H2020-FETHPC-1-2014 ANTAREX-671623

AutoTuning and Adaptivity approach for Energy efficient eXascale HPC systems

http://www.antarex-project.eu/

Deliverable D3.4: Final Application-Level Self-tuning Framework and APIs

Deliverable Title:	Final Application-Level Self-tuning Framework and APIs		
Lead beneficiary:	Politecnico di Milano (Italy)		
Keywords:	Autotuner framework, prototype, accompanying report		
Author(s):	Davide Gadioli (POLIMI), Emanuele Vitali (POLIMI), Gianluca Palermo (POLIMI), Cristina Silvano (POLIMI)		
Reviewer(s):	Pedro Pinto (UPORTO), Daniele Cesarini (ETHZ).		
WP:	WP3	**Task:**	T3.2
Nature:	Other	**Dissemination level:**	Public [1]
Identifier:	D3.4	**Version:**	V1.4
Delivery due date:	May 31, 2018	**Actual submission date:**	June 14, 2018

Executive Summary:

This document is the accompanying report associated with the **Deliverable D3.4** released as the **mARGOt autotuning framework** together with the APIs to interface with the application, the application-level monitoring framework, the related DSL to describe the autotuning framework and the engine for on-line learn the application behaviour (AGORA).

The mARGOt framework represents the results of the activities carried out by the project partners in **Task 3.2 "Application Autotuning"** under the leadership of POLIMI.

This deliverable is organized as follows:

- **Section 1** describes the features of the initial prototype of the framework;
- **Section 2** describes the code organization in the repository;
- **Section 3** describes how to install the framework;
- **Section 4** describes a usage example;
- **Section 5** presents a summary the major updates on the mARGOt functionalities introduced with respect to the previous releases.

The self-tuning framework repository is released open-source under the LGPGL v.2.1 license, publicly available at https://gitlab.com/margot_project/core

As Appendix of the Deliverable, there is the mARGOt paper submitted to IEEE Transactions on Computers.

Approved and issued by the Project Coordinator: 	**Date:** June 14, 2018

Project Coordinator: Prof. Dr. Cristina SILVANO – Politecnico di Milano
e-mail: silvano@elet.polimi.it - **Phone:** +39-02-2399-3692- **Fax:** +39-02-2399-3411

[1] This version is an accompany report to the release of the initial prototype of the mARGOt framework.

Table of Contents

1 Final Application-Level Self-tuning Framework and APIs

The focus of this deliverable is the dynamic autotuning framework, named mARGOt, representing the result of the activities carried out in Task 3.2.

