European Community Seventh Framework Programme

Theme FP7-ICT-2011-7

Computing Systems

Automatic Online Tuning (AutoTune)

D4.2

Prototype Plugins

Laurent Morin

Date of preparation (latest version): October 15th, 2013

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	D4.2
Deliverable Name	Report on the Prototype Plugins
Authors	Enes Bajrovic (UNIVIE), Eduardo Cesar (UAB), Houssam Haitof, Michael Gerndt (TUM), Carla Guillen (LRZ), Renato Miceli (ICHEC), Laurent Morin (CAPS), Carmen Navarrete (LRZ), Antonio Pimenta (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

Executive Summary

The AutoTune project proposes to improve the performance of parallel applications on a wide range of machines and architectures with an innovative and ambitious auto-tuning and performance analysis software: the Periscope Tuning Framework. The toolset is composed of an auto-tuning infrastructure providing the support for OpenMP and MPI performance analysis; and of a set of plugins extending the tuning capabilities of the system to many different aspects: compilation options, MPI runtime parameters, GPU execution schemes, or energy consumption. This report describes all the contributions made by the AutoTune partners in their implementation of the tuning plugin, and the improvements they achieved on the overall tuning framework during the second year of the project.

The first major contribution of the Periscope Tuning Framework is its capability to provide not only a complete infrastructure for high performance auto-tuning but also a theoretical and a methodological approach that has been very useful for advanced plugin developers for the design of their plugin. From this effort, the project steered a strong collaboration between partners that led to the improvement of both the terminology and the implementation of the tuning framework.

Each advanced tuning plugin has been implemented into a first prototype. The plugin for High level Parallel Patterns for GPGPU is now able to manage five tuning parameters and tuning actions, and by the way tune the runtime optimization of pipeline patterns. This first prototype has been fully integrated in the Periscope infrastructure with the support of a new pipeline execution time metric. The Energy Consumption plugin has solved complex integration issues with the operating system, the energy measurement system, and the tuning infrastructure. The proposed solution with an independent library and corresponding daemons will probably be useful for other plugin developers to take into account the energy consumption aspect during the optimization process. The advanced MPI plugins provide new tuning and optimization strategies to the Periscope infrastructure. It is certain that the MPI programming model optimization is key in the performance of future parallel applications. The compiler parameter optimization showed that the infrastructure is also able to the support less intrusive but important tuning strategies based on the build process. This development will open new optimization opportunities to other plugins.

The results obtained after the second year of the AutoTune project are very encouraging and will motivate the plugin developers to finish the development of their plugins and start the most interesting and promising part: the combination of all the technologies and techniques into even more innovative tuning strategies for the benefit of HPC application performance.

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 multi-level 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 implementation of the tuning plugins that enable PTF to automatically tune applications. It covers the evolution of the PTF design improved with the work of all partners. The main part of this deliverable describes the implementation of an initial prototype of each of the advanced tuning plugins already described in the deliverable D4.1.

1.1 Status by the end of the year one

The first year of the AutoTune project led to the definition of a theoretical and practical framework designed to develop quickly and efficiently tuning plugin for a wide set of cases and contexts. Most of the infrastructure, based on the Periscope software, has been implemented during this period, and experimented through a basic demo plugin. From that period, we could deduce and prepare the design of each plugin with a strong methodology and theoretical approach. From that extensive work, the second year of the project has been dedicated to the implementation of the first prototype of each plugin.

1.2 Year two contributions

The key result obtained during the first year of the project was the implementation of a first demo plugin in the PTF infrastructure. This first demo plugin has been the basis for the implementation of all tuning plugins during the second year. The theoretical efforts made during the previous period have been in most cases successfully validated with the experimentation of the first implementation. In some cases, the advanced tuning plugin has evolved in the way that it is integrated inside the PTF infrastructure. In all cases, the strong collaboration between the Periscope Framework developers and the advanced tuning plugin developers helped in improving the tuning terminology, the infrastructure, and the effectiveness of the plugins. By the end of the second year, we can already establish promising results from all advanced tuning plugins, but also foresee future collaborations between some plugins. We can mention the development of a flexible energy measurement library that can be useful to other plugins. We can also mention the flexibility of the compilation option tuning plugin – that does not need anymore any instrumentation or strong interaction with the monitoring system. This flexibility will simplify the combination of different plugins to enable multi-aspect application tuning.

2 Tuning Plugin Framework Developments & Evolutions

2.1 Framework Overview

AutoTune develops the Periscope Tuning Framework (PTF) as an extension of Periscope. It 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 is 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.

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 of regions.

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. The agent hierarchy is afterward built on the distributed nodes of the application.

Periscope uses **an analysis strategy**, e.g. for MPI, OpenMP and single core analysis, to guide the search for performance properties – the particular aspect or metric we want to optimize. 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 started 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 is controlled by a specific **plugin strategy**. This strategy guides the search for a tuned version. The search space is 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 are able to run the experiments without an application restart, e.g. the energy efficiency tuning plugin; while semi-online plugins requires 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 has to be leveraged. The execution of experiments with and without application restart is already supported by Periscope.

Once the tuning process is finished, PTF generates a tuning report documenting the resulting 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.

2.2 Tuning Framework Improvements

The initial design of the framework as presented in deliverable D4.1 Design of the Tuning Plugins has proven to be viable for the individual tuning plugin prototypes developed in the second year. The tuning plugins are based on the Tuning Plugin Interface (TPI) that allows to drive the plugins from inside the PTF frontend by calling standardized functions provided by the plugin. The plugin execution is based on four scenario pools that keep the information about scenarios to be executed and measured on the target system. 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.

The basic execution of the plugins happens in the following steps:

- **Initialization**: First, the plugin is initialized and the tuning parameters are created.
- **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 properties, either via a generic search algorithm, e.g., exhaustive search, or by its own search algorithm.
- **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.
- **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.
- **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 properties are returned to the plugin.

- **Process Results**: The plugin accesses the properties, which are standard Periscope properties. Each property 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.

This initial design was extended in the following ways:

Multiple Tuning Steps: The plugins can go through multiple steps of scenario creation, preparation, execution, and result processing.

Tuning Step Pre-Analysis: Each tuning step optionally starts with a Periscope analysis that provides performance properties for guiding the tuning.

Scenario Execution Analysis: In each experiment a Periscope analysis can be performed to check the influence of the tuning variants on the detected performance properties.

2.2.1 Multiple Tuning Steps

The extended version of the Tuning Plugin Interface (TPI) supports multiple tuning steps. The motivation for this extension is that a plugin might go through multiple rounds of executing scenarios. In each tuning step, it can for example optimize a subset of the tuning parameters that is independent of the other tuning parameters. The tuning steps can also be used to first run a specific analysis that requires the execution of multiple scenarios to guide the tuning in the second tuning step. For example, the scalability of OpenMP regions can be explored before tuning the number of threads executing each region to reduce the energy consumption. The scalability information can guide the selection of the range for the tuning parameter.

The TPI extension provides functions *startTuningStep()/finishTuningStep()* to start and end a tuning step. After the plugin initialization, multiple tuning steps can be executed. Each step starts with a tuning step initialization and ends with a tuning step finish operation. In the initialization, the tuning parameters to be explored in that step can be initialized. After the initialization, the scenarios are created, prepared, executed, and evaluated. At the end of the execution of the scenarios, the tuning step is finished. It performs any post-processing of the results obtained from this tuning step. The PTF frontend calls an additional function *tuningFinished()* to check whether an additional tuning step has to be started.

