AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications

Renato Miceli[1], Gilles Civario[1], Anna Sikora[2], Eduardo César[2], Michael Gerndt[3], Houssam Haitof[3], Carmen Navarrete[4], Siegfried Benkner[5], Martin Sandrieser[5], Laurent Morin[6], and François Bodin[6]

[1] Irish Centre for High-End Computing, Trinity Tech & Ent Campus, Dublin, Ireland
{renato.miceli,gilles.civario}@ichec.ie
[2] Universitat Autònoma de Barcelona, CAOS Department, Barcelona, Spain
ania@caos.uab.cat, eduardo.cesar@uab.es
[3] Technische Universität München, Institut für Informatik, Garching, Germany
{gerndt,haitof}@in.tum.de
[4] Leibniz-Rechenzentrum, The Bavarian Academy of Sciences, Garching, Germany
carmen.navarrete@lrz.de
[5] University of Vienna, Faculty of Computer Science, Wien, Austria
{sigi,ms}@par.univie.ac.at
[6] CAPS Entreprise, Immeuble CAP Nord, 4 Allée Marie Berhaut, Rennes, France
{laurent.morin,francois.bodin}@caps-entreprise.com

Abstract. Performance analysis and tuning is an important step in programming multicore- and manycore-based parallel architectures. While there are several tools to help developers analyze application performance, no tool provides recommendations about how to tune the code. The AutoTune project is extending Periscope, an automatic distributed performance analysis tool developed by Technische Universität München, with plugins for performance and energy efficiency tuning. The resulting Periscope Tuning Framework will be able to tune serial and parallel codes for multicore and manycore architectures and return tuning recommendations that can be integrated into the production version of the code. The whole tuning process – both performance analysis and tuning – will be performed automatically during a single run of the application.

1 Introduction

The pervasiveness of multi- and many-core processors nowadays makes any computer a parallel system. Most embedded devices, desktop machines, servers and HPC systems now include multicore processors coupled to accelerators (GPGPUs, FPGAs). This recent shift to multi- and many-core architectures hinders the development of hardware-optimized applications. Programming parallel architectures requires careful co-optimization of the following interrelated aspects:

– Energy consumption: Energy reduction has become a major issue on HPC architectures, given their power costs almost reach the purchase price over a lifetime. Careful application-specific tuning can help reduce energy consumption without sacrificing an application's performance.

- Inter-process communication: The overall scalability of parallel applications is significantly influenced by the amount and the speed of communication required. Reducing the communication volume and exploiting the physical network topology can lead to great performance boosts.
- Load balancing: Implicit/explicit process synchronization and uneven distribution of work may leave a process idle waiting for others to finish. The computing power within all parallel processes must be exploited to the fullest, otherwise program scalability may be limited.
- Data locality: Frequent accesses to shared or distant data creates a considerable overhead. Reducing the contention for synchronization resources and ensuring data locality can yield significant performance improvements.
- Memory access: Even the best arithmetically-optimized codes can stall a processor core due to latency in memory access. Careful optimization of memory access patterns can make the most of CPU caches and memory bandwidth on GPGPUs.
- Single core performance: To achieve good overall performance each core's compute capabilities need to be optimally exploited. By providing access to the implementation details of a targeted platform, application optimizations can be specialized accordingly.

Application tuning addressing these areas is an important step in program development. Developers analyze an application's performance to identify code regions that can be improved. They then perform different code transformations and experiment with setting parameters of the execution environment in order to find better solutions. This search is guided by experience and by the output of performance analyses. After the best set of optimizations is found for the given – and supposedly representative – input data set, a verification step with other input data sets and process configurations is executed.

The research community and vendors of parallel architectures developed a number of performance analysis tools to support and partially automate the first step of tuning process, i.e. application analysis. However, none of the current tools supports the developer in the subsequent step of tuning, i.e. application optimization. The most sophisticated tools can automatically identify the root cause of a performance bottleneck but do not provide developers with hints about how to tune the code.

