H2020-FETHPC-1-2014 ANTAREX-671623

AutoTuning and Adaptivity approach for Energy efficient eXascale HPC systems

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

Deliverable D3.2: Initial Application-Level Self-tuning Framework and APIs

European Commission
Horizon 2020
European Union funding
for Research & Innovation

Deliverable Title:	Initial Application-Level Self-tuning Framework and APIs		
Lead beneficiary:	Politecnico di Milano (Italy)		
Keywords:	Autotuner framework, prototype, accompany 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:	D3.2	**Version:**	V1.2
Delivery due date:	February 28th, 2017	**Actual submission date:**	March 14th, 2017

Executive Summary:

This document is the accompanying report associated with the **Deliverable D3.2** released as the **mARGOt autotuning framework** together with the APIs to interface with the application, the application-level monitoring framework and the related DSL. The mARGOt framework represents the results of the activities carried out by the project partners from M07 to M18 in **Task 3.2 "Initial Application-Level Self-tuning Framework and APIs"** 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** describes the ongoing work.

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

Approved and issued by the Project Coordinator: [illegible]	**Date:** March 14th, 2017

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 Initial Application-Level Self-tuning Framework and APIs

Figure 1 shows the overall ANTAREX architecture. The focus of this deliverable is a prototype of mARGOt, the autotuner framework, which enhances the application with an adaptation layer. While the Power Manager and the Hardware Monitor operate at system-level, the autotuner framework selects the most suitable configuration at application-level.

