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

Automatic Online Tuning (AutoTune)

D4.1
Design of the Tuning Plugins

Laurent Morin (CAPS)

Date of preparation (latest version): October 14th, 2012

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	D4.1
Deliverable Name	Design of the Tuning Plugins
Authors	Enes Bajrovic (UNIVIE), Eduardo Cesar (UAB), Michael Gerndt (TUM), Carla Guillen (LRZ), Renato Miceli (ICHEC), Laurent Morin (CAPS), Carmen Navarrete (LRZ), Antonio Pimenta (UAB), Anna Sikora (UAB)
Responsible Author	Laurent Morin (CAPS)
Keywords	Periscope Tuning Framework, Tuning Plugins
WP/Task	WP4 / Task 4.1, 4.2, 4.3, 4.4

DISSEMINATION LEVEL

PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

COPYRIGHT NOTICES

ABSTRACT

This document reports on the design of four online tuning plugins proposed for the Periscope Tuning Framework (PTF). Each plugin will extend PTF's ability to tune application performance in a specific aspect: GPGPU computing, energy efficiency, large-scale parallel computing using MPI, and compiler flag usage.
First, we present a summary of the plugin interface and workflow proposed in deliverable *D2.1 Tuning model and PTF Design* followed by the description of a simple plugin used as template for the remaining tuning plugins.

Next, we present the design of the "HMPP Codelet Tuning Plugin": a GPGPU plugin interface based on the HMPP technology provided by CAPS entreprise. HMPP provides a directive-based solution for hybrid programming of many-core devices like GPUs, which the plugin uses to optimize the performance of GPU code generation.

We also propose a second GPGPU-oriented plugin interface – the "Plugin for Tuning of High-Level Parallel Patterns for GPGPU". It is based on the high-level pipelining pattern technology provided by the University of Vienna.

Then, we present the design of an energy efficiency tuning plugin based on a CPU frequency scaling technology – the "Plugin for Energy Consumption Tuning via CPU Frequency". This plugin dynamically adjusts the power consumption of code regions, using a technology is provided by the Leibniz-Rechenzentrum (LRZ) research center.

Then we present the design of two complementary MPI performance plugins. The first one, the "MPI Runtime Plugin", focuses on the optimization of runtime aspects of parallel execution. The second one, the "Master-Worker MPI Plugin", focuses on the MPI programming using the Master-Worker Programming Pattern technology provided by the Universitat Autònoma de Barcelona (UAB)

And finally, we present the design of the "Compiler Flag Selection Plugin": a compiler option exploration plugin in charge of the exploration of the best set of options to provide to a compilation chain to get performance. This plugin will be provided by the Technische Universität München and is called "Compiler Flag Selection Plugin".

To conclude this document, we will describe the set of possible extensions and improvements that we could propose in the future. It encompasses changes to specific plugins as well as to the PTF infrastructure in a whole.

TABLE OF CONTENTS

1 Introduction

The goal of AutoTune is to combine performance analysis and tuning into a single tool and thus to simplify development of efficient parallel programs on a wide range of architectures. The focus of this project is on automatic tuning for multicore and manycore-based parallel systems ranging from parallel desktop systems to petascale and future exascale HPC architectures. Especially in the context of large-scale HPC architectures the aspect of performance tuning will and has to be extended towards co-optimization of performance and energy efficiency.

AutoTune proposes to develop a new framework called the Periscope Tuning Framework (PTF) for the design of auto-tuning tools targeting various types of applications and optimization opportunities. The framework will be based on the performance analysis tool Periscope and will offer a set of plugins to automatically tune the application performance on aspects as diverse as the selection of compilation options, the energy consumption, the execution efficiency of MPI, or the execution time of GPU kernels. After an auto-tuning run, the user will be given recommendations on how the application can be improved from a plugin's point of view.

In this deliverable, we describe the design of the tuning plugins that enable PTF to automatically tune applications. It covers the plugin of the PTF Demonstrator that is used to demonstrate the feasibility of the PTF design. The main part of the deliverable describes the design of an initial version of each of the advanced tuning plugins proposed in the project's technical annex. These initial designs will be implemented in the next year and then gradually be extended.

The following two subsections are basically copies from the technical annex giving an overview of Periscope and the Periscope Tuning Framework.

1.1.1 Periscope

Periscope is an automatic performance analysis tool for highly parallel applications written in MPI and/or OpenMP developed at Technische Universität München (TUM). It is a representative for a class of automatic performance analysis tools that automate the whole analysis procedure. Unique to Periscope is its ability to work online in a distributed fashion. This means that the analysis is done while the application is executed (online) by a set of analysis agents, each searching for performance problems in a subset of the application's processes (distributed). The properties found by Periscope point to code regions that might benefit from further tuning.

Automation in Periscope is based on formalized performance properties, e.g. inefficient cache use or load imbalance. Based on a repository of performance properties the analysis agents can search for these properties in the program execution under investigation. They automatically determine which properties to search for, which measurements are required, which properties were found and which are more specific properties to look for in the next step.

The overall search for performance problems is determined by search strategies. A search strategy defines in which order an analysis agent investigates the multidimensional search space of properties, program regions, and processes. Many of Periscope's search strategies are multistep strategies, i.e., they consist of multiple search steps. A very important concept for multistep strategies is a program phase that is determined by a certain repetitive program region. A good example for such a phase region is the body of the time loop in scientific simulations where each iteration simulates one time step. Thus, within each iteration a new step of the search strategy can be executed.

Based on the search strategies, Periscope executes an algorithm based on a hierarchical distribution of analysis agents helped by monitors to analyze the performance of the application and retrieve the information to the frontend.

On top of the frontend, Periscope provides a graphical user interface (GUI) based on Eclipse and the Parallel Tools Platform (PTP). The GUI allows the programmer to define a project with all the source files, start a performance analysis via the frontend and, most importantly, investigate the performance properties found by Periscope.

1.2 Periscope Tuning Framework

AutoTune will develop the Periscope Tuning Framework (PTF) as an extension of Periscope. It will follow Periscope's main principles, i.e., the use of formalized expert knowledge in form of properties and strategies, automatic execution, online search based on program phases, and distributed processing.

Periscope will be extended by a number of tuning plugins that fall into two categories: online and semi-online plugins. An online tuning plugin performs transformations to the application and/or the execution environment without requiring a restart of the application; a semi-online tuning plugin is based on a restart of the application but without restarting the agent hierarchy.

Figure 1 illustrates the control flow in PTF. The tuning process starts with a preprocessing of the application source files. This preprocessing performs instrumentation and static analysis. Periscope is based on source-level instrumentation for C/C++ and Fortran. The instrumenter also generates a SIR file (Standard Intermediate Representation) that includes static information such as the instrumented code regions and the nesting.

When the preprocessing is finished, the tuning can be started via the Periscope frontend either interactively or in a batch job. As done in Periscope, the application will be started by the frontend before the agent hierarchy is created.

As presented in section 1.1.1, Periscope uses **an analysis strategy**, e.g. for MPI, OpenMP and single core analysis, to guide the search for performance properties. This overall control strategy now becomes part of a higher-level **tuning strategy**. The tuning strategy controls the sequence of analysis and tuning steps. Typically, the analysis determines the application properties to guide the selection of a tuning plugin as well as the tuning actions performed by the plugin. After the plugin finishes, the tuning strategy might restart the same or another analysis strategy to continue on further tuning. We will begin by developing **single plugin tuning strategies**, i.e., for each tuning plugin a separate strategy. Similar to the analysis strategies in Periscope, **combined plugin tuning strategies** will be developed later to combine multiple tuning plugins to perform multi-aspect application tuning.

Each of the tuning plugins will be controlled by a specific **plugin strategy**. This strategy will guide the search for a tuned version. The search space will be restricted based on the properties resulting from the previous analysis as well as by other plugin specific means, such as expert knowledge or machine learning. Typically, the selection of tuning actions ends with a number of possibilities that have to be evaluated experimentally. Online tuning plugins will be able to execute the experiments without an application restart, e.g. the energy efficiency tuning plugin; while semi-online plugin will require a restart, e.g. the compiler flag selection plugin. The plugin strategy itself can also be iterative. For the analysis of the experiments, Periscope's performance analysis support will be leveraged. The execution of experiments with and without application restart is already supported by Periscope.

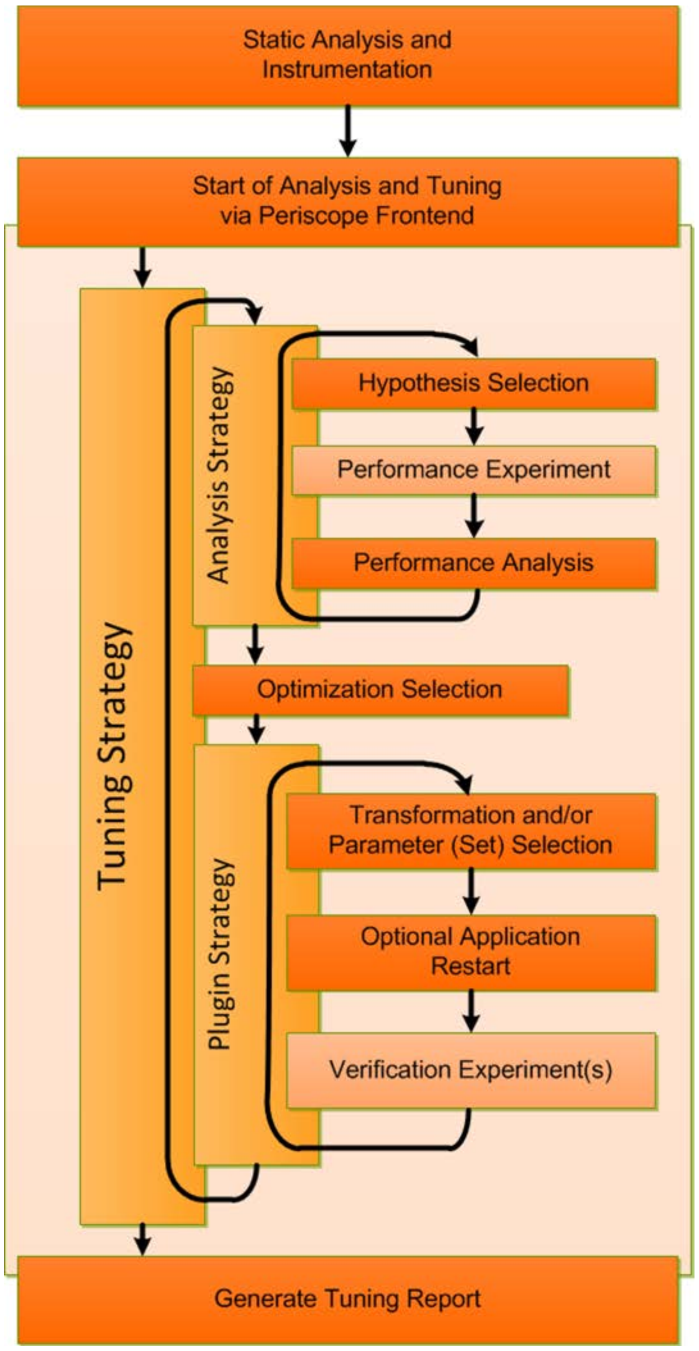
Static Analysis and Instrumentation
Start of Analysis and Tuning via Periscope Frontend
Tuning Strategy
Analysis Strategy
Hypothesis Selection
Performance Experiment
Performance Analysis
Optimization Selection
Plugin Strategy
Transformation and/or Parameter (Set) Selection
Optional Application Restart
Verification Experiment(s)
Generate Tuning Report

Figure 1: Tuning Control Flow

Once the tuning process is finished, PTF will generate a tuning report documenting the remaining properties as well as the tuning actions recommended. These tuning actions can then be integrated into the application such that subsequent production runs will be more efficient.