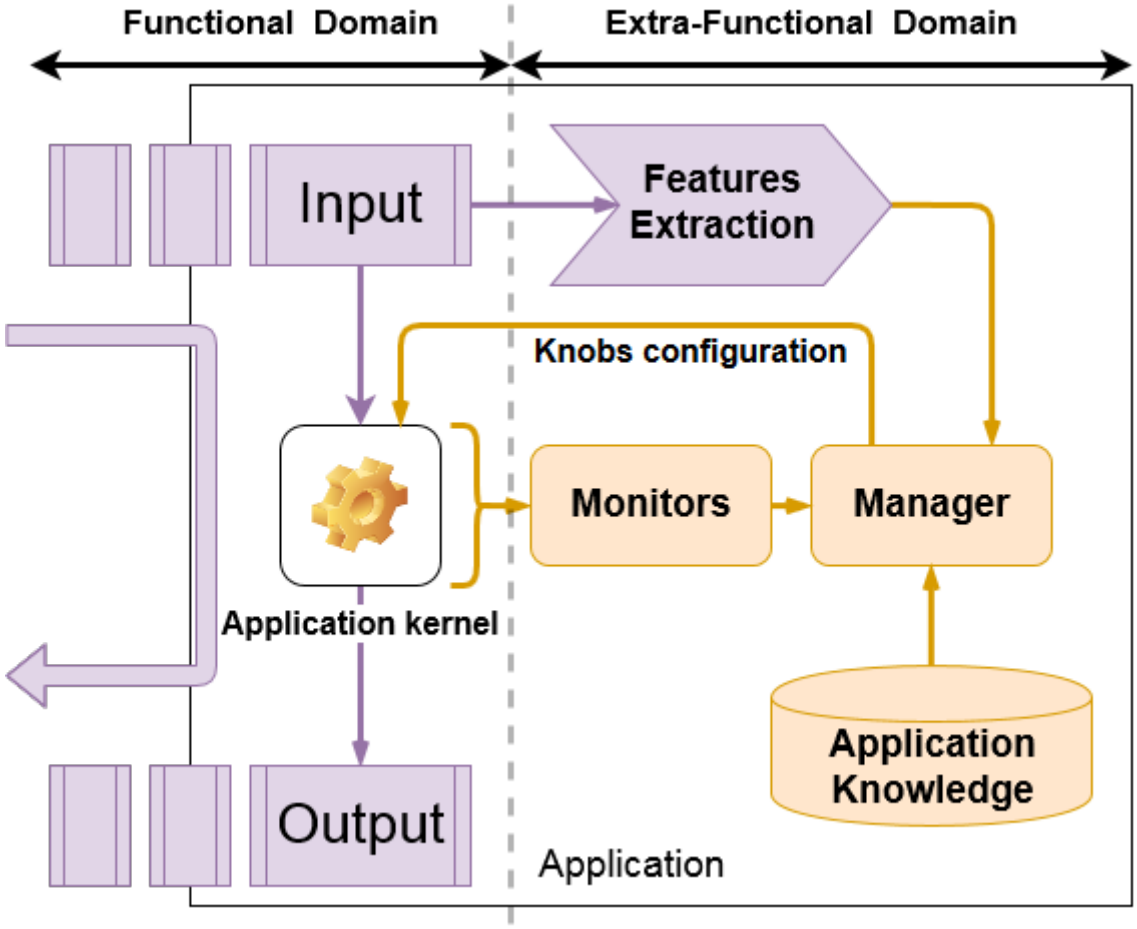

Figure 1

Three modules compose the framework: the application manager, the application monitors and the application knowledge. Application monitors gather new insight on the actual behaviour of the application at runtime; Application manager is in charge of selecting the most suitable configuration, according to application requirements; Application knowledge describes the extra-functional behaviour.

Figure 1, depicts the overview of the current implementation of the framework. In particular, mARGOt is a C++ library, which must be included to the target application (the white box) at linking time to add self-optimization capabilities. In this way, each instance of the application is able to take autonomous decisions.

Although mARGOt is agnostic about the structure of the application, in the remainder of the document we assume that it is composed by one kernel (the purple elements)

that elaborates a set of Input Data. The region of code that performs such elaboration, named "*block*" in the framework, is the one tuned and profiled by the framework (the orange elements). However, mARGOt is able to profile and tune more than one *block* of code of the application, if it is required.

1.1 Application-specific monitors

To react to changes in the execution environment, the autotuner framework relies on application-specific monitors to observe the value of the metric of interest for the application.
The framework ships a suite of monitors to observe the most common metrics. However, it is trivial to implement a custom monitor to observe an application-specific metric (e.g. the accuracy), since the monitor infrastructure uses a modular design.
In particular, the framework provides the following monitors:

- **Time** monitor: it uses the *std::chrono* environment. The user is able to select the granularity of the measure (between µs, ms and s)
- **Throughput** monitor. The unit of measure is [data/sec], where the user specify the amount of data at each measure.
- **Frequency** monitor: parses the *"scaling_cur_freq"* metafile
- **Temperature** monitor: uses the *libsensors* facilities and retrieves the average temperature of the cores.
- **System CPU usage** monitor: parses the *"/proc/stat"* metafile
- **Process CPU usage** monitor: it uses either hardware counter or the *rusage* facilities, depending on the user preferences
- **PAPI** monitor: wrapper to the *PAPI* hardware counter monitor
- **Memory** monitor: it parses the *"/proc/self/statm"* metafile
- **Energy** monitor: it uses the *Intel/RAPL* facilities. The user is able to specify the domain of interest.
- **Collector** monitor: it is a proxy for the Monitoring Framework developed in T3.1

1.2 Application knowledge

The application knowledge describes the interaction between software-knobs, metrics of interest and input data features as list of Operating Points. In particular, each Operating Point relates a software-knobs configuration with the reached performance and with the input data feature cluster.