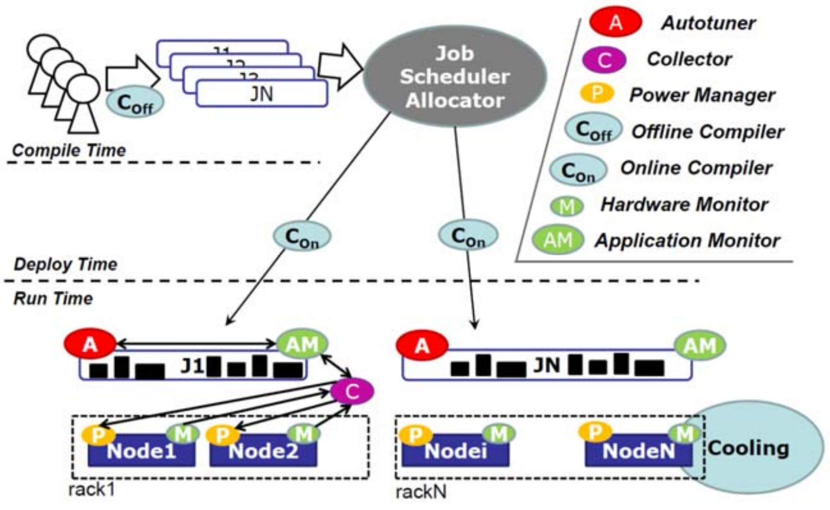

Figure 1: ANTAREX Runtime Architecture

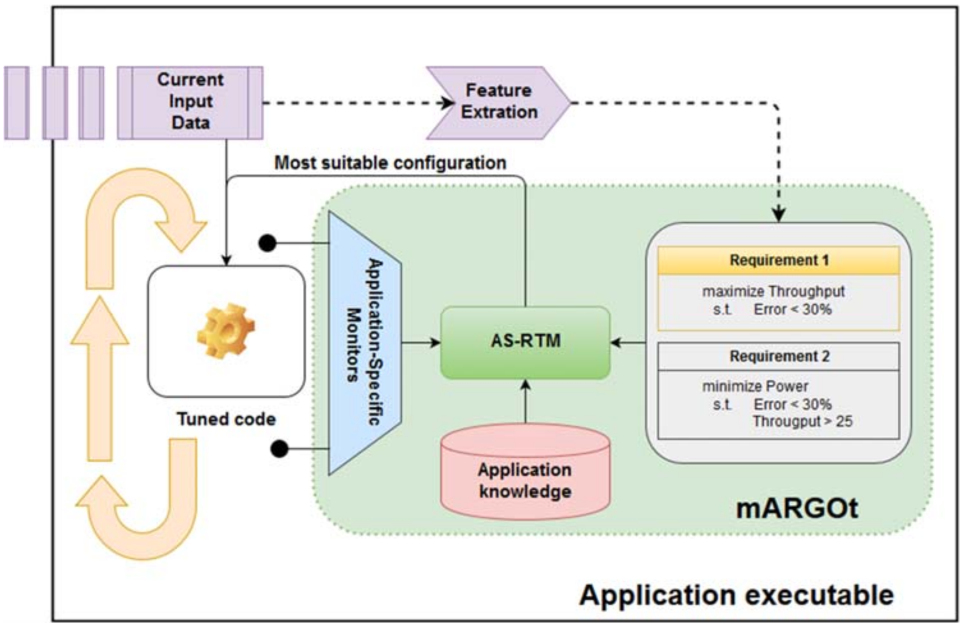

Figure 2: Overview of the mARGOt framework

Two modules compose the framework: the application monitors and the Application-Specific Run-Time Manager (AS-RTM). The application monitors gather new insight on the actual behaviour of the application at runtime, while the AS-RTM is in charge of selecting the most suitable configuration, according to the application requirements.

Figure 2 depicts the overview of the current implementation of the framework. In particular, mARGOt is a C++ library (the dashed green box), which must be included to the target application (the white box) at linking time to add autotuning capabilities. In this way, each instance of the application is able to take autonomous decisions.

The framework is agnostic about the structure of the application. For the remainder of the document, we assume that the target application must elaborate a set of Input Data (the purple box). The region of code that performs such elaboration, named "*block*" in the framework, is the one tuned and profiled by the framework (the box with a yellow gear). The framework is able to profile and tune more than one *block* of code of the application, if it is required.

1.1 The 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 The application-specific runtime manager (AS-RTM)

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

1. The application knowledge
2. The application requirements
3. The feedback information from the monitors

The application knowledge contains the expected behavior of the application. The framework is agnostic about the software knobs of the application and the metrics of interest. In the current implementation, the application knowledge is learned at design time through a Design Space Exploration and it is stored in an Operating Points list. Each Operating Point relates a configuration of the software knobs with the expected performance.

The application requirements define the concept of "most suitable" from the point of view of the application designer. 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 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.

The framework enable 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.

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

In the current implementation, it is possible to use features of the application input, to change the application requirements. In this way, it is possible to achieve a simple proactive behavior of the framework (e.g. according to the size of the input).

2 Framework repository

We have released the initial implementation of the mARGOt framework in a public Gitlab repository (https://gitlab.com/margot_project/core).

The repository is organized as follows:

- framework/monitor -> contains the source code of the application monitors
- framework/asrtm -> contains the source code of the AS-RTM
- margot_heel -> contains an high-level interface generator

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 definition of Operating Point (which are the metrics of interest and the software knobs), 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).
On Fedora, these dependencies can be installed with dnf/yum:

```
>$ dnf install cxxtest
>$ dnf install papi-devel
>$ dnf install libpfm-devel
>$ dnf install lm_sensors-devel
>$ dnf install doxygen
```

Install:

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)
PEDANTIC_CHECK	[ON], OFF	Enables additional checks at run-time (additional overhead)
SAFE_CASTING	[ON], OFF	Take advantage of RTTI while casting (additional overhead)
WITH_DOC	ON , [OFF]	Generate the Doxygen documentation

If we have **cxxtest**, the framework generates a unit test executable. Assuming that we have installed the framework in *<path>*, we can test mARGOt by issuing:

```
>$ cd <path>/bin
>$ ./margot_test
```

The expected output is:

```
>running cxxtest tests (119 tests)..................................... OK
```

NOTE: The number of tests depends on the installed dependencies, and the compiler flags. However, the final "*OK*" in the output states that we have successfully compiled the framework.

4 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***.

First step: set up the experiment:

At first, we get the toy application:

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

Then we get and compile the mARGOt framework:

```
>$ git clone https://gitlab.com/margot_project/core.git
>$ cd core
>$ mkdir build
>$ cd build
>$ cmake ..
>$ 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 the number of trials influences the accuracy of the computation, expressed by a metric called "error". We also suppose that the application designer is interested in the execution time of the block (named "exec_time").
We suppose also that the application requirements are the following:

minimize error
  s.t. exec_time < 600000 (us)

We also suppose that the application designer is interested on observing the energy consumed in the elaboration, even if it is not used to influence the selection of the configuration.

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