2.2.2 Tuning Step Pre-Analysis

The plugins might need to run a tuning plugin specific or a general Periscope analysis first, before the tuning can be performed based on the analysis results. For example, the MPI analysis of Periscope indicates the relative importance of MPI bottlenecks for individual call sites. This information can be used by the plugin to select those call sites that will be tuned subsequently with respect to the MPI library parameters.

To enable this pre-analysis in each tuning step, the plugin provides a function preanalysisRequest() that is called by the PTF frontend after startTuningStep(). This function can return a request object that determines which analysis strategy has to be executed. Before the PTF frontend will continue with triggering 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.

2.2.3 Scenario Execution Analysis

This TPI extension allows the execution of 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. This can be for example the PTF's MPI analysis, not only to obtain the execution time of the phase region of the application but also to receive performance properties for the MPI calls. A comparison of the returned properties with the properties obtained in the Tuning Step Pre-Analysis for the original program execution can give a detailed insight on the effect of selecting special algorithms implementing collective MPI operations.

The analysis is requested for an experiment. This request is just an extension of the defineExperiment() TPI operation. This operation assembles an experiment from the prepared scenarios and can now also return an analysis strategy request to the PTF frontend. This analysis will then be executed during the scenario experiment. The found properties are returned to the plugin in a special property pool.

With these extensions, the Tuning Plugin Interface provides the following operations:

```
virtual void initialize(string sirFilePath);
```

This function initializes the tuning plugin.

```
virtual void startTuningStep(void);
```

It starts a new tuning step.

```
virtual const StrategyRequest* preanalysisRequest();
```

It requests the tuning step pre-analysis by returning a strategy request object with the information which analysis strategy is to be executed and optional strategy parameters.

```
virtual void createScenarios(void);
```

This function creates the scenarios into the CSP either via a predefined search algorithm or by creating the scenarios with a plugin specific algorithm.

```
virtual void prepareScenarios(void);
```

Some scenarios might not be directly executable but have to be prepared, e.g., some parameter files have to be defined or the code has to be recompiled.

```
virtual const StrategyRequest* defineExperiment(int numprocs);
```

It assembles scenarios that can be executed in a single experiment. The number of MPI processes can be used to assign the scenarios to individual processes to profit from parallel evaluation. It returns a strategy object, if an analysis strategy should be run during the experiment.

```
virtual bool restartRequired(string*, int*, string *, bool *);
```

This function indicates to the PTF frontend, that the application has to be restarted. It can return parameters for the application startup, for example the MPI parameter setting.

```
virtual bool searchFinished(void);
```

The PTF frontend executes scenario experiments as long as there are scenarios available in the CSP. This function is called when the CSP is empty to check whether additional scenarios will be created based on the results of the previous experiments. This allows to execute multi-step search strategies.

```
virtual void finishTuningStep(void);
```

At the end of a tuning step this function is called to postprocess the results of the execution.

```
virtual bool tuningFinished(void);
```

This function checks whether the plugin finished the tuning. If it return true, a next tuning step is started.

```
virtual void getAdvice(void);
```

This function generates the tuning advice for the programmer.

```
virtual void finalize(void);
```

The plugin finalizes the execution.

```
virtual void terminate(void);
```

The plugin terminates.

2.3 Tuning Framework Terminology Evolution

The terminology used in the project is very stable and did change only slightly since the end of the first year. The major terminology change is that we no longer use the term Tuning Point for an aspect of the application that can be modified. This term was changed into Tuning Parameter since this is more frequently used by other projects and is less ambiguous. The major confusion resulting from the term Tuning Point is that people are tempted to think about code regions to which the modifications are applied. Although a clear definition in the project would solve this problem from an academic point of view, the change to a more intuitive term allows the project to more easily disseminate and exploit the results of the project.

An additional change in the terminology is the clarification of the term Objective Function. Before, we used the term objectives for the properties that are evaluated by the PTF analysis agents for the tuned region. Since these objectives are standard properties in the terminology of PTF, we now use the term property in the scenario specification (see below). The term Objective Function is now used in the same way as in other auto-tuning works. It is a function that determines the best scenario. PTF provides predefined objective functions that determine the maximum or minimum of the severity of

the properties returned for all the scenarios. The plugin developer can also define its own objective functions instead, if, for example, a weighted sum of the severities of multiple properties determines the fitness of a scenario, such as in the case of combining execution and energy consumption.

In addition to this renaming, we also had to come up with a more precise definition of an experiment that is to be executed to evaluate the effect of one or more scenarios. This definition is outline based on a formal grammar notation.

```
<experiment>          =  <scenario list>
```

A single experiment consists of a list of scenarios that will be executed and evaluated.

```
<scenario>            = (<tuned region>, <property request>,
                          <tuning spec list>, <scenarioID>)
```

Each scenario consists of a tuned region for which a number of properties are evaluated, a tuning specification list and a scenario id. The scenario id is used to match back the properties to the scenario and to evaluate the objective function. The tuning specification determines the tuning parameter settings which determine the modification of the program execution that is to be evaluated.

```
<scenarioID>          = <int>
```

The scenario is simply an integer value identifying the scenario.

```
<tuned region>        = <program region>
```

Tuned region is a program region for which the plugin optimizes the objective function.

```
<property request>    = (<list of property ids>,<ranks>)
```

The property request is a list of property ids and a list of MPI ranks where the properties will be evaluated. The requested properties represent the input to the objective function that will be optimized by the plugin. The properties are standard performance properties of Periscope, for example, returning the execution time of the program region. This specification allows the measurement of the tuning effect in a specific process while the tuning actions might be executed in other processes.

```
<ranks>               = ALL | <int> | <range list>
```

This is a specification of ranks for the properties or the tuning actions. The specification provides shortcuts for all MPI processes and ranges of MPI processes.

```
<range>               = (<int>,<int>) | <int>
```

Specification of process ranges.

```
<tuning spec>         = (<variant context list>,<ranks>,<variant>)
```

The tuning specification determines the regions and ranks to which a certain variant is applied. If a list of regions is given, the same variant is applied to all of them in all the processes specified via ranks.

```
<variant context>    = <variant region> | <file> | PROGRAM
```

The variant context can either be a program region, the constant PROGRAM if the tuning actions for the variant are to be applied to the program restart or at the very beginning of the execution, or a file name. The latter specification is for example used in the CFS plugin to specify which files of the program need to be recompiled with the given switches.

```
<variant>            = list of (<tuning parameter>, <value>)
```

The variant is the list of tuning parameters with the values that determine the modification to the program execution that are to be evaluated for a given variant context.

3 Advanced Tuning Plugin Prototypes

3.1 Tuning of High level Parallel Patterns for GPGPU

3.1.1 Tuning Prototype Overview

We have developed a prototype tuning plugin for high-level pipeline patterns for CPU/GPU-based systems. The plugin supports automatic tuning of applications that are based on the well-known pipeline pattern.