The framework is agnostic about the geometry of the Operating Point, meaning that application developers are able to define an arbitrary number of software-knob, metrics of interest and input features.

Application developers may derive the application knowledge at design time or at runtime. In the first case, it is possible to use a well-known technique analysed in the state-of-the-art. In the second case, it is possible to use the Agora framework to learn the application knowledge at runtime.

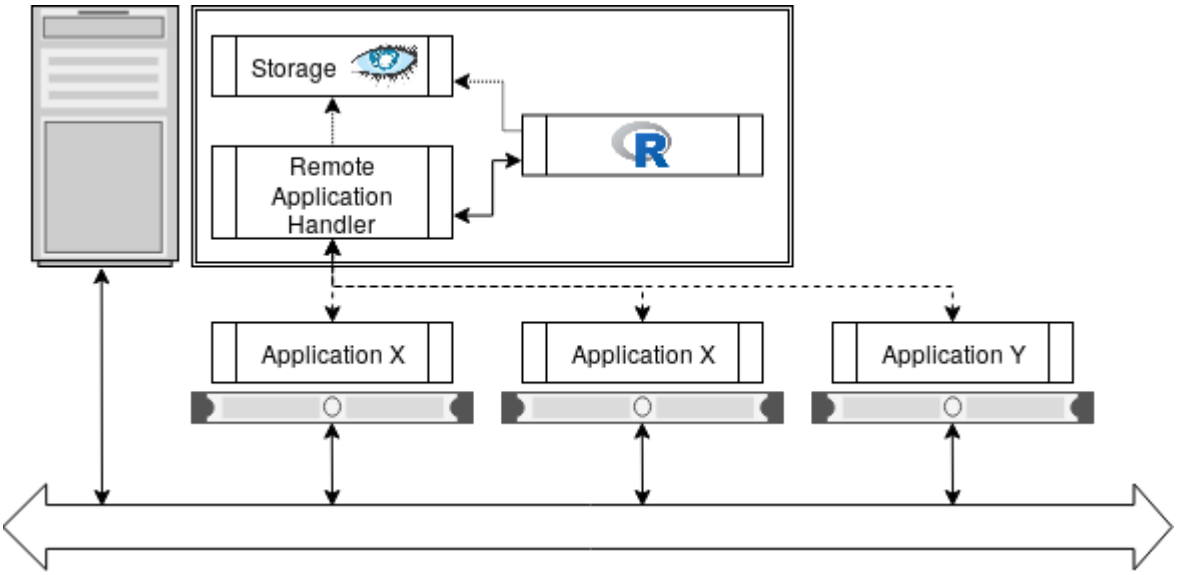

Figure 2

Figure 2 shows the overall structure of Agora. Two elements compose the framework: the Remote Application Handler (server) and the Local Application Handler (client).

The Local Application Handler is a service thread running along each instance of the application and it performs two main tasks: it manipulates the application knowledge to force mARGOt to select the evaluated configuration and it sends to the Remote Application Handler telemetry information. It uses the MQTT or MQTTs protocols to communicate with the server, therefore outside of the typical MPI protocol.

The Remote Application Handler is a thread-pool that interacts with each application instance to learn the application model. It uses the Cassandra database to store data and it leverages state-of-the-art approach to model application behaviour. The common workflow is as follows:

1. Each client notifies its existence to the server.
2. The server asks for information regarding the application, such as software-knobs domain or the required Design of Experiment.
3. After computing the required configuration to evaluate, it starts the Design Space Exploration, dispatching configuration to clients, in a round robin fashion.
4. Once the last configuration has been evaluated, the server computes the application knowledge to broadcast to clients.

The Agora framework is designed to be resilient to clients or server crashes; moreover, a new client might join the computation in any time, contributing to the DSE or directly receiving the application model.

The benefits of this component are two-fold. On the main hand, it enables a distributed DSE, leveraging the platform parallelism to decrease the time-to-knowledge. On the other hand, it is possible to use standard visualization tools to display high-level execution trace or the derived application model.

