows and l columns. Each vector γ (i.e. a row in P) includes a large set of characteristics, such as the instruction count per instruction type (for both static and dynamic analysis), information on the memory access pattern and information characterizing the control flow (e.g. the number and length of the basic blocks, average and maximum loop nesting, etc.). Many of these application characteristics (columns of matrix P) are correlated to each other in a complex way. A simple example of this correlation is the instruction mix information collected during the static analysis and the instruction mix information collected during the dynamic profiling (even though these are not completely the same). A less intuitive example is between the distribution of basic block lengths and data related to the instruction memory reuse distance. The presence of many correlated columns in P implies that the information stored in a vector γ can be well represented with a vector α of smaller size.

Both PCA and EFA are statistical techniques aimed at identifying a way to represent γ with a shorter vector α while minimizing the information loss. Nevertheless, they rely on different concepts for organizing this reduction [Thompson 2002; Gorsuch 1988]. In both cases, output values are derived by applying the dimension reduction and are no longer directly representing a certain feature. While in PCA the components are given by a combination of the observed features, in EFA the factors are representing the hidden process behind the feature generation. In both cases, there is no way to indicate by name the output columns, since they are not directly observable.

In PCA, the goal is to identify a summary of γ. To this end, a second vector ρ of the same length of γ (i.e. l) is organized by a variable change. Specifically, the elements of ρ are obtained through a linear combination of the elements in γ. The way to combine the elements of γ for obtaining ρ is decided upon the analysis of the matrix P, and is such that all elements in ρ are orthogonal (i.e. uncorrelated) and are sorted by their variance. Thus the first elements of ρ (also named principal components) carry most of the information of γ. The reduction can be obtained by generating a vector α to keep only the first most significant principal components in ρ, because the least significant ones carry little information content. Note that principal components in ρ (thus in α) are not meant to have a meaning; they are only used to summarize the vector γ as a signature.

In EFA, the elements in the vector of reduced size are meant to explain the structure underlying the variables γ, while α, represents a vector of *latent variables* that cannot be

directly observed. The variables γ are expected to be a linear combination of the variables in α. In EFA, this relationship explains the correlation between the different variables in γ; that is, correlated variables in γ are likely to depend on the same hidden variable in α. The relationship between the latent α and the observed variables is regressed by exploiting the maximum likely method based on the data in matrix P.

When adopting PCA, each variable in α tends to be a mixture of all variables in γ. Therefore, it is rather hard to tell what a component represents. When adopting EFA instead, the components α tend to depend on a smaller set of elements in γ that are correlated with each others. That is, when applying EFA, α is a compressed representation of γ, where elements in γ that are correlated (i.e. that carry the same information) are compressed into a reduced number of elements in α. Note that reducing the profile size by means of EFA results in a α that better describes the type of application under analysis in reference to PCA [Jin and Cheng 2008].

Consequently, having obtained γ through any of the characterization techniques, a preprocessing filtering should be applied to ensure that the least noise has come through and the final P is eligible to be summarized by *EFA*. That implies manually i) removing the zero columns in l and ii) removing the redundant columns of l given that no column l is a linear combination of another l. In contrast, the algorithmic approach to tackle this is that P needs to be transformed, as to obtain the final γ in *positive-definite covariance* form [Bhatia 2009]. Different techniques have been described in the literature on how to transform a *non-positive-definite* matrix to a *positive-definite* one which exceeds the scope of this paper, but interested readers can refer to [Lee and Mathews 1994; Tanaka and Nakata 2014] or use packages in *R* statistical tool [R Gentleman 2012] i.e., *nearPD* to compute nearest positive definite matrix.

3.3. Bayesian Networks

Bayesian Networks are powerful to represent the probability distribution of different variables that characterize a certain phenomenon. The phenomenon to be investigated in this work is the optimality of compiler optimization sequences.

Let us define a Boolean vector $\boldsymbol{o}$, whose elements o_i are the different compiler optimizations. Each optimization o_i can be either enabled, $o_i = 1$, or disabled, $o_i = 0$. In this work, the *phase ordering* problem [Kulkarni et al. 2009] is not taken into account. but rather we consider how different optimizations o_i are organized in a predefined order embedded in the compiler. A compiler optimization sequence represented by the vector $\boldsymbol{o}$ belongs to the n size Boolean space $\mathcal{O} = \{0, 1\}^n$, where n represents the number of compiler optimizations under study.

An application is parametrically represented by the vector $\boldsymbol{\alpha}$ of the k reduced components computed either via PCA or via EFA from its software features. Elements α_i in vector $\boldsymbol{\alpha}$ generally belong to the continuous domain.

The optimal compiler optimization sequence $\bar{o} \in \mathcal{O}$ that maximizes the performance of an application is generally unknown. However it is known that the effects of a compiler optimization o_i might depend on whether another optimization o_j has been applied. Additionally, it is known that the compiler optimization sequence that maximizes the performance of a given application depends on the application itself.

The reason why the optimal compiler optimization sequence $\bar{o}$ is unknown a priory is because it is not possible to capture, in a deterministic way, the dependencies among the variables in the vectors $\bar{o}$ and $\boldsymbol{\alpha}$. There is no way to identify an analytic model to exactly fit the vector function $\bar{o}(\boldsymbol{\alpha})$. As a matter of fact, the best optimization sequence $\bar{o}$ depends also on other factors that are somewhat outside our comprehension, *the unknown*. It is exactly to deal with *the unknown* that we propose not to predict the best optimization sequence $\bar{o}$ but rather to infer its probability distribution. The uncertainty stored in the probability distribution models the effects of *the unknown*.

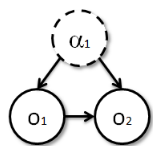


Fig. 2: A *Bayesian Network* example.

As underlying probabilistic model, we selected *BN* because of the following features of interest for the target problem:

— Their expressiveness allows one to include heterogeneous variables in the same framework such as Boolean variables (in the optimization vector $\boldsymbol{o}$) and continuous variables (in the application characterization $\boldsymbol{\alpha}$).
— Their capabilities to model *cause-effect* dependencies. Representing these dependencies is suitable for the target problem, as we expect that the benefits of some compiler optimizations (effects) are due to the presence of some application features (causes).
— It is possible to graphically investigate the model to visualize the dependencies among different compiler optimizations. If needed, it is even possible to manually edit the graph for including some *a priori* knowledge.
— It is possible to bias the probability distribution of some variables (the optimization vector $\boldsymbol{o}$) given the *evidence* on other variables (the application characterization $\boldsymbol{\alpha}$). This enables one to infer an *application-specific distribution* for the vector $\boldsymbol{o}$ from the vector $\boldsymbol{\alpha}$ observed by analyzing the target application.

A *Bayesian Network* is a direct acyclic graph whose nodes represent variables and whose edges represent the dependencies between these variables. Figure 2 reports a simple example with one variable α_1 representing the application features and two variables o_1, o_2 representing different compiler optimizations. In this example, the probability distributions of the two optimizations depend on the program features represented by α. Additionally, the probability distribution of o_2 depends on whether the optimization o_1 is applied. Dashed lines are used for nodes representing observed variables whose value can be input as evidence to the network. In this example, the variable α_1 can be observed and, by introducing its evidence, it is possible to bias the probability distributions of other variables.

Training the Bayesian model. Tools exist to construct *BN* automatically by fitting the distribution of some training data [Murphy 2001]. To do so, first the graph topology is identified and then the probability distribution of the variables including their dependencies is estimated.

The identification of the graph topology is particularly complex and time consuming. The *dimension reduction* technique applied on the SW features plays a key role in obtaining reasonable training times by limiting to k elements in the vector $\boldsymbol{\alpha}$, thus reducing the number of nodes in the graph.

For efficiency reasons, the algorithm used for selecting the graph topology is an heuristic algorithm, named *K2*, initialized with the *Maximum Weight Spanning Tree* (MWST) ordering method as suggested in the Matlab toolbox in use [Murphy 2001]. The initial ordering of the nodes for the MWST algorithm is given to let the elements $\boldsymbol{\alpha}$ to appear first and then the elements of $\boldsymbol{o}$. Even if the final topological sorting of the nodes changes according to the algorithm described in [Heckerman and Chickering 1995], by using this initialization criterion, it always happens that the dependencies are directed from elements of $\boldsymbol{\alpha}$ to elements of $\boldsymbol{o}$ and not vice versa. When using the K2 algorithm, the network topology is selected as follows. The graph is initialized with no edges to represent the fact that each variable is independent.

