Davide Gadioli, Emanuele Vitali, Gianluca Palermo, Cristina Silvano
mARGOt: a Dynamic Autotuning Framework for Self-aware Approximate Computing
IEEE Transactions on Computers, 2018

The published version is available online at: https://doi.org/10.1109/TC.2018.2883597

mARGOt: a Dynamic Autotuning Framework for Self-aware Approximate Computing

Davide Gadioli, Emanuele Vitali, Gianluca Palermo, *Member, IEEE*, Cristina Silvano, *Fellow, IEEE*

Abstract—In the autonomic computing context, the system is perceived as a set of autonomous elements capable of self-management, where end-users define high-level goals and the system shall adapt to achieve the desired behaviour. Runtime adaptation creates several optimization opportunities, especially if we consider approximate computing applications, where it is possible to trade off the accuracy of the result and the performance. Given that modern systems are limited by the power dissipated, autonomic computing is an appealing approach to increase the computation efficiency.
In this paper, we introduce *mARGOt*, a dynamic autotuning framework to enhance the target application with an adaptation layer to provide self-optimization capabilities. The framework is implemented as a C++ library that works at function-level and provides to the application a mechanism to adapt in a reactive and a proactive way. Moreover, the application is capable to change dynamically its requirements and to learn online the underlying application-knowledge. We evaluated the proposed framework in three real-life scenarios, ranging from embedded to HPC applications. In the three use cases, experimental results demonstrate how, thanks to *mARGOt*, it is possible to increase the computation efficiency by adapting the application at runtime with a limited overhead.

Index Terms—Autonomic Computing, Dynamic Autotuning, Adaptive Applications, Self-optimization, Approximate Computing

1 Introduction

With the end of Dennard scaling [1], the performance of modern systems are limited by the power dissipated. This shifted the focus of system optimization towards energy efficiency in a wide range of scenarios, not only related to embedded systems but also related to high-performance computing (HPC) [2]. To further improve efficiency, several approaches aim at finding *good enough* results for the end-user, thus saving the unnecessary computational effort. A large class of applications implicitly expose software-knobs at the algorithmic-level to find accuracy-throughput tradeoffs, especially in image processing applications [3] and whenever it is possible to use approximation techniques, such as loop perforation [4] and task skipping [5]. Examples of software-knobs can be the number of samples in a Monte Carlo simulation, the resolution of an output image or the number of software threads used by an application.

Among the implications of this trend, application requirements are increasing in complexity. Typically, the end-user has complex requirements which involve extra-functional properties (EFPs) in conflict with each other, such as power consumption, throughput, and accuracy. Moreover, these extra-functional properties might depend on the actual inputs of the application, on the resources available for the application and on the configurations of the underlying architecture (such as the core frequencies). Therefore, it is not simple to define the relationship between a software-knob configuration and the EFPs of interest.

• *E. Vitali, D. Gadioli, G. Palermo, and C. Silvano are with Dipartimento di Elettronica, Infomazione e Bioingegneria, Politecnico di Milano, Italy.*

This work is supported by the European Commission Horizon 2020 research and innovation program under grant agreement No 671623, FET-HPC ANTAREX.

To address these problems, in this paper we propose *mARGOt*, a dynamic autotuning framework to enhance the target application with a flexible adaptation layer. The main idea is that the end-user specifies high-level goals, such as "maximize accuracy given a throughput of at least 25 frames per second", while *mARGOt* tunes the application software-knobs accordingly. Given that *mARGOt* is coupled with the application execution, it is able to identify and seize optimization opportunities at runtime. On one side, *mARGOt* is capable to react to changes in the execution environment: if the application performance degrades due to a change of the core frequency, the violation of the high-level goals triggers *mARGOt*, which selects a different configuration to compensate. On the other side, *mARGOt* is capable to leverage input features to select a configuration tailored for the current input.

From the methodology point of view, *mARGOt* falls in the context of autonomic computing [6] as an implementation of the well-known **M**onitor **A**nalyze **P**lan **E**execute feedback loop, based on application **K**nowledge (MAPE-K). From the implementation point of view, *mARGOt* is implemented as a standard C++ library to be linked to the target application and working at function-level. With this approach, each instance of the application can take decisions autonomously. Our seminal work on autotuning started as a component coupled with a resource manager [7]. Then, we have shown in [8], [9] the benefits of using a dynamic autotuning framework as a stand-alone component. The main contributions of this paper are:

- We introduce a mechanism to adapt the application in a reactive and proactive way, according to the input features;
- We introduce the possibility to express a confidence in the application requirements;

- We investigate the possibility to derive the application-knowledge online;
- We evaluate *mARGOt* in a set of approximate applications, from an embedded to an HPC context.

Furthermore, we publicly released the *mARGOt* source code, along with build instructions and user manuals for integration and implementation details [10]. Our goal is to let application developers to easily integrate *mARGOt* in their applications for improving the application efficiency.

The rest of the paper is organized as follows. Section 2 provides an overview of the state-of-the-art, outlining the *mARGOt* contributions, while Section 3 formalizes the proposed approach by focusing on the adaptation mechanisms. Section 4 shows the integration workflow, describing the required effort. Section 5 validates the proposed framework in terms of introduced overhead and exploitation in adaptive applications. Finally, Section 6 concludes the paper.

2 Related Work

The proposed framework belong to the domain of autonomic computing [6]. In this context, a computing system is perceived as a set of autonomic elements capable of self-management without a human-in-the-loop. According to the proposed vision, an autonomic element must have self-configuration, self-optimization, self-healing and self-protection capabilities. Self-configuration is the capability to incorporate in the system new components whenever they become available, as in the Rainbow framework [11]. Self-healing is the capability to recover from hardware or software failures, as proposed in [12]. Self-protection is the capability to defend itself against malicious attacks or failures not corrected by any self-healing mechanism, as proposed in [13]. Eventually, self-optimization is the capability to identify and seize opportunities to improve the application performance or efficiency. Even if some previous works (such as the ABLE framework [14]) aim at defining a common interface to derive an autonomic manager, the problem how to design a manager that provides self-* properties is still an open question. The goal of *mARGOt* is to enhance an existing application with an adaptation layer that provides the ability of *self-optimization*. Previous surveys [15], [16] provide a more general overview of the research area.

The definition of a system in the context of autonomic computing involves both hardware and software. Therefore, in literature, there are several works that aim at optimizing the system performance or efficiency. We might divide them into three main categories: resource managers, static autotuners, and dynamic autotuners.

Resource managers address system adaptability through resource management and allocation: in the data center context [17], [18], in the grid computing context [19], in the multi/manycore node context [20], [21], [22] and for embedded platforms [23], [24]. These works are indeed interesting, however, *mARGOt* aims at leveraging the assigned resources to reach the end-user requirements, therefore it takes orthogonal decisions.

Application autotuning frameworks aim at selecting the most suitable configuration of the software-knobs to leverage the assigned resources. Among these frameworks, there are *static autotuners* to select the most suitable configuration before the production phase, and *dynamic autotuners* to select the most suitable configuration during the production phase.

2.1 Static Autotuning Frameworks

The Design Space (DS) of an application grows exponentially with respect to the number of software-knobs, thus increasing the complexity of the Design Space Exploration (DSE). Typically, static autotuning frameworks focus on finding the configuration that maximizes/minimizes a utility function in a large design space given a reasonable amount of time.

Active Harmony [25], ATune-IL [26] and AutoTune [27] are frameworks targeting application-agnostic software-knobs, such as tiling size, loop unrolling factor, compiler options and/or algorithm selection. The main goal is to tailor the application configuration for the underlying hardware. OpenTuner [28] and ATF framework [29] are also targeting application-specific software-knobs. However, they are usually applied in a predictable execution environment and they target software-knobs that have a loose relationship with the actual input set. QuickStep [30], Paraprox [31] and PowerGAUGE [32] target parallel regions of an application and perform code transformation (or binary transformation) without preserving the semantics. The idea is to automatically expose and leverage the accuracy-throughput tradeoff. These works are typically applied in a predictable execution environment and they usually target a different class of software-knobs with respect to dynamic autotuners. By choosing a configuration at design-time, it is impossible to react to changes related to either the application requirements or to the observed performance, for example due to a change on the core frequency for thermal issues. Moreover, the decision algorithm does not leverage the input features.

In the context of High-Performance Computing, there are several autotuning frameworks, however, they are tailored to specific tasks. Some examples of them are ATLAS [33] for matrix multiplication routines, FTTW [34] for FFTs operations, OSKI [35] for sparse matrix kernels, SPIRAL [36] for digital signal processing, CLTune [37] for OpenCL applications, Patus [38] and Sepya [39] for stencil computations.

2.2 Dynamic Autotuning Frameworks

The fundamental characteristic of the dynamic autotuning frameworks is to continuously tune the software-knobs configuration at runtime. The main idea is to leverage information about the actual execution context, rather than the average behaviour, when they decide which is the most suitable software-knob configuration to apply. Usually, they rely on the application-knowledge to predict the behaviour of a configuration and to drive the decision process.

Configuring an application at runtime has been an appealing idea investigated in literature for a long time. The ADAPT framework [40], the work in [41] leveraging on Bayesian networks, and the autotuner derived in the work that proposes the ABLE framework [14] are some pioneering works in this area.

More recent works on approximate computing evaluate the possibility of relaxing the constraints on functional correctness to improve the efficiency as long as they tolerate a lower accuracy of the results. A large class of applications, such as multimedia, implicitly defines application-specific software-knobs that affect the output quality. When it is complex to identify the software-knobs, works in the literature describe techniques to expose them. For example, it is possible to fail task on purpose [5] or to skip iterations of a loop [4]. A later work [42] investigates the effect of loop perforation by using a large set of applications from the PARSEC benchmark suite [43], showing how a small loss in accuracy might lead to a significant performance increment.

The Sage framework [44], the Green framework [45] and PowerDial [46] are examples of autotuners falling in this category. The idea is to maximize the throughput given a lower bound on the computational error. Later work in [47] proposes a blueprint of a controller to extend the one in PowerDial to handle a tradeoff among several metrics by introducing limitations on the software-knobs. Although very interesting, these works on approximate computing provide a limited flexibility to end-users to define application requirements. Moreover, they do not leverage input features to further improve computational efficiency.