1.3 Application manager

This is the core element of the mARGOt framework, since it is in charge of selecting the most suitable configuration. To achieve this result, application manger takes into account four kinds of information:

1. The application knowledge
2. The application requirements
3. (Optionally) The feedback information from the monitors
4. (Optionally) The features of the actual input

The application knowledge contains the expected behavior of the application. The application requirements define the concept of "most suitable" from the point of view of end-users. In particular, it is defined as a multi-objective constrained optimization problem. This definition might involve any metric or software knob in the application knowledge. The user might express different requirements and she/he can choose, at runtime, the one that suites the current situation.

The feedback information reported from monitors is used to adjust the application knowledge according to the evolution of the execution environment. In particular, it uses a linear error propagation.

Optionally, the application manager uses data features of the current input to classify the current input according to feature clusters, identifying additional optimization opportunities. For example, if the current input is deemed "simple", we may spare unneeded computation effort.

The framework enables dynamic adaptation, since all the previous information might change at runtime. In particular, it is possible to:

- Add / Remove Operating Points at runtime
- Add / Remove constraints from the application requirements
- Change the value of a constraint
- Change the objective function to be maximized/minimized
- Switch to a different requirement.
- Add / Remove feature clusters
- Select the most suitable feature cluster for the current data.
- Start the Application Local Handler.

Since the time that the application manager takes to select the most suitable configuration is stolen from the application, the framework must be lightweight. For this reason, the application manager is designed to minimize the introduced overhead.

2 Framework repository

We have released the implementation of the mARGOt framework in a public Gitlab repository (https://gitlab.com/margot_project/core). It also includes the Agora framework as an optional component.

The repository is organized as follows:

- /framework -> contains the source code of the mARGOt library
- /agora -> contains the source code of Agora server executable
- /margot_heel -> contains an high-level interface generator
- /doc -> contains the user manual that explain how to use the framework
- /framework/doc -> contains the sources of the Doxygen documentation

The framework is versatile, which means that it exposes to the application designer a high degree of customization. In particular, the application designer should provide the Operating Point definition (which are the metrics of interest, the software knobs and the input features), define the application requirements and choose the monitors to use at runtime.

To perform these operations, the application designer must have a deep knowledge of framework APIs. For this reason, we also provide a python script that generates a high-level interface from XML configuration files. The advantages of this approach are two-fold. On one hand, it eases the integration process. On the other hand, it provides separation of concerns between the functional part (the application code) and the extra-functional behaviour (the XML files).

In particular, the high-level interface generates the following functions:

1. An initialization function, that creates all the required objects
2. An update function, to retrieve the most suitable configuration
3. A function that starts all measures on the application monitors
4. A function that stops all the measures on the application monitors
5. A function that logs the framework decisions and the observed metrics

In the ANTAREX project, we use a DSL to automatically integrate the autotuner in the target application. The high-level interface enables a reuse of the LARA strategies, since it hides all the implementation details.

The high-level interface supports several independent regions of code to be tuned. Besides the "init" function, which is unique per application, *margot_heel* generates a high-level interface for each block of code.

3 How to install the framework

Dependencies:
The framework is completely written in C++ 11. However, the implementation of several monitors assumes that the Operating System is Unix-like. Moreover, some monitors rely on external components to retrieve the target information.
Therefore, the framework itself requires only **cmake** and a recent compiler (e.g. **g++** > 5.0). However, some monitors require additional libraries (optional):

- PAPI framework: to monitor the performance counters
- pfm library: a dependence of PAPI
- Sensors library: to monitor the temperature
- Examon: to use the Collector proxy

Moreover, mARGOt ships with a unit to test the built libraries (optional). The test unit requires **cxxtest** to generate the actual executable. If we want to generate a doxygen documentation from the source files, we also need the doxygen environment (optional).

