User Guide

mARGOt heel

Draft version

April 23, 2018

Contents

1 Introduction **1**

2 High-Level interface **1**

3 Configuration overview **3**

4 Knowledge configuration file **4**

5 Adaptation file overview **6**
5.1 Describing the monitor element 8
5.2 Describing the geometry of the application knowledge 11
5.3 Online Design Space Exploration 13
5.4 Describing the plan element 14

6 mARGOt command line interface **16**

A Monitor implementation details **i**
A.1 Collector Monitor i
A.2 Energy Monitor i
A.3 Frequency Monitor ii
A.4 Odroid Energy Monitor ii
A.5 Odroid Power Monitor iii
A.6 Papi Monitor iii
A.7 Process CPU Usage Monitor iv
A.8 System CPU Usage Monitor iv
A.9 Temperature Monitor v
A.10 Throughput Monitor v
A.11 Time Monitor v

1 Introduction

The main goal of mARGOt is to provide a dynamic autotuning framework, to enhance an application with an adaptation layer. The mARGOt user guide, provides an high level description of the framework and example to help the integration process. This document describes mARGOt heel, a collection of tools that aim at easing the integration process and the management of the application knowledge. Before reading this document, we advise to go through the mARGOt user manual.

In particular, mARGOt heel is composed by two elements: the mARGOt command line interface (*margot_cli*) and the mARGOt high-level interface (*margot_if*). margot_cli is a tool written in python that provides utility feature to manage the application knowledge and create a simple Design Space Exploration script, based on *make* files. margot_if is CMake-based library which generates an high-level interface of mARGOt, according to XML configuration files, using margot_cli to generate the required glue code. The main idea is that, provided the application requirements described in XML, it is possible to generate a simple interface, composed by few functions, to hide as much as possible implementation details of the autotuner framework.

This document is structured as follows, at first it describes the high level interface generated by margot_if, then it describes the syntax of the related XML configuration files. The last part of the document provides an overview of all the utility commands provided by margot_cli. For an example of integration, please refer to the tutorial repository on GitLab[1]

2 High-Level interface

Section 5 of the mARGOt user manual shows an example of manual integration in the target application. Even if the manual integration provides great flexibility, it has two major flaw: it requires a deep knowledge of the internals data structure of mARGOt and it requires a fair amount of code to be inserted in the target application. To overcame these limitations, mARGOt heel provides a mechanism to generate a very high-level interface that ease the integration process. In particular, starting from XML configuration files, it generates the following main functions:

init
: 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.

start_monitor
: 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.

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

stop_monitor 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.

log 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.

update 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.

As reported in their description, the actual prototype of these functions depends on the XML configuration provided by the user. Beside generating the definition of these function, the high-level interface expose directly to the application all the objects created in the high-level library, using a standard hierarchy of namespaces. In this way, it is possible to use directly the API exposed by the autotuner. In particular, this is the used hierarchy of namespaces, under the assumption that the application is composed by only one block of code named “foo”:

margot This is the global namespace, to avoid name clashing with other tools, and it defines the global *init* function.

foo This is the namespace for the region of code “foo”. It defines the start_monitor, stop_monitor, log and update function. It holds the definition of the manager, as an object named “manager”. Moreover, it defines two enums which relate the index of a field of the Operating Point with its name. For example, if this region uses a knob named $knob1$, the related enumerator is margot::foo::Knob::$KNOB1$ and it represents its index. The enum for the metrics of interest is named *Metric*.

monitor This is the namespace which holds the definition of the monitor objects.

goal This is the namespace which holds the definition of the goal objects.

Since the autotuner framework is written in C++, the complete high-level interface is generated in C++. However, margot_if generates also a small C interface, that expose a C version of the five main functions. Since it is not possible to access directly the objects stored in the namespace hierarchy, the C interface declares additional utility functions which cover the most common operations:

- For each goal, it generates a function that changes its value.
- For each manager, it generates a function that switch the active state.
- For each manager, it generates a function that notify when a configuration is applied

The signature of the generated functions, try to mimic the namespace hierarchy. For example, to change the value of the goal “goal1” in the region of code ”foo” to the value 2, the C++ statement is *margot::foo:goal::goal1.set(*2*)*, while the C statement is *margot_foo_goal_goal1_set_value(*2*)*.

