This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

Automatic Software Tailoring for Optimal Performance

J. M. Aragón-Jurado, J. C. de la Torre, P. Ruiz, P. L. Galindo, A. Y. Zomaya and B. Dorronsoro

Abstract—Efficient green software solutions require being aware of the characteristics of both the software and the hardware where it is executed. Separately optimizing them leads to inefficient results, and there is a need for a perfect synergy between software and hardware for optimal outcomes. We present a novel combinatorial optimization problem for the minimization of the software execution time on a specific hardware, taking into account the existing uncertainty in the system. A solution to the problem is a sequence of LLVM code transformations, and a cellular genetic algorithm is used to find it. Assuming that hardware does not change, reducing the software runtime typically leads to a greener version with lower consumption. To cope with the uncertainty, two novel approaches relying on bootstrap method to compute confident intervals of the software runtime at negligible cost are proposed and compared to three other techniques and -O3 Clang compilation flag over four hardware architectures. Results show how the proposed approach effectively copes with the uncertainty, providing more robust solutions with respect to the compared methods. The execution time of the raw program is reduced from 28.1% to up to 63.2%, outperforming -O3 flag by 13.9% to 26.3%, for the different architectures.

Index Terms—Software performance, LLVM, execution time, uncertainty, source code transformation, evolutionary computation.

1 Introduction

Sustainability of IT systems can be enhanced by making a more efficient use of the available resources of software programs (SW) running on a given hardware platform (HW). This is usually accomplished during the compilation process where compilers perform a number of modifications in the code with the aim of optimizing some attribute(s) of the resulting program, as its memory usage, execution time, or consumption, among others. For that, a generic combination of code transformations is used, that is expected to optimize the performance of typical programs.

However, using a generic approach does not allow fully exploiting the specific properties of both the SW and the HW architecture for an optimal performance [1]. Therefore, current compilers fail in making an efficient use of the underlying HW (mainly due to the large number of different existing architectures) [2], [3], [4], [5], [6]. This purpose is extremely difficult, especially when new architectures with well differentiated HW elements are continuously being released (changing the instruction pipeline). Moreover, even similar HW platforms present differences in their architecture, which might require distinct operations to optimize a given SW, bringing to light the unsuitability of a generic optimization.

Furthermore, SW plays an important role in how HW behaves. In order to optimize the performance of some SW, the sequence of instructions the program performs needs to be taken into account. Decision making not only needs to be influenced by the types of instructions but also by the order in which they are executed and the objective of the SW. Thus, current generic optimization approaches are not optimal because both SW and HW properties need to be taken into account for an effective SW optimization.

Consequently, an *ad hoc* compilation approach is necessary to get the best performance of a SW in a given HW. Nevertheless, current compilers only apply static sequences of transformations for any SW. Therefore, there is a need for a new generation of smart compilers which could decide the right sequence of code transformations to apply according to the specific SW and HW properties. A relevant compiler infrastructure which supports multiple programming languages and sets of instructions is LLVM [7]. It offers a high number of code optimizations, called *passes* (there are 87 passes available in LLVM version 9.0.1), and its optimizer, called `opt`, allows the selection of any of these passes to modify the considered SW.

Unfortunately, the impact of a single code transformation on the SW performance is generally very low; to the point that it is very difficult to quantify. However, this influence may considerably increase when it is applied together with other transformations [8]. Additionally, the impact of a set of transformations is subject to the order in which they are applied (this is known as the phase ordering problem [2], [3]), and also performing some transformations several times is often beneficial [9]. Moreover, the optimal length of the sequence of code transformations, i.e. the optimal number of code transformations, may vary depending on the considered HW and SW.

Therefore, the main purpose of a tailored sequence of code transformations for specific SW on a given HW is to improve its performance. This improvement can be measured in terms of different features of the SW, being the execution time among the main ones, and directly related to the green software paradigm. However, measuring SW

- *J.M. Aragón-Jurado, J. C. de la Torre and P. L. Galindo, are with the University of Cádiz, Spain.*
  E-mail: {josemiguel.aragon,juan.detorre,pedro.galindo}@uca.es
- *A. Y. Zomaya is with The University of Sydney. Australia*
  E-mail: albert.zomaya@sydney.edu.au
- *P. Ruiz and B. Dorronsoro are with the University of Cádiz, Spain and The University of Sydney. Australia*
  E-mail: {patricia.ruiz,bernabe.dorronsoro}@uca.es

Manuscript received May 11, 2023; revised August 20, 2023.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

runtime is not an easy task [10]. There are factors such as the simultaneous execution of processes by the operating system which cause high uncertainty in runtime measurements when a SW program is run several times on the same HW [10], [11]. This uncertainty causes deceptive runtime measurements, which are not fully representative of the SW performance. Therefore, its treatment is mandatory to assess meaningful runtime values and avoid any negative bias in the search of optimal transformation sequences.

In this work, we model as a combinatorial optimization problem the task of finding an optimal sequence of code transformations so that it optimizes the runtime of the considered SW on a given HW, taking into account the uncertainty in the runtime measurements. Solving this problem leads to an automatically generated tailored compiled SW version that allows the fully exploitation of both, the SW and HW specificities for optimal performance. The LLVM optimizer, `opt`, is used to apply this sequence to the studied SW, and all available LLVM passes are considered. Moreover, we compare five different uncertainty treatment approaches in our search for accurate and robust SW runtime evaluations. The problem is tackled with a cellular genetic algorithm (cGA), where the notion of distance between solutions is introduced, so that only those located next to each other are allowed to exchange information. The pursued effect is a slow spread of good information in the population, helping to keep diversity for longer and considerably increasing the search space exploration capabilities of the algorithm, thus reducing the risk of premature convergence with respect to other classical GAs [12].

Several contributions are made in this work. First of all, a mathematical definition is proposed to model the software code optimization problem (that we call SCOP for short) as a combinatorial problem to optimize SW runtime. Second, we propose the use of bootstrap technique to build confident intervals for the fitness value of the actual performance of solutions under uncertain environments with statistical confidence. Third, five different methods are used in order to deal with the uncertainty present in the SW runtime measurements, two of them are proposed in this work for the first time, and make use of bootstrap technique to build confident intervals.

We have performed extensive experiments, where several benchmark programs are optimized for four devices with different processors (Intel Core and ARM) and purposes (low-power and high-performance computing). A comparative study about the proposed approaches in all considered architectures is performed, analyzing the impact of uncertainty in the cGA evolution, and evaluating the quality of the reported solutions. Finally, we analyze the sequences of transformations proposed in the best solutions for each architecture, and their comparison against the state-of-the-art `-O3` Clang compilation flag.

The document is organized as follows: in Section 2, several related works are revised and compared with our study. Next, the SCOP problem is defined in Section 3. The methodology followed in this work is presented in Section 4, while the experimental setup is detailed in Section 5. Finally, Sections 6 and 7 present and discuss the results obtained, and in Section 8 our final conclusions as well possible lines of future work are described.

2 Related work

This section presents an overview of the most important papers in the topic of this work. It is first described in Section 2.1 the importance of SW runtime and its impact on the consumption of the device executing it. The most relevant works in the field of SW runtime optimization are then presented in Section 2.2, while the main techniques proposed to deal with uncertainty in optimization problems are reviewed in Section 2.3. Finally, a short overview on the different ways in which SW can contribute to sustainability is given in Section 2.4.

2.1 The impact of software runtime on its sustainability

Different factors markedly influence HW consumption, as the memory size and speed, the storage capacity, or the CPU frequency, among many others. Indeed, a list of key parameters influencing the carbon footprint of SW programs recently published in [13] include the specific HW characteristics together with runtime and geographic location. Given that the purpose of this work is to tailor a SW for optimal performance on a given HW architecture, runtime arises as the most influencing parameter in HW consumption, given that none of the other aforementioned parameters change.

SW consumption is estimated as a directly proportional value of its runtime in a number of recent works [14], [15]. According to these works, we can say that, under the assumption that the HW architecture does not change, reducing the execution time of a given SW leads to a greener version of itself, that requires less energy to compute.

It is known that reducing the speed of a task so that it ends just before its deadline is not power-efficient [16]. The reason is that when the processor slows down it consumes less, but other peripheral components remain activated longer, causing a higher overall consumption (CPU consumption supposes roughly 60% of the consumption of the device [17]). Additionally, accomplishing the tasks earlier allows hibernating those resources (disks, displays, network interfaces).

Compilers are useful tools that can help reducing power consumption of SW in several ways [16]. For example, as discussed in [16], they can apply common performance-oriented optimizations (such as the ones considered in this work) that lead to energy savings by reducing the execution time. Also, the work emphasizes how compilers can also reduce memory accesses to reduce energy dissipated in these accesses. This can be done by eliminating redundant load and store operations, by keeping data as close as possible to the processor (i.e., in registers or lower-level caches), or by altering data traversal patterns through the modification of loops, among others. However, as stated in [16], some of the compiler-level energy-saving techniques might negatively affect others. For instance, exploiting the full bandwidth of a banked memory architecture is an appropriate technique to reduce memory consumption, but it affects cache locality and energy consumption. Therefore, finding a sequence of transformations to optimize a given SW is a difficult problem, that also heavily depends on the characteristics of the SW itself, as well as of those of the HW were it executes.

The focus in this paper is finding a sequence of transformations at compilation time so that the runtime of the

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

SW program is minimized. Such transformations are generic source code optimizations that compilers usually make use of, and they consist of changing the memory use, promoting the use of registers, or changing the set and sequence of instructions to perform different operations, among other actions. Such transformations lead to changes in the order in which tasks are executed, and it directly affects the state of the cache, the memory use pattern, or the stress of CPU and other components, influencing performance and consumption [16]. As stated in [18], many typical code optimizations, which target performance improvement, also reduce energy consumption, as energy is the product of runtime and power. However, it is known that some performance optimizations may sometimes increase resource usage and peak power dissipation [16]. Nevertheless, the impact of shortening the SW runtime in consumption is considerably higher than that. An experimental study is performed at the end of Section 6 to illustrate this fact.

2.2 Optimization of software runtime

Optimization of SW performance has been a key research topic for many years, in most cases measured in terms of its runtime. There are mainly two different strategies for SW optimization in the literature, depending on whether or not modifications on its semantics are allowed. They are reviewed next.

2.2.1 Optimization with software semantic modifications

Genetic programming (GP) has been used for automatically reducing SW runtime. Petke et al. propose the use of GP to optimize runtime of MiniSAT satisfiability optimization problem solver [19]. Being aware of the runtime uncertainty, authors use the number of lines executed as a measure of the SW performance. However, this is an imprecise measurement of runtime that can hardly lead to accurate solutions. The improvement obtained is mentioned to be comparable to average expert human performance. More recently, Haraldsson et al. [20] used GP to optimize runtime of ProbAbel, a widely used SW in bioinformatics for Genome Wide Association (GWA) studies. In this case, the fitness of a solution was computed as the average of the runtime of 20 executions of the program, in order to overcome the uncertainty. This approach requires roughly 330 hours of computation time to optimize the performance of the SW by 0.5%.

GP was also used to optimize SW energy consumption [21]. However, in that work the optimization is based on estimations of the consumption of the CPU, not taking into account other important components as the memory or I/O operations. Therefore, this preliminary approach cannot offer reliable results.

In general, all GP-based approaches present important handicaps. A major one is that they do not preserve the semantics of the original program, strongly limiting their application. Another limitation is that they can only be applied to small programs, because individuals encode the whole program code. In contrast, the approach proposed ensures that the semantics of the original program is preserved, and it also can be applied to large SW programs with low overhead.

2.2.2 Optimization without software semantic modifications

Software optimization approaches without introducing semantic alterations already existed in the late 90s. Bacon et al. [22] highlight that the application of transformations to improve SW performance is not trivial and an in depth understanding is required. Additionally, it presents an analysis of the dependencies between available code transformations at that time. Later in that same decade, specific transformations for reducing the redundant computations in SW programs were proposed [23]. These transformations are predefined and keep the semantic unaltered, however they are specific for each program. Therefore, these techniques were mainly used for low computational resources devices like embedded systems [24].

Lee et al. [25] propose finding a specific instruction schedule at the compiler level to reduce the execution time on Very-Long Instruction Word (VLIW) architectures, and the performance is measured in terms of the instructions transition cost.