Therefore, the AutoTune project's goal is to close the gap in the tuning process and simplify the development of efficient parallel programs on a wide range of architectures. To achieve this objective, we aim to develop the Periscope Tuning Framework (PTF), the first framework to combine and automate both analysis and optimization into a single tool. AutoTune's PTF will:

- Identify tuning variants based on codified expert knowledge;
- Evaluate the variants online (i.e. within the execution of the same application), reducing the overall search time for a tuned version; and
- Produce a report on how to improve the code, which can be manually or automatically applied.

This project focuses on automatic tuning for multicore- and manycore-based parallel systems, ranging from parallel desktop systems with accelerators, to petascale and future exascale HPC architectures. The next sections show how AutoTune addresses the automatic analysis and tuning of parallel applications.

2 Related Work

The complexity of today's parallel architectures has a significant impact on application performance. In order to avoid wasting energy and money due to low utilization of processors, developers have been investing significant time in tuning their codes. However, tuning implies searching for the best combination of code transformations and parameter settings of the execution environment, which can be fairly complicated. Thus, much research has been dedicated to the areas of performance analysis and auto-tuning.

The explored techniques can be grouped into the following categories:

– Self-tuning libraries for linear algebra and signal processing like ATLAS, FFTW, OSKI and SPIRAL [3,4,5,6];
– Tools that automatically analyze alternative compiler optimizations and search for their optimal combination [7,8,9,10,11];
– Auto-tuners that search a space of application-level parameters that are believed to impact the performance of an application [12,13]; and
– Frameworks that try to combine ideas from all the other groups [14,15].

Performance analysis and tuning are currently supported via separate tools. AutoTune aims to bridge this gap and integrate support for both steps in a single tuning framework.

3 Approach: The Periscope Tuning Framework (PTF)

AutoTune is developing the Periscope Tuning Framework (PTF) as an extension to Periscope [2]. It follows Periscope's main principles, i.e. the use of formalized expert knowledge and strategies, automatic execution, online search based on program phases, and distributed processing. Periscope is being extended by a number of tuning plugins, each of which performs the tuning according to a certain code aspect [1].

PTF's tuning process begins by pre-processing the source files. It takes codes written in C/C++ or Fortran using MPI and/or OpenMP and performs instrumentation and static analysis to generate a SIR file (Standard Intermediate Representation). Extensions to support HMPP and OpenCL codes and parallel patterns are under development. The tuning is then started via the frontend either interactively or in a batch job.

Periscope's analysis strategy becomes part of a higher-level tuning strategy, which controls PTF's sequence of analysis and optimization steps. The analysis guides the selection of a tuning plugin and the actions it performs. After the plugin execution ends, the tuning strategy may restart the same or another analysis strategy to continue with further tuning. All plugins – each with its own tuning strategy, which may employ expert knowledge or machine learning – combine with Periscope's analysis strategies to perform a multi-aspect application tuning.

A tuning plugin is controlled by a plugin strategy meant to guide the search for a tuned version. This strategy often performs several iterations of selection and transformation of tuning parameters and experimental evaluation. Plugins try to tune the code by changing the values of certain tuning parameters, which are the features that influence the performance of a code region.

Code regions are often influenced not by one but by several tuning parameters – the tuning space. The plugin experimentally assesses a code region for a code variant, which is the set of values assigned to region's tuning space. Before experimenting, however, the plugin restricts the search space based on the output both of the previous analyses and of each plugin executed. The reduced search space, called variant space, defines the remaining code variants to be evaluated experimentally.

In the tuning process, the plugin's search strategy explores the variant space to optimize certain tuning objectives. A tuning objective is a function that takes a code region and a variant and outputs a real value, generally a performance measurement (e.g. runtime, energy consumed). The plugin executes a tuning scenario, evaluating one or more tuning objectives for a specific code region and variant.

Once the tuning process is finished, PTF generates a tuning report to document the recommended tuning actions. These tuning actions, i.e. the changes performed to tune the code, can be integrated either manually or automatically into the application for production runs.