If application developers plan to use the Agora framework, other dependencies are required:

- An internet connection to download the Cassandra driver and the MQTT client
- The OpenSSL library
- The libuv library
- Support for pthreads (should be already available on any Linux distribution)

On Fedora/CentOS, these dependencies can be installed with dnf/yum:

```
    >#  dnf  install  cxxtest  papi-devel  libpfm-devel  lm_sensors-devel  doxygen
libuv-devel openssl-devel
```

On Ubuntu, these dependencies can be installed with apt:

```
    >#  apt  install  cxxtest  papi-tools  libpapi-dev  libpfm4-dev  libsensors4-dev
doxygen libuv1-dev libssl-dev
```

Please, note that all the above packages are optional. They are only required if the related compile option is activated. However, for the usage example in this section they are not required.

Framework installation procedure:
The first step is to clone the git repository: create a mARGOt folder and run the git clone from there:

```
    >$ mkdir margot
    >$ cd margot
    >$ git clone https://gitlab.com/margot_project/core.git
```

After cloning the repository, we can compile the framework (assuming to install the framework in *<path>*)

```
>$ cd core
>$ mkdir build
>$ cd build
>$ cmake -DCMAKE_INSTALL_PREFIX:PATH=<path> ..
>$ make && make install
```

It is possible to customize the behaviour of the framework, changing compile options in CMake. The following list states the available flags, their default value and their purpose.

Option name	Values [default]	Description
LIB_STATIC	[ON], OFF	Build a static library (otherwise it is shared)
WITH_DOC	ON , [OFF]	Generate the Doxygen documentation
WITH_TEST	ON , [OFF]	Build the cxxtest application, to test the framework
USE_COLLECTOR_MONITOR	ON , [OFF]	Include the wrapper monitor for Examon (by ETHz)
USE_PAPI_MONITOR	ON , [OFF]	Include the monitor of Perf events (using PAPI interface)
USE_TEMPERATURE_MONITOR	ON , [OFF]	Include the temperature monitor (requires lm_sensors)
WITH_AGORA	ON , [OFF]	Enable the AGORA application handler
WITH_BENCHMARK	ON , [OFF]	Build a benchmark to evaluate the overheads

Depending on build options, the framework provides several executable. In particular, assuming that we have installed the framework in *<path>*, we might have the following binaries:

- **bin/margot_test:** Run test cases to check if mARGOt behaves correctly
- **bin/agora:** The Application Remote Handler
- **bin/margot_benchmark:** An utility benchmark that evaluates the mARGOt overheads

The Remote Application Handler requires additional command line options, use "--help" for help. For reference, the following section provides a usage example.

Plugin system to learn application knowledge:

In the current version of mARGOt, we introduced a plugin system to learn the application behaviour at runtime. This means the application developers are able to easily deploy a custom method to model an application-specific metric, such as a quality metric.

However, in the repository we ship a default plugin, which uses a well-known approach to interpolate and predict the performance of the application, using splines, implemented in R.
Therefore, in order to use it (as in the usage example in Section 4), we need additional dependencies:

- A python interpreter
- The R execution environment
- The Cassandra driver in python
- The R packages:
  - crs
  - Hmisc

To install the R execution environment, we need to install the following packages:
On Fedora, there is a meta-package:

```
>$ dnf install R
```

On Ubuntu:

```
>$ apt install r-base r-cran-littler
```

On CentOS we need to enable the epel repository (or build it from sources)

```
>$ yum install epel-release
>$ yum install R
```

To install the python dependencies using pip, issue the following command:

```
>$ pip install --user --upgrade cassandra-driver
```

To install the R packages, issue those commands from an R shell:

```
install.packages('Hmisc')
install.packages('crs', dependencies = TRUE)
```

NOTE: The R package "crs" requires OpenGL headers and libraries, you might install it with the following packages

- For Fedora: mesa-libGL-devel mesa-libGLU-devel
- For Ubuntu: libgl-dev libglu-dev

Usage example

The Tutorial repository (https://gitlab.com/margot_project/tutorial.git) contains a toy application as example of a program integration. The source code of the original application is very simple:

```
1.  void do_work (int trials)
2.  {
3.      std::this_thread::sleep_for(std::chrono::milliseconds(trials));
4.  }
5.
6.  int main()
7.  {
8.      int trials = 1;
9.      int repetitions = 10;
10.     for (int i = 0; i<repetitions; ++i)
11.     {
12.         do_work(trials);
13.     }
14. }
```

The toy application calls ten times a function named *do*_work. The function exposes a software knob (named *trials*) that alters the function behaviour.