Then, for each variable i, following their initial ordering, each possible edge from j to i (where $j < i$) is considered as candidate to be added to the network. A candidate edge is added to the topology if it increases the probability that the training data were generated from the probability distribution the new topology describes. This method has a polynomial complexity with respect to the number of variables involved and the number of lines in the training data set.

During the model training, we consider the *softmax* function for modeling the cumulative probability distribution of the Boolean elements in vector $\boldsymbol{o}$ [Murphy 2001]. This is a mathematical necessity to map in the *Bayesian framework* the dependencies of Boolean variables in $\boldsymbol{o}$ with respect to continuous variables in $\boldsymbol{\alpha}$. In particular, thanks to the use of *softmax* variables, we can express the conditional probability $P(o_i = b \mid \alpha_j = x)$, where o_i is a Boolean variable and α_j is a continuous variable.

The coefficients of the functions describing the probability distribution of each variable as well as their dependencies are tuned automatically to fit the distribution in the training data [Murphy 2001]. Training data are gathered by analyzing a set A of training applications (Figure 1a). First, application features are computed for each application $a \in A$ to enable the principal component analysis. Thus, each application is characterized by its own principal component vector $\boldsymbol{\alpha}$. Then, an experimental compilation campaign is carried out for each application by sampling several compiler optimization sequences from the compiler optimization space $\mathcal{O}$ with a uniform distribution. For each application, we select the 15% best-performing compilation sequences among the sampled ones. The distribution of these sequences is learned by the *Bayesian Network* framework in relation to vector $\boldsymbol{\alpha}$ characterizing the application.

Inferring an application-specific distribution. Once the *Bayesian Network* has been trained, the principal component vector $\boldsymbol{\alpha}$ obtained for a new application can be fed as evidence to the framework to bias the distribution of the compiler optimization vector $\boldsymbol{o}$. To sample a compiler optimization sequence from this biased distribution, we proceed as follows. The nodes in the direct acyclic graph describing the *Bayesian Network* are sorted in topological order, i.e. if a node at position i has some predecessors, those appear at positions $j, j < i$. At this point, all nodes representing the variables $\boldsymbol{\alpha}$ appear at the first positions[2]. The value of each compiler optimization o_i is sampled in sequence by following the topological order such that all its parent nodes have been decided. Thus, the marginal probability $P(o_i = 0 \mid \mathcal{P})$ and $P(o_i = 1 \mid \mathcal{P})$ can be computed on the basis of the parent node vector value $\mathcal{P}$ (each parent being either an evidence α_j or a previously sampled compiler optimization o_j). Similarly, by using the maximum likelihood method, it is possible to compute the most probable vector from this biased probability distribution. When sampling from the application-specific probability distribution inferred through the *Bayesian Network*, we always consider to return the most probable optimization sequence as first sample.

4. EXPERIMENTAL EVALUATION

The goal of this section is to assess the benefits of the proposed methodology. In this work, we run the experimental campaign on an ARMv7 Cortex-A9 architecture as part of a TI-OMAP 4430 processor [Instruments 2012] with *ArchLinux* and *GCC-ARM 4.6.3*.

4.1. Benchmark Suites

To assess the proposed methodology, we have used two major benchmark suites separately: i) *cBench* [Fursin 2010] and ii) *PolyBench* [Grauer-Gray et al. 2012; Pouchet 2012]. Each consists of different classes of applications and kernels ranging from security and cryptography

[2]This is by construction due to the initialization of the MWST and the K2 algorithms used to discover the network topology.

algorithms to office and image-processing applications. Readers can refer to Table I for the list of applications selected in the two benchmark suites.

Table I: Benchmark suites used in this work

(a) *cBench* applications selected for this work

cBench list	**Description**
automotive_bitcount	Bit counter
automotive_qsort1	Quick sort
automotive_susan_c	Smallest Univalue Segment Assimilating Nucleus Corner
automotive_susan_e	Smallest Univalue Segment Assimilating Nucleus Edge
automotive_susan_s	Smallest Univalue Segment Assimilating Nucleus S
security_blowfish_d	Symmetric-key block cipher Decoder
security_blowfish_e	Symmetric-key block cipher Encoder
security_rijndael_d	AES algorithm Rijndael Decoder
security_rijndael_e	AES algorithm Rijndael Encoder
security_sha	NIST Secure Hash Algorithm
telecom_adpcm_c	Intel/dvi adpcm coder/decoder Coder
telecom_adpcm_d	Intel/dvi adpcm coder/decoder Decoder
telecom_CRC32	32 BIT ANSI X3.66 crc checksum files
consumer_jpeg_c	JPEG kernel
consumer_jpeg_d	JPEG kernel
consumer_tiff2bw	convert a color TIFF image to grey scale
consumer_tiff2rgba	convert a TIFF image to RGBA color space
consumer_tiffdither	convert a TIFF image to dither noisespace
consumer_tiffmedian	convert a color TIFF image to create a TIFF palette file
network_dijkstra	Dijkstra's algorithm
network_patricia	Patricia Trie data structure
office_stringsearch1	Boyer-Moore-Horspool pattern match
bzip2d	BurrowsWheeler compression algorithm
bzip2e	BurrowsWheeler compression algorithm

(b) Linear-algebra/applications of the *PolyBench* suite selected for this work

PolyBench list	**Description**
2mm	2 Matrix Multiplications (D=AB; E=CD)
3mm	3 Matrix Multiplications (E=AB; F=CD; G=EF)
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
cholesky	Cholesky Decomposition
doitgen	Correlation Computation
gemm	Matrix-multiply C = AB + C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
mvt	Matrix Vector Product and Transpose
symm	Symmetric matrix-multiply
syr2k	Symmetric rank2k operations
syrk	Symmetric rankk operations
trisolv	Triangular solver
trmm	Triangular matrix-multiply

4.1.1. cBench. The *cBench* suite [Fursin 2010] is a collection of open-source programs with multiple data sets assembled by the community to enable realistic workload execution and targeted by many different compilers such as *GCC, LLVM*, etc.. The source code of individual programs is simplified to facilitate portability; therefore, it has been targeted in *autotuning* and *iterative compilation* research work. Of the available data sets for every individual kernel, we have selected five and sorted them in a way that dataset1 is always the smallest and dataset5 the largest. This ensures that for every kernel we have exposed enough of the input load to be able to measure fair runtime executions.

Table II: Compiler optimizations under analysis (beyond -O3)

Compiler Transformation	Abbreviation	Short Description
-funsafe-math-optimizations	*math-opt*	Allow optimizations for floating-point arithmetic that (a) assume valid arguments and results and (b) may violate IEEE or ANSI standards.
-fno-guess-branch-probability	*fn-gss-br*	Do not guess branch probabilities using heuristics.
-fno-ivopts	*fn-ivopt*	Disable induction variable optimizations on trees.
-fno-tree-loop-optimize	*fn-tree-br*	Disable loop optimizations on trees
-fno-inline-functions	*fn-inline*	Disable optimization that inline all simple functions.
-funroll-all-loops	*funroll-lo*	Unroll all loops, even if their number of iterations is uncertain
-O2	*O2*	Overwrite the -O3 optimization level by disabling some optimizations involving a space-speed trade-off

4.1.2. PolyBench. The *PolyBench* benchmark suite [Pouchet 2012; Grauer-Gray et al. 2012] consists of benchmarks with static control parts. The purpose is to make the execution and monitoring of applications uniform. One of the main features of the *PolyBench* suite is that there is a single file per application, tunable at compile-time and used for kernel instrumentation. It performs extra operations such as cache flushing before the execution, and can set real-time scheduling to prevent OS interference. We have defined two different data sets for each individual application to expose the main function with different input loads. PolyBench has a variety of benchmarks, i.e. 2D and 3D matrix multiplication, vector decomposition, etc.. This suite is also suitable for parallel programming, which is beyond the focus of this work.

