A Bayesian Network Approach for Compiler Auto-tuning for Embedded Processors

Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo and Cristina Silvano
Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy
{amirhossein.ashouri, giovannisiro.mariani, gianluca.palermo, cristina.silvano}@polimi.it

Abstract—The complexity and diversity of today’s architectures require an additional effort from the programmers in porting and tuning the application code across different platforms. The problem is even more complex when considering that also the compiler requires some tuning, since standard optimization options have been customized for specific architectures or designed for the average case. This paper proposes a machine-learning approach for reducing the cost of the compiler auto-tuning phase and to speedup the application performance in embedded architectures. The proposed framework is based on an application characterization done dynamically with micro-architecture independent features and based on the usage of Bayesian Networks. The main characteristic of the Bayesian Network approach consists of not describing the solution as a strict set of compiler transformations to be applied, but as a complex probability distribution function to be sampled. Experimental results, carried out on an ARM platform and GCC transformation space, proved the effectiveness of the proposed methodology for the selected benchmarks. The selected set of solutions (less than 10% of the search space) demonstrated to be very close to the optimal sequence of transformations, showing also an applications performance speedup up to 2.8 (1.5 on average) with respect to -O2 and -O3 for the cBench suite. Additionally, the proposed method demonstrated a 3x speedup in terms of search time with respect to an iterative compilation approach, given the same quality of the solutions\(^1\).

I. INTRODUCTION

Nowadays, most SW applications are first developed in a source high-level programming language (e.g. C, C++) and then compiled to generate their executable binary implementations. Compiler optimizations (e.g. loop unrolling, register allocation, etc.) might lead to substantial benefits in reference to several performance metrics (e.g. execution time, power consumption, memory footprint). Application developers are relying on compilers intelligence to optimize the application performance, while being often unaware of how the compiler optimization process can modify the application itself. Today compilers’ interfaces expose different options that allow the programmer to specify the optimizations to be applied. It is usually possible to specify optimization levels to automatically include a set of predefined optimizations, known to be beneficial for application performance in most cases [12]. There are also several specific compiler optimizations that are beyond the ones included in predefined optimization levels. The effects of enabling these compiler optimizations are quite complex. These effects mostly depend on the features of the target application and on the decision whether or not enabling some other compiler optimizations. It is rather hard to decide the

compiler optimizations to be enabled to maximize application performance.

When considering application-specific embedded systems, where the application is compiled once and then deployed on the market, exploiting the available compiler optimizations to achieve the best performance prior to the deployment represents an even more crucial task. So far researchers proposed two main solutions to identify the best compiler optimization strategy: a) iterative compilation [5] and b) machine learning predictive techniques [22]. The former approach consists of recompiling several times the application using different compiler optimizations and then choosing the executable binary providing the best performance. However, Due to the high number of compilation runs to be carried out, iterative compilation is a time consuming procedure. The latter approach consists of machine learning techniques, where some compiler optimizations shall be applied in reference to some software features. Once the machine learning model has been trained, given a target application, the model predicts a sequence of compiler optimizations to maximize performance. Once trained, the use of machine learning does not require to carry out several compilations. However, the performance of the final executable binary is typically worse than the one found with iterative compilation.

In this work, we propose an innovative approach to tackle the problem of the identification of the compiler optimizations that maximize the performance of a target application. We apply a statistical methodology to infer the probability distribution of the compiler optimizations to be enabled to achieve the best performance. Then we start to drive an iterative compilation process by sampling from this probability distribution. Similarly to machine learning approaches, we use a set of training applications to learn the statistical relationships between application features and compiler optimizations. Bayesian networks are used to learn the statistical model. Given a new application not included in the training set, its software features are fed to the Bayesian network as evidence on the distribution. This evidence generates a bias on the distribution, since compiler optimizations are correlated to software features.

