Chapter 8. Runtime Autotuning: the mArgot Approach

Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Cristina Silvano

Politecnico di Milano, Milano, Italy
{davide.gadioli, emanuele.vitali, gianluca.palermo, cristina.silvano}@polimi.it

Abstract

In the autonomic computing context, applications are perceived as autonomous agents that are able to adapt at runtime, according to the evolution of the system. The proposed framework aims at enhancing a target application with an adaptation layer, to provide self-optimization capabilities. In particular, *mARGOt* is a C++ library requiring a limited intrusiveness in the target application to identify the region of interest and the software knobs to be manipulated. The library is instantiated and customized according to extra-functional requirements of the application specified in a configuration file. *mARGOt* exploits design-time knowledge and multi-objective requirements expressed by the user, to drive the autotuning process at the runtime.

1 Introduction

The source code of an application describes the procedure to obtain the desired output. Beside the functional correctness of the result, end-user requirements involve extra-functional requirements [1]. For example, they may have a constraint on the time to solution or on a minimum accuracy level. However, extra-functional properties (EFPs) also depend on the underlying architecture, on the actual input, and on the system workload.

Hence introducing software-knobs, such as the parallelism level or the number of trials in a Monte-Carlo simulation, is a common practice to make the application tunable. The key idea is that a change in the software-knobs configuration leads to a change on the EFPs as well.

In this context, the main goal of the proposed dynamic autotuning framework, named *mARGOt*[2], is to select the most suitable configuration at runtime, according to application requirements, input data-features and observations of the actual application performance.

Self-managing software is an appealing idea widely explored in literature [3, 4]. Focusing on application autotuning, there are static and dynamic approaches. Static autotuning approaches select the best configuration, or the best strategy to select a configuration, at design-time or deploy-time. On the main hand, we have frameworks that aim at optimizing a specific class of kernels, for example: ATLAS [5] for matrix multiplication routines, FTTW [6] for FFTs operations or OSKI [7] for sparse matrix kernels. On the other hand, there are frameworks that are application agnostic, for example: Petabricks [8], OpenTuner [9], ATune-IL [10] and Active Harmony [11]. Dynamic autotuning approaches usually employ a two-phase strategy, where they obtain application knowledge at design or deploy time, while they adapt at runtime. *mARGOt* falls in this category. Examples of this category are PowerDial [12], Capri [13], Green [14] and IRA [15].

The *mARGOt* framework main contributions with respect to state-of-the-art dynamic autotuners can be summarized as follows:

1. it enables to express complex application requirements;
2. it is software-knobs agnostic;
3. it leverages reactive and proactive mechanisms to adapt at runtime;
4. it minimizes the target application intrusiveness, in terms of lines of code to be changed.

Moreover, it has been released as an Open Source framework[1].

2 Software description

The proposed framework is a C++ library that manages regions of code of the target application. We might abstract each region of interest as a function ($f(.)$) that produces an output (o), given an input (i) and a configuration of software-knobs ($\bar{x}$), i.e. $o = f(\bar{x}, i)$, characterized by a set of extra-functional properties ($\bar{p}_{f(\bar{x},i)}$). The goal of *mARGOt* is to find the software-knobs configuration $\bar{x}$ that satisfy the application requirements, as defined in Equation 1:

$$\begin{array}{rlccll} \max / \min & Q(\bar{p}_{f(\bar{x},i)}) & & & & \\ \text{s.t.} & C_1 : & \bar{p}_{f(\bar{x},i)} & \propto & k_1 & with \quad \alpha_1 \quad confidence \\ & C_2 : & \bar{p}_{f(\bar{x},i)} & \propto & k_2 & \\ & C_3 : & \bar{x} & \propto & k_3 & \end{array} \tag{1}$$

where Q is the objective function, defined as a combination of extra-functional properties of the given region. Each constraint C_i states the target extra-functional property or software-knob, the comparison function $\propto$ (i.e. $\geq, >, \leq, <$), and the threshold value k_i. If the target extra-functional property is a stochastic variable, end-user may specify a confidence value.

