User Guide

mARGOt framework

Draft version

November 27, 2017

Contents

1 Introduction

This document describes the problem addressed by mARGOt and it shows how the framework aims at solving it, highlighting the boundaries between what is expected as input and what is the produced output. Moreover, this guide will describe the framework interface exposed to the user and how to integrate mARGOt in the target application. This document aims at providing the "big picture" of the framework, for implementation details refers to doxygen documentation.

Together with the autotuning framework, it is possible to use an utility, named *mARGOt heel*, which generates an high level interface, hiding as much as possible the implementation details of the integration process. However, the latter is considered out of scope from this document.

1.1 Target problem

The source code describes the functional behavior of the application and it might be abstracted as a function that produces the desired output, starting from an initial input, i.e. $o = f(i)$. A large class of application might expose software knobs which alter its behavior, i.e. $o = f(i, k_1, k_2, \ldots, k_n)$. The idea is that a change on the configuration of those knobs, alter the extra-functional properties (EFP) of the function f (e.g. the execution time or the power consumption) and of the output (e.g. result accuracy or size).

While the functional behavior of the application is utterly described in the source code, its extra-functional behavior is more subtle, since it usually depends on the underlying architecture, on the system workload, on the assigned computational resources and on the current input. Since the user typically has requirement on the application EFP, selecting the most suitable configuration of the software knobs is not a trivial task. Moreover, since the system workload, the assigned resources and the input may change at runtime, choosing a one-fits-all configuration may lead to a sub-optimal solution.

In the context of autonomic computing, an application is seen as an autonomous element, which is able to automatically adapt. The mARGOt autotuning framework aims at enhancing the target application with a runtime adaptation layer, to continuously provide the most suitable configuration according to application requirements and to observation of its actual EFP.

1.2 Framework overview

The main idea behind mARGOt is the MAPE[1] loop. In this context, an autonomic manager is in charge to manage the application, implementing a loop composed by five elements. The **M**onitor element provides the ability to gather insight on the actual application behavior, the **A**nalyse element extract information useful for the **P**lan element, which select the action actuated by the **E**xecute element. The effect of the selected action will be observed by the monitor element, which close the loop. The fifth element of the MAPE loop is the application knoledge, which is leveraged by all the other elements.

[1] Kephart, Jeffrey O., and David M. Chess. "The vision of autonomic computing." Computer 36.1 (2003)

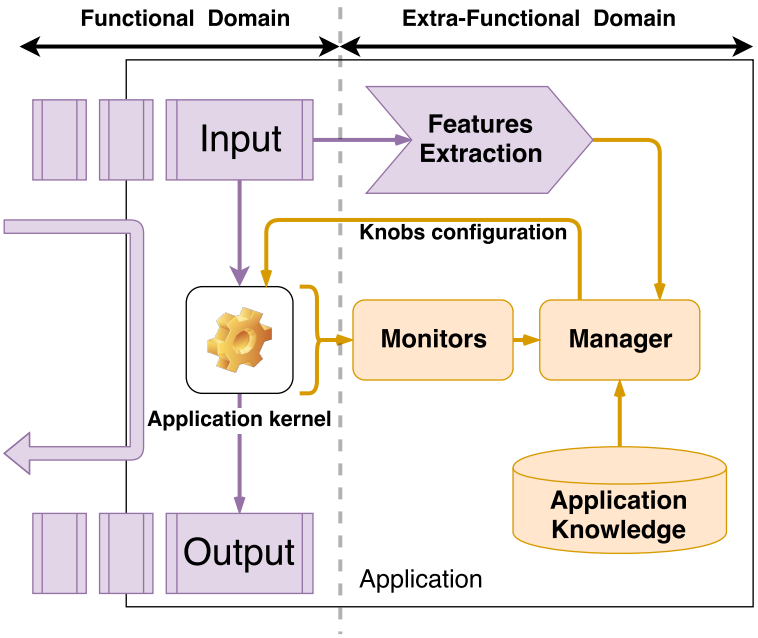

Figure 1: High level overview of the mARGOt structure (highlighted in orange). Purple elements, represents application code.

The mARGOt framework is an implementation of the MAPE loop, which focus on providing to the application the self-optimization property. To improve the versatility of the framework, the Execute element is not part of the framework, but it is the application that is in charge of actuating the action. In particular, if we model the application as $o = f(i, k_1, k_2, \ldots, k_n)$, mARGOt will provide to the application the most suitable values of the software knobs (i.e. $k_1, k_2, \ldots, k_n$). For example, if one knob of the application is the number of threads, mARGOt provides the most suitable number of threads, but it is the application in charge of actually spawn or join threads. This decision, enables the autotuning framework to be agnostic with respect to the nature of the software knobs.