4.2. Compiler Transformations

The compiler transformations analyzed have been reported in Table II. We based our design space on the work of [Chen et al. 2012]. The authors implemented sensitivity analysis over a vast majority of the compiler optimizations and defined with a list of promising passes. Building upon their work, we selected the compiler optimizations with a speedup factor greater than 1.10. They are applied to improve application performance beyond the standard optimization level -O3 and have not yet been included in any prior optimization level. The optimizations can be enabled/disabled by means of the respective compiler optimization flags. The standard optimization level -O3 has been also used to collect the dynamic software-features for each application on both *training* and *inference* phases.

The application execution time has been estimated by using the Linux-perf tool. The execution time is done by averaging five loop-wraps of the specific compiled binary with one second of sleep in between five different executions of those loop-wraps. Therefore, in total, each individual transformed binary has been executed 25 times as five packages of five loop-wraps to ensure better accuracy of estimations and fairness among the generation of executions. This technique is used both in the *training* and the *inference* phases.

4.3. Bayesian Network Results

In this work, Matlab environment [Murphy 2001; Santana et al. 2010] has been used to train the *Bayesian Network*. We have used *Exploratory Factor Analysis* (EFA) of application features for the seven compiler optimization flags listed in Table II. As stated in Section 3.2, one of the features of using EFA is that the factors are linear combinations that maximize the shared portion of the variance. Therefore, as prerequisite, the covariance matrix should be

Table III: Kaiser test results

Application	**Characterization Method**	**Original No. of Factors**	**Range of Selected Factors By Kaiser Test**
cBench	MICA (Dynamic)	99	[7-11]
cBench	MILEPOST (Static)	53	[4-6]
cBench	Hybrid	143	[8-10]
polyBench	MICA (Dynamic)	99	[5-7]
polyBench	MILEPOST (Static)	53	[4-6]
polyBench	Hybrid	143	[4-5]

positive definite. This pre-processing helps purify the highly correlated application characterization columns that are linearly correlated. In theory, *PCA* accepts any matrix ignoring the aforementioned condition and that is why we think applying factor analysis as our dimension reduction technique tends to obtain the most important factors and correlate them with the compiler optimizations. Decision on the numbers of factors have been derived from the *Kaiser* test [Kaiser 1958]. This test implies taking only the factors having greater than 1 in the covariance matrix. In other words, the Kaiser rule is to drop all components with eigenvalues under 1, this being the eigenvalue equal to the information accounted for by an average single item. Table III reports the factors derived for each individual benchmark and characterization method.

Table III represents the number of features that have been produced both originally by the different feature selection techniques and by the Kaiser test. The third column is the *original number of features* and the last one refers to *range of selected factors* in each specific benchmark suite/feature selection method. Note that the last column reports the range of selected factors rather than a number as we have used cross-validation approach in the experimental campaign, thus different applications/datasets/feature selection techniques can result in a different number of factors to be used in COBAYN's framework.

In this work, while training has been carried out using each application/dataset pair separately, the validation has been done through an application-level cross-validation (*Leave-One-Out* cross-validation, LOO). We train different BNs, each by excluding an applications (together with all its input dataset) from the training set.

Using *BN* enables us to investigate graphically the dependencies between the variables involved in the compiler optimization problem and to correlate them with the selected factors of the program characterization. We train a final *Bayesian Network* including all applications in the training set. The resulting network topology is a directed acyclic graph *DAG*, as shown in Figure 3. By removing *security_rijndael_e* application from the training set, the graph topology slightly changes, mainly in terms of the different edges connecting the *Principal Components (PC)/program factors (FA)* nodes to the compiler optimization nodes. This is due to the change in the program features and its factors, which are computed in a different way. For the sake of conciseness, we do not report all graph topologies derived by the LOO technique for each individual trained *Bayesian Network*.

The Nodes of the topology graph reported in Figure 3 are organized in layers. The first layer reports the FAs that are the observable variables (reported as dashed lines). The second layer contains the compiler optimizations whose parents are the PC nodes (or FA nodes depending weather PCA or EFA is used). Therefore the effects of these compiler optimizations depend only on the application characterization in terms of its features. In the third layer, the compiler optimization nodes whose parents include optimization nodes from the second layer are listed. Once a new application is characterized for a target application data set, the evidence related to the PCs (or FAs) of its features is fed to the network in the first layer. Then, the probability

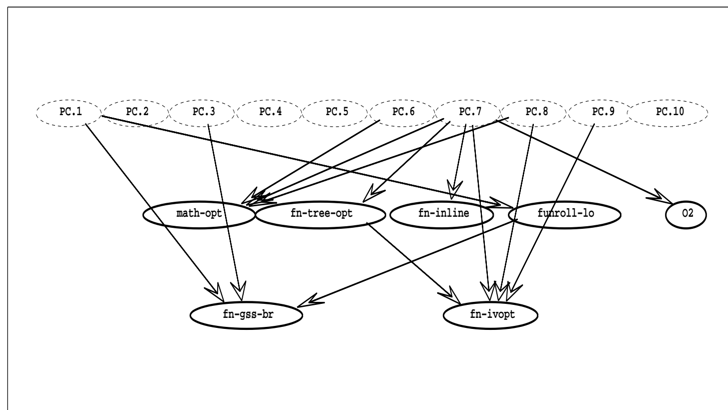


Fig. 3: Topology of the *Bayesian Network* if *security_rijndael_e* is left out of the training set

distributions of other nodes can be inferred in turn on the second and third layers. There are two nodes in the third layer of Figure 3. The first one is the *fn-gss-br* node that depends on *funroll-lo* because unrolling loops impacts the predictability of the branches implementing these loops. Moreover, *funroll-lo* impacts the effectiveness of the heuristic branch probability estimation, thus *fn-gss-br*. The second node in the third layer is the *fn-ivopt* node, which depends on *fn-tree-opt* as parent node in the second layer. Both these optimizations work on trees and therefore their effects are interdependent. While sampling compiler optimizations from the *Bayesian Network*, the decisions of whether to apply *fn-gss-br* and *fn-ivopt* are taken after deciding whether to apply *funroll-lo* and *fn-tree-opt*.

Table IV shows the fine-grain breakdown of the timing when we use COBAYN framework. We have reported the time spent for each phase of the proposed technique, both on the training phase (done offline) and on the inference phase (done online). Constructing COBAYNs network is a one-time process and depends on the number of applications in the training set. The time needed to collect the training data is on the other side, dependents not only on the number but also on the applications and data-set used for the training. The same happens for the time needed for compiling and executing the target application during the online compiler autotuning phase. To that end, Table IV reports the numbers for each specific phase considering the cBench as training set and Susan as target application. During the offline training-phase, the time needed for data collection on the case of cBench, is around 2 days. It includes the time needed for each benchmark to compile and execute, considering all set of configuration and the feature collection phase. The time needed to post-process the data and to generate the Bayesian Network model is around 70 seconds.

During the online phase (inference phase), the time needed for extracting the software features from the target application is 14.4 seconds while, querying BN is less than 1 second. The compilation and execution-time on the target platform for Susan are 4.5 and 8.9 seconds, respectively. Those numbers show that the initial overhead in adopting the proposed methodology on the user-side (composed of the software feature extraction and BN inference) is less than 2 compilation/executions pairs, for this specific example.