An interesting work leveraging input features is Capri [48], which inspired us for this work. At design time, Capri uses a set of representative inputs to model a cost metric (e.g. execution time or energy) and an error metric, as a function of software-knobs configuration and input features. The controller selecting the most suitable configuration is based on the Valiant's probably approximately correct (PAC) theory [49]. Given that Capri addresses applications with a single input instead of a stream of inputs, it does not use any reaction mechanism to adapt the application-knowledge. Moreover, due to the chosen formulation and the target applications, the feasible region given by the error function does not depend on the actual input. This might miss some optimization opportunities when input features are related to the error, as in the *Probabilistic time-dependent routing* application (Section 5.4).

A rather different approach is Anytime Automaton [50], which does not rely on the application-knowledge. It suggests radical source code transformations to re-write the application in a pipeline style. The idea is that the longer the given input executes in the pipeline, the more accurate the output becomes.

Another approach that does not rely on the application-knowledge is the IRA framework [51]. IRA investigates several features of each input (such as the mean value or its autocorrelation) to generate a smaller input for searching for the fastest configuration given a bound on the minimum accuracy. However, in a certain class of applications, such as in the *GeoDock* application (Section 5.3), it is not simple to sub-sample the inputs due to its heterogeneous information, limiting the framework applicability.

Besides autotuning frameworks, Petabricks [52] is a language to expose algorithmic choices to the compiler. The Petabrick framework (including compiler and autotuner) analyzes the code and generates a strategy, embedded in the executable, to select the fastest algorithm and configuration according to the input size. In later work, Petabricks has been enhanced to leverage the accuracy-throughput tradeoffs [53] and to take in consideration input features [54]. Petabricks is indeed interesting, however, it generates the adaption strategy at design-time, without building any application-knowledge, therefore it is not flexible to change application requirements and it relies on a predictable execution environment. Also Siblingrivarly [55] is based on the Petabriks language, but it targets a very unpredictable execution environment.

TABLE 1: Comparison of dynamic autotuning frameworks In the assigned score, a higher number of stars is better.

Framework	Tradeoffs	Reactivity	Proactivity	Integration effort	Runtime overhead
ADAPT [40]	⋆	⋆ ⋆	⋆	⋆ ⋆ ⋆	Look-up table
H.Guo [41]	⋆	⋆	⋆ ⋆ ⋆	⋆	Decision tree
Sage [44]	⋆ ⋆	⋆	⋆	⋆ ⋆ ⋆	Decision tree
Green [45]	⋆ ⋆	⋆	⋆	⋆ ⋆	Decision tree
PowerDial [46]	⋆ ⋆	⋆ ⋆	⋆	⋆ ⋆ ⋆	Look-up table
Capri [48]	⋆ ⋆	⋆	⋆ ⋆ ⋆	⋆ ⋆	Model query
Anytime Automaton [50]	⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆	Kill interrupt
IRA [51]	⋆ ⋆	⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆	DSE on small input
Petabricks [54]	⋆ ⋆	⋆	⋆ ⋆ ⋆	⋆	Decision tree
Sibling Rivalry [55]	⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆	⋆	Genetic algorithm
mARGOt	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆ ⋆	⋆ ⋆	Look-up table

To summarize the results of our analysis of the state-of-the-art, Table 1 compares dynamic autotuning frameworks by scoring their main characteristics in terms of: trade-off analysis, reactivity, proactivity, integration effort and runtime overhead.

In the second column, we score the flexibility of the framework to leverage tradeoffs among different extra-functional properties of interest for the end-user. More in detail, we classified with a single star the frameworks using a single metric in the optimization process, while two stars are assigned to frameworks based on a fixed optimization problem (such as to maximize accuracy given a constraint on throughput). The highest score (three stars) is given to frameworks providing to the end-user the possibility to define an arbitrary optimization problem.

The third column classifies the capability of a framework to runtime adapts in a reactive mode. One star is assigned to a framework that either does not provide any reaction mechanism or it is based on a trial-and-error approach. Two stars are given to a framework that leverages a structured approach to react, for example by using control theory. The three-star rating is assigned to a framework that also offers to the end-user the possibility to change the application requirements according to external stimuli.

The fourth column targets the capability to adapt in a proactive mode according to the features of the current input. The lowest score represents a framework that relies on the average behaviour of the application. Two stars are assigned to frameworks that leverage only the size of the current input. If a framework leverages on more information

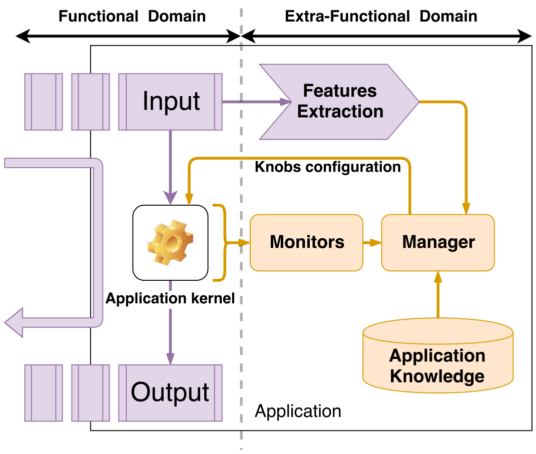


Fig. 1: Global architecture of the proposed framework. Purple elements represent application code, while orange elements represent *mARGOt* high-level components. The black box represents the executable boundary.

from the current input, we rated it with three stars.

The fifth column targets the effort required by the application developer to integrate the framework in the target application. The lower rating represents frameworks that explicitly require a porting of the application in a new language or to rewrite the source code. The higher rating represents frameworks that explicitly provide mechanisms that automatically integrate the framework. The two-star symbol represents a framework requiring a limited integration effort.

Finally, the last column reports the method used to solve the optimization problem, whose complexity is used to compare the overheads of each evaluated framework. In all cases, the overhead is largely amortized by the advantages introduced by the dynamic autotuning.

Given the limitations emerged in the analysis of the literature, our goal is to overcome them by introducing the following contributions:

- Flexibility to express application requirements represents one of the key points of the methodology. In *mARGOt*, application requirements are expressed as a constrained multi-objective optimization problem, given an arbitrary number of constraints and addressing an arbitrary number of EFPs as well.
- Capability to leverage on actual information rather than the expected average behaviour. *mARGOt* provides mechanisms to react to changes in the application performance and requirements. *mARGOt* also provides a mechanism to adapt in a proactive way according to input features.
- Limited integration effort: *mARGOt* minimizes as much as possible the number of lines of code to be changed and we designed the interface as a *wrapper* around the managed region of code, therefore limiting the intrusiveness.

3 Proposed Methodology

Figure 1 shows an overview of the *mARGOt* framework and how it interacts with an application. We assume that the application is composed of a single kernel g that elaborates an input i to generate the desired output o, however *mARGOt* can manage different blocks of code of a single application in an independent way. We assume that the kernel algorithm exposes software-knobs that alter its EFPs, such as the number of Monte Carlo simulations or the parallelism level. Let $\overline{x} = [x_1, \ldots, x_n]$ the vector of software-knobs, then we might define a kernel as $o = g(\overline{x}, i)$. In this description and in the rest of the paper, we assume for simplicity that the application is defined by a single kernel, or *block* of code.

Given this abstraction of the target application, the end-user requirements are defined as follows. We denote the metrics of interest (i.e. EFPs) as the vector $\overline{m} = [m_1, m_2, \ldots, m_n]$. Let us suppose that the application developers are capable to extract features of the current inputs, for example the ones analyzed in IRA [51]. We denote such properties as the vector $\overline{f} = [f_1, f_2, \ldots, f_n]$. The end-user is capable to define the application requirements as in Eq. 1:

$$\begin{array}{lllll} \max(\min) & r(\overline{x}; \overline{m} \mid \overline{f}) & & & \\ \text{s.t.} & C_1 : \omega_1(\overline{x}; \overline{m} \mid \overline{f}) & \propto & k_1 & \textit{with } \alpha_1 \textit{ confidence} \\ & C_2 : \omega_2(\overline{x}; \overline{m} \mid \overline{f}) & \propto & k_2 & \\ & \ldots & & & \\ & C_n : \omega_n(\overline{x}; \overline{m} \mid \overline{f}) & \propto & k_n & \end{array} \tag{1}$$

where r denotes the objective function (named *rank* in *mARGOt* context) and it is defined as a composition of any variable defined either in $\overline{x}$ or in $\overline{m}$ by using their mean values. Let C be the set of constraints, where each C_i is a constraint expressed as the function ω_i, defined over the software-knobs or the EFPs, that must satisfy the relationship $\propto \in \{<, \leq, >, \geq\}$ with a threshold value k_i and with a confidence α_i (if ω_i targets a statistical variable). Being agnostic to the distribution of the target parameter, the confidence is expressed as the number of times to consider its standard deviation. If the application is input-dependent, the value of the rank function r and the constraint functions ω_i also depend on the features of the input $\overline{f}$.

In this formulation, the main goal of *mARGOt* is to solve the following optimization problem: finding the configuration $\hat{\overline{x}}$ that satisfies all the constraints C and maximizes (minimizes) the objective function r, given the current input i. The application must have a configuration, even if it is not feasible to satisfy all the constraints. For this reason, *mARGOt* might relax some of the constraints, until a feasible solution is found, starting by relaxing the constraint with the lowest priority. Therefore, the end-user must sort the set of constraints by their priority. As shown in Figure 1, the *mARGOt* framework is composed of the application manager, the monitors module, and the application-knowledge. The next sections explain each component in more detail.