2.1 Software Architecture

The framework architecture implements the **M**onitor, **A**nalyze, **P**lan and **E**xecute loop, based on application **K**nowledge (MAPE-K) [3]. Figure 1 represents the framework overall structure, assuming that the application is composed of just one kernel. The left side of Figure 1 represents the original application source code that describes the functional behavior (i.e. $o = f(\bar{x}, i)$). The right side of the image represents the three main elements of the framework. The monitors might sense the actual application performance and system counters. The manager is in charge of solving the optimization problem by inspection, using application knowledge and features of the current input (if available). The knowledge is obtained at design time through a profiling phase. In literature, several tools are available to efficiently explore the alternative configurations to find Pareto-optimal solutions [16, 17]. The application knowledge, as well as the adaptivity requirements, are considered the inputs of the framework. In particular, the application knowledge is composed of a look-up table in terms of software-knobs configurations and expected extra-functional properties (metrics). Each line of the table represents a possible *Operating Point* (OP) of the application. While the value of a software knob is deterministic, it is possible to describe a metric as a stochastic variable with a mean and a standard deviation. In case of data features, additional columns are added to accommodate such values. Clustering techniques can be used to reduce the possible values of data features.

To ease the integration process, *mARGOt* includes a utility tool, named *mARGOt* HEEL, that generates from two XML configuration files a high-level interface tailored for the target

[1] `https://gitlab.com/margot_project`

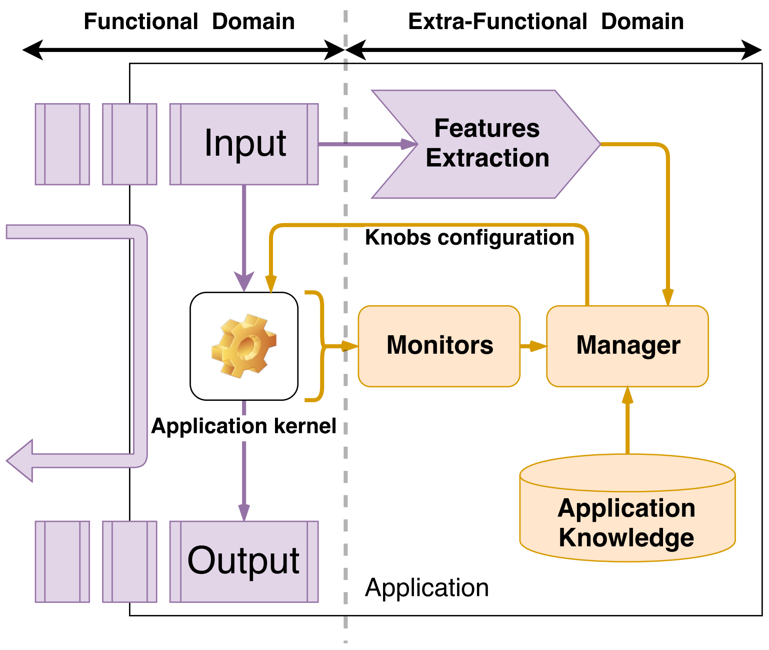


Figure 1: Architecture of the *mARGOt* dynamic autotuning framework. Purple elements represents application code, while orange elements represents framework elements.

application. The main configuration file defines extra-functional concerns, such as the application requirements or the metric of interest to be monitored. The second configuration file describes the application knowledge. *mARGOt* HEEL uses these files to generate an interface composed of five functions:

1. `init`: customize *mARGOt* according to configuration files. This is a global function that initialize the whole framework objects. This function is meant to be called just once in the application and initialize all the monitors, goals and managers. It may have only input parameters, depending on the constructor parameters of the monitors of interest.

2. `update`: find the best software-knob configuration. For each region of code managed by mARGOt, it is generated this function which interact with the manger to solve the optimization problem and fetch the new most suitable configuration of the software knobs. Each function is meant to be called before the start_monitor of each region of code tuned by mARGOt. It has as output parameter all the software knobs of the related region of code. It has as optional input parameter the features of the current input. The return value of this function is a boolean, which state if the selected configuration is different with respect to the previous one. Please note that whenever the configuration changes, the application should actuate the new configuration and notify the autotuner once the actuation is done. Moreover, if the user is interested only on monitor the application behavior, this function does not change the configuration taken in input and returns always false.

3. `start_monitors`: start the required observations. For each region of code managed by mARGOt, it is generated this function which starts all the monitor stated by the user. Each function is meant to be called before the start of each region of code tuned by mARGOt. It may have only input parameters, depending on the start parameters of the monitors of interest. If there are no monitors of interest, this function becomes optional.

4. `stop_monitors`: stop the required observations. For each region of code managed by mARGOt, it is generated this function which stops all the monitor stated by the user. Each function is meant to be called after the end of each region of code tuned by mARGOt. It may have only input parameters, depending on the stop parameters of the monitors of interest. If there are no monitors of interest, this function becomes optional.