2 Tuning Plugin Design

Given the number of programming models, parallel patterns and hardware targets to be supported by PTF, a sufficiently generic tuning plugin design was required. In this section, we present the PTF Demonstrator which realizes a prototype of the PTF architecture given in deliverable *D2.1 Tuning Model and PTF Design*. We first repeat the terminology of the tuning model and then specify the tuning control flow and the plugin interface. At the end of the section we present the implemented Periscope extensions.

2.1 Terminology

This section is a copy of section 2 of deliverable *D2.1 Tuning Model and PTF Design*. It defines the terminology used in this deliverable as well.

The tuning plugins will try to improve the application execution by influencing certain tuning points.

Tuning points $TP = \{v_1, v_2, \ldots\}$ are the features for influencing the execution of a region. Each tuning point has a name and an enumeration type or an interval of integer values with stride. For example, a tuning point is the clock frequency of the CPU which determines the overall energy consumption. This tuning point has an enumeration type with the different possible clock frequencies. Tuning points can also be given by the application, e.g. a set of different GPGPU code variants provided by the application developer or an external tool.

All tuning points of a tuning plugin define a multidimensional tuning space.

Tuning space of a tuning plugin P is the cross product of the individual tuning points, i.e., $TS_P = TP_1 \times TP_2 \times \ldots \times TP_k$

For a program region the tuning plugin will select a set of variants that may lead to a potential improvement and that need to be evaluated by experiments.

The **variant space** VS_r of a program region *r* is a subset of the overall tuning space, i.e., $VS_r \subseteq TS_P$. A **variant** of a code region *r* is a concrete vector of values for the region's tuning points $\upsilon_r = (v_1, \ldots, v_k)$

The variant space is explored by a search strategy to optimize certain objectives.

An **objective** is a function obj: $REG_{appl} \times TS_P \rightarrow \Re$ where REG_{appl} is the set of all regions in the application. A single or multiple objectives are to be optimized by the tuning plugin for a given program region over the regions variant space.

The tuning plugin creates a sequence of tuning scenarios that are executed by Periscope to determine the values of one or more objectives.

A **tuning scenario** is a tuple $sc_r=(r,\upsilon_r,\{obj_1, \ldots, obj_n\})$ where *r* is the program region, $\upsilon_r \in VS_r$ is a variant of the region's variant space, and $obj_1 \ldots obj_n$ are the objectives to be evaluated.

During the execution of a tuning scenario, tuning actions will be executed to select the individual values of the tuning points.

A **tuning action** TA_i is executed for each tuning point TP_i with $1 \leq i \leq k$ during the execution of a tuning scenario. It enforces the value v_i for the tuning point *i* given by the variant $\upsilon_r = (v_1, \ldots, v_k)$.

2.2 Tuning Control Flow

The PTF frontend controls the overall tuning process. For auto-tuning, the frontend enforces a predefined sequence of operations that are implemented by the tuning plugins. The control flow follows the plugin strategy described in section 3 of deliverable *D2.1 Tuning Model and PTF Design*.

The predefined sequence of operations has to fulfill the requirements of all tuning plugins developed in AutoTune. Therefore, it is quite complex. In this deliverable we present a simplified version in Figure 2. All steps are involved in creating and processing the scenarios that need to be evaluated by experiments. Scenarios are stored in pools that are accessed and shared by the plugins as well as the frontend.

These pools are:

- Created Scenario Pool (CSP): Scenarios that were created by a search algorithm.
- Prepared Scenario Pool (PSP): Scenarios that are already prepared for execution.
- Experiment Scenario Pool (ESP): Scenarios that are selected for the next experiment.
- Finished Scenario Pool (FSP): Scenarios that were executed.

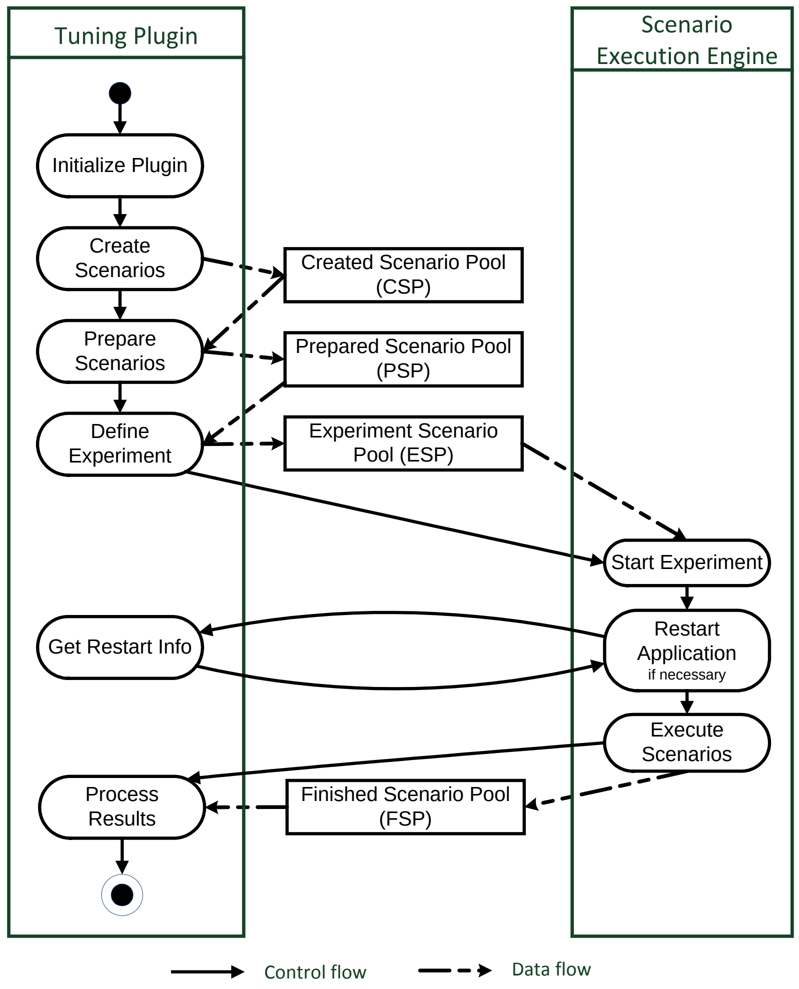
Tuning Plugin
Scenario Execution Engine
Initialize Plugin
Create Scenarios
Prepare Scenarios
Define Experiment
Created Scenario Pool (CSP)
Prepared Scenario Pool (PSP)
Experiment Scenario Pool (ESP)
Start Experiment
Get Restart Info
Restart Application if necessary
Execute Scenarios
Process Results
Finished Scenario Pool (FSP)
Control flow
Data flow

Figure 2: Simplified control and data flow of a tuning plugin with the management of the scenarios via scenario pools

Figure 2 presents the sequence of steps followed by a tuning plugin.

1. Initialization: First, the plugin is initialized and the tuning points are created.
2. Scenario Creation: From the defined tuning space, the plugin creates the scenarios and inserts them into the CSP. Here, the plugin first selects the variant space to be explored. It then creates the individual scenarios, which combine the region, a variant, and the objectives, either via a generic search algorithm, e.g., exhaustive search, or by its own search algorithm.
3. Scenario Preparation: Scenarios are selected from the CSP, prepared and moved into the PSP. The preparation of scenarios typically covers tuning actions that cannot be executed at runtime, e.g., recompilation with a certain set of compilation flags or generation of special source code for the scenario's variant. Only the plugin can decide whether certain scenarios can be prepared at the same time. For example, two scenarios requesting different compiler

flag combinations for the same file cannot be prepared at the same time. If no preparation is required, the plugin simply copies all the created scenarios to the PSP.

4. Define Experiment: A subset of the prepared scenarios is then selected for the next experiment and moved into the ESP. When the plugin selects the scenarios for the next experiment it has to take constraints into account. For example, different scenarios for the same program region cannot be executed in the same experiment unless they can be assigned, for example, to different processes of the MPI application. The assignment of scenarios to processes or threads is decided by the plugin in this step.
5. Experiment Execution: The Scenario Execution Engine (SEE) is responsible to execute the experiment. It will first check with the plugin, whether a restart of the application is necessary to implement the tuning actions. For example, the scenarios generated by the MPI tuning plugin explore certain parameters of the MPI runtime environment. These can only be set via environment variables before launching the application. After the potential restart of the application, the SEE will run the experiment by releasing the application for a phase, i.e., the execution of the phase region. If multiple phases are required to gather all the measurements for objectives, the SEE will automatically take care of that. It will even restart the application if it terminates before all the measurements were finished. At the end of this step, the executed scenarios are moved into the FSP and the objectives are returned to the plugin.
6. Process Results: The plugin accesses the objectives, which are implemented as standard Periscope properties. Each objective specifies its scenario. The objectives' value is then used to select the best scenario and return the tuning recommendation.

The individual steps might need to be repeated if scenarios still remain in the scenario pools. These loops in the tuning process are described in the next section in the context of plugin functions implementing the steps outlined in this section.

2.3 Tuning Plugin Interface

In this section, the major methods of the Tuning Plugin Interface (TPI) are described. These methods must be implemented by all plugins and their conformance is checked when the plugin is loaded.

2.3.1 Initialize

After the frontend has initialized itself and is ready to start the tuning process, it loads the plugin specified by the user. Before the plugin can be utilized, it needs to be instantiated and initialized. The frontend instantiates the plugin and then invokes this method to do so.