Regarding the use of metaheuristics for automatic SW optimization, one of the most common problems tackled in the literature is the phase ordering problem, which is to find the correct order in which a predefined list of transformations must be applied to enhance SW performance [2], [3], [9]. Machine learning and data mining models are used for accelerating the selection of the best LLVM optimizations in [26] and [27], respectively. This problem is addressed in the current work simultaneously with the decision on which transformations are to be used, and how many times each one should be applied.

Another combinatorial problem studied in the literature is to find those compiler flags that optimize the SW performance. Different metaheuristics have been used to address this problem, targeting the optimization of the execution time [28], [29], as well as Bayesian optimization [30]. The latter makes use of a surrogate model in which the runtime performance of the compiled program with the selected flags is estimated from actual runtime performance data obtained with subsets of those flags. This problem is limited to decide which optimization flags should be used in the compilation of the program. In contrast, the problem proposed in this work considers the possibility of applying the same flag several times (known to be beneficial in many cases), while optimizing at the same time the order in which they should be applied.

De la Torre et al. present in [8] a sensitivity analysis of the impact of the LLVM transformations on the SW execution time. Results show a low direct influence of each transformation, but they can highly impact SW performance when applied in combination with others, what evinces the complexity of the problem at hands.

As all works mentioned in this subsection, our proposed work keeps the semantics of the original SW. However, the problem proposed in this work is more complete than those explored in the literature, in the sense that it simultaneously considers the problem of finding a sequence of transformations (including repetitions of them) and the phase ordering problem. Solving this problem will lead to better performing programs. Additionally, effective mechanisms are proposed in this work to cope with uncertainty in runtime measure-

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

ments, with the aim of building specific solutions for the SW and HW considered.

In fact, most of the works discussed in sections 2.2.1 and 2.2.2 do not take into account the runtime uncertainty. The main methods they use to cope with it are briefly discussed next. Ashouri et al. [24] use the average time of just 3 runs, a clearly low number of samples to offer statistical significance. Haraldson et al. use the mean of 20 executions of the optimized SW in [20]. This is still a low number for statistical confidence, and the approach takes as much as 600 seconds to evaluate one single solution (the original program runtime is 30 seconds, approximately). In [8], the average of 32 runs of the SW was used for the studied performed on the influence of LLVM passes on the performance of SW. Other works make use of metrics that can estimate the performance of the SW without actually executing it, as the number of instructions [24], the instructions transition cost [25], or simply the number of lines of code [19]. However, these estimations are not fully representative of runtime.

Despite the fact that SW runtime measurements present high uncertainty, as it has been discussed before, there is a gap in the literature on how different techniques can cope with it. Additionally, most works considering the system uncertainty propose underestimations of the SW runtime (e.g., using approaches as the worst case, or average values) [20], not always a valid strategy. For instance, in the case of embedded systems (where runtime uncertainty is generally due to the values of the SW input parameters), underestimating the runtime does not allow guaranteeing timing constraints for real-time performance. For this case, a probabilistic approach is proposed in [31] to estimate the runtime. This paper proposes two novel ways to efficiently cope with runtime uncertainty, based on the use of bootstrapping technique to generate confident intervals of the SW runtime from as low as 5 runtime measurements. The accuracy on the SW runtime of the proposed techniques is shown to outperform others from the literature, and they are used to guide the optimization process towards up to 63.2% runtime improvements. Next, a brief description of the main methods to deal with uncertainty in optimization problems is given.

2.3 Optimization under uncertainty

Uncertainty is a major concern in different fields like artificial vision or embedded systems where if not tackled correctly can lead to a critical system failure. Most of the solutions found in the literature propose robust systems within an appropriate operating interval [32], [33]. Finding a good trade-off between robustness and efficiency is key, especially in embedded systems [34].

According to [35] uncertainty can affect to three different aspects listed next and explained below: decision variables, fitness function and environment variables (also called dynamic optimization). For a detailed review of the existing methodologies to overcome uncertainty in optimization problems, please refer to [36].

2.3.1 Fitness function

The noise affecting the objective evaluations generates errors in performance estimations.

- **Approximated fitness function**: The use of meta-models or surrogate models for computing the fitness value often lead to errors in the approximation to the actual fitness value. In order to cope with these errors, different methodologies have been proposed: i) problem approximation [37], ii) data-driven functional approximation [38], or iii) fitness inheritance, fitness imitation, and fitness assignment [39].
- **Inherent additive noise to the fitness function with zero-mean**: different fitness values can be obtained from the same genotype due to random initizalizations of the weights and multimodality of the error function. In order to cope with it, different methodologies have been proposed: i) explicit averaging [40], ii) implicit averaging [41] or iii) modifying selection [42].
- **Interval values for fitness function**: when the noise is not uniformly distributed, additive or with zero mean, it is more appropriate to model the fitness function as an interval [35]. Different methodologies have been proposed: i) interval analysis [43], [44], ii) worst case optimization [45], iii) include the interval features into optimization objectives [46], iv) interval type-2 fuzzy controllers [47]. In order to build a confident interval, a large number of samples is needed. This can be a highly costly process, specially if fitness evaluation is computationally or time demanding, as it is the case of the problem considered in this work. This work proposes the use of bootstrapping technique to overcome this issue. It allows generating a large set of samples from a few ones at negligible computational cost.
- **Customizing the fitness function**: some works add the robustness as one of the weighted sum of objectives, and it only allows solutions within an interval [48]. In [49] constrained multiobjective optimization is applied to analyze uncertainty. Uncertainty is also considered as a risk and modeled as a fitness penalization in [50].

2.3.2 Decision variables

Perturbations or environmental parameters may change after optimization. Mainly two methodology are commonly used to faced this type of uncertainty:

- **Optimizing the expected fitness**: similar methodologies to noisy environment are applied: i) explicit averaging [51] and ii) implicit averaging [52].
- **Multiobjective approaches**: not only the fitness value but also the median and the standard deviation are considered as objectives [53]. In [54], [55] the robustness is optimized together with the fitness.

2.3.3 Environment variables

Variations of the operational conditions cause changes of the problem specifications over the time. The main approaches to deal with this problem are:

- **Generate diversity after a change**: when a variation in the environment is detected, specific actions are taken to improve diversity, as hypermutation [56].

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

- **Maintain diversity throughout the run**: preventing the convergence of the population so it can adapt easily to changes [57].
- **Memory-based approaches**: the algorithm maintains a memory with relevant information about previous generations [58].
- **Multipopulation approaches**: the algorithm works with not one but several subpopulations tackling different areas of the search space [59].

2.4 Software sustainability

Software sustainability [60] is a large topic covering a wide range of domains. It is not only related to the impact of SW on the environment, the topic covered in this work, but also on any other sustainability dimension. Additionally, we must differentiate between sustainable SW and SW for sustainability [61]. The former refers to SW that is sustainable by itself (in some sense), independently of its application, and the latter means SW that helps enhancing the sustainability of some system, process, business, etc.

Sustainability of SW can be introduced at different levels, targeting distinct aspects in the SW engineering process, as minimizing its technical debt [62], or optimizing its design [63], [64], its development process [65], [66], or its environmental impact [67], among others [68], [69]. We propose a method to automatically turn SW into environmentally sustainable SW, frequently referred as Green Software, that can reduce the consumption of computing devices, many of them being battery-based ones and/or running in a 24/7 basis. This is a highly relevant concern nowadays, given that the global consumption of IT devices was around 4-6% of global electricity use in 2020 [70], and it is expected to rise to 21% by 2030, growing by 7% per year, more than double the growing pace of global electricity demand (3% per year) [71], [72].

Green SW is achieved in this work by finding a sequence of back-end operations to modify the original SW in order to speed up its execution time. Under the assumption that HW does not change, as it is the case in this work, we can generally say that reducing the execution time of a program leads to reductions in consumption [14], [15]. Additionally, reducing the SW runtime will allow the resources to become available for other tasks sooner, or to switch into energy saving mode if idling.

3 Software Code Optimization Problem

This section introduces the Software Code Optimization Problem, that we call SCOP. It is modelled as a combinatorial optimization problem that targets the minimization of the runtime of a given SW program by means of the application of generic source code transformations.

Given a source program P and a set of generic source code transformations $T = t_1, ..., t_k$, which transform P into a semantically equivalent code P', SCOP is defined as finding a sequence of such transformations $S = [s_1, ..., s_N]$, $s_i \in T$, such that the runtime of the resulting source code P' is minimized:

$$Minimize \;\; F(P, \vec{S}) = Runtime(P') \;\; , \tag{1}$$

where P' is obtained after applying the sequence of transformations $\vec{S}$ to P:

$$P \xrightarrow{\vec{S}} P' \equiv P \xrightarrow{s_1} P_1 \xrightarrow{[s_2,...,s_N]} P' \tag{2}$$

$$\equiv P \xrightarrow{s_1} P_1 \xrightarrow{s_2} P_2 \ldots \xrightarrow{s_{N-1}} P_{N-1} \xrightarrow{s_N} P' \;\; , \tag{3}$$

where P_i is the resulting program of sequentially applying transformations s_1 to s_i on the original program P. Every s_i takes the value of the identifier of a transformation, and the sequence $\vec{S}$ defines the order in which the different transformations must be applied. This order is important because it has a direct influence on the resulting output code. In addition, it is allowed for the same transformation to be applied more than once (*i.e.*, it can happen that $s_i = s_j$, $i \neq j$) since, for most existing transformations, its repetitive application produces further changes in the source code. Additionally, the impact of a code transformation on the performance of the resulting program is difficult to predict. In previous work, a sensitivity analysis was carried out to analyze the impact of all LLVM code transformations on different benchmark programs. Results showed that most of them have very low direct impact in the performance by themselves, however this impact becomes significant (roughly one order of magnitude higher), when applied in combination with other code transformations [8].

Figure 1 illustrates a general overview of the runtime optimization process of a SW by means of code transformation. As it can be seen, the main goal of the optimization algorithm is to find the optimal sequence of code transformations among all the available ones (note that they can appear more than once) that minimizes the runtime of a specific SW program on its underlying architecture.

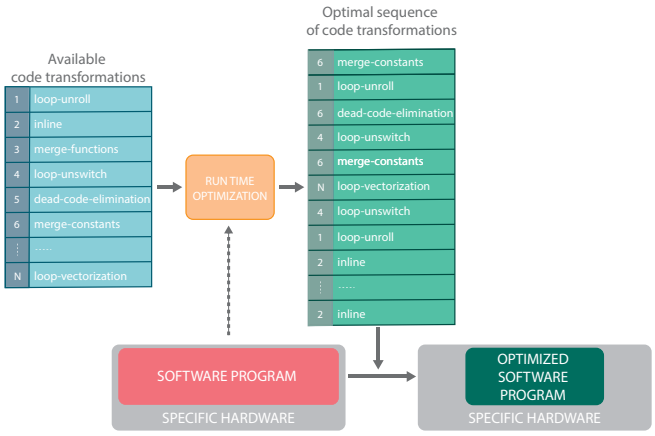


Fig. 1. Illustration of the process for optimizing the runtime of a specific software on a specific hardware by means of code transformations.

4 Methodology

SCOP makes use of a set of source code tansformations to optimize it. In this work, we make use of the transformations LLVM offers, called passes, introduced in Section 4.1. Next, how the execution times are measured in this work, as well as a study of the impact of uncertainty in the measurements is presented in Section 4.2. After that,

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

Section 4.3 describes how this uncertainty can be modeled using interval-based fitness values. Then, Section 4.4 and Section 4.5 present the approaches studied to deal with uncertainty and the optimization algorithm used, respectively. Finally, Section 4.6 gives an overall picture of the optimization process done.

4.1 LLVM compiler infrastructure

LLVM[1] is a well-known compiler infrastructure that supports several programming languages (e.g., C, C++, Objective-C, CUDA, or Julia, among others) and instruction sets (including IA-32, x86-64, ARM, Qualcomm Hexagon, MIPS, or NVIDIA). One of its outstanding features is its *Intermediate Representation* (IR), a pseudo assembly programming language that can be generated from all supported languages. This LLVM IR code is generated using the Clang compiler on the original source code. Additionally, LLVM also implements a high number of operations to analyze and optimize IR code, called passes. All passes implemented in LLVM guarantee that the semantic of the code remains unaltered.