4.4. Comparison Results

It is well known that *Random Iterative Compilation* (RIC) can improve application performance compared with static handcrafted compiler optimization sequences [Agakov et al. 2006]. Additionally, given the complexity of the *iterative compilation* problem, it has been proved that drawing compiler optimization sequences at random is as good as applying other

Table IV: COBAYN timing breakdown for offline training and online inference for *Susan* application

Phase	Tag & Category	Time
Offline training	(A) Offline data-collection	2 days
	(B) Construct BN	70 sec
Online inference	(C) SW Feature Collection	14.4 sec
	(D) BN Inference	0.85 sec
	(E) Susan compilation	4.5 sec
	(F) Susan execution	8.9 sec

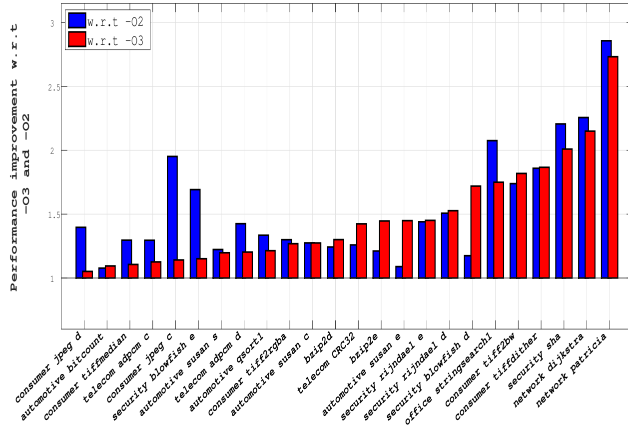
Table V: COBAYN (BN using EFA) speedup w.r.t standard optimization levels (-O2 and -O3) and *Random Iterative Compilation* (RIC) and our previous approach of BN in [Ashouri et al. 2014] using PCA

Benchmarks	**Features**	**COBAYN Speedup** w.r.t			
		-O2	**-O3**	**RIC**	**BN w/ PCA**
cBench	Dynamic	1.6093	1.528	1.2029	1.0744
cBench	Static	1.5447	1.478	1.1143	1.0543
cBench	Hybrid	1.5858	1.5066	1.2086	1.0654
cBench Average		1.5795	1.5035	1.1743	1.0617
PolyBench	Dynamic	1.9845	1.8387	1.3230	1.0921
PolyBench	Static	1.9353	1.8215	1.1518	1.0724
PolyBench	Hybrid	1.9441	1.7726	1.2333	1.1078
PolyBench Average		1.9541	1.8101	1.2350	1.0901
Overall (Harmonic Mean)		**1.7669**	**1.6571**	**1.2052**	**1.0771**

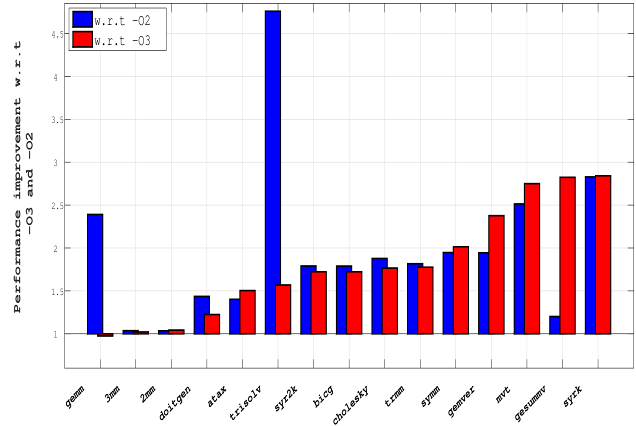
optimization algorithms such as genetic algorithms or simulated annealing [Agakov et al. 2006; Cavazos et al. 2007; Chen et al. 2012]. Accordingly, to evaluate the proposed approach, we compared our results with **i)** standard optimization levels -O2 and -O3 **ii)** the Random Iterative Compilation (RIC) methodology that samples compiler optimization sequences from the uniform distribution and **iii)** two advanced state-of-the-art methodologies, namely *a)* (Section 4.6.1) coupling machine learning with an iterative methodology, and *b)* (Section 4.6.2) a non-iterative methodology derived by *predictive modeling* based on different machine-learning algorithms to predict the final application speedup.

The proposed methodology samples different compiler optimization sequences from the BN. The performance achieved by the best application binary depends on the number of sequences sampled from the model. In this section, the result of applying the proposed methodology using two benchmark suites with respect to standard optimization levels and the *random iterative compilation* have been reported. The performance speedup on the first comparison section is measured in reference to -O2 and -O3, which are the optimization levels available for GCC. In addition, we show the speedup of the proposed methodology with respect to our previous work [Ashouri et al. 2014].

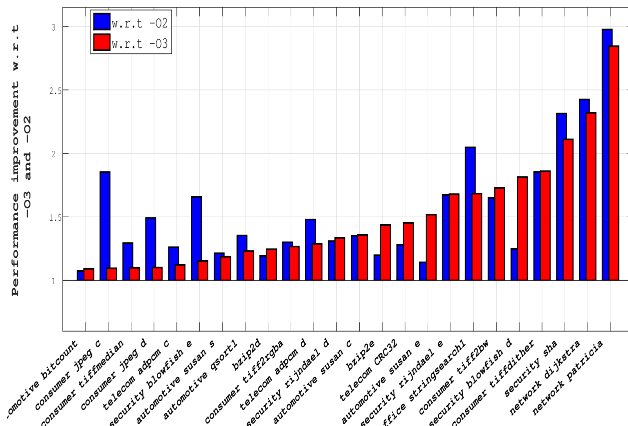
4.4.1. Bayesian Networks Performance Evaluation. Table V reports COBAYN's speedup achieved over the standard optimization levels of *-O2* and *-O3* and *Random Iterative Compilation (RIC)*. The last column represents the average speedup achieved by revising our *Bayesian Network* engine and using the *Explanatory Factor Analysis (EFA)* described in the Section 3.2 with respect to PCA in [Ashouri et al. 2014]. Note that all speedup values have been averaged using Harmonic mean. It is observed that in all categories, COBAYN outperforms standard optimization levels and the previous approach. The comparison with respect to the RIC has been



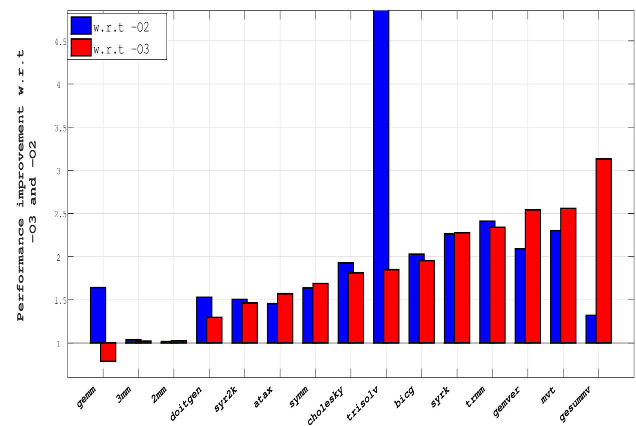
(a) BN with dynamic features on cBench



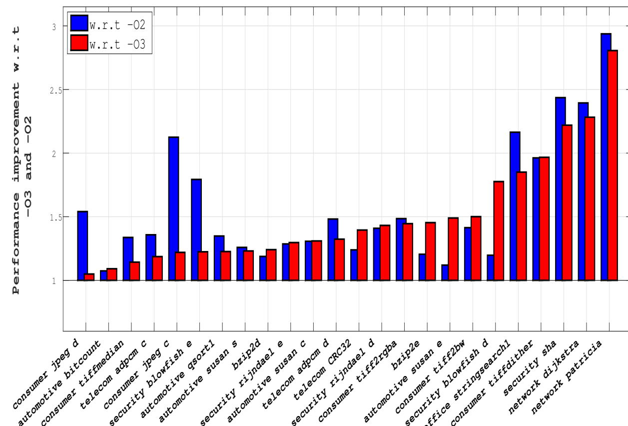
(b) BN with dynamic features on PolyBench



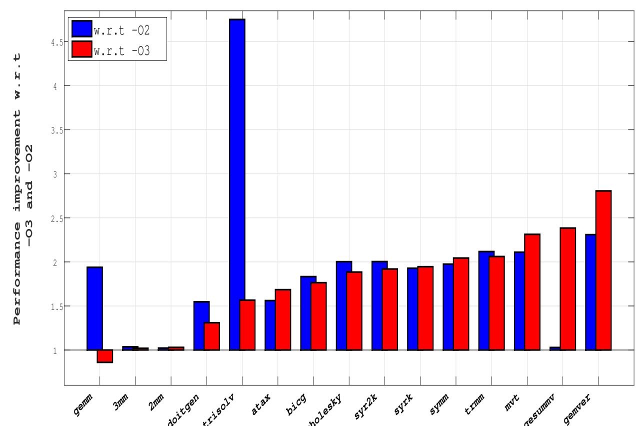
(c) BN with static features on cBench



(d) BN with static features on PolyBench



(e) BN with dynamic+static features on cBench



(f) BN with dynamic+static features on PolyBench

Fig. 4: Performance speedup w.r.t -O2 and -O3

Table VI: Evaluation of different BigSet formation in COBAYN Model Construction. Note that COBAYN's default refers to the version of COBAYN trained on a single benchmark set.

