Software Qual J (2018) 26:1063–1096
DOI 10.1007/s11219-017-9370-x

A multi-aspect online tuning framework for HPC applications

Michael Gerndt[1] · Siegfried Benkner[2] · Eduardo César[3] · Carmen Navarrete[4] · Enes Bajrovic[2] · Jiri Dokulil[2] · Carla Guillén[4] · Robert Mijakovic[1] · Anna Sikora[3]

Published online: 16 May 2017

Abstract Developing software applications for high-performance computing (HPC) requires careful optimizations targeting a myriad of increasingly complex, highly interrelated software, hardware and system components. The demands placed on minimizing energy consumption on extreme-scale HPC systems and the associated shift towards hete

✉ Eduardo César
eduardo.cesar@uab.cat

Michael Gerndt
gerndt@in.tum.de

Siegfried Benkner
siegfried.benkner@univie.ac.at

Carmen Navarrete
carmen.navarrete@lrz.de

Enes Bajrovic
enes.bajrovic@univie.ac.at

Jiri Dokulil
jiri.dokulil@univie.ac.at

Carla Guillén
carla.guillen@lrz.de

Robert Mijakovic
mijakovic@in.tum.de

Anna Sikora
anna.sikora@uab.cat

[1] Technical University of Munich, Munich, Germany

[2] University of Vienna, Vienna, Austria

[3] Autonomous University of Barcelona, Barcelona, Spain

[4] Leibniz Supercomputing Centre, Garching bei München, Germany

rogeneous architectures add yet another level of complexity to program development and optimization. As a result, the software optimization process is often seen as daunting, cumbersome and time-consuming by software developers wishing to fully exploit HPC resources. To address these challenges, we have developed the Periscope Tuning Framework (PTF), an online automatic integrated tuning framework that combines both performance analysis and performance tuning with respect to the myriad of tuning parameters available to today's software developer on modern HPC systems. This work introduces the architecture, tuning model and main infrastructure components of PTF as well as the main tuning plugins of PTF and their evaluation.

Keywords Automatic performance tuning · High-performance computing · Performance optimization · Parallel architectures · Energy tuning · OpenCL

1 Introduction

The majority of today's computer systems expose several levels of hardware parallelism for the application developer to exploit. Servers consist of multiple multicore processors organized in a shared non-uniform memory access (NUMA) architecture. The server architecture can be extended via graphics processing units (GPUs) attached to the main processor, with such GPUs having their own graphics memory offering high memory bandwidth to feed data to the high number of cores. HPC systems couple hundreds of thousands of nodes, each possibly equipped with accelerators (GPUs or Intel Xeon Phi), via a high-performance network in a distributed memory architecture.

Developing efficient applications for such complex and highly parallel systems requires careful and increasingly sophisticated optimizations. To achieve good overall performance, the individual core compute capabilities need to be exploited in the best possible way. For example, many modern processors provide some form of vector units (e.g., Intel x86's MMX, SSE and AVX instructions), which need to be exploited as efficiently as possible. In addition, careful optimization of memory access patterns is required to make best use of the processor caches or software managed memories as on GPUs and to avoid that processors stall while waiting for data to arrive. Reduction of the communication volume as well as carefully mapping the processes onto the processors to best exploit the physical network topology are important for overall scalability of applications. The distribution of load has to be optimized so that none of the processes is idle at any time.

Energy reduction has become increasingly important for HPC architectures since the power bill for such machines is high relative to the costs for the machine over its entire life time. Careful application-specific tuning can help reducing the energy consumption without sacrificing application performance.

Achieving the best performance for applications on such systems is an increasingly complex, cumbersome and time-consuming task. In order to facilitate the study of complex application behaviors and associated optimization techniques, different suites of mini-apps or proxy-apps have been proposed (e.g., CESAR (https://cesar.mcs.anl.gov/content/software); CORAL (https://asc.llnl.gov/CORAL-benchmarks)). In addition, insights provided by various cost-models (e.g., the roofline model (Williams et al. 2009)) assist users in reasoning about the performance potential and limitations of their algorithms on certain types of architectures.

In a typical work cycle, application developers first analyze the performance of the application to identify regions of the application that can potentially be improved, and subsequently perform experiments with different code transformations or parameter settings of the execution environment to find an optimal solution with respect to either time or energy (and in ideal cases, both). This process is guided by the experience of the developer and the results of the performance analysis. However, due to the complexity of the optimizations, the developer often needs to implement and evaluate a large number of versions of the application before finding the optimal version. Upon implementing the optimal version of the application for a representative input data set, a verification step with alternative input data sets and runtime configurations typically follows.