Listing 1 gives an example of such pattern, where face detection is performed over the stream of images. Pipelines are realized based on while-loops with source-code annotations and pipeline stages correspond to calls to multi-architectural components, for which multiple implementation variants may be provided.

```
...
#pragma pph pipeline with buffer (UNORDERED, N*2)
while ( inputstream >> file ) {
  ReadImage(file, image);
  ResizeAndColorConvert (image , outimage);
  #pragma pph stage replication(rfactor)
  DetectFace(outimage);
  #pragma pph stage with buffer (PRIORITY, N*2)
  WriteFaceDetectedImage(file, outimage);
}
...
```

Listing 1: Example of a high-level pipeline pattern code for image processing. Pipeline consists of four stages. For the compute intensive DetectFace stage different component implementation variants for GPU and CPU exist and the PTF determines the best replication factor such that all execution units of the target architecture are exploited and execution time is minimized.

The associated high-level language, transformation framework and runtime system has been originally developed the EU project PEPPHER [1]. Our plugin has been fully integrated into PTF and successfully evaluated with a real-world face detection application on a state-of-the-art CPU/GPU based system. The currently supported tuning parameters include the stage replication factor, buffer size, number of execution units, and runtime scheduling policy.

Within the PEPPHER framework, pipeline patterns are realized based on while-loops with source-code annotations. Pipeline stages usually correspond to calls to multi-architectural components, for which multiple implementation variants may be provided. Such component implementation variants may be optimized for different execution units of a heterogeneous target architecture (e.g.: systems equipped with GPUs).

The high-level pipeline code is transformed by a source-to-source compiler into code that utilizes the pipeline coordination layer. The pipeline coordination layer manages all aspects of pipelined 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, the pipeline coordination library utilizes the StarPU [2] heterogeneous runtime system, which is responsible for dynamically selecting suitable component

implementation variants for pipeline stages and for scheduling their execution to the different execution units of a heterogeneous many-core system in a performance- and resource-efficient way. StarPU also manages data transfers between execution units, ensures memory coherency, and provides support for different scheduling strategies, with the goal of utilizing all execution units of the target architecture.

Our pipeline coordination layer has been extended in such a way that it supports dynamic reconfiguration by exposing a set of tuning parameters, thus allowing external tuners like PTF to automatically tune the performance of applications using this pattern.

3.1.2 Tuning Plugin Design

The current version of the plugin interacts with the pipeline coordination layer in order to construct and explore a variant space and find the best scenario with respect to the tuning objective. We aim to optimize overall pipeline throughput, which translates to minimum execution time of the whole pipeline.

The pipeline coordination layer currently exposes five tuning parameters:

- the stage replication factor, which determines the number of stage instances that may be executed in parallel,
- the sizes of buffers to hold data packets passed between pipeline stages,
- the number of CPU cores,
- the number of GPUS to be used, and
- the scheduling strategy used by StarPU for scheduling component calls to free execution units of the target system.

The relevant classes and respective tuning parameters of the pipeline coordination layer are shown in Figure 1.

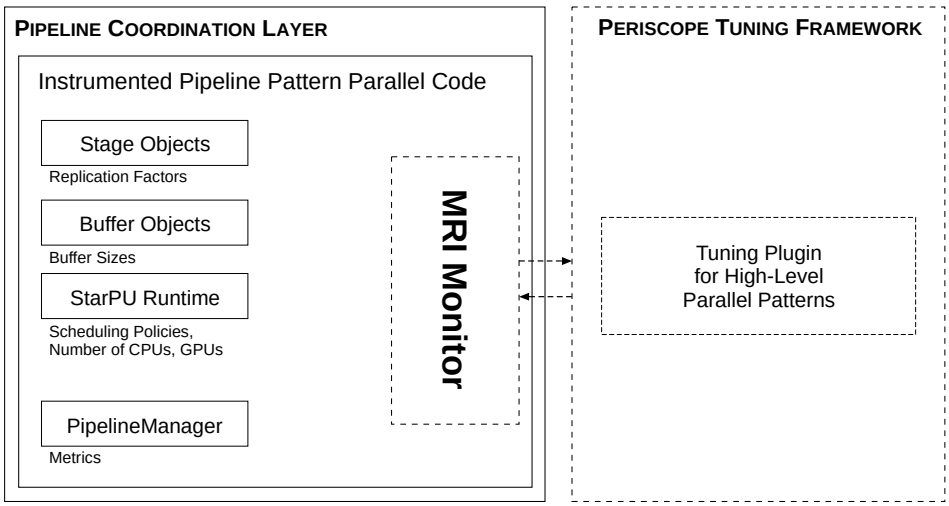

Figure 1: Simplified view of the interaction between PTF and the Pipeline Coordination Layer. Left side of the figure shows the coordination layer and exposed tuning parameters. The PipelineManager class provides methods for passing measured data upon request from MRI. The instrumentation of the application includes calls to MRI Monitor, which interacts with the PTF and the tuning plugin. Dotted lines signify the components of the PTF, while the full lines depict the coordination layer.

The main interaction of the plugin and the coordination layer happens through the Monitoring Request Interface (MRI). This interface is used by PTF to retrieve the information and perform *selective* monitoring. In order to interface the pipeline coordination layer and PTF, corresponding MRI calls have been integrated in the coordination layer and a corresponding Standard Intermediate Representation (SIR) file is generated for PTF. In order to construct a variant space (all possible tuning scenarios), the plugin processes the SIR file during the initialization phase and collects information about tuning parameters and pipeline regions.

3.1.3 Tuning Plugin Developments

The plugin aims to tune an instrumented version of the code that utilizes the pipeline coordination layer.

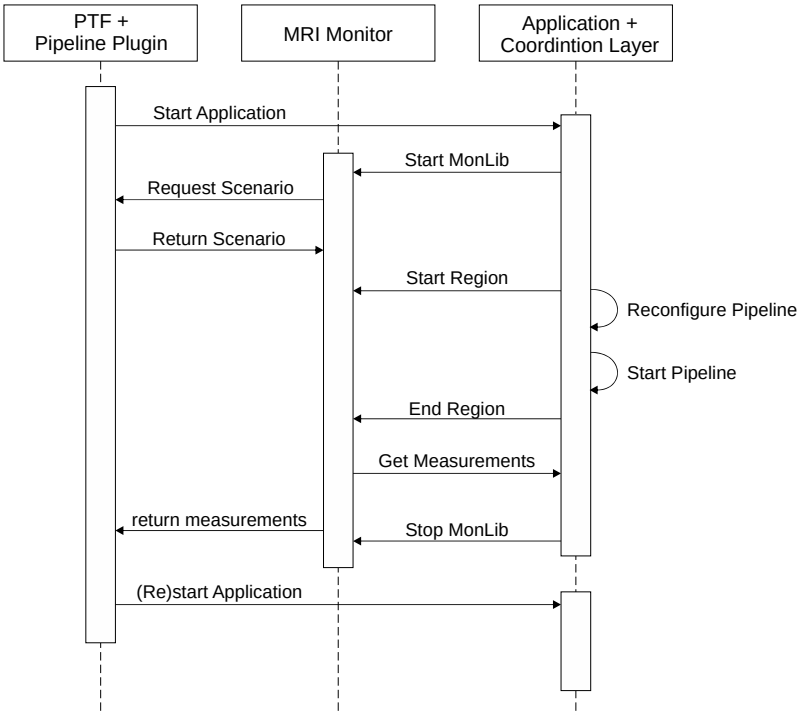

Figure 2: Conceptual diagram depicting the interaction of relevant parts of PTF with the coordination layer