BigSet Combination		**Speedup w.r.t. COBAYN's Default**
cBench	**PolyBench**	
24 (All)	15 (All)	1.1143
15	15	1.0743
10	10	1.0432
5	5	0.9896

reported by the Harmonic average over the speedup data derived by dividing the COBAYN's performance data by the RIC data in full space. It can be seen that dynamic feature selection brings best results followed by the hybrid and static method. However, in certain cases (cBench using hybrid SW features), it narrowly reaches the performance of dynamic feature selection.

Using two major benchmarks and three different application characterization techniques, we report six different plots showcasing the benefits of the proposed methodology with respect to the GCC standard optimization levels. Figure 4, reports the speedups by considering a sample of eight different compiler optimization sequences. For each benchmark, the results have been averaged on the different data sets. All results have been sorted by the speedup values of -O3 and have been matched with their corresponding -O2 value. The bar plot is colored in blue and red, respectively, for the speedup achieved with respect to -O2 and -O3. All applications have achieved a speedup in reference to the performance of -O2 and -O3. This happens with the exception of *gemm* in reference to -O3 for static and hybrid feature-selection techniques and *consumer-jpeg-d* in reference to -O3 when using the dynamic method for feature selection. These applications reach their best performance using -O3 for two data sets out of five, and it was not possible to surpass this maximum by relying on the compiler transformations under consideration. On average for *cBench*, the speedups are of **1.57** and **1.5** in reference to -O2 and -O3, respectively, and **1.95** and **1.81** for *PolyBench*. The maximum speedup observed is **3.1**$\times$ and **4.7**$\times$. Table V reports the speedup gained using COBAYN compared with the standard optimization levels, *Random Iterative Compilation (RIC)* and our previous approach exploiting *PCA* as dimension-reduction method.

Analysis of the Portability of COBAYN in Different Scenarios. The results reported in this section are computed by means of LOO cross-validation on the two individual benchmark suites separately, one with 24 and the other with 15 applications. As the nature of these two benchmark suites is totally different, we believed it would be unfair to train on one and test on the other, so we analyzed the feasibility of mixing these applications in a fair heterogeneous set of *BigSet* so that COBAYN's engine gets evaluated. To this end, we tried 4 different scenarios, where the BigSet is obtained by: (i) including all 39 available applications, (ii) 15 applications of cBench and 15 applications of PolyBench, (iii) selecting 10 cBench and 10 PolyBench and finally (iv) 5 applications from each of those. Therefore, the BigSet was initialized with 39, 30, 20 and 10 different applications, and LOO cross-validation was carried-out. Table VI reports the speedup gained in these scenarios. It is observed that COBAYN framework benefits from having *a)* more applications, and *b)* heterogeneous applications in the training set. The speedup listed in Table VI is higher when BigSet accounts for more applications and, even just 10 applications per benchmark suite, it is higher than one (the default setting for the experimental results in this work refers to the COBAYN trained only on one of the two benchmark suites).

4.4.2. Performance Improvement. Let us define the *Normalized Performance Improvement* (NPI) as the ratio of the performance improvement achieved over the potential performance improvement:

$$NPI = \frac{E_{ref} - E}{E_{ref} - E_{best}} \quad (1)$$

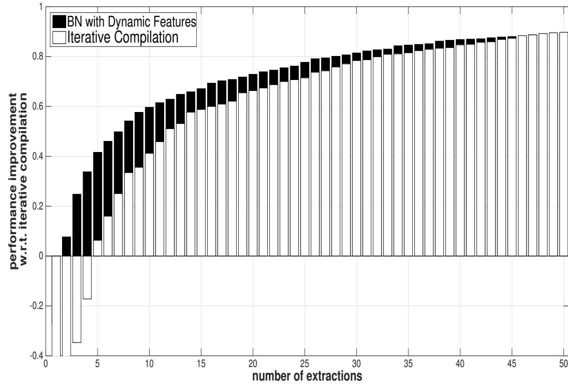
where E is the execution time achieved by the methodology under consideration, E_{ref} is the execution time achieved with a reference compilation methodology and E_{best} is the best execution time computed through an exhaustive exploration of all possible compiler optimization sequences (in our case 128 different sequences). As the execution time E of the iterative compilation methodology under analysis gets closer to the reference execution time E_{ref}, the NPI gets closer to 0, reporting that no improvement is returned. In the same way, as E gets closer to the best execution time E_{best}, while NPI gets close to 1, reporting that the entire potential performance improvement has been achieved.

Figure 5 reports for six different benchmark/feature selection methods; the NPI achieved by the *proposed optimization* technique and by the RIC technique in reference to the execution time obtained by -O3 (E_{ref}). It is noticeable that NPI has the upper-hand on performance on every number of extractions with respect to RIC. For readability purposes, we have only reported the first 50 extraction of the design space. The trend is continuously applied to the rest of the extractions until both get the maximum performance value of 1 at extraction no. 128, which accounts for the optimal compiler sequence given the specific application (also it is the optimal performance using exhaustive search). The comparisons reported in Figure 5 were carried out by considering the same number of compiler optimization sequences sampled for both the RIC and the *proposed* approach. We acknowledge the fact that there is still room for improvement in future work. However, NPI figures show that in all cases, the proposed method was superior in terms of performance and that 30 extractions, on the current scale, reach 80% of the optimality.

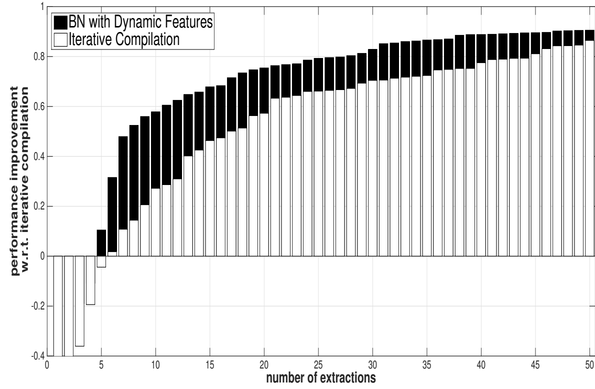
4.5. A Practical Usage Assessment

When using *iterative compilation* in realistic cases, we need to decide how much effort should be spent on the optimization itself. This effort can be measured in terms of optimization time, which is directly proportional to the number of compilations to be executed. Thus, in this section, we evaluate the proposed optimization approach in terms of the application performance reached after a fixed number of compilations. In particular, we fix this number to eight which represents 6.25% of the overall optimization space. Our model has been compared with RIC and in Figure 6, we report the *violin* plot for application speedup, while keeping the compilation effort of the proposed methodology to eight compilations (or extractions) and varying the compilation efforts of the RIC to explore more compiler space in the long run. Each individual distribution in Figure 6 represents the performance of the proposed work with respect to *RIC* across different extractions. The red cross marks the *mean* and the green square marks the *median* of each violin distribution. It can be seen that the proposed methodology with BN inference achieved an at least $3\times$ reduction in exploration process effort compared with the same extraction of RIC. Here we define *exploration speedup* as the factor measuring the aforementioned metrics, enabling the researchers to traverse the compiler design space more efficiently.