Assuming that the target application is composed by one kernel which elaborate an input to produce an output, Figure 1 shows an overview of the framework structure. In particular, mARGOt is a C++ library which is linked against the target application. Therefore, each instance of the application is able to take autonomous decisions. All the source code that the developer wrote, represented by purple elements, mainly describes the functional domain of the application, i.e. how to produce the output from given the actual input. All the main modules which compose the autotuning framework, represented by orange elements, aims at tuning dynamically the application kernel according to Extra-Functional requirements and they are mostly orthogonal with respect to the functional aspects. The only shared region between the user code and the autotuning framework is in charge of extracting features of the input (if any), which might be leveraged by the autotuning to select the most suitable configuration.

The autotuning framework is composed by three modules. Obviously, the Monitors module is in charge of acquiring insight on the actual behavior of the application, representing the monitor element of the MAPE loop. The Manager module represents the analyze and plan elements of the MAPE loop, to provide to the application the most suitable configuration according to the monitor information, the application knowledge, the features of the current input and the user requirements. The application knowledge is an abstraction of the extra-functional behavior of the application, gathered at design-time through a Design Space Exploration. However, it is possible to change the application knowledge at runtime.

1.3 Framework integration

The purpose of this document is to describe how it is possible to integrate the mARGOt framework in the target application. Following sections of this document will describe in details the interface and purpose of the monitors, the manger and the application knowledge. Finally, Section 5 will provide an example of integration in a toy application.

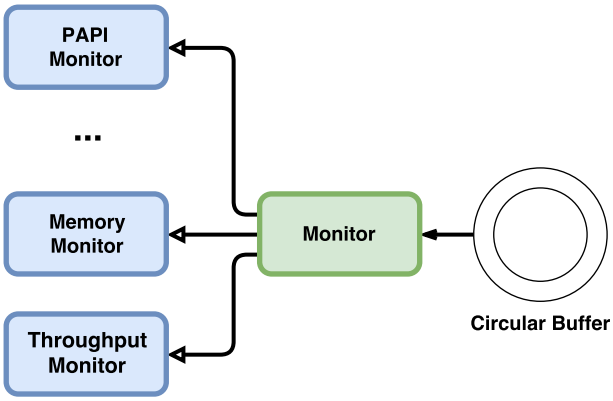

Figure 2: Overview of the monitors module.

2 Monitors module

The most important step in order to adapt, is to acquire insight on the current behavior of the application EFP of interest, which defines the performance of the application. For example, the application developer might be interested on the execution time of the kernel or in the accuracy of the result. While the first metric might be applied to every class of applications, the accuracy of the result is meaningful only in the context of an application. For example, a video encoder might use the Peak to Signal Noise Ration (PSNR) as accuracy metric, while a scientific application might be more interested on the difference between the actual result and the ground truth value.

For this reason the monitors module must be flexible enough to let the user to define custom monitor to observe application-specific metrics. Figure 2 shows the overview of the structure of the monitors module. A base class (*Monitor*) implements a circular buffer which store the last n (user defined) observations and provides to the user the possibility to extract statistical properties over observed data. Moreover, this class implements all the methods required to interact with all the other framework elements. It is possible to specify the type of the elements stored in the circular buffer and the precision used to compute the statistical properties (by default it uses at least 32bit float). Each time a statistical properties is computed, the monitor uses a cache-like mechanism to avoid to recompute the property.

The actual monitor must extend the latter class, with the functionality that actually gather the target metric and push the value in the circular buffer (using the *push*) method. mARGOt ships with the following suite of monitors:

- Energy monitor (uses the RAPL environment)
- Frequency monitor (uses the CPUfreq environment)
- Memory monitor
- Odroid power monitor

```
1   // monitor definitions
2   margot::Monitor<float> my_float_monitor(3);
3   margot::Monitor<int, double> my_int_monitor;
4   margot::Monitor<double>  my_double_monitor;
5   margot::TimeMonitor timer;
6
7   // how to observe a metric
8   timer.start();
9   timer.stop()
10  my_float_monitor.push(3.2f);
11
12  // how to extract statistical properties
13  const auto avg_time = timer.average();
14  const auto avg_float = my_float_monitor.average();
```

Figure 3: Example of C++ code to define and use monitors.

- Odroid energy monitor
- PAPI monitor (uses the PAPI environment)
- Process and system CPU usage
- Temperature monitor (uses the Sensors environment)
- Throughput monitor
- Time monitor

Figure 3 shows some examples on how it is possible to define and use monitors. In particular, line 1 defines a basic monitor which stores float and has a circular buffer of three elements. Line 2 defines a monitor of integer, however it specify that the statistical properties such as average and standard deviation should be performed using 64bit of precision. Moreover, the default constructor set the circular buffer size to just one element. Line 3 defines a monitor of doubles. Since the default type used to compute statistical properties is a float, mARGOt automatically promote the latter to double, to prevent precision loss. Line 4 defines a time monitor, which is a specialization of the generic monitor, suitable to record the elapsed time from a start and stop method (lines 8,9). In general each specialized monitor provides its own method to observe the target metric. For a full description of each monitor, refer to the doxygen documentation. To "observe" a value using a custom monitor (line 10), it is required to use the *push* method. Regardless of the monitor type, to extract statistical properties (lines 13-14), it is enough to use the dedicated method. In particular, it is possible to extract the following statistical properties (defined in the enum *DataFunctions*):