3 Configuration overview

The mARGOt autotuning framework implements a Monitor-Analyze-Plan-Execute loop based on application knowledge (MAPE-K). While the Analyze and Execute elements of the loop do not require any configuration, the Monitor, Plan and Knowledge elements need some input from the user. In particular, the Monitor element requires the user to specify the monitors of interest for the application and, for each monitor, the related statistical properties that must be exposed. The Plan element need to know the definition of “best” configuration from the user, defined as a constrained multi-objective problem. Finally, the application Knowledge should describe the application behavior, by varying the selected configuration. In particular, the knowledge is a list of a paired information. Each pair relates an application configuration with performance reached using that configuration.

The autotuner handles a single block of code of the application. To match phases of the application, we might use more than one autotuner, where each instance handles a different block of code of the application. For this reason, the configuration of the MAPE-K elements of an autotuner, are related to a single block of code. Moreover, to employ separation of concerns, the Knowledge of the application is split in a different file with respect to the other information. For this reason we have one common configuration file for all the Monitor and Plan elements of each block of code of the application; but we have different files for the Knowledge of different blocks of code of the application.

To clarify the concept, Figure 1 shows a pseudo-code of a very simple application. The whole computation is a loop that executes two different kernels on the same data. In this example we suppose that the developer is interested on managing two different blocks of the application (highlighted in red). The first block is named “kernel1”, while the second block is named ”kernel2”. We would like to stress the fact that a block of code might involve several lines of code, not just a function call. In this case we have a single

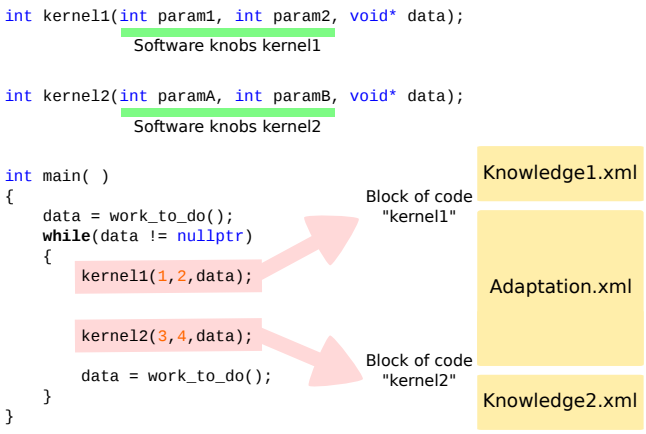

Figure 1: Pseudo-code of an application with two block of code to be tuned.

configuration file for the Monitor and Plan elements (*Adaptation.xml*) of both the blocks of code; while we have two different files for the application Knowledge (*Knowledge1.xml* and *Knowledge2.xml*).

4 Knowledge configuration file

The knowledge file describes the behavior of a single block of code of the application, by varying the configuration of the software knobs. Typically, the knowledge is the output of a Design Space Exploration and the most simple representation is as a list of Operating Points. Each Operating Point relates a configuration with the performance of the observed block of code.

Figure 2 shows an example of the syntax of the knowledge file. In particular, the root XML element is the tag *points*, which must define the attribute *block*. The latter is the name of the block of the application, described by the Operating Point list. The name of the block must be a valid C++ namespace identifier. The list is composed by a sequence of Operating Points, each one is represented with the tag XML *point*. The examples shows two different Operating Points (they have a different configuration of the software knobs). Each Operating Point is composed by a pair of information: the configuration of the software knobs and the performance reached using such configuration.

