European Community Seventh Framework Programme
Theme FP7-ICT-2011-7
Computing Systems

Automatic Online Tuning (AutoTune)

D3.3
Fully Extended Analysis
Eduardo Cesar

Date of preparation (latest version): 2015-07-31
Copyright © 2015 The AutoTune Consortium

The opinions of the authors expressed in this document do not necessarily reflect the official opinion of the European Commission.
PROJECT INFORMATION

Project acronym: AutoTune
Project full title: Automatic Online Tuning
Grant agreement no: 288038
Call (part) identifier: FP7-ICT-2011-7
Funding scheme: Collaborative project

DOCUMENT INFORMATION

Deliverable Number: D3.3
Deliverable Name: Fully Extended Analysis
Authors: Robert Mijacovic (TUM), Carmen Navarrete (BADW-LRZ), Carla Guillien (BADW-LRZ), Siegfried Benkner (UNIVIE), Martin Sandrieser (UNIVIE), Enes Bajrovic (UNIVIE), Laurent Morin (CAPS), Gertvjola Saveta (UAB), Anna Sikora (UAB)
Responsible Author: Eduardo Cesar (UAB)
  e-mail: Eduardo.Cesar@uab.cat
  phone: +34 935812614
Reviewed by: Michael Gerndt (TUM)
Approved by: Renato Miceli (NUI GALWAY)
Keywords: Periscope Tuning Framework, Online Performance Analysis
WP/Task: WP3 / Task 3.2, 3.3, 3.4, 3.5

DISSEMINATION LEVEL

<table>
<thead>
<tr>
<th>PU</th>
<th>Public</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP</td>
<td>Restricted to other programme participants (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td>RE</td>
<td>Restricted to a group specified by the consortium (including the Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO</td>
<td>Confidential, only for members of the consortium (including the Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>

COPYRIGHT NOTICES

© 2015 AutoTune Consortium Partners. All rights reserved. This document is a project document of the AutoTune project. All contents are reserved by default and may not be disclosed to third parties without the written consent of the AutoTune partners, except as mandated by the European Commission contract 288038 for reviewing and dissemination purposes. All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Status</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2014-05-18</td>
<td>Initial version</td>
<td>First versions of the contents</td>
</tr>
<tr>
<td>0.2</td>
<td>2014-05-27</td>
<td>Draft</td>
<td>Consolidated versions of analysis for pattern, DFVS, MW, MPI and CFS plugins; analysis for HMPP missing</td>
</tr>
<tr>
<td>0.3</td>
<td>2014-05-30</td>
<td>Draft</td>
<td>Complete versions of analysis for pattern, DFVS, MW, MPI and CFS plugins; ready for the first review</td>
</tr>
<tr>
<td>0.4</td>
<td>2014-06-01</td>
<td>Draft</td>
<td>1st Review</td>
</tr>
<tr>
<td>0.5</td>
<td>2014-06-04</td>
<td>Draft</td>
<td>First version with HMPP</td>
</tr>
<tr>
<td>0.6</td>
<td>2014-06-07</td>
<td>Draft</td>
<td>Repeated review of the document including HMPP</td>
</tr>
<tr>
<td>0.7</td>
<td>2014-06-12</td>
<td>Final version</td>
<td>Last minor corrections and final formatting.</td>
</tr>
<tr>
<td>0.8</td>
<td>2014-06-14</td>
<td>Final version</td>
<td>Final formatting.</td>
</tr>
<tr>
<td>1.0</td>
<td>2015-07-31</td>
<td>Updated version</td>
<td>Pseudo codes were added on request of the reviewers.</td>
</tr>
</tbody>
</table>
Executive summary

The AutoTune project has developed the Periscope Tuning Framework (PTF) that combines performance analysis and tuning to facilitate performance engineering of parallel applications. The approach is based on tuning plugins that encapsulate the knowledge about the tuning of applications with respect to a specific aspect. The selection of the plugins and the tuning process of plugins can significantly profit from the performance analysis capabilities of PTF. This document introduces the work done in WP 3 Performance Analysis during the third year of the project, and summarizes all the achievements accomplished during the whole project.

In the first year, the monitoring system of PTF was extended for measuring GPU performance data, energy consumption, and performance metrics related to high-level heterogeneous patterns. Then, in the second year, the work focused on implementing performance properties and search strategies for OpenCL/HMPP codes as well as for energy consumption tuning. Finally, during the third year the main objective has been to develop performance properties and search strategies for optimizing MPI codes. In addition, new strategies and properties have been implemented for improving energy consumption tuning and Compiler Flag Selection plugin.

The new analysis strategy for supporting energy consumption tuning is able to collect all the necessary information for all code regions of the application in a single step. In this way, this plugin avoids costly reexecutions of the applications for testing regions of finer granularity. With a similar objective, the compiler flag selection plugin incorporates a new analysis strategy for identifying files containing the most time consuming routines.

Periscope’s MPI analysis was extended in Y3 with new properties and analysis capabilities in the MPI monitor. On the one hand, properties for analyzing MPI master-worker applications, which provide the inputs necessary for the models used for predicting the values of the tuning parameters associated to this kind of applications and, on the other hand, a property for analyzing the convenience of tuning the eager limit parameter of general MPI applications. These properties, obtained through the analysis strategies included in PTF, are used for shrinking the search space of the M-W plugin and the MPI Parameters plugin respectively.
Contents

1 Introduction 7
   1.1 Extensions in Y3 7
   1.2 Document structure 8

2 PTF analysis concepts 8

3 PTF analysis extensions 10
   3.1 Pre-analysis 10
   3.2 Scenario analysis 11
   3.3 Results pools 12
   3.4 Pre-analysis with a certain variant 12
   3.5 Configurable analysis 13

4 Analysis in the AutoTune plugins 14
   4.1 Parallel Pattern Plugin 14
      4.1.1 Pipeline Coordination Layer 14
      4.1.2 Analysis 15
   4.2 HMPP Plugin 16
      4.2.1 HMPP analysis usage 17
      4.2.2 HMPP Profiling Module 18
      4.2.3 OpenCL/HMPP analysis strategy 24
   4.3 DVFS Plugin 25
      4.3.1 The EnergyGranularity and the EnergyGranularityBF strategy 26
      4.3.2 The scenario analysis 29
      4.3.3 The enopt library 29
   4.4 Master-Worker Plugin 30
      4.4.1 Analysis background 31
      4.4.2 Analysis for obtaining the model parameters 32
   4.5 MPI Parameters Plugin 33
      4.5.1 Analysis background 34
      4.5.2 Eager limit strategy 35
   4.6 Compiler Flag Selection Plugin 37
4.6.1 Significant region analysis ................................................. 37
4.6.2 Scenario analysis ................................................................. 39

5 Conclusions ...
1 Introduction

The AutoTune project develops the Periscope Tuning Framework (PTF) including several plugins targeting performance improvements as well as to reduce energy consumption of applications.

One of the main advantages of PTF over other tuning frameworks is its capability to combine tuning and analysis strategies to simplify and speed up the tuning process. To support the plugins with required information a number of performance analysis strategies are developed.

This document summarizes the analysis infrastructure of the release version of PTF, including the extensions that allow to automatically perform analysis at the beginning of each tuning step and within an experiment executing a certain tuning scenario. This can result in properties that guide the performance tuning carried out by the plugins and give more detailed information about the application behavior for a given scenario.

In addition, it includes the summary of the analysis infrastructure integrated in the release versions of all the plugins developed in the project: Parallel Pattern Plugin, HMPP Plugin, DVFS Plugin, Master-Worker Plugin, MPI Parameters Plugin, and Compiler Flag Selection Plugin. For each plugin we cover the provided properties, the external components used in the implementation, extensions required to the standard intermediate program representation (SIR), its generation, and their integration with the plugins.