Therefore, passes to LLVM IR codes can be used for optimizing a software program independently of its programming language or its hardware architecture. Moreover, its modularity and extensive documentation make easy the adaptation of any compiler to the LLVM infrastructure.

The optimizer allows three different passes over LLVM IR: i) **analysis**: calculates software info to be used by other passes, or for debugging; ii) **transformation**: mutates the program while keeping its functionalities; and iii) **utility**: provides useful information not included in the two previous categories. In this work, we are interested on the transformation passes, which are the ones modifying the source code. All available transformation passes are considered as candidates to be part of the solution (and they can be applied a number of times).

The **opt** command (in charge of applying the transformations) is the main optimizer and analyzer in LLVM. First, Clang compiler is used to generate the LLVM IR code from the original SW. Then `opt` applies a sequence of passes to this IR code. For example, `opt` can apply a sequence of four passes such as -tti, -inline, -loop-unroll and -verify, modifying the original LLVM IR. Finally, the compiler Clang is again used to generate the executable of the transformed LLVM IR code for the selected architecture. This must be done ensuring that the compiler does not introduce any further code modifications.

4.2 Analysis of the execution times

The technique applied to measure the execution time of a software program highly affects the results obtained, and actually there exist many different techniques to measure it. Indeed, programming languages usually provide methods for that; however, these methods perform system calls to communicate with the operating system, introducing a delay that leads to inaccurate measurements.

1. Since December 2011, "LLVM" is officially no longer an acronym and simply a brand that applies to the LLVM umbrella project. For more information, see https://www.llvm.org.

In this work we use perf, a performance analysis tool of the GNU/Linux operating system that provides accurate execution times (up to the nanosecond) and some statistical analysis of other system features. Moreover, running the same software program on the same hardware might result in different execution times because of many factors, as for example the simultaneous execution of routines of the operating system, the preemption of tasks due to user and I/O device requests [16], or any combination of adverse operation conditions [73]. They can be considered as a source of uncertainty. A common technique in the literature to deal with this non-determinism is to take the average value of a few independent executions.

In order to analyze the impact of uncertainty and check whether the probability distribution of the execution times follows a normal distribution, the same SW program used for the experiments is executed $1,000$ times on the same Rasberry Pi 3B+ device later used in the experiments. Figure 2 presents a comparative analysis between the histograms of the samples obtained from the original SW considered in this work and its optimized version (as detailed in Section 6), both executed on a Raspberry Pi 3B+ device with Raspbian 10. The impact of uncertainty causes variability in the measurement values, with the original software exhibiting a maximum difference between measurements of 0.4 seconds, while in the optimized version it is 0.5 seconds. However, it can be observed that both distributions behave similarly.

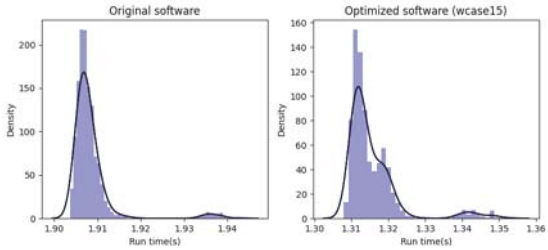


Fig. 2. Histogram of the runtime of $1,000$ independent executions of the original (left) and optimized (right) versions of the SW considered in this work on a Rasberry Pi 3B+ device.

Moreover, Figure 3 shows a comparison between the histograms of the samples from the original software executed on two Raspberry Pi 3B+ devices with different operating systems, one with Raspbian 10 and the other with Ubuntu 22.04. The distribution of the sample values differs significantly between the two operating systems, even leading to variations in the average execution time of the SW despite the use of identical HW and SW program. This clearly reflects the influence of the operating system in the uncertainty.

Additionally, an examination of the impact of HW on the SW runtime uncertainty is performed in Figure 4. For this purpose, the histograms of the samples from the original software on two different hardware devices are compared, namely a Latte Panda Alpha and an Intel Core i7-7700K, both running Ubuntu 22.04. Once again, significant differences in the distributions of the execution times are evident for the two architectures, emphasizing the influence of the hardware on the uncertainty.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

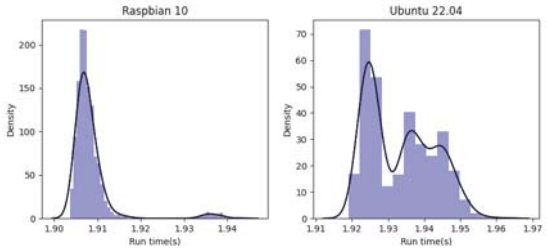


Fig. 3. Histogram of the runtime of $1,000$ independent executions of the SW considered in this work on a Rasberry Pi 3B+ device with Raspbian 10 (left) and Ubuntu 22.04 (right) operating systems.

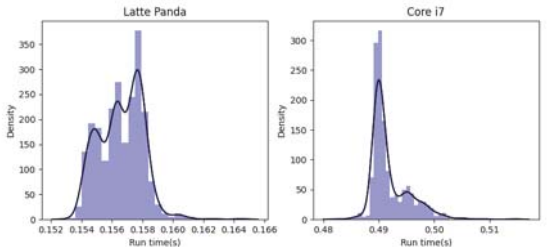


Fig. 4. Histogram of the runtime of $1,000$ independent executions of the SW considered in this work on a Latte Panda Alpha (left) and an Intel Core i7-7700K (right).

Each histogram shows that samples do not show a normal distribution. In fact, different normality tests were accomplished (i.e. Shapiro-Wilk, D'Agostino and Anderson-Darling tests) with 95% significance level, and all concluded that the samples do not follow a normal distribution.

The search process to optimize SCOP is guided by the runtime of the resulting compiled SW versions. Generating meaningful measurements of the SW runtime is of upmost importance to correctly solve the problem, because inaccurate measurements will mislead the optimization process. Therefore, runtime uncertainty must be taken into account in the evaluation of solutions.

4.3 Modeling the uncertainty with confidence intervals

As shown, using the average value of a few independent executions (as typically done in the literature) or any other technique assuming a normal distribution of the data is not the most suitable technique to deal with the uncertainty in SW execution times.

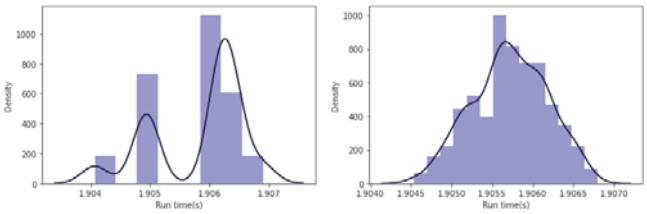
In this work, we are dealing with an optimization problem where the uncertainty is found in the fitness function. The computed runtime values can suffer from inaccuracy of the meter, operating system, the hardware of the computer, etc. Therefore, in this work we are proposing to tackle the uncertainty by using interval values for the fitness function. This technique is suitable for non-normal probability distributions and represents the fitness value without the need of estimating the real execution time (a range of potential values is used instead). The latter supposes including the uncertainty in the optimization process.

4.3.1 Bootstrap method for building intervals

The construction of the interval from a set of runtime values is not trivial. As the sample does not follow a normal distribution, the bootstrap method is applied to build a confidence interval from the percentiles of the distribution probability.

In this work, for every fitness evaluation, the proposed solution is executed five times in most cases (see Section 5 for more information) due to computational limitations. From these five executions and a statistical parameter, the bootstrap method creates a number of bootstrap subsets and builds the confidence interval. The significance level is set to 5% so that we obtain a 95% confidence interval.

The number of bootstrap subsets to create and the statistical parameter to use are decision variables with a high influence on the results. In order to find the most suitable configuration, the number of subsets was set to 500 (large enough) and both the average and the median of each of them were calculated. Figure 5 shows the median (Figure 5(a)) and average (Figure 5(b)) histograms. As it can be seen, the median histogram does not follow the central limit theorem for such low number of replicas. Thus, the median is not an appropriate statistic to build the confidence intervals. However, the average histogram behaves as a normal distribution and can thus be used for the creation of a confidence interval.



(a) Histogram of the median (b) Histogram of the average

Fig. 5. Median and average values over the 500 bootstrap subsets from the five executions for the fitness evaluation of a solution.

In terms of the number of bootstrap subsets to replicate, the main goal is to reduce the variability of the results obtained. For that, an analysis of the influence of the number of subsets created is shown in Figure 6. The center of the confident interval for the median parameter with a 5% significance is shown, ranging from only 2 bootstraps subset to up to $1,000$. As it can be seen, differences become negligible when considering around 400 subsets, where the interval generated for a specific sample is stable. Therefore, the number of subsets is set to 500, and it takes only 21.2 ms to create the interval (100 different confidence intervals were created to compute this time).

4.3.2 Comparison of confidence intervals

Literature reveals there does not exist any works facing the optimization of the execution time using the analysis of intervals. In this work, the simple partial order relation proposed in [74] for interval comparison is used. Given two arbitrary intervals A and B: $A = [a_l, a_r] = \langle a_c, a_w \rangle$ and $B = [b_L, b_R] = \langle b_c, b_w \rangle$, being x_c and x_w the centre and the width of the interval, respectively. The partial order relation is defined as shown in Equation 4.

$$A \leq^{min} B \iff \begin{cases} a_w \leq b_w & \text{if } a_c = b_c \\ a_c < b_c & \text{otherwise} \end{cases} \quad (4)$$

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

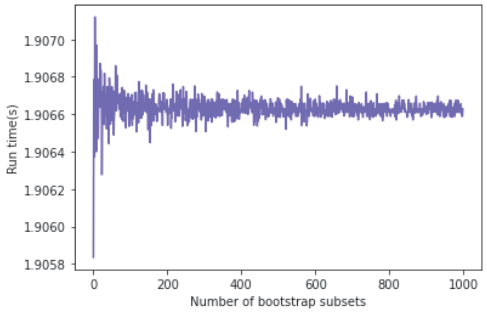


Fig. 6. Center of the interval in terms of the number of boostrap subsets.

4.4 Approaches for dealing with uncertainty

In order to make an exhaustive analysis of the uncertainty in the execution times of a software program five different techniques are implemented in this work, explained next. The first three ones consider a single value during the fitness evaluation of the optimization process, however the two last ones model the fitness as a confident interval.

- **One execution** (*1run*): the quality of individual is the value of one single execution of the software program.
- **Median** (*median*): in each fitness evaluation the potential solution is executed five times and the quality of the individual is set to median value.
- **Worst case** (*wcase*): in each fitness evaluation the potential solution is executed five times and the quality of the individual is set to the worst measured value.
- **Interval** (*interval*): in each fitness evaluation the potential solution is executed five times and the quality of the individual is set to the confidence interval obtained using the bootstrap method explained in Section 4.3.1.
- **Worst case interval from 15 executions** (*wcase15*): this is a new approach proposed in this work where in each fitness evaluation the potential solution is executed fifteen times and the quality of the individual is set to the confidence interval obtained using the bootstrap method previously explained to the five worst runs (from the 15 possible).

4.5 Cellular genetic algorithm

We propose in this work using a cellular genetic algorithm (cGA) [12] to find accurate solutions to the problem. It was chosen because it is known to generally maintain the diversity in the population for longer, with respect to other classical genetic algorithm designs (e.g., panmictic ones) [75], an important feature when dealing with complex problems because it delays premature convergence. Cellular GAs are characterized by their structured population, where individuals are arranged in a toroidal mesh, introducing the concept of distance between individuals in the population (absent in traditional –panmictic– GAs).

A pseudocode of the cGA used in this work is given in Algorithm 1. The algorithm starts by creating a population of randomly generated individuals, which are evaluated to obtain their fitness values (Line 2). Then, the algorithm starts the evolutionary loop (lines 3 to 15). Each iteration of this loop is called a generation. In each generation, the algorithm applies the evolutionary operators to evolve all individuals one by one (lines 6 to 12). First, the algorithm gets the neighbouring individuals to the current one in Line 6. Different neighborhoods can be used, and the larger the neighborhood the faster the diversity loss of individuals' information in the population [12]. The two parents are selected from the neighborhood, and they must be different (lines 7 and 8). Then, they are recombined, with some probability, to generate one offspring (Line 9). After that, its variables are mutated with some typically low probability (Line 10), and the resulting solution is evaluated (Line 11). It is then considered to be inserted in the place of the current individual in the population following a given replacement policy (Line 12). Different selection, recombination, mutation, and replacement methods can be adopted from the GA literature.