In particular, the configuration is represented by the XML tag *parameters*, while the value of each software knob is represented by the XML tag *parameter*. The latter XML element must define two attributes: the name of the target software knob (the attribute *name*) and its actual value (the attribute *value*). The name of each software knobs must be an unique valid C++ enumerator. The name is not case sensitive.

The performance of the block of code is represented by the XML tag *system_metrics*, while each metric of interest (that defines the performance) is represented by the XML tag *system_metric*. The latter XML element must define two attributes: the name of the target metric (the attribute *name*) and its actual value (the attribute *value*). Optionally,

```xml
<?xml version="1.0" encoding="UTF-8"?>
<points xmlns="http://www.multicube.eu/" version="1.3" block="kernel1">

  <point>
    <parameters>
      <parameter name="param1" value="1"/>
      <parameter name="param2" value="100"/>
    </parameters>
    <system_metrics>
      <system_metric name="quality" value="100" standard_dev="1.0"/>
      <system_metric name="throughput" value="324.279" standard_dev="1.0"/>
    </system_metrics>
  </point>

  <point>
    <parameters>
      <parameter name="param1" value="5"/>
      <parameter name="param2" value="3"/>
    </parameters>
    <system_metrics>
      <system_metric name="quality" value="500" standard_dev="1.0"/>
      <system_metric name="throughput" value="17.487" standard_dev="1.0"/>
    </system_metrics>
  </point>

</points>
```

Figure 2: Example of a knowledge configuration file; it refers to the block *kernel1* in Figure 1 (Knowledge1.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<points xmlns="http://www.multicube.eu/" version="1.3" block="foo">

  <dictionary param_name="compiler_flag">
    <value string="-O2" numeric="1" />
    <value string="-O3" numeric="2" />
  </dictionary>


  <point>
    <parameters>
      <parameter name="compiler_flag" value="1"/>
    </parameters>
    <system_metrics>
      <system_metric name="time" value="123"/>
    </system_metrics>
  </point>

</points>
```

Figure 3: Example of a knowledge configuration file with string values for software knobs

it may provide the standard deviation of the measure, using the XML tag *standard_dev*. The name of each metric must be an unique valid C++ enumerator. The name is not case sensitive.

The attribute *value* (and *standard_dev*) of each field of the Operating Point must be a numeric value. If the user requires strings as a value of a software knob, then it must also define a dictionary that translates the string to a numeric value. In particular, for each non-numeric software knob, the user must define a *dictionary* XML element. The dictionary is composed by a list of *value* elements that must define two attributes: the string value (with the attribute *string*) and its related numeric value (with the attribute *numeric*). Figure 3 shows an example of knowledge file, where the software knob "compiler_flag" requires string values. In the current version is not supported having string values for any metric of interest.

If the application knowledge takes into account features of the input, it is possible to enrich the definition of the Operating Points with that information. In particular, all the input features of each Operating Point are represented by the XML tag *features*. Each data feature is represented by the XML tag *feature*; the attribute *name* represents the name of the data feature, while the attribute *value* represents the numerical value of the feature.

5 Adaptation file overview

This section describes the adaptation XML file, which define the Monitor and Plan elements for every block of code that will be tuned by mARGOt. The root element of the XML configuration file is the *margot* tag, which holds the definition of all the block of code of

```xml
<?xml version="1.0" encoding="UTF-8"?>
<points xmlns="http://www.multicube.eu/" version="1.3" block="foo">


  <point>
    <parameters>
      <parameter name="compiler_flag" value="1"/>
    </parameters>
    <system_metrics>
      <system_metric name="time" value="123"/>
    </system_metrics>
    <features>
      <feature name="size_x" value="100"/>
      <feature name="size_y" value="10" />
    </features>
  </point>

</points>
```

Figure 4: Example of a knowledge configuration file with data features

```xml
<margot application="my_application_name" version="my_version">

  <block name="kernel1">
    <!-- The configuration of block kernel1: -->
    <!--     - Description of the Monitor element -->
    <!--     - Description of the Plan element -->
  </block>