1.1 Extensions in Y3

The following extensions to the analysis capabilities of the PTF infrastructure and the tuning plugins were implemented in Y3:

- PTF: Implementation of the configurable analysis strategy and the results pool.
- Parallel Pattern Plugin and HMPP Plugin: minor bug fixes and improvements.
- HMPP Plugin: extension to support the OpenCL measurements in the quality expression specified in D3.2.
- DVFS Plugin: new analysis strategy that significantly reduces the number of necessary experiments for determining the code regions suitable to be tuned for energy consumption. In addition, a scenario analysis was integrated to perform measurements for tunable regions.
- Compiler Flag Selection Plugin: new analysis strategy that identifies files containing routines that consume a significant amount of the application execution time and a scenario analysis to identify the tuning effect for significant regions.
- Master-Worker Plugin: extensions of the MPI analysis for obtaining the inputs for the performance models used to generate the plugins’ search space.
• MPI Parameters Plugin: extensions of the MPI analysis for identifying the range of values to be tested for the eager limit parameter.

1.2 Document structure

The organization of this document includes in Section 2 the summary of PTF analysis concepts. Then, Section 3 describes the analysis extensions implemented the AutoTune project. Next, Section 4 summarizes the analysis strategies for each plugin. In the case of the M-W and MPI Parameters plugins, this section describes the extended MPI analysis support that has been completely implemented in the third year. Finally, Section 5 concludes this deliverable.

2 PTF analysis concepts

The Periscope Tuning Framework consists of Periscope and the tuning plugins developed in the AutoTune project. The most important novelty of PTF is the close integration of performance analysis and tuning. It enables the plugins to shrink the search space, to increase the efficiency of the tuning plugins and to gather detailed information during the evaluation of tuning scenarios. The performance analysis determines information about the execution of an application in the form of performance properties.

A performance property (e.g. load imbalance, communication, cache misses, redundant computations, etc.) characterizes a specific performance behavior of a program and can be checked by a condition. Conditions are associated with a confidence value (between 0 and 1) indicating the degree of confidence about the existence of a performance property. In addition, for every performance property a severity value is provided that specifies the importance of the property. The higher the severity, the more important or severe a performance property is.

A performance property is called a performance problem, if and only if its severity is greater than a tool-defined threshold. The most important performance problem is called the performance bottleneck.

The search for performance properties in PTF is not only based on dynamic information but also on information about the program’s static structure, e.g., the nesting of regions or data structure usage information. In our environment such information is generated by the instrumenter in form of the Standard Intermediate Representation (SIR). The SIR was defined by the APART group and is an XML format. The SIR file of an application contains the structure of the instrumented application, the instrumented code regions and data structures. To support automatic tuning, the SIR was extended to contain also the information about new region types and the tuning parameters.

The overall architecture of PTF is shown in Figure 1. It consists of the user interface, frontend, analysis agent network, and the MRI monitor that is linked to the application. The analysis capabilities of the original Periscope are implemented by all these layers.
Figure 1: Architecture of the Periscope Tuning Framework.
The user interface allows to inspect the found performance properties in the Eclipse IDE, linking the properties with the application’s source code. It provides powerful sorting and filtering techniques as well as clustering techniques to identify common behavior of individual MPI processes.

The frontend triggers the performance analysis strategies, e.g. investigating certain performance properties related to specific programming models such as MPI. It is also responsible for starting the application, if necessary multiple times, if the analysis strategy cannot finish its search in a single program run. The frontend also converts the properties into XML format and writes them to a property file that can then be inspected with the Eclipse user interface.

The real analysis is performed by the analysis agents, the leaf nodes of the agent hierarchy. Each analysis agent is responsible for a subset of the MPI processes and goes through one or more analysis cycles. Each cycle starts with creating a set of hypotheses. The next step is the configuration of the MRI monitor to measure the required performance data during the next experiment. The experiment is then performed and the measurements are retrieved from the monitor. The candidate properties are then evaluated and another analysis cycle might be started by refining the found performance properties. When the analysis is finished, the properties are propagated through the agent hierarchy to the frontend.

Thus, in the center of the performance analysis support in Periscope are the performance analysis strategies and the performance properties.

In Y3 the configurable analysis strategy was implemented as well as the support for the scenario and analysis results pools.

3 PTF analysis extensions

In the course of the AutoTune project, Periscope’s analysis capabilities were extended to support automatic tuning plugins via specialized analysis strategies and properties. This section describes the general extensions added to Periscope. The specific strategies, properties, and support systems developed for the individual plugins are presented in the plugin-specific sections.

The general analysis extensions of PTF are the support of executing a performance analysis at the start of each tuning step called pre-analysis and as part of the execution of a tuning scenario called scenario analysis. These analyses are supported through extensions to the Tuning Plugin Interface (TPI).

3.1 Pre-analysis

The tuning step pre-analysis allows plugins to run a plugin-specific or a general Periscope analysis first, before the tuning is performed based on the analysis results. For example,
class IPlugin {
public:
    // Plugins are required to implement the full API
    virtual void initialize(string sirFilePath) = 0;
    virtual void startTuningStep(void) = 0;
    // Plugin function for pre-analysis
    virtual bool analysisRequired(StrategyRequest** strategy) = 0;
    virtual void createScenarios(void) = 0;
    virtual void prepareScenarios(void) = 0;
    // Plugin function defining an experiment
    virtual void defineExperiment(int numprocs, bool *analysisRequired,
        StrategyRequest** strategy) = 0;
    virtual bool restartRequired(string *env, int *np, string *cmd,
        bool *instrumented) = 0;
    virtual bool searchFinished(void) = 0;
    virtual void finishTuningStep(void) = 0;
    virtual bool tuningFinished(void) = 0;
    virtual void getAdvice(void) = 0;
    virtual void finalize(void) = 0;
    virtual void terminate(void) = 0;
};

Figure 2: Extension to the Tuning Plugin Interface for pre-analysis and scenario analysis.

the MPI analysis of Periscope provides a new property that indicates the distribution of messages for different message lengths. With this information the MPI Parameter Tuning plugin is able to shrink the search space for eager limit tuning.

The source code in Figure 2 shows the extended Tuning Plugin Interface. To enable this pre-analysis in each tuning step, the plugin provides a function that is called by the PTF frontend after starting a tuning step. The function `analysisRequired` of the TPI returns true, if a pre-analysis is to be executed. The `StrategyRequest` defines which analysis strategy is requested. Before the PTF frontend triggers the creation of scenarios, it will execute the requested analysis. The resulting performance properties are returned to the plugin for inspection in a special `property pool` (Section 3.3).

3.2 Scenario analysis

The second TPI extension allows to execute a Periscope analysis as part of the scenario execution experiment. The motivation for this feature is to combine the effect of the execution variant on the tuning objective value for the tuned region with an in-depth analysis of the application behavior. For example, the Compiler Flag Selection plugin uses this extension to gather the execution time of significant routines during an experiment
with a certain compiler flag combination. While the scenario is executed, the analysis is run and provides the ExecTime property for those routines. Based on this information, file-specific recommendations can be returned to the user.

The analysis for a tuning experiment is requested via an extension to the defineExperiment function of the TPI. It now provides two new output parameters. The strategy request is returned via strategy and the flag analysisRequired indicates whether an analysis is to be executed during the experiment.

### 3.3 Results pools

Performance properties are returned from the analysis strategies as well as for the objectives of the tuning plugin. The analysis agents are responsible for marking the properties with a flag indicating whether these are analysis properties or objective properties. The frontend automatically sorts the properties into specialized data structures. The objective properties are stored in the Scenario Results Pool (SRP) while the analysis properties go into the Analysis Results Pool (ARP).

The SRP keeps two maps that identify the objective properties with respect to the search step and the scenario for which they were returned. A search step includes all scenarios that were generated at the beginning of the step and evaluated by the plugin. Each step creates a new set of scenarios.

Thus, the plugin search step and the plugin can easily access the tuning results (objectives) relevant to take new decisions or to output the advice. The ARP maintains two lists, one per tuning step of the plugin and the other for each experiment. Thus, the plugin has fast access to the results of the last experiment, either a pre-analysis or the scenario analysis in a scenario execution.

### 3.4 Pre-analysis with a certain variant

Several of the plugins prefer to execute a pre-analysis with enforcing a certain variant of the tuning space. This can easily be implemented based on the support described above (Figure 3). The analysis request for the pre-analysis consists of a request for the
tune strategy which is configured with a scenario and a sub-analysis strategy. The tune strategy is implicitly executed during the plugin’s evaluation of the available scenarios and is here explicitly requested for running the pre-analysis.