3.1 Application-knowledge

For a generic application, the relation between the software-knobs, the EFPs of interest and the input features is complex and unknown a priori. Therefore, we need a model of the application extra-functional behaviour to solve the optimization problem stated in Eq. 1. *mARGOt* uses a list of *Operating Points* (OPs) as application-knowledge, where each Operating Point θ expresses the target software-knob configuration and the achieved EFPs with the given input features; i.e. $\theta = \{x_1, \ldots, x_n, f_1, \ldots, f_n, m_1, \ldots, m_n\}$. We choose this solution mainly for three reasons: to solve efficiently the optimization problem by inspection, to guarantee

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <points version="1.3" block="example">
3    <point>
4     <parameters>
5      <parameter name="knob1" value="3.4"/>
6      <parameter name="knob2" value="100"/>
7     </parameters>
8     <system_metrics>
9      <system_metric name="metric1" value="212.862" standard_dev="6.49"/>
10     <system_metric name="metric2" value="27.6" standard_dev="0.9"/>
11    </system_metrics>
12    <features>
13     <feature name="feature1" value="100"/>
14     <feature name="feature2" value="10" />
15    </features>
16   </point>
17 </points>
```

Fig. 2: XML configuration file to define the application-knowledge for an application.

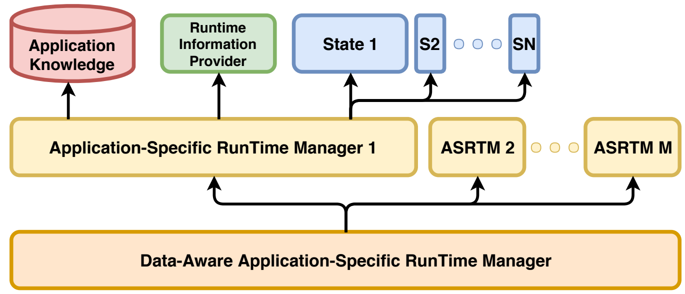


Fig. 3: Overview of the Application Manager implemented in *mARGOt*, based on a hierarchical approach.

that *mARGOt* will not choose an illegal configuration for the application and to provide a great flexibility in terms of management.

Figure 2 shows an example of application-knowledge configuration file in XML, with a single Operating Point (lines 3-16). Let us suppose that the target application exposes two software-knobs (*knob1* and *knob2*), there are two metrics (*metric1* and *metric2*) and it is possible to extract two features from the current input (*feature1* and *feature2*). In this example, the OP is composed of three sections: the target software-knobs configuration (lines 4-7), the reached performance distribution (lines 8-11) and the related feature cluster (lines 12-15).

The OP list is considered a required input and *mARGOt* is agnostic to the methodology used to obtain it. Typically this methodology is a design-time task, known as Design Space Exploration (DSE) in literature. This task is a well-known problem and there are several previous approaches to find the Pareto Set in an efficient way [56], [57], [58]. Moreover, we implemented the possibility to change the application-knowledge at runtime. Section 3.4 describes the proposed approach for the online DSE.

3.2 Monitors

This module enables *mARGOt* to observe the actual behaviour of either the application or the execution environment. This feature is critical for an autonomic manager, because it provides feedback information, thus enabling self-awareness ability [59]. The application-knowledge defines the expected behaviour of the application, however, it might change according to the evolution of the system. For example, a power capper might reduce the frequency of the processor due to thermal reasons. In this case, we would expect that the application notices a performance degradation thus reacting by using a different configuration to compensate. This adaptation is possible only if we have some feedback information.

From the implementation point of view, *mARGOt* provides to application developers a suite of monitors to observe the most common EFPs such as throughput, system-wide CPU usage or Perf events through the PAPI interface [60]. However, implementing a monitor to observe a custom EFP, such as the output quality, is straightforward.

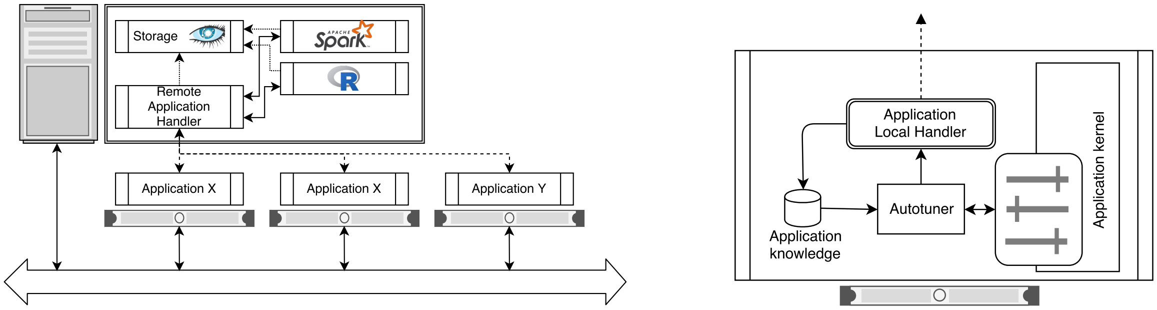
Given that measuring quality metrics might be expensive, *mARGOt* does not require a continuous observation of a metric. The application developers can choose if monitoring an EFP at each iteration, periodically or sporadically. Obviously, decreasing the observation frequency delays the reactions of *mARGOt*. If it is not possible to monitor an EFP at runtime, *mARGOt* can rely only on the expected behaviour, thus operating in an open-loop.

3.3 Application Manager

This component is the core of the *mARGOt* dynamic autotuner, which provides the self-optimization capability. From the methodology point of view, this component is in charge of solving the optimization problem stated in Eq. 1: to find the software-knobs configuration $\hat{\bar{x}}$, while reacting to changes in the execution environment and adapting in a proactive way according to the input features.

From the implementation point of view, the application manager has a hierarchical structure, as shown in Figure 3, where each sub-component solves a specific problem. The *Data-Aware Application-Specific Run-Time Manager* (DA AS-RTM) provides a unified interface to application developers to set or change the application requirements, to set or change the application-knowledge and to retrieve the most suitable configuration $\hat{\bar{x}}$. Internally, the DA AS_RTM clusters the application-knowledge according to the input features $\bar{f}$ by creating an *Application-Specific Run-Time Manager* (AS-RTM) for each cluster of Operating Points with the same input features. Therefore, the clusters of OPs are implicitly defined in the application-knowledge. Given the input features of the current input, the DA AS-RTM selects the cluster with the features closer to the ones of the current input. It is possible to use either a Euclidean distance between the two vectors or a normalized one, in case an element of the vector $\bar{f}$ is numerically different with respect to the others. Moreover, it is possible to express some constraints on the selection of the cluster. For example, it is possible to enforce that the feature $f_i^{cluster}$ of the selected cluster must be lower (higher) or equal than the feature f_i^{inpt} of the current input, i.e. $f_i^{cluster} \propto f_i^{inpt}$. Once the cluster for the current input is selected, the corresponding (AS-RTM) solves the optimization problem by relying on the following components.

The *State* element is in charge of solving the optimization problem by using a differential approach. The initial optimization problem does not have any constraint (i.e. $C = \emptyset$) and the objective function minimizes the value of the first software-knob. From this initial state, the application



(a) Global structure of the distributed DSE framework

(b) Structure of an application instance

Fig. 4: Proposed approach for distributed online Design Space Exploration, using a dedicated server outside of the computation node. We used MQTT protocol for extra-node communication.

ALGORITHM 1: How the State component builds the internal representation of the optimization problem.

Data: Application-knowledge OP_{list}, optimization function r, list of constraints C
Result: list of valid OPs L_valid, lists of invalid OPs L_{c_i}
$L_{valid} = OP_{list}$;
for $c_i \in C$ *(ascending priority order)* **do**
  $L_{c_i} = \emptyset$;
  for $OP_j \in L_{valid}$ **do**
    if OP_j *does not satisfy* c_i **then**
      $L_{c_i} = Lc_i \cup OP_j$;
    end
  end
  $L_{valid} = L_{valid} \setminus L_{c_i}$;
  $L_{c_i} = sort(L_{c_i}, dist(OP_j, c_i))$;
end
$L_{valid} = sort(L_{valid}, r)$;

ALGORITHM 2: How the State element solves the optimization problem.

Data: list of valid OPs L_valid, lists of invalid OPs L_{c_i}, list of constraints C
Result: most suitable Operating Point $\overline{OP}$
if $L_{valid}! = \emptyset$ **then**
  return $L_{valid}[0]$;
else
  for $c_i \in C$ *(descending priority order)* **do**
    if $L_{c_i}! = \emptyset$ **then**
      return $L_{c_i}[0]$;
    end
  end
end

might dynamically add constraints, define a different objective function or change the application-knowledge. The solver can find efficiently the new optimal configuration evaluating only the involved OPs, by building an internal representation of the optimization problem. Algorithm 1 shows the pseudo code for its initialization. At first, it assumes that the application-knowledge satisfies all the constraints, therefore L_{valid} contains all the OPs. Then, for each constraint c_i, *mARGOt* iterates over the set of OPs in L_{valid} and it performs three operations: (1) It creates the list L_{c_i} of all Operating Points invalidated by the constraint c_i. (2) Then it removes from the set of valid OPs (L_{valid}) those not satisfying c_i (L_{c_i}). (3) Eventually, it sorts all the OPs in L_{c_i} according to their distance from satisfying the constraint c_i. After iterating over the constraints, *mARGOt* sort the list of valid OPs L_{valid} according to the objective function r. Using this representation, each time that *mARGOt* is invoked to solve the optimization problem, it updates the internal structure and then it follows the Algorithm 2. In particular, if the list L_{valid} is not empty, *mARGOt* returns the one that maximizes the rank function, i.e. $L_{valid}[0]$. Otherwise, *mARGOt* iterates over the constraints according to their priority, in reverse order, until it finds a constraint c_i with a non-empty L_{c_i}. Then the best OP is the closest to satisfy the constraint c_i, i.e. $L_{c_i}[0]$. This algorithm must always return a single OP. Therefore, if there is more than one OP at the same distance from c_i, *mARGOt* uses the constraints with a lower priority than c_i and the objective function r to select the best OP.

Given that the end-user might have different requirements according to different phases of the application, it is possible to define different states and switch among them at runtime. As an example, in a video surveillance application, the end-user would run a more accurate computation or a more energy-efficient one, according to the presence of an interesting scenario to analyze.

The *Runtime Information Provider* relates an EFP of the application-knowledge with an application monitor. In particular, it compares the observed behaviour with the expected one and it computes an error coefficient defined as $e_{m_i} = \frac{expected_i}{observed_i}$, where e_{m_i} is the error coefficient for the i-th EFP. To avoid the zero trap, we add 1 to the numerator and denominator when $observed_i$ is equal to zero. Since it is impossible to observe the error coefficient also for other configurations (the application uses only one configuration each time), we assume that their error coefficients are equal to the observed one. This implies that if we observe a degradation of the performance of 10% with respect to the current configuration, we assume that also the other configurations will have a performance degradation of 10%. Therefore we scale the constraint value accordingly to react. For example, assuming that the end-user would like a throughput of at least $25fps$ and that we are using a configuration that has an expected throughput of $30fps$, but we observe a throughput of $15fps$. Then, the *Runtime Information Provider* will double the constraint value to compensate.

3.4 Online Design Space Exploration

The *mARGOt* implementation lets the application developer to define the application-knowledge at runtime, thus enabling the possibility to perform online learning. In particular, we propose an approach to distribute the Design Space

Exploration among all the instances of an unknown application, integrated with *mARGOt*, at runtime. Figure 4a shows the overall picture of the approach, highlighting the two main actors: the *Remote Application Handler* and the running application instances. Each instance of the application has an *Application Local Handler* (client) (as shown in Figure 4b) which interacts with the *Remote Application Handler* (server) through either MQTT or MQTTs protocols. The *Application Local Handler* is an asynchronous utility thread that manipulates the client application-knowledge and sends to the *Remote Application Handler* telemetry information. The *Remote Application Handler* is a worker thread-pool that interacts with the clients to obtain the application-knowledge. The server stores information in a Cassandra database and it uses a plugin system to model and interpolate the relations between the EFPs, the software-knob configurations and the input features clusters, including also a wrapper interface for R and Spark. Although the implementation of a plugin to derive a metric is straightforward, *mARGOt* provides two default plugins. The first one is rather simple and it computes the mean value and standard deviation for each observed software-knobs configuration. This can be used for a full-factorial Design Space Exploration, observing the whole Design Space including the possible input features. The second plugin leverages a well-known approach [61] to interpolate application performance implemented by the state-of-the-art R package [62].

The typical workflow used for the online Design Space Exploration on an unknown application can be described as follows:

1) Each client notifies itself to the server.
2) The server sends a request for information, such as the domain of each software-knob, the name of the plugin that models each EFP of interest and the desired Design of Experiment technique.
3) The server generates a DoE for the application and it starts to dispatch configurations to each client in a round robin fashion.
4) Each client manipulates the *mARGOt* application-knowledge to force the selection of the software-knob configuration sent by the server.
5) After each kernel execution, the client sends the observed performance and input features to the server.
6) Once the clients have observed all the configurations in the DoE phase, the server builds the application-knowledge and it broadcasts the Operating Point list to the clients.

The system is designed to be resilient to server and client crashes without interfering with MPI traffic. As soon as a client becomes available, it can join other clients at any time, thus contributing to the DSE or receiving directly the application-knowledge. As future work, we plan to implement additional well-known techniques to model the application performance, such as those described in [63].

The benefits of the proposed online DSE architecture are twofold. On one hand, it leverages the parallelism of the platform to reduce the DSE time. On the other hand, it uses standard tools to visualize extra-functional values stored in the database (e.g. execution traces of the application instances running on the platform or to query the application-knowledge).

3.5 Summary of *mARGOt* Main Features

The *mARGOt* framework provides a runtime self-optimization layer to adapt applications in a reactive and in a proactive way. Differently from static autotuner frameworks, *mARGOt* focuses on application-specific software knobs, whose optimal value depends on the system workload, on changes in the application requirements or on features of the actual input. In particular, *mARGOt* might change the software-knobs configuration if: 1) the application requirements change, 2) the application-knowledge changes, 3) the expected performance differs from the observed one, and 4) according to the features of the current input. Moreover *mARGOt* has been designed to be lightweight and flexible to enable its deployment in a wide range of scenarios.

A key feature of *mARGOt* is how to derive the application-knowledge. We offer to application developers two possibilities. First, they might leverage on well-known techniques to run a DSE at design-time. Second, we provide a software architecture to run the DSE directly at runtime by leveraging the *mARGOt* capability to change the application-knowledge.

4 Integration in the Target Application

In this section, we describe the effort required by end-users and application developers to integrate *mARGOt* in their application. In this context, end-users are the final users of the application, therefore they are in charge of defining the application requirements and identifying the input features (if any). Application developers are the experts writing the application source code, therefore they are in charge of identifying the software-knobs and extracting the features from the input (if any). From the implementation point of view, we designed the framework: (i) to apply the separation of concern approach between functional and extra-functional properties; (ii) to limit the code intrusiveness in terms of the number of lines of code to be changed and (iii) to propose an easy-to-use instrumentation of the code. To ease the integration process in the target application, *mARGOt* provides a utility tool that starting from an XML description of the extra-functional concerns, it generates a high-level interface tailored for the target application. The main configuration file describes the adaptation layer by defining:

1) The monitors of interest for the application;
2) The geometry of the problem, i.e. the EFPs of interest, the application software-knobs, and the data features of the input;
3) The application requirements, i.e. the optimization problem stated in Eq. 1.

If the application developers derive the application-knowledge at design-time, the second configuration file states the list of Operating Points as shown in Figure 2.

Starting from this high-level description of the layer, the utility tool generates a library with the required glue code to hide, as much as possible, the *mARGOt* implementation