5. `log`: optional, it generates a log file. For each region of code managed by mARGOt, it is generated this function which logs on the standard output and on a log file information regarding observed metrics (if any), the goals value (if any), the expected behavior of the application (if there is the application knowledge) and the selected configuration (if the region of code is tuned and there is an application knowledge). Each function is meant to be called after the stop_monitor of each region of code tuned by mARGOt. This function is always without any parameter and it is optional, useful only for tracing the behavior of the tuned region of code.

The `init` function shall be called once and it initializes the framework. All the other functions are paired to a single region of code managed by *mARGOt*. In particular, `start_monitors` and `stop_monitors` wraps the region of interest and they are used for observing the performance. The function `update` retrieves the software-knobs configuration for the current elaboration. All the *mARGOt* implementation details, the XML syntax and the full semantics of the high-level interface are described in the user manuals available in the repository [2]. An illustrative example is described in Section 3.

2.2 Software Functionalities

The main objective of *mARGOt* is to enhance an application with an adaptation layer. On one hand, it must be as flexible as possible to interact with different classes of applications. On the other hand, it must provide a mechanism to adapt at runtime with negligible overheads.

We designed the framework to be application agnostic, which means that application developers are able to specify the OP geometry according to their software-knobs and to their metrics of interest. Moreover, it is possible to cluster OPs according to input features, defined as an n-ary array of values. Finally, the application knowledge, the input features clusters and the application requirements may be changed at runtime.

To react to changes in the execution environment, it is possible to associate a monitor with a metric of the OP. If there is a difference between the observed performance and its expected value, *mARGOt* consider a linear error propagation to take adaptation decisions. For example, suppose that we are considering a video surveillance application where we target the maximization of the output quality, provided that we achieve a throughput of at least $1.5fps$. Let's consider that in the selected software-knobs configuration we expect a mean throughput of $2fps$. If we observe a mean throughput of $1fps$, *mARGOt* detects the performance mismatch and assumes that we have the same degradation in all the other OPs. Therefore, it will then adjust its selection towards a configuration that leads to a throughput of at least $3fps$ to achieve the goal.

Moreover, *mARGOt* uses input features to adapt in a proactive fashion. In particular, it uses features of the actual input to select the closest application knowledge cluster to solve the optimization problem. It is possible to select between two distance types: the Euclidean one and a normalized one. The latter is useful if numerical values of the input features components

[2]`https://gitlab.com/margot_project/core`

```
 1 int main()
 2 {
 3   // initialization code
 4   // swap_array_mask − vector with the id of the available swaptions
 5
 6   // main loop of the application
 7   while (std::chrono::steady_clock::now() < time_limit)
 8   {
 9     // mix the swaptions array mask, to have random inputs
10     std::shuffle(swap_array_mask.begin(), swap_array_mask.end(), generator);
11     std::cout << "\tRunning␣iteration␣[" << iteration_counter << "]" << std::endl;
12
13     // initialize the error to zero
14     error = 0;
15
16     // do the computation
17     for (int i = 0; i < num_threads; i++)
18     {
19       threadsID[i] = swap_array_mask[i];
20       pthread_create(&threads[i], &pthread_custom_attr, worker, &threadsID[i]);
21     }
22
23     // join the threads
24     for (int i = 0; i < num_threads; i++)
25     {
26       pthread_join(threads[i], NULL);
27     }
28
29     // compute the average error
30     error /= static_cast<float>(num_threads);
31     iteration_counter++;
32   }
33 }
```

Figure 2: The significant C++ code of the *swaptions* application.

differ significantly. Moreover, it is possible to express constraints on the selection of input features.

To summarize, *mARGOt* might select a new configuration in case of changes on the application requirements, application knowledge, current input features or if the observed performance is different with respect to the observed one.

3 Illustrative Examples

In this section, we show how it is possible to integrate *mARGOt* in the *swaptions* application, from Parsec benchmark suite [18]. It aims at pricing a stream of swaptions, using a Monte-Carlo approach. In this example[3], we used a portfolio of 128 swaptions. The main loop of the application (lines 9-31), spawns threads to price swaptions (lines 17-21). To emulate a random stream, we shuffle the swaptions portfolio (line 10). The output of the application is the mean value of a swaption with the related uncertainty.

In this example, we aim at tuning the number of samples used in the Monte-Carlo simulation. In particular, we are interested in minimizing the computation error ϵ, given a lower bound on the throughput. We define the computational error ϵ as ratio between the uncertainty of the selected configuration and the uncertainty of the reference configuration with 1000000 trials [12].