The current prototype of the plugin operates in the following way. First, PTF (enabled with the Pipeline Plugin) starts an instrumented pipeline pattern parallel code that initializes the pipeline coordination layer and MRI monitoring library. The plugin reads the SIR file and processes the information on pipeline regions and tuning parameters. The example SIR file is shown in Listing 2.

```
. . .
  <codeRegion type="pipeline" name="" id="...">
    <position startLine="..." endLine="...">
      <file name="..."/>
    <position>
    <plugin pluginId="Pipeline">
      <selector tuningActionType="VAR" tuningActionName="<tuning
parameter variable name>" numberOfVariants="<value>"/>
    </plugin>
  </codeRegion>
. . .
```

Listing 2: SIR file example which shows elements specific to the pipeline plugin

After the SIR processing is done, the plugin uses the data to create the tuning scenarios. Each scenario represents a single configuration based on concrete values for tuning parameters.

```
. . .
startMonLib(); /*PERISCOPE: MRI start */

/* Calls to MRI variable map variable */
std::string srep = "<REPLICAS>";
std::string sncpus = "<NCPUS>";
...
psc_map_tp_var(&srep[0], &replicas, 0);
psc_map_tp_var(&sncpus[0], &ncpus, 0);
...
startRegion(42,1,54,0,-1);
...
  RuntimeConfiguration conf2(STARPU_RUNTIME, ncpus, ngpus, sshed);
  stagex->set_replication_factor(replicas);
  ...
  Manager.ReconfigureRuntime(&conf);

  /* start pipeline */
  manager.StartPipeline();
  manager.WaitForPipelineToFinish();
...
endRegion(42,1,54,0,-1);
...
stopMonLib();
. . .
```

Listing 3: Instrumented application that utilizes the pipeline coordination layer

Concrete values for the tuning scenarios are transferred to the coordination layer when the MRI startRegion() function is invoked (see Listing 3). Here, values for relevant variables are assigned (e.g: srep is a string variable mapped by the “psc_map_tp_var” and assigned when “startRegion()” is called). Afterwards, the coordination layer reconfigures the pipeline and the runtime with a new set of values for the tuning parameters and executes the pipeline. Finally, the endRegion() call will request the performance measurements from the coordination layer and report it to the plugin. Currently an exhaustive search to find the optimal execution time of the pipeline is employed.

3.1.3.1 Tuning parameters and tuning actions

In the first prototype of the plugin we introduced five tuning paramters and actions:

- The replication factor of individual stages (number of stage instances that can be run in parallel)
  - Tuning parameter: 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
- The size of the buffers to hold data packets passed between the stages
  - Tuning parameter: 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
- The number of CPU cores
  - Tuning parameter: Integer values in a predefined range used as a parameter for runtime configuration of number of CPU workers
  - Tuning action: MRI variable tuning action assigning the value to the runtime configuration parameter
- The number of GPUs
  - Tuning parameter: Integer values in a predefined range used as a parameter for runtime configuration of number of GPU workers
  - Tuning action: MRI variable tuning action assigning the value to the runtime configuration parameter
- The scheduling policy used by the underlying runtime system (StarPU)
  - Tuning parameter: Integer values in a predefined range used as a parameter for runtime configuration of number of CPU workers
  - Tuning action: MRI variable tuning action assigning the value to the runtime configuration parameter

3.1.4 Prototype Integration & Validation

The tuning plugin has been integrated in the Periscope Tuning Framework. To this purpose we needed to expose relevant metrics and define relevant performance properties. Currently we use pipeline execution time metric to define a new PIPEEXECTIME performance property.

In order to provide measurements to the plugin, the pipeline coordination layer is extended. The *PipelineManager* class now features methods for starting and stopping the execution of the pipeline, as well as methods for providing measurement data upon the request from MRI monitor.

3.2 Energy Consumption via CPU Frequency Tuning plugin

3.2.1 Tuning Prototype Overview

The Energy Tuning plugin has the goal to reduce the energy consumption of an application by selecting the right CPU frequency and governor setting that leads to minimal energy consumption for the application execution. The plugin performs an exhaustive search through the available setting of those two tuning parameters. The energy measurements and the tuning actions required to set the tuning parameters at runtime to the appropriate values are implemented in the enopt library, developed at the LRZ-BAdW.

3.2.2 Tuning Prototype Design

3.2.2.1 Tuning parameters and tuning actions

Once an application has been tuned for performance, it is then eligible for optimization of energy- and time-related costs. For the Energy Tuning Plugin we have defined two different tuning parameters: the available governors; and the frequencies to be used. The governors tuning parameter supports five governors or policies, namely "performance", "powersave", "user space", "conservative", and "ondemand". More details are given below. The frequency tuning parameter supports the frequencies available on the Sandy Bridge processors of SuperMUC, i.e., 2.7, 2.6, 2.4, 2.2, 2.0, 1.8, 1.6, 1.4, and 1.2 GHz.

3.2.2.2 Enopt Library

The enopt library provides the energy measurements and the tuning actions required by the tuning plugin. The core of the library was developed in the first year of the project; in this second year we extended the functionality and, due to several critical changes done on the kernel of the Linux operating system, we developed additional methods for the access control to the kernel space devices and infrastructures.

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, such as MPI, OpenMP, hybrid MPI-OpenMP and sequential. Depending on the model used, the application has a different topology. This topology 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. For this issue, the computing model layer is in charge of.
- Counter and CPUFreq layer: This layer is responsible of fulfilling the aim of the enopt library and therefore the ones of the energy plugin of the project. This layer is the one that performs the hardware counter measurements and changes the clock frequency and policies of the CPUs, through the CPUFreq [3] kernel infrastructure.