```
1.  <margot>
2.        <block name="foo">
3.
4.          <!-- MONITOR SECTION -->
5.          <monitor name="my_energy_monitor" type="energy">
6.              <expose var_name="avg_energy" what="average" />
7.          </monitor>
8.          <monitor name="my_elapsed_time_monitor" type="Time">
9.              <creation>
10.                 <param name="time granularity">
11.                     <fixed value="margot::TimeMeasure::Microseconds"/>
12.                 </param>
13.             </creation>
14.             <expose var_name="avg_computation_time" what="average" />
15.         </monitor>
16.
17.           <!-- GOAL SECTION -->
18.         <goal name="my_execution_time_goal" monitor="my_elapsed_time_monitor" dFun="Averag
    e" cFun="LT" value="600000" />
19.
20.         <!-- SW-KNOB SECTION -->
21.         <knob name="num_trials" var_name="trials" var_type="int"/>
22.
23.         <!-- OPTIMIZATION SECTION -->
24.         <state name="my_optimization" starting="yes" >
25.             <minimize combination="linear">
26.                 <metric name="error" coef="1.0"/>
27.             </minimize>
28.             <subject to="my_execution_time_goal" metric_name="exec_time" priority="20" />
29.         </state>
30.       </block>
31. </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 4-15). The second section states the desired goals of the **foo** block (line 18). The third section states the software-knobs of the **foo** block (line 21). Finally, the fourth section states the definition of "most suitable" for the application designer of the **foo** block (lines 24-29).

The second configuration file describes the application knowledge learned at design-time for the **foo** block (named *oplist.conf*). For this tutorial, we suppose that we have performed a Design Space Exploration to profile two configurations (with the number of trials equals to 50 and 100). This is the content of the configuration file:

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <points xmlns="http://www.multicube.eu/" version="1.3" block="foo">
3.      <point>
4.          <parameters>
5.              <parameter name="num_trials" value="50"/>
6.          </parameters>
7.          <system_metrics>
8.              <system_metric name="error" value="0.55" var="1.0"/>
9.              <system_metric name="exec_time" value="50000" var="1.0"/>
10.         </system_metrics>
11.     </point>
12.     <point>
13.         <parameters>
14.             <parameter name="num_trials" value="100"/>
15.         </parameters>
16.         <system_metrics>
17.             <system_metric name="error" value="0.3" var="1.0"/>
18.             <system_metric name="exec_time" value="100000" var="1.0"/>
19.         </system_metrics>
20.     </point>
21. </points>
```

This file is a list of Operating Points. Each Operating Point maps a configuration to the profiled performance. For instance, if we consider the point where "num_trials" is equal to 50 (lines 3-11), we notice how this configuration leads to the performance stated in lines 7-10.

Using these XML configuration files, 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
>$ cp config/oplist.conf margot_heel_if/config/oplist.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();
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).
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: execute the application

We are finally able to execute the application and see the output of the log function. At each step of the loop, mARGOt prints four lines:

1. The observed metrics from the monitors
2. The goal values (the one on the execution time)
3. The selected configuration
4. The expected values of the metrics (defined in the Operating Points)

```
>$ ./tutorial
Monitored values: [ avg_energy = 2.14294][ avg_computation_time = 100125]
Goal values: [ my_execution_time_goal = 600000]
Knob values: [ NUM_TRIALS = 100]
Known metrics: [ EXEC_TIME = 100000][ ERROR = 0.3]
Monitored values: [ avg_energy = 2.11932][ avg_computation_time = 100123]
Goal values: [ my_execution_time_goal = 600000]
Knob values: [ NUM_TRIALS = 100]
Known metrics: [ EXEC_TIME = 100000][ ERROR = 0.3]
…
```

The purpose of the 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/benchmark.git) there are some public applications, already integrated with the framework.

5 Ongoing work

This release of the autotuner framework is the initial prototype that provides the core functionality. We are currently working to improve its implementation mainly with two key features:

- **On-line learning** of the application knowledge. In the current implementation, the application knowledge is obtained at design-time, through a Design Space Exploration. We are evaluating techniques to move this phase at runtime.
- **Proactive framework**. In the current implementation, the data features alter the application requirements (i.e. they are expressed as constraints). In the final release, we plan to embeds the concept of data features in the framework. This will reduce the framework overhead and relieve their management from the application designer.