```xml
1  <margot application="toy_app" version="v1">
2   <block name="foo">
3
4     <!-- MONITOR SECTION -->
5     <monitor name="exec_time_monitor" type="Time">
6       <expose var_name="avg_exec_time" what="average"/>
7     </monitor>
8     <monitor name="error_monitor" type="Custom">
9       <spec>
10        <header reference="margot/monitor.hpp"/>
11        <class name="margot::Monitor&lt;float&gt;"/>
12        <type name="float"/>
13        <stop_method name="push"/>
14      </spec>
15      <stop>
16        <param>
17          <local_var name="error" type="float"/>
18        </param>
19      </stop>
20      <expose var_name="avg_error" what="average"/>
21    </monitor>
22
23    <!-- APPLICATION GEOMETRY -->
24    <knob name="k1" var_name="knob1" var_type="int"/>
25    <knob name="k1" var_name="knob1" var_type="int"/>
26    <metric name="exec_time" type="float" distribution="yes"/>
27    <metric name="error" type="float" distribution="yes"/>
28    <features distance="euclidean">
29      <feature name="feature1" type="double" comparison="-"/>
30      <feature name="feature2" type="double" comparison="LE"/>
31    </features>
32
33    <!-- ADAPTATION SECTION -->
34    <goal name="exec_time_goal" metric_name="exec_time" cFun="LE" value="2"/>
35    <adapt metric_name="exec_time" using="exec_time_monitor" inertia="3"/>
36    <state name="normal" starting="yes">
37      <minimize combination="simple">
38        <metric name="error" coef="1.0"/>
39      </minimize>
40      <subject to="exec_time_goal" confidence="1" priority="10"/>
41    </state>
42
43  </block>
44 </margot>
```

Fig. 5: The main XML configuration file for the toy application, stating extra-functional concerns

details. In particular, the high-level interface exposes five functions to the developers:

- **init**. A global function that initializes the data structures.
- **update**. A block-level function that updates the application software-knobs with the most suitable configuration found.
- **start_monitor**. A block-level function that starts all the monitors of interest
- **stop_monitor** A block-level function that stops all the monitors of interest
- **log** A block-level function that logs the application behaviour

These functions hide the initialization of the framework and its basic usage. For example, the update function takes as output parameters the software-knobs of the application and as input parameters the features of the current input. It uses the features to select the most suitable cluster and then it sets software-knobs parameters according to the most suitable configuration found by *mARGOt*. However, if application developers need a more advanced adaptation strategy, such as changing the application requirement at runtime, they need to use the *mARGOt* interface on top of the high-level one.

To show the integration effort, we focus on a toy application with two software-knobs (*knob1* and *knob2*) and two input features (*feature1* and *feature2*). The application algorithm is rather simple: it is composed of a loop that continuously elaborates new inputs. In this toy application, we assume that the end-user is concerned about execution time and computational error. In particular, he/she would like to minimize the computational error given an upper

```cpp
#include <margot.hpp>

int main()
{
 margot::init();

 int knob1 = 4;
 int knob2 = 2;
 float error = 0.0f;

 while (work_to_do())
 {
  new_input = get_input();
  const double feature1 = extract_feature1(new_input);
  const double feature2 = extract_feature2(new_input);