  <block name="kernel2">
    <!-- The configuration of block kernel2: -->
    <!--     - Description of the Monitor element -->
    <!--     - Description of the Plan element -->
  </block>

</margot>
```

Figure 5: Minimal adaptation configuration file.

the application. Optionally, the user is encouraged to specify the attributes *application* and *version* to provide further information about the configuration. The configuration of each block of code is inserted in the *block* tag, which has also the attribute *name* which holds the name of the block of code. The name of the block must be a valid C++ namespace identifier. Figure 5 shows a simple example on how it is possible to generate an empty configuration for the blocks depicted in Figure 1.

Since each block of code is independent from each other, in the remainder of the documentation we focus only in a single block of code, without losing in generality.

5.1 Describing the monitor element

The description of the Monitor elements consists in the list of Monitors that observe a metric of interest for the developer. The mARGOt framework ships with a suite of monitors for the most common metrics. However, the configuration file is able to handle custom monitors as well.

In particular, Figure 6 shows an example of the full declaration of a monitor that observe a metric of interest. Each tag *monitor* states the characteristic of the monitor of interested and it must specify the identifier of the monitor (with the attribute *name*) and its type (with the attribute *type*). The identifier of the monitor must be a valid C++ identifier. The type of the monitor is an enumeration of all the available monitor in mARGOt, plus the *custom* enumerator which represents a user-defined monitor.

In current implementation, the available monitor in mARGOt are:

Frequency	For the frequency monitor
Memory	For the memory monitor
PAPI	For the PAPI monitor
CPUPROCESS	For the process CPU usage monitor
CPUSYSTEM	For the system-wide CPU usage monitor
Temperature	For the temperature monitor
Throughput	For the throughput monitor
Time	For the time monitor
Collector	For the ETHz monitoring framework
ENERGY	For the energy monitor, using RAPL
ODROID_POWER	For the power monitor on Odroid
ODROID_ENERGY	For the energy monitor on Odroid

```
<monitor name="my_custom_monitor" type="custom">

  <spec>
    <header reference="my_monitor.hpp" />
    <class name="MyCustomMonitor" />
    <type name="double" />
    <stop_method name="my_stop" />
    <start_method name="my_start" />
  </spec>

  <creation>
    <param name="window_size">
      <fixed value="1" />
    </param>
  </creation>

  <start>
    <param>
      <local_var name="start_param" type="int" />
    </param>
  </start>

  <stop>
    <param name="error">
      <local_var name="error" type="double" />
    </param>
  </stop>

  <expose var_name="avg_quality" what="average" />

</monitor>
```

Figure 6: Example of the definition of a monitor element.

The type of the monitor is not case sensitive. If the type of the monitor is *custom*, it means that the developer is planning to use a monitor which is not shipped with the framework. For this reason it should specify also the monitor specification from a C++ point of view. The latter is defined within the XML tag *spec*. In particular, the developer should specify:

1. The header of the C++ which define the monitor object, using the *reference* attribute of the XML tag *header*
2. The name of the C++ class, using the *name* attribute of the XML element *class*
3. The type of the observed values stored in the monitor, using the *name* attribute of the XML tag *type*
4. If any, the name of the method that starts a measure, using the attribute *name* of the XML tag *start_method*
5. If any, the name of the method that stops a measure, using the attribute *name* of the XML tag *stop_method*

The remainder of the specification for the monitor have two purposes: defining the **input** and **output** variables for the monitor. In particular, the XML tag *creation* states all the parameters of the C++ constructor of the monitor. The XML tag *start* states all the parameters of the C++ method that starts a measure. Finally, the XML tag *stop* states all the parameters of the C++ method that stops a measure.

Syntactically, all the parameters are specified by the XML tag *param*, however its actual content depends on the nature of the parameter. In particular, if the parameter is an immediate value, then it is represented by the attribute *value* of the XML tag *fixed*. If the parameter must be an l-value or it is not known at compile time, it is possible to forward its value to the developer, relying on function parameters exposed to the developer. In this case the developer must specify the XML tag *local_var*, which is defined by the C++ type of the parameter (using the attribute *type*) and the name of the parameter (using the attribute *name*). Semantically, the parameters depends on the monitor, see A for a complete list of parameter for each monitor shipped with mARGOt.

Since each monitor collects the observations in a circular buffer, the output variables for the monitor are the statistical properties of interest for the developer (i.e. the average). The XML tag *expose* identifies such property. In particular the attribute *name* states the name of the variable that holds such value (therefore it must be a valid C++ identifier), while the attribute *what* identify the target statistical property. The following is the list of the available statistical properties:

AVERAGE To retrieve the mean value of the observations.

STDDEV To retrieve the standard deviation of the observations.

MAX To retrieve the maximum value of the observations.

MIN To retrieve the minimum value of the observations.