In this method, the plugin needs to set up its internal data structures for tuning points, the tuning space, search algorithms to be used, and the objectives.

2.3.2 Create Scenarios

After the plugin has initialized its data structures and search algorithm, the next step is to create scenarios. The plugin generates the scenarios using a search algorithm and inserts them into the CSP, so that the Frontend has access to them. The search algorithm might go through multiple rounds of scenario generation. The selection of new scenarios that are generated in the next step might depend on the objective values for the scenarios in the previous step. Before the frontend calls the final method to process the results, it checks if the search algorithm needs to generate additional scenarios. If so, the frontend triggers an additional iteration of creation, preparation, and execution of scenarios.

2.3.3 Prepare Scenarios

Some scenarios require preparation before experiments can be executed. If a set of scenarios needs preparation, it should be done in this method. However, if no preparation is necessary, the Prepare Scenario method can simply move the scenarios from the CSP to the PSP.

After the execution of an experiment, the frontend checks if the CSP is empty. If there are still scenarios, the frontend calls the Prepare Scenarios method again.

2.3.4 Define Experiment

Once generated and prepared, the scenarios need to be assembled into an experiment. An experiment will go through at least one execution of the phase region of the application. There are two ways to execute multiple scenarios in a single experiment. Either they can be assigned to a single process because they affect different regions or they can be assigned to different processes. Only the plugin can decide whether this is possible or not. Therefore the frontend calls the Define Experiment method to decide which scenarios are executed in the next experiment and to assign the executing process to the scenarios. Scenarios selected by the plugin for the next experiment are moved from the PSP to the EPS.

After the plugin defined the experiment, the frontend transfers the control to the scenario execution engine, which forwards the scenarios to the analysis agents and triggers the experiment. At the end of the experiment the objectives of the scenarios are returned to the plugin.

The frontend checks after the execution of an experiment if there are additional prepared scenarios in the PSP, and calls the Define Experiment method again to evaluate a next set of scenarios.

2.3.5 Get Restart Info

This method is called by the scenario execution engine and returns *true* if a restart of the application is necessary for the execution of the experiment. For example, a restart is necessary if the application was recompiled according to the scenario with a special combination of compiler flags.

It also permits to return parameters to the application launch command, e.g., if a scenario requires certain configuration parameters of the MPI library to be set during the launch of the application.

2.3.6 Process Results

If the CSP is empty and the search algorithm is finished, the frontend calls the Process Results method. In this method, the plugin analyzes the acquired properties and either commands that there are extra steps necessary for tuning or indicates that the tool is finished and generates the tuning advices for the user.

2.4 Periscope Tuning Framework Implementation

In this subsection, we present the main components added to Periscope to build the PTF Demonstrator. We present the extensions to the frontend, the analysis agents, and the monitor in separate subsections.

2.4.1 Frontend Extensions

Most of the extensions to the class hierarchy in PTF were added to the frontend. The following classes implement the concepts specified in section 2.1:

- TuningPoint;

- Variant;
- VariantSpace; and
- ScenarioDescription.

These classes come with helper classes to define for example tuning point ranges and constraints.

In addition interface classes for the tuning plugins and the search algorithms were added. These are:

- IPlugin; and
- ISearchAlgorithm.

These interfaces have to be implemented by the tuning plugins and the search algorithms. For the demonstrator, a demo tuning plugin and the exhaustive search algorithm were implemented.

The frontend was also extended by the scenario execution engine that drives the experiments for the evaluation of scenarios.

To implement the overall control of the tuning process, a finite state machine was implemented in the frontend based on Boost (http://www.boost.org/).

2.4.2 Analysis Agent Extensions

The behavior of the agent hierarchy in PTF remains the same as in Periscope, except for the addition of a tuning analysis strategy to the analysis agents. This strategy controls the execution of an experiment in the analysis agent. It

- receives the scenarios from the frontend,
- selects the scenarios assigned to the application processes controlled by this analysis agent,
- configures the tuning actions to be executed during runtime for the regions specified in the scenarios[1],
- requests the measurement of performance data required to evaluate the objectives of the scenarios,
- releases the application processes for the experiment,
- retrieves the performance data at the end of the experiment,
- evaluates the objectives, and
- propagates the objectives up the agent hierarchy to the frontend.

In addition to the tuning analysis strategy, the execution time objective used in the Demo plugin described in section 3 was added to the analysis agent. An objective is a property and thus extends the Property class of Periscope. The following code snippet outlines the definition of the objective. Among other methods it implements the shown four important methods.

- *Get_required_info()* specifies, which measurement are required to evaluate that property. For this objective it is only the execution time of the tuning region and of the phase region.

[1] The configuration of tuning actions that are executed at runtime is done via the MRI.

- *Evaluate()* is called after the experiment by the tuning analysis strategy. It reads the required data from the agent's performance database and copies the execution time to a local attribute.
- *condition()* indicates whether the property holds. In this case, it always returns true since the measurement was executed.
- *severity()* indicates the importance of the property. For objectives that are evaluated on request by the scenario it determines the objective value. Here it is the percentage of the execution time of the phase region spent in the tuned region.

```
class ExecTimeProp : public Property
{
     private:
          ...
     public:
          Gather_Required_Info_Type get_required_info();
          void evaluate();
          bool condition();
          double severity();
          ...
};
```

2.4.3 Monitor Extensions

The monitor of Periscope was extended with two types of runtime tuning actions (called MRI Tuning Actions):

- Variable Tuning Actions: The tuning action assigns the value of the tuning point specified in the variant of the scenario to a variable of the application.
- Function Tuning Actions: This tuning action calls a function provided by the application with the value of the tuning point.

Both tuning actions are configured by new MRI requests. These requests specify the tuning region, the name of the tuning point, and the value.

Additional extensions to the monitor for performance and energy measurements are described in deliverable *D3.1 Extended Monitoring System.*

3 Demo Plugin

We developed the Demo plugin to demonstrate and validate the PTF architecture. In this section, we describe the plugin itself as well as a simple Fortran application used for evaluation.

3.1.1 Tuning Objective

The tuning objective of this plugin is to minimize the execution time of a code region for which several variants are defined in the program. The region is marked in the Fortran source file using an AutoTune directive. The directive defines a single tuning point that is manipulated through a variable tuning action. Each value of the variable selects a single variant.

3.1.2 Tuning points and tuning actions

This tuning plugin can process a single tuning point defined in the program via directives. The directives defining the tuning point are shown below in a Fortran snippet of the sample application:

```
do k=1,20
     variant=k
     !$MON USERREGION TP name(Test) variable(variant) variants(10)
     tstart=MPI_Wtime()
     !<user compute code depending on the value of variable variant.>
     tend=MPI_Wtime()
     !$MON END USERREGION
enddo
```

Here, the tuning region is defined by the directives *user region* and *end user region*. The specification of a tuning point starts with the marking “TP” in the directive, followed by the name of the tuning point, the name of the variable and the number of variants. The range of values of this tuning point is 1 to 10.

As we can see in the code, the value of variable *variant* is initially set to *k* and the monitor overwrites its value at runtime. Setting the value of *variant* to *k* allows the code to be run without PTF. It sweeps over all the 10 variants.

When instrumenting the application, the PSC instrumenter generates the SIR file. The instrumenter parses the Fortran source file and extracts the information from the AutoTune directives. The tuning points (only one in this case) are then added to the SIR file, which the frontend uses as input. The instrumentation also adds a mapping function to the monitor that defines a pairing between the name of the tuning point and the tuning variable. Whenever the region is entered, i.e., the monitor’s start_region(…) function is executed, the monitor assigns a value to the tuning variable to trigger the execution of a certain variant. The following code snippet outlines the instrumented program version.

```
DO k = 1, 20
   variant = k
   CALL psc_map_tp_var('<TEST>',variant)
   CALL start_region(27,1,485,0,-1)
   !$MON USER REGION
       ...
   !$MON END USER REGION
   CALL end_region(27,1,485,0,-1)
END DO
```

3.1.3 Tuning Plugin Interface

In this subsection we describe the implementation of the Demo plugin in terms of its Tuning Plugin Interface (TPI) methods. The intention of this plugin is to be a simple demonstrator; therefore, most of the methods implement only the minimum functionality required by a plugin conforming to the TPI.

3.1.3.1 Initialize

As mentioned in section 2.3.1, the plugin must create the following data structures at initialization time: tuning points, search algorithms, and objectives. In this plugin, a single tuning point is created from the information in the SIR file. As explained earlier, the TP entry was put in the SIR file after parsing the AutoTune directives from the Fortran source file of the sample application. The created tuning space is thus one-dimensional and the execution time objective is selected.

The search algorithm used is a decision made by the plugin itself as desired by the plugin developer. The Demo plugin is based on exhaustive search since the one-dimensional tuning space is small.

3.1.3.2 Create Scenarios

This method of the Demo plugin generates all possible scenarios and puts them into the scenario pool. The reason for this is that we assume a small number of scenarios due to the one-dimensional tuning space.

3.1.3.3 Prepare Scenarios

The method Prepare Scenarios simply copies the scenarios from the scenario pool to the prepared scenario pool. This is because no preparation is necessary.

3.1.3.4 Define Experiment

In this case, every single scenario defines an execution of the region with a different value for the manipulated variable (named "*variant*" in the sample application) that is entirely independent from the values of the same variable in other processes. The method Define Experiment of the Demo plugin takes this into account and defines for an experiment exactly as many scenarios as there are MPI processes available. Each scenario is assigned to a unique MPI rank. If there are less MPI processes then scenarios, the plugin has to go through multiple steps of defining and executing scenarios until all the scenarios are executed.

When the agent network is finished with the experiment, the objectives are propagated to the frontend. The frontend then sets them up in a properties pool that is available to both the plugin and the search algorithm. The frontend also moves all the scenarios from the ESP to the FSP.

3.1.3.5 Get Restart Info

This method returns *false* to indicate that no restart is required.

3.1.3.6 Process Results

This method is called by the frontend when the scenario pool is empty. The Demo plugin retrieves the optimum configuration via the search algorithm's interface and prints the result as its tuning advice.

4 Advanced Tuning Plugin Design

4.1 Tuning of High level Parallel Patterns for GPGPU[2]

We investigate the tuning of high-level parallel programming patterns for heterogeneous systems equipped with GPUs. Our approach is based on the premise that users can explicit programming patterns available for exploitation in high-level input programs using C/C++ source-code directives. Based on this user-provided information, intelligent programming tools can then automatically apply both static (compile-time) and dynamic (execution-time) decisions. Hence, we follow a multi-layered methodology covering language and compiler functionality as well as runtime support.

