User Guide
mARGOt framework

Draft version

November 9, 2017

Contents

1 Introduction **1**
1.1 Target problem . . . 1
1.2 Framework overview . . . 1
1.3 Framework integration . . . 3

2 Monitors module **4**
2.1 Application Goals . . . 6

3 Application knowledge **7**
3.1 Operating Point geometry . . . 7
3.2 How to obtain the application knowledge . . . 8

4 Application-Specific RunTime Manager **9**
4.1 Rutime Information Provider . . . 9

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)

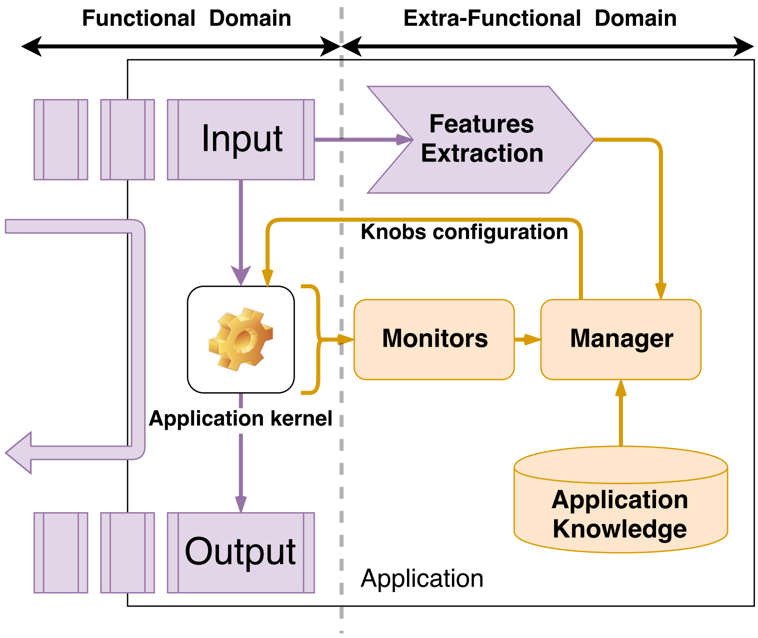

Figure 1: High level overview of the mARGOt structure (highlighted in orange). Purple elements, represents application code.

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, the section will provide an example of integration in a toy application.

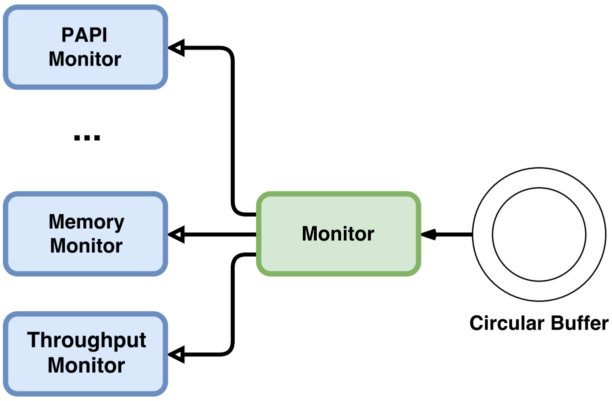

Figure 2: Overview of the monitors module.

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.

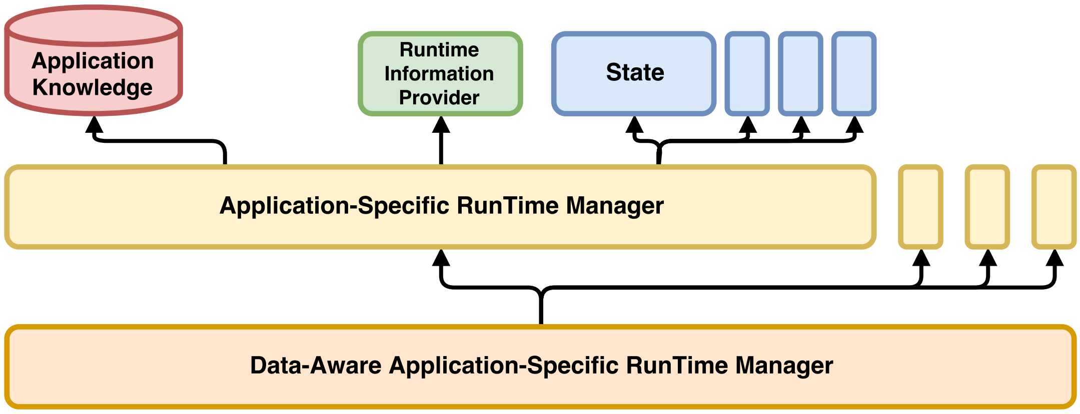

Figure 6: Overview of the manager module.

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.