```
<!-- SW-KNOB SECTION -->
<knob name="num_threads" var_name="threads" var_type="int"/>
<knob name="num_trials" var_name="trials" var_type="int"/>

<!-- METRIC SECTION -->
<metric name="throughput" type="float" distribution="yes"/>
<metric name="quality" type="float" distribution="no"/>

<!-- FEATURE SECTION -->
<features distance="euclidean">
  <feature name="size_x" type="int" comparison="LE"/>
  <feature name="size_y" type="int" comparison="-"/>
</features>
```

Figure 7: Example of definition for the geometry of the application knowledge.

5.2 Describing the geometry of the application knowledge

The autotuning framework leverage as much as possible, compile-time information to specialize the definition of the manger. In particular, it requires to know the geometry of the application knowledge, i.e. the software knobs, the metrics of interest and the data features (if any). Without those information, it is possible to use the high-level interface for profiling purpose, but no manager will be instantiated.

Software-knob section

This section relates the software-knob states in the application knowledge, with C++ variables that must be exported to the application in the update function. In particular, each software knob is represented by the XML tag *knob* and must define:

- the attribute *name*, which relates to the name of a software knob in the application knowledge
- the attribute *var_name*, which is the name of the exported variable, therefore it must be a valid C++ identifier.
- the attribute *var_type*, states the C++ type of the exported variable.

These information – together with the the data features (if any) – are used to compose the signature of the update function, exposed to the user.

Metric section

This section define the metrics of interest. In particular, each metric is represented by the XML tag *metric* and must define:

- the attribute *name*, which attach a label to the metric index

- the attribute *type*, shall define the minimum C-type required to express the metric in the application knowledge.
- the attribute *distribution*, which states whether the metric is a distribution or not. If a metric is declared as distribution, the application knowledge shall define also the standard deviation of the metric. If a metric is not declared as distribution, any information about its standard deviation in the application knowledge is discarded.

Heterogeneous metrics will be automatically handled when the high-level library is generated.

Feature section

This section is optional and defines the input features of the block of code and how to handle them. In particular, all the features are represented by the XML tag *features*. The latter shall define the attribute *distance*, which states how the framework should compute the distance between data features. In the current version, the available values are:

NORMALIZED For computing a normalized distance between data feature, useful if the data features have values which differs by orders of magnitudes

EUCLIDEAN For computing a classic euclidean distance.

As child of the *features* element, each XML tag *feature* represents a data feature. The latter element shall define:

- the attribute *name*, which relates to the name of a software knob in the application knowledge and the name of the variable exposed to the update function. Therefore should be a valid C identifier.
- the attribute *type*, shall define the minimum C-type required to express the metric in the application knowledge. This will be also the type of the variable exposed to the update function.
- the attribute *comparison*, which states the validity comparison for the defining data feature. It might assume the following values:

  GE To express the fact that the defining data feature of the selected feature cluster must be greater or equals to the value of the current input.

  LE To express the fact that the defining data feature of the selected feature cluster must be less or equals to the value of the current input.

  \- To express the fact that there is no validity requirement for the defining data feature.

These information – together with the software knobs – are used to compose the signature of the update function, exposed to the user.