3.5 Configurable analysis

In addition to the specialized analysis strategies that were developed in the context of some plugins, e.g., the energy plugin, the HMPP plugin, and the CFS plugin, a new general analysis strategy was implemented, the configurable analysis. This analysis strategy can be configured with plugin defined property requests. Each property request specifies:

- List of properties to be evaluated.
- List of regions for which the properties are requested.
- List of processes where the properties will be evaluated.

The configurable analysis can be used as a pre-analysis or a scenario analysis. Several plugins use this strategy to determine detailed information about certain regions during a scenario execution, e.g., the DVFS plugin evaluates the effect of a specific governor and frequency setting on several program regions and the CFS plugin explores the effect of compiler flags for significant routines (Sections 3.2 and 4.6.2). The tuning advice can thus include optimized settings for the individual regions without running experiments for each region and each setting.

The algorithm of the configurable analysis performed in the analysis agents is outlined in Algorithm 1.

Algorithm 1 Configurable Analysis Strategy

Input: list of property requests \( R \), where each request \((\text{proplist}, \text{reglist}, \text{proclist})\) consists of a list of properties, a list or regions, and a process list.

Output: Found properties

for all \( \text{req}=(\text{proplist}, \text{reglist}, \text{proclist}) \) in \( R \) do
  for all \( p \) in \( \text{proplist} \), \( \text{reg} \) in \( \text{reglist} \), \( \text{proc} \) in \( \text{proclist} \) do
    Instantiate a new candidate property \( p \) for region \( \text{reg} \) and process \( \text{proc} \).
  end for
end for

Request for all candidate properties measurements from MRI monitor
Release application for phase region
Receive measurements at end of phase region
Evaluate all candidate properties
Return found properties to plugin

First, the requests are taken from the list of property requests. The strategy then creates the requested candidate properties. Based on the candidate properties, the monitor is
configured. At the end of the experiment, the measured performance data are retrieved from the monitor and the candidate properties are evaluated. The found properties are then returned to the tuning plugin.

4 Analysis in the AutoTune plugins

4.1 Parallel Pattern Plugin

The Parallel Pattern plugin builds on a high-level programming framework developed in the context of the European PEPPHER project [2]. PEPPHER developed a methodology for improving programmability and performance portability for single-node heterogeneous CPU/GPU systems using a component-based programming approach in combination with a task-parallel execution model. The central idea of the PEPPHER approach is to provide performance-critical parts (typically functions) of applications as components with multiple implementation variants, called multi-architectural components. Each such variant is tailored for a different type of target architecture (CPU, GPU, accelerator) that may be utilized within a heterogeneous many-core system. Programmers may construct applications at a high level of abstraction by invoking component functionality from C/C++ codes via their interfaces and by using source code annotations to delineate asynchronous component calls and to specify pipeline patterns. A source-to-source compiler generates the corresponding target code that uses the pipeline coordination layer and runtime system described below. With this approach, a sequential program spawns component calls, which are then scheduled for task-parallel execution by the runtime system.

The analysis capabilities for the Parallel Pattern plugin were not extended in Y3. Only minor bug fixes and improvements were done.

4.1.1 Pipeline Coordination Layer

The Parallel Pattern plugin relies on the coordination layer to perform pipeline related performance measurements and exposes a number of tuning parameters that may be changed at runtime according to the tuning scenarios provided by the PTF. This interaction has also been addressed in deliverables D4.2 and D4.3.

The pipeline coordination layer manages all aspects of execution on a heterogeneous many-core architecture, including the automatic management of buffers for data passed between pipeline stages, the replication of individual stages, and the coordination of task-parallel execution of pipeline stages. Internally, it utilizes the StarPU [1] heterogeneous runtime system.

The pipeline coordination layer enables dynamic reconfiguration by exposing a set of tuning parameters, thus allowing users or external tools to tune the execution of the pipeline in order to achieve a desired goal (e.g., to maximize pipeline throughput).
4.1.2 Analysis

Depending on the pipeline structure (number of stages), the search space for improving the overall pipeline throughput can become very large. Hence, the plugin pre-analysis step focuses on finding those stages in the pipeline where tuning can have the highest impact. Consequently, we focus on stages with the highest execution times, also referred to as limiter stages. To detect limiter stages fine-grained performance information about the pipeline is important.

The detection of the limiter stages in a Pipeline region is outlined in Algorithm 2.

Algorithm 2 Pipeline Region Pre-Analysis

**Output:** Updated tuning parameters for the limiter stage

Request the configurable analysis strategy for the Pipeline region. It returns to the plugin the list of properties that include PipeStageExecutionTime property for each pipeline stage.

\[
\text{Max} \leftarrow \text{Stage Execution Time for stage S1} \\
\text{LimiterStage} \leftarrow S1 \\
\text{for all StageExecutionTime in returned list do} \\
\quad \text{if StageExecutionTime}_i > \text{Max then} \\
\quad \quad \text{Max} \leftarrow \text{StageExecutionTime}_i \\
\quad \quad \text{LimiterStage} \leftarrow S_i \\
\quad \text{end if} \\
\text{end for} \\
\text{Adjust stage replication factor tuning parameter for the LimiterStage} \\
\text{Adjust stage buffer size tuning parameter for the LimiterStage}
\]

To provide performance information about the pipeline, the coordination layer supports measuring pipeline characteristics such as individual stage execution times, data-transfer times, and buffer wait times. The coordination layer has further been extended with an simple interface so that PTF can request such measurements. Figure 4 depicts a high level overview of this integration. The PipelineManager provides pipeline pattern specific measurements to the MRI monitor, which are then propagated via agents to the tuning plugin.

For the ability to request pipeline related measurements in PTF, several related metrics and performance properties had to be integrated into Periscope. In the final version of the plugin, the following pipeline-related properties are supported:

- **PipelineExecutionTime**: Overall execution time for the whole pipeline region.
- **PipelineStageExecutionTime**: Individual pipeline-stage execution time.
- **PipelineStageBufWaitTime**: Individual wait time for stage buffer.
4.2 HMPP Plugin

The performance analysis strategy for OpenCL/HMPP has been implemented in coordination with the HMPP plugin. It exploits all the efforts made for the integration of the CAPS compiler with PTF, and all the work done to make the system flexible and easy to use (see next section). It searches for performance properties in OpenCL/HMPP codes.

The design of the performance analysis for OpenCL/HMPP has been chosen to take into account the evolution of the HPC market concerning the usage of directive based programming models and the emerging hybrid many-core architectures like the XeonPHI. The OpenACC standard is quickly spreading in the domain of hybrid many-core programming. The nature of problems addressed in the programming of very different hybrid many-core accelerators, such as GPUs or the XeonPHI, forces tool developers to apply a flexible approach in for performance analysis. In AutoTune, we implemented a performance analysis strategy that is non-specific to a given architecture family but easy to configure by expert users for future architectures and the OpenACC programming model.

The core of an analysis strategy in PTF is the definition of properties and the automatic search for such properties during program execution. The first section describes the basic manipulation required to activate the system inside PTF. In a second section, we present a parameterized OpenCL/HMPP performance analysis based on a description of the properties to be evaluated at runtime, and the specification of the advices or diagnosis.
that can be inferred about the application. This approach was implemented in the HMPP Profiling Module, a library fully integrated in PTF and provided with the CAPS compiler. In a third section, we present a property defined using this approach and targeting the performance analysis of CUDA kernels relative to memory transfers. The performance analysis strategy was extended in Y3 to support the OpenCL measurements in the quality expression specified in D3.2.

4.2.1 HMPP analysis usage

The HMPP performance analysis strategy is included in the prototype of the HMPP plugin. The prototype is able to perform a diagnosis on the runtime performance considering the ratio compute/transfers of a kernel. This HMPP performance analysis strategy is automatically activated if no tuning regions are set in the application or if the application was not instrumented.