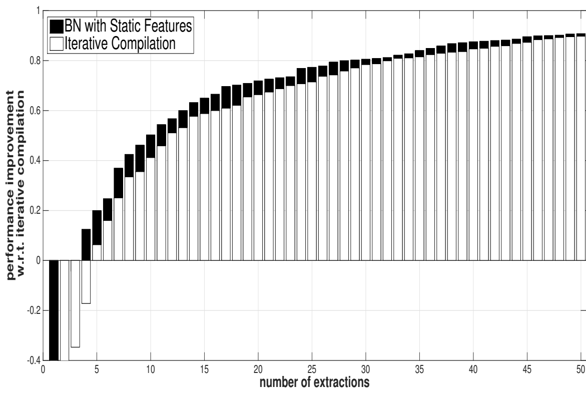
Accordingly, by increasing the compilation efforts on *RIC*, while keeping the exploration efforts of the proposed approach constant, the application speedup of COBAYN decreases. On average, RIC needs 24-32 extractions to achieve the application performance obtained with eight extractions by COBAYN. This means that COBAYN provides a speedup of 3-4$\times$ in terms of optimization efforts, that is only slightly impacted by the initial overhead (less than 2 evaluations) reported in Section 4.3. Furthermore, at the most extreme case, when RIC exhaustively enumerates and explores the full-space, 8 extractions of COBAYN, on average, still could gain up to 91% of the optimal solution. This is shown on the final distribution of each *violin* plot separated by a vertical dashed-line.



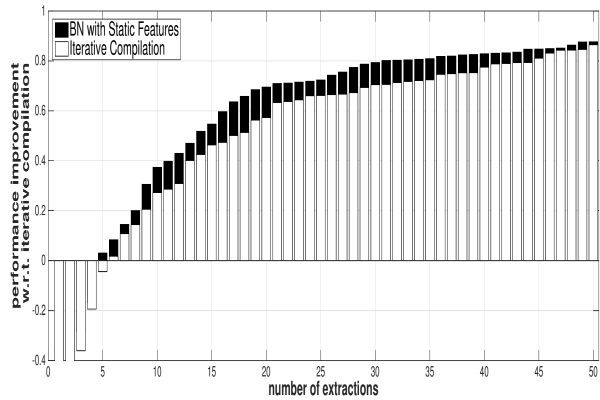
(a) BN with dynamic features on cBench



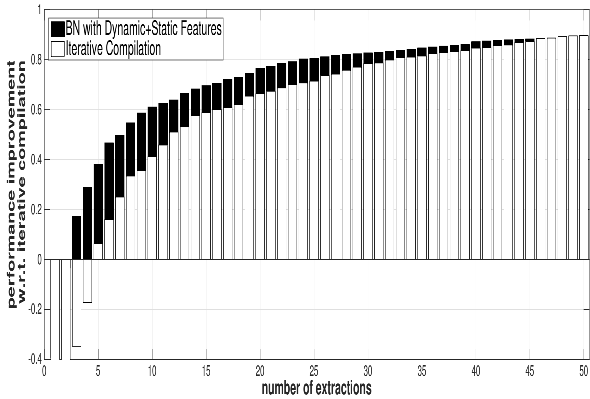
(b) BN with dynamic features on PolyBench



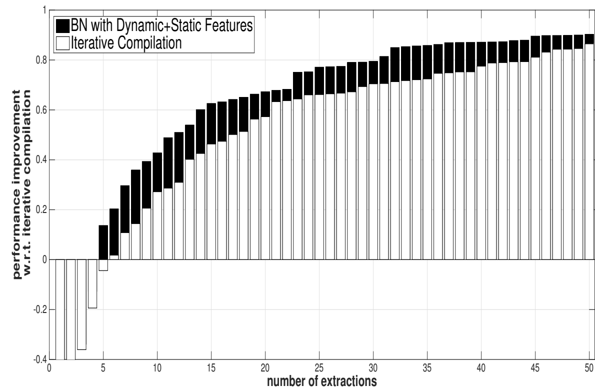
(c) BN with static features on cBench



(d) BN with static features on PolyBench

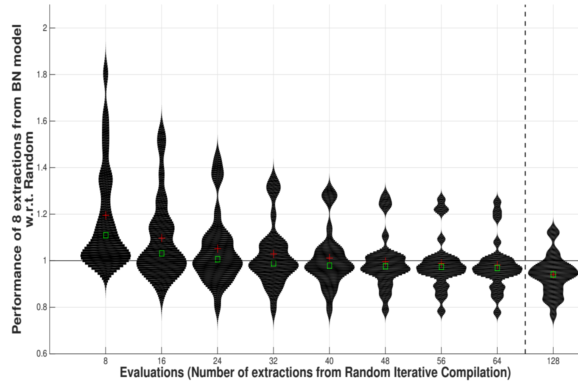


(e) BN with hybrid features on cBench

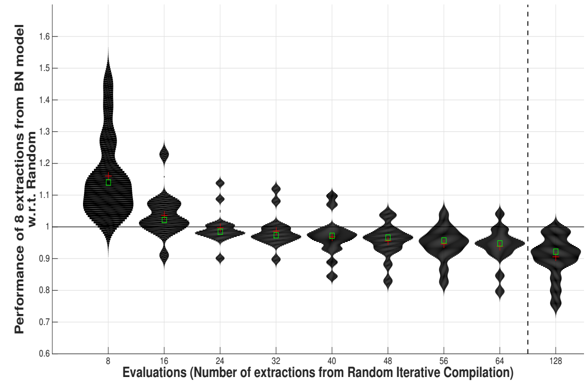


(f) BN with hybrid features on PolyBench

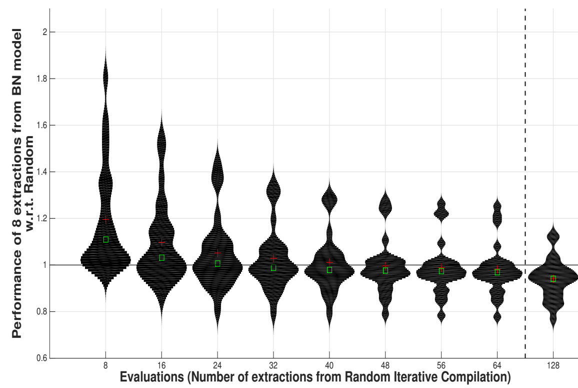
Fig. 5: Normalized performance improvement (NPI) w.r.t. RIC model



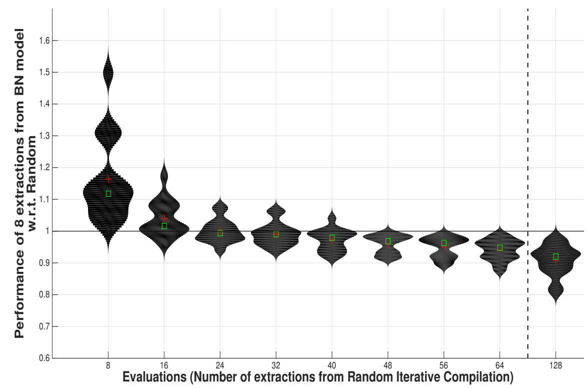
(a) BN with dynamic features on cBench



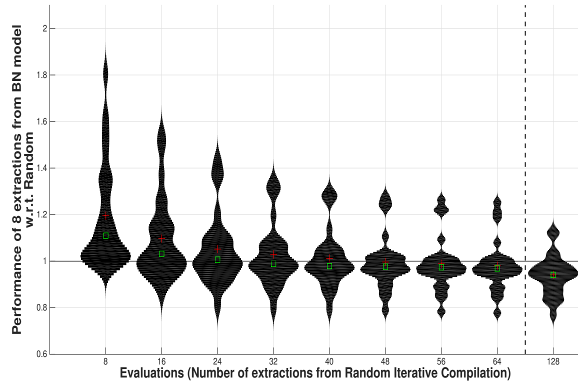
(b) BN with dynamic features on PolyBench



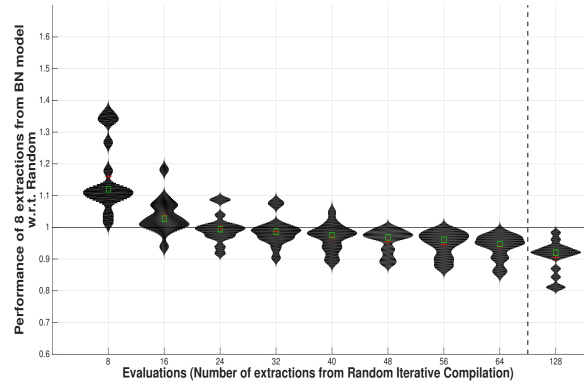
(c) BN with static features on cBench



(d) BN with static features on PolyBench



(e) BN with dynamic+static features on cBench



(f) BN with dynamic+static features on PolyBench

Fig. 6: Exploration speedup of 8 extractions w.r.t different evaluations of RIC

4.6. Comparison with State-of-the-Art Techniques

In this section, we compare the quality of the COBAYN results with respect to approaches that derived from (A) an *iterative compilation* and (B) a *non-iterative compilation* methodology.