  MARGOT_MANAGED_BLOCK_FOO
  {
     do_job(new_input, knob1, knob2);
     error = compute_error(new_input);
  }
 }
}
```

Fig. 6: Stripped C++ code of the target toy application, after the *mARGOt* integration.

bound on the execution time.

In the context of this toy application, Figure 5 shows the main XML configuration file that expresses the extra-functional concerns. This file is composed of three sections: the monitor section (lines $4-21$), the application geometry section (lines $23-31$) and the adaptation section (lines $33-41$).

The monitor section lists all the monitors of interest for the user. In this example, we have an execution time monitor (lines $5-7$) and a custom monitor for observing the error (lines $8-21$). All the monitors might expose to application developers a statistical property over the observations, such as the average value in this example (line 6 and 20). If the end-user is not interested in observing the behaviour of the application, he/she might omit this section.

The application geometry section lists the application software-knobs (lines $24, 25$), the metrics of interest (lines $26, 27$) and the features of the input (lines $28-31$). In particular, it is possible to specify how to compute the distance between feature vectors (line 28) and to specify constraints on their selection, as described in Section 3.3. For example, if we consider *feature2* (line 30), we state that a cluster is eligible to be selected only if its *feature2* value is lower or equal than the *feature2* value of the current input. If we consider *feature1* (line 29) instead, we state that we do not impose any requirement on a cluster to be eligible. This mechanism provides to *mARGOt* a way to adapt proactively by sizing optimization opportunities according to the actual input.

While the application geometry describes the boundaries of the problem, the adaptation section states the application requirements of the end-user. In particular, it states the application goals (line 34), the feedback information from the monitor (line 35) and the constrained multi-optimization problem (lines $36-41$). In the definition of a constraint (line 40), it is possible to specify a confidence and a priority. The confidence specifies how many times *mARGOt* has to take into account the standard deviation to improve the resilience against the noise with respect to the average behaviour. The priority is used to sort the constraints by their importance for the end-user. Application goals and feedback information provide to *mARGOt* the capacity to adapt in a reactive way. A violation of a goal in the optimization problem or a discrepancy between the observed and expected behaviour of the application, triggers an adaptation from *mARGOt*, thus reacting to the event.

Starting from this configuration file, *mARGOt* automatically generates the glue code accordingly, exposing to application developers a high-level interface tailored to the specific problem. For a complete description of the XML syntax and semantics, please refer to the user manual in the *mARGOt* repository [10].

Figure 6 shows the source code of the toy application after the integration with *mARGOt*. To highlight the required effort, we hide the application algorithm in three functions: *work_to_do* (line 11) tests whether input data are available, *get_input* (line 13) retrieves the last input to elaborate and *do_job* (line 19) performs the elaboration. The integration effort requires to the application developers to include the *mARGOt* header (line 1), to initialize the framework (line 5) and to wrap the block of code managed by *mARGOt* (lines 17, 18, 21). Due to the structure of the code, it is possible to use a pre-processor macro to hide the five functions described above.

Even if we minimized the integration effort, we still require from application developers to identify and to write code to extract meaningful features from an input (lines 14, 15) and a function to compute the elaboration error (line 20). Although these metrics are heavily application-dependent, a large percentage of works in literature analyze generic error metrics [46] and generic input features [51]. Application developers might consider these previous works as starting points to identify more customized metrics for their applications.

5 Experimental Results

This section aims at validating and assessing the benefits of the proposed dynamic autotuning framework.

First, we evaluate the overheads introduced by *mARGOt* in different scenarios. Then, we show how it is possible to leverage the dynamic adaptation to improve the computation efficiency in three different use cases. To emphasize the applicability of *mARGOt*, we selected real-world applications taken from three completely different application domains: image processing, computation chemistry and a Monte Carlo approach. These domains are very important in the context of embedded systems and High-Performance Computing. Moreover, the three use cases have also been selected to validate the different features of the framework. In particular, in Section 5.2 we assess the reactive behaviour, in Section 5.3 the online learning module, finally in Section 5.4 the proactive behaviour by using the input features.

Given the flexibility of *mARGOt*, we deployed it on different platforms ranging from embedded to HPC. As a representative embedded platform, we used a Raspberry Pi (R) 3 model B. The board has a quad-core ARMv7 (R) (@ 1.2 Ghz) CPU with 1 GB of memory. To represent a typical HPC node, we used a platform composed of two Intel(R) Xeon(R) CPU E5-2630 v3 (@ 2.40GHz) with 128 GB of memory with dual channel configuration (@1866 MHz). All the experiments described in this Section use the Intel platform, except the ones related to *Stereomatching* (Section 5.2) based on the ARM platform.

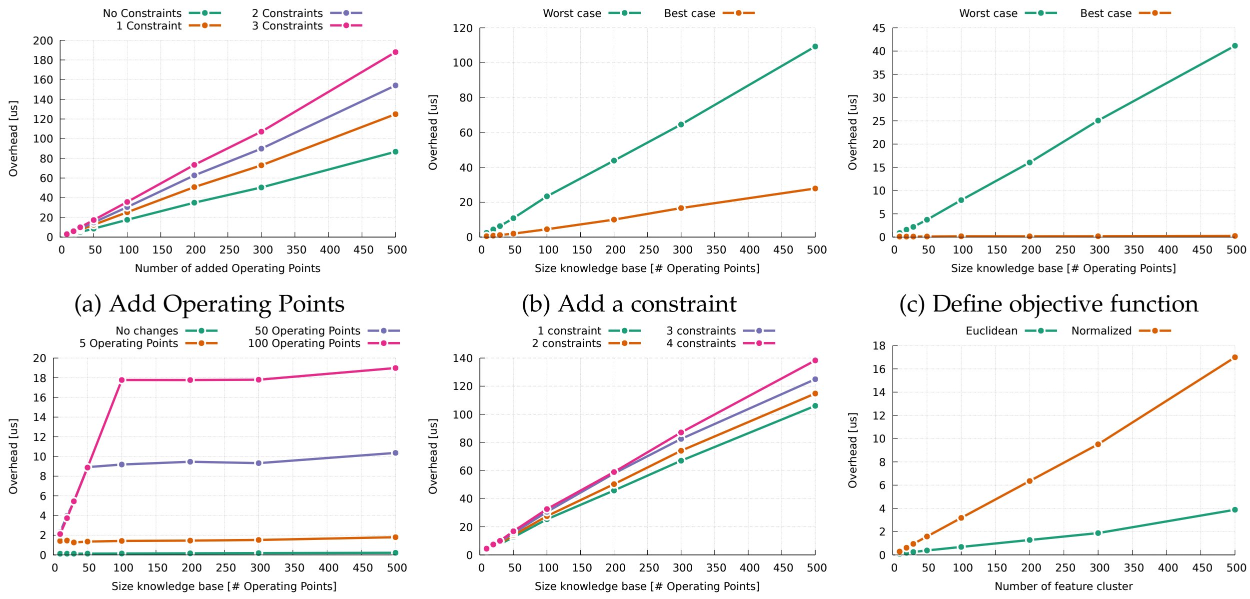
5.1 Overhead Evaluation

The proposed framework enables application developers to introduce the adaptation layer by instrumenting the source code using a C++ library, that executes synchronously with the application. Therefore, the time spent by the *mARGOt* library to select a new configuration, to change the knowledge base, or to update the internal structures that represent application requirements can be considered as an overhead introduced to the target application.

This experiment is focused on evaluating the overheads introduced by *mARGOt* in the most significant operations exposed to application developers. Instead of providing a single value, in this experiment, we increase the problem complexity to show the trend of the overheads. Before discussing the results, it is important to remember that the *mARGOt* implementation follows a differential approach to solve the optimization problem efficiently. Even if the worst-case complexity of the algorithm is the same, it reduces the complexity of the average- and best-case scenarios.

Figure 7 shows the introduced overheads by varying the size of the application-knowledge or the input feature clusters across the evaluated operations. In particular, Figure 7a shows the overhead for introducing Operating Points in the application-knowledge by varying their number. Given that each constraint uses a dedicated “view” over the OPs, the introduced overhead also depends on their number. Figure 7b shows the overhead for introducing a new constraint in the optimization problem. The overhead depends on how many OPs are admissible for the new constraint. Even when no OPs are admissible, the introduced overhead is due to the building of a dedicated “view”, which involves all the OPs in the knowledge base. Figure 7c shows the overhead of defining a new objective function for the problem. In this case, the overhead depends on the number of OPs that satisfy all the constraints of the optimization problem. Figure 7d and 7e show the overhead of solving the optimization problem by inspection. While the previous operation might be considered as an initialization cost, this overhead is paid each time the application enters in the managed region of code. As shown in Figure 7d, the introduced overheads depend only on the number of OPs involved in the change with respect to the previous time that the optimization problem was solved. Figure 7e shows the introduced overhead in the worst-case scenario, which is not only due to the fact that all the Operating Points are involved in the change, but it takes into consideration also the solver algorithm, by using a knowledge base to stress the implementation. This means that all the OPs have the same value for the metrics related to the constraints and to the objective function. Figure 7f shows the overhead of selecting the closest feature cluster of the current input, where the feature vector is composed of three values. Even this overhead is paid each time the application enters in the managed region of code and it shall be added to the overhead of solving the optimization problem.

It is important to notice how the overheads measured in these experiments must be related to the execution time of each iteration of the target application. If needed, the *mARGOt* activation period can be tuned to maintain the overhead below a given threshold. For the applications



(a) Add Operating Points (b) Add a constraint (c) Define objective function

(d) Find best configuration (flat) (e) Find best configuration (scaling) (f) Select features cluster

Fig. 7: Evaluation of the overheads introduced by *mARGOt* at runtime

discussed in the following sections, the introduced overhead is less than 1%.

5.2 *Stereomatching* Application

The first use case targets the *Stereomatching* application, that computes a disparity map of a scene captured by a stereo camera. The output of this application is required for estimating the depth of the objects in the scene. In this use case, a smart camera is deployed either on a drone or on a battery-powered surveillance system.

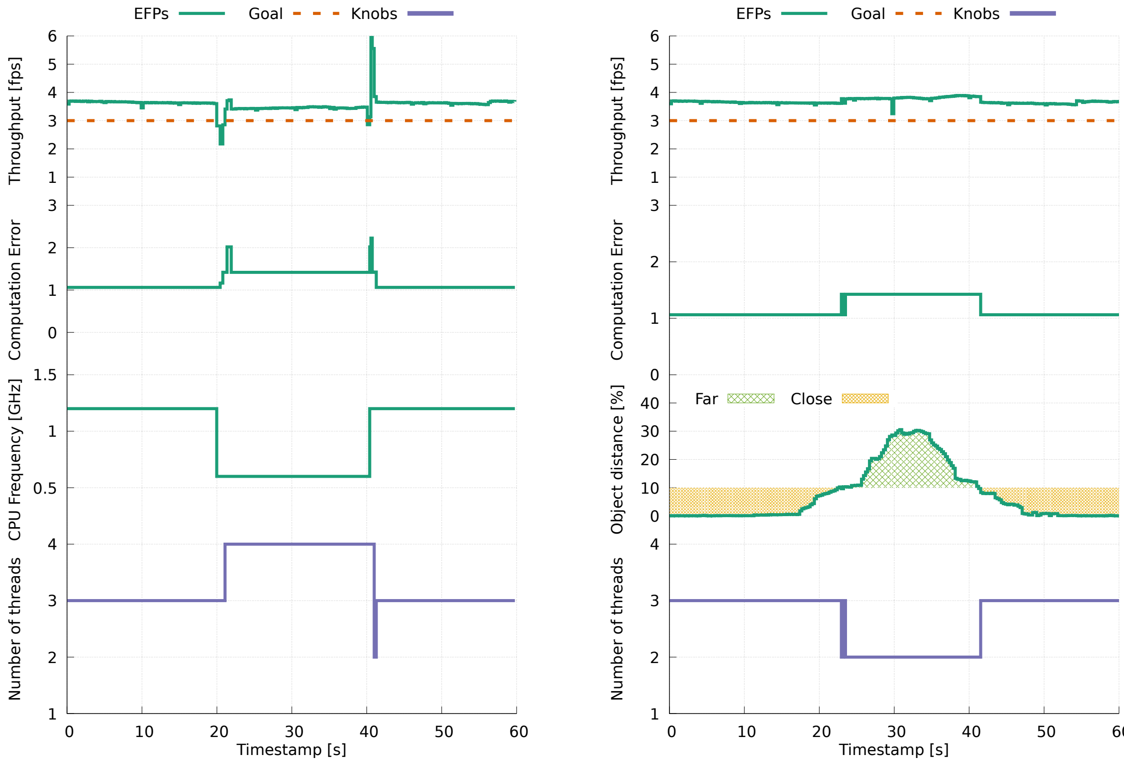
The algorithm derived by [64] builds adaptive-shape support regions for each pixel of an image, based on colour similarity, and then it tries to match them on the other image, computing its disparity value. The algorithm implementation [3] exposes five application-specific knobs to modify the effort spent on building the support regions and on matching them in the second image to trade off the accuracy of the disparity image (the output of the *Stereomatching*) and the execution time (and thus the reachable application throughput). The accuracy metric is the *disparity error*, defined as the average intensity difference per pixel, in percentage, between the computed output and the reference output. The application has been parallelized by using OpenMP, making available as sixth software-knob the number of threads used for the computation.