To achieve this result, the application developer needs to collect the application knowledge by means of a profiling phase and to write a configuration file including the application require-

[3] `https://gitlab.com/margot_project/applications`

```
<points version=”1.3” block=”pricing”>
  <point>
    <parameters>
      <parameter name=”num_trials” value=”100000”/>
    </parameters>
    <system_metrics>
      <system_metric name=”error” value=”212.71” standard_dev=”6.69”/>
      <system_metric name=”throughput” value=”28.81” standard_dev=”0.45”/>
    </system_metrics>
  </point>
  <!-- Other Operating Points -->
</points>
```

(a) Application knowledge configuration file

```
<margot application=”swaptions” version=”v1”>
  <block name=”pricing”>
    <!-- MONITOR SECTION -->
    <monitor name=”thr_monitor” type=”Throughput”>
      <stop>
        <param name=”num_swaptions_elaborated”>
          <local_var name=”num_threads” type=”int” />
        </param>
      </stop>
      <expose var_name=”avg_throughput” what=”average” />
    </monitor>
    <monitor name=”error_monitor” type=”Custom”>
      <spec>
        <header reference=”margot/monitor.hpp” />
        <class name=”margot::Monitor&lt;long_double&gt;” />
        <type name=”long_double” />
        <stop_method name=”push” />
      </spec>
      <stop>
        <param name=”error”>
          <local_var name=”error” type=”long_double” />
        </param>
      </stop>
      <expose var_name=”avg_error” what=”average” />
    </monitor>
    <!-- OPERATING POINT GEOMETRY -->
    <knob name=”num_trials” var_name=”num_trials” var_type=”int”/>
    <metric name=”throughput” type=”float” distribution=”yes”/>
    <metric name=”error” type=”float” distribution=”yes”/>
    <!-- ADAPTATION SECTION -->
    <goal name=”thr_goal” metric_name=”throughput” cFun=”GE” value=”2” />
    <adapt metric_name=”throughput” using=”thr_monitor” inertia=”3” />
    <state name=”normal” starting=”yes” >
      <minimize combination=”simple”>
        <metric name=”error” coef=”1.0”/>
      </minimize>
      <subject to=”thr_goal” confidence=”1” priority=”10” />
    </state>
  </block>
</margot>
```

(b) Application requirement configuration file

Figure 3: Example of XML configuration files handled by *mARGOt* HEEL.

ments. Figure 3 shows the application knowledge (Figure 3a), which is a list of OPs, and the configuration file (Figure 3b).

Figure 3a) shows just one OP (lines 2-10) as an example. Each OP is composed of two types of information: the target software-knob configuration (lines 3-5) and related performance (lines 6-9) in terms of target metrics. Each field of the OP is a name-value pair that can include also the standard deviation for measured metrics.

Figure 3b shows the *mARGOt* configuration file. In this example we manage a single region of code, thus a single *block* section is included (lines 2-39). Each block section is composed of three main subsections: the metrics to be monitored (lines 4-25), the geometry of the problem (lines 27-29) and the definition of the application requirements (lines 31-38). The monitor section declares the usage of a throughput (lines 4-11) and an error (lines 12-25) monitors. The geometry of the problem states the managed software-knobs (line 27, *num_trials*) and the list of metrics of interest (lines 28-29, *throughput* and *error*). The application requirements includes the goal definition (line 31, throughput ≥ 2), the optimization problem definition (lines 33-38, minimize error subject to the throughput goal) and we ask to *mARGOt* to use the monitor *throughput* to adapt the metric *throughput*. Please, refer to the user manual for the complete syntax and semantics of the configuration file.

Starting from these configuration files, *mARGOt* HEEL generates the high-level interface described in Section 2.1. Figure 4 shows the code after *mARGOt* integration. In particular, we need to include the *mARGOt* header file (line 1), initialize the framework (line 6), update the software-knobs configuration (lines 22-25) and profile the kernel execution (line 26 and 45). In this example, we have also chosen to log the results (line 46).

To demonstrate how the enhanced application is able to react to changes in the execution environment while respecting the optimization goals shown in Figure 3b, we planned a simple experiment running the application for $300s$. After $100s$ we lower the frequency of the CPUs at $1.2Ghz$, to simulate the behavior of a possible power capper, using the *cpupower* tool, and after $200s$ we restore the original frequency.