A proper execution of the analysis requires the installation of the PTF infrastructure with the HMPP plugin activated, the installation of the CAPS Many-Core compiler (commercial version 3.3.4), and the installation of the HMPP Profiling module. For the installation of the first two, refer to the standard installation manuals. The installation of the HMPP Profiling module consist first in unpacking somewhere (Called here "$HPHOME") the package "hmppprofiling-1.0.tar.bz2" included in the PTF distribution in the "autotune/plugins/hmpp/Convolution" directory. Then, only a few environment variables are needed to activate the full system:

- The library search path environment variable must be appended with the lib directory for a proper dynamic linking process,

  ```
  export LD_LIBRARY_PATH="${HPHOME}/lib"
  ```

- The CAPS profiling path environment variable must be set to the location of the HMPP Profiling plugin path,

  ```
  export PHMPP_PLUGIN_PATH="${HPHOME}/plugins/hmpp"
  ```

- The capstune environment variable must be set to the location of the tuning script.

  ```
  export CAPSTUNE_PATH="${HPHOME}/plugins/hmpp/?.luac"
  ```

*Warning: a proper installation of the system must be performed before the compilation of an OpenHMPP or OpenACC application. However, no modifications are needed on*
the application build system: the CAPS compiler will automatically take into account the management of the PTF feature at compile time.

For debugging purpose, this environment can be completed with another environment variable activating the generation of messages on the profiling and analysis process of the HMPP Profiling module: set to 1, a basic list of activity information is printed; set to 2, more detailed information is presented.

```
export HMPP_TUNING_VERBOSITY=2
```

Once the system is installed and the environment set-up, the application can be compiled as usual with the CAPS MC compiler. The application will run normally without side effects if it is not launched by the Periscope frontend. If the Periscope frontend is used, the HMPP performance analysis strategy will automatically be enabled and the result of the analysis will be printed at the end of the run with potentially an optimization advice.

### 4.2.2 HMPP Profiling Module

The CAPS compiler measurement system and the definition of the properties are regrouped in the library HMPP Profiling Module. This module provides support for the measurement of metrics, e.g., the execution time of codelets, and the evaluation of performance properties based on these measurements. The module includes a Quality Index Builder that creates properties from abstract expressions, called quality expressions. The library also supports the CAPS PTF Interface that was developed to integrate the CAPS compiler with PTF.

Figure 5 shows a synthetic view of the HMPP Profiling Module linking the CAPS runtime and the MRI monitor.

Three major parts compose the library. First, an Event Recorder module makes the link with the CAPS runtime. This link enables the monitoring of all CAPS runtime and application events during the application execution, such as the start of a CUDA kernel or the transfer of data between the CPU and the GPU. Then, the Property Builder takes quality expressions from PTF and creates the necessary event record entries and measurement table entries to evaluate the given quality expression. Finally, the CAPS PTF interface links Periscope’s MRI monitor with the application. This part is Periscope-specific and might be extracted to make an independent library.

**The Event Recorder Module** The Event Recorder enables the monitoring of all CAPS runtime and application events during the application execution using the HMPP Profiling API. The HMPP profiling interface enables the analysis of a wide set of events. Each generated event describes the beginning, the asynchronous wait, or the end of an operation in the application, e.g., the start and end of a CUDA kernel. An event is defined by its family and its name. A family marks a group of consistent events, for
example ”all events related to OpenACC directives” or ”all events related to low level CUDA operations”.

Once an event is chosen, we also have to choose the data we want to monitor about the event. This is defined by a metric and an aggregation operator on that metric. So, in addition to the event monitoring system, the module provides a set of metrics and operators for all recorded events; each metric covering various aspects of the behavior of the application.

In the implementation, a metric is defined by a name and an aggregation operator. For example, the metric's name can be "time" for the timing of events (based on time stamps), or "size" for the operation size related to the event (typically for transfers, allocations...).

Aggregation operators are mathematical operations applied to the data set generated during the profiling. For example, they can be one of the following: the maximum value, the minimum value, the average value, the accumulated value.

These metrics and aggregation operators will be extended in the future to cover more aspects of the application behavior.

**The Quality Index Builder and the definition of quality expressions**  The Quality Index Builder receives from the MRI monitor a string expression containing the quality expressions to evaluate. It is a regular computational expression made of operators and operands. It uses as operands the events, the metrics, and the aggregation operators available in the Event Recorder and computes them. By default none of them are activated, they are activated one by one depending on the property requested.

The simplified view of the syntax of a quality expression is the following (the full one
implements the precedence of the operators):

\[
\langle\text{event name}\rangle : \langle\text{metric operator}\rangle \ \langle\text{metric name}\rangle \ [(\langle\text{quality operator}\rangle \\
\langle\text{event name}\rangle : \langle\text{metric operator}\rangle \ \langle\text{metric name}\rangle) \ | \ \langle\text{immediate}\rangle])^* \]

Figure 6: CAPS quality expression syntax.

For example, to express the couple "maximum CUDA transfers sizes", and "CUDA kernel average time", the expression is the following:

\[
\text{CudaMemoryTransfer}:+\text{size};\text{CudaMemoryTransfer}:'\text{time}'
\]

Note that in this example, we use a "list" quality operator with the character ";;".

**Quality indexes and the HMPP Profiling Module** The event families currently available in the HMPP Profiling Module are:

- OpenHMPP Events
- OpenACC Events
- Cuda Events
- OpenCL Events

Likewise, the current prototype offers three kinds of metrics, and for each, from 3 to 4 aggregation operators. The available metrics are the following:

- Time values (accumulated time, average time, maximum time, minimum time)
- Size values (accumulated size, average size, maximum size, minimum size)
- Bandwidth values, i.e. aggregation of sizes divided by the event times (average bandwidth, maximum bandwidth, minimum bandwidth)

Currently, only the list operator ";;" has been validated in quality expressions. The wider list of implemented property operators - and in particular the relational operators that permit to create triggers - still need to be validated.

**Quality expression full grammar specification** The quality expression syntax is built using a typical grammar for the computation of arithmetic expressions with support for event measurements. The grammar is described using the BNF format. The start unit typically returns a list of quality indexes, but can return an empty list.
A list of quality indexes is linear delimited by ";". A non-terminal `<QualIndexEntryUnit>` is defined as an entry for expressions in parentheses (defined later in the grammar).

The following rules define a set of comparison operators used to detect when some measurements reach some given thresholds. In the future, the grammar should be extended to have conditional operators in order to adjust measurements with the actual behavior of the events measured.

The typical arithmetic operations are supported with the appropriate precedence.

No unary operator is yet supported. However, for the definition of thresholds in comparisons, absolute double floating point values are defined. Precedence of computations can be controlled by the typical parenthesis syntax. A measurement is defined by the non-terminal `<QualIndexMeasure>`.
Measurements are defined by an event `<EventName>` and by a computation `<MetricComputation>` performed on the fly by the profiling module. The combination is stored as a double floating point value.

The grammar supports four major event families: the two high level programming standards OpenHMPP and OpenACC, and the two low level programming models used by the CAPS many-core compiler to program accelerators, CUDA and OpenCL.

Relying on the notion of Codelet, OpenHMPP events are grouped around the management of the accelerator device, the data, and the Codelet computation.

OpenACC events concern the major syntax of the programming model around parallel sections and kernel sections. Events related to the management of the memory are also available.

In addition to the transfers and to the kernel calls, CUDA events are more low level and are able to measure the cost of the GPU management: the streams and the contexts.
The OpenCL events support the same kind of events as CUDA with the addition of the on the fly "build" cost of CL kernel source codes.

Metric computations currently operate on the execution time of the events and on the size of the object. Derived from these computations, the bandwidth is also available.

The time metric is available for all events. It is computed on the fly by the profiling module and supports four aggregation operators: ‘|’ for the sum, ‘∼’ for the average (the sum divided by the event count), ‘+’ for the maximum value, ‘-’ for the minimum value.

The size metric is not applicable for all events. It is used typically for events related to the memory or objects (allocation, transfers). The same operators as for "time" are supported.
The bandwidth metric is applicable when the size metric can be evaluated. All operators except the sum are provided (the sum makes no sense for a bandwidth).

Metrics can be easily extended if new measurements are needed. The event families do not provide all hardware events but the events are grouped in categories to simplify their usage. This segmentation can be modified or extended in the future if the categories do not provide the necessary information for advanced users.