- Average
- Standard deviation

```
1   margot::TimeMonitor timer(margot::TimeUnit::MILLISECONDS);
2   margot::Goal<int, margot::ComparisonFunctions::LESS> my_goal(2);
3
4   timer.start();
5
6   // application kernel
7
8   timer.stop();
9
10  if (my_goal.check<float,float,margot::DataFunctions::AVERAGE>(timer))
11  {
12    std::cout << "We are fast" << std::endl;
13  }
14  else
15  {
16    std::cout << "We should improve" << std::endl;
17  }
```

Figure 4: Example of C++ code to check if the elapsed time to execute the application kernel is below $2ms$.

- Maximum element observed
- Minimum element observed

2.1 Application Goals

Since the application developer might have requirements on the lower bound or upper bound on a metric of interest. mARGOt uses the class *Goal* to represents this concept. In particular, it is possible to test if a statistical properties of the monitor achieve the goal or to compute its absolute o relative error. In the current implementation, a support four standard comparison functions (greater than, grater or equal than, less than and less or equal than), defined in the enum *ComparisonFunctions*.

Figure 4 shows an example on how it is possible to use the goals and monitors, to check if the application kernel execution time is below $2ms$. In particular, line 1 defines a time monitor with a resolution of milliseconds. Line 2 defines a goal with value 2 and comparison function "less than", the numerical value of the goal might be changed dynamically. After that we have instrumented the kernel of the application (lines 4-8), we check if the execution time was below $2ms$ (line 10), and we print an output message accordingly (lines 10-17).

```
1   // define the Operating Point geometry
2   using KnobsType = margot::OperatingPointSegment< 2, margot::Data<int> >;
3   using MetricsType = margot::OperatingPointSegment< 2, margot::Distribution<float> >;
4   using MyOperatingPoints = margot::OperatingPoint<KnobsType, MetricsType>
5
6   // declare the application knowledge
7   std::vector< MyOperatingPoints > application_knowledge = {
8     { // first Operating Point
9       {1, 2},
10      {margot::Distribution<float>(1, 0.1), margot::Distribution<float>(1, 0.1)}
11    },
12    { // second Operating Point
13      {2, 3},
14      {margot::Distribution<float>(1, 0.1), margot::Distribution<float>(2, 0.1)}
15    }
16  };
```

Figure 5: Example of C++ code to define the application knowledge.

3 Application knowledge

The application knowledge describes the expected behavior of the application, in terms of its EFP of interest. To represent the latter concept, mARGOt uses Operating Point objects. An Operating Point is composed by two segments: the *software knobs* segment and the *metrics* segment. The former represents a configuration of the application, the latter represents the performance reached by the application, using the related configuration. The collection of several Operating Points define the application knowledge. Which means that, any configuration provided by the autotuner, must be part of the application knowledge.

3.1 Operating Point geometry

mARGOt implements each segment of the Operating Point as an array of DataType. The DataType is a type suitable to describe a value of either a configuration or a metric. In particular, if the target field of the Operating Point has a known static value, we can represent such field using just its mean value. For examples, the number of threads or the quality of the results may be fully described with a single number. If the target field of the Operating Point is a measured metric, with a stochastic component, we must represent such field using (at least) its mean and standard deviation values. For examples, the execution time of the kernel or its process CPU usage are a random variable, with a mean and a standard deviation.

To improve the efficiency of the framework, the geometry (the C type) of the Operating Point must be known at compile-time. In this way, mARGOt is able to exploit C++ features, to specialize the implementation of the framework according to the application knowledge. Figure 5 shows an example on how it is possible to define the application knowledge. In particular, lines 1-4 defines the geometry of the Operating Point, while lines lines 7-16 defines the actual knowledge of the application. In this example, the kernel has

two software knobs, which values might be defined using only integer numbers (line 2). Moreover, the application developer is interest on only two EFP, represented as a distribution of floats (line 3). Finally, the global geometry of the Operating Point is defined in line 4. The actual application knowledge might be represented using any STL container. In this example, we have chosen a std::vector, brace-enclosed initialized lines(6-16). For a full description of the Operating Point implementation, please refer to the Doxygen documentation.

3.2 How to obtain the application knowledge

The application knowledge is considered as an input of the framework, usually derived from a Design Space Exploration. Since this is a well known topic in literature, the application developer is free to choose the most suitable approach to derive the application knowledge. For example, if the design space is small, it is possible to perform a full-factorial Design of Experiment and evaluate all the possible configuration of the software knobs. If the design space is too big, to perform an exhaustive search, it is possible to employ response surface modeling technique to derive the application knowledge. For example, it is possible to evaluate only a subset of the design space and then interpolate the missing point.