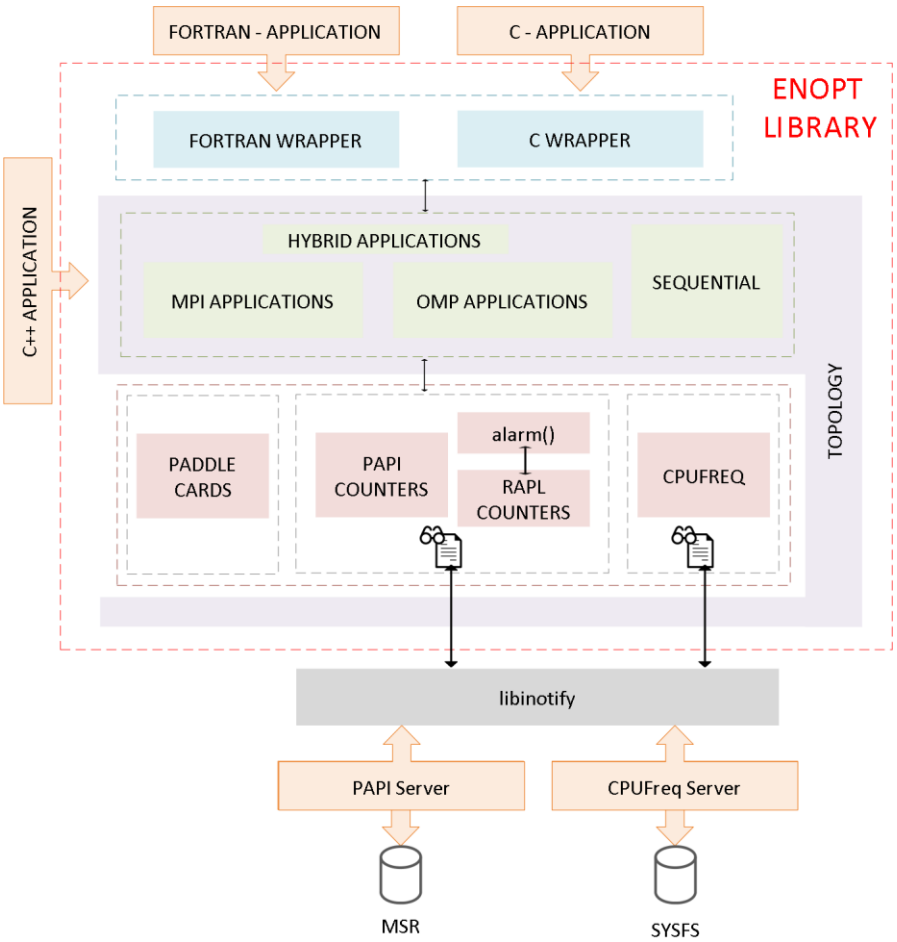

Figure 3: Energy Consumption Plugin, Architecture Overview

The library needs to perform operations over several files belonging to the superuser. These actions cannot be directly executed since unprivileged processes have no permissions to access to 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.

UNIX and Linux implementations distinguish two categories of processes, for the purpose of performing permission checks:

- "privileged processes", whose effective user ID is 0, referred to as superuser or root.
- "unprivileged processes", whose effective user ID is nonzero, ordinary user.

Privileged processes bypass all kernel permission checks, while unprivileged processes are subject to full permission checking based on the credentials of the process, such as effective user ID (EUID), effective GID (EGID) or other supplementary access group lists.

In order to change the CPU frequency and governor of a certain machine, write permissions on files which belong to the superuser are needed. As it has been explained before, by default, user space applications, like for example Periscope, cannot access those devices, hardware, or memory belonging to the root user. For this reason, a privileged application belonging to the superuser is needed, which will change the frequency and governor. These privileged applications are known as "Daemon application" or simply "daemon".

On the other hand, doing performance measurement implies accessing the hardware counters. These typically belong to the root user and therefore also need a daemon for this task. This daemon is triggered by the user application through the enopt library, accesses the hardware counters, and returns the measurements back to the user application.

3.2.3 Tuning Plugin Developments

3.2.3.1 Daemons development

Two external daemons were developed. Both communicate with the MRI monitor through the enopt library:

- The CPUFreq daemon

  Some time ago the cpufreq infrastructure brought a daemon, which allowed users to work with the parameters of the governors from a userspace application. At the time of writing this document, this daemon is no longer provided by the cpufreq packages, nor maintained[1]. Therefore, we were forced to develop our own daemon for changing CPU frequencies and governors in real time for the Autotune project.

  There are two different kinds of governors: the static and the dynamic ones. The static ones are those which communicate with the kernel through files (known as sysfs entries) that are created just once at boot time. Inside this group we have the "performance", the "powersave" and the "userspace" governors. On the other hand, the sysfs entries of the dynamic governors are recreated with superuser permissions each time that the specific governor is triggered. To this group of governors belong the "conservative" and "ondemand" governors. This issue of the dynamic governors prohibits user space applications, running in the user space permission level, to modify the parameters of the “ondemand” and “conservative” governors thus blocking applications and libraries to change the configuration parameters of the dynamic governors.

  The developed CPUFreq daemon provides a kernel based solution that solves the problem of modifying parameters of the dynamic governors' entries allowing user space applications running with the corresponding permissions to modify in real time the cpufreq filesystem parameters.

  When PTF changes the frequency or the governor of a CPU, it starts a request within the enopt library writing the identifier of that CPU in a certain communicator file. This action triggers the daemon which will read the communicator file to get the identifier of the CPU whose parameter will be changed. The daemon changes then the ownership of the sysfs entry that corresponds to the CPU finishing the request. Once the communicator file has been closed, the user space application has full permissions to change either the governor or the frequency to the desired value.

- The PAPIServer daemon

  Until March 2013, operations involving performance counters needed special high permissions level related to the EGID, i.e. users who belonged to a certain user group could perform operations related to hardware counters. Since March 2013, due to a security vulnerability in the

[1] http://www.linux.it/~malattia/wiki/index.php/Cpufreqd

Linux kernel (CVE-2013-2094)[2] which allowed local users to scale their privileges and gain root access, a more restricted access to the performance counters was set, owned in most of the cases only by the root user. This forced us again to use a daemon which is launched as root user and can perform measurements of the hardware counters. As in the CPUFreq daemon, the user space application can get the energy performance values by sending special requests to the daemon. This action is made by the enopt library. The communication between the user space application and the daemon is based on a message queue interprocess communicator which is created by the enopt library at the initialization point of PTF's MRI monitor.

3.2.3.2 Library development

The basic functionalities of the enopt library were developed for fulfilling the needs of the first year of the project and it has been extended for adding more features needed for an advanced behavior of the plugin. The library now provides methods to measure not only the energy consumption of the cores, but also provides the node level measurements based on hardware in the power supplies. These measurements are carried out through special cards called paddle cards. The access to the paddle cards is provided by a specific kernel module from IBM (ibmaem) available in the kernel of each node of the SuperMUC machine.

The enopt library supports FORTRAN or C/C++ applications parallelized either with MPI, OpenMP, or both (hybrid parallelization). At the moment, it is available for Sandy Bridge processors. The library provides classes to monitor energy counters as well as other PAPI performance counters in order to be able to find correlations between energy consumption, performance counters (such as cache misses, number of cycles, instructions per second, etc.), and the application runtime.

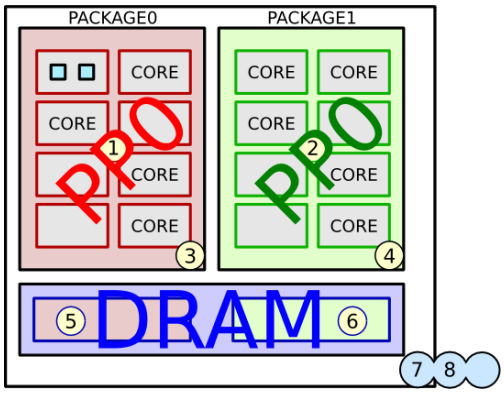

Figure 4: Graphical location of the RAPL counters on the SandyBridge micro-architecture. Blocks with red background belong to the PACKAGE0, whereas the green ones belong to PACKAGE1. The violet block represents the DRAM device. The yellow circles represent the RAPL sensors whereas the blue one represents the ***ibmaem*** counters.

For achieving the objective of the plugin, the PAPI-RAPL component [4] and the native ibmaem kernel module have been integrated in the enopt library (Figure 7):

[2] http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2094

- PAPI Component

  PAPI aims to provide the tool designer and application engineer with a consistent interface and methodology for use of the performance counter found in most microprocessors. PAPI enables software engineers to analyze the relationship between software performance and processor events. One of the components of the PAPI library is the PAPI-RAPL (Running Average Power Limit) component that makes use of the RAPL sensors available in the SandyBridge microarchitectures.