The concrete output of the AutoTune project is the Periscope Tuning Framework and its tuning plugins for:

- High-level parallel patterns for GPGPUs;
- Hybrid manycore HMPP codelets;
- Energy consumption via CPU frequency;
- MPI runtime; and
- Compiler flag selection.

The framework follows an extensible architecture so that additional plugins can expand its functionalities. The effectiveness, efficiency and software quality of PTF will be demonstrated with real-world multi- and many-core applications.

3.1 General Design of a Tuning Plugin

PTF's high-level flow is controlled by the frontend; for auto-tuning, a predefined sequence of operations is enforced. However, the frontend allows loadable plugins to specify what is done in certain steps of the execution and to determine how many iterations are required for certain loops.

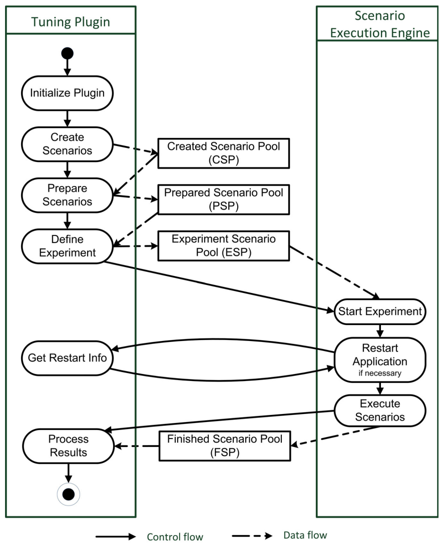
Tuning Plugin
Scenario Execution Engine
Initialize Plugin
Create Scenarios
Prepare Scenarios
Define Experiment
Created Scenario Pool (CSP)
Prepared Scenario Pool (PSP)
Experiment Scenario Pool (ESP)
Start Experiment
Get Restart Info
Restart Application if necessary
Execute Scenarios
Process Results
Finished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

Fig. 1. Simplified work flow of a tuning plugin

The predefined sequence has to cover all possible scenarios given the programming models and parallel patterns supported for tuning, besides any preparation steps required by the system (software and hardware) where the tool is running. As a consequence, the full state machine is relatively complex. For illustration purposes, a simplified version of PTF's flow is presented in Figure 1.

All steps are involved in the creation and processing of the scenarios to be experimented. Scenarios are stored in pools accessible by all plugins as well as the frontend. These pools are:

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

All steps in a plugin's workflow relate to the Tuning Plugin Interface (TPI). All methods in this interface must be implemented by all plugins; PTF checks their conformance at loading time. The TPI's major methods are the following:

Initialize: This method is called when the frontend instantiates the plugin. The plugin's internal data structures, tuning space and search algorithms are initialized and the tuning parameters are established.

Create Scenarios: From the defined variant space, the plugin generates the scenarios using a search algorithm and inserts them into the CSP, so the frontend can access them. The plugin combines the region, a variant, and the objectives (e.g. execution time and energy consumption) to generate the scenarios, using either a generic search algorithm (like exhaustive search) or its own search algorithm. The search algorithm may go through multiple rounds of scenario generation. Selecting new scenarios to be generated may depend on the objective values for the previous scenarios. 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.

Prepare Scenarios: In this method, scenarios are selected from the CSP, prepared and moved into the PSP. Before experiments can be executed, some scenarios require preparation, which typically covers tuning actions that cannot execute at runtime – e.g. recompiling with a certain set of compiler flags or generating special source code for the scenario's variant. Only the plugin can decide which scenarios can be simultaneously prepared. For example, scenarios requesting conflicting compiler flags for a same file cannot be prepared together. If no preparation is needed, this method simply moves all created scenarios from the CSP to the PSP.

After the execution of an experiment, the frontend checks if the CSP is empty. If it is not, the frontend calls the Prepare Scenarios method again.

Define Experiments: Once generated and prepared, the scenarios need to be assembled into an experiment. An experiment goes through at least one execution of the application's phase region. Multiple scenarios can be run in a single experiment assigned either to a single process (if they affect different regions) or to different processes; only the plugin can decide whether this is possible or not. For example, two scenarios for the same program region of an MPI application can only be executed in a single experiment if assigned to different processes. The plugin decides upon the assignment of scenarios to processes/threads in this method. A subset of the prepared scenarios is selected for the next experiment and moved from the PSP to the ESP. Once the experiment is defined, the frontend transfers the control to the Scenario Execution Engine (SEE), which triggers the experiment.