4.2.3 OpenCL/HMPP analysis strategy

The OpenCL/HMPP analysis strategy implemented in PTF is triggered by the HMPP plugin and identifies performance properties in HMPP codes. The quality expressions needed for the evaluation of the performance criteria are automatically forwarded by the plugin to the analysis strategy. The analysis strategy creates Periscope properties from these expressions and goes through the normal monitor configuration, application execution, and retrieval of the measurements. Then, the HMPP Profiling Module is in charge of gathering the measurements and providing the measurements to the analysis strategy. The strategy verifies the candidate properties and returns the found properties to the HMPP plugin. Finally, the properties can be output to inform the application developer or be used in a future version of the plugin to guide the search.

Evaluation of the OpenCL/HMPP analysis strategy for NVidia GPUs. As described in the previous section, the objective of the performance analysis strategy for OpenCL/HMPP is to provide a parameterized framework for OpenACC and OpenHMPP applications for different families of accelerators. The current prototype contains a hard-coded diagnosis in the HMPP plugin specifically for NVidia GPUs. Further analysis should be specified inside the plugin with the quality expressions and the associated diagnosis.

The proposed hardcoded diagnosis is the ratio between computation and data transfers of OpenHMPP Codelets using the NVidia CUDA architecture. The ratio must be lower than a threshold (10 in our case). The quality indexes used are:

- "CudaSynchronize:~time": expresses the average time taken to finish the execution of the GPU kernels in the region,
• “CudaMemoryTransfer: ~time”: expresses the average time taken to operate all memory transfers on the GPU for the execution of the GPU kernels in the region.

When a GPU kernel is executed, the memory transfer overhead must be as low as possible. The Periscope property is defined by the formulas in Figure 7.

\[
\text{Condition} = \text{severity} > 0.1 \\
\text{Confidence} = 1 \\
\text{Severity} = \frac{\text{CudaMemoryTransfer: ~time}}{\text{CudaSynchronize: ~time}}
\]

Figure 7: Severity of \text{HMPP\_TransferSeverity} property.

The condition checks whether if the computation time of the kernel is too small by comparing the severity to a predefined threshold. The confidence is always 1 and the severity is the ratio between the memory transfer time and the kernel execution time. This property might indicate that the kernel should better be run on the CPU instead of on the accelerator.

4.3 DVFS Plugin

The DVFS plugin tunes the application’s energy consumption by setting the processor frequency and governor appropriately. The release version of the plugin constrains the governor to \text{userspace}; a policy which sets the frequency to the constant value requested by the user (it won’t be scaled by the kernel). This governor is used in the plugin’s tuning technique since the plugin uses an energy model to predict the best frequency.

The search space, consisting of the different available processor frequencies, grows very fast when the settings are to be investigated for individual program regions.

It is the goal of the energy analysis strategy to support the plugin in shrinking the search space by identifying regions which are suited for this tuning technique. Regions are only suited if they have a granularity that is greater than the time overhead for measuring energy and executing tuning actions.

Figure 8 shows an example of all the code regions (including nested ones). The outer region \text{A} in Figure 8 contains the performance information of the entire code (with regions \text{B} and \text{D} inclusive). Refining region \text{A} means that region \text{B} and \text{D}, the nested regions of \text{A}, are measured by the strategy. Regions \text{C} and \text{E} have a granularity that is too small to allow energy tuning and are shown in red.

The \text{EnergyGranularityBF} analysis strategy was implemented in Y3 and is described in the following section. It significantly reduces the number of necessary experiments in the pre-analysis for determining the code regions suitable to be tuned for energy consumption.
In addition, the plugin was extended with a scenario analysis for gathering measurements for such tunable regions.

4.3.1 The EnergyGranularity and the EnergyGranularityBF strategy

There are two strategies which are available for the energy plugin in the pre-analysis phase. Either the energy granularity strategy (EnergyGranularity) or the energy granularity breadth-first strategy (EnergyGranularityBF) can be used.

**Algorithm 3** EnergyGranularity analysis strategy

**Input**: list of property requests \( R \), where each request \((proplist, reglist, proclist)\) consists of a list of properties, a list or regions, and a process list.

**Output**: Suitable regions for energy tuning

\[
R \leftarrow \text{phaseRegion}(proclist)
\]

**for all** \( req \) \((proplist, reglist, proclist)\) in \( R \) **do**

**energy** \( \leftarrow \text{Calculate EnergyGranularity for region } \text{reg} \text{ and process } \text{proc} \)

**if** \( \text{energy} > \text{GRANULARITY\_THRESHOLD} \)** **then**

**suitableRegions** \( \leftarrow \text{reg} \)

\( R \leftarrow R + \text{subRegions(reg)} \)

**end if**

**end for**
The EnergyGranularity strategy (see Algorithm 3) will identify regions that are suitable for energy tuning. If the region is suitable for tuning, it will refine the search. Otherwise, these fine granular nested regions will be ignored.

Algorithm 4 EnergyGranularityBF analysis strategy

Input: list of property requests \( R \), where each request \((\text{proplist}, \text{reglist}, \text{proclist})\) consists of a list of properties, a list of regions, and a process list.

Output: \( \text{suitableRegions} \): Suitable regions for energy tuning

\[
\text{for all req} = (\text{proplist}, \text{reglist}, \text{proclist}) \text{ in } R \text{ do }
\]
\[
\text{for all } p \text{ in proplist, reg in reglist, proc in proclist do }
\]
\[
\text{energy} \leftarrow \text{Calculate EnergyGranularity for region reg and process proc}
\]
\[
\text{if } energy > \text{GRANULARITY\_THRESHOLD} \text{ then }
\]
\[
\text{suitableRegions} \leftarrow \text{suitableRegions} + \text{reg}
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]

The pre-analysis strategy that the plugin uses by default is the EnergyGranularityBF strategy (see Algorithm 4). The term breadth-first refers to going through all the regions without discarding the measurements of the regions that are known to be non-suitable. For example, if there would be another nested region within \( C \) in Figure 8, the breadth-first strategy, while determining suitability, would measure it despite \( C \) not being suitable. The outcome of the strategy is still a list of suitable regions, just like in the EnergyGranularity strategy. The advantage of the EnergyGranularityBF strategy is that the entire analysis is done in a single search step. This reduces the restarts of the application if no phase region is given. However, the user must instrument the code appropriately to control the measurement overhead of the fine granular regions.

Both strategies collect the same performance characteristics and process the same properties. In the example shown in Figure 8, the performance and energy values of region \( A \), sub-region \( B \), and \( D \) are collected.

The strategies use the SuitedForEnergyConfiguration property which analyzes whether a region is suitable for energy tuning. The SuitedForEnergyConfiguration property has a predefined minimum time which is determined by physical limitations of the measurement device. Each individual instance of a region has to have an execution time that exceeds this threshold. Only then the region is suited for energy tuning. The inner regions \( C \) and \( E \) in Figure 8 exemplify this since the execution of each instance is only \( 1/npoints \) of the surrounding region. They are, thus, left out of the analysis due to a small granularity. Their contribution to the runtime, energy, and performance data are part of \( B \) and \( D \) respectively.

In the case of the DVFS plugin the minimum time is given by the time overhead and sampling rate of the enopt library. These overheads correspond to the time taken to measure the energy consumption, plus the time overhead required to apply changes to the frequency. Overhead plus sampling rate of a device will be referred to as device
SuperMUC’s thin islands, equipped with Sandy Bridge-EP CPUs, have a maximum latency of 10 ms for setting the frequency. Therefore, it is not possible to tune regions with an execution time that is not significantly larger than 10 ms. Given that RAPL measurements and frequency changes are done in parallel, only the device latency, which predominates, has been taken.

Table 1 shows the different latencies involved. The CPUFreq has the maximum device latency, thus, the suitable regions have been defined to have an execution time per call of more than 100 ms (ten times the device latency of the CPUFreq).

<table>
<thead>
<tr>
<th>Tool</th>
<th>Technology</th>
<th>Device latency</th>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAPL</td>
<td>MSR</td>
<td>1 ms</td>
<td>Socket, DRAM</td>
</tr>
<tr>
<td>CPUFreq</td>
<td>Kernel subsystem</td>
<td>10 ms</td>
<td>CPU</td>
</tr>
</tbody>
</table>

Table 1: Latencies of CPUFreq and RAPL.