[2] Code name: PTF_PATTERN_TP

Our programming framework builds on a set of source-code directives, a code-generator as well as a runtime layer for execution of parallel high-level patterns. This framework has been partially developed within the EU Project PEPPHER. In our current work, we support the pipeline programming pattern.

A pipeline consists of several inter-connected stages that exchange data. Data is usually passed from one stage to another via buffers for each input and output port of a stage. This "data-flow" execution has data-dependencies for single iterations from first to last stage invocation but usually multiple input data packets can be processed in parallel. Hence, for this form of execution, the pipeline pattern supports classic task-parallel execution of individual stages. In addition, data-parallelism can be exploited in two ways: (1) through replication of individual stages, and (2) through intra-stage parallel stage implementations (i.e. a highly parallel GPU kernel).

In our approach, pipelines are formulated in C/C++ programs as while-loops with stage invocations in the loop body. A stage invocation, represented as a function invocation, is a call to an abstract interface function. Such functions, in this context also called "components", may feature different implementations, each tailored for a different hardware architecture or optimization goal. Pipeline while-loops as well as stage invocations are annotated with our source-code directives and a code generator then transforms the high-level input program to a form that utilizes appropriate stage implementations in conjunction with our pipeline runtime-layer.

Even though also the code-generation phase of our framework could highly profit from automatic, feedback-directed tuning, we concentrate our work in the first project phase on automatic online-tuning of the pipeline runtime layer.

The pipeline runtime layer controls the execution of pipelined output programs and offers coordination of stage invocation as well as buffer structures with different characteristics. For example, it enables to dynamically reconfigure the number of active stage replications in a program or buffer sizes for individual stage buffer ports. In the scope of AutoTune, we interface this dynamic pipeline coordination layer with Periscope so that the PTF can automate the tuning of pipelined programs.

4.1.1 Tuning Objective

Supported target applications utilize the pipeline runtime-layer. Our approach to tuning the pipeline patterns focuses mainly on the maximization of the pipeline throughput. Hence, we utilize time measurements for individual pipeline stages as well as the full pipeline code-regions, previously indicated in the original input program by while-loop annotations.

Since the overall throughput is dominated by the slowest stage in the pipeline, individual stage tuning can provide improved global results. In particular, replicating the slowest stage and executing those stage replicas in parallel can improve the global throughput if the target architecture provides enough execution units (CPU cores or GPUs). Another significant mechanism to ensure high processing-unit occupancy is the minimization of stage wait-times due to blocking of buffer structures. Therefore, in addition to the global throughput objective, we also aim at minimizing individual buffer wait times.

Our initial tuning approach and our runtime-layer focus on single-node hybrid many-core systems, comprising GPUs in addition to conventional CPUs.

4.1.2 Tuning points and tuning actions

For the previously described tuning objectives – maximization of pipeline throughput and minimization of buffer wait-times – we identify the following tuning points and actions:

- Replication factor of individual stages.
    - Tuning points: Integer values in a predefined range used as a parameter for the stage invocation of the pipeline runtime-layer.
    - Tuning action: MRI variable tuning action assigning the value to the stage's replication factor.
- Buffers sizes.
    - Tuning points: Integer values in a predefined range used as a parameter for buffer configuration of input and output ports of individual stages.
    - Tuning action: MRI variable tuning action assigning the value to the buffer configuration parameter.
- Splitting and merging of the stages.
    - Tuning points: Integer values in a predefined range used as a parameter for function pointer array representing different stage configurations that can be passed to the runtime system.
    - Tuning action: MRI variable tuning action assigning the value to the function pointer array parameter.

4.1.3 Tuning Plugin Interface

This subsection describes the general steps undertaken by the pipeline plugin for tuning high-level pipelined program regions.

After the source-to-source tool translated the high-level input program, it outputs an application that comprises calls to our pipeline coordination layer. In addition, the Periscope SIR format output file stores relevant information about the pipeline regions and code annotations for stage and buffer configuration. The SIR file comprises information about the location of pipeline regions in the code and their related tuning points for stage and buffer tuning. In addition, we aim at representing user-provided tuning hints expressible via code-directives in the SIR file.

4.1.3.1 Initialize

During the initialization, the plugin must parse and store the generated output SIR file internally for subsequent XML processing. Next, the method extracts the comprised pipeline code-regions from the SIR representation and gathers tuning points for each stage invocation and utilized buffer structure accordingly. Subsequently, the plugins stores tuning points in an internal representation to get a fast access for future processing.

4.1.3.2 Create Scenarios

Based on the extracted tuning points, the method creates the tuning scenarios. Each scenario represents a single pipeline variant. A variant denotes one particular configuration of an execution based on concrete values for all tuning points in a pipeline region. Scenarios are generated and put into a scenario pool. We consider scenarios for each tuning region (pipeline) individually.

Although the ranges for individual tuning points can be specified, the resulting variant space for each pipeline region can be very large. This is especially relevant for pipelines that feature a high number of stages and input/output buffer ports. Therefore, the plugin might need to execute several steps to reduce the number of variants to evaluate. Apart from simple exhaustive search, we investigate techniques based on historical execution data incorporating static information about the concrete runtime environment (e.g., number of CPU cores, number of GPUs, number of GPU multiprocessors).

4.1.3.3 Prepare Scenarios

For the evaluation of scenarios no additional preparation is required.

4.1.3.4 Define Experiment

An experiment can include multiple scenarios, but for each pipeline region only a single scenario.

The scenarios are executed in the experiment for *N* times. After the scenarios have been sampled, the plugin stores the measured times and executes the next scenarios. This builds on the assumption that runtime characteristics related to the input data do not change between scenario invocations.

4.1.3.5 Get Restart Info

The tuning points do not require a restart of the application. The function returns *false*.

4.1.3.6 Process Results

After all scenarios have been evaluated, the plugin processes and sorts the returned objective values. The plugin returns the best variants for the executed configurations.

4.2 Hybrid Manycore tuning: HMPP Codelet Tuning Plugin[3]

4.2.1 Introduction to HMPP

HMPP Workbench is a feature-rich tool that simplifies the utilization of GPUs and many-cores systems. This section provides the basic information about how to get started with the OpenHMPP directive set. We introduce:

- The basic concepts of OpenHMPP (e.g. RPC, codelet, memory model, gridification);
- The basic directives (e.g. codelet / callsite);
- Code generation and optimization directives

Furthermore, the HMPP Workbench brings the support to the OpenACC [1] [2] standard directives – a parallel-programming standard announced at Supercomputing 2011 and supported by NVIDIA, Cray Inc., the Portland Group (PGI), and CAPS entreprise.

For more information, refer to the HMPP Workbench documentation [3].

4.2.1.1 Basic concepts

4.2.1.1.1 Remote procedure call (RPC)

HMPP allows to program hardware accelerators (HWA) using the Remote Procedure Call paradigm. Hardware accelerators commonly come in the form of discrete cards connected to the CPU through a fast bus interconnect like PCI-Express. Therefore, in order to offload computations to them, HMPP understands that the HWA is distant from the host and that in most cases the memory address spaces are separate.

[3] Code name: PTF_HMPP_CODELET_TP

An RPC sequence consists of 5 steps:

- Allocate the HWA and the memory;
- Transfer the input data: Host => HWA;
- Execute the sequence of the instructions;
- Transfer the output data: HWA => Host; and
- Release the HWA and the memory.

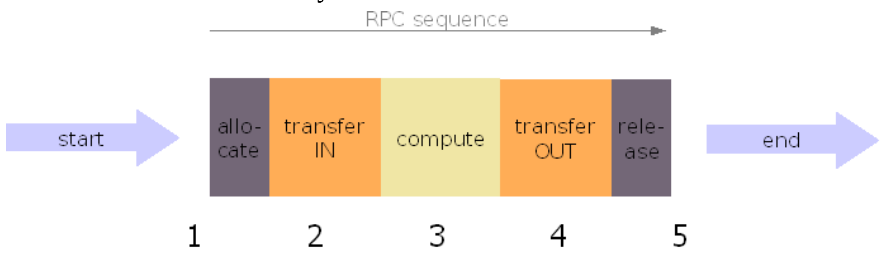

Figure 3: The 5 steps of an HMPP RPC

With HMPP directives, these steps may be performed somewhat implicitly or controlled using the relevant directives.

4.2.1.1.2 Parallelism / Gridification

In the context of HMPP, "gridification" refers to the transformation of codelets' loop nests into computations that can run in parallel on a hardware accelerator such as a GPU. Gridification is about translating blocks of loop iterations into a GPU kernel. Such a pool of computations can be a loop or a loop nest. Nested loops can be merged into a single thread-space, while single loops constitute independent kernels. A single codelet may lead to more than one GPU kernel if it contains multiple loops, therefore a series of loops generates as many kernels. The gridification is complementary to other parallelization methods like the "*vectorization*" or the "*multi-threading*".

In the example below, the **hmppcg gridify** directive is rightfully used since the *i,j* loop nest is parallel, given that HMPP cannot automatically detect it as such yet:

```
#pragma hmppcg gridify(i,j)
for( i = 0 ; i < n; i++ ) {
  for( j = 0 ; j < n; j++ ) {
    float prod = 0.0f;
    for( k = 0 ; k < n; k++ ) {
      prod += A[k*n+i] * B[j*n+k];
    }
    C[j*n+i] = alpha * prod + beta * C[j*n+i];
  }
}
```

Figure 4: A good usage of the hmppcg gridify directive

4.2.1.1.3 Memory model

The memory addresses managed at the host level and at the HWA level are different (see Figure 5). The application and the HWA have their own private memory. HMPP deals with this in a transparent way for the user; it can be seen as the glue between target-specific programming environments and general-purpose programming techniques.

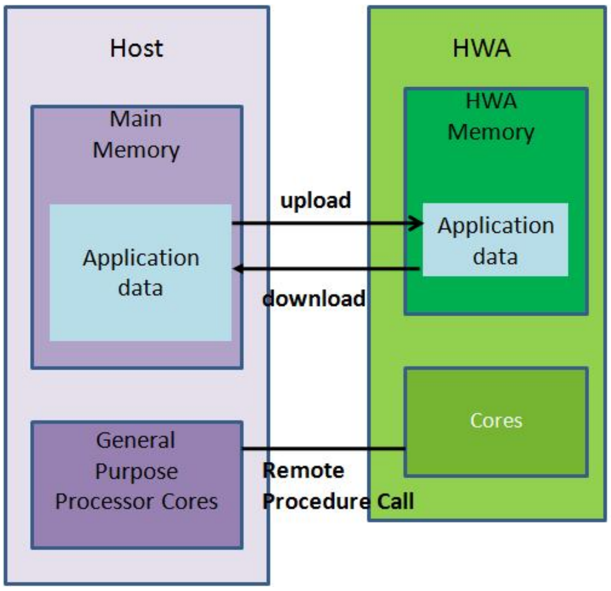

Figure 5: HMPP memory model

4.2.1.1.4 Codelet

A codelet is a computational part of a program located in a function inside the application. It takes several parameters, performs a computation on these data and returns. A codelet is typically a C function or a Fortran subroutine HMPP automatically translates into HWA-specific code such as CUDA or OpenCL [4]. The execution of a codelet is considered atomic: the execution does not have an identified intermediate state or data.