4.6.1. Comparison to a Iterative Compilation Methodology. [Agakov et al. 2006] introduced machine-learning models to focus on the exploration of the compiler optimization for the most promising region. Their methodology exploits a Markov chain oracle and an independent identically distributed (IID) probability distribution oracle. These two offline-learnt models bias certain optimizations over others and replace the uniform probability distribution we applied earlier for the RIC reference methodology. [Agakov et al. 2006] reports significant speedup by coupling these machine-learning models with a nearest-neighbor-classifier. When predicting the probability distribution of the best compiler optimizations for a new application, the classifier first selects the training application having the smallest Euclidean distance in the feature vector space (derived by PCA). Then it learns the probability distribution of the best compiler optimizations for this neighboring application either by means of the Markov chain model or by using an IID model. These probability distribution learnt is then used as the predicted optimal distribution for the new application. It has been reported that the Markov chain oracle outperforms the IID oracle, followed by the RIC methodology using a uniform probability distribution.

We constructs the $P(S)$ probability matrix reported in Section 4.2 and 4.3 of [Agakov et al. 2006] as:

$$P(S_{IID}) = s_1, s_1, ..., s_L = \prod_{i=1}^{L} P(s_i) \tag{2}$$

$$P(S_{Markov}) = P(s_1) \prod_{i=2}^{L} P(s_i | s_{i-1}) \tag{3}$$

where $P(S_{IID})$ and $P(S_{Markov})$ define the probability of the specific sequence with IID and Markovian property for the optimization $t_1, t_1, ..., t_L$. Using LOO cross-validation, we find the closest neighbor for each cBench application trained by the two oracles, and we sample from their probability distributions. To comply with the original work in [Agakov et al. 2006], we consider only the five most relevant principal components PCs and account only for the *static program features* (also when applying COBAYN). The results are depicted in Figure 7. It shows that COBAYN is faster in reaching higher speedup values. The results are scaled and normalized with respect to -O3 by using the NPI value (Equation 1). COBAYN is able to capture a more realistic probability matrix of the compiler optimization problem and achieves with faster convergence towards the optimal result. It brings 1.25× and 1.47× speedup with respect to IID and the Markov oracle, respectively.

4.6.2. Comparison to a Non-iterative Compilation Methodology. [Park et al. 2013], used a polyhedral compiler framework capable of predicting the speedup for an unseen application. They used certain loop-optimizations in their *design-space* and surfed the *full-search* of the space. They reported the average speedup gained with respect to standard optimization O3 by using different machine-learning models on *WEKA* [Hall et al. 2009] machine-learning environment.

i) Their predictive models are based on performance counters that are collected from the underlying architecture while running the applications. Therefore, the program features to be exposed to the model are architecture dependent, and the model loses its portability when it is used for a different architecture.

ii) We also explore a different compiler optimization space. They have explored polyhedral optimization space including loop transformations, whereas we focus on GCC optimization

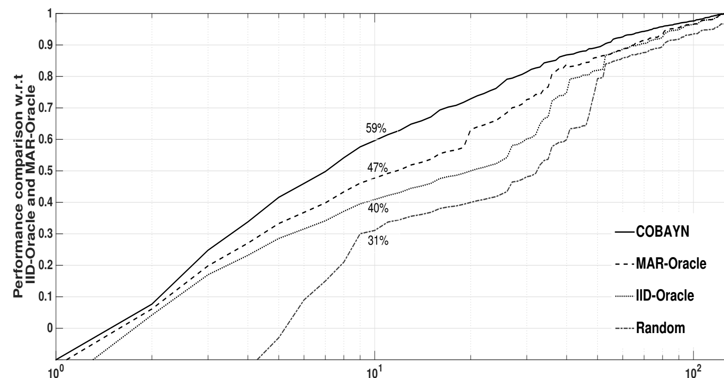


Fig. 7: NRI-scales speedup comparison of COBAYN with IID-oracle and MAR-oracle reported in [Agakov et al. 2006]

space including loop transformations and other optimizations such as *inlining, math optimizations*, etc..

iii) Furthermore, our model is based on a statistical analysis and BN, whereas they use a different set of machine-learning techniques, namely, predictive models. [Park et al. 2013].

Nonetheless, we compare their methodology by applying it to the problem at hand. Table VII reports the results obtained by using the machine-learning models in [Park et al. 2013] on our compiler optimization space. We reproduced the data on both 1-shot and 8-shot scenarios to conform with the number of inference (predictions) COBAYN has in the current work. We use the Harmonic mean to average the speedup here. Note that Harmonic mean is always less than or equal to the arithmetic mean [Hoefler and Belli 2015]. In all cases, COBAYN outperforms the reference methodology; specifically we have at least 1.3× and up to 2.04× speedup compared with the best achieved results reported in [Park et al. 2013].

5. CONCLUSIONS

This paper presents COBAYN, a methodology to infer by means of a *Bayesian framework* the best compiler optimizations to be applied for optimizing the performance of a target application. The methodology uses target independent software features to sample a statistical model built using Bayesian Networks to extract a set of suitable compiler configurations. Feature reduction techniques have been adopted to reduce the complexity and training time of the Bayesian model while also eliminating possible noise in the data and improving the quality of the results. The proposed approach has been evaluated on an ARM-based platform, using GCC compiler. The experimental results demonstrated that the proposed technique outperforms both standard optimization levels and state-of-the-art iterative and not iterative compilation techniques while using the same number of evaluations.

ACKNOWLEDGMENTS

This work is partially supported by the European Commission Call H2020-FET-HPC program under the grant ANTAREX-671623 and conducted in the context of the joint ASTRON and IBM DOME project and funded by the Netherlands Organization for Scientific Research (NWO), the Dutch Ministry of EL&I and the Province of Drenthe.

REFERENCES

Felix Agakov, Edwin Bonilla, John Cavazos, Björn Franke, Grigori Fursin, Michael FP O'Boyle, John Thomson, Marc Toussaint, and Christopher KI Williams. 2006. Using machine learning to focus iterative optimization. In *Pro-*

Table VII: COBAYN speedup w.r.t. the average speedup gained with predictive modeling in both 1-shot and 8-shot scenarios reported in [Park et al. 2013].

Algorithm and Parameter Configuration	COBAYN Speedup	
	w.r.t 1-shot	w.r.t 8-shot
LR -S 0	1.7551	1.6673
LR -S 1	1.7590	1.6710
LR -S 2	1.7191	1.6331
SVM NormalizedPolykernel -C 1.0 -E 8.0	1.5437	1.4665
SVM RBFKernel -C 2.0 -G 0.0	1.5206	1.4445
SVM RBFKernel -C 2.0 -G 25.0	1.5082	1.4327
SVM RBFKernel -C 2.0 -G 50.0	1.5045	1.4292
SVM RBFKernel -C 2.0 -G 75.0	1.5029	1.4277
SVM RBFKernel -C 2.0 -G 30.0	1.4927	1.4180
SVM RBFKernel -C 4.0 -G 30.0	1.5073	1.4319
SVM RBFKernel -C 0.01 -G 30.0	1.5073	1.4374
SVM RBFKernel -C 4.0 -G 50.0	1.5045	1.4292
IBk -K 1	1.4447	1.3724
IBk -K 2	1.4667	1.3933
IBk -K 5	1.4887	1.4142
M5P -M 1.0	1.4281	1.3566
M5P -M 2.0	1.4282	1.3567
M5P -M 4.0	1.4282	1.3568
M5P -M 10.0	1.4575	1.3846
M5P -M 50.0	1.4913	1.4167
K* -B 0 -M a	1.5192	1.4432
K* -B 20 -M a	1.5216	1.4455
K* -B 25 -M a	1.5258	1.4495
K* -B 50 -M a	1.4740	1.4003
K* -B 75 -M a	1.4737	1.4001
K* -B 100 -M a	1.5172	1.4413
K* -B 0 -M n	1.5208	1.4447
K* -B 20 -M n	1.5216	1.4455
K* -B 25 -M n	1.5258	1.4495
K* -B 50 -M n	1.4740	1.4003
MLP -L 0.3 -N 500 -H a	2.0435	1.9413
MLP -L 0.05 -N 500 -H a	1.6738	1.5901
MLP -L 0.1 -N 500 -H a	1.7246	1.6383
MLP -L 0.5 -N 500 -H a	1.8138	1.7231
MLP -L 0.9 -N 500 -H a	1.5250	1.4487
MLP -L 0.4 -N 500 -H a	1.9426	1.8454
MLP -L 0.5 -N 1000 -H a	1.8535	1.7608
MLP -L 0.5 -N 1500 -H a	1.7388	1.6518
MLP -L 0.5 -N 500 -H t	1.5579	1.4801
AVERAGE (Harmonic Mean)	**1.5622**	**1.4841**

ceedings of the International Symposium on Code Generation and Optimization. IEEE Computer Society, 295–305.