Figure 5 shows the execution trace of the application. The x-axis is the timestamp, while on the y-axis are shown the observed throughput, the expected computation error, the frequency of the CPUs and the select number of trials. After $100s$ we notice how thanks to *mARGOt*, the application decreases the number of trials to compensate for the performance loss, required to reach the goal. After $200s$, *mARGOt* observes the increment of performance, therefore it increases the number of trials to minimize the computation error.

A tutorial including a step-by-step integration procedure is included on the *mARGOt* repository[4].

4 Overhead analysis

The proposed framework is synchronous with respect to the target application. Therefore, the time used to solve the optimization problem or to change the application requirements represents an application overhead. For this reason, we have designed the framework to be lightweight.

To evaluate the *mARGOt* overhead, the framework repository includes also a benchmark set to support the measure on the target machine. Given that the application knowledge might have an arbitrary number of OPs, constraints or feature clusters, it is not possible to determine an upper bound. Figure 6 shows the overhead evaluation of all the actions that might be

[4]`https://gitlab.com/margot_project/tutorial`

```cpp
#include <margot.hpp>

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

  // initialization code
  // swap_array_mask - vector with the id of the available swaptions

  // main loop of the application
  while (std::chrono::steady_clock::now() < time_limit)
  {
    // mix the swaptions array mask, to have random inputs
    std::shuffle(swap_array_mask.begin(), swap_array_mask.end(), generator);
    std::cout << "\tRunning iteration [" << iteration_counter << "]" << std::endl;

    // initialize the error to zero
    error = 0;

    // update the configuration and start the monitors
    if (margot::pricing::update( num_trials ))
    {
      margot::pricing::manager.configuration_applied();
    }
    margot::pricing::start_monitor();

    // do the computation
    for (int i = 0; i < num_threads; i++)
    {
      threadsID[i] = swap_array_mask[i];
      pthread_create(&threads[i], &pthread_custom_attr, worker, &threadsID[i]);
    }

    // join the threads
    for (int i = 0; i < num_threads; i++)
    {
      pthread_join(threads[i], NULL);
    }

    // compute the average error
    error /= static_cast<float>(num_threads);

    // stop the monitors
    margot::pricing::stop_monitor( num_threads, error );
    margot::pricing::log();

    iteration_counter++;
  }
}
```

Figure 4: The C++ code of swaptions, after the integration with *mARGOt*.