A codelet has the following properties:

- It is a pure function;
- The number of arguments is fixed (i.e. no variable number of arguments like C *vararg*);
- It is not recursive; and
- Its parameters are assumed to be non-aliased.

These properties ensure that a codelet RPC can be effectively remotely executed by a HWA. This RPC and its associated data transfers can be performed asynchronously.

4.2.1.2 Basic directives

Running a first code on a HWA requires few directives. This section introduces the basic directives.

HMPP directives follow the same principle as in OpenMP. Each directive starts, in C, with:

```
#pragma hmpp
```

And in Fortran, with:

```
!$hmpp
```

4.2.1.2.1 codelet / callsite directives

The pair of directives codelet/callsite is the minimum required to get a function to run on an HWA. The codelet directive must be placed on the function declaration, and the callsite directive must be placed on the occurrence of the function call to be offloaded to the HWA. In principle, the codelet should contain parallel loops to get performance on a manycore device. This is the case in the examples below.

```
#pragma hmpp myCall codelet, args[*].transfer=atcall, target=CUDA
void myFunc(int n, int A[n], int B[n]) {
  int i;
  for (i=0; i<n ; ++i)
    B[i] = A[i] + 1;
}

void main(void) {
  int X[10000], Y[10000], Z[10000];
// ...
#pragma hmpp myCall callsite
  myFunc(10000, X, Y);
// ...
  myFunc(1000, Y, Z);
// ...
}
```

Figure 6: Codelet-Callsite example in C

4.2.1.2.2 Compilation and execution

A program with HMPP directives can be compiled by adding the keyword “hmpp” to the beginning of its compilation command line:

```
hmpp gcc basic_codelet_callsite_example.c -o tstC.exe
```

Figure 7: Compiling a C source file with HMPP directives using the GNU C compiler and HMPP

An application compiled with HMPP – i.e. a codeleted application – can be executed by simply running the resulting binary.

4.2.2 Tuning Objective

The objective of the HMPP Codelet Tuning is to tune the performance of a codelet computation in an application using the HMPP Workbench. The plugin targets many-core accelerators like GPGPUs. The performance is evaluated by analyzing the execution time of the codelet.

There are no restrictions to the kind of accelerator used by the application other than it should be available via HMPP. As a portable programming language HMPP provides a way to access a varied set of hybrid accelerators either using the standard OpenCL language or using hardware-specific languages like CUDA for NVIDIA GPUs. However, in this project we restrict the plugin to Linux host systems and will be released with a prototype of the HMPP Workbench different from the Release version provided by CAPS entreprise.

4.2.3 Tuning points and tuning actions

The HMPP Codelet Tuning Plugin has to manage a wide set of parameters that depend on the targeted architecture. For this reason, we propose for this plugin an open approach where the user will be able to express in his own way the tuning experience with HMPP. This open solution has the benefit of providing an API open to more specific and autonomous auto-tuning methods in the future; for example, a user-friendly graphical interface for a specific domain or architecture, or automatic tools provided for a particular machine and application domain.

4.2.3.1 HMPP Codelet Plugin Approach

The tuning object is the HMPP codelet. A codelet may be characterized in many ways that are specific to the target and the computation. There are two kinds of tuning points to consider:

- Static Codelet Tuning Points: operations, transformations, or algorithms used to implement a codelet computation. The user provides Static Codelet Tuning Points with the programming of all the different codelet alternatives. He is free to make any modification inside each codelet alternative. The HMPP "*selector*" extension is used at runtime to execute the various alternatives provided. Example of static Codelet Tuning Points based on HMPP code generation directives are the following:
    - Unrolling factor;
    - Grid size;
    - Loop Permutations;
    - …
- Dynamic Codelet Tuning Points: variable or callback available at runtime during the execution of a codelet, which have an impact on the performance. These Dynamic Codelet Tuning Points can be target-specific.

For now, we simplify this view by considering that all Codelet Tuning Points are dynamic, and that the Static Codelet Tuning Points are directly managed by the user and driven by a callback made available at runtime to control the selector extension. Thus, the HMPP codelet plugin will be a semi-online plugin, i.e. it will require multiple executions of the same application.

The HMPP tuning points and tuning actions are expressed using the following notation:

- The HMPP Region Tuning Point (HRTP) is defined by a callsite – the location of a hybrid computation call in HMPP – that has to be tuned. This HMPP Region Tuning Point is specific to a particular HMPP computation on the accelerator. In the API, it is defined by a unique identifier: the HRTP ID.
- The Codelet Tuning Point (CTP) describes a user-defined static or dynamic tuning point in a codelet and is defined in the API by:
    - A unique ID (CTP ID);
    - A type (one of Boolean, integer, floating point or string); and
    - A HRTP ID (the HMPP Program Tuning Point it refers to).
- The Codelet Tuning Tuple (CTT) is defined by the set of Codelet Tuning Points specific to a particular HMPP Region Tuning Point. It differs from the Variant by the fact that it is local to a particular codelet/callsite. The Codelet Tuning Tuple is defined in the API by:
    - A unique ID (CTT ID);
    - A list of couples made of a CTP ID plus the value taken; and
    - The HRTP ID it refers to.

- The Codelet Tuning Search Space (CTSS) is an ordered list of Codelet Tuning Tuple. This element is like a Variant Space except that it is restricted to one particular HMPP Program Tuning Point.

The Variant Space is built with the full list of Codelet Tuning Search Space in the application. Several Search Algorithms are proposed and one of them is selected by a command in the API, among which are the following:

- Exhaustive search;
- Random Search.

More Search Algorithms could be proposed in the future.

4.2.3.2 A concrete view of the HMPP Codelet Plugin

The HMPP Codelet Plugin has been designed to evolve during time. As a first implementation, we propose a system based on a single compilation scheme, with the extraction at compile time of all the information required by the HMPP Codelet Tuning Plugin. An extended version of the SIR file, called H.SIR file, is used to store this information, although a separate XML file can be used if necessary.

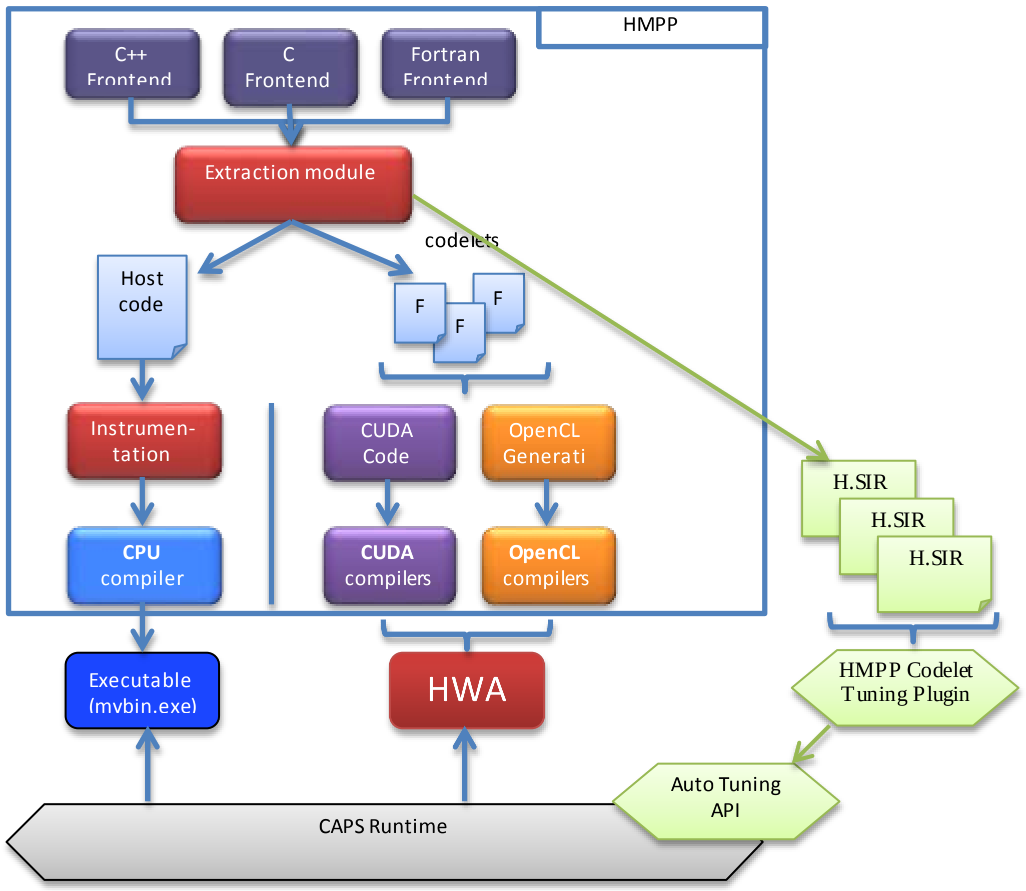

Figure 8: HMPP codelet tuning plugin compilation and execution scheme.

The initial state requires that the user defines in the source code the auto-tuning scope via the specific API. The user has to specify all pieces of information required by the plugin:

- The beginning and the end of the HMPP regions tuning points;

- The definition of the codelet tuning points by the specification of a variable name (the selector) at runtime:
  - The name will lead to the definition of a new environment variable that will drive the value of the selector in the code at runtime.
  - Note that this variable can be used to perform a codelet version selection through a prototype HMPP syntax extension called the "*selector*".
- The definition of codelet tuning search space using a flat list of codelet tuning tuples.
- The selection of a Search Algorithm by an identifier or a name.

The H.SIR file is filled with information extracted at compile time by the HMPP compiler. PTF starts after the compilation stage and no more HMPP compilation should be needed during its execution. From the H.SIR file, PTF together with the HMPP codelet tuning plugin operate the exploration and the selection of the best CTT for each region.

Search algorithms are initially taken from the generic list provided by the infrastructure, like the exhaustive search or the random search. In the future, we may need more advanced search algorithms based on the tuning semantic exhibited by different codelets; for instance, using a hierarchical approach.