The obtained probability distribution is application-specific and effectively focuses on the iterative compilation process limited to the most promising compiler optimizations. Experimental results carried out on an embedded ARM-based computing platform demonstrate a 3x speedup in reference to state of the art iterative compilation techniques [5]. To summarize, the contributions of this work are:

\(^1\)This work was supported in part by the EC under the grants FP7-HARPA-612069 and FP7-CONTREX-611146
A Bayesian model capable to capture the relationships among different compiler optimizations and program features. This approach is based on statistical analysis and represents the leaned relationships as a directed acyclic graph that can be easily inspected and understood by humans.

The integration of the Bayesian network model within a compiler optimization framework. The probability distribution of the compiler optimizations to be used for a new, previously unobserved, program is inferred by means of the Bayesian network to focus the optimization itself.

The application and the assessment of the proposed methodology to focus on for the optimization of an embedded ARM-based platform.

The rest of the paper is organized as follows. Section II provides a brief discussion on the related work. In Section III, we introduce the proposed statistical approach to iterative optimization. Section IV presents experimental evaluation of the proposed methodology on an ARM platform. Finally, Section V summarizes the outcomes of the work and envision some future research paths.

II. RELATED WORK

Optimizations carried out at compilation time are used broadly, especially 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. The related work in this field can be categorized as a) iterative compilation and, b) machine learning based approaches.

Iterative compilation was introduced as a technique capable of outperforming static handcrafted optimization sequences, those usually exposed by compiler interfaces as optimization levels. Since the beginning of its introduction [3], [14], the goal of iterative compilation was to identify the most appropriate compiler passes for a target application.

More recent literature discusses about down-sampling techniques to reduce the search space [21] with the goal of identifying compiler optimization sequences per each application.

Other authors exploit iterative compilation jointly with architectural design space exploration for VLIW architectures [2]. Their intuition is that the performance of a computer architecture is dependent on the executable binary that, in turn, depends on the optimizations applied at compilation time. Thus, by studying the two problems jointly, the final architecture is optimal in reference to the compilation technique in use and the effects of different compiler optimizations are identified at the early design phases.

Given that compilation is a time consuming task, several researches 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 [1], [8], [18]. Milepost [18], 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 [8]. Authors in [16], introduced a methodology for learning static code features to be used within compiler heuristics that drive the optimization process.

There are previous works focusing on the use of empirical modeling to build application-specific performance models for compiler optimizations. Authors in [24], tried to capture the interactions between compiler flags and the micro-architectural parameters without any prior knowledge. The outcome could benefit the compiler writers to exploit the best effective parts of the design space to further improve the performance.

In [23], authors suggest to include iterative compilation within the compiler itself to enable the heuristics implementing the iterative compilation mechanism. In this way, iterative compilation is carried out at the level of each code segment but it is significantly sped up by focusing the research as defined by the heuristics embedded within the compiler. Authors in [5] evaluate the effects of dataset variations on the iterative compilation process. It turns out that iterative compilation is a robust approach capable of achieving more that 83% of the potential speedup for all datasets for the selected benchmarks.

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. Our work is, to some extent, related to the one proposed in [1] where machine learning was used to capture the probability distribution of different compiler transformations. However in this work we propose the use of Bayesian networks 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 the Bayesian network to induce an application-specific bias on the probability distribution of compiler optimizations. Additionally, in our approach, program features are obtained through micro-architectural independent dynamic profiling [11] rather than static code analysis. The adoption of dynamic profiling provides insight data about the actual program execution with the purpose to give more weight to the code segments executed more often (i.e. code segments whose optimization would lead to higher benefits).

III. THE PROPOSED METHODOLOGY

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 generate a parametric representation of its dynamic features. A statistical model based on Bayesian networks correlates these features to the compiler optimizations such as to maximize the application performance.

The optimization flow is represented in Figure 1 and it consists of two main phases. During the initial training phase, the Bayesian network is leaned on the base of a set of training applications (see Figure 1a). During the final exploitation phase, new applications are optimized by exploiting the knowledge stored in the Bayesian network.

During both phases, an optimization process is necessary to identify the best compiler optimizations to be enabled
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. This DSE engine compiles, executes and measures application performance by enabling and disabling different compiler optimizations. The compiler optimizations to be enabled are decided by a Design of Experiments (DoE) plan. In our approach the DoE plan is obtained by sampling from a given probability distribution that is either an application-specific distribution (during the training phase as in Figure 1a) or an application-specific distribution inferred through the Bayesian network (during the exploitation phase as in Figure 1b).