Algorithm 1 Pseudocode of a cGA

```
1:  proc cGA()
2:  pop = newPopulation();
3:  while !TerminationCondition() do
4:    for x = 1 to WIDTH do
5:      for y = 1 to HEIGHT do
6:        n_list = ComputeNeighb(pop,x,y);
7:        parent1 = Selection(n_list);
8:        parent2 = Selection(n_list);
9:        offspring = Recombination(parent1, parent2);
10:       Mutation(offspring);
11:       Evaluation(offspring);
12:       Replacement(pop,x,y,offspring);
13:     end for
14:   end for
15: end while
16: end proc cGA;
```

4.6 Description of the optimization process

The proposed SW optimization framework is composed of three main components:

- Compilation framework: provides a set of source code transformations, as well as the capability to compile the SW and generate its executable file.
- Combinatorial optimization algorithm: looks for sequences of transformations (or passes) to optimize the SW performance. Solutions of the algorithm represent sequences of a fixed number of passes. Every time a new solution is generated, its quality (called *fitness value*) needs to be evaluated by measuring the performance of the resulting SW version after being modified by the sequence of passes. Better solutions are preferred by the algorithm, and this way the search performed by the algorithm is guided towards better solutions.
- HW platform: used to evaluate the performance of the generated SW versions. Additionally, the optimization algorithm runs on this platform too, which

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

must stall while the SW performance is being evaluated, in order to avoid any interference with the measurements.

For our experiments, LLVM is selected as the compilation framework, a cGA is used as a combinatorial optimization algorithm, and four different HW platforms are considered, as described in Section 5.

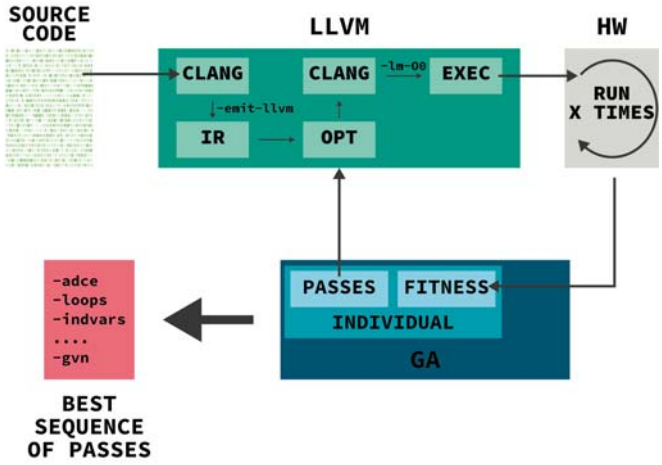


Fig. 7. Interaction of the components used in the proposed methodology.

Figure 7 graphically describes the novel optimization process proposed in this work and the interaction between all its modules. As it can be seen in it, the evaluation of any tentative solution involves several steps. First, the IR code of the original program P is generated with `clang` and the `--emit-llvm` flag. For efficiency, this step can be done only once at the beginning of the process, and the IR code of P is stored so that it can be used when required. Second, all transformations defined by the individual are applied on the IR code of P using `opt` tool from LLVM. These passes must be applied in the same order in which they appear in the solution represented by the individual (the first gene of the individual is the first pass to apply, and so on). Third, the modified program is compiled to generate an executable file. For that, we use `clang` with options `-disable-llvm-passes` and `-disable-llvm-optzns` to assemble the executable file of the resulting modified program without allowing the compiler performing any further optimizations in it. Fourth, the optimized program is executed several times, measuring its runtime. Fifth, some statistics are computed from the runtime values obtained, in order to compute accurate estimations of the SW runtime, taking into account the uncertainty in the system. The different statistical methods studied in this work are presented in Section 4.4. Sixth, the computed SW runtime estimation is then assigned to the fitness value of the solution represented by the individual.

After the GA finishes the optimization process, typically when a maximum number of fitness function evaluations is performed, the algorithm returns as an output the sequence of passes from the best solution found in the search process.

5 Experimental Setup

The problem is defined as finding the sequence of LLVM passes that minimizes the program runtime. The fact that LLVM passes are applied to IR code allows our methodology to be generic, and it can be applied without any modifications to any software code implemented in one of the languages LLVM supports. Additionally, using LLVM allows avoiding any further code modifications from the compiler, out of our control, when generating the executable. In this work, we are using LLVM version 9.0.1, providing a set of 87 generic code optimization *passes*.

We include an additional pass to those offered by LLVM that we call `NONE`. It does not perform any change in the IR code, and it is considered in order to allow the algorithm finding sequences of passes shorter than the established chromosome length.

As already mentioned, the optimization algorithm used is a cGA, and it is executed 30 times in every experiment for obtaining confident results. Its configuration parameters were experimentally chosen, and they are shown in Table 1. The population is composed of 100 individuals randomly initialized, with a square shape (10×10) and a linear 5 (or L5) neighborhood, composed by the individual itself and the ones located at its north, east, west and south positions. The chromosome of each individual is an array of 300 integers. This value is chosen because `-O3` Clang optimization flag uses a similar number of transformations. The genetic operators are: binary tournament for the parent selection, two points crossover for the recombination, random mutation, and replace if better as the replacement operator. The individual always recombine and the offspring suffers a mutation with a probability of $1/300$ that consists of changing the gene of the chromosome by another randomly chosen one (among the allowed values).

Using a chromosome of 300 variables implies a huge search space, composed by all possible combinations with repetitions of 88 elements (the 87 passes available in LLVM 9.0.1 plus `NONE`, included to mean that no operation is performed) to form sequences of 300 elements. The search space is composed by $1.80e+88$ different solutions, and a brute force method would take roughly $1.09e+81$ years to compute in the Raspberry Pi 3B+ device. Therefore, the use of approximate methods to find accurate solutions in reasonable time is justified.

For the fitness evaluation, we consider five different alternatives, as explained in Section 4.4.

Four different hardware platforms are used in this work, two based on Intel Core processor, and two others based on ARM architecture. For each kind of architecture, we analyze both a low power device, highly suitable for embedded solutions, and a high performance server. The details are given next:

- Raspberry Pi 3B+: a Raspberry Pi 3 Model B+. It is a low power device with 4 ARM cores that operate with $0.6 - 1.4$GHz frequency range and 1GB RAM memory. It runs Raspbian 10.
- Latte Panda: a Latte Panda Alpha. It is a high throughput low power device with an Intel Core M3-8100Y Dual-Core processor operating with $1.1 -$

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

TABLE 1
Configuration parameters for the experimentation.

Boostraping	
Number of subsets	500
Significance interval	5%
Statistic parameter	average
Number of samples	5
Cellular Genetic Algorithm	
Length of individuals	300
Population size	100
Population shape	10×10
Population initialization	Random
Neighborhood	L5
Parents selection	Current indiv. + Binary Tournament
Recombination	Two Points Crossover
Recombination probability	$\rho_c = 1.0$
Mutation	Random Value
Mutation probability	$\rho_0 = 1/300$
Replacement	Replace if better
Independent runs	30

3.4GHz frequency range and 8GB RAM memory. It runs Ubuntu 22.04.

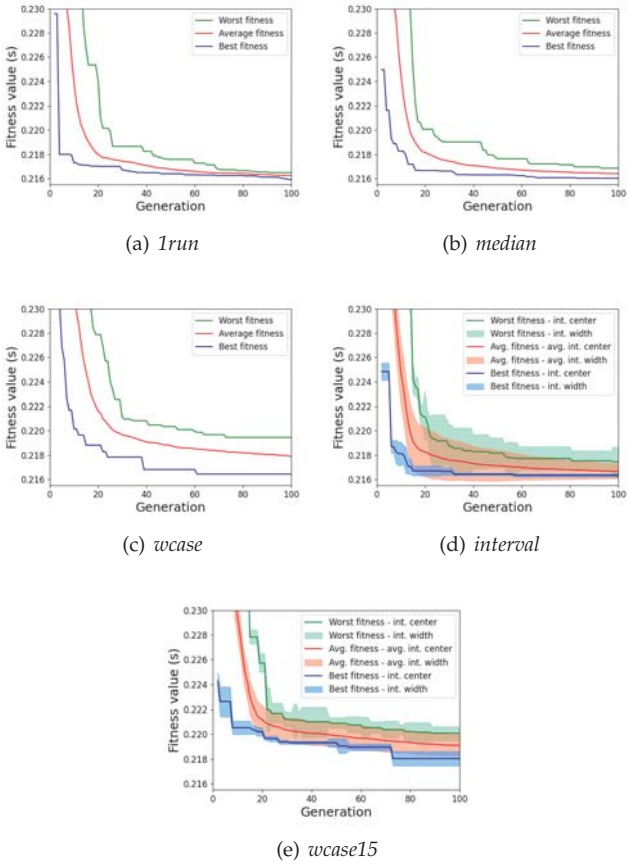
- Core i7: an Intel Core i7-7700K 4.20 GHz server with 4 cores and 16GB RAM memory. It runs Ubuntu 20.04.
- Kunpeng 920: a Huawei TaiShan 2280 V2 server with 2 Kunpeng 920-4826 CPUs of 48 cores each operating with 2.6GHz frequency. It is a massively parallel server with 96 ARM computing cores, distributed into 4 NUMA nodes. It runs Ubuntu 20.04.1 LTS.

Finally, the software program used for evaluating our optimization algorithm is a subset of the well known Polybench benchmark [76]. We selected a number of software programs from this benchmark and executed them sequentially, one after the other, in order to create a complete software, making use of a wide range of different operations, but not highly computational expensive. Some programs from the benchmark were discarded because of their long runtimes. The studied SW is composed by the following programs:

- **2mm**, two matrices multiplication.
- **3mm**, three matrices multiplication.
- **atax**, matrix transpose and vector multiplication.
- **bicg**, BiCG Sub Kernel of BiCGStab Linear Solver.
- **cholesky**, Cholesky decomposition.
- **doitgen**, multiresolution analysis kernel.
- **gemm**, matrix-multiplication $C = \alpha AB + \beta C$.
- **gemver**, vector multiplication and matrix addition.
- **gesummv**, scalar, vector and matrix multiplication.
- **mvt**, matrix vector product and transpose.
- **symm**, symmetric matrix-multiply.
- **syr2k**, symmetric rank-2k operations.
- **syrk**, symmetric rank-k operations.
- **trisolv**, triangular solver.
- **trmm**, triangular matrix-multiply.

6 Results

The cGA is applied to solve the SCOP problem with the same SW program over four different architectures using five different techniques. The evolution of the population of the cGA for the best solution found is shown in Figure 8 for the Kunpeng 920 architecture for a sample cGA run. Similar results were obtained for the other architectures.