We will try to extend this approach with more advanced and automatic tuning strategies directly in the plugin, and with more automatic code transformations minimizing the user's intervention.

4.2.3.3 Analysis Agent Extensions

We may need to extend the Analysis Agents to perform tuning actions. Our tuning actions will be operated using the operating system environment variables as specified by the H.SIR file and described in section 4.2.3.2. This operation can be performed by the plugin during the preparation step or by the analysis agents.

4.2.3.4 Monitor Request Interface Extensions

We will need to develop an extension to the MRI monitor in case we want to support several HMPP region tuning points within the same application. This extension will use the HMPP profiling interface called "*PHMPP*" [5] to extract the exact execution time of the codelet. By default we use the execution time of the full application provided generically by the PTF infrastructure.

4.2.4 Tuning Plugin Interface

In this subsection, we describe an implementation of the plugin in terms of its Tuning Plugin Interface (TPI) methods. An overview is proposed in the Figure 9.

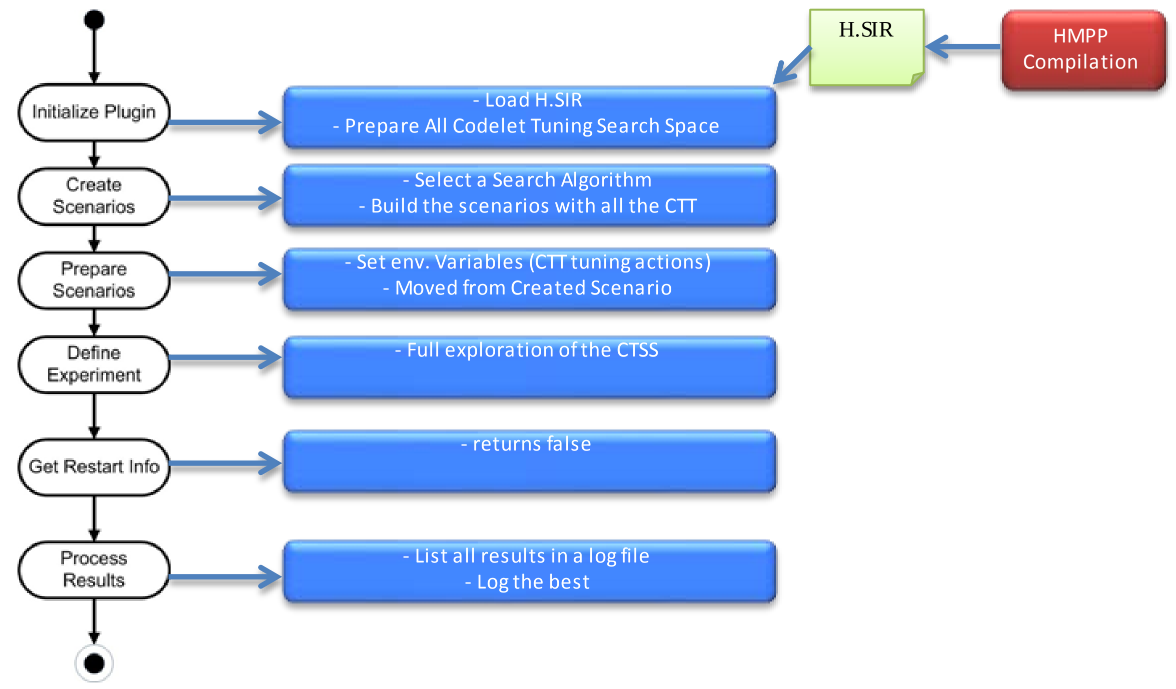
H.SIR
HMPP Compilation
Initialize Plugin
- Load H.SIR
- Prepare All Codelet Tuning Search Space
Create Scenarios
- Select a Search Algorithm
- Build the scenarios with all the CTT
Prepare Scenarios
- Set env. Variables (CTT tuning actions)
- Moved from Created Scenario
Define Experiment
- Full exploration of the CTSS
Get Restart Info
- returns false
Process Results
- List all results in a log file
- Log the best

Figure 9: Overview of the HMPP codelet tuning plugin operation inside the PTF.

4.2.5 Initialize

The initialization sequence reads and analyzes all the HMPP Region Tuning Points extracted from the H.SIR file and builds the basic data structure that operates the plugin like the search algorithm methods. This method also sets some parameters on PTF, to indicate that the plugin requires a measurement of the execution times as well as an automatic restart until the ESP is empty.

4.2.5.1 Create Scenarios

For each HMPP region tuning point, the appropriate search algorithm generates the full set of scenarios from the list of codelet tuning tuples for the region and puts them into the CSP. Inside the same region tuning point, all scenarios are mutually exclusive. However, in theory, all scenarios relative to one region should be independent from those relative to another region. A scenario has to specify also the hardware requirements for a valid execution: the type and the number of accelerators needed and if the scenarios require hardware exclusivity.

4.2.5.2 Prepare Scenarios

In this first design of the HMPP codelet tuning plugin, we establish that the user is responsible for the generation of all codelet versions before PTF runs. If the set of environment variables associated to the scenario is not initialized by analysis agents, the Prepare Scenario step has to do it. Otherwise, the Prepare Scenario’s only duty is to move the scenario from the CSP to the PSP.

4.2.5.3 Define Experiment

An experiment is built by associating the list of available scenarios in the PSP with the list of available accelerators on the target machine. Depending on the hardware requirement, one or more scenarios will be included in an experiment.

4.2.5.4 Get Restart Info

In the first version, no restart of the application should be required since the tuning actions can be executed at runtime. In the future, a restart could be initiated if the HMPP callsite falls into the fallback mechanism (runtime error).

4.2.5.5 Process Results

From all the scenarios executed, the plugin generates a full log with the description of each CTT with the measured performance. The user receives the codelet tuning tuple associated to the lowest execution time among all measurements as the tuning recommendation.

4.3 Energy Consumption via CPU Frequency Tuning plugin[4]

4.3.1 Introduction to the Energy measurement framework

The plugin joins the *cpufreq* and the PAPI-RAPL features to allow therefore the optimization of the energy consumption of instrumented applications at runtime by changing the frequency and governor policy of processors on certain instrumented code regions.

The plugin is mainly composed of two modules: the *cpufreq* module, which provides a mapping of the *cpufreq* kernel subsystem and is in charge of changing the frequencies and governors; and the PAPI counters module, which provides a high-level API for hiding the intrinsic operations and elements of the PAPI interface (see Figure 10) and stand for the energy and performance measurements.

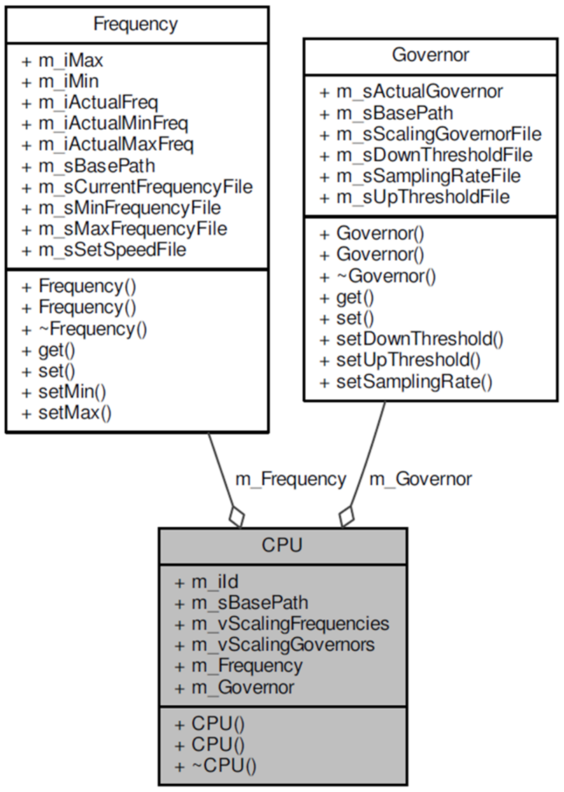

Figure 10: CPU Freq interface

[4] Code name: PTF_EE_CPUFREQ_TP.

Starting with Linux kernel v.2.6.0, processor frequencies can be dynamically scaled through the *cpufreq* subsystem. This dynamic scaling of the clock speed gives some control in throttling the system to consume less power when not operating at full capacity.

The *cpufreq* infrastructure makes use of governors (policies) and daemons for setting a static or dynamic power policy for the system. The dynamic governors can switch between CPU frequencies based on processor utilization to allow for power savings while not sacrificing performance.

There are five possible kernel governors available for use with the *cpufreq* subsystem. These governors set the processor frequency based on certain criteria; some dynamically change the frequency as inputs are changed either by the system or the user [6]:

- Performance: The Performance governor statically sets the processor to the highest frequency available. The range of frequencies available to this governor can be also per parameter adjusted. Generally, this governor is not selected to save power unless the application consistently demands 100% CPU; however, it might be selected to obtain consistent, minimal latency times.
- Ondemand: Introduced in Linux kernel v.2.6.10, the Ondemand governor was the first in-kernel governor to dynamically change processor frequency based on processor utilization. The Ondemand governor checks the processor utilization and if it exceeds a configurable threshold, the governor sets the frequency to the highest available. If the governor finds the utilization to be less than the threshold, it steps down the frequency to the next available one. At this scenario, if the system remains underutilized, the governor continues stepping down the frequency until the lowest available is reached or the utilization increases. The Ondemand governor permits to control the range of frequencies available, the rate at which the governor checks utilization on the system and the utilization threshold. Because the governor requires time to respond to changes in system load, performance might be reduced if the workload utilization changes frequently.
- Conservative: Based on the Ondemand governor, the Conservative governor is similar in the way it dynamically adjusts frequencies based on processor utilization; however, the Conservative governor behaves a little differently and allows for a more gradual increase in power. This governor checks the processor utilization and if it is above the utilization of an "up-threshold" or below the utilization of a "down-threshold", the governor steps up or down the frequency to the next available one instead of just jumping to the highest frequency as the Ondemand governor does. The Conservative governor permits to control the range of frequencies available, the rate at which the governor checks utilization on the system, the utilization thresholds and the frequency step rate.

Powersave: Opposite to the Performance governor, the Powersave policy statically sets the processor to the lowest available frequency. As on the other cases, the range of frequencies available to this governor can be adjusted. The purpose of this governor is to run at the lowest speed possible at all times; the system will never rise above this frequency no matter how busy the processors are. The application performance might decrease without obtaining energy savings.