The uniform distribution adopted during the training phase allows us to explore uniformly the compiler optimization space $O$ thus to discover and learn what are the most promising regions of this space. The application-specific distribution used during the exploitation phase allows us to speedup the optimization by focusing on the most promising region of the compiler optimization space $O$.

A. Application characterization

In this work, we use PIN [17] based dynamic profiling framework to analyze the behavior of the different applications at execution time. In particular, the selected profiling framework provides a high level Micro-architectural Independent Characterization of Applications (MICA) [11] that is suitable for characterizing applications in a cross-platform manner. Furthermore, there is no static syntactic analysis but the whole framework is based solely on MICA profiling.

In our experimental setup, the application is compiled and profiled on an x86 host processor, while the target architecture where the application executes (i.e. the architecture for which the application shall be optimized) is an embedded ARM-based platform. Thanks to the high level abstraction of the application characterization carried out with MICA, we can easily change the target architecture without the need of changing the profiling infrastructure.

The MICA framework reports data about instruction types, memory and register access pattern, potential instruction level parallelism and a dynamic control flow analysis in terms of branch predictability. Overall, the MICA framework characterizes an application in reference to 99 different metrics (or features). However, many of these 99 features are strongly correlated (e.g. the number of memory reads with stride smaller than 1K is bounded by the number of reads with stride smaller than 2K). Furthermore, generating a Bayesian network is a process whose time complexity grows up super-linearly with the number of parameters in use. It is practically expensive to include all the 99 features in the Bayesian network model. Thus, with the goal of speeding up the construction of the Bayesian network, we apply Principal Component Analysis (PCA) to reduce the number of parameters to characterize the application. PCA is a technique to transform a set of correlated parameters (application features) into a set of orthogonal (i.e. uncorrelated) principal components [13]. The PCA transformation aims at sorting the principal components by descending order based on their variance. For instance, the first components include the most of the input data variability, i.e. they represent the most of the information contained in the input data. To reduce the number of input features, while keeping most of the information contained in the input data, it is simply needed to use the first $k$ principal components as suggested in [11]. In particular, we set $k = 10$ to trade off the information stored in the application characterization and the time required to train the Bayesian network.

B. The Bayesian framework

Background on Bayesian networks. Bayesian networks is a powerful tool 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 $o$ 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 [15] is not taken into account but rather we consider that different optimizations $o_i$ are organized in a predefined order embedded in the compiler. A compiler optimization sequence represented by the vector $o$ belongs to the $n$ dimensional Boolean space $O = \{0,1\}^n$, where $n$ represents the number of compiler optimizations under study.

An application is parametrically represented by the vector $\alpha$ whose elements $\alpha_i$ are the first $k$ principal components of its dynamic profiling features. Note that elements $\alpha_i$ in the vector $\alpha$ generally belong to the continuous domain.

The optimal compiler optimization sequence $\tilde{o} \in O$ that maximizes the performance of an application is generally unknown. However it is known that the effects of a compiler optimization $o_i$ may depend on whether or not another optimization $o_j$ is applied. Additionally it is known that the com-

![Fig. 1: Overview of the proposed methodology.](image-url)
piler 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 priori is because it is not possible to capture in a deterministic way the dependencies amongst the variables in the vectors \( \bar{o} \) and \( \alpha \). There is no way to identify an analytic model to exactly fit the vector function \( \bar{o}(\alpha) \). In 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.

Given the peculiarities of the problem, we selected as underlying probabilistic model the Bayesian networks. In particular, Bayesian networks have the following features of interest for the target problem:

- Their expressiveness allows to include in the same framework heterogeneous variables such as Boolean variables (in the optimization vector \( \bar{o} \)) and continuous variables (in the application characterization \( \alpha \)).
- They are suitable for modeling cause-effect dependencies: for the target problem 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 priory knowledge.
- It is possible to bias the probability distribution of some variables (the optimization vector \( \bar{o} \)) given the evidence on other variables (the application characterization \( \alpha \)). This enables to infer an application-specific distribution for the vector \( \bar{o} \) from the vector \( \alpha \) observed by analyzing the target application.
- It is a general optimization model making it feasible to infer a probabilistic model both with and without the kernel characterization i.e. static analysis or micro-architecture independent characterization.