Get Restart Info: During the experiment execution, the SEE first checks 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 that can only be set before launching the application. This method returns *true* if an application restart is needed to execute the experiment. It also permits to return parameters to the application launch command, e.g. MPI configuration parameters to be set upon application launch as required by the scenario.

After the potential restart of the application, the SEE runs the experiment by releasing the application to execute a phase region. The SEE takes care of the execution in case multiple phases are required to gather all the measurements for the objectives, and even restarts the application if it terminates before gathering all measurements. At the end, the SEE moves the executed scenarios from the ESP to the FSP and returns the objectives to the plugin.

Process Results: The frontend calls this method if the CSP is empty and the search algorithm is finished. Here the plugin analyzes the objectives and acquired properties – implemented as standard Periscope properties – and either commands that there are extra steps necessary for tuning; or indicates that the tool is finished, selects the best scenario and generates the tuning advices for the user.

The individual steps and methods may be repeated if scenarios still remain in the scenario pools.

As introduced here, the tuning control flow is dictated by the frontend. While the general flow is predetermined, the specific combination of valid execution paths is determined by the plugin based on its internal logic and the search algorithms used.

The specific implementation of our tuning plugins based on the TPI follows in separate sections.

3.2 Tuning of High-Level Patterns for GPGPU

Tuning Objective. We investigate tuning of high-level patterns for single-node heterogeneous manycore architectures comprising CPUs and GPUs. Our focus is on pipeline patterns, which are expressed in C/C++ programs with pragma-annotated while-loops, where pipeline stages correspond to functions for which different implementation variants may exist. The associated programming framework has been developed in the EU project PEPPHER [16] and comprises a component framework, a transformation system, a coordination layer and a heterogeneous runtime system.

The following example gives an impression of a high-level image processing pipeline for face detection, where the computational stages are based on OpenCV library routines [17]. For the functions of the middle stage, different implementation variants – one for CPUs and one for GPUs – are provided. Annotations enable the user to specify a stage replication factor, which results in the generation of multiple stage invocations processing different data packets in parallel. This may be advantageous if a stage executes for much longer than its predecessor stage. Moreover, the size of the buffers that are automatically generated by the framework to pass data between stages may be specified. Merging stages can be achieved by enclosing two or more function calls within a *stage* pragma.

```
#pragma pph pipeline with buffer (PRIORITY, N*2)
while (inputstream >> file) {
    readImage(file, image);
    #pragma pph stage replicate(N)
    {
        resizeAndColorConvert(image);
        detectFace(image, outimage);
    }
    writeFaceDetectedImage(file, outimage);
}
```

Such a high-level code is transformed into a representation that utilizes a coordination layer and the StarPU [18] heterogeneous runtime system to exploit pipeline parallelism across stages. The coordination layer and runtime system decide when to schedule a stage for execution and which implementation variant to execute on which execution unit of the targeted heterogeneous architecture.

The goal of tuning pipeline patterns is to maximize the throughput by efficiently exploiting all CPU cores and GPUs of a specific targeted architecture.

Tuning Parameters and Tuning Actions. The main tuning parameters for pipeline patterns comprise stage replication factors and buffer sizes. Moreover, the tuning of the underlying coordination and runtime layers, including implementation variant selection, runtime scheduling policy and resource allocation strategy will be investigated.

To support these tuning parameters, the PTF monitor and the coordination layer provide extensions to measure the time spent executing the stages, processing the buffers and executing the overall pipeline. Tuning actions vary these tuning parameters aiming to maximize the pipeline throughput.

To restrict the variant space, a range is specified per tuning parameter. Moreover, the tuning plugin may take into account the historical execution data and static information about the concrete runtime environment (e.g. number of CPU cores, number of GPUs).