Both the research community and HPC vendors have developed a number of performance analysis (PA) tools that support and partially automate the first step of the process described above. But none of the current PA tools also supports the application developer in the subsequent tuning step. The most sophisticated tools are able to automatically identify the root cause of a performance bottleneck, but they do not provide the application developer with any hints on how to tune their code. As a result, much research has been dedicated in recent years to the development of auto-tuning strategies and tools. A key point is to automatically and efficiently search for the best combination of code transformations and parameter settings of the execution environment.

In this paper, we present the Periscope Tuning Framework (PTF), which is among the first tools to support the application developer not only in the performance analysis but also in providing the developer with hints on how to subsequently tune the application to improve performance. PTF has been developed in the context of the European AutoTune project and is publicly available at the PTF website.[1] PTF is an open and extensible framework that supports the tuning of different aspects of parallel programs based on the concept of tuning plugins. PTF identifies tuning alternatives based on codified expert knowledge, and evaluates the alternatives within the same execution run of the application (online), dramatically reducing the overall search time for a tuned code version. The application is executed under the control of the framework either in interactive or batch mode, where the performance and energy measurements are forwarded to tuning plugins that then determine application alternatives and evaluate different tuned versions of the application during the application's execution. At the end of the application run, detailed recommendations are provided to the code developer on how to improve the code with respect to performance and energy consumption.

Six major tuning plugins have been developed within the AutoTune project: the Compiler Flag Selection (CFS) and the MPI Parameters plugins both target the improvement of the performance of applications by choosing the best combination of flags or parameters used either by the compiler in the build process, or by the MPI library at runtime, respectively. The Parallel Pattern and the Master-Worker plugins address performance optimization at the programming paradigm level. The Parallel Pattern plugin tunes high-level pipeline patterns for accelerated parallel systems, while the Master-Worker plugin tunes the computational load and the number of workers for applications using this common parallel programming model. The Dynamic Voltage Frequency Scaling (DVFS) plugin is unique within this set of plugins as it addresses more than one tuning objective, namely energy and power consumption. Finally, the OpenCL plugin addresses software applications targeting

[1]PTF web site: http://periscope.in.tum.de.

heterogeneous many-core or “accelerator”-based applications with automatic optimization of a kernel specific execution parameter.

The major contributions of this paper include the development of a novel automatic tuning framework (PTF) for HPC systems that tightly integrates performance analysis and autotuning via expert knowledge. PTF is a plugin-based system providing extensibility with respect to different aspects of performance and energy tuning. This paper focuses on novel tuning plugins for selection of compiler flags, for tuning of MPI parameters, for energy tuning and for tuning of OpenCL applications on GPUs and Xeon Phi coprocessors. Each of these plugins is described in detail and has been evaluated using several benchmarks and real applications. The obtained results show that significant improvements in the applications' execution time and power consumption can be automatically achieved applying the suggestions given by PTF plugins.

The rest of this paper is organized as follows. In Section 2, we provide an overview of related work in auto-tuning. Section 3 outlines the architecture, tuning model and main infrastructure components of PTF. Section 4 presents selected tuning plugins of PTF and their evaluation. Section 5 presents a summary and an outlook to future work.

2 Related work

Much research has been dedicated to the area of auto-tuning in the past few years and many different ideas have been gathered, including (1) self-tuning libraries for linear algebra and signal processing like ATLAS (Whaley et al. 2001; Demmel et al. 2005), FFTW (Frigo and Johnson 1998; 2005), OSKI (Vuduc et al. 2005) and SPIRAL (Püschel et al. 2004); (2) tools that automatically analyze alternative compiler optimizations and search for their optimal combination (Triantafyllis et al. 2003; Haneda et al. 2005; Pan and Eigenmann 2006; Leather et al. 2009; Fursin et al. 2011); (3) auto-tuners that search a space of application-level parameters that are believed to impact the performance of an application (Chung and Hollingsworth 2004; Nelson et al. 2008; Morajko et al. 2005); (4) frameworks that try to combine ideas from all the other groups (Tiwari et al. 2009; Ribler et al. 1998).

2.1 Self-tuning libraries

The first category contains special-purpose libraries that are highly optimized for one specific area. Automatically Tuned Linear Algebra Software (ATLAS) supports the developers in creating numerical programs. It automatically generates and optimizes the popular Basic Linear Algebra Subroutines (BLAS) kernels for the currently used architecture. Similarly, FFTW is a library for computing the discrete Fourier transformation on different systems. Due to the FFTW design, an application using it will perform well on most architectures without modification.