The purpose of this tutorial is to integrate mARGOt in this toy application.
In particular, the application designer wants to tune the region of code that performs the elaboration (lines 11-13). We name such region of code block ***foo***.

With respect to the example in D3.2, in this example, we use Agora to derive the application knowledge at runtime. Although not required to compile the Agora executable, for its usage we need a running Cassandra database and a MQTT broker.

In a real scenario, the Remote Application Handler (i.e. Agora) runs in a different node with respect to mARGOt. For simplicity sake, in this example we will run the database, the broker, the server and the client on the same machine.

Prerequisite installation: set up a Cassandra database and mosquito:

This step aims at starting a Cassandra cluster and at running a MQTT broker in the machine without authentication and without establishing an encrypted channel.

We need to download, to compile and to start the Apache Cassandra database. The first step is to check if we have to install the following dependency:

- Java SDK – version 1.8
- ant – At least version 1.8
- git – At least version 1.7

After that we have installed all the required dependencies, we can compile the Apache Cassandra databse, issuing the following commands (it could take a while):

```
>$ git clone git://git.apache.org/cassandra.git
>$ cd cassandra
>$ git checkout cassandra-3.11.2
>& ant release
```

To execute the database we issue the command:

```
>$ ./bin/cassandra -f
```

Note: the previous command start Cassandra in foreground, if you prefer have it in background, omit the "-f" flag

After that we have Cassandra, we need to install the MQTT broker. The easiest procedure is to install it from the repository. On Fedora issue the command:

```
>$ sudo dnf install mosquitto
```

It is possible to use system to start/stop the MQTT broker. In particular, to start the broker, issue:

```
>$ sudo systemctl start mosquitto
```

Please, notice that for the following steps, we assume to have a running Cassandra database and a running MQTT broker.

First step: set up the experiment:

At first, we get the toy application from the "online" branch of the repository:

```
>$ git clone –b online https://gitlab.com/margot_project/tutorial.git
>$ cd tutorial
```

Then we get and compile the mARGOt framework, with AGORA support (please see Section 3 for the dependencies list):

```
>$ git clone https://gitlab.com/margot_project/core.git
>$ cd core
>$ mkdir build
>$ cd build
>$ cmake -DWITH_AGORA=ON ..
>$ make
>$ cd ../..
```

We are now able to start the integration process of the mARGOt framework in the target application.

Second step: create the high-level interface:

The toy application targets only the **foo** block, which exposes only one software-knob. We assume that end-user is interested on the following extra-functional properties: "error" and "exec_time" (i.e. execution time).

We suppose also that the application requirements are the following:

minimize error
s.t. exec_time < 40 (ms)

Given these requirements, the application designer must express them in an xml file (named *autotuning.conf*), as follows:

```
<margot application="tutorial" version="v2_online">
<block name="foo">
   <!-- MONITOR SECTION -->
   <monitor name="my_time_monitor" type="Time">
      <creation>
         <param name="time_measure">
         <fixed value="margot::TimeUnit::MILLISECONDS"/>
         </param>
      </creation>
      <expose var_name="avg_time" what="average" />
   </monitor>
   <monitor name="my_error_monitor" type="custom">
      <spec>
         <header reference="margot/monitor.hpp" />
         <class name="margot::Monitor< float >" />
         <type name="float" />
         <stop_method name="push" />
      </spec>
      <stop>
         <param name="error">
         <local_var name="error" type="float" />
         </param>
      </stop>
      <expose var_name="avg_error" what="average" />
   </monitor>
   <!-- SW-KNOB SECTION -->
   <knob name="num_trials" var_name="trials" var_type="int"/>
   <!-- METRIC SECTION -->
   <metric name="exec_time" type="int" />
   <metric name="error" type="float" />
   <!-- LOCAL APPLICATION HANDLER -->
   <agora address="127.0.0.1:1883" username="" password="" qos="2" doe="full_factorial" obse
rvations="5" broker_ca="" client_cert="" client_key="">
      <explore knob_name="num_trials" values="10,20,30,40,50"/>
      <predict metric_name="exec_time" prediction="crs" monitor="my_time_monitor"/>
      <predict metric_name="error" prediction="crs" monitor="my_error_monitor"/>
   </agora>
   <!-- GOAL SECTION -->
   <goal name="my_execution_time_goal" metric_name="exec_time" cFun="LT" value="40" />
   <!-- RUNTIME INFORMATION PROVIDER -->
   <adapt metric_name="exec_time" using="my_time_monitor" inertia="5" />
   <!-- OPTIMIZATION SECTION -->
   <state name="my_optimization" starting="yes" >
      <minimize combination="linear">
         <metric name="error" coef="1.0"/>
      </minimize>
      <subject to="my_execution_time_goal" confidence="1" priority="20" />
   </state> </block> </margot>
```