3.3 Tuning of HMPP Codelets

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

A codelet is a computational unit of a HMPP program [20]. It is typically implemented as a C function or a Fortran subroutine annotated with OpenHMPP directives [19]. The CAPS compiler automatically translates codelets into a hardware-specific language such as CUDA or OpenCL. Therefore, the execution of a codelet from the CPU perspective is considered atomic (no identified intermediate state or data).

As a portable programming standard, HMPP provides a way to access a varied set of hybrid accelerators; the CAPS compiler can currently generate code for NVIDIA and AMD GPUs and Intel Xeon Phi coprocessors. In spite of this, we restrict the plugin to Linux host systems; the plugin will come with a prototype of the CAPS Workbench different from the mainstream release version.

Tuning Parameters and Tuning Actions. This plugin has to manage a wide set of parameters that depend on the targeted architecture. Thus, this plugin allows the user to define the HMPP tuning experience via the plugin API. This API is currently accessed via OpenHMPP directives to the HMPP compiler, although more specific and autonomous auto-tuning methods – e.g. a user-friendly graphical interface for a specific domain or architecture, or automatic tools for a particular machine or application domain – may come in the future.

The tuning object is the HMPP codelet. Relative to a codelet, there are two kinds of tuning parameters to consider:

- Static codelet tuning parameters: operations, transformations or algorithms used to implement a codelet. Examples are the unrolling factor, the HMPP grid size and loop permutations.
- Dynamic codelet tuning parameters: variable or callback available at run-time, which are generally target-specific.

The variant space will be initially searched using either exhaustive or random search. Further search algorithms may be added in the future.

In this first implementation, the plugin looks for the tuning regions, the tuning parameters and the search space in the code, as they should be statically defined by the user. The information about the tuning parameters is extracted at compilation time into an extended SIR file. PTF starts after the HMPP compilation and no more compilation steps are necessary.

The scenarios executed by this plugin also specify the hardware requirements for a valid execution – the type and number of accelerators needed and if the scenario requires hardware exclusivity – since the CAPS Workbench can cope with various models of accelerators but the tuning is hardware-specific.

This plugin also extends the monitoring infrastructure provided by Periscope to deal with hardware accelerators. The HMPP profiling interface (PHMPP [21]) is tailored to extract the exact execution time of a codelet; for NVIDIA GPUs, its implementation is partially based on the NVIDIA CUPTI [22].

3.4 Tuning of Energy Consumption via CPU Frequency

Tuning Objective. The main tuning objective 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-intensive. On the hardware side we can change frequency policies and read the RAPL counters (Running Average Power Limit [23]). We specifically target the IBM System x iDataPlex thin-node islands on LRZ's SuperMUC machine, whose nodes contain two processors in a shared-memory fashion.

The power consumption can only be measured per package according to the RAPL counter: PP0_ENERGY:PACKAGE0, PP0_ENERGY:PACKAGE1, PACKAGE_ENERGY:PACKAGE0, PACKAGE_ENERGY:PACKAGE1. Each package also includes un-core elements such as the last level cache, Integrated IO, QPI and Memory controllers.

Tuning Parameters and Tuning Actions. We define two different tuning parameters: the available governors/policies and the frequencies to be used. Once an application is tuned for performance, its energy- and time-related costs can be optimized.

Regarding the first tuning parameter, there are five governors: Performance, Powersave, Ondemand, Conservative and Userspace. We discarded the Performance and the Powersave governors from our search space since they are special cases of the Userspace governor. Thus, we define the tuning parameter as an indexed vector of the three governors Ondemand, Conservative and Userspace.

Regarding the second tuning parameter, the Sandy Bridge-EP Intel Xeon E5-2680 supports, among others, the following frequencies: 2.7, 2.4, 2.2, 2, 1.8, 1.6, 1.4 and 1.2 GHz. We leave the turbo mode frequency[1] outside the range of explored frequencies.