(a) *1run* (b) *median*

(c) *wcase* (d) *interval*

(e) *wcase15*

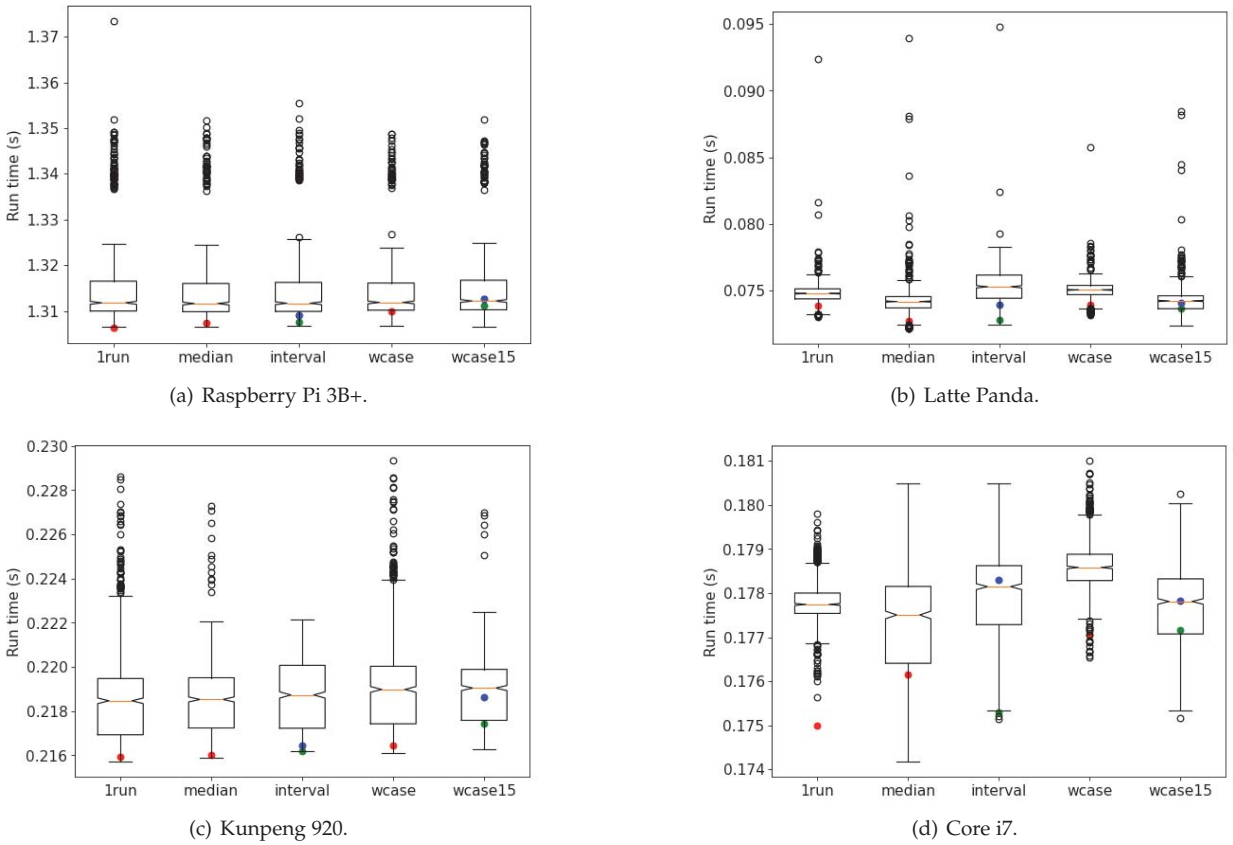
Fig. 8. Evolution of the best solution found over the 100 generations on the Kunpeng 920 architecture for every compared technique.

Figures 8(a), 8(b), and 8(c) correspond to those techniques using a single value for the fitness, while the interval based ones are presented in Figures 8(d) and 8(e). The plots represent the evolution of the best, worst, and average fitness values in the population in every generation, and the shaded area around the curves represent the interval width (only for the interval based techniques). All figures are plotted in the same scale. We can appreciate a quick improvement of the quality of the solutions in the first 20 generations, followed by a steady slower improvement. This is a typical behavior in genetic algorithms. It can be seen how most of the curves shown tend to be horizontal towards the end of the run, where the three curves get close values, meaning that the algorithm will find difficulties to further improve the fitness of the best solution found.

The two techniques based on the worst case, namely *wcase* and *wcase15*, are the ones with the highest population diversity (i.e., worst and best individuals curves are more separated) and the slowest improvement speed of the best solution. This causes that the average fitness of the population is still evolving at the end of the run, which hopefully could lead to further improvements of the best solution found in longer runs.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671



(a) Raspberry Pi 3B+.

(b) Latte Panda.

(c) Kunpeng 920.

(d) Core i7.

Fig. 9. Comparison of the performance of the best result provided by the GA, after $1,000$ runs, on the four considered architectures. Red dot is the fitness computed during the GA execution for that solution. Blue and green dots represent the fitness of the solutions for interval-based ones.

In order to compare the solutions obtained and their capabilities of dealing with uncertainty, the best solution found over the 30 independent runs of the cGA using each of the five techniques is executed in their corresponding architectures for $1,000$ times. Figure 9 shows the boxplots of the results obtained. The red dot is the fitness reported by the cGA for the optimized program, the green and blue dots representing the lower and upper values of the computed confidence interval.

As it can be observed in the Raspberry Pi 3B+ architecture (Figure 9(a)), there are no statistical differences between any of the proposed techniques in terms of the execution time of the best found optimized programs. It can clearly seen that the *1run* fitness fits worst. The best value proposed by the cGA matches the lower whisker, and it is far below the interquartile range of the boxplot. The *median*, *interval* and *wcase* estimations also fall out of the box, however the latter fits better being the best solution provided by the cGA at the bottom of the interquartile range. The approach that fits best is *wcase15* because the proposed interval is exactly in the middle of the box, meaning that the treatment of the uncertainty is appropriate.

The behaviour of the performance of the different techniques is similar for the other ARM-based architecture studied, i.e., the Kunpeng 920 server. It can be seen in Figure 9(c) that *wcase15* is the best fitting approach, and *1run* the worst one. In this case, we can observe that there are statistical differences between the two approaches using the worst case (*wcase* and *wcase15*) and the other three. However the highest difference observed is lower than 0.6 ms (0.218464563 s for *1run* and 0.2190407495 s for *wcase15*).

Regarding the Latte Panda device (Figure 9(b)), *wcase15* stands out as the best approach both in terms of solution quality estimation and runtime of the optimized program. The runtime estimations of the other fitness functions are rather poor, as observed for the previous architectures. In this device, the programs found with *1run* and *interval* techniques are significantly the worst performing ones.

Finally, the *wcase15* approach is also the best fit for the Core i7 server (see Figure 9(d)). We can observe there are statistical differences among all the proposed solutions, being the best one the *median* approach, followed by *wcase15*. However, as stated before, although statistically significant, the difference in the median value is less than 0.7 ms between the best and the worst optimized programs (0.1804904635 s for *median* approach and 0.1811781475 s for *wcase*). Regarding the estimations of the program runtime performed by the different techniques, *wcase15* is, again, the most accurate one, having both interval limits within the box. The other techniques generally perform bad estimations, with the exception of *interval*, which is able to place its upper limit interval value in the box.

In general, the novel approach proposed in this work, *wcase15*, is the best performing one in any of the four studied architectures. The confidence interval used as the quality indicator of the individual is allocated in the interquartile

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

range of the box plots in all cases. Thus showing that the treatment of uncertainty is highly suitable. On the contrary, the *1run* approach (that ignores the uncertainty) shows the worst behaviour. As it can be seen, the proposed solution is always in the lowest values of the box plots. This mean that the optimization algorithm is overestimating the execution time of the program. One possible reason for this behaviour is that the cGA probably executed one individual (not necessarily being the best possible sequence of passes) over extreme favorable conditions (no other programs executing) and better solutions were not executed in such a favorable environment being thus lost in the optimization process.

Thus, in order to validate the representativeness of these results, an additional study is carried out. The best solution found in each of the 30 independent executions of the genetic algorithm is executed $1,000$ times and the number of times that the fitness is within the interquartile range of the $1,000$ samples is considered. This study is accomplished for the four architectures, and the results are given in Table 2. The study confirms that *wcase15* is clearly the most accurate technique, followed by *wcase*. The worst ones, are *1run*, that does not consider uncertainty, and *median*, commonly used in the literature. This result highlights that the proposed approach, *wcase15*, successfully handles the uncertainty in the execution times of a software program, while the others are generally not able to bear it.

TABLE 2
Percentage of times that the fitness estimation is representative of the real program time performance, for the 30 solutions found by the cGA in every architecture.

	Rasp. Pi3B+	Latte Panda	Kunpeng 920	Core i7
1run	0.00%	40.00%	0.00%	70.00%
median	0.00%	0.00%	0.00%	53.33%
interval	30.00%	0.00%	16.67%	63.33%
wcase	96.67%	73.33%	60.00%	93.33%
wcase15	100.00%	93.33%	100.00%	100.00%

In order to quantify the improvement achieved by the proposed techniques on the performance of the optimized SW, Figure 10 compares them against the runtime of the SW program without any optimization (`-O0` flag) and compiled with the state of the art optimization `-O3` flag for the Core i7 architecture.

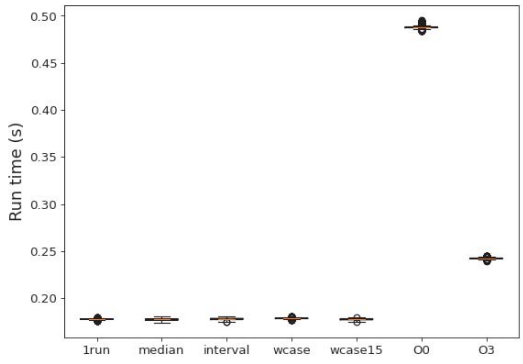


Fig. 10. Comparison of the optimized results against the software compiled with `-O0` (i.e., no optimizations performed) and `-O3` flags. Sample case of Core i7 architecture.

This comparison was performed on all devices and similar results were obtained for all of them. As it can be seen, the optimization process proposed in this work clearly outperforms the performance of the program itself and the one optimized with `-O3` Clang compilation flag with statistical significance. The fastest SW in this architecture was found using *median* technique, and the improvement obtained over the non-optimized software program is 63.20%, and it is 26.34% faster than the program compiled with `-O3` flag. Moreover, the improvements of the *wcase15* over the `-O0` and `-O3` compilation flags are 63.15% and 26.25%, respectively (less than 0.1% lower than the *median*).

Finally, a summary of the improvement of the optimization process using the *wcase15* technique in any of the studied architectures over the `-O0` and `-O3` compilation flags is presented in Table 3. It can be seen that optimizing the SW with the proposed solution leads to 19.13% faster runtimes, in average, for the studied architectures, with respect to the `-O3` optimization flag.

TABLE 3
Performace improvement of the *wcase15* technique over the `-O0` and `-O3` compilation flags for the four different architectures.

Architectures	-O0	-O3
Raspberry Pi 3B+	31.2%	15.2%
Latte Panda	42.6%	21.1%
Kunpeng 920	28.1%	13.9%
Core i7	63.2%	26.3%

Impact of SW runtime on its energy consumption.

In order to analyze the influence of the runtime of a specific SW on its sustainability, the energy consumption of different optimized SW programs and their original version are compared. Particularly, Figure 11 shows the energy consumption (in mW) of the original studied SW program (compiled with `-O0` flag), as well as its consumption after being optimized with the five proposed techniques on the Raspberry Pi 3B+ architecture. A tailored device (patent pending) based on Texas Instruments INA256 component was used to measure the input current to the device. The boxplots represent the results obtained after measuring $1,000$ times each SW version.

As it can be seen, the energy consumption of any of the proposed optimization techniques is around 30% lower than the original program, highlighting that the impact of the minimization of the runtime on the sustainability of the software is straightforward. These results support that a greener version of a SW running on a specific HW can be created by applying a tailored sequence of code transformations, even under uncertain environments.

Additionally, Figure 12 presents the instant current consumption signals ($1,650$ samples per second are taken in the experiment) of both the original SW (compiled with -O0 flag) and its optimized version, computed using the proposed wcase15 technique. As depicted, the optimized SW exhibits a considerably shorter signal length due to its reduced execution time. Moreover, amperage values over time are slightly higher in the optimized SW (roughly 5%), caused by the more intensive usage of the hardware pipeline [16] that leads to the decreased execution time.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

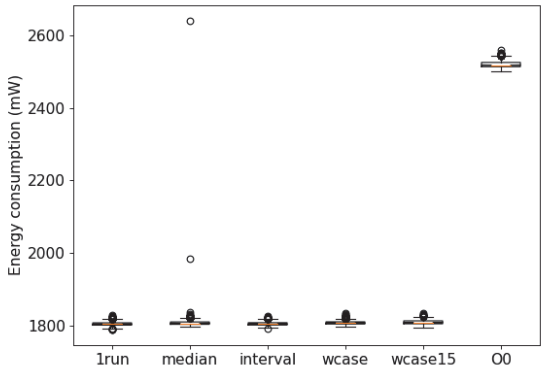


Fig. 11. Comparison of the energy consumption of the optimized results against the software compiled with `-O0` (i.e., no optimizations performed). Sample case of Raspberry Pi 3B+ architecture.