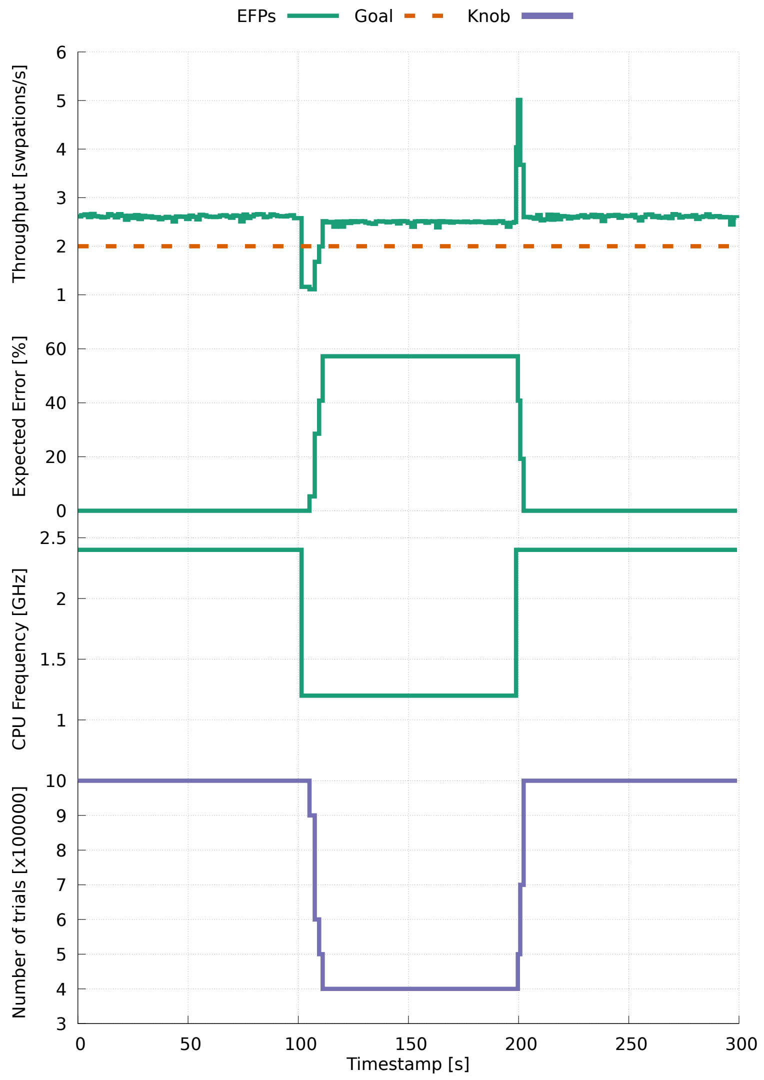
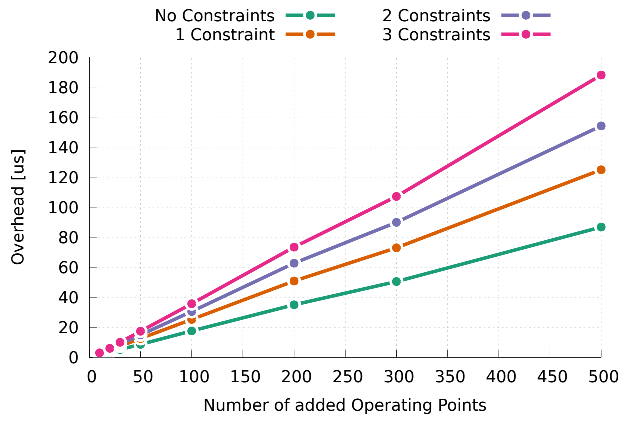
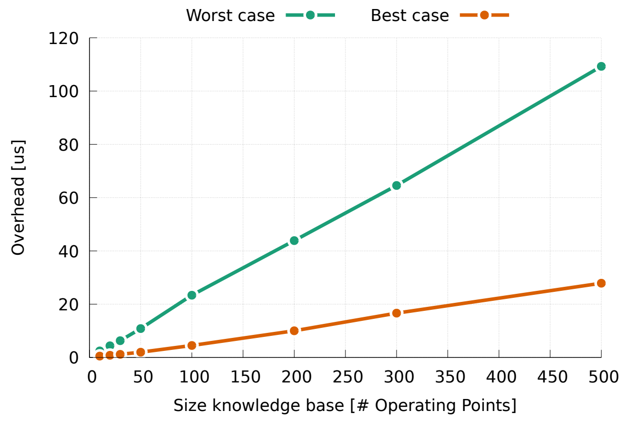


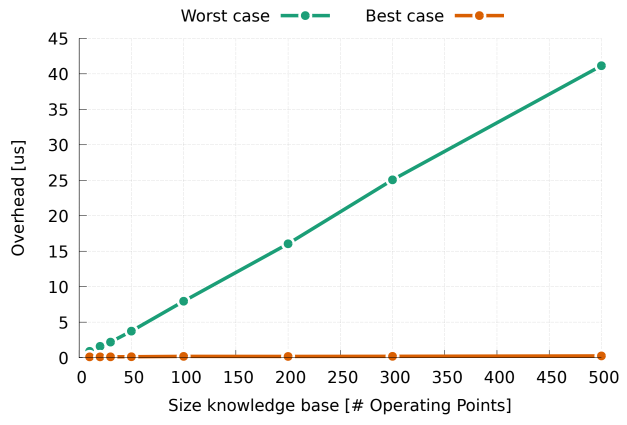
Figure 5: Execution trace of the *swaptions* application.