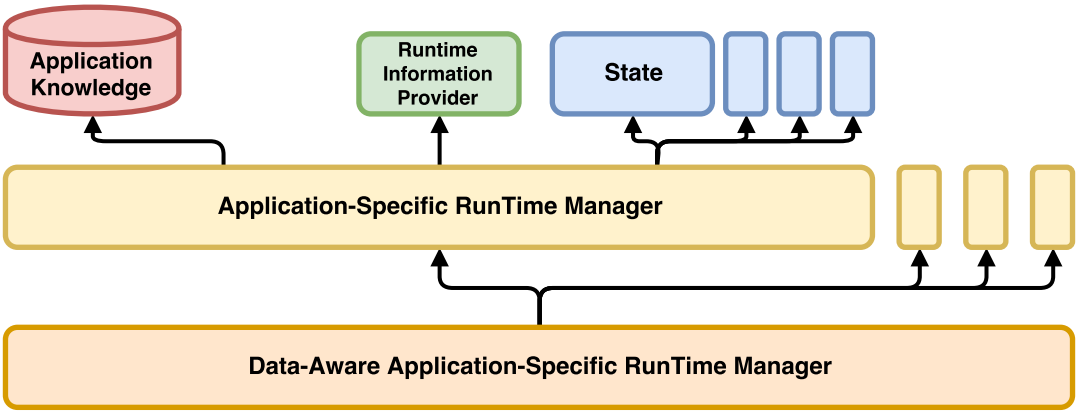

Figure 6: Overview of the manager module.

4 Application-Specific RunTime Manager

This section describes the heart of mARGOt: the Manager module. This module is in charge of selecting the most suitable configuration for the application. Figure 6 shows the overall architecture of the module and the interaction between its principal elements. In the remainder of the section, each element is explained in more details, following a bottom up approach. However, the application developer has to interact only with the Application-Specific RunTime Manager (ASRTM) element or with the Data-Aware Application-Specific RunTime Manager element (DA-ASRTM).

4.1 Rutime Information Provider

This element relates the expected behavior of the application, in terms of EFPs, with the ones observed by the monitors. In particular, since the framework will provide to the application a configuration which belong to the application knowledge, mARGOt knows the expected behavior of the application. Therefore, if there is some difference between the expected behavior and the actual behavior of the application, the idea is that the framework must adjust the application knowledge accordingly. To achieve this goal, the runtime information manage define the **coefficient error** c_e^i for the *i-th* field of the Operating Point as reported in Eq. 1.

$$c_e^i = \frac{expected_i}{mean_i} \tag{1}$$

Where $mean_i$ represents the mean value observed by the related monitor and $expected_i$ represents the expected value, contained in the application knowledge. Therefore, if c_e^i is equal to 1, it means that the $i-th$ field of the application knowledge matches perfectly with information provided by the monitor. For numerical stability, if the observed value of the metric is exactly equal to zero, the framework adds a padding value of 1 to the numerator and to the denominator. In this case, c_e^i tends to underestimate the actual value.

Typically, each measured value is affected by noise, produced either by the operating system or by the technique used to gather the measure. To be robust with respect to this kind of noise, if the mean value observed by the monitor is within one standard deviation with respect to the expected mean value, we set c_e^i equal to one. The latter scenario requires that the target field is defined as a distribution in the application knowledge. The idea is to adapt only if the difference is statistically significant.

Since some metrics may have spikes due to exceptional events, for instance when a process is migrated to another computing element, the runtime information provider store the last n c_e^i in a circular buffer, initially filled with ones (we assume that the application knowledge matches the reality). The framework names **inertia** the value n. If we set an high inertia to the *i-th* field of an Operating Point, we are less prune to react over extraordinary events, but we are also less responsive in case of abrupt changes on the application behavior. This parameter is exposed to the end user.

Therefore, the actual coefficient error used by the framework $\bar{c_e^i}$ is the average of all the c_e^i gathered at runtime, as defined in Eq. 2. Where j indicates the *j-th* c_e^i in the circular buffer and n indicates the inertia of the *i-th* field.

$$\bar{c_e^i} = \frac{\sum_{j=0}^{n-1} c_{e,j}^i}{n} \tag{2}$$

mARGOt uses $\bar{c_e^i}$ to perform a linear error propagation. For examples, if with the selected configuration we expect a throughput of $10fps$, but the throughput monitor observe a throughput of $8fps$, mARGOt assumes that also all the other Operating Points will have a throughput that is 20% slower than expected. Since the error coefficient for the $i-th$ field of the Operating Point is computed in an independent way from the error coefficient of the $j-th$ field, mARGOt is able to perform a fine grained scaling of the application knowledge, to face changes in the execution environment.