Ideally, measurements are captured for all combinations of the available tuning parameters to select the best performing set per governor. However, trying all three governors each with all possible frequencies per region may be too time consuming (24 frequencies per region). Our experience shows that the range of frequencies describes a soft parabolic-like shape to the energy consumed. Therefore, we perform a ternary search per governor, akin to the bisection method. First, we run three experiments: the highest (f2), the lowest (f0) and the median frequency (f1). The two neighboring frequencies that resulted in the lowest energy consumption define the upcoming search interval, and the search continues until the best performing frequency is found.

3.5 Tuning of MPI Runtime

Tuning Objective. Using Periscope's standard MPI analysis, this plugin implements a combined tuning strategy to determine the values of multiple tuning parameters: a set of MPI environment variables and variants of MPI communication functions.

There are many environment variables associated with specific implementations of the MPI library. In particular, the IBM MPI library on SuperMUC offers more than 50 configurable parameters. Changes to some of these parameters could significantly affect the time an application spends in communication. The plugin assumes that the inputs are SPMD applications, so the same optimization is applied to all processes.

The MPI variables that relate to the communication buffer/protocol and to the application mapping have potential for tuning. Tuning the former variables could be effective on applications with clustered communications, while tuning the latter variables could be particularly effective on applications that exchange messages of a uniform size. The 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 of systematically launching the application using different combinations of values for both variables. Finally, the tuning recommendation consists of the pair of values that led to the application's lowest execution time.

Tuning Parameters and Tuning Actions. For this plugin we have two kinds of tuning parameters: the MPI environment parameters and the code variants for the MPI communications.

[1] In theory, the Sandy Bridge-EP Intel Xeon E5-2680 can reach a turbo frequency of 3.5GHz; in practice, however, this frequency varies, depending on several factors such as the CPU load, the quality of thermal solution and the ambient temperature.

The MPI environment parameters may be set before the application executes. We propose two tentative pairs of parameters:

– MPI application mapping: adapting tasks per node/core, adapting the affinity of the processes.
  • MP_TASK_PER_NODE: Specifies the number of tasks run on each physical node; and
  • MP_TASK_AFFINITY: Attaches a parallel task to one of the system's cpusets.
– MPI communication buffer/protocol: adapting the sending/receiving buffer, analyzing the message size patterns, adapting the communication protocol (eager/rendezvous).
  • MP_BUFFER_MEM: Controls the amount of memory for buffering data from early arrival messages[2]; and
  • MP_EAGER_LIMIT: Changes the message size threshold that defines the message passing protocol used (eager or rendezvous).

These parameters can be adjusted using environment variables or *mpirun* options, forcing all processes in the SPMD application to use the same options. The tuning action is to set the values of the MPI parameters and run a new experiment.

Regarding the code variants for the MPI communications, the user must provide different versions of the functions within the application (in function *main*) and annotate the code with user regions and their attributes:

– 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 parameter and exported to the SIR file.
– Pragma *dependency_mchoice* indicates a condition and a range of variants to explore when the condition is satisfied. For example, *dependency_mchoice v==5, bsize=128-128KB* means that for function version 5, the range for *bsize* is 128B to 128KB.

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

3.6 Tuning of Compiler Flag Selection

Tuning Objective. The tuning objective is to reduce the execution time of the application's phase region. Besides the choice of algorithm and the way the high-level source code is written, the most influential factor to the runtime is the quality of the generated machine code. Compilers apply a large number of

[2] Message data sent without knowing if the corresponding receive is posted is said to be sent eagerly. A message data arriving before its corresponding receive is posted is called an early arrival and must be buffered at the receiving side.

code transformations to generate the best code for a given architecture, e.g. loop interchange, data prefetching, vectorization and software pipelining. Although the compiler ensures the correctness of the transformations, it is very difficult to predict the performance impact and select the right sequence of transformations. Therefore, compilers offer a long list of flags and directives to allow the programmer to guide the compilation in the optimization phase.

Due to the required background knowledge about compiler transformations, compiler interactions with the application and hardware, and the large number of flags, programmers find it difficult to select the best flags and 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.