A Bayesian network is a direct acyclic graph whose nodes represent variables and whose edges are the dependencies between these variables. Figure 2 reports a simple example with one variable \( \alpha_1 \) representing the application features and two variables \( o_1, o_2 \) representing different compiler optimizations. Let us consider that \( \alpha_1 \) represents the number of profiled HW-loop instructions while \( o_1, o_2 \) are two optimizations operating on loops (e.g. loop unrolling \( o_1 \) and loop tiling \( o_2 \)). Of course, the presence of loops can be observed by profiling \( \alpha_1 \) and it is the primary cause of loop optimizations’ effects. Additionally the effectiveness of whether or not to apply loop tiling might depend on the decision taken about applying or not loop unrolling. All these dependencies can be automatically discovered by training the Bayesian model and are represented as edges on the network topology graph (Figure 2). Note that we use dashed lines for nodes representing observable variables whose value can be input as evidence to the network. In this example, the variable \( o_1 \) can be observed and, by introducing its evidence, it is possible to bias the probability distributions of other variables.

**Training the Bayesian model.** Off-the-shelf tools exist to construct Bayesian networks automatically by fitting the distribution of some training data [19]. To do so, first the graph topology shall be identified and then the probability distribution of the variables including their dependencies shall be estimated.

The identification of the graph topology is particularly complex and time consuming. To avoid unreasonable training times, we decided to reduce the number of nodes in the graph by limiting the number \( k \) of elements in the vector \( \alpha \). This is the reason for using PCA during application characterization. Nonetheless, for efficiency reasons, the algorithm used for selecting the graph topology is a greedy algorithm, named K2, initialized with the Maximum Weight Spanning Tree (MWST) ordering method as suggested in [10]. Note that the initial ordering of the nodes for the MWST algorithm is given to let the elements \( \alpha \) to appear first and then the elements of \( o \). Even if the final topological sorting of the nodes changes according to the algorithm described in [10], by using this initialization criterion it always happens that the dependencies are directed from elements of \( \alpha \) to elements of \( o \) and not vice versa.

The coefficients of the functions describing the probability distributions of each variable as well as their dependencies are tuned automatically to fit the distribution in the training data [19]. Within the proposed Bayesian model, a relationship between two different variables (e.g. between a program feature \( \alpha_j \) and a compiler optimization \( o_i \)) is modeled as a conditional probability function \( P(o_i = b | \alpha_j = x) \). To estimate these functions, 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 \( \alpha \). Then, an experimental compilation campaign is carried out for each application by sampling several compiler optimization sequences from the compiler optimization space \( O \) with an uniform distribution. For each application we select the 15% best performing compilation sequences among the sampled ones. The distribution of these sequences are learned by the Bayesian network framework in relation to the vector \( \alpha \) characterizing the application.

**Inferring an application-specific distribution.** Once the Bayesian network is trained, the principal component vector \( \alpha \) obtained for a new application can be fed as evidence to the framework to bias the distribution of the compiler optimization vector \( o \). To sample a compiler optimization sequence from

---

**Fig. 2:** A Bayesian network example.
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 \( \alpha \) 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 \( P(o_i = 0 \mid P) \) and \( P(o_i = 1 \mid P) \) can be computed on the base of the parent node vector value \( P \) (being each parent either an evidence \( \alpha_j \) 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.