- Userspace: This governor allows a frequency to be manually selected and set. It is useful for setting a unique power policy that is not present or available from the other governors. Note that the Userspace governor itself does not dynamically change the frequency; rather, it allows a user-space program to dynamically select the processor frequency.

The summarized overview about customizable parameters per governor can be seen in **Table 1**.

	scaling_max_freq	scaling_min_freq	up_threshold	down_threshold	sampling_rate	scaling_setspeed
performance	x	x				
powersave	x	x				
ondemand	x	x	x		x	
conservative	x	x	x	x	x	
userspace						x

Table 1: Governor parameters

Setting a certain governor policy, configuring its parameters, or even changing the processor to work at a certain frequency is done by writing that governor's name, its parameters or the fixed frequency in the correspondent file located at the *cpufreq* file-system that maps the *cpufreq* subsystem.

For choosing the appropriate governor for a given running application, it must be taken into account that a lower processor frequency does not necessarily reduce energy consumption. For example, a certain application might require high processor usage; if the processor frequency is set too low, this application might run for a much longer time and therefore use more energy in the overall than at a higher processor frequency.

The Performance Application Programming Interface (PAPI) [7] aims at providing tool designers and application engineers with a consistent interface and methodology for the use of the performance counter hardware found in most major microprocessors. PAPI enables software engineers to see, in near real-time, the relation between software performance and processor events. One of the new components of the PAPI library is the so-called PAPI-RAPL component [8] that makes use of the RAPL [9] sensors available on the Sandy Bridge microarchitecture [10] [11]; these sensors allow to measure the power consumption of CPU-level components and examine the Model Specific Registers (MSR).

4.3.2 Tuning Objective

The main tuning objective of this plugin is to minimize the energy consumption of an application.

The code may belong to any application field; however, emphasis is given to scientific fields where codes are usually arithmetic-operation-intensives.

On the hardware side we can change frequency policies and read the RAPL counters as described in section 4.3.1. The hardware that will be used is the IBM System x iDataPlex thin-node islands on SuperMUC at LRZ. This system has two processors per node, i.e. two processors with shared memory.

The power consumption can only be measured per package [11] according to the RAPL counter: PP0_ENERGY:PACKAGE0 and PP0_ENERGY:PACKAGE1. Each package corresponds to a processor and consists of 8 Sandy Bridge cores. It also includes the so-called un-core elements such as the last level cache, Integrated IO, QPI, Memory controller, etc.; which are common to all cores in a package.

4.3.3 Tuning points and tuning actions

For the Energy Frequency Tuning plugin we define two different tuning points:

- The available governors; and

- The frequencies to be used.

Once an application has been tuned for performance, it is then eligible for optimization of energy- and time-related costs.

From the point of view of the first tuning point, there are five governors or policies as described in section 4.3.1. Given that the Performance governor and the Powersave governor are special cases of the Userspace, we do not consider them as part of our search space. Thus, we can define the tuning point as an indexed vector of the three governors: Ondemand, Conservative and Userspace.

From the point of view of the second tuning point, the Sandy Bridge system supports, among others, the following frequencies: 2.7, 2.4, 2.2, 2, 1.8, 1.6, 1.4, 1.2 – all provided in GHz. We are leaving the TurboMode (2.701 GHz) frequency outside the range of explored frequencies. Please refer to section 4.3.1 for a detailed description of the available frequency policies.

If experiments demonstrate that it takes a reasonable time for going through all the available tuning points, measurements will be done on all the available tuning points until we find the minimum of the selected strategy. However, trying all the three governors with all possible frequencies in each region might be too time-consuming (24 frequencies per region). As we have already seen in many applications the range of frequencies describes a soft parabolic shape (or similar) to the energy consumed. In such a case we will try a ternary search for each governor. So per governor we will have three experiments:

- The highest frequency (f2);
- The lowest frequency (f0);
- The median frequency (f1).

These 3 frequencies are used to redefine the next search interval until the minimum is found. The upcoming interval is defined by the two neighboring frequencies that resulted in the lowest energy consumption.

4.3.4 Tuning Plugin Interface

4.3.4.1 Initialize

This method creates tuning points (governors and frequencies) from a configuration file.

4.3.4.2 Create Scenarios

The variant space is defined as the cross product of governors and frequencies. Each scenario is a combination of frequency and governor for the phase region. The scenarios are created by exhaustive search.

Note that if exhaustive search is too time consuming we will use a search algorithm as described in section 4.3.3.

4.3.4.3 Prepare Scenarios

Since no recompilation is needed, the created scenarios are moved directly to the PSP.

4.3.4.4 Define Experiment

The experiments are initially executed to request the objective energy and execution time per node. Each experiment must initialize the library by calling the *enopt_init()* API function, and finalize it to close the opened handlers with the energy counters and set the system to the initial frequency and governor settings.

4.3.4.5 Get Restart Info

At the moment, we explore the energy savings only on the phase region. Therefore, this method simply returns *false* to indicate that no restart is required.

4.3.4.6 Process Results

The best combination of energy savings for the phase region with its respective found governor and frequency will be selected and provided to the user as a recommendation.

4.4 Master-Worker MPI Plugin[5]

4.4.1 Tuning Objective

The plugin uses the standard MPI analysis of Periscope to determine the values of the two tuning parameters of Master-Worker applications: the partition factor for the tasks to be processed by the workers and the adequate number of workers for the application.

First, by introducing a data partition factor, instead of distributing the whole set of tasks among workers and then waiting for the results, the master partially distributes the tasks by dividing this set of tasks into different portions called "batches". The number of tasks assigned to each batch depends on the distribution strategy, and it may be different from one batch to another. 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 it a new chunk from the next batch; 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. Still, we assume that communications are free; the application executes on a dedicated and homogeneous platform; we achieved a perfect load balancing; and the computation also scales ideally. In this ideal world, any available resource that can be assigned to the application must be assigned, because it can be efficiently used to improve the application's performance.

In the real world, however, we can observe that 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, the tuning objective of this plugin is to balance the application execution and adapt the number of workers used for the execution. Both a balanced execution and an adequate number of workers, where each worker is efficiently used, reduce the total execution time.

[5] Code name: PTF_WPMPI_TP

4.4.2 Tuning points and tuning actions

The tuning points of this plugin are two variables:

- The partition factor; and
- The number of workers.

The tuning action is to change the value of these variables and perform a new experiment, i.e. execute a small number of iterations of the application. Both tuning actions are performed in the Master code. Depending on the partition factor, the application is executed with less or more partitioned input data. The number of workers indicates how many processes Periscope must create.

4.4.3 Tuning Plugin Interface

Before the execution of the experiments, the application must be prepared for tuning. In this case, a user should create the configuration file specifying the MPI application pattern and the tuning points that correspond to the variables in the code, for example number of workers to run. Moreover, certain tuning points must be annotated using Periscope directives in the source code; for example, a partition factor. These actions must be performed before the instrumentation of the application code. Finally, the instrumentation is done and the SIR file is generated.

4.4.3.1 Initialize

One of the tuning points is extracted directly from the SIR file and created after parsing the user code; the tuning point contains the information provided using the Periscope pragmas. This is the case for the partition factor, which is linked to a variable in the code. The other main tuning point – the one controlling the number of workers – can be obtained from the configuration file. The tuning space is composed of those tuning points and is explored by an exhaustive search.

4.4.3.2 Create Scenarios

The scenarios are created iteratively. Each iteration consists in creating two sets of scenarios, each one to obtain the best value for a tuning point:

- First, all values for the partition factor are tested, so scenarios are created for each value and a fixed amount of workers. After executing these scenarios, the best value for the partition factor is obtained.
- Then, the second round of scenarios is created, by fixing the partition factor on the obtained value and varying the number of workers. If the best value for the number of workers is different from the previous one, then another iteration starts.

4.4.3.3 Prepare Scenarios

As long as no configuration files are changed no recompilation is needed. The created scenarios are simply moved to the PSP.

4.4.3.4 Define Experiment

The scenarios run separately. They cannot be executed in parallel since multiple threads are needed for each experiment.

4.4.3.5 Get Restart Info

When exploring the number of workers, the plugin needs to restart the application with a new set of processes to suit the needs of the varying size of the worker pool for each execution scenario.

4.4.3.6 Process Results

The best combination of values for the tuning points is selected based on execution-time metrics and provided to the user as a recommendation.

4.5 MPI Runtime Plugin[6]

4.5.1 Tuning Objective

This plugin uses the standard MPI analysis of Periscope to implement a combined tuning strategy to determine the values of multiple tuning parameters: a set of MPI environment variables and a set of communication functions chosen among different variants.

First, there are many environment variables associated to specific implementations of the MPI library; in particular, the IBM MPI library for SuperMUC offers more than 50 configurable parameters. Changes to some of these parameters could significantly affect the communication times of an application. The plugin assumes inputs are SPMD applications, applies the same optimization to all processes, and takes into consideration two sets of variables:

- MP_BUFFER_MEM (amount of memory for buffering data from early arrival messages) plus MP_EAGER_LIMIT (threshold value of the message size for changing from the eager to the rendezvous protocol); and
- MP_TASK_PER_NODE (number of tasks run on each physical node) plus MP_TASK_AFFINITY (for attaching parallel jobs to system cpusets).

Tuning the first pair of variables could be particularly effective on applications that interchange messages of uniform size, while tuning the second pair could be effective on applications with clustered communications. This plugin’s initial tuning objective is to find a proper combination of values for the pair of variables MP_BUFFER_MEM – MP_EAGER_LIMIT. In addition, the tuning strategy consists in systematically launching the application using different combinations of values for these variables. At the end, the tuning recommendation consists of the values for these variables that led to the application’s lowest execution time.

Second, depending on the input data, the communication efficiency can also be significantly affected by the MPI functions chosen for performing communication. Consequently, implementing different variants for the communication code and being able to choose among them depending on the input data can be effective for improving the performance of certain applications. Therefore, the tuning strategy of this plugin consists in testing the application using different variants of the communication code provided by the user, and the tuning recommendation is the variant that led to the lowest execution time.

Summarizing, this plugin involves a combined tuning strategy in the search space defined by the values tested for the chosen MPI environment variables and the variants of the communication code provided by the user.

[6] Code name: PTF_RTMPI_TP.

4.5.2 Tuning points and tuning actions

For this plugin, we have two kinds of tuning points:

- MPI environment parameter; and
- Different code functions that provide different implementations of the same functionality.