  The PAPI-RAPL component provides energy consumption measurements of the CPU-level components by examining the MSR registers. The specific RAPL domain counters available in the Intel platforms vary across "product segments". Platforms targeting the server segment, such as for example the SuperMUC system, support:

  - Package (PKG)
  - The power plane PP0 that refers to the processor cores
  - DRAM

  PAPI-RAPL provides a set of PAPI native events to interact with the RAPL interface. These events are, among others:

  - PACKAGE_ENERGY:PACKAGEx: Energy used by chip package 0 or 1, respectively
  - DRAM_ENERGY:PACKAGEx: Energy used by the DRAM on package 0 or 1, respectively
  - PP0_ENERGY:PACKAGEx: Energy used by all cores in package 0 or 1, respectively

- Power supply paddle cards

  The IBM paddle cards are hardware devices located on the motherboard for measuring the AC and DC power consumption at board level. For obtaining these energy measurements, the ibmaem together with the HWMON[3] kernel module are needed. They provide two different devices for querying the energy consumption of the system node. In our case, every two neighboring nodes share a power supply paddle card; the AC energy value comes from the AC site of the power supply and measures the energy consumption of two adjacent nodes. In the same way, the DC value is measured at the power supply, but it is unique for each node.

  The DC and AC counters provided by the ibmaem kernel module, whose measurements are obtained from the power supply paddle card, are updated three times per second and those temporal values are stored and finally summed up by the kernel infrastructure.

3.2.3.3 Tuning Plugin Interface Integration

Following the design explained on the last year deliverable D4.1, the energy consumption plugin has been integrated with the periscope tool by using the following methods:

[3] https://www.kernel.org/doc/Documentation/hwmon/sysfsinterfaces

- Initialize: creates the tuning parameters (governors and frequencies) from a configuration file.
- Create Scenarios: the variant space has been defined as the cross product of governors and frequencies.
- Prepare Scenarios: since no recompilation is needed, the created scenarios are moved directly to the PSP.
- Define Experiment: the experiments are initially executed to request the objective energy and execution time per node. Each experiment must initialize the enopt library to initialize the above mentioned daemons and the internal structures needed for carrying out the aim of the plugin.
- 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.
- Process Results: the best energy savings for the phase region with its respective governor or frequency settings is selected and provided to the user as a recommendation.

3.2.4 Prototype Integration & Validation

In the case of the energy plugin not only the prototype must be integrated and validated, but also the enopt library and the developed daemons. This allows user space applications, instrumented with PTF to communicate and request information either to or from these daemons.

3.2.4.1 Daemons integration

Both developed daemons, the cpufreq daemon and the PAPI daemon, have been integrated with the enopt library. Once the user space application is linked against the enopt library, they will have full access either to the cpufreq sysfs entries or to the hardware performance counters through the previously described daemons.

Both daemons communicate with the user space application (or viceversa) through special files created by the user space application. Writing on those files triggers in real time the corresponding daemon. By reading this file, the daemon will perform the requested task of the application. Polling the local filesystem to detect when a certain file has been accessed can be really time consuming. To avoid polling from the application and for detecting when these special files are accessed, we use a library, inotify[4], which is used by the Linux kernel and which permits to set watchers to either a directory or a file. Once the watched files are accessed, either for an open, read, modify or write operation, a configurable action is automatically performed. In case of the CPU frequency and governor daemon, the action is to perform granting permissions on the corresponding sysfs entry, while in the case of the PAPI daemon, the action is to stop and start the requested hardware counters (in this case, the energy hardware counters provided by the PAPI-RAPL infrastructure).

3.2.4.2 Prototype validation

Since we are performing an exhaustive search, the validation of the plugin is simple: we only need to check that we produce the rights measurements within the library.

[4] http://man7.org/linux/man-pages/man7/inotify.7.html

To validate the results obtained by the enopt library, we have compared those results with the ones obtained using four different tools. The tools used are two MSR (Model Specific Registers)-based tools, PAPI with the RAPL component and the LIKWID [5] tool suite. LIKWID provides a set of easy to use command line utilities to support application optimization. We use two measuring devices, the IBM paddle cards (accessed by the ibmaem and the HWMON kernel modules[5]) that measure incoming AC and the corresponding DC output at the power supply level, and secondly, the measurements carried out directly in the PDU (Power Device Unit). A description of the characteristics of the four used applications and their measurement resolution can be seen in tables 1 and 2.

Tool	DRAM	SOCKET	NODE	RACK
LIKWID	X	X		
PAPI-RAPL	X	X		
Paddle Card			X	
PDU				X

Table 1. Comparison of domains that can be measured with each of the tools used.

Tool	Technology	Resolution	Domains
LIKWID	MSR	1MHz	Socket, DRAM
PAPI-RAPL	MSR	1MHz	Socket, DRAM
Paddle Card	Ibmaem	3Hz	Node
PDU	Power meter	0.17Hz	Rack

Table 2: Comparison of the theoretical features of the four tools and technologies used to measure the energy consumption of applications on SuperMUC. The four tools used are not able to measure all the available domains.

Up to now, the MSR-based tools are only capable of measuring short code paths due to a limited register size and the resulting cyclical overflows. For comparison purposes, the PAPI-RAPL tool and the ibmaem have been integrated into a library that overcomes these overflows by saving intermediate values. In order to build comparable measurements among the non-MSR based tools, long running experiments were executed and the measurements appropriately aggregated.

The validation of the enopt library consists of two steps. On one hand, the results of the library need to be compared to a physically different energy measurement method; on the other hand, library implementation errors can be found by comparing the results with the output of other tools using the same measurement equipment. Thus, we decided to compare the results obtained by the enopt library with the measurements of the PDUs of SuperMUC and also with the results of the LIKWID tool:

[5] https://www.kernel.org/doc/Documentation/hwmon/sysfsinterfaces

- PDU

  A higher level of power measurement instrumentation in SuperMUC is the use of "smart" PDUs. The PDU is a device fitted with multiple outputs designed to distribute electric power, especially to racks of computers and networking equipment. SuperMUC uses IBM 46M4004 PDUs for powering its compute nodes and switches each featuring 12 IEC-320-C13 outlets [6]. Each pair of outlets within the PDUs features a built-in power-meter circuitry capable of integrating the energy consumed or reporting instantaneous power. All PDUs in SuperMUC are connected to an Ethernet based service network that can be used for switching individual PDU ports and for retrieving the power consumption. A management node queries the PDUs once per minute. As the performance of the Ethernet based service network across the entire machine is limited, up to 8 nodes share one measurement and measuring at higher frequencies is not feasible. Yet, PDU based measurements provide important insight into the total power consumption of a node including the losses incurring during the power conversion step in the power supply unit.