```
<agora address="tcp://127.0.0.1:1883" username="" password="" qos="2"
      doe="full_factorial" observations="5" broker_ca="" client_cert="" client_key="">
  <explore knob_name="num_threads" values="1,2,3,4"/>
  <explore knob_name="skip_factor" values="5,6,7"/>
  <predict metric_name="time_ms" prediction="crs" monitor="my_time_monitor"/>
  <predict metric_name="error" prediction="crs" monitor="my_error_monitor"/>
  <given feature_name="size_x" values="10,20,30"/>
  <given feature_name="size_y" values="10,20,30"/>
</agora>
```

Figure 8: Example of information definition for remote application local handler.

5.3 Online Design Space Exploration

The autotuning framework provides to the end-user a mechanism to perform a design space exploration at runtime. In particular, on one hand a remote application handler typically runs in a dedicated machine and acts as a server. Each application may contact the remote application handler to let it handle the Design Space Exploration on its behalf. On the other hand, each instance of the application start a service thread, named local application handler, that communicate with the server and manipulates the application knowledge. In this way the server will dispatch the configuration of the Design of Experiment to the available instance of the application to shorten the time to knowledge. Once the application knowledge is available, the server dispatch it to each application to start the exploitation phase.

If the end user would like to use this mechanism, it is required to state the information of the Design Space Exploration, as depicted in Figure 8. In particular, the XML tag *agora* represents the local application handler and shall define:

- the attribute *address* with the URI of the MQTT broker, in the form “<protocol>://<address>:<port>”, where the supported protocols are *tcp* and *ssl*.
- the attributes *username* and *password* for the authentication. Leave empty if it is not required.
- the attribute *qos* to define the quality of service of the MQTT protocol, which is integer between 0 and 2.
- the attribute *doe* to define the name of the Design of Experiments (DoE) technique used by the Design Space Exploration.
- the attribute *observations* to define the number of times to repeat each configuration in the DoE.
- the attribute *broker_ca* with the path to the broker CA. Leave empty if it is not required.
- the attribute *client_crt* with the path to the client certificate. Leave empty if it is not required.

- the attribute *client_key* with the path to the client key. Leave empty if it is not required.

Since the remote application handler is designed to interact with unknown applications, the *agora* element must specify information about the available range of software knobs, the required range of data features to bring in the model and the way to observe and predicts the target metrics. In particular, for each software knob of the block, the user must insert the XML tag *explore* with the name of knob (attribute *knob_name*) and the range of possible values (attribute *values*) as coma separated list of numbers. Moreover, for each data features, the user must specify the XML tag *given* with the name of the data feature (attribute *feature_name*) and the values to include in the knowledge base (attribute *values*) as a coma separated list of numbers. Finally, for each metric, the user must specify the XML tag *predict* with the name of the metric (attribute *metric_name*), the name of the plugin used to perform the prediction (attribute *prediction*) and the name of the monitor that observe that metric at runtime (attribute *monitor*).

5.4 Describing the plan element

The main goal of the plan element to state the concept of "best" for the application developer. Internally, mARGOt represent such definition as a constrained multi-objective optimization problem. Since the application might have different definitions of "best", according to its evolution or external events, it is possible to state more than one optimization problem (or states in mARGOt context).

Figure 9 shows an example of a definition of a plan element. In particular, it includes the definition of two sections: the goal section and the optimization section. These two sections are related with each other, with the geometry of the application knowledge, the monitors stated in the definition of the monitor element and the application knowledge. The remainder of this section explains in details such relations.

Goal section

This section states all the goals that the developer would like to express. Each goal represents a target that must be reached (i.e. $< subject >$ must be $< comparison_function >$ than $< value >$) The mere definition of a goal does not influence the selection of the configuration.

A goal is represented by the XML tag *goal*. The attribute *name* identify the goal and it has to be a valid C++ identifier. The attribute *value* states the actual numeric value of the goal. The attribute *cFun* states the comparison function of the goal. The available comparison functions are:

GT For the "greater than" comparison function.

GE For the "greater or equal than" comparison function.

LT For the "less than" comparison function.