The reported severity of the SuitedForEnergyConfiguration property is defined as execution time measured for each invocation of the region. The monitoring library reports execution time as accumulated time during execution. Calculating execution time per call requires that the execution time is divided by the number of calls of the region during the accumulation period. The property condition is set to one if and only if the severity is higher than the threshold defined by the latency of the measurement device. The confidence level is always set to one.

\[
\text{Condition} = 1 \quad \text{iff} \quad t_{\text{ExecPerCall}} > 10 \cdot t_{\text{DeviceLatency}}
\]

\[
\text{Confidence} = 1
\]

\[
\text{Severity} = t_{\text{ExecPerCall}} = \frac{t_{\text{Exec}}}{N_{\text{Calls}}}
\]

Where \( t_{\text{ExecPerCall}} \) is the average time to execute a region, \( t_{\text{DeviceLatency}} \) is the minimum time to get one sample from the device energy counters and to apply changes in the frequency, \( t_{\text{Exec}} \) is the accumulated time to execute a region, and \( N_{\text{Calls}} \) is the number of times a region was executed.

The SuitedForEnergyConfiguration property provides four additional metrics which are collected and passed to the plugin. These metrics have no impact on the selection of regions for tuning, but are important for the evaluation of the energy model within the DVFS plugin. The gathered data includes the L3 cache misses, L2 cache misses, instructions, CPU cycles, average power, and runtime.
4.3.2 The scenario analysis

The ConfigAnalysis strategy is used to measure the effect of a certain frequency on the suited regions determined by the EnergyGranularityBF strategy. This strategy is configured to check the ExecTime and EnergyConsumption properties for all suited regions and is executed as part of the scenario evaluation. The properties return the time and the energy used for the region. Thus, the best frequency for the region can be determined by the plugin. The results for every run can be obtained from this strategy, and the best combination of frequencies.

4.3.3 The enopt library

The strategies rely on the enopt library for performing the measurements and for setting the frequency and governor.

Figure 9 shows the different layers of the library.

![enopt library architecture overview](image)

Figure 9: enopt library architecture overview.

The library structure can be now split into the following components:
• The language interoperability layer. Since the library has been developed in C++, codes using the library which have been written in other languages (typically C and Fortran) need additional procedures to access the objects and methods provided by the core of the library. These methods are defined in language dependent files which provide transparent access to the functionalities of the library. These files are traditionally called *wrappers* due to their purpose.

• Computing model layer. Parallel applications can be written using different parallel models (MPI, OpenMP, hybrid MPI-OpenMP) and it also supports sequential applications. The pinning of the processes and the affinity of threads (for example, set with the `KMP_AFFINITY` environment variable) must be known to the library in order to avoid conflicts while accessing the hardware counters or even to know which process has requested a frequency change on its processor.

• Counter and *CPUFreq* layer. This layer performs the hardware counter measurements and changes the clock frequency and policies of the CPUs, through the CPUFreq [4] kernel infrastructure.

The library needs to perform operations over several files belonging to the superuser (root). These actions cannot be directly executed since unprivileged processes have no permissions to access the kernel. For this reason, privileged processes with full access to the kernel are needed. The privileged processes are typically called *servers* and the communication between the server and the library is provided through an application called *daemon*. By default, user space applications, like for example Periscope, cannot access those devices, hardware, or memory belonging to the root user. A daemon launches the servers which will change the frequency and governor and access the performance hardware counters. The servers listen to incoming requests from the enopt library, perform these changes, and return the measurements back to the user application.

### 4.4 Master-Worker Plugin

Performance of master-worker applications mainly depends of two factors. First, it is important to get a balanced computational load among workers; and, second, it is important to decide the appropriate number of workers for the application. The goal of the master-worker plugin (M-W plugin) is to minimize the execution time of a master-worker application by tuning these two factors.

The analysis strategies and properties used by this plugin have been designed and implemented during Y3 as planned. For this reason, we include a section introducing the background for the performance analysis of master-worker applications and a section for describing how the plugin uses the PTF analysis capabilities.
4.4.1 Analysis background

First, a partition strategy of the set of tasks can be used for balancing the load among workers. Instead of distributing the whole set of tasks among workers and then waiting for the results, the master makes a partial distribution of the tasks dividing the original set into portions (called batches) of decreasing size. The idea is to distribute the first of these batches among workers in chunks of (roughly) the same number of tasks. When a worker ends the processing of its assigned chunk, the master sends to that worker a new chunk; the process continues until all batches are completely distributed. This way, workers that received tough tasks will not receive more work, and workers that received lighter tasks are employed to do more work. Logically, smaller batches lead to better load balancing but increase the communication overhead, while bigger batches could lead to poorer load balancing and less communications.

Second, in an ideal master-worker application the total execution time would be equal to the sequential execution time divided by the number of workers. Nevertheless, this assumes that communications are free, the application executes on a dedicated and homogeneous platform, there is a perfect load balancing, and the computation also scales ideally. In this ideal world, any available resource that can be assigned to the application could be assigned since it can be efficiently used to improve the performance of the application. In the real world, however, the speedup of the application usually decreases as new resources are assigned to it, indicating a loss in efficiency. Moreover, at some point, assigning more resources to the application produces drops in performance because the introduced costs are bigger than the advantages brought about by the new resources.

Consequently, there are two tuning parameters associated to a master-worker application, a partition factor that determines the appropriate number of tasks of each batch to be sent to the workers, and the best number of workers to be used. Both factors can be tuned automatically by modeling the behavior of the application and taking some measurements on the computation time of workers and communication cost of tasks during the application execution.

The Master-Worker plugin uses two analytical models for estimating the value of these parameters (determining the plugin search space), and the pre-analysis capacity of PTF for obtaining the necessary model parameters. The order followed for tuning these factors is relevant because changing the number of workers makes little sense if the computational load has not been previously balanced. Consequently, this plugin uses two tuning steps, the first for tuning the partition factor, and the second for tuning the number of workers.

Finally, the Master-Worker plugin assumes that a master-worker application consists of a master process and an undetermined number of worker processes following the structure shown in Algorithm 5. The master is responsible for distributing tasks among the workers and gathering the results produced by these workers. Task distribution is done sending messages to the workers, so, there is a specific MPI_Send call for sending the tasks. In addition, there is a partial distribution strategy in place, controlled by a variable (the partition factor) indicating the amount of work to be sent to each worker. Workers are
responsible for computing the tasks received from the master, and this computation is done by a specific function in the worker process.

**Algorithm 5** Structure of the master-worker application execution.

<table>
<thead>
<tr>
<th>Input: Partition factor $f$, number of tasks to be processed $#tasks$, number of workers $n$, number of iterations $#iterations$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
</tr>
<tr>
<td>Master:</td>
</tr>
<tr>
<td>for all iterations do</td>
</tr>
<tr>
<td>//according to the partition factor</td>
</tr>
<tr>
<td>Calculate batches for a given $f$</td>
</tr>
<tr>
<td>while available batches do</td>
</tr>
<tr>
<td>Distribute work (MPI_Send)</td>
</tr>
<tr>
<td>Get results</td>
</tr>
<tr>
<td>end while</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>Worker:</td>
</tr>
<tr>
<td>while not finished do</td>
</tr>
<tr>
<td>receive work</td>
</tr>
<tr>
<td>process work (calling processing function)</td>
</tr>
<tr>
<td>send results back</td>
</tr>
<tr>
<td>end while</td>
</tr>
</tbody>
</table>

All the external parameters required for the analysis are defined in a configuration file. These parameters include information about the communication network (LATENCY, LAMBDA), the imbalance threshold (THRESHOLD), and the position in the code of the relevant MPI_Send call in the master (SENDOPERATIONS) and task processing function in the worker (WORKERLOOP).

### 4.4.2 Analysis for obtaining the model parameters

During the first tuning step, a pre-analysis is executed using a fixed number of workers and a partition factor of 1 (distributing all tasks at once). This allows to determine if there is a load imbalance and, in the case there is an imbalance, to get the parameters needed by the model: the average task processing time of each worker, and the number of tasks sent to each worker.

Specifically, this plugin uses the configurable analysis strategy and requests the ExecTime property of the function executed by the workers to process the received tasks. This property is used for getting the total computing time of each worker.

In addition, the plugin also requests the MPITime property of the MPI_Send call used by the master for sending the tasks to the workers. This property is used for getting the number of bytes sent to each worker.