IV. EXPERIMENTAL EVALUATION

A. Experimental Setup

The experimental results have been generated by using an ARMv7 Cortex-A9 architecture on a TI OMAP 4430 processor [20] with ArchLinux and GCC-ARM 4.6.3 running on top of it. We considered the cBench benchmark suite [7] including 24 applications and 5 datasets. Table I reports the compiler optimizations analyzed in this paper. This set of compiler transformations has been derived from [5]. In particular, we selected the compiler optimizations whose speedup factor is reported to be greater than 1.10. These compiler optimizations can be enabled/disabled by means of their respective compiler optimization flags. They are applied to improve application performance beyond the predefined optimization level -O3. The application execution time is estimated by using the Linux-Perf tool. Execution time required to process a given dataset by a compiled application binary is estimated by averaging five different executions.

B. Bayesian Model Analysis

In our experimental setup, we have used the Matlab environment [19] to train the Bayesian network by correlating the first 10 Principal Components (PCs) of application features to the 7 compiler optimizations listed in Table I. This step takes about 1.5 minutes on an Intel Core i7 processor at 2.4 GHz. An analysis on the feasibility of applying the approach over a larger number of compiler optimization flags has been carried out and demonstrated that, training a Bayesian model including 24 compiler optimizations takes about 1 hour per application.

In this work, the validation has been carried out through cross-validation. We train different Bayesian networks, each one by excluding from the training application set one of the applications. Validation data are then gathered on the application excluded from the training set.

With the purpose of investigating graphically the dependencies between the variables involved in the compiler optimization problem, we trained a final Bayesian network including all the applications in the training set. The resulting network topology is shown in Figure 3. By removing one application from the training set, the graph topology changes slightly, mainly in terms of the different edges connecting PC nodes to compiler optimization nodes. This effect is due to the removal of the application in the principal component analysis and thus to the fact that principal components are computed in a different way. For conciseness we do not report the graph topologies for each one of the trained Bayesian network.

Nodes of the topology graph reported in Figure 3 are organized in layers. The first layer reports the PCs that are the observable variables (reported as dashed lines). In the second layer, there are represented the compiler optimizations whose parents are the PC nodes. Thus the effects of these compiler optimizations depend only on the application characterization in terms of its features. In the third layer, there are compiler optimization nodes whose parents include optimization nodes from the second layer. Once a new application is characterized for a target application-dataset, the evidence related to the PCs of its features are fed to the network in the first layer. Then, probability 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 on the predictability of the branches implementing these loops. Moreover, funroll-lo impacts on the effectiveness of the heuristic branch probability estimation, thus on fn-gss-br. The second node in the third layer is the fn-ivopt node, that 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 or not to apply fn-gss-br and fn-ivopt are taken after deciding whether or not to apply funroll-lo and fn-tree-opt.

C. Comparison Results

The proposed methodology samples different compiler optimization sequences from the Bayesian network. The performance achieved by the best application binary depends on the number of sequences sampled from the model. Performance

\(^2\)This is by construction due to the initialization of the MWST and the K2 algorithms used to discover the network topology.
TABLE I: Compiler optimizations under analysis, to be introduced beyond -O3

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

![Fig. 4: Performance improvement obtained by sampling 8 optimization sequences from the probability distribution inferred by the proposed Bayesian model (in reference to -O2 and -O3).](image)

speedup is measured in reference to -O2 and -O3, which are the optimization levels available for GCC.

Figure 4 reports these speedups by considering a sample of 8 different compiler optimization sequences (results are averaged on the different datasets). All applications have been sped up in reference to -O2 and only consumer-jpeg-d has been slowed down in reference to -O3. For two datasets out of the five tested for this application, -O3 returns the best performing executable binary and it is not possible to exceed its performance. On average, the speedups are of 1.56 and 1.48 in reference to -O2 and -O3 respectively and the max speed-up of 2.85 and 2.7 has been observed for network-patricia w.r.t -O2 and -O3 respectively.

It is known that Iterative Compilation can generally improve application performance in reference to static hand-crafted compiler optimization sequences [1]. 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 optimization algorithms such as genetic algorithms or simulated annealing [1], [4], [5]. For these reasons, to evaluate the proposed approach we compare to a random methodology that samples compiler optimization sequences from the uniform distribution.