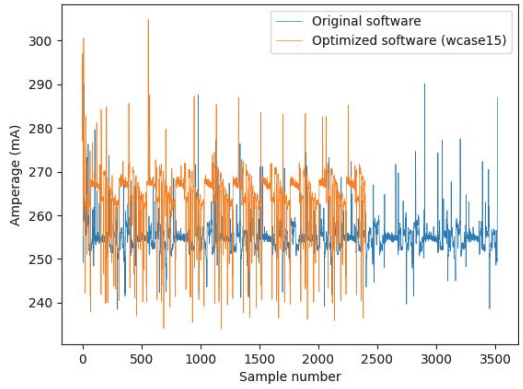
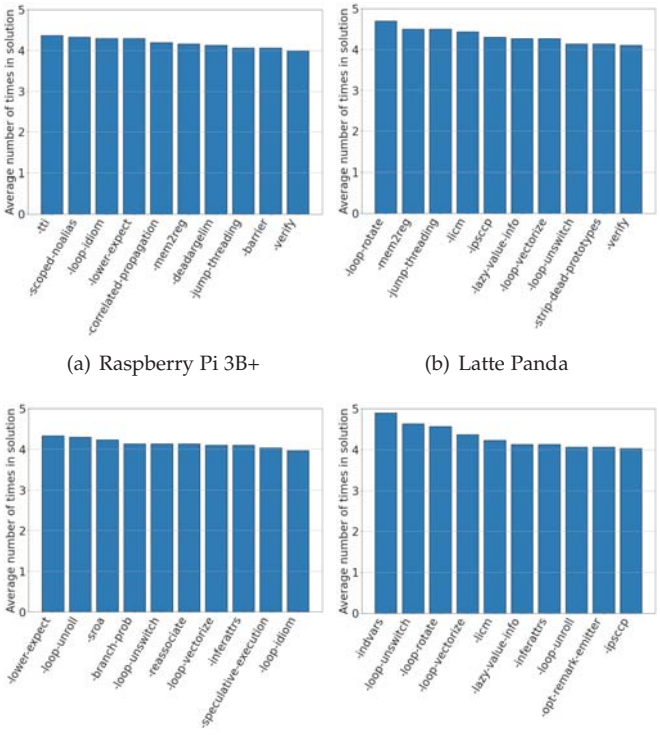


Fig. 12. Comparison of the consumption signal of the original software (compiled with -O0 flag) against the optimized version (wcase15).

However, despite the marginal increase in amperage, the significant reduction in execution time contributes to decreasing energy consumption of the optimized SW by a meaningful 39.3%: 1809.25 mWs against 2520.10 mWs of the original SW (these values are averaged after 100 independent runs). These findings confirm that the proposed runtime optimization technique enhances SW sustainability, endorsing the fact that notable reductions in energy consumption can be obtained by optimizing SW runtime.

7 Discussion on the Results

In this section, we focus on the solutions obtained with *wcase15*, the best performing fitness function among the studied ones, originally proposed in this work. We analyze in Section 7.1 the most and least used passes in the solutions reported by the GA in the 30 independent runs performed. Then, we show in Section 7.2 the overall best solutions found for every architecture, comparing them, and also against the sequence of passes applied by `-O3` Clang optimization compilation flag, which represents the state of the art.



(a) Raspberry Pi 3B+ (b) Latte Panda

(c) Kunpeng 920 (d) Core i7

Fig. 13. Ten most used passes in the solutions obtained in the 30 independent runs of the GA using the *wcase15* fitness function.

7.1 Usefulness of passes on the different architectures

We investigate in this section the passes resulting the most and least frequently used in the different solutions found by the cGA, for all architectures used.

First, we observed how often every pass appears at least once in the 30 solutions reported by the cGA, for every architecture. An interesting finding is the usefulness of the considered passes, given that most of them are used in over 85% of the obtained solutions. Besides some sparse exceptions, `-instcombine` and `-gvn` passes are clearly the least used operators. The former was only used in Kunpeng 920 server (and only 2 times out of 30), while the latter was present in only 0.06% of the solutions. These two passes perform operations in instructions to either eliminate redundant ones (`-gvn`) or to replace them with fewer and simpler ones (`-instcombine`). Together with these two passes, the `NONE` operator is also rarely used: it was found in only 15% of the solutions. Therefore, the choice of the algorithm is making use of the maximum allowed number of passes, in general.

We also investigated the number of times every pass is applied in the solutions. Figure 13 shows the average number of times the ten most frequently used passes are applied in a solution on the four architectures. In general, it can be seen how every shown pass is applied, on average, more than four times in every solution. The two architectures sharing the highest number of passes in Figure 13 are the two Intel-based ones, namely Core i7 and Latte Panda. They share six passes as the most useful ones: `-loop-rotate`, `-loop-unswitch`, `-loop-vectorize`, `-licm`, `-ipsccp`,

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

and `-lazy-value-info`. The four first ones are loop optimization/parallelization operations, while the other two ones target the removal of constants, variables, and dead code sections.

The other two architectures sharing the highest number of passes among the most used ones are the HPC architectures, Kunpeng 920 and Core i7, for which passes `-loop-unswitch`, `-loop-unroll`, `-loop-vectorize`, and `-inferattrs` are among the most useful ones. The two former passes simplify the loops and reduce their number of jumps (the first one reduces the number of iterations of the loop, and the second one takes conditional branches out of the loop). The resulting simpler loops hopefully improve the instructions' pipeline behavior, and they allow the application of further optimizations too, as it is the case of `-loop-vectorize`, which takes profit from the vectorial parallel operations in these architectures for faster executions.

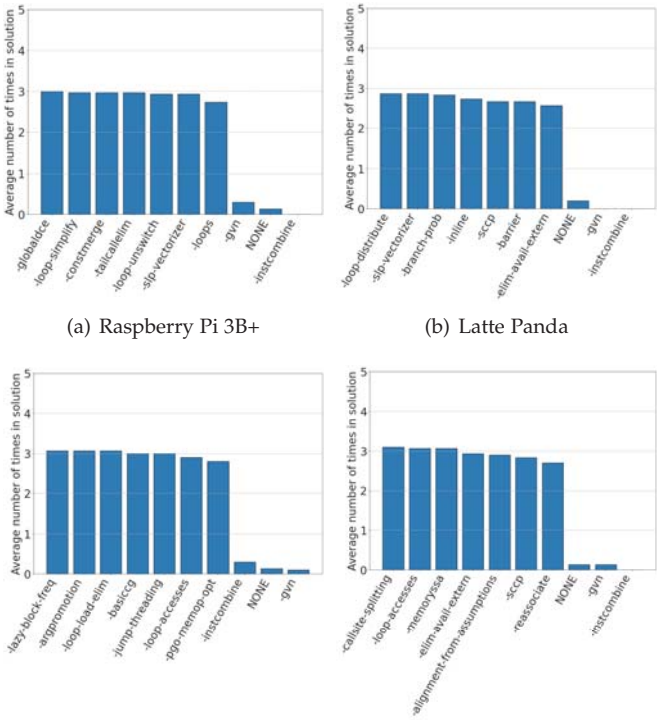
The two low-power architectures studied have three passes in common among the ten most useful ones, namely `-mem2reg`, `-jump-threading`, performing operations to maximize the use of registers and reduce the number of jumps, respectively, and `-verify`, which simply verifies the LLVM IR code is correct, and makes effective the operations performed by previous passes.

Highly different HW platforms as Core i7 and Raspberry Pi 3B+ do not share any useful pass, while Kunpeng 920 and Latte Panda (a high throughput low power device) have only two common useful passes (`-loop-vectorize` and `-loop-unswitch`).

If we now pay attention to the passes that are in the list of 10 most used ones for all fitness functions used, we find `-loop-unswitch` and `-loop-vectorize` for Kunpeng 920; `-loop-idiom` for Raspberry Pi 3B+; `-loop-rotate`, `-loop-unroll`, and `-loop-unswitch` for Core i7; and `-loop-rotate` for Latte Panda. All of them are optimizations making different transformations on loops.

We extended the study to the analysis of the 10 least used passes for every architecture. It can be seen in Figure 14 that the passes in the list are applied in average around three times in every solution. There is not a single pass in common in the lists of the 10 least used ones in any architecture (with very little exceptions), besides the previously mentioned `-gvn`, `-instcombine` and `NONE`, and this result is extensive to the results found with all fitness functions. Additionally, it is an interesting finding that some passes in the list of 10 most useful ones for some architectures are in the list of 10 least used ones for others, evidencing the need of the application of *ad hoc* sequences of optimizations on different HW architectures. Furthermore, the small difference with the number of times the most used ones appear in the solutions (between 4 and 5 times) elucidates how all passes are important to optimize the considered program.

In summary, the results obtained show that most studied passes are highly important in the optimization of SW, and it is difficult to draw conclusions on the most/least useful ones, because it depends on the HW architecture and on the characteristics of the SW code itself. This conclusion evinces the need of highly advanced adaptive compilers, able to take decisions on how many times every pass should be applied, as well as the order in which all transformations are made, according to the particular characteristics of the SW itself, as well as of the target architecture.



(a) Raspberry Pi 3B+ (b) Latte Panda

(c) Kunpeng 920 (d) Core i7

Fig. 14. Ten least used passes in the solutions obtained in the 30 independent runs of the GA using the *wcase15* fitness function.

7.2 Analyzing the best obtained solutions

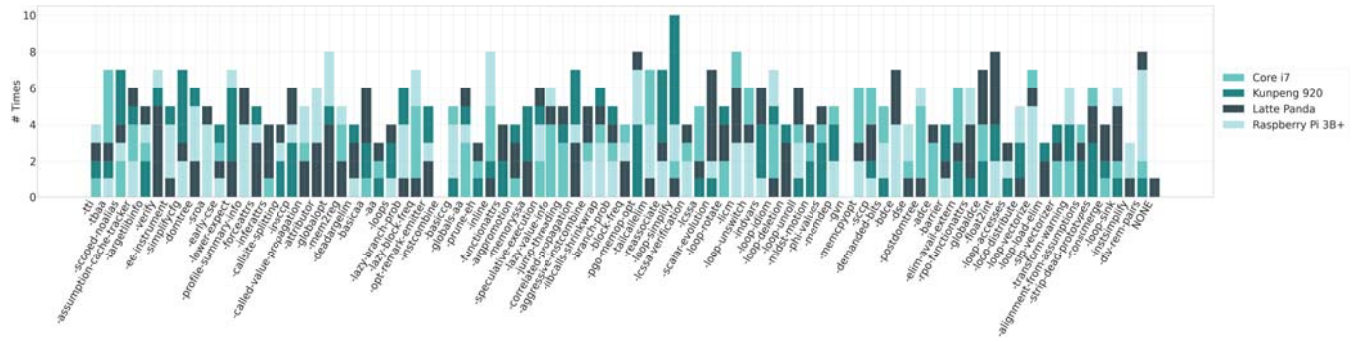
Figure 15(a) shows the number of times every pass has been applied in the best solution found for every architecture using the *wcase15* fitness function. It can be seen how the most used passes differ for the different architectures. Indeed, the three most used passes for the four platforms are different. Additionally, we can find some passes that are commonly used in some architectures and among the least used ones in others. For instance, one of the most used passes in Raspberry Pi 3B+ (`-inline`) is among the least used ones for the other low-power device studied, i.e., Latte Panda. Another similar example in the case of the high-performance architectures is `-loop-simplify`, used ten times for Kunpeng 920 (the most used one) and only two times for Core i7.

It also calls our attention that all architectures discard six passes except Kunpeng 920 (only three), and they are different in all solutions (only `-gvn` and `-instcombine` are unused passes in all solutions). The other discarded passes are `-memoryssa`, `-reassociate`, `-rpo-functionattrs` and `NONE` for Core i7, `-strip-dead-prototypes`, `-globaldce`, `-inferattrs` and `-phi-values` for Raspberry Pi 3B+, and `-demanded-bits`, `-basiccg`, `-transform-warning` and `-memdep` for Latte Panda.