2.2 Automatic compiler optimization tools

The growing diversity of parallel application areas requires a more general auto-tuning strategy. Thus, a substantial research has been done in a different application-independent approach of auto-tuning. This is based on the automatic search for the right compiler optimizations on the specific platform. Such tools can be separated into two groups according to their methodology: iterative search tools and those using machine learning techniques. There has been much work in the first category as described in Triantafyllis et al. (2003) and

Haneda et al. (2005). All these tools share the idea of iteratively enabling certain optimizations. They run the compiled program and monitor its performance. Based on the outcome, they decide on the new tuning combination. Due to the huge size of the search space, these tools are relatively slow. An algorithm called combined elimination (CE) (Pan and Eigenmann 2006) has been developed, which greatly improves on the former search-based methods.

The second branch of compiler-based auto-tuners applies a different strategy to look for the best optimization settings. They use knowledge about the program's behavior and machine learning techniques to select the optimal combination (Leather et al. 2009). This approach is based on an automatically built per-system model that maps performance counters to good optimization options. This model can then be used with different applications to guide their tuning. Current research work is also targeting the creation of a self-optimizing compiler that automatically learns the best optimization heuristics based on the behavior of the underlying platform (Fursin et al. 2011). Zhelong Pan did pioneering work on tuning compiler flags. He developed several algorithms for walking the large search space of gcc compiler optimization flags (Pan and Eigenmann 2006). Combined elimination iteratively identifies negative combinations and eliminates these in a one by one greedy fashion. The paper confirms that the tuning result is very application dependent. Improvements from only a few up to 60% were shown. The Milepost project investigated machine learning-based compilation for gcc for Intel, AMD, and ARC CPUs (Fursin et al. 2011). Tuning results are forwarded into a public tuning database to improve machine learning results across multiple codes. The developed machine learning plugins are based on probabilistic and transductive approaches to predict good combinations of over 100 optimization flags. Pedro Bruel et.al. uses OpenTuner to tune compiler parameters of the CUDA compiler generating PTX code (Bruel et al. 2015). The experiments showed speedups compared to standard optimization flags from a few up to 30%. The results are highly application dependent. The PTF compiler flags tuning plugin presented here uses an overall tuning infrastructure similar to the work of Bruel et.al. using OpenTuner. It provides specialized search algorithms as in the work of Pan, and machine learning as in the Milepost project. It integrates these techniques into PTF's online tuning approach.

Much work has also been done in the area of autotuning stencil operations, which are at the heart of many numerical methods. A prominent example is the Berkeley autotuner (Datta et al. 2008), which focuses on optimizing the performance of stencil kernels by automatically selecting optimization strategies and corresponding tuning parameters. The Pochoir compiler (Tang et al. 2011) is a compiler and runtime system for implementing stencil computations formulated in a domain-specific language on multicore processors.

2.3 Autotuners

Among the tools in the third category is the Active Harmony system (Chung and Hollingsworth 2004; Tiwari and Hollingsworth 2011). It is a runtime parameter optimization tool that helps focus on the application-dependent parameters that are performance critical. The system tries to improve performance during a single execution based on the observed historical performance data. It can be used to tune parameters such as the size of a read-ahead buffer or what algorithm is being used (e.g., heap sort vs. quick sort). As compared with Active Harmony, the work from Nelson et al. (2008) uses a different approach that interacts with the programmer to get high-level models of the impact of parameter values. These models are then used by the system to guide the search for optimization parameters. This approach is called model-guided empirical optimization where models and empirical techniques are used in a hybrid approach.

MATE (Monitoring, Analysis and Tuning Environment) is a tuning environment for MPI parallel applications (Morajko et al. 2005) developed by one of the AutoTune partners, Universitat Autònoma de Barcelona. MATE automatically instruments the running application at runtime in order to gather information about the application's behavior. The analysis phase receives events, searches for bottlenecks by applying a performance model and determines solutions for overcoming such performance bottlenecks. Finally, the application is dynamically tuned by setting appropriate runtime parameters. All these steps are performed automatically and continuously during application execution by using *dynamic instrumentation* provided by the Dyninst library (Buck and Hollingsworth 2000; Ravipati et al. 2007). MATE was designed and tested for cluster (Morajko et al. 2007) and grid environments (Costa et al. 2008; Costa et al. 2014).

2.4 Hybrid frameworks

Popular examples for the last group of auto-tuning tools are the Parallel Active Harmony and the Autopilot framework. Parallel Active Harmony (Tiwari et al. 2009) is a combination of the Harmony system and the CHiLL compiler framework (Chen et al. 2008). It is an auto-tuner for scientific codes that applies a search-based auto-tuning approach. While monitoring the program performance, the system investigates multiple dynamically generated versions of the detected hot loop nests. The performance of these code segments is then evaluated in parallel on the target architecture and the results are processed by a parallel search algorithm. The best candidate is integrated into the application. Autopilot (Ribler et al. 1998; Ribler et al. 2001) is an integrated toolkit for performance monitoring and dynamical tuning of heterogeneous computational grids based on closed loop control