From the integration point of view, the autotuner API enable the application developer to dynamically:

- Relate a monitor to a field of the Operating Point.
- Remove all the information providers.

4.2 State

The application requirements are represented as a multi-objective constrained optimization problem. Therefore, the autotuner is able to provide to the application the most suitable configuration, solving by inspection the optimization problem. The state element is in charge of solving this problem, in en efficient way. Let $x = [x_1, \ldots, x_n]$ be the vector of the tunable parameters and $p = [p_1, \ldots, p_n]$ the vector of the non-tunable parameters. The tunable parameters are the software knobs that influence the application behavior, while the non-tunable parameters are the metric of interests, represented as distribution with a mean value and a standard deviation. The constrained problem is described as in Eq. 3.

$$
\begin{array}{lrcllll}
\max(\min) & f(x;p) & & & & & \\
\text{s.t.} & C_1 : \omega_1(x;p) & \propto & k_1 & with & \alpha_1 & confidence \\
& C_2 : \omega_2(x;p) & \propto & k_2 & & & \\
& C_3 : \omega_3(x;p) & \propto & k_3 & with & \alpha_2 & confidence \\
& \cdots & & & & & \\
& C_n : \omega_n(x;p) & \propto & k_n & & &
\end{array}
\tag{3}
$$

In the description of the problem, f denotes the objective function (named *rank* in the autotuner context), which is defined as a composition of any of the variables defined in x or p, 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 tunable and non-tunable parameters, that must satisfy the relationship $\propto \in \{<, \leq, >, \geq\}$ with the threshold value k_i and confidence α_i (if the parameter is a distribution). Since we are agnostic about the distribution of the target parameter, the confidence is expressed as the number of times to consider its standard deviation. These values are collected in a vector $k = [k_i]$; together the vectors p and k are the input parameters of the problem.

If there is at least one configuration of the software knobs that satisfies the constraints, then the autotuner selects the one that maximize the value of the objective function. Otherwise, the autotuner start to relax the constraints, starting from the "least important", to find the configuration closer to be valid. For this reason, it is important to attach a *priority* to each constraint, which is a numerical value that indicates how much important is a constraint. Two constraints shall not have the same priority. Moreover, it is not possible to express a constraint with = sign, since in case mARGOt must relax the constraint, it is not possible to check if it is better to get a greater value or a lesser value. In case the application user would like to express the = relation, it must define two constraints ($\leq$ and $\geq$). Since two constraints shall not have the same priority, mARGOt is able to tell in which direction is best to find the most suitable configuration, according to the priority of the constraints.

From the implementation point of view of the autotuner, a *constraint* is defined against a *goal* that state a single condition. For example, C_1 is a *constraint* that is defined by the *goal* g_1. The goal g_1 define the condition $\omega_1(x;p) \propto k_1$. For this reason, declaring a new *goal* $goal_{new}$ does not change the optimization problem. However, creating a *constraint* that refers to the *goal* $goal_{new}$ does change the optimization problem.

Regarding the definition of the objective function, the framework exposes three ways of combine any variables defined in x or p:

Geometric With this composition method, the objective function f is defined as in Eq. 4,

$$
f = \omega_1(x;p)^{c^1} \cdot \omega_2(x;p)^{c^2} \cdot \ldots \cdot \omega_n(x;p)^{c^n} \tag{4}
$$

where the terms are multiplied by each other, using a user specified weights $c^1, c^2, \ldots c^n$.

Linear With this composition method, the objective function f is defined as in Eq. 5,

$$
f = \omega_1(x;p) \cdot c^1 + \omega_2(x;p) \cdot c^2 + \cdots + \omega_n(x;p) \cdot c^n \tag{5}
$$

where the terms are added together, using a user specified weights $c^1, c^2, \ldots c^n$.

Simple With this composition method, the objective function f is defined as the term $\omega_i(x;p)$

Since the time spent to solve the optimization problem is stolen from the application, the state element rearrange the application knowledge to minimize this overhead. In particular, the introduced overhead is proportional to the amount of changes since the last time that the problem was solved. Thus, if there are no changes, the time spent to solve the problem is negligible.

From the integration point of view, the autotuner API enable the application developer to dynamically:

- Add/Remove a *constraint*.
- Define the *rank* function.
- Change the value of a *goal*, thus on all the related *constraints*.

4.3 Application-Specific RunTime Manager

This element orchestrates the application knowledge, the runtime information provider and the application states. In fact, since the application might have different requirements according to the evolution of the system, it is possible to define several states. Each state represents the multi-objective constrained optimization problem described in the previous section.