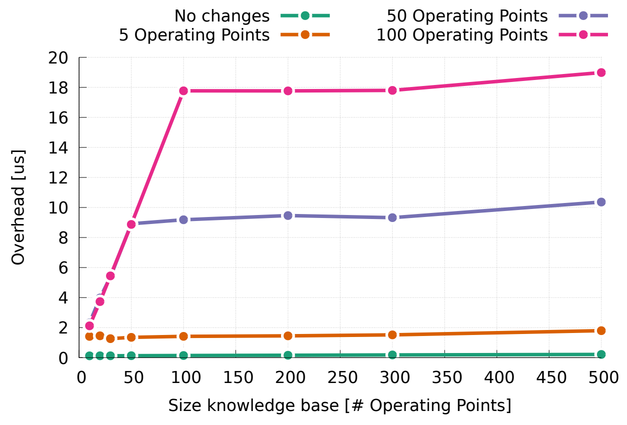
(a) Add Operating Points



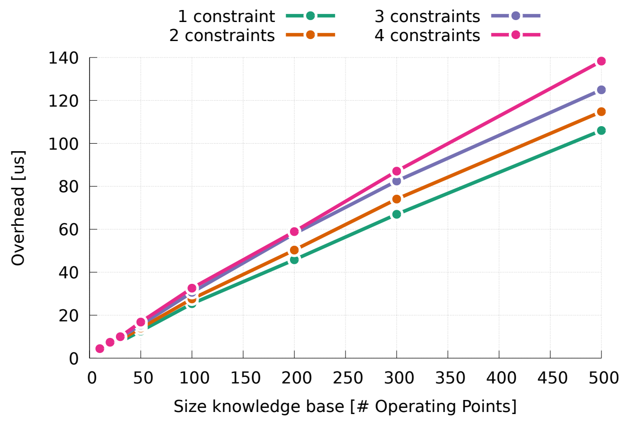
(b) Add a constraint



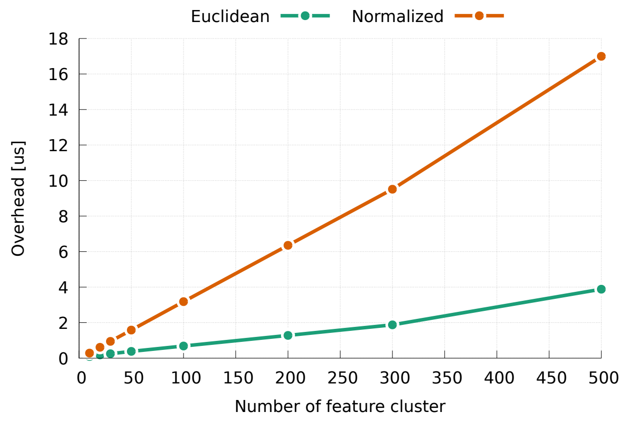
(c) Define objective function



(d) Find best configuration (flat)



(e) Find best configuration (scaling)



(f) Select features cluster

Figure 6: Evaluation of the overheads introduced by *mARGOt* at runtime

performed at runtime, within a reasonable problem definition and using an Intel(R) Xeon(R) CPU E5-2630 v3 machine.

First, Figure 6a–6c shows the overhead in defining the target problem, by varying the size of the knowledge base. Figure 6a shows the overhead of introducing OPs in the knowledge base. Since each constraint uses a "view" of the OPs, the overhead increases with the number of constraints as well. Figure 6b shows the overhead of introducing a new constraint. This evaluation includes only the worst and the best cases since the overhead depends on the number of OPs that satisfy the new constraint. Figure 6c shows the overhead of defining a new objective function. *mARGOt* evaluates the value of only the OPs that satisfy all the constraints. The worst and best cases refer respectively when all OPs or no-one are valid.

Figure 6d and 6e show the overhead of solving the optimization problem. Internally, *mARGOt* uses a differential approach to solve the optimization problem, with respect to previous iterations. If the situation is changed (e.g. a constraint is removed), the overhead is proportional only on the number of OPs that are impacted by the change (Figure 6d). However, Figure 6e represents the worst case for this operation when all the OPs of the knowledge base are involved.

Finally, Figure 6f shows the additional overhead of using the data features, by varying the number of available data-feature clusters. In this experiment, a feature cluster is defined by three values, e.g. the size of a 3D matrix.

5 Usage in the ANTAREX Project

Currently, we are using *mARGOt* in the context of the ANTAREX European project [19] targeting two HPC applications.

First, we enhanced a molecular docking application (LigenDock [20]) to tradeoff accuracy and throughput. In the drug discovery process, the molecular docking task aims at estimating the three-dimensional pose of a given molecule when it interacts with the target protein. Due to the high number of degrees of freedom of the problem, docking applications (such as LigenDock) includes parameters to approximate the solution while decreasing the computation effort. In this context, given a database of molecules to be evaluated, the number of available resources and the allocated time budget to perform a virtual screening, *mARGOt* permits to achieve the highest quality of the results by tuning application software-knobs.

Second, we faced the smart cities context where traffic prediction and cooperative routing algorithms are examples of activities to ease the life of citizens. In this use case, we adopted *mARGOt* both on client-side and on server side. On server side, we enhanced a Monte-Carlo based approach to determine the arrival time distribution of a car navigation system by adopting the proposed framework. *mARGOt* is used to minimize the number of Monte-Carlo simulations for reaching a predetermined accuracy on the estimation of the arrival time. On client side, we are employing *mARGOt* to manage the amount of information exchanged with the server, to have always the best navigation experience without exceeding a predetermined monthly data-bandwidth.