With these information, the plugin computes the application imbalance severity using expression $\text{4}$, the number of tasks processed by each worker using expression $\text{5}$, and the
average execution time for each worker using expression 6. Using these parameters, the plugin calculates the partition factor that leads to the most balanced execution with the smallest communication overhead. If there is no load imbalance the partition factor is set to 1, i.e., all tasks will be distributed equally at the beginning of each iteration.

\[
Severity = \frac{(\max(\text{Workers}_{\text{ExecTime}}) - \min(\text{Workers}_{\text{ExecTime}}))}{\min(\text{Workers}_{\text{ExecTime}})} \times 100 \quad (4)
\]

\[
N\text{tasks}_{wi} = \frac{\text{MsgSize}_{wi}}{\text{TaskSize}} \quad (5)
\]

\[
\text{AverageExecTime}_{wi} = \frac{\text{ExecTime}_{wi}}{N\text{tasks}_{wi}} \quad (6)
\]

Once, the appropriated partition factor has been calculated, the plugin can go to the second tuning step to compute the adequate number of workers for the application.

During the second tuning step, for evaluating the expressions used for estimating the number of workers, we need almost the same measurements than for computing the partition factor, i.e., the total execution time of the workers and the total number of bytes communicated. Consequently, in the case the application was balanced from the beginning (partition factor is 1) this input data is already available from the pre-analysis executed during the first tuning step. However, if the application was imbalanced and the partition factor has been modified accordingly, a new pre-analysis using the new partition factor should be done in order to update these input parameters. For achieving this, the plugin uses the Tune analysis strategy to set the partition factor parameter and, then, a configurable pre-analysis sub-strategy for obtaining the \text{ExecTime} and \text{MPITime} properties as shown in Figure 3.

At the end of this tuning step, the plugin has estimated the partition factor and number of workers that should optimize the execution time of the application. Now, the scenarios to be tested in order to validate the estimated values could be generated.

4.5 MPI Parameters Plugin

MPI [7] is the "de facto" standard for interprocess communications in distributed parallel programs and, thus, it represents a key factor in the optimization of the MPI-based applications. However, a library setup for a specific system might not perform equally in a different environment (e.g. different architecture or interconnection network), so, to increase portability, most MPI implementations provide multiple configuration parameters. For example, IBM MPI, Intel MPI, and OpenMPI include from more than 50 to more than 150 parameters and, in this set, from ten to several tens of them can influence performance. These parameters are usually set by experienced users who have a
deep understanding of a specific MPI application and how it might behave on the target architecture. Consequently, manual tuning can be very difficult and extremely time consuming.

The analysis strategies and properties used by this plugin have been designed and implemented during Y3 as planned. For this reason, we include a section introducing the motivation for combining different approaches for tuning MPI applications: using heuristic search strategies and developing performance models associated to specific library parameters. As an example of the last approach, we include a section for describing the new property (EagerLimitDependency) defined to support the analysis of the eager limit parameter and how the plugin uses the PTF capabilities in this case.

4.5.1 Analysis background

The fact that tuning the MPI parameters allows for significantly improving the application performance is demonstrated by the development of tuning tools such as mpitune [5] and OPTO [3].

The plugin for MPI Parameter tuning aims to automatically optimize the values of a user selected subset of MPI configuration parameters. The integration with PTF provides the plugin with on-line measurements in the form of high level properties allowing it to make tuning decisions based on the actual performance of the application. The plugin generates the scenarios to represent specific MPI configurations in the form of tuples of parameter-value pairs (i.e. specific combinations of values for the selected subset of MPI parameters). These scenarios are executed as experiments via PTF and evaluated using the resulting properties.

The main problem when trying to optimize the parameters of the libraries that handle the communication among processes in parallel applications is that there are many parameters that users can configure, and predicting the behaviour of the library for each configuration is non-trivial. This makes it very difficult to select good values for these parameters. The fact that we have so many parameters, several with many possible values, makes it difficult to exhaustively explore all the possible configurations. These are the main motivations for automating the process of testing configurations, and the reasons for providing heuristic search algorithms to explore the search space in a reasonable time.

However, specific analysis strategies can be developed for certain parameters in order to reduce the search space. In particular, we have developed a special analysis strategy for the case of the eager limit parameter in combination with the memory buffer one. These parameters have been chosen because of their potential influence on the applications’ performance.

All these parameters may be set using environment variables or mpirun options (flags). We assume that tuned applications are SPMD, hence all application processes are optimized in the same way. The tuning action is to set the value of the MPI parameter (changing the value of the environment variable or indicating an adequate value for the
mpirun flag) and execute a new experiment with the new value.

4.5.2 Eager limit strategy

The eager limit parameter, included in most MPI implementations, allows users to establish the maximum size of messages (in bytes) that will be sent using the eager protocol. This parameter is usually limited by an upper bound by MPI implementations. For example, in the case of IBM MPI the maximum is 64 KB, and it can range from a few bytes up to this limit. Because there are many possible values for this parameter, evaluating every value exhaustively can generate many scenarios in the plugin search space.

The eager limit parameter can affect the performance of point to point communications significantly (those using MPI Send and related operations). This type of communication is affected by the actual protocol used in the communication: the protocol used affects how the messages are buffered, and determines if there are handshakes required before communication. Sending a message eagerly means that the sender is sure that the receiver has enough buffer space for storing the message, so it simply sends the message, avoiding the handshake costs (round trip plus transfer delays) of other protocols, such as the rendezvous protocol. Using the eager protocol may reduce communication time (typically between 10% to 60%), depending on the message size. However, the eager protocol introduces also some disadvantages, such as the necessity of bigger memory buffers, which can negatively affect the application performance, and the potential under-utilization of these buffers in cases where the application’s traffic consists mostly of messages larger than the set limit.

Because of the performance impact of this parameter, it is worthwhile to define a specific performance property that is related to the optimal values (application dependent) for this parameter. This property will allow for a significant reduction of the plugin search space and, as a consequence, the overall tuning time. This performance property will be automatically generated, when found, by the MPI and configurable analysis strategies (provided by the framework, and therefore also usable by other plugins).

To detect this property on MPI applications, 8 new metrics were added to the framework:

- **PSC_MPI_MSG_P2P_THR**: this metric contains the total number of bytes transferred near the eager limit (currently between 1KB and 64KB).
- **PSC_MPI_MSG_P2P_TOT**: total number of bytes transferred using the MPI point to point operations.
- **PSC_MPI_MSG_P2P_<2K-64K>**: total number of messages (count) at certain size ranges. The first one contains messages up to 2KB, while the rest are the counts of messages greater than the previous slot and under the KB value in the metric’s name (for example, the 32K metric contains the message count of transfers between 16KB + 1 and 32KB).
These metrics are collected by the monitoring library, and passed to the agents network through the MRI (as all other metrics). The collection takes place in the MPI wrapper provided by periscope. Through the wrapper, the values are computed based on the size of the MPI Datatype and the count, as specified by the application. Each time a point-to-point MPI operation is reached, the wrapper updates the metrics as shown in Algorithm 6.

Algorithm 6 Eager Limit Dependency Metrics Update

| Input:  | MPI type (mpitype) and count (count) |
| Output: | Updated Eager Limit Dependency Metrics |

\[
\text{bytes} \leftarrow \text{sizeof(mpitype)} \times \text{count}
\]
\[
PSC\_MPI\_MSG\_P2P\_TOT \leftarrow PSC\_MPI\_MSG\_P2P\_TOT + 1
\]
\[
\text{if } \text{bytes} < 64K \text{ then}
\]
\[
PSC\_MPI\_MSG\_P2P\_THR \leftarrow PSC\_MPI\_MSG\_P2P\_THR + 1
\]
\[
\text{if } \text{bytes} < 2K \text{ then}
\]
\[
PSC\_MPI\_MSG\_P2P\_2K \leftarrow PSC\_MPI\_MSG\_P2P\_2K + 1
\]
\[
\text{else if } \text{bytes} < 4K \text{ then}
\]
\[
PSC\_MPI\_MSG\_P2P\_4K \leftarrow PSC\_MPI\_MSG\_P2P\_4K + 1
\]
\[
\text{else if } \text{bytes} < 8K \text{ then}
\]
\[
PSC\_MPI\_MSG\_P2P\_8K \leftarrow PSC\_MPI\_MSG\_P2P\_8K + 1
\]
\[
\text{else if } \text{bytes} < 16K \text{ then}
\]
\[
PSC\_MPI\_MSG\_P2P\_16K \leftarrow PSC\_MPI\_MSG\_P2P\_16K + 1
\]
\[
\text{else if } \text{bytes} < 32K \text{ then}
\]
\[
PSC\_MPI\_MSG\_P2P\_32K \leftarrow PSC\_MPI\_MSG\_P2P\_32K + 1
\]
\[
\text{else}
\]
\[
PSC\_MPI\_MSG\_P2P\_64K \leftarrow PSC\_MPI\_MSG\_P2P\_64K + 1
\]
\[
\text{end if}
\]
\[
\text{end if}
\]

Once the metrics are collected at the Analysis Agent, the MPI or configurable analysis can generate the new property. This property is called EagerLimitDependency, and as its name implies, when found it means that the generated point to point traffic, by the running application, is sensible to alterations to the eager limit. In that sense, the performance of the application is dependent of the eager limit setting.