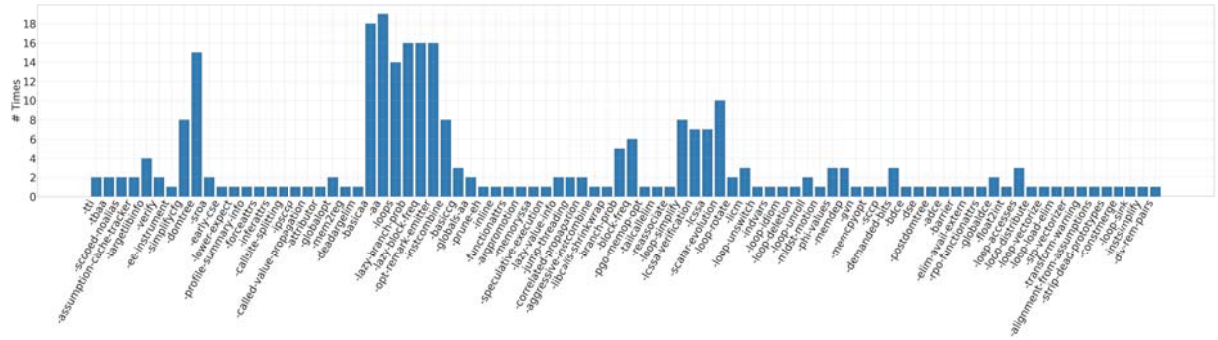
It can be surprising that pass `-instcombine`, which performs several different optimizations to reduce the number of instructions in the code (e.g., transform comparison

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671



(a) The passes used in the best solution found with *wcase15* for all considered architectures.



(b) The passes used by `-O3` compilation flag.

Fig. 15. Number of times every pass is applied by the best solutions found, and by `-O3` Clang compilation optimization flag.

with boolean values into logical operations, comparison instructions like $<$, $>$, $\leq$, $\geq$ into $=$ or $\neq$, or multiplications by power of two into shifts, among others) is not used in any of our solutions. In fact, this pass is used eight times by Clang `-O3` compilation flag, as shown in Figure 15(b).

As it can be seen in Figure 15(b), the passes applied the highest number of times by `-O3` are `-aa` adn `-basicaa`, used 19 and 18 times, respectively, while our solutions use them 2.25 and 4 times in average. The three second most used passes in `-O3` are `-lazy-branch-prob`, `-lazy-block-freq` and `-opt-remark-emitter`, applied 16 times. However, these passes are used, in average, 3.75, 4.75 and 3.75 times, respectively, in our solutions.

To end this section, we revise the most used passes in the best solutions found for every architecture (those in Figure 15(a)) that, in contrast, are barely used in `-O3` sequence (Figure 15(b)). For Kunpeng 920 architecture, passes `-scoped-noalias`, `-reassociate` and `-correlated-propagation` are among the most used ones, i.e., 7 times, but rarely used in `-O3` sequence: 2, 1, and 2 times respectively. In the case of Core i7, we find important differences on all most used passes: `-licm`, `-tbaa`, `-loop-vectorize`, `-tailcallelim` and `-pgo-memo-opt` appear in our solution 8 times the former and 7 times the rest, while in `-O3` they are applied 3 and 2 times the first two ones, and just once the others. For Raspberry Pi 3B+, we find that `-globalopt` and `-inline` were used 8 times in our solution and 2 and 1 times in `-O3`, and `-loop-idiom`, `-lower-expect`, `-div-rem-pairs`, `-prog-memo-opt` and `-verify` appear 7 times in our solution and only once in `-O3` (except the last one, which is applied twice). In the case of Latte Panda device, we also found a different usage of the passes, where `-licm`, `-loop-vectorize` and `-float2int` are applied 8 times in our solutions and one in `-O3`, and `-bdce` and `-globalbce` appear 7 times in our solution and 1 and 2 in `-O3`, respectively.

In summary, we appreciate that the best optimization sequences found by the GA for every architecture highly differ among them, and they all present at the same time important differences with respect to the sequence applied by `-O3` Clang compilation flag. The results presented here are obviously biased by the software and architectures selected, but they evince the need of *ad hoc* optimizations in software compilation process.

8 Conclusions and Further Work

In this work, we have modeled for the first time the software code optimization problem (SCOP for short). It is a combinatorial problem that aims at finding the optimal sequence of code transformations minimizing the execution time of a software program, and taking into account the intrinsic uncertainty in the runtime measurements. The proposed methodology leads towards the automatic generation of more sustainable SW, in the sense that it makes a more efficient use of the available resources.

We propose solving the problem using a cellular genetic algorithm. In order to give a robust solution to the problem, a new approach that creates a confidence interval from the five worst executions out of fifteen (*wcase15*) is proposed as a quality indicator for the optimization process, and its performance is compared to other four different techniques.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

Moreover, extensive analysis have been accomplished on four hardware architectures with different processors (ARM and Intel) and purposes (low and high performance computing): Raspberry Pi 3B+, Kunpeng 920, Latte Panda alpha, and Intel Core i7.

Results show that the sequences of code transformations found using *wcase15* generate the best performing optimized programs in all architectures, robust to the uncertainty of the execution times. Moreover, an in depth analysis of the consistency of the technique shows that the confidence interval computed by *wcase15* accurately estimates the runtime of the program with a 100% accuracy, compared to the 60% or less achieved by the other considered methods. The reported solutions outperform the execution time of the original software program at least over a 28%, and roughly by 14% with respect to the state of the art `-O3` Clang compilation flag in the studied architectures.

Important differences were found in the sequences of passes obtained for the four considered architectures, as well as with `-O3` sequence. These results evince the need of new generation smart compilers that perform *ad hoc* optimization sequences in terms of the features of the SW program and the HW where it will be executed. The analysis of the solutions found show how most of the passes LLVM offers are important for the program optimization, but the most important passes differ from one architecture to the other.

As future work, it will be interesting to research the most appropriate optimization algorithms to solve the new proposed SCOP problem, offering better performance or even similar ones with shorter execution times. We will work on new definitions of the problem too, targeting other features that could further enhance the energy efficiency of the program, such as reducing its memory usage. Moreover, we are working towards extending our work with more programs having different features, and finding recommendations of the passes that can be more useful for the given program and hardware architecture, according to their characteristics. That will be the cornerstone for a next generation of smart compilers that perform *ad hoc* optimizations on the SW for optimal performance.

Acknowledgments

This work was supported by the Iunta de Andalucía and ERDF (GENIUS – P18-2399), the Ministerio de Ciencia, Innovación y Universidades y AEI /10.13039/501100011033 – PID2019-109465RB-I00) and ERDF (OPTIMALE – FEDER-UCA18-108393). This publication is part of TED2021-131880B-I00 project (eFracWare), funded MCIN/AEI/10.13039/501100011033 and the European Union "NextGenerationEU"/PRTR and project eMob (PID2022-137858OB-I00) funded by Spanish MCIN, the AEI and the ERDF on MCIN/AEI/10.13039/501100011033/FEDER, UE. This work was partially supported by the Spanish Ministry of Science and Innovation and ERDF (RED2018-102472-T). J.C. de la Torre and J.M. Aragón-Jurado would like to acknowledge the Spanish Ministerio de Ciencia, Innovación y Universidades for the support through FPU17/00563 and FPU21/02026 grants, respectively. J.M. Aragón-Jurado acknowledges the grant from Plan Propio - UCA 2022-2023. B. Dorronsoro and P. Ruiz acknowledge "ayuda de recualificación" funded by Ministerio de Universidades and the European Union-NextGenerationEU.

References

[1] G. Jin, J. Mellor-Crummey, and A. Qasem, "Improving performance with integrated program transformations," Tech. Rep., 2004.

[2] P. A. Kulkarni, S. R. Hines, J. D. H. David B. Whalley, J. W. Davidson, and D. L. Jones, "Fast and efficient searches for effective optimization-phase sequences," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 2, pp. 165–198, 2005.

[3] P. Kulkarni, D. Whalley, G. Tyson, and J. Davidson, "Exhaustive optimization phase order space exploration," in *Int. Symposium on Code Generation and Optimization (CGO'06)*, 2006, pp. 13 pp.–318.

[4] R. Pinkers, P. Knijnenburg, M. Haneda, and H. Wijshoff, "Statistical selection of compiler options," *Proceedings - IEEE Computer Society's Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS*, pp. 494–501, 2004.

[5] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *Proceedings of the 15th annual workshop on Microprogramming*. IEEE Press, 1982, pp. 125–133.

[6] D. Whitfield and M. L. Soffa, "An approach to ordering optimizing transformations," in *Proceedings of the second ACM SIGPLAN symposium on Principles & Practice of Parallel Programming*. ACM Press, 1990, pp. 137–146.

[7] LLVM, "The llvm compiler infrastructure," https://llvm.org/.

[8] J. C. de la Torre, P. Ruiz, B. Dorronsoro, and P. L. Galindo, "Analyzing the influence of LLVM code optimization passes on software performance," in *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Applications*. Springer International Publishing, 2018, pp. 272–283.

[9] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for reduced code space using genetic algorithms," in *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 1999, pp. 1–9.

[10] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 International Symposium on Memory Management*. New York, NY, USA: Association for Computing Machinery, 2013, pp. 63—-74.

[11] C. Moreno and S. Fischmeister, "Accurate measurement of small execution times—getting around measurement errors," *IEEE Embedded Systems Letters*, vol. 9, no. 1, pp. 17–20, 2017.

[12] E. Alba and B. Dorronsoro, *Cellular Genetic Algorithms*, ser. Operations Research/Computer Science Interfaces. Springer-Verlag Heidelberg, 2008, vol. 42.

[13] L. Lannelongue, J. Grealey, A. Bateman, and M. Inouye, "Ten simple rules to make your computing more environmentally sustainable," *PLOS Computational Biology*, vol. 17, no. 9, p. e1009324, 2021.

[14] L. F. W. Anthony, B. Kanding, and R. Selvan, "Carbontracker: Tracking and predicting the carbon footprint of training deep learning models," ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems, July 2020, arXiv:2007.03051.

[15] L. Lannelongue, J. Grealey, and M. Inouye, "Green algorithms: Quantifying the carbon footprint of computation," *Advanced Science*, vol. 8, no. 12, p. 2100707, 2021.

[16] V. Venkatachalam and M. Franz, "Power reduction techniques for microprocessor systems," *ACM Computing Surveys*, no. 37, pp. 195–237, 2005.

[17] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Communication Surveys & Tutorials*, vol. 18, no. 1, pp. 732–794, 2016.

[18] S. Reda, *Power Modeling and Characterization of Computing Devices: A Survey*, ser. Foundations and Trends in Electronic Design Automation. NOW, 2012, vol. 6, no. 2.

[19] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 137–149.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

[20] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, A. V. Smith, and V. Gudnason, "Genetic improvement of runtime and its fitness landscape in a bioinformatics application," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. New York, NY, USA: Association for Computing Machinery, 2017, p. 1521–1528.

[21] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2015, pp. 1327–1334.

[22] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys (CSUR)*, vol. 26, no. 4, pp. 345–420, 1994.

[23] M. Hall and J. P. McNamee, "Improving software performance with automatic memoization," *Johns Hopkins APL Technical Digest*, vol. 18, no. 2, p. 255, 1997.

[24] J. Villarreal, D. Suresh, G. Stitt, F. Vahid, and W. Najjar, "Improving software performance with configurable logic," *Design Automation for Embedded Systems*, vol. 7, no. 4, pp. 325–339, 2002.

[25] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai, "Compiler optimization on VLIW instruction scheduling for low power," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 2, pp. 252–268, 2003.

[26] A. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, J. Cavazos, A. Bignoli, G. Palermo, and C. Silvano, "MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 29, 2017.

[27] L. Martins, R. Nobre, J. Cardoso, A. Delbem, and E. Marques, "Clustering-Based Selection for the Exploration of Compiler Optimization Sequences," *ACM Transactions on Architecture and Code Optimization*, vol. 13, no. 1, pp. 1–28, 2016.

[28] T. Sandran, N. Zakaria, and A. Pal, "An optimized tuning of genetic algorithm parameters in compiler flag selection based on compilation and execution duration," in *Proceedings of the International Conference on Soft Computing for Problem Solving*. Springer, 2011, pp. 599–610.

[29] U. Garciarena and R. Santana, "Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2016, pp. 1159–1166.

[30] J. Chen, N. Xu, P. Chen, and H. Zhang, "Efficient compiler autotuning via bayesian optimization," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1198–1209.

[31] M. Qiu, H. Li, and E. H.-M. Sha, "Heterogeneous real-time embedded software optimization considering hardware platform," in *Proceedings of the 2009 ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2009, pp. 1637—-1641.

[32] T. Nakajima, M. Sugaya, and S. Oikawa, "Operating systems for building robust embedded systems," in *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2005, pp. 211–218.