For the integration point of view, this element expose to the user a single interface to interact with Manager module of the framework. In particular, the autotuner API enable the application developer to dynamically:

- Add/Remove Operating Points
- Relate a monitor to a field of the Operating Point.
- Remove all the information providers.
- Create/Remove a state
- Select the active state

Those operations work at AS-RTM level, which means that are independent from the active state. If you wold like to dynamically change the current active state, you might:

- Set a different rank function
- Add/Remove a constraint
- Solve the optimization problem
- Get the most suitable configuration

If you want to change only the numerical value of a constraint, the application must change the value of the related goal instead. Please, refer to the doxygen documentation for implementation details about the exposed API.

4.4 Data-Aware Application-Specific RunTime Manager

The AS-RTM assumes that the application behavior is either input independent or that there are few abrupt changes in the input, distributed over time. Which means that the AS-RTM assumes either that the application EFPs depends only on the configuration of the software-knobs or that the adaptation due to the runtime information provider is enough to capture the relation between the EFPs of the application and the input data.

To overcome this limitation, mARGOt uses the Data-Aware AS-RTM (DA-ASRTM). If the user is able to extract features of the input which are related to the behavior of the application, then it is possible to define the latter in terms of software knobs and data features. In particular, the idea is to cluster the space of data features using the most representative cases and select, at run-time, the cluster closer to the actual input.

mARGOt represents each cluster of data feature with an AS-RTM. This means that each cluster might have a different application knowledge, therefore the solution of the same optimization problem found by two different cluster, may be different. For example, suppose that the input of the application kernel managed by mARGOt is a matrix, which must be elaborated. Suppose also that the application kernel exposes a software knob that changes the precision of the elaboration and that the application user would like to maximize the quality of the elaboration given a time budget. In this case we might use the size of the matrix as data feature, defining few clusters for the most representative sizes. According to the application knowledge, a “small” input might be elaborated with a greater precision than a “big” input.

In general, mARGOt attach to its cluster a label that characterize the cluster, defined as an array of numeric values. The idea is that each value represent a feature of the input. In the previous examples, those numbers were the dimension of the input matrix. Moreover, mARGOt defines a function that selects the cluster closer to the features of the actual input of the application. In order to do so, mARGOt expose to the user two parameters to define the distance function: the validity constraints and the type of distance. The validity constraints define, for each field of the feature cluster, if the value of the evaluated feature cluster must be $\leq$, $\geq$ with respect of the value of the actual input. However, if it is not important for the user, it might specify the “don’t care” validity. The validity constraints, help the autotuner to consider only the valid ones, when it search for the data feature cluster closer to the actual input data. Since each dimension of the feature cluster label can be of different order of magnitudes, mARGOt expose to the end-user the type of distance to evaluate. The default type is *euclidean*, in this case the selected data cluster which label is valid and closer to the one of the actual input. Otherwise, it is possible to compute the *relative* distance, where the distance on each field is normalized in a independent way.

From the implementation point of view, the DA-ASRTM expose to the user a unified interface that overrides the one of the AS-RTM. In addition to the methods of the AS-RTM, it enables the application to:

- Add/Remove a feature cluster.
- Select the active cluster.

Please notice that mARGOt enforce the all the data clusters share the same optimization problem. Which means that all the operations that manipulate the state (e.g. adding a constraint or defining the objective function) and all the operations related to the runtime information providers are applied to all the data feature clusters. Moreover, when a new feature cluster is created, mARGOt initialize the corresponding AS-RTM cloning the AS-RTM of another feature cluster, copying all the defined states and information providers. The only information that is not copied are related to the application knowledge. For this reason, the only methods of the AS-RTM which are related to active cluster are:

- Add/Remove Operating Points
- Solve the optimization problem
- Get the most suitable configuration

For the implementation details, please refer to the Doxygen documentation.

```
1   // kernel function
2   void do_work( std::vector<float> input_data, const int knob );
3
4
5   int main()
6   {
7
8     while(work_to_do())
9     {
10      const auto current_input = get_work();
11      do_work(current_input, 2);
12    }
13
14  }
```

Figure 7: The C++ code of the toy application.

5 Integration in the target application

In Section 4 we have defined all the elements that compose the core of mARGOt. This section aims at explaining by examples, how to integrate the autotuner in a target application. We consider a toy application for clarity purpose.

5.1 The target application

Figure 7 shows the code of the toy application that we are targeting. In particular, line 1 declares the kernel function *do_work* that we want to tune. It takes as input the data, represented as a vector of float and a software knobs that alter the function EFPs. In this example, we suppose that the knob performs an approximation of the computation, for instance driving the loop perforation. Therefore, by using a small value of the knob we approximate less, while using a large number, we approximate more. The application is defined as a main loop (lines 8-12) that continuously elaborates new inputs. Due to the expertise of the developers, they have set a one-fit-all default value for the knob, hardcoded to the value 2. Obviously, we would like to manage the body of the loop.

In our example, the application developer is interested in two metrics: the execution time and quality of results. In particular, they would like to maximize the quality of results, provided that the execution time is below a certain threshold. Suppose that the quality of results is input independent, while the execution time depends on both, the size of the vector and the value of the software knob.