All the requirements targets only the **foo** block, therefore they are contained in a single *block* element (line 2). The first section of the configuration file declares the list of the monitors deployed for the **foo** block (lines 4-25). The second section states the geometry of the Operating Point: its metrics and software-knobs (lines 27-30). The third section states the link to the Remote Application Handler and the application information (lines 32-36). The fourth section states the goal of the **foo** block (line 28). The fifth section defines the feedback information, and in this example, it states that we use the monitor "my_time_monitor" to adapt the metric "exec_time" (line 40). Finally, the fourth section states the definition of "most suitable" for the application designer of the **foo** block (lines 42-47).

In this example, we use Agora to learn the application knowledge at runtime, therefore we don't need the configuration about the Operating Points.

Using this XML configuration file, we are able to generate the high-level interface for the toy application. To perform this action, we use the margot_heel tool, and we must proceed as follow:

1. Copy the margot_heel template folder in the root of the tutorial project

```
>$ cp -r core/margot_heel/margot_heel_if/ .
```

2. Delete the default configuration files

```
>$ rm margot_heel_if/config/*.conf
```

3. Write the configuration files for the toy application, as explained before (or copy the one in the config folder of the repository)

```
>$ cp config/autotuning.conf margot_heel_if/config/autotuning.conf
```

4. Build the high-level interface

```
>$ cd margot_heel_if/
>$ mkdir build
>$ cd build
>$ cmake -DMARGOT_CONF_FILE=autotuning.conf ..
>$ make
>$ cd ../..
```

We have now a high-level interface for the application. Besides the library itself, the mARGOt framework and margot_heel generates a pkg-config and a CMake find package file, to facilitate the integration in the building system of the target application.

Third step: integrate the autotuner in the toy application code

Once the interface has been created, we can proceed with the required modification in the code of the application, as follows:

```
1.  #include <margot.hpp>
2.  #include <chrono>
3.  #include <thread>
4.
5.  int main()
6.  {
7.      margot::init();
8.      int trials = 1;
9.      int repetitions = 10;
10.     for (int i = 0; i<repetitions; ++i)
11.     {
12.         //check if the configuration is different wrt the previous one
13.         // NOTE: trials is an output parameter!
14.         if (margot::foo::update(trials))
15.         {
16.             // Writing the "trials" variable is enough to change configuration
17.             margot::foo::manager.configuration_applied();
18.         }
19.         //monitors wrap the autotuned function
20.         margot::foo::start_monitor();
21.         do_work(trials);
22.         margot::foo::stop_monitor(3.4f);
23.         //print values to file
24.         margot::foo::log();
25.     }
26. }
```

First, the margot_heel header needs to be added (line 1). Then, we need to call the margot::init() function to create all the mARGOt data structures (line 7), such as the AS-RTM and the monitors.

All the other functions are block specific, as you can see from the **foo** namespace. In particular, we:

1. **update** the most suitable configuration (lines 12- 18). Please notice that the parameters of the *update* function are output parameters. The signature of the *update* function depends on the software knobs stated in the XML file.
2. **Start** all the monitors stated in the XML file (line 20)
3. **Stop** all the monitors stated in the XML file (line 22). In this toy example we have hardcoded the error value at 3.4, regardless of the configuration.
4. (Optionally) **Log** on the standard output (and in a log file) the behaviour of the autotuner.