[33] M. Brezak, I. Petrović, and E. Ivanjko, "Robust and accurate global vision system for real time tracking of multiple mobile robots," *Robotics and Autonomous Systems*, vol. 56, pp. 213 – 230, 2008.

[34] L. Lacassagne, A. Manzanera, and A. Dupret, "Motion detection: Fast and robust algorithms for embedded systems," in *2009 16th IEEE International Conference on Image Processing (ICIP)*, 2009, pp. 3265–3268.

[35] Z. He, G. G. Yen, and J. Lv, "Evolutionary multiobjective optimization with robustness enhancement," *IEEE Transactions on Evolutionary Computation*, vol. 24, no. 3, pp. 494–507, 2019.

[36] Y. Jin and J. Branke, "Evolutionary optimization in uncertain environments-a survey," *IEEE Transactions on evolutionary computation*, vol. 9, no. 3, pp. 303–317, 2005.

[37] J. J. Grefenstette and J. M. Fitzpatrick, "Genetic search with approximate function evaluations," in *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, 1985, pp. 112–120.

[38] M. Bhattacharya and G. Lu, "DAFHEA: a dynamic approximate fitness-based hybrid EA for optimisation problems," in *The 2003 Congress on Evolutionary Computation*, vol. 3, 2003, pp. 1879–1886 Vol.3.

[39] M. Reyes-Sierra and C. Coello, "A study of fitness inheritance and approximation techniques for multi-objective particle swarm optimization," in *2005 IEEE Congress on Evolutionary Computation*, vol. 1, 2005, pp. 65–72.

[40] A. D. Pietro, R. L. While, and L. Barone, "Applying evolutionary algorithms to problems with noisy, time-consuming fitness functions," *Proceedings of the 2004 Congress on Evolutionary Computation*, vol. 2, pp. 1254–1261, 2004.

[41] L. M. Rattray and J. Shapiro, "Noisy fitness evaluation in genetic algorithms and the dynamics of learning," in *4th Workshop on Foundations of Genetic Algorithms*, 1997, p. 117–139.

[42] G. Rudolph, "A partial order approach to noisy fitness functions," in *Proceedings of the 2001 Congress on Evolutionary Computation*, vol. 1, 2001, pp. 318–325.

[43] X. Sun, L. Zhao, P. Zhang, L. Bao, and Y. Chen, "Enhanced NSGA-II with evolving directions prediction for interval multi-objective optimization," *Swarm and Evolutionary Computation*, vol. 49, pp. 124–133, 2019.

[44] S. Chanas and D. Kuchta, "Multiobjective programming in optimization of interval objective functions — a generalized approach," *European Journal of Operational Research*, vol. 94, no. 3, pp. 594–598, 1996.

[45] N. Wang, X. Zhang, and Y. Yang, "A hybrid genetic algorithm for constrained multi-objective optimization under uncertainty and target matching problems," *Applied Soft Computing*, vol. 13, no. 8, pp. 3636–3645, 2013.

[46] H. Ishibuchi and H. Tanaka, "Multiobjective programming in optimization of the interval objective function," *European Journal of Operational Research*, vol. 48, no. 2, pp. 219–225, 1990.

[47] P. Rakshit and A. Konar, "Extending multi-objective differential evolution for optimization in presence of noise," *Information Sciences*, vol. 305, pp. 56–76, 2015.

[48] G. Karer and I. Škrjanc, "Interval-model-based global optimization framework for robust stability and performance of pid controllers," *Appl. Soft Comput.*, vol. 40, no. C, p. 526–543, mar 2016.

[49] P. Ruiz, B. Dorronsoro, E.-G. Talbi, and P. Bouvry, "Finding a robust configuration for the aedb information dissemination protocol for mobile ad hoc networks," *Applied Soft Computing*, vol. 32, pp. 494–508, 2015.

[50] R. C. Goodfellow and R. Dimitrakopoulos, "Global optimization of open pit mining complexes with uncertainty," *Applied Soft Computing*, vol. 40, pp. 292–304, 2016.

[51] J. Branke, "Creating robust solutions by means of evolutionary algorithms," in *Parallel Problem Solving from Nature — PPSN V*, A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 119–128.

[52] S. Tsutsui and A. Ghosh, "Genetic algorithms with a robust solution searching scheme," *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 3, pp. 201–208, 1997.

[53] T. Ray, "Constrained robust optimal design using a multiobjective evolutionary algorithm," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 1, 2002, pp. 419–424 vol.1.

[54] Y. Jin and B. Sendhoff, "Trade-off between performance and robustness: An evolutionary multiobjective approach," in *Evolutionary Multi-Criterion Optimization*, C. M. Fonseca, P. J. Fleming, E. Zitzler, L. Thiele, and K. Deb, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 237–251.

[55] B. Dorronsoro, P. Bouvry, J. A. Cañero, A. A. Maciejewski, and H. J. Siegel, "Multi-objective robust static mapping of independent tasks on grids," in *IEEE Congress on Evolutionary Computation*, 2010, pp. 1–8.

[56] H. G. Cobb, "An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments," Naval Research Laboratory, Washington, USA, Tech. Rep. AIC-90-001, 1990, accessed August 2022. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA22915

[57] J. J. Grefenstette, "Genetic algorithms for changing environments," in *Parallel Problem Solving from Nature — PPSN*, 1992, p. 137–144.

[58] D. E. Goldberg and R. E. Smith, "Nonstationary function optimization using genetic algorithm with dominance and diploidy," in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. USA: L. Erlbaum Associates Inc., 1987, p. 59–68.

[59] R. K. Ursem, "Multinational GAs: Multimodal optimization techniques in dynamic environments," in *Proceedings of the 2nd Annual Conference on Genetic and Evolutionary Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, p. 19–26.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Sustainable Computing. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSUSC.2023.3330671

[60] C. Calero, M. A. Moraga, and M. Piattini, *Software Sustainability*. Springer Cham, 2021.
[61] C. Calero, M. A. Moraga, and F. García, "Software, sustainability, and UN sustainable development goals," *IT Professional*, vol. 24, no. 1, pp. 41–48, 2022.
[62] C. Lilienthal, *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. Rocky Nook, 2019.
[63] C. Bunse and S. Stiemer, "On the energy consumption of design patterns," *Softwaretechnik-Trends*, vol. 33, no. 2, 2013.
[64] D. Feitosa, R. Alders, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, "Investigating the effect of design patterns on energy consumption," *Journal of Software: Evolution and Practice*, vol. 29, no. 2, 2017.
[65] C. Marantos, L. Papadopoulos, C. P. Lamprakos, K. Salapas, and D. Soudris, "Bringing energy efficiency closer to application developers: An extensible software analysis framework," *IEEE Transactions on Sustainable Computing*, pp. 1–14, 2022.
[66] S. U. K. Nasir Rashid, "Agile practices for global software development vendors in the development of green and sustainable software," *Journal of Software: Evolution and Practice*, vol. 30, no. 10, p. e1964, 2019.
[67] B. S. . A. Agrawal, "Developing green software," Software & Services Group, Intel Corporation, Folsom, CA, USA, Tech. Rep., 2011.
[68] M. P. Coral Calero, *Green in Software Engineering*. Springer, 2015.
[69] J. Swacha, "Models of sustainable software: A scoping review," *Sustainability*, vol. 14, no. 1, p. 551, 2022.
[70] The Parliamentary Office of Science and Technology, "Energy consumption of ICT," Westminster, London, UK, Tech. Rep. 677, September 2022.
[71] N. Jones, "How to stop data centres from gobbling up the world's electricity," *Nature*, vol. 561, pp. 163–166, 2018.
[72] J. Morley, K. Widdicks, and M. Hazas, "Digitalisation, energy and data demand: The impact of internet traffic on overall and peak electricity consumption," *Energy Research & Social Science*, vol. 38, pp. 128–137, 2018.
[73] P. Koutsovasilis, C. D. Antonopoulos, N. Bellas, S. Lalis, G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "The impact of CPU voltage margins on power-constrained execution," *IEEE Transactions on Sustainable Computing*, vol. 7, no. 1, pp. 221–234, 2022.
[74] A. K. Bhunia and S. S. Samanta, "A study of interval metric and its application in multi-objective optimization with interval objectives," *Computers & Industrial Engineering*, vol. 75, pp. 169 – 178, 2014.
[75] E. Alba and B. Dorronsoro, "The exploration/exploitation tradeoff in dynamic cellular genetic algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 2, pp. 126–142, 2005.
[76] L.-N. Pouchet, "Polybench/c – homepage of louis-noël pouchet," http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2010, last access: 2022-09.

José Miguel Aragón-Jurado received his M.S. degree from University of Cádiz, Spain, in 2022. He is currently pursuing his Ph.D. degree in Computer Science Engineering at the same university, financed by the competitive FPU Spanish national call. His research interests are mainly the design of AI-based solutions for reducing consumption of computing devices in the context of the Interent of Things.

Juan Carlos de la Torre received his M.S. degree from the University of Cádiz, Spain, in 2018. Since then, he is working for his Ph.D. degree in Computer Science Engineering. He is financed by a competitive Spanish national FPU contract, and his main research interests are in the field of green software and green computing.

Patricia Ruiz is a tenured lecturer at University of Cádiz. She obtained her PhD degree in Computer Science at the University of Luxembourg in 2013, and the Telecommunication Engineering degree at the University of Málaga in 2009. She earned the Spanish national award by Telefónica foundation for the best master thesis in 2010. She worked for several european projects (funded with several million euros) at the Public Research Center in Luxembourg and the University of Luxembourg in the context of Intelligent Transportation Systems. In 2015, she also earned a Juan de la Cierva – Formación grant by the Spanish Ministry of Economy and Competitiveness. Her main research areas include dynamic networks, data visualization and optimization techniques. She is PI of several national and international projects. Moreover, she serves as external expert reviewer of H2020 for the Research Executive Agency (REA) of the European Commission.

Pedro L. Galindo is Full Professor of Computer Science and Engineering and leader of the Intelligent Systems group at the University of Cadiz, Spain. He has published more than 100 articles in international journals in the fields of Statistics, Artificial Intelligence, Neural Networks and Image Processing. He is author of the PPA software for peak detection and strain analysis from electron microscopy images. His main research interest is the use of Artificial Intelligence and Image Processing to solve real-world problems under the Industry 4.0 paradigm.

Albert Y. Zomaya is currently the chair professor of High Performance Computing & Networking in the School of Information Technologies, The University of Sydney. He is also the director in the Centre for Distributed and High Performance Computing, which was established in late 2009. He published more than 500 scientific papers and articles and is a author, co-author or editor of more than 20 books. He served as the editor in chief of the IEEE Transactions on Computers (2011-2014). He serves as an associate editor for 22 leading journals, such as, the ACM Computing Surveys, IEEE Transactions on Computational Social Systems, IEEE Transactions on Cloud Computing, and Journal of Parallel and Distributed Computing. He delivered more than 150 keynote addresses, invited seminars, and media briefings and has been actively involved, in a variety of capacities, in the organization of more than 600 national and international conferences. He received the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), the IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), and the IEEE Computer Society Technical Achievement Award (2014). He is a chartered engineer, a fellow of AAAS, IEEE, and IET (United Kingdom). His research interests are in the areas of parallel and distributed computing and complex systems. He is a fellow of the IEEE.

Bernabé Dorronsoro holds a Ph.D. in Computer Science (2007) from the University of Málaga, Spain. He worked for more than seven years in Luxembourg and Lille universities, and he currently serves as an associate professor at the University of Cádiz (Spain), where he leads the GOAL research group (https://goal.uca.es). His main research interests include green computing, sustainable transportation, Grid and Cloud computing, Intelligent Transportation Systems, complex problems optimization, and machine learning. He has published over 50 journal papers and seven books, and he is PI of several national research projects. Dr Dorronsoro is associate editor of the International Journal of Metaheuristics, and member of the editorial board of several journals, including Engineering Applications of Artificial Intelligence and Progress in Artificial Intelligence. His main research interests are in topics related to green computing, sustainable transportation, machine learning and optimization.

Authorized licensed use limited to: UNIVERSIDAD DE CADIZ. Downloaded on November 07,2023 at 07:59:55 UTC from IEEE Xplore. Restrictions apply.
© 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.