Other applications enhanced using *mARGOt* can be found in literature ([21, 22, 23, 24, 2]).

6 Conclusions

Energy efficiency and system complexity are two cross-cutting challenges for nowadays computing systems. Among the different approaches to address these challenges, runtime adaptivity has emerged as a promising path together with having extra-functional requirements as first-class citizens in programming and modeling languages.

In this chapter, we presented *mARGOt*, a dynamic autotuning framework that enhances the application with an adaptation layer, to provide self-optimization capabilities. Although self-managing software as defined in the autonomic computing vision is still far to be reached, we believe that the flexibility offered by *mARGOt* is a further step in this direction. In fact, it offers to the target application a mechanism to adapt, in a reactive and proactive manner, to the evolution of the system. Moreover, it provides to the application developer a way to specify and modify at runtime the application requirements and the application knowledge. The framework is a standard C++ library that must be linked to the application. The integration process is simplified by the presence of a code generator that produces the needed high-level interface configured for the target application and related requirements. Due to its flexibility, it might be used for several classes of applications, ranging from embedded to High-Performance Computing systems.

References

[1] Marc Duranton, Koen De Bosschere, Christian Gamrat, Jonas Maebe, Harm Munk, and Olivier Zendra. The hipeac vision 2017, 2017.

[2] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Christina Silvano. margot: a dynamic autotuning framework for self-aware approximate computing. *IEEE Transactions on Computers*, 2018.

[3] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[4] Markus C Huebscher and Julie A McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys (CSUR)*, 40(3):7, 2008.

[5] R Clint Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 1998.

[6] Matteo Frigo and Steven G Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 2005.

[7] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*. IOP Publishing, 2005.

[8] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM, 2009.

[9] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on*, pages 303–315. IEEE, 2014.

[10] Christoph A Schaefer, Victor Pankratius, and Walter F Tichy. Engineering parallel applications with tunable architectures. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010.

[11] Ananta Tiwari and Jeffrey K Hollingsworth. Online adaptive code generation and tuning. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE, 2011.

[12] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. *ACM SIGPLAN Notices*, 47, 2012.

[13] Xin Sui, Andrew Lenharth, Donald S Fussell, and Keshav Pingali. Proactive control of approximate programs. *ACM SIGOPS Operating Systems Review*, 50(2):607–621, 2016.

[14] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*. ACM, 2010.

[15] Michael A Laurenzano, Parker Hill, Mehrzad Samadi, Scott Mahlke, Jason Mars, and Lingjia Tang. Input responsiveness: using canary inputs to dynamically steer approximation. *ACM SIGPLAN Notices*, 51(6):161–176, 2016.

[16] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, et al. Multicube: Multi-objective design space exploration of multi-core architectures. In *VLSI 2010 Annual Symposium*, pages 47–63. Springer, 2011.

[17] Gianluca Palermo, Cristina Silvano, and Vittorio 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*, 28(12):1816–1829, 2009.

[18] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[19] Cristina Silvano, Giovanni Agosta, Stefano Cherubin, Davide Gadioli, Gianluca Palermo, Andrea Bartolini, Luca Benini, Jan Martinovič, Martin Palkovič, Kateřina Slaninová, et al. The antarex approach to autotuning and adaptivity for energy efficient hpc systems. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 288–293. ACM, 2016.

[20] Claudia Beato, Andrea R Beccari, Carlo Cavazzoni, Simone Lorenzi, and Gabriele 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.

[21] Davide Gadioli, Gianluca Palermo, and Cristina Silvano. Application autotuning to support runtime adaptivity in multicore architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 173–180. IEEE, 2015.

[22] Davide Gadioli, Ricardo Nobre, Pedro Pinto, Emanuele Vitali, Amir H. Ashouri, Gianluca Palermo, Joao Cardoso, and Cristina Silvano. Socrates - a seamless online compiler and system runtime autotuning framework for energy-aware applications. 2018.

[23] Edoardo Paone, Davide Gadioli, Gianluca Palermo, Vittorio Zaccaria, and Cristina 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*, pages 161–168. IEEE, 2014.

[24] Emanuele Vitali, Davide Gadioli, Gianluca Palermo, Andrea Beccari, and Cristina Silvano. Accelerating a geometric approach to molecular docking with openacc. In *Proceedings of the 6th International Workshop on Parallelism in Bioinformatics*, PBio 2018, pages 45–51, New York, NY, USA, 2018. ACM.