Fourth step: build the application

To build the application, its building system must include both the mARGOt framework and the high-level interface. In the toy application this step is already done. Thus, to compile the application we just need to:

```
>$ mkdir build
>$ cd build
>$ cmake ..
>$ make
```

Fifth step: start the Remote Application handler

In the previous steps, we built the Remote Application Handler (agora) and the tutorial application (client). The next step is to start the server, which waits for any client. Since we are running the Database and the Broker locally, without encryption nor authentications, the default parameters are correct.
Assuming that *<tutorial>* is the absolute path of the root of the tutorial repository, we can launch Agora with the command as follows:

```
      <tutorial>/core/build/agora$ ./agora --workspace_folder
<tutorial>/core/build --plugin_folder <tutorial>/core/agora/plugins
```

The first parameter is an absolute path to a folder with read and write permissions, while the second parameter is the absolute path to the plugin folder. For a complete list of parameters, run the application with *--help*.

Sixth step: execute the application

We are finally able to execute the application and see the output of the log function.
In a different terminal we can launch the tutorial application using the following commands:

```
      <tutorial>/build$ ./tutorial
```

The Local Application Handler does not block the application execution flow in any case. Therefore, we have different outputs according to the progress of the DSE.

While the Local Application Handler is negotiating the Design Space Exploration with the Remote Application Handler, the application is using the default configuration, therefore its output is:

```
      [ Knob num_trials = 1]
      [ Expected error = N/A] [ Expected exec_time = N/A]
      [ Goal my_execution_time_goal = 40]
      [ avg_time = 1] [ avg_error = 3.4]
```

Here we can notice, how the application is using the default software-knob configuration (num_trials=1), which is outside of the explored parameters (possible values = 10,20,30,40,50) stated in the configuration file. Moreover, since we don't have an application knowledge, the expected value of the metrics are not available.

Once that the Remote Application Handler computes the Design of Experiment (DoE), it starts to dispatch configuration to observe. Therefore, the clients start to evaluate the configurations and the output is as follows:

```
      [ Knob num_trials = 20]
      [ Expected error = N/A] [ Expected exec_time = N/A]
      [ Goal my_execution_time_goal = 40]
      [ avg_time = 20] [ avg_error = 3.4]
```

```
[ Knob num_trials = 40]
[ Expected error = N/A] [ Expected exec_time = N/A]
[ Goal my_execution_time_goal = 40]
[ avg_time = 40] [ avg_error = 3.4]
```

As we can notice from the output, the application is trying the DoE configurations in a round robin fashion. Once, all of them are being evaluated, the Remote Application Handler computes the application model and it broadcast the application knowledge to clients.
Therefore, the final output of the application contain the most suitable configuration for the target application and its output is the following:

```
[ Knob num_trials = 20]
[ Expected error = 3.4] [ Expected exec_time = 20]
[ Goal my_execution_time_goal = 40]
[ avg_time = 20] [ avg_error = 3.4]
```

As you can see, in the selected software-knob configuration satisfies the application requirements stated in the configuration file (my_execution_time_goal < 40).

The purpose of this toy application is merely to show an integration tutorial of the framework in a target application. To evaluate mARGOt framework in the context of real applications, in the benchmark repository (https://gitlab.com/margot_project/applications.git) there are some public applications, already integrated with the framework. In particular, the “master” branch contains the scripts that generate the application knowledge in a design time phase. The “online” branch contains the scripts that generate the application knowledge at runtime.

4 Summary of major updates from D3.2 (Initial version)

With respect to the initial prototype released in D3.2, this release of the autotuner framework introduces two key features:

- **On-line learning** of the application knowledge. In the current implementation, the application knowledge might be obtained at runtime using a state-of-the-art approach.
- **Proactive framework**. In the current implementation, the autotuning framework is able to cluster Operating Points according to data features, enabling a proactive adaptation policy.

Moreover, while the application-level API has been kept as in the previous version, most of code of the mARGOt-core framework has been rewritten to enable the previous features. The current version of the framework provides also the opportunities to make the autotuning working for ANDROID applications (as will be applied on the client side of UC2).