This use case has been chosen to assess the benefits of using reaction mechanisms provided by *mARGOt* in terms of changes of application requirements and knowledge.

The end-user does not require the application to sustain the throughput of the input video stream, but he/she requires that the application must reach a minimum throughput for detecting the position and depth of the objects in the scene. In this use case, we set this high priority constraint to $3fps$. On top of this constraint, we envisioned two different application requirements according to the scene observed from the stereo camera. First, if in the previous scene there is no object close to the camera, the objective function minimizes the disparity error with an additional low-priority constraint for executing the application by using a single software thread. Second, if there are objects close to the camera, the objective function minimizes the geometric mean between the disparity error and the number of software threads, without any other constraint except the one on the throughput. The philosophy behind these two *states* is that in the first one we try to execute in a "low-power" mode, because there is nothing interesting in the scene, while in the second *state* we focus on the output quality, without forgetting that the smart camera is placed on a battery-powered device.

To demonstrate the adaptivity added to the *Stereomatching* application, we focused on two different scenarios as shown in Figure 8a and 8b. The first scenario (Figure 8a) shows how the feedback information from the monitors triggers the adaptation reacting to a change of the application performance. The second scenario (Figure 8b) shows the benefits of reacting to changes in the application requirements (such as switching from one *state* to the other) according to the system evolution. Figure 8 shows the results of these experiments, while Figure 9 reports the application-knowledge (i.e. the Pareto-optimal Operating Points). For clarity reasons, in Figure 8, we omitted to report the software-knobs that are not relevant for the experiment.

In the first scenario (Figure 8a), we execute *Stereomatching* for $60s$. After $20s$, we reduce the frequency of the platform cores by using the CPUfreq framework, for example to simulate the effect of a power capping due to thermal reasons, and then we restore the original frequency of the cores after $20s$. The whole experiment is executed under the assumption that there is an object close to the camera. Figure 8a shows the execution trace of this experiment in terms of CPU frequency, number of threads, computation error and throughput.



(a) Reaction to a throughput degradation

(b) Reaction to a change in the requirements

Fig. 8: Execution trace of *Stereomatching* in an embedded platform. The x-axis shows the timestamp of the experiment, while the y-axes show extra-functional properties of the system and the number of software-threads.

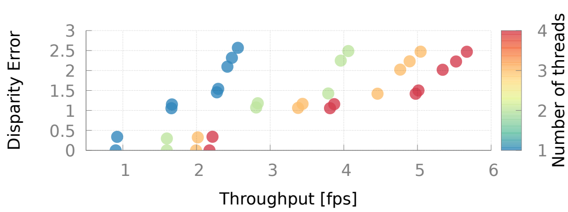


Fig. 9: Application-knowledge of the *Stereomatching* application. Each circle represents an Operating Point. The x-axis represents the expected average throughput, the y-axis the expected average error, and the color range the parallelism level.

At the beginning of the experiment, *mARGOt* selects among the configurations that satisfy the constraint on the throughput, the one that minimizes the error and resource usage. When we reduce the frequency of the cores, the throughput monitor observes a degradation on the performance with respect to the expected one, triggering the adaptation. In particular, *mARGOt* chooses among the valid configurations, the one that minimizes the objective function, while providing the requested throughput adjusted by the measured degradation. When we restore the original frequency, the throughput monitor observes a performance improvement and triggers the second adaptation. Given that we restored the original condition, the selected configuration is the same as the initial one.

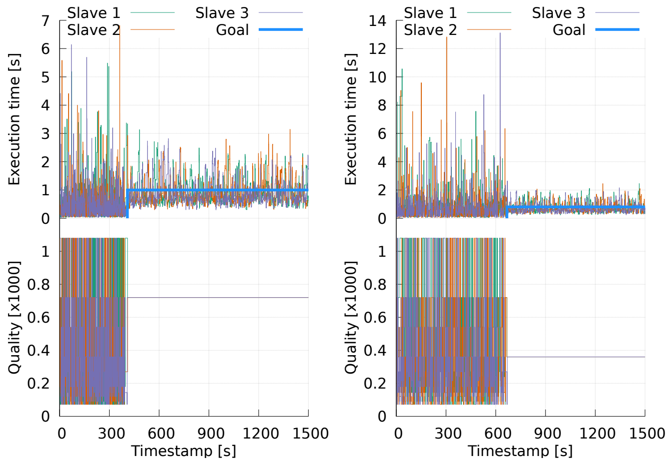
In the second scenario (Figure 8b), we processed a video stream captured from the stereo camera, while it slowly moves from one close object (from 0s to around 20s) to another one (around 40s to 60s). During the transition between the two objects, there is a period (around 20s to 40s) where there is no object close to the camera. Figure 8b shows the execution trace of this experiment in terms of measured object distance, number of threads, computation error and throughput.

At the beginning and at the end of the experiment, when there is an object close to the camera, the configuration selected by *mARGOt* is the same used to start the previous experiment (the conditions are the same). However, when at time $22s$ there are no more objects close to the camera, *mARGOt* switches to a more power safe state, which introduces the constraint on a single thread execution. From the knowledge base (see Figure 9), we notice that on this platform there is no configuration reaching a throughput of $3fps$ by using a single thread. For this reason, *mARGOt* automatically relaxes the lower priority constraint, selecting the configuration which is closest to satisfy it, i.e. using two threads. Among the software-knob configurations that use two threads, *mARGOt* selects the one that minimizes the objective function.

In this use case, the overhead introduced by *mARGOt* is always less than 0.1% of the application execution time.

5.3 *GeoDock* Application

The second use case is given by a docking application running on HPC resources. In the context of a drug discovery



(a) *Small pocket* (*1b9v*) (b) *Large pocket* (*1cvu*)

Fig. 10: Execution trace of *GeoDock* learning phase with the same ligand library, but using different target pockets. Each trace shows the execution time and the quality of the elaboration.

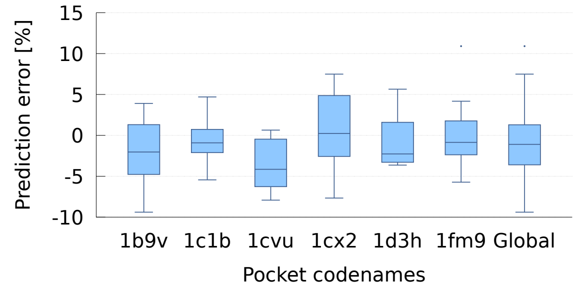


Fig. 11: The distribution of the prediction error on the time to solution by considering different pockets.

process, the molecular docking task aims at estimating the 3-dimensional pose of a molecule, named *ligand*, after the interaction with the target binding site of a second molecule, named *pocket*. Molecular docking is employed in the early stages of the drug discovery process for the virtual screening of a huge library of ligands, to find the ligands with the strongest interaction with the target pocket.

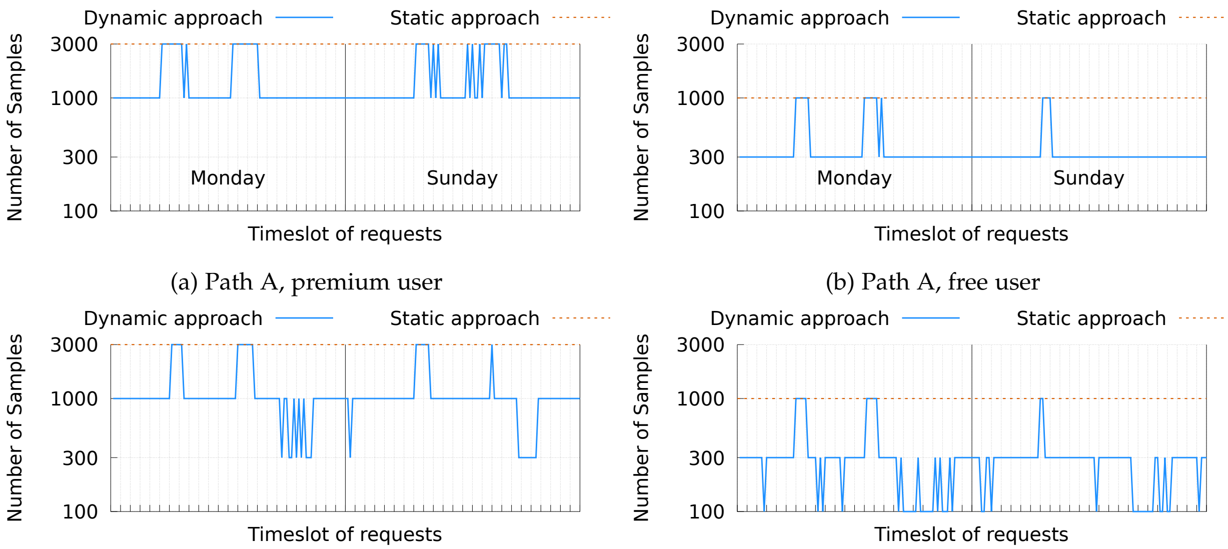
GeoDock is part of the LiGenDock application [65] and it performs a fast estimation of the ligand pose by using only geometrical information, to prune the ones that are unable to fit in the target pocket. A subsequent module of LiGenDock performs the actual simulation of chemical and physical interactions to obtain an accurate pose estimation, forwarding to the next stages of the process only the most promising ligands. The complexity of the problem is not only due to the number of ligands to evaluate but also to a large number of degrees of freedom involved in the docking of a ligand in the target pocket. To deal with this complexity, *GeoDock* implements an iterative greedy algorithm. This application exposes two software-knobs that approximate the elaboration by increasing the granularity of the pose optimization process.

Concerning the extra-functional properties, we have to consider that the typical end-user is a pharmaceutical company that runs experiments on an HPC platform. Therefore, the metrics of interest are the time-to-solution (which is directly related to the cost of the computation resources) and the quality of the elaboration. Given that the *GeoDock* purpose is only to prune the ligands that are incompatible with the target pocket, the considered quality metric associated to a configuration is measured by considering the number of evaluated poses for each ligand. This application represents a typical batch job, where the end-user company rents a pre-allocated set of resources from an HPC centre and it would like to maximize the elaboration quality given the time budget reserved for the job. Given that the later stages of the drug discovery process require a monetary effort to perform tests *in-vivo*, the reproducibility of the experiment really matters. Therefore, once *mARGOt* has selected the most suitable configuration to perform the experiment, we are not allowed to adapt anymore.

In this scenario, the goal of *mARGOt* is not only to tune the application according to its requirements (maximizing the quality, while satisfying a fixed time-to-solution) but also to show the benefits of learning at runtime the application-knowledge. The introduced overhead is negligible because *mARGOt* tunes the application only once, at the beginning of the screening process.

Figure 10 shows the initial $1500s$ of the execution trace of *GeoDock* with a "small" (Figure 10a) and a "large" pocket (Figure 10b) by considering the same constraint on the time-to-solution. We use a library of $113k$ ligands, different in terms of the number of atoms (between 28 and 153) and internal degree of freedoms (between 4 and 106). While on the x-axis we report the time passed from the beginning of the experiment, on the y-axis we show the execution time to evaluate a single ligand-pocket pair and the reached quality of the elaboration. In both cases, in a first phase (up to around 400s and 700s respectively considering the *small* and *large* pocket) the application exploits the *mARGOt* remote DSE framework to learn for each configuration the average ligand-pocket docking time by randomly sampling the target ligand library and exploiting the parallelism of the HPC resources. In a second phase, the application restarts to evaluate the library with a configuration selected by *mARGOt* according to the application requirements (the remaining time) and the online gathered application-knowledge. From the results, we notice how the constraint on the execution time is similar for both pockets, while the quality is higher for the *small* pocket and lower for the *large* pocket.

To validate the quality of the model learned by *mARGOt*, we used six pockets (*1b9v*, *1c1b*, *1cvu*, *1cx2*, *1dh3* and *1fm9*) derived from the RCSB Protein Databank (PDB) [66], in PASS version [67] and the same ligand library used in the previous experiment. Figure 11 shows the distribution of the prediction error by considering different pockets. For each pocket, we execute 10 experiments by using a library of 4000 ligands randomly selected from the full library. To compute the application-knowledge, each configuration of the DoE is evaluated by using 200 ligands. We predict the time to solution as the average execution time to elaborate a single ligand multiplied by the dimension of the ligand library and divided by the number of slaves. Despite the large variability in terms of ligands size and the small learning