Tuning Parameters and Tuning Actions. This plugin's tuning parameters are the individual compiler flags. Each parameter can be switched either ON or OFF, hence the parameters have only two values. The tuning action is to switch a flag in the program recompilation. All tuning actions for each parameter are combined in the preparation step.

4 Evaluation

4.1 The Application Repository

The Application Repository is a central workspace composed of representative and reliable inputs for test cases. It is the main toolbox to be used during the design, development and integration of the tuning techniques and plugins to PTF. Each tool is an application – comprised of source code, configuration files and input data sets – that composes a use case of HPC software and hardware, specifically selected to match the requirements of our tuning plugins.

The repository applications currently have a dual objective:

- To guide the development of the tuning techniques and plugins; and
- To assess the quality of automatic tuning via the plugins.

The applications will serve as input to the tuning plugins and hence will provide a global view of the AutoTune behavior on scientific applications. Because of this, the applications must display the characteristics required by each plugin, be they MPI, HMPP, OpenCL, OpenACC, pipelining, data distribution or the master-worker pattern.

We also used the application repository to perform a preliminary validation of our tuning techniques by manually applying them over the applications. It aimed to provide first insights into the approach and practical implementation of the tuning plugins, besides representing proof why investing in plugin development is a worthy task.

With our manual tuning, our tuning techniques could find code variants that performed at least 25% better than the worst performing variant. Half of our techniques excelled and could find variants performing 75% better. All tuning techniques managed to find optimized code variants, on average 60.43% more efficient than the least performing variant.

4.2 The Proof of Concept

We developed a demo plugin to demonstrate and validate the auto-tuning extensions to Periscope. In this section we describe the plugin itself as well as a simple Fortran application used for evaluation.

Sample Application and 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 with an AutoTune pragma, which defines a single tuning parameter manipulated through a variable tuning action. Each value of the variable defines a unique variant.

Tuning Parameters and Tuning Actions. This tuning plugin can process a single tuning parameter as defined in the program via a directive, as it is shown below in the sample Fortran application:

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

The code region to be tuned must be surrounded by the directives “USERREGION” and “END USERREGION”. The tuning parameter is specified in the directives using the keyword “TP” followed by (i) the name of the tuning parameter, (ii) the variable identifier and (iii) the number of variants.

In this code, the value of variable *var* is initially set to k; however, since *var* is a tuning parameter, the monitor overwrites its value at runtime upon entry in the tuning region. On the other side, setting *var* to k allows the code to be run without PTF.

When instrumenting the application, the Periscope instrumenter parses the Fortran source file to extract information from the AutoTune pragmas. The tuning parameters (variable *var* in our example) are then added to the SIR file, which the frontend takes as input. The instrumentation also maps between the name of a tuning parameter and the variable. Whenever the region is entered, the monitor assigns a value to the variable to trigger the execution of a certain variant. In our example, PTF sweeps over all variants from 1 to 10.

5 Conclusions and Future Works

This paper presents the Periscope Tuning Framework, currently under development in the FP7 project AutoTune. PTF is a framework that allows for the easy development of tuning plugins. The tuning plugins explore tuning parameters based on performance information resulting from PTF's analysis strategies; different code variants are explored by running experiments that return measurements for each variant.

We implemented a first demonstration prototype of PTF to validate the overall design. In the next project cycles, we will develop initial versions of the outlined tuning plugins and test them on the application repository. Although the initial versions will be based on exhaustive or random search, we plan to add intelligence in the form of expert knowledge or model-driven search.

We also intend to design and develop a plugin for MPI programs that display the Master-Worker pattern. Furthermore, we will conduct a feasibility study and design a plugin to address I/O bottlenecks, in order to enable its implementation as a future PTF plugin. Other plugins, such as for OpenMP thread affinity, are under consideration for development and integration into PTF.

Finally, we plan to investigate combined plugin tuning strategies for inclusion into PTF. Combined strategies use multiple plugins to perform code tuning, taking different application aspects into consideration at the same time. We will especially target the design and analysis of tuning strategies for justified trade-off between energy tuning and runtime tuning.

Acknowledgements. The authors thank the European Union for supporting this project under the Seventh Framework Programme, grant no. 288038.