```
<!-- GOAL SECTION -->
<goal name="threads_g" knob_name="num_threads" cFun="LT" value="2" />
<goal name="quality_g" metric_name="quality" cFun="LT" value="80" />

<!-- RUNTIME INFORMATION PROVIDER -->
<adapt metric_name="exectime" using="my_time_monitor" inertia="1" />

<!-- OPTIMIZATION SECTION -->
<state name="my_optimization_1" starting="yes" >
  <minimize combination="linear">
    <knob name="num_threads" coef="1.0"/>
    <metric name="exectime" coef="2.0"/>
  </minimize>
  <subject to="quality_g" priority="30" />
</state>

<state name="my_optimization_2" >
  <maximize combination="geometric">
    <metric name="quality" coef="1.0"/>
  </maximize>
  <subject to="threads_g" priority="10" />
  <subject to="exectime_g" priority="20" confidence="3" />
</state>
```

Figure 9: Example of the definition of the plan element.

LE For the “less or equal than” comparison function.

For the generation of the high-level interface, the goal subject is a field (either a metric or a software-knob) of the application knowledge. In particular, if the goal targets a metric, it must define the attribute *metric_name*, which holds the name of a metric in the related Operating Point (see Section 4). If the goal targets a software-knob, it must define the attribute *knob_name*, which holds the name of a software-knob in the related Operating Point.

Runtime adaptation section

This section states all the feedback loops required by the application developer. In particular, mARGOt might use runtime information from the monitors to adapt the application knowledge. Therefore, it is possible to state that a given metric is observed by the related monitor by using the XML tag *adapt*. The latter XML element shall define the following attributes:

- *metric_name* to identify the target metric in the application knowledge geometry. In particular, its value shall be the name of a metric stated in the metrics section.
- *using* to identify the target monitor. In particular, its value shall be the name of the monitor stated in the monitors section.

- *inertia* to define the inertia of the adaptation. This field shall be a positive integer number.

Optimization section

This section states the list of states, which is the list of constrained multi-objective optimization problems. In particular, each state is represented by the XML tag *state*, using the attribute *name* to identify it. If the developer specify more than one state, he also have to specify which is the starting state, using the attribute *starting* and setting its value to "yes".

Each state is composed by an objective function that must be maximized or minimized and the list of constraints. The objective function is represented by either the XML tag *maximize* or *minimize*, according to the requirements of the application. Since the objective function is a composition of fields (thus metrics and/or knobs), the attribute *combination* specify the composition formula. In the current implementation are available two kind of compositions:

1. a geometrical combination, represented by the value "geometric"
2. a linear combination, represented by the value "linear"
3. a simple combination, i.e. it targets only a single field, represented by the value "simple"

As child of the objective function tag, there is the list of fields that compose the objective function. In particular, the XML tag *metric* represents a metric of the application knowledge, specified by the attribute *name*; while the XML tag *knob* represents a software-knob of the application knowledge, specified by the attribute *name*. Both cases require to define the attribute *coefficient*, which states the numeric value of the coefficient of each term in the combination. This hold true also in case of a "simple" combination.

The list of constraints is represented by the list of XML tags *subject*, using the attribute *to* to specify the related goal name (see Section 5.4), the attribute *priority* to specify the priority of constraint and (optionally) the attribute *confidence* to specify the confidence level of the constraint. The priority is a number used to order by importance the constraints, therefore it is required that each constraint of a state must have a different priority. If the constraints relates to a goal whose subject is a monitored value, it is required to specify the name of the related metric in the application knowledge, using the attribute *metric_name*. The confidence is the number of times to take into account the standard deviation in the Operating Point evaluation, therefore it must be an integer number.

6 mARGOt command line interface

The mARGOt command line interface (margot_cli) is a python script to provide to the end-user utility functions to manage the mARGOt heel configuration files. The usage of the script is explained in the its help command. The most important command is *generate*