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 38–49. DOI:http://dx.doi.org/10.1145/1542476.1542481

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 303–316.

Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo, and Cristina Silvano. 2016. Predictive Modeling Methodology for Compiler Phase-Ordering. In *Proceedings of 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 5th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms*. ACM.

Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, and Cristina Silvano. 2014. A Bayesian network approach for compiler auto-tuning for embedded processors. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*. IEEE, 90–97.

Amir Hossein Ashouri, Vittorio Zaccaria, Sotirios Xydis, Gianluca Palermo, and Cristina Silvano. 2013. A framework for Compiler Level statistical analysis over customized VLIW architecture. In *Very Large Scale Integration (VLSI-SoC), 2013 IFIP/IEEE 21st International Conference on*. IEEE, 124–129.

Rajendra Bhatia. 2009. *Positive definite matrices*. Princeton University Press.

François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. 1998. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*.

Karsten M Borgwardt and Hans-Peter Kriegel. 2005. Shortest-path kernels on graphs. In *Data Mining, Fifth IEEE International Conference on*. IEEE, 8–pp.

John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. 2007. Rapidly Selecting Good Compiler Optimizations Using Performance Counters. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 185–197. DOI:http://dx.doi.org/10.1109/CGO.2007.32

Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. 2012. Deconstructing iterative optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 3 (2012), 21.

Keith D Cooper, Philip J Schielke, and Devika Subramanian. 1999. Optimizing for reduced code space using genetic algorithms. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 1–9.

Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May OReilly, and Saman Amarasinghe. 2015. Autotuning algorithmic choice for input sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 379–390.

Shuangde Fang, Wenwen Xu, Yang Chen, Lieven Eeckhout, Olivier Temam, Yunji Chen, Chengyong Wu, and Xiaobing Feng. 2015. Practical iterative optimization for the data center. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 2 (2015), 15.

Grigori Fursin. 2010. Collective benchmark (cbench), a collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization. (2010).

Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, and others. 2011. Milepost gcc: Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming* 39, 3 (2011), 296–327.

Grigori Fursin, Cupertino Miranda, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Ayal Zaks, Bilha Mendelson, Edwin Bonilla, John Thomson, Hugh Leather, and others. 2008. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*.

Richard L Gorsuch. 1988. Exploratory factor analysis. In *Handbook of multivariate experimental psychology*. Springer, 231–258.

Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*. IEEE, 1–10.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

David Heckerman and David M. Chickering. 1995. Learning Bayesian networks: The combination of knowledge and statistical data. In *Machine Learning*. 20–197.

Torsten Hoefler and Roberto Belli. 2015. Scientific benchmarking of parallel computing systems: twelve ways to tell the masses when reporting performance results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 73.

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

Kenneth Hoste and Lieven Eeckhout. 2008. Cole: compiler optimization level exploration. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 165–174.

Texas Instruments. 2012. PandaBoard. *OMAP4430 SoC dev. board, revision A* 2 (2012).

Zhanpeng Jin and A.C. Cheng. 2008. Improve simulation efficiency using statistical benchmark subsetting - An implantbench case study. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. 970–973.

Henry F Kaiser. 1958. The varimax criterion for analytic rotation in factor analysis. *Psychometrika* 23, 3 (1958), 187–200.

Christos Kartsaklis, Oscar Hernandez, Chung-Hsing Hsu, Thomas Ilsche, Wayne Joubert, and Richard L Graham. 2012. HERCULES: A pattern driven code transformation system. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 574–583.

Toru Kisuki, Peter MW Knijnenburg, Mike FP O'Boyle, François Bodin, and Harry AG Wijshoff. 1999. A feasibility study in iterative compilation. In *High Performance Computing*. Springer, 121–132.

Toru Kisuki, Peter M. W. Knijnenburg, and Michael F. P. O'Boyle. 2000. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques (PACT'00), Philadelphia, Pennsylvania, USA, October 15-19, 2000*. 237–248. DOI:http://dx.doi.org/10.1109/PACT.2000.888348

Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. 2009. Practical exhaustive optimization phase order exploration and evaluation. *ACM Trans. Archit. Code Optim.* 6, 1, Article 1 (April 2009), 36 pages. DOI:http://dx.doi.org/10.1145/1509864.1509865

Sameer Kulkarni and John Cavazos. 2012. Mitigating the compiler optimization phase-ordering problem using machine learning. *ACM SIGPLAN Notices* 47, 10 (2012), 147–162.

Hugh Leather, Edwin Bonilla, and Michael O'Boyle. 2009. Automatic feature generation for machine learning based optimizing compilation. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 81–91.

Junghsi Lee and V John Mathews. 1994. A stability condition for certain bilinear systems. *IEEE transactions on signal processing* 42, 7 (1994), 1871–1873.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. DOI:http://dx.doi.org/10.1145/1065010.1065034

Luiz GA Martins, Ricardo Nobre, João MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. 2016. Clustering-Based Selection for the Exploration of Compiler Optimization Sequences. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 1 (2016), 8.

Sanyam Mehta and Pen-Chung Yew. 2015. Improving compiler scalability: optimizing large programs at small price. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 143–152.

Kevin P. Murphy. 2001. The Bayes Net Toolbox for MATLAB. *Computing Science and Statistics* 33 (2001), 2001.

Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. 2005. Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing* 1, 3 (2005), 305–316.

EunJung Park, John Cavazos, and Marco A Alvarez. 2012. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 196–206.

Eunjung Park, John Cavazos, Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and P Sadayappan. 2013. Predictive modeling in a polyhedral optimization space. *International Journal of Parallel Programming* 41, 5 (2013), 704–750.

EunJung Park, Christos Kartsaklis, and John Cavazos. 2014. HERCULES: Strong Patterns Towards More Intelligent Predictive Modeling. In *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 172–181.

Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]* (2012).

Suresh Purini and Lakshya Jain. 2013. Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 56.

R Ihaka R Gentleman. 2012. R Statistical tool. (2012). http://www.r-project.org/

Roberto Santana, Concha Bielza, Pedro Larranaga, Jose A Lozano, Carlos Echegoyen, Alexander Mendiburu, Rubén Armananzas, and Siddartha Shakya. 2010. Mateda-2.0: Estimation of distribution algorithms in MATLAB. *Journal of Statistical Software* 35, 7 (2010), 1–30.

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. *ACM SIGPLAN Notices* 49, 6 (2014), 53–64.

Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Notices*, Vol. 38. ACM, 77–90.

Mirai Tanaka and Kazuhide Nakata. 2014. Positive definite matrix approximation with condition number constraint. *Optimization Letters* 8, 3 (2014), 939–947.

Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huyng, Xibai Li, and Rick Siow Mong Goh. 2015. Optimizing and auto-tuning scale-free sparse matrix-vector multiplication on Intel Xeon Phi. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*. IEEE, 136–145.

Bruce Thompson. 2002. Statistical,practical, and clinical: How many kinds of significance do counselors need to consider? *Journal of Counseling & Development* 80, 1 (2002), 64–71.