5.2 Integration without data feature

This section explain how to integrate mARGOt in the target application, without considering any feature of the input. Below, it is stated the integration from the code point of view, what it lacks is the integration from the building system point view. Since mARGOt wants to be as agnostic as possible regarding the building systems, during the compilation of the

library it generates two files to help the integration of mARGOt in the building system of the target application. In particular, it creates a *FindMARGOT.cmake* file for integrating the framework in a CMake projects, while it creates a *margot.pc* file for integrating the framework in projects that uses Make/autotools.

Figure 8 shows the whole code of the application to manually integrate mARGOt in the target application. For demonstration purpose, it exploits all the features of the AS-RTM. For simplicity purpose we have omitted the required headers and we assume to use the namespace margot. As you can see from Figure 8, to minimize the introduced overhead, mARGOt tries to specialize as much as possible its data structures using template arguments.

Since in this example mARGOt doesn't use any feature of the input, it leverages the runtime information provider to react to changes in the input. If we have few abrupt changes in the input size, mARGOt is able to sense those changes and react accordingly.

Declaring the application knowledge

The first step to integrate mARGOt is to define the application knowledge. In this example, we assume that the application developer performed a Design Space Exploration on the software knob, evaluating the execution time and the error using two values of the knob: 1 and 2. Before defining the application knowledge, it is required to specify the Operating Point geometry (lines 1-4). In particular, line 2 states that the each Operating Point has only one software knob, which is a integer value. Therefore, to describe the software knob it is enough to state its mean value. Line 3 states that each Operating Point has 2 metrics, which are a distribution of floats. Therefore, to describe each metric it is required to state its mean value and its standard deviation.

The actual knowledge is defined later (lines 11-21) and it is composed by two Operating Points. The first Operating Point (lines 13-16) states that when the software knob *knob* is equal to 1, the application has an execution time of $75 \pm 1ms$ and achieve a quality of results of 1. The whole list of Operating Points define the application knowledge, therefore the autotuner can select for the software knob *knob* either 1 or 2. In particular, after the declaration of the manger (line 24), we set the knowledge base (line 27).

Adding the runtime information provider

Since we are not using any data feature, we exploit the feedback information of a monitor on the execution time to adapt the knowledge base, working in closed loop. In this configuration we are able only to react to changes in the input, therefore we assume that there will be few abrut changes in the size of the input. To achieve this behavior, we creates a time monitor with granularity of milliseconds and with a circular buffer that stores the last 3 observation (line 23); while we use an information provider with inertia 5 on the metric with index 0 (line 28).

Defining the optimization problem

Given the application requirement, we need only one state which we call "my optimization". Therefore, we need to create the new state (line 29), select the new state as the active one (line 30). From now on, all the commands that alter the state refer to active one.

To define the optimization problem, we first have to define the objective function. For simplicity, we have created an alias which refer to the average value of the second metric, thus with index 1 (lines 31-34). Afterward, we have defined the rank to maximize the average value of the second metric (line 35). To represent the constraint, we need to define the related goal first (line 22), then we can add the constraint (line 36). In particular, the latter statement creates a constraint related to the first metric, thus with index 0. In this example, we would like to have some confidence that the constraint is satisfied by the configuration, we created the constraint taking into account 3 sigma of its value. Since the goal has the relation <, the autotuner evaluates the upper bound of each Operating Point. For example, the expected execution time of the second Operating Point is $41ms = 35ms + 6ms$.

Obtaining the most suitable configuration

All the previous code was related to the initialization of the problem, to actually adapt we need to solve the optimization problem (line 43) and to retrieve the most suitable configuration (line 44). To take advantage of the observed values, mARGOt must be notified when the application has terminated to actuate the proposed configuration. In this example, the actuation of the configuration is trivial (line 48), therefore we can send immediately the notification (line 45). To leverage the runtime information about the execution time, we need to profile the tuned region of code (lines 47 and 49). The autotuner will automatically adapt according to the observed situation.

5.3 Integration with data feature

In this scenario, we don't rely anymore on the assumption of having few abrupt changes in the input, but we allow the possibility that in each iteration of the loop we might have a different size of the input. For this reason is not enough to be reactive, but we need to be proactive, taking into account the size of the input. Figure 9 shows the integration code required. In particular, we have defined two clusters that represent the case when we have the array of size 100 and of size 1000.

For the implementation point of view, the integration code is very similar. The difference is that we need to define an Operating Point list for each cluster (lines 12-13). The type of the manager is no more an AS-RTM, but is a DA-ASRTM (line 14,15). In particular, beside the geometry of the Operating Points, we need to define the data feature. In this example, we have just one data feature, which is the size of the input array, of type integer. Moreover, we need to specify the distance type and the validity function. In this example, we use the euclidean distance and we are just interested on choosing the closer data cluster, therefore we use the "don't care" enumerator.