The severity of the EagerLimitDependency property is computed based on the fraction of the total MPI point to point traffic that took place near valid eager limit settings. That is simply the division of the PSC_MPI_MSG_P2P_THR metric over the PSC_MPI_MSG_P2P_TOT metric, represented as a percentage. The rest of the metrics are embedded in the extra information fields of the new property, and can be used by the plugin to detect where exactly did the traffic occur; this extra information is then used toclip the search space and greatly accelerate the search in the dimension of this parameter.
4.6 Compiler Flag Selection Plugin

The goal of the Compiler Flag Selection Plugin (CFS) is to find the best combination of compiler flags with respect to the application’s execution time. The flags to be tested are defined in a configuration file. Individual combinations are measured based on a recompilation of the code and its execution on the target machine.

The CFS plugin exploits the analysis capabilities of Periscope for a pre-analysis in the first tuning step as well as for a scenario analysis in each executed experiment. The pre-analysis is used to identify files that include routines consuming a significant portion of the execution time. The scenario analysis provides information about the effect of a certain combination of compiler flags selected in an experiment on the significant routines. Based on that information, best compiler flag combinations can be recommended for individual files instead only for the entire application compilation.

In Y3 the significant region analysis identifying the files containing routines that consume a significant amount of the application execution time was implemented and a scenario analysis was integrated to identify the tuning effect for significant regions.

4.6.1 Significant region analysis

Each experiment of the CFS plugin requires a recompilation of the code with a new flag combination. Since the compilation of the whole code can be extremely time consuming, the plugin identifies first those files that include routines with a significant portion of the execution time. Only those files need to be recompiled.

The significant region analysis is outlined in Algorithm 7.

Algorithm 7 Significant Region Analysis

Output: List of significant routines \( F \) and list of files with those routines.

Request the importance analysis strategy. It returns to the plugin the list of all routines that take at least 10% of the execution time.

\[ F = \emptyset \]

\[ \text{aggregatedPercentage} = 0 \]

Order returned list of routines by percentage of execution time.

\[ \text{while } (\text{aggregatedPercentage} < 70\%) \text{ do} \]

\[ \text{select next } f \text{ in returned list} \]

\[ \text{add } f \text{ to } F \]

\[ \text{aggregatedPercentage} += \text{percentage of } f \]

\[ \text{end while} \]

Determine the list of files including routines in \( F \)

return list of files for recompilation and \( F \) for scenario analysis.

The plugin executes a new performance analysis strategy called Importance analysis strategy as a pre-analysis before starting the search for the best compiler flag combination.

The strategy determines routines that have an exclusive execution time above a cer-
tain threshold with a default value of 10% of the phase execution time. The searched performance property is called \textit{ExecTimeImportance}.

\[ \text{Severity}_s = \frac{T(s) - \sum_{s' \in \text{calls}} T(s')}{\text{phaseCycles}} \times 100 \]  

(7)

Formula (7) specifies the severity of the property for a given subroutine \( s \). It is computed by subtracting the execution time spent in called subroutines from the routines execution time and computing the percentage of the phase execution time. The condition of the property checks whether the severity is larger than 10%. The confidence is 1.

The search strategy is a single step strategy. It creates for each subroutine and each process an \textit{ExecTimeImportance} candidate property. Each of the properties determines the required information, i.e., the execution time of the subroutines and of all call sites in that routine. All the required measurements are executed in a single experiment.

Since applications typically include also subroutines with a very small granularity, the overhead of the measurements can be quite high. This overhead can be controlled by the user via \textit{selective instrumentation}. Files with fine granular subroutines can be excluded from instrumentation via the instrumentation configuration file \texttt{psc\_inst\_config}. The instrumentation of the file can be controlled by giving the region types to be instrumented. If a file contains several routines, of which only some are too fine granular, this approach does not work well. Switching of the instrumentation for all routines in the file might be to coarse granular and some significant routines might be missed. Therefore, selective instrumentation can be combined with one of Periscope's \textit{automatic reinstrumentation strategies}.

Periscope offers three automatic reinstrumentation strategies: \texttt{overhead}, \texttt{all\_overhead}, and \texttt{analysis}. The \texttt{overhead} strategy is based on an estimation of the instrumentation overhead, while the \texttt{all\_overhead} strategy measures the overhead induced by the instrumentation. The \texttt{analysis} reinstrumentation strategy is a combination of the previous strategies and a performance analysis strategy. Details can be found in the PTF User’s Guide. For the \texttt{Importance} analysis strategy used in the CFS plugin, the \texttt{all\_overhead} instrumentation strategy can be used. It determines all regions where the instrumentation overhead is more than 5%. In a first run, those regions are determined, and then the instrumentation is removed automatically via a recompilation of the application. After the instrumentation is optimized, the analysis strategy is executed and the overhead of instrumentation is limited by the percentage chosen in the automatic reinstrumentation strategy.

The performance properties found in the pre-analysis are propagated to the frontend and used in the plugin to determine the files that are to be recompiled for each scenario.

Besides the new Periscope analysis strategy, the plugin can also make use of the profiling support built into Intel’s C and Fortran compilers. This function profiling is based on sampling and thus is a low overhead analysis providing the same information. The im-
portant limitations of the Intel compiler based analysis are that it is limited to the Intel compilers and is available only for sequential codes.

The significant region analysis in Algorithm 7 takes the properties returned from the Importance Analysis and builds a list of significant routines such that 70% of the execution time is covered. For those routines the files are determined. Only those files are then recompiled for the experiments.

4.6.2 Scenario analysis

The plugin determines for each scenario the objective in form of the execution time of the tuned region, i.e., the phase region. In addition, it applies the configurable analysis strategy that was implemented in the AutoTune project to determine the effect of the selected compiler flags on individual routines. The configurable analysis strategy is configured to evaluate the \texttt{ExecTime} property for the significant routines determined in the pre-analysis. It is started with the pedantic flag to ensure that all candidate properties are returned to the plugin. The severity of the property is the percentage of the subroutine’s execution time on the phase execution time. The extra information propagated with the property gives the real execution time in form of a floating point value.

5 Conclusions

In the third year of the AutoTune project, the set of analysis strategies was further extended to provide analysis capabilities required by the tuning plugins. This work used the monitoring capabilities implemented in the first year of the project, and the experience gained implementing the first set of analysis strategies during the second year. These analysis strategies are used to provide support to the M-W plugin and MPI parameters plugin. In addition, new strategies have been also implemented for DVFS and the CFS plugins.

For the M-W plugin, the defined analysis strategies and properties allow to estimate the parameters of an analytical model that produces a very reduced search space. For the MPI parameters plugin, the defined strategies and properties demonstrate that it is possible to define tuning strategies for specific parameters. In the case of DVFS plugin, the new strategy significantly reduces the number of necessary experiments for tuning the energy consumption of the application. The analysis strategy supporting the CFS plugin identifies significant regions for parallel codes which allows the plugin to extract the files for selective recompilation.

In all cases, the analysis strategies support tuning plugins to reduce the search space by means of removing complete regions and/or limiting the size of a tuning parameter for a region, which will reduce the search space and the tuning time.
References