Let us define the normalized speed-up $\tau$ as the ratio of the achieved performance improvement over the whole potential performance improvement:

$$\tau = \frac{E_{ref} - E}{E_{ref} - E_{best}}$$  \hspace{1cm} (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). Note that, as the execution time $E$ of the iterative compilation methodology under analysis gets closer to the reference execution time $E_{ref}$, the $\tau$ 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 $\tau$ gets close to 1 reporting that the whole potential performance improvement has been achieved.

Figure 5, reports the $\tau$ achieved by the proposed optimization technique and by the random iterative compilation technique in reference to execution time obtained by -O3 ($E_{ref}$). Similarly, Figure 6 reports the normalized speed-up $\tau$ of the proposed approach in reference to the random iterative compilation technique ($E_{ref}$). Moreover, to highlight the benefits of including the application characterization based on MICA in the Bayesian model, Figure 6 reports the evaluation of the proposed approach when including (continuous line) or not including (dotted line) them in the model. The comparisons reported in Figures 5 and 6 have carried out by considering
The same number of compiler optimization sequences sampled for both the random iterative compilation and the proposed approach.

We can observe in Figure 5 that, by sampling a number of compiler optimization sequences greater than 4, both approaches under analysis outperform the reference optimization level of -O3 reporting positive values of \( \tau \). The normalized speed-up with respect to -O3 grows quickly, thus resulting in the 80% of the potential performance when extracting 20 and 24 compiler optimizations sequences respectively for the the proposed approach and the random iterative compilation approach. We can also observe that the proposed approach outperforms the iterative methodology. By increasing the number of extractions, the achieved normalized potential improvement goes from about 15% (from 10 to 20 extractions) to about 10% (above 20 extractions). This is reported more clearly in Figure 6. When targeting the extraction of very few compiler optimization sequences (x axis, from 1 to 9 extractions), the proposed methodology achieves about 25% of the potential performance improvement w.r.t. the random iterative compilation methodology. Furthermore, the proposed methodology has been evaluated with both including/excluding the platform independent characterizations and performance improvement utilizing the MICA is clearly observable.

**Evaluation of a practical usage.** When using iterative compilation in practice, a typical approach is to decide the efforts to be put on the optimization itself. These efforts can be measured in terms of optimization time, that is directly proportional to the number of compilations to be executed. Thus, we evaluated the proposed optimization approach in terms of application performance reached after a fixed number of compilations. In particular we fix this number to 8 that represents 6.25% of the overall optimization space.

This evaluation has been carried out in reference to the random iterative compilation methodology as it has been demonstrated to be able to outperform static/dynamic compilation techniques on the long-run [1], [6], [9]. The goal is to estimate the speedup achieved in terms of both application performance and compilation efforts. Figure 7 reports the box-plot analysis of the application speedup, while keeping the compilation efforts of the proposed methodology to 8 compilations (or extractions) and varying the compilation efforts of the random iterative compilation. Each box represents the data excluding the outliers having set the two quantiles respectively 90% and 10%. The central box notch reports the median.

When using the same compilation efforts of 8 extractions for the random approach, we obtain an application speedup of 1.09 on average. The box-plot distribution demonstrates that it happens very rarely to obtain a slowdown in reference to the random iterative compilation when considering the same compilation efforts.

Of course, by increasing the compilation efforts of the random iterative compilation above 8 extractions while keeping constant the exploration efforts of the proposed approach, the application speedup decreases. The random iterative compilation needs an efforts of 24 extractions to achieve on average the application performance obtained with 8 extractions from the proposed methodology. This means that the proposed methodology provides a speedup of 3× in terms of optimization efforts.

**V. Conclusions**

This paper presents a methodology to infer within a Bayesian framework the best compiler optimizations to be applied for optimizing the performance of a target application. The methodology has been validated for an ARM-based platform considering the GCC compiler. In average, it delivers 1.56 and 1.48 performance speedup respectively in reference to the optimization levels -O2 and -O3 when considering an optimization effort of 8 compilations (about 6% of the considered compiler optimization space). A factor of 3× speedup in terms of optimization efforts has been empirically
measured in reference to state of the art iterative compilation approaches.

**REFERENCES**