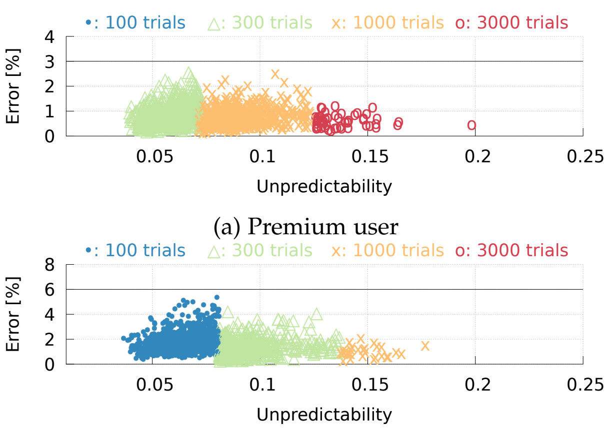
(a) Path A, premium user

(b) Path A, free user

(c) Path B, premium user

(d) Path B, free user

Fig. 12: Number of samples used by the adaptive PTDR application by changing the starting time of the request.



(a) Premium user

(b) Free user

Fig. 13: Validation campaign of the dynamic approach, where each dot is a *PTDR* request evaluation. The x-axis represents the observed unpredictability, while the y-axis the observed error. Color and shape of each dot represents the number of samples.

set, the prediction error reported in Figure 11 is kept limited (<10%) providing to *mARGOt* the possibility to increase the computation efficiency.

5.4 Probabilistic Time-dependent Routing Application

The third use case represents the tuning of a Monte Carlo simulation used to estimate the travel time distribution in a processing pipeline for a car navigation system. In smart cities, traffic prediction and cooperative routing are examples of activities to ease the life of citizens. In particular, the *Probabilistic Time-Dependent Routing* (*PTDR*) algorithm [68] is a crucial component of a cooperative routing task to compute the estimated travel time distribution. Then, later stages of the navigation system leverage this information to select the best solution among different routes.

To generate this output, *PTDR* must first estimate the travel time distribution and then extract statistical properties to be forwarded to the later stages of the navigation system. Each trial of the Monte Carlo simulates an independent route traversal over an annotated graph in terms of speed profiles. Given a sufficient number of trials, the sampled distribution of travel times will asymptotically converge towards the real distribution. Using this distribution, the application derives the statistical property of interest (such as the average or the 3rd quartile), which represents the actual output of the application.

The application is designed and already optimized to leverage the resources of the target HPC platform [69] and exposes as software-knobs, the number of Monte Carlo samples to be used to compute the output. The error metric is defined as the difference between the value extracted with a limited number of samples and the one extracted with a very large (theoretically infinite) number of samples (we used 1M samples). Moreover, as defined in [69], we can differentiate among paths with a large or narrow distribution of speed profiles, resulting respectively less or more predictable in terms of travel time estimation. We call this feature, that can be easily extracted before running the *PTDR*, *unpredictability* and we use it as data-feature to be provided to *mARGOt* for selecting the right software-knob configuration for each simulation.

In terms of application requirements, the end-user would like to minimize the number of samples used to compute the output, with a limit on the error upper bound. This use case has been selected to demonstrate the benefits of using *mARGOt* in a proactive fashion, to tune the number of trials according to the current input. Without dynamic adaptation, the end-user should find the minimum number of samples that leads to a satisfying computation error for the worst case scenario. Moreover, end-user would like to differentiate the threshold on the computation error constraint, according to whether the request is generated from a *premium user* (error $< 3\%$) or a *free user* (error $< 6\%$).

Before running the application, we performed an experimental campaign by using random requests from 300 paths in the Czech Republic [68], in different moments of the week, to build the application-knowledge. Moreover, we limited the software-knob values to $[100, 300, 1000, 3000]$ according to the previous analysis of the application [68]. Furthermore, to increase the robustness of the approach we consider three times the standard deviation of a software-knob configuration for the constraint on the computation error.

Figure 12 shows the selected number of samples in an experiment that generates four types of requests every $15min$ on two days of the week, Monday and Sunday. In particular, for each type of user, we consider two different paths. Figure 12c and 12a shows the results for the premium user, while Figure 12d and 12b shows the results for the free user. On one hand, this experiment shows how changing the application requirements (premium and free users) decreases the number of samples used to satisfy the request, considering both static and dynamic approaches. On the other hand, this experiment shows how using input features (dynamic approach) decreases the number of samples with respect to using a single conservative configuration (static approach). This is due to different path characteristics, defined by their *unpredictability*. For example, countryside requests are more predictable than those coming from an urban area. In this experiment, the proposed approach easily implemented by using *mARGOt* uses approximately the 30% of the number of samples of a static approach, with an overhead comparable of computing 2 samples.

The second experiment focuses on validating the dynamic approach. Figure 13 shows the computation error on 1500 requests from routes of the Czech Republic with different starting time and considering different types of users (premium user in Figure 13a and free user in Figure 13b). The x-axis represents the extracted input feature, while the y-axis represents the observed error. Each dot in the plot represents a request and their shape and colour highlight the chosen number of samples by the dynamic approach. The plot makes easy the identification of the switching points among the knob configurations according to the input feature (e.g. 0.07 and 0.125 for Figure 13a and 0.07 and 1.15 for Figure 13b). The results show how by using *mARGOt* all the requests have been satisfied with the target error level by using fewer samples than a static approach leveraging on the input features.

In this use case, the overhead introduced by *mARGOt* is around 1% of the smallest number of samples for the Monte Carlo simulation.

6 Conclusions

In this article, we propose *mARGOt*, a dynamic autotuning framework to enhance an application with an adaptation layer. In particular, the application end-user specifies high-level goals and *mARGOt* provides to the application the most suitable configuration of the software-knobs leveraging on the application-knowledge. Moreover, *mARGOt* provides mechanisms to adapt in a reactive and proactive way by identifying and seizing optimization opportunities at the runtime. It is also possible to avoid the Design Space Exploration by learning the application-knowledge online. Due to its flexibility, *mARGOt* can be successfully applied to a wide range of applications domains from embedded to HPC. In this work, we have shown the benefits of dynamic adaptation in three different application domains. Experimental results have shown how *mARGOt* reacts to changes in the execution environment and in the application requirements, while leveraging on input features for seizing optimization opportunities for the actual inputs. Moreover, *mARGOt* might learn the application-knowledge at runtime, for capturing complex relations between the software-knobs, the actual input set and the metrics of interest. Finally, the *mARGOt* framework is released as open-source [10] along with user manuals and doxygen documentation.

Acknowledgments

The authors would like to thank Kateřina Slaninová, Jan Martinovič and Martin Golasowski from IT4Innovations for their support for the *PTDR* application. The authors would like to thank Andrea Beccari and Candida Manelfi from Research and Innovation, Dompé Farmaceutici and Nico Sanna and Carlo Cavazzoni from SuperComputing Applications and Innovation, CINECA for their support for the *GeoDock* application.

References

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[2] M. Duranton, K. De Bosschere, C. Gamrat, J. Maebe, H. Munk, and O. Zendra, "The HiPEAC Vision 2017," 2017.

[3] E. Paone, G. Palermo, V. Zaccaria, C. Silvano, D. Melpignano, G. Haugou, and T. Lepley, "An exploration methodology for a customizable opencl stereo-matching application targeted to an industrial multi-cluster architecture," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2012, pp. 503–512.

[4] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," 2009.

[5] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 324–334.

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[7] D. Gadioli, S. Libutti, G. Massari, E. Paone, M. Scandale, P. Bellasi, G. Palermo, V. Zaccaria, G. Agosta, W. Fornaciari *et al.*, "Opencl application auto-tuning and run-time resource management for multi-core platforms," in *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 127–133.

[8] E. Paone, D. Gadioli, G. Palermo, V. Zaccaria, and C. Silvano, "Evaluating orthogonality between application auto-tuning and run-time resource management for adaptive opencl applications," in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*. IEEE, 2014, pp. 161–168.

[9] D. Gadioli, G. Palermo, and C. Silvano, "Application autotuning to support runtime adaptivity in multicore architectures," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE, 2015, pp. 173–180.

[10] "mARGOt framework git repository," https://gitlab.com/margot_project/core, 20018.

[11] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[12] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "Towards architecture-based self-healing systems," in *Proceedings of the first workshop on Self-healing systems*. ACM, 2002, pp. 21–26.

[13] R. Ananthanarayanan, M. Mohania, and A. Gupta, "Management of conflicting obligations in self-protecting policy-based systems," in *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*. IEEE, 2005, pp. 274–285.

[14] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, "ABLE: A toolkit for building multiagent autonomic systems," *IBM Systems Journal*, vol. 41, no. 3, pp. 350–371, 2002.

[15] M. C. Huebscher and J. A. McCann, "A survey of autonomic computingdegrees, models, and applications," *ACM Computing Surveys (CSUR)*, vol. 40, no. 3, p. 7, 2008.

[16] S. Mahdavi-Hezavehi, V. H. Durelli, D. Weyns, and P. Avgeriou, "A systematic literature review on methods that handle multiple quality attributes in architecture-based self-adaptive systems," *Information and Software Technology*, vol. 90, pp. 1–26, 2017.

[17] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, 2004, pp. 70–77.

[18] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*. IEEE, 2006, pp. 65–73.