which automatically generates the source code of the high level interface, from the XML configurations files. This is the command leveraged by margot_if to generate the high-level interface library. Beside this command, margot_cli expose the following operations to manage the application knowledge:

plotOPs It takes in input the XML configuration file of the application knowledge and it prints on the standard output a gnuplot script that display the application knowledge, according to the fields specified at command line.

pareto It takes in input the XML configuration file of the application knowledge and it prints on the standard output the XML configuration file of application knowledge with only the Operating Points which belong to the Pareto set, according to the criteria specified at command line.

csv2xml It takes in input the CSV configuration file of the application knowledge and it prints on the standard output the XML configuration file of application knowledge.

xml2csv It takes in input the XML configuration file of the application knowledge and it prints on the standard output the CSV configuration file of application knowledge

The CSV format of the application knowledge shall represents the Operating Point list using the following conventions. Each row is considered a different Operating Point. Each column is considered a field of the Operating Point. All the column of the file shall have an header to identify the field:

- Each software knob shall have the prefix "#". For example, to represent the knob "compiler_flag", the header name should be "#compiler_flag".
- The metric mean value and standard deviation must be in two separated columns. In particular, the column with the mean value of the metric is written in plain, while its standard deviation shall have the "_standard_dev" suffix. For example to represent the mean value and the standard deviation of the metric execution time, the header of the column that represents the mean value is "exec_time", while the header of the column that represents its standard deviation is "exec_time_standard_dev".
- Each data feature shall have the prefix "@". For example, to represent the data feature "size", the header name should be "@size".

All the required suffixes and prefixes will be stripped from the actual name of the field.

A Monitor implementation details

This appendix lists, for each monitor, all the parameters of its constructor, start measure method and stop measure method. These information might be used when a monitor is declared in the XML configuration file.

A.1 Collector Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom connection constructor
    - Name of the topic of interest (string)
    - Address of the MQTT broker (string)
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.2 Energy Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom domain constructor
    - Domain of interest
        (a) margot::EnergyMonitor::Domain::Cores
        (b) margot::EnergyMonitor::Domain::Uncores
        (c) margot::EnergyMonitor::Domain::Ram
        (d) margot::EnergyMonitor::Domain::Package

- Size of the observation window (default = 1)
- Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.3 Frequency Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.4 Odroid Energy Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom period time
    - Period of polling (unsigned integer)
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.5 Odroid Power Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.6 Papi Monitor

Constructor parameters

1. Trivial Default constructor

    N/A

2. Custom event constructor
    - PAPI event of interest
        (a) margot::PapiEvent::L1_MISS
        (b) margot::PapiEvent::L2_MISS
        (c) margot::PapiEvent::L3_MISS
        (d) margot::PapiEvent::INS_COMPLETED
        (e) margot::PapiEvent::NUM_BRANCH
        (f) margot::PapiEvent::NUM_LOAD
        (g) margot::PapiEvent::NUM_STORE
        (h) margot::PapiEvent::CYC_NO_ISSUE
        (i) margot::PapiEvent::CYC_TOT
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.7 Process CPU Usage Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom CPU time counter constructor
    - CPU time counter
        (a) margot::CounterType::SoftwareCounter
        (b) margot::CounterType::HardwareCounter
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.8 System CPU Usage Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.9 Temperature Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A

A.10 Throughput Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

1. Amount of data processed (float)

A.11 Time Monitor

Constructor parameters

1. Default constructor
    - Size of the observation window (default = 1)
    - Minimum number of observations (default = 1)
2. Custom time unit constructor

- Time resolution
  (a) margot::TimeUnit::NANOSECONDS
  (b) margot::TimeUnit::MICROSECONDS
  (c) margot::TimeUnit::MILLISECONDS
  (d) margot::TimeUnit::SECONDS
- Size of the observation window (default = 1)
- Minimum number of observations (default = 1)

Start method parameters

N/A

Stop method parameters

N/A