- LIKWID

  LIKWID is a tool suite that provides a set of easy to use command line utilities to support application optimization. It is targeted towards performance-oriented programming in a Linux environment, it does not require any kernel patching and it is suitable for Intel and AMD processor architectures. Multi-threaded and even hybrid shared/distributed-memory parallel codes are supported. One of the command line utilities is the so called likwid-powermeter which is a tool that allows querying the energy consumed within a CPU package for a given time. Using the elapsed runtime it also computes the resulting power consumption. Additionally, the supported Turbo Mode steps of all Turbo mode equipped processors (except the EX variants) can be also queried. In order to obtain this information, LIKWID reads the MSR registers. Another way of retrieving the energy consumption of an application is accessing the RAPL counters using the likwid-perfctr command line.

For exhaustive information regarding the validation results, refer please to document D5.3 "Report on the test results of the integrated prototype".

3.2.5 Use Case example

We used a modified version of the APEX-MAP benchmark [7] that generates artificial calculations and memory accesses for measurements. We linked this testbench against the above explained enopt library. The initial idea of the Apex project is the assumption that the performance behavior of any scientific application can be modeled by a set of specific performance factors. The two most dominant factors are the memory accesses and computational density. By simulating memory bound or compute bound codes, APEX-MAP avoids a hardware specific model and can be used to simulate a typical scientific application.

Experiments carried out within our test case demonstrate that it takes a reasonable time for going through all the available tuning parameters and thus, the measurements have been taken on all the available tuning parameters until we find the minimum of the selected strategy (see the report D5.3 for more information). However, for other applications it can happen that exploring the complete set of scenarios will take so much time, that it is no longer feasible to try all the combinations. In such a case we will provide an energy estimation model based on frequency and hardware counters. This model will enable us with the next plugin version to reduce the search space.

3.3 Master-Worker MPI Plugin

It is the goal of the Master-Worker MPI Plugin to tune master-worker type of application with respect to the size of the tasks as well as with respect to the number of workers executing those tasks.

3.3.1 Tuning Prototype Overview

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

The first tuning parameter determines the size of batches of tasks assigned to the workers. Instead of distributing the whole set of tasks among workers and then waiting for the results, the master partially distributes the tasks by dividing the 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 that worker 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 since it can be efficiently used to improve the performance of the application.

In the real world scenario, 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.

Due to changes in PTF and the evolution of the plugin some of its characteristics have changed, the fact that custom properties can now be requested for a specific experiment allows for a more flexible analysis. In addition, there have been some complications with the variation of the number of workers because changing the number of workers requires to adapt the number of agents controlling the application processes.

This lead to the decision of making PTF and the plugins support both instrumented and non-instrumented applications. Non-instrumented applications are normally compiled and therefore do not have no calls to the MRI monitor library inside. Non-instrumented applications cannot be controlled by Periscope, stopped and resumed at will, or tuned dynamically but they can be started faster and don't need the agents to be deployed. The fact that the agents are not deployed allows the plugin to restart the application with a different number of processes.

The complete functionality of this plugin requires that the application is instrumented, indicating the variables that will be modified during the experiments. An example of such variable is the variable representing the partition factor mentioned above, which controls the partitioning of the data in chunks before distributing it among the workers. Due to limitations of the PTF instrumenter, the code has been instrumented manually to include the MRI calls that allow the Periscope tool to do the modification of the variable needed.

3.3.2 Tuning Plugin Developments

The Master-Worker plugin has been developed to support both instrumented and non-instrumented applications.

In the case of non-instrumented applications, the partition factor cannot be changed using a tuned variable (it could still be changed by allowing its modification through an input argument, but this solution would require a restart procedure). On the other hand, the number of workers variable is much easier to modify since no hierarchy of agents is necessary.

The prototype, working with instrumented applications, is able to dynamically modify the values of the partition factor and measure the execution time of each experiment.

3.3.2.1 Tuning parameters and tuning actions

The tuning parameters of this plugin are the two following variables:

- The partition factor and
- The number of workers.

The tuning action is to change the value of these two variables and perform a new experiment, i.e. run a small number of iterations of the application. Both tuning actions are performed in the Master process 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.

3.3.2.2 Tuning Plugin Interface Integration

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 MPI application pattern and the tuning parameters that correspond to the variables in the code, for example the number of workers to run. Moreover, certain tuning parameters must be annotated using Periscope directives in the source code, such as 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. The instrumentation step can also be done manually, inserting the MRI directives in the code, in this case, in addition to the regions, the partition factor variable has to be annotated in the code.

Initialize

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

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 parameter:

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. However, as stated before, the prototype can only tune one of these parameters at a time, depending on whether the application is instrumented or not.

Prepare Scenarios

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

Define Experiment

In this function we set the required properties and return the request so the frontend knows which properties are to measure. This is only done in the first experiment, and the properties retrieved will be used to set the values on the next experiment.

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

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. This is done in the second phase explained above, and requires a reorganization of the agent hierarchy in the case of instrumented application tuning.

Process Results

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

3.3.3 Prototype Integration & Validation

The prototype Master-Worker plugin integrates with PTF using its interface as described above. However, for the instrumented version, the exploration of the number of workers tuning parameters was not developed. However, the request of custom properties is not yet used in the prototype so it performs a blind search on a range of user defined values.

The basic functionalities of the plugin have been successfully integrated. The plugin is now capable of trying different partition factors automatically and of getting time metrics from those experiments. Moreover, the number of workers can be automatically changed in the non-instrumented mode of PTF.

3.3.4 Use Case example

The prototype was tested with an application counting the number of primes in a list of integers. This application iteratively serves chunks of a vector of integers to a group of workers which sift through them to count the number of primes. A synthetic workload was generated with a notable unbalance which is used to assess the plugin functionality.

3.4 MPI Runtime Plugin

The MPI Runtime Plugin optimizes the execution of MPI application based on adapting MPI library parameters during the startup of the application. The settings are passed to the library via MPI environment variables.

3.4.1 Tuning Prototype Overview

This plugin uses the standard MPI analysis of Periscope to implement a tuning strategy to determine the values of tuning parameters that represent a set of MPI environment variable. The plugin is able to tune any of the available parameters for a set of user defined ranges.

In the case of IBM MPI 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 that the application processes behave in the same way (SPMD programs) and applies the same optimization to all processes. It takes into consideration two sets of variables:

- MP_BUFFER_MEM: amount of memory for buffering data from early arrival messages
- MP_EAGER_LIMIT: threshold value of the message size for changing from the eager to the rendezvous protocol

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. 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.

The initial idea was to include the possibility using tuning parameters to select code variants, that is regions of code where several functions can be used alternatively, however due to time limitations this approach has not yet been implemented.

3.4.2 Tuning Plugin Developments

This plugin also supports both instrumented and non-instrumented applications. However, using instrumented applications produces more accurate time measurements and allows for a deeper analysis of the impact of different parameter configurations, which could be useful in the future to implement intelligent search algorithms.

3.4.2.1 Tuning parameters and tuning actions

The MPI environment parameters may be set before the application executes. Two tuning parameters are proposed, that correspond to the two aforementioned IBM MPI configuration parameters:

- 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.

These variables may be set by the environment variables or by the mpirun options (flags). 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.

3.4.2.2 Tuning Plugin Interface Integration

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, in addition we can specify which MPI implementation is being used so the plugin will configure it accordingly. Finally, the instrumentation is done and the SIR file is generated.