References

1. Miceli, R., Civario, G., Bodin, F.: AutoTune: Automatic Online Code Tuning. In: NVIDIA GPU Technology Conference 2012 (GTC 2012), San Jose, USA (2012)
2. Benedict, S., Petkov, V., Gerndt, M.: PERISCOPE: An Online-Based Distributed Performance Analysis Tool Tools for High Performance Computing 2009, pp. 1–16. Springer, Heidelberg (2010)
3. Whaley, C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the atlas project. Parallel Computing 27, 2001 (2000)
4. Frigo, M., Johnson, S.G.: Fftw: An adaptive software architecture for the fft, pp. 1381–1384. IEEE (1998)
5. Vuduc, R., Demmel, J.W., Yelick, K.A.: Oski: A library of automatically tuned sparse matrix kernels. Institute of Physics Publishing (2005)
6. Püschel, M., Moura, J.M.F., Singer, B., Xiong, J., Johnson, J., Padua, D., Veloso, M., Johnson, R.W.: Spiral: A generator for platform-adapted libraries of signal processing algorithms. Journal of High Performance Computing and Applications 18, 21–45 (2004)
7. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 204–215. IEEE Computer Society (2003)
8. Haneda, M., Knijnenburg, P., Wijshoff, H.: Automatic selection of compiler options using non-parametric inferential statistics. In: 14th International Conference on Parallel Architectures and Compilation Techniques (PACT 2005), pp. 123–132 (September 2005)

9. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO), pp. 319–332 (2006)
10. Leather, H., Bonilla, E.: Automatic feature generation for machine learning based optimizing compilation. In: Code Generation and Optimization (CGO), pp. 81–91 (2009)
11. Fursin, G., Kashnikov, Y., Wahid, A., Chamski, M.Z., Temam, O., Namolaru, M., Yom-tov, E., Mendelson, B., Zaks, A., Courtois, E., Bodin, F., Barnard, P., Ashton, E., Bonilla, E., Thomson, J., Williams, C.K.I.: Milepost gcc: machine learning enabled self-tuning compiler (2009)
12. Chung, I.H., Hollingsworth, J.: Using information from prior runs to improve automated tuning systems. In: Supercomputing. Proceedings of the ACM/IEEE SC2004 Conference, vol. 30 (November 2004)
13. Nelson, Y., Bansal, B., Hall, M., Nakano, A., Lerman, K.: Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008), pp. 1–8 (April 2008)
14. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.: A scalable autotuning framework for compiler optimization. In: IEEE International Symposium on Parallel Distributed Processing (IPDPS 2009), pp. 1–12 (May 2009)
15. Ribler, R., Vetter, J., Simitci, H., Reed, D.: Autopilot: adaptive control of distributed applications. In: Proceedings of the Seventh International Symposium on High Performance Distributed Computing, pp. 172–179 (July 1998)
16. Benkner, S., Pllana, S., Traff, J., Tsigas, P., Dolinsky, U., Augonnet, C., Bachmayer, B., Kessler, C., Moloney, D., Osipov, V.: Peppher: Efficient and productive usage of hybrid computing systems. IEEE Micro. 31(5), 28–41 (2011)
17. Gary, B.: Learning opencv: Computer vision with the opencv library (2008)
18. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A., Inria, L., Sud-ouest, B.: Author manuscript, published in "euro-par (2009)" starpu: A unified platform for task scheduling on heterogeneous multicore architectures (2009)
19. CAPS Entreprise: HMPP Directives Reference Manual, version 3.2.0 (2012)
20. CAPS Entreprise: The HMPP Workbench, `http://www.caps-entreprise.com/products/hmpp/` (accessed on October 16, 2012)
21. CAPS Entreprise: H4H - HMPP Profiling Event specification, version 2.3.3 (2012)
22. The CUDA Profiling Tools Interface, `http://docs.nvidia.com/cupti/` (accessed on October 16, 2012)
23. David, H., Gorbato, E., Hanebutte, U., Khanna, R., Le, C.: RAPL: memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (2010)