Another difference is that, we need to create and select the data clusters (lines 18,19,21,22). Moreover, we have to set the corresponding application knowledge (lines 20,23). After the initialization of the autotuner, we are able to leverage the input feature, by selecting the closer data cluster for each input (line 39).

```cpp
// defining the Operating Point geometry
using software_knob_geometry = OperatingPointSegment< 1, Data<int> >;
using metrics_geometry = OperatingPointSegment< 2, Distribution<float> >;
using MyOperatingPoint = OperatingPoint< software_knob_geometry, metrics_geometry >;

// kernel function
void do_work( std::vector<float> input_data, const int knob );

int main()
{
  // defining the required objects
  std::vector< MyOperatingPoint > knowledge = {
    { // Operating Point 1
      {1},
      {margot::Distribution<float>(75.0, 1.0), margot::Distribution<float>(1.0, 0.0)}
    },
    { // Operating Point 2
      {2},
      {margot::Distribution<float>(35.0, 2.0), margot::Distribution<float>(0.6, 0.1)}
    }
  };
  Goal<float, ComparisonFunctions::LESS> my_time_goal(50);
  TimeMonitor timer(margot::TimeUnit::MILLISECONDS, 3);
  Asrtm<MyOperatingPoint> manager;

  // initialize the AS-RTM
  manager.add_operating_points(knowledge);
  manager.add_runtime_knowledge<OperatingPointSegments::METRICS, 0, 5>(timer);
  manager.create_new_state("my_optimization_problem");
  manager.change_active_state("my_optimization_problem");
  using my_obj_fun = OPField<OperatingPointSegments::METRICS,
                             BoundType::LOWER,
                             1,
                             0>;
  manager.set_rank< RankObjective::MAXIMIZE, FieldComposer::SIMPLE, my_obj_fun>(1.0f);
  manager.add_constraint<OperatingPointSegments::METRICS,0,3>(my_time_goal,10);

  while(work_to_do())
  {
    const auto current_input = get_work();

    // get the most suitable configuration
    manager.find_best_configuration();
    const auto best_configuration = manger.get_best_configuration();
    manager.configuration_applied();

    timer.start();
    do_work(current_input, best_configuration.get_mean<0>());
    timer.stop();
  }

}
```

Figure 8: The C++ code of the toy application, integrated with mARGOt without using any data features

```cpp
// defining the Operating Point geometry
using software_knob_geometry = OperatingPointSegment< 1, Data<int> >;
using metrics_geometry = OperatingPointSegment< 2, Distribution<float> >;
using MyOperatingPoint = OperatingPoint< software_knob_geometry, metrics_geometry >;

// kernel function
void do_work( std::vector<float> input_data, const int knob );

int main()
{
  // defining the required objects
  std::vector< MyOperatingPoint > knowledge_1 = // definition
  std::vector< MyOperatingPoint > knowledge_2 = // definition
  DataAwareAsrtm<MyOperatingPoint,int,
                 FeatureDistanceType::EUCLIDEAN,FeatureComparison::DONT_CARE> manager;

  // initialize the AS-RTM
  manager.add_feature_cluster({{100}});
  manager.select_feature_cluster({{100}});
  manager.add_operating_points(knowledge_1);
  manager.add_feature_cluster({{1000}});
  manager.select_feature_cluster({{1000}});
  manager.add_operating_points(knowledge_2);
  manager.add_runtime_knowledge<OperatingPointSegments::METRICS, 0, 5>(timer);
  manager.create_new_state("my_optimization_problem");
  manager.change_active_state("my_optimization_problem");
  using my_obj_fun = OPField<OperatingPointSegments::METRICS,
                             BoundType::LOWER,
                             1,
                               0>;
  manager.set_rank< RankObjective::MAXIMIZE, FieldComposer::SIMPLE, my_obj_fun>(1.0f);
  manager.add_constraint<OperatingPointSegments::METRICS,0,3>(my_time_goal,10);

  while(work_to_do())
  {
    const auto current_input = get_work();

    // get the most suitable configuration
    manager.select_feature_cluster({{current_input.size()}})
    manager.find_best_configuration();
    const auto best_configuration = manger.get_best_configuration();
    manager.configuration_applied();

    timer.start();
    do_work(current_input, best_configuration.get_mean<0>());
    timer.stop();
  }
}
```

Figure 9: The C++ code of the toy application, integrated with mARGOt using the data features

6 Conclusion

This document describes mARGOt, a dynamic autotuner framework, from an high-level point of view. Moreover, it provides examples on how it is possible to enhance an application with an adaptation layer. For a complete reference on the implementation details, please refer to the Doxygen documentation.

To further simplify the integration process, mARGOt ships with an utility tool, named *margot_cli*, which provides to the application developer additional features to ease the integration process. In particular, it can generate an high level interface, named *margot heel*, that hides as much as possible implementation details. The guide to the utility is covered in a separate user guide.