Initialize

The tuning parameters are extracted from a configuration file containing the configurable options of the MPI library and a range of values valid to each of them. Each option becomes a tuning parameter and has different possible values. Depending on the type of the configuration option, the valid values for each tuning parameter may vary; some configuration options require a Boolean value while others need a string of characters or a range of integers. The tuning space is a two-dimensional space generated by the multiple values of each MPI option. 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.

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 parameters. 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.

Prepare Scenarios

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

Define Experiment

If necessary an analysis step is requested in this function to obtain detailed information on the impact of the configuration. This could be useful in the future for search algorithms that can take into account low level information on the application performance.

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.

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. The information provided by the plugin in this stage is the new command-line string to re-launch the application with the new configuration.

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.

3.4.3 Prototype Integration & Validation

The integration with PTF is achieved by developing the interface functions mentioned above. The prototype has been developed by implementing each of those functions and has been validated with unit and regression tests.

A real application was used in the validation process. FSSIM, one of the proposed application in the beginning of the project was selected to validate the behavior of the plugin. Several experiments were performed manually to test different scenarios in the target system SuperMUC. In addition some automatic tests were developed.

This automatic tests consists on a unit test suite and a regression test that will allow a continuous integration of the plugin. The regression test uses the same application as in the validation process to test the plugin.

3.4.4 Use Case example

As an example execution we performed an exploration of a simple search space with only one parameter. FSSIM was used to test the selection of the eager limit parameter in the SuperMUC system. The goal for the use case example is to test that the MPI library is properly configured and that the impact on the application can be measured with Periscope.

From previous experiments we know the ranges of the eager limit value that perform better in this application, so we used that information to build a test that launches several experiments with different values and chooses the best one. This result is checked against the reference value we got from the manual tuning.

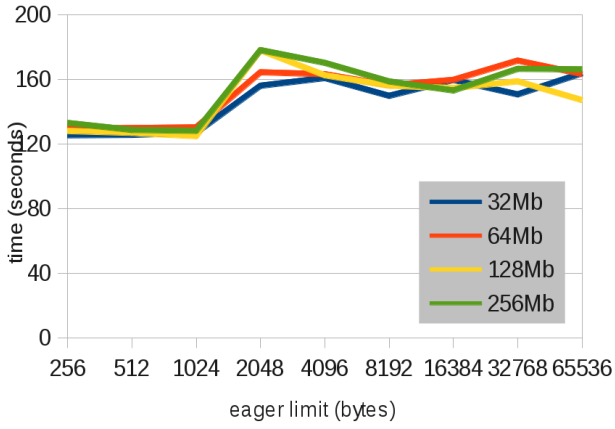

Figure 5: Eager limit and buffer size parameters in FSSIM

Figure 5 shows the impact that the eager limit and the buffer size parameters have in the performance of the FSSIM application. Based on this data we tested the tuning prototype using the eager limit for values in the ranges where the performance drop happens.

The validity of the output of the application is also tested. FSSIM was modified so its output is deterministic for a given input so we can use a reference output to check whether the tuning interfered with the computations of the simulator.

3.5 Compiler Flag Selection Plugin

3.5.1 Tuning Prototype Overview

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 the machine code from the high-level source code. Compilers are applying a large number program transformation to generate the best code for a given processor, 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, the 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 switch on a default optimization approach in the compiler.

3.5.2 Tuning Plugin Developments

3.5.2.1 Tuning parameters and tuning actions

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

3.5.2.2 Tuning Plugin Interface Integration

The tuning parameters for a given compiler are initialized from a configuration file. The configuration file will specify which flags can be used and in which manner. It can also specify fixed combinations of flags that can be applied by the plugin. In addition, a mapping of single node performance properties to compiler flags that might help to improve the performance problem can be specified.

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

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

Depending on some parameters in the configuration file, for each scenario the entire code will be recompiled or only the relevant portion will be compiled if the selective compilation is selected. The purpose of the selective compilation is to lower the total time needed for recompilation by only recompiling the most relevant files. This selection of files is done after a profiling step that identifies the most relevant functions in terms or execution time; and only marks their files for recompilation.

The selected flags will be enforced in the compilation step by modification of the compiler flags variable that is specified in the configuration file.

Between every recompilation, the plugin requests an application restart without any additional parameters.

At the end the plugin will retrieve the best variant from the search algorithm and return this combination as a tuning recommendation to the user.

3.5.3 Prototype Integration & Validation

The plugin integrates perfectly with PTF. Several tests and regression tests were performed to validate the results.

3.5.4 Use Case example

This plugin was successfully tested with NPB-SER using different compiler flags as well as a full and a selective compilation. The plugin could function with both the instrumented and the non-instrumented versions of NPB.

4 Future works

This document described the important work made for the implementation of a new set of auto-tuning plugins inside the Periscope Tuning Framework. The document provides a good overview of the global structure of all the plugins and opened some new improvement opportunities.

The first objective in the next period is to finalize the development of all plugins. The integration into the full infrastructure requires more work and experimentations using the application repository.

After a small period of consolidation of the framework plugins, we will get a better understanding of the behavior and the pertinence of each plugin on its area. From this experience, we now expect to create some new collaboration in the project for the design and the development of new plugins or multi-aspect auto-tuning plugins.

Share plugin performance analysis technologies

Several plugins have designed new analysis strategies and performance properties. The new analyses, e.g., the energy efficiency analysis for example, might be useful for other plugins as well. We will investigate possibilities to use those to enhance the plugin tuning strategy.

Developing new plugins

We could demonstrate the flexibility of the framework by the variety of plugins developed in this project. The demo plugin developed last year has significantly improved the development speed of all other plugins. From the process, we realized that a simplified and parameterized tuning plugin would be useful for the quick development of basic tuning plugins. Well documented, it could be a perfect gateway for developers willing to develop quickly a plugin.

Mixing plugins

The plugin development has shown the emergence of two kinds of plugins: plugins requiring the instrumentation of application and applied to some regions in the code; and plugins performing application wide performance tuning. It appears that the combination of a plugin from each kind is by construction simplified for some aspects: the compilation chain and the management of tuning parameters. For example the "Compiler Flag Selection Plugin" could be combined with the "MPI Parameter Plugin". We plan to investigate the implementation of a multi-aspect plugins using this approach based on the combinations of two plugins.

5 Bibliography

[1] S. Benkner, S. Pllana, J. L. T. ff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney and V. Osipov, "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems," *IEEE Micro,* vol. 31, no. 5, p. 28–41, 2011.

[2] C. Augonnet, S. Thibault, R. Namyst and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience,* vol. 23, no. Special Issue: EuroPar, pp. 187-198, 2009.

[3] I. Corporation, "The CPUFreq governors," http://publib.boulder.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=/liaai/cpufreq%2FTheCPUFreqGovernors.htm.

[4] M. J. K. K. J. R. P. L. D. T. a. S. M. V. Weaver, "Measuring Energy and Power with PAPI," *The 1st International Workshop on Power-Aware Systems and Architectures,* 2012.

[5] H. G. W. G. Treibig J, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops,* pp. 207-216, 2010.

[6] I. Corporation, "New IBM Switched and Monitored family of Power Distribution Units makes it easy to protect and manage high-availability rack based systems.," 2010.

[7] S. H. Strohmaier E, "Architecture independent performance characterization and benchmarking for scientific applications," *Proceedings of the "The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems",* pp. 467-474, 2004.

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

[12] 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.

[13] 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].