The MPI environment parameters may be set before the application executes. Two tentative pairs of parameters are proposed:

- MPI application mapping: adapting tasks per node/core, adapting the affinity of the processes. MPI parameter - tuning point:
    - MP_TASK_PER_NODE (To specify the number of tasks to be run on each of the physical nodes),
    - MP_TASK _AFFINITY (Setting this environment variable attaches task of a parallel job to one of the system cpusets)
- MPI communication buffer/protocol: adapting the sending/receiving buffer, analyzing the size pattern of the messages, adapting the communication protocol (eager/rendezvous). MPI parameter - tuning point:
    - MP_BUFFER_MEM (To control the amount of memory MPI allows for the buffering of early arrival message data. Message data that is sent without knowing if the receive is posted is said to be sent eagerly. If the message data arrives before the receive is posted, this is called an early arrival and must be buffered at the receive side),
    - MP_EAGER_LIMIT (To change the threshold value for message size, above which rendezvous protocol is used)

All these variables may be set by the environment variables or by the *mpirun* options (flags). We assume that our 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.

For the code functions tuning point, the user must provide different code versions of the function inside the application code (in *main()* function). He/she must annotate the code (using user regions and adding attributes to them):

- A new pragma *mchoice* (multiple choice) indicates that there are many versions of the same functions. This pragma has an attribute *v* that indicates how many versions exist. This attribute is treated as a tuning point and is exported to the SIR file.
- A new pragma *dependency_mchoice* will indicate a condition and a range of variant to explore when the condition is true. For example, *dependency_mchoice v==5, bsize=128-128KB* means that for the version 5 of the function, the range for bsize is 128 to 128KB.

The tuning action is to execute the application changing the implementation of the functions and using different attribute values (if it is the case).

4.5.3 Tuning Plugin Interface

Before the execution of the experiments, the application must be prepared for tuning. In this case, the user should create the configuration file specifying the configuration options for the MPI library and a range of valid values for each of them. Moreover, certain tuning points must be annotated using Periscope directives in the source code; for example, the user must indicate the version number of the code variant that he/she had chosen. These actions must be performed before the instrumentation of the application code. Finally, the instrumentation is done and the SIR file is generated.

4.5.3.1 Initialize

Some tuning points are extracted from a configuration file containing the configurable options of the MPI library and a range of values valid to each of them. Additionally, the user may provide other tuning points with pragmas (Periscope directives) in the source code, referring to multiple implementations of the same algorithm.

Each option becomes a tuning point, and has different possible values. Depending on the type of the configuration option, the valid values for each tuning point may vary; some configuration options require a Boolean value while others need a string of characters or a range of integers. For the tuning point regarding multiple implementations of a function, the values select one among the several options provided, so the value type is integer.

The tuning space is a two-dimensional space generated by the multiple values of each MPI option and function implementation. The search algorithm used to explore the tuning space performs an exhaustive search of all the combinations; thus, it is necessary to cut down the amount of MPI options being tuned to those with greater impact on the application performance.

4.5.3.2 Create Scenarios

For the initial plugin design, as explained in section 4.5.3.1, the tuning space is searched exhaustively, so scenarios need to be created for each combination of values for the tuning points. These scenarios are separate executions of the application, using specific combinations of values for the MPI options, and are monitored to obtain the execution time of the relevant regions.

4.5.3.3 Prepare Scenarios

As long as no configuration files are changed no recompilation is needed. The created scenarios are simply moved to the PSP.

4.5.3.4 Define Experiment

The experiments are initially carried out for a single scenario, which uses all the resources to obtain relevant data for large executions. Monitoring takes place globally.

4.5.3.5 Get Restart Info

Application restart is requested with the new MPI options. The application has to be restarted in order for the changes in configuration to take effect.

4.5.3.6 Process Results

The best combination of values is selected based on the execution-time metrics obtained. These values are returned to the user as the recommended configuration for the MPI library.

4.6 Compiler Flag Selection Plugin[7]

4.6.1 Tuning Objective

The tuning objective is to reduce the execution time of the application's phase region. The most important factor, besides the choice of the algorithm and the way the program is written, is the compiler which is generating machine code from the high-level source code. Compilers apply a large number of program transformations to generate the best code for a given architecture, e.g., loop interchange, data prefetching, vectorization, and software pipelining. The compiler ensures the correctness of the transformations but it is very difficult to predict the performance impact and to select the right sequence of transformations. Therefore, compilers offer a long list of compiler flags and even directives to allow the programmer to guide the compiler in the optimization phase.

Due to the large number of flags and the required background knowledge in the compiler transformations and their interaction with the application and the hardware, it is very difficult for the programmer to select the best flags and to guide the compiler by inserting directives. Thus, typically, only the standard flags O2 and O3 are used to change the approach of the compiler optimization.

4.6.2 Tuning points and tuning actions

The tuning points of this plugin are the individual compiler flags of the compiler. Each tuning point can either be switched ON or OFF. Thus, these tuning points have only two values. The tuning action is to switch on the flag in the program recompilation. All tuning actions of the individual tuning points are combined in the preparation step.

4.6.3 Preparation for Tuning

In order to prepare the application for automatic tuning, the *makefile* needs to be adapted so that the compiler switches selected can be applied when recompiling the application. This adaptation is done by incorporating special environment variables into the compilation step. Furthermore, the code's phase region needs to be instrumented using the PSC instrumenter.

4.6.4 Tuning Plugin Interface

4.6.4.1 Initialize

The tuning points for a given compiler are initialized from a configuration file. The configuration file specifies which flags can be used together and which should never be used in the same compilation step. It can also specify fixed combinations of flags that can be applied by the plugin. In addition, the configuration file may specify a mapping of single-node performance properties to compiler flags that might help improve the performance issues.

The tuning space created from the individual tuning points can then be shrunk by taking into account the performance properties found for the program's phase region. To obtain the performance properties, the plugin needs to run a single-core performance analysis with exclusive instrumentation of the phase region.

[7] Code name: PTF_CFS_TP

4.6.4.2 Create Scenarios

The plugin inspects the tuning space created from the specified flags. The prototype is based on an exhaustive search of the tuning space and applies the standard search algorithm. For each possible variant, the plugin creates a scenario. The region specification is the application's phase region; the variant identifies the flags; and the objective is the overall execution time.

4.6.4.3 Prepare Scenarios

For each scenario the entire code needs to be recompiled. Thus, only a single scenario can be prepared for each experiment. The selected flags are enforced in the compilation step by the modification of some predefined *makefile* variables used for tuning which have to be appropriately employed by the programmer.

In order to reduce the recompilation time, the plugin can restrict the recompilation to the most time-consuming files by touching only those source files and not executing *make clean* before recompilation.

4.6.4.4 Define Experiment

This function only moves the prepared scenario into the experiment scenario pool. It specifies process 0 as rank of the scenario. All the processes and threads run the compiled code and the objective needs to be evaluated only in a single process.

4.6.4.5 Get Restart Info

It requests an application restart without any additional parameters.

4.6.4.6 Process Results

This method retrieves the best variant from the search algorithm and returns this combination as a tuning recommendation.

5 Future works

This document described an extensive work in the design of a new set of auto-tuning plugins. The implementation of the full infrastructure will be useful to get a better understanding of the behavior and the pertinence of each plugin on its area. From this experience, we will be able to improve probably in many ways the potential of this innovative tool.

However, during the multiple meetings the partners had for the design of the PTF, we have already detected several opportunities for improvement that may be explored in the future.

Mixing plugins

As already described in the introduction, PTF has been designed to support at the same time multiple types of independent plugins. In other words, it consists in merging several different performance aspects of an application into a single auto-tuning method; for example, MPI optimizations combined with energy optimization, or optimization of HMPP codelets combined with compilation-option optimization.

This feature requires first to address the technical integration of multiple plugins, via for instance a "*meta*" plugin that manages or drives several others. But secondly, and much more complex, it also needs that we define tuning strategies and methods to combine variants in order to drive the auto-tuning sequence in an efficient way – with convergence and a better performance when compared to the execution of the same plugins in sequence, one after the other. Methods using variant space algebras and polytope representations could be explored.

Solving constraints while building experiments

The variant spaces given to a search algorithm can be quite complex since various constraints might apply (e.g. triangular or sparse variant spaces). The problem of enumerating the feasible points in such complex spaces has been scientifically studied for many years under the name of Constraint Satisfaction Problems (CSP). The literature proposes a large set of solutions to operate this computation with good performance and these techniques could be integrated into the PTF framework.

Detecting advanced patterns for variant search algorithms

The enumeration of variants can explode in complexity. In order to reduce the amount of variants to execute, we could introduce pattern-matching techniques to integrate the detection of good or useless variants, based on either previous runs or on user-provided expert knowledge.

Developing new plugins

Based on the flexibility of the framework proposed and the variety of different plugins introduced in this project, a wide set of new tuning plugins will be possible in the future. In a longer term, new plugins could be introduced, such as for OpenMP-threads optimization and algorithm method explorations.

6 Bibliography

[1] NVidia, "NVIDIA, Cray, PGI, CAPS Unveil 'OpenACC' Programming Standard for Parallel Computing," 11 14 2011. [Online]. Available: http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09 &version=live&prid=821214 &releasejsp=release_157.. [Accessed 01 09 2012].

[2] HPCwire, "CAPS Entreprise Now Supports OpenACC Standard," 02 05 2012. [Online]. Available: http://www.hpcwire.com/hpcwire/2012-05-02/caps_entreprise_now_supports_openacc_standard.html.

[3] CAPS entreprise, *HMPP Directives Reference Manual, Version 3.2.0*, 2012.

[4] The OpenCL Specification v1.1 r36, "The OpenCL Specification," 30 9 2010. [Online]. Available: http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[5] CAPS entreprise, "H4H - HMPP Profiling Event specification; Version 2.3.3," Rennes, 2012.

[6] IBM Corporation, "The CPUFreq governors," 5 10 2012. [Online]. Available: http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=%2Fliaai%2Fcpufreq%2FTheCPUFreqGovernors.htm. [Accessed 5 10 2012].

[7] Innovative Computing Laboratory, ICL, "PAPI, Performance Application Programming Interface," 20 09 2012. [Online]. Available: http://icl.cs.utk.edu/papi. [Accessed 5 10 2012].

[8] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra and S. Moore, "Measuring Energy and Power with PAPI," *The 1st International Workshop on Power-Aware Systems and Architectures*, 2012.

[9] R. A. Lupusoru, "Github, Intel-RAPL-via-Sysfs," 10 06 2011. [Online]. Available: https://github.com/razvanlupusoru/Intel-RAPL-via-Sysfs. [Accessed 5 10 2012].

[10] Intel Corporation, " Intel Xeon Processor," 2012. [Online]. Available: http://www.intel.com/xeon. [Accessed 5 10 2012].

[11] Intel Corporation, Intel 64 and IA-32 Architectures Software Developer Manual, 2012.