[19] D. Abramson, R. Buyya, and J. Giddy, "A computational economy for grid computing and its implementation in the Nimrod-G resource broker," *Future Generation Computer Systems*, vol. 18, no. 8, pp. 1061–1074, 2002.

[20] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: a general and extensible framework for self-aware computing," 2011.

[21] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moretó, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor *et al.*, "Tessellation: refactoring the os around explicit resource containers with continuous adaptation," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 76.

[22] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau, "Resource-aware programming and simulation of MPSoC architectures through extension of X10," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*. ACM, 2011, pp. 48–55.

[23] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSAadaptive quality of service architecture," *Software: Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[24] S. Fang, Z. Du, Y. Fang, Y. Huang, Y. Chen, L. Eeckhout, O. Temam, H. Li, Y. Chen, and C. Wu, "Performance portability across heterogeneous socs using a generalized library-based approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 2, p. 21, 2014.

[25] C. Ţăpuş, I.-H. Chung, J. K. Hollingsworth *et al.*, "Active harmony: Towards automated performance tuning," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2002, pp. 1–11.

[26] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-IL: An instrumentation language for auto-tuning parallel applications," in *European Conference on Parallel Processing*. Springer, 2009, pp. 9–20.

[27] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin *et al.*, "Autotune: A plugin-driven approach to the automatic tuning of parallel applications," in *International Workshop on Applied Parallel Computing*. Springer, 2012, pp. 328–342.

[28] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*. IEEE, 2014, pp. 303–315.

[29] A. Rasch, M. Haidl, and S. Gorlatch, "ATF: A Generic Auto-Tuning Framework," in *High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2017 IEEE 19th International Conference on*. IEEE, 2017, pp. 64–71.

[30] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing sequential programs with statistical accuracy tests," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 2s, p. 88, 2013.

[31] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 35–50, 2014.

[32] J. Dorn, J. Lacomis, W. Weimer, and S. Forrest, "Automatically exploring tradeoffs between software output fidelity and energy costs," *IEEE Transactions on Software Engineering*, 2017.

[33] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998, pp. 1–27.

[34] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[35] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[36] M. Püschel, J. M. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *The International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, 2004.

[37] C. Nugteren and V. Codreanu, "CLTune: A generic auto-tuner for OpenCL kernels," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*. IEEE, 2015, pp. 195–202.

[38] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011, pp. 676–687.

[39] S. A. Kamil, *Productive high performance parallel programming with auto-tuned domain-specific embedded languages*. University of California, Berkeley, 2012.

[40] M. J. Voss and R. Eigenmann, "ADAPT: Automated de-coupled adaptive program transformation," in *Parallel Processing, 2000. Proceedings. 2000 International Conference on*. IEEE, 2000, pp. 163–170.

[41] H. Guo, "A bayesian approach for automatic algorithm selection," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI03), Workshop on AI and Autonomic Computing, Acapulco, Mexico*, 2003, pp. 1–5.

[42] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 25–34.

[43] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[44] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Microarchitecture (MICRO), 2013 46th Annual IEEE/ACM International Symposium on*. IEEE, 2013, pp. 13–24.

[45] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 198–209.

[46] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *ACM SIGPLAN Notices*, vol. 46, no. 3. ACM, 2011, pp. 199–212.

[47] A. Filieri, H. Hoffmann, and M. Maggio, "Automated multi-objective control for self-adaptive software design," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 13–24.

[48] X. Sui, A. Lenharth, D. S. Fussell, and K. Pingali, "Proactive control of approximate programs," *ACM SIGOPS Operating Systems Review*, vol. 50, no. 2, pp. 607–621, 2016.

[49] L. G. Valiant, "A theory of the learnable," *Communications of the ACM*, vol. 27, no. 11, pp. 1134–1142, 1984.

[50] J. S. Miguel and N. E. Jerger, "The anytime automaton," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 545–557.

[51] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: using canary inputs to dynamically steer approximation," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 161–176, 2016.

[52] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, *PetaBricks: a language and compiler for algorithmic choice*. ACM, 2009, vol. 44, no. 6.

[53] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and compiler support for auto-tuning variable-accuracy algorithms," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, pp. 85–96.

[54] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. OReilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 379–390.

[55] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe, "Siblingrivalry: online autotuning through local competitions," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2012, pp. 91–100.

[56] G. Palermo, C. Silvano, and V. Zaccaria, "ReSPIR: a response surface-based Pareto iterative refinement for application-specific design space exploration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1816–1829, 2009.

[57] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–9.

[58] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can search algorithms save large-scale automatic performance tuning?" *Procedia Computer Science*, vol. 4, pp. 2136 – 2145, 2011, proceedings of the International Conference on Computational Science, ICCS 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050911002924

[59] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*. Springer, 2005, pp. 257–269.

[60] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[61] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 185–194.

[62] Z. Nie and J. S. Racine, "The crs package: Nonparametric regression splines for continuous and categorical predictors." *R Journal*, vol. 4, no. 2, 2012.

[63] V. Zaccaria, G. Palermo, F. Castro, C. Silvano, and G. Mariani, "Multicube explorer: An open source framework for design space exploration of chip multi-processors," in *Architecture of Computing Systems (ARCS), 2010 23rd International Conference on*. VDE, 2010, pp. 1–7.

[64] K. Zhang, J. Lu, and G. Lafruit, "Cross-based local stereo matching using orthogonal integral images," *IEEE transactions on circuits and systems for video technology*, vol. 19, no. 7, pp. 1073–1079, 2009.

[65] C. Beato, A. R. Beccari, C. Cavazzoni, S. Lorenzi, and G. Costantino, "Use of experimental design to optimize docking performance: The case of ligendock, the docking module of ligen, a new de novo design program," 2013.

[66] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne, "The protein data bank," *Nucleic Acids Res*, vol. 28, pp. 235–242, 2000.

[67] G. P. Brady and P. F. Stouten, "Fast prediction and visualization of protein binding pockets with pass," *Journal of computer-aided molecular design*, vol. 14, no. 4, pp. 383–401, 2000.

[68] R. Tomis, L. Rapant, J. Martinovič, K. Slaninová, and I. Vondrák, "Probabilistic time-dependent travel time computation using monte carlo simulation," in *International Conference on High Performance Computing in Science and Engineering*. Springer, 2015, pp. 161–170.

[69] M. Golasowski, R. Tomis, J. Martinovič, K. Slaninová, and L. Rapant, "Performance evaluation of probabilistic time-dependent travel time computation," in *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 2016, pp. 377–388.

Davide Gadioli received his Bachelor Degree in Computer Engineering in 2010 and his Master of Science in Computer Engineering in 2013 both from Politecnico di Milano (Italy). Since November 2014, he is a PhD Student at Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano. In 2015, he was a Visiting Student at IBM Research (The Netherlands). His main research interests are in application autotuning, autonomic computing and approximate computing.

Emanuele Vitali graduated in 2015 from Politecnico di Milano (Italy) after completing his Master of Science in Computer Engineering. Since then, he has been working at Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of Politecnico di Milano, first as a Research Fellow and then, since May 2017, as PhD Student. His main research interests include hardware architectures and application autotuning.

Gianluca Palermo received his Master of Science degree in Electronic Engineering, in 2002, and the PhD degree in Computer Engineering, in 2006, from Politecnico di Milano (Italy). He is currently an Associate Professor at the Department of Electronics, Information and Bioengineering (DEIB) at the same University. Previously, he was Consultant Engineer at the Low-Power Design Group of AST - STMicroelectronics working on Network-on-Chip, and Research Assistant at the Advanced Learning and Research Institute (ALaRI) of the Universita' della Svizzera Italiana (Switzerland). His research interests include design methodologies and architectures for embedded and HPC systems focusing on autotuning aspects. He is an active member of the scientific community serving in organizing and program committees of several conferences in his research areas. Since 2003, he published more than 100 scientific papers in peer-reviewed conferences and journals. He is member of IEEE, ACM and HiPEAC.

Cristina Silvano is a Professor of Computer Architectures at Politecnico di Milano, Italy. Her research interests are in energy-efficient embedded systems, design space exploration of manycore architectures and application autotuning for HPC. She has published more that 160 scientific papers in peer-reviewed journals and conferences, five books and she holds several patents in collaboration with Group Bull and STMicroelectronics. She was Project Coordinator of three European projects: H2020-ANTAREX, FP7-2PARMA and FP7-MULTICUBE. She has served in the organizing and program committees of several major conferences in computer architectures, embedded systems and electronic design automation. She is Associate Editor of ACM TACO. In 2017, she has been elevated to the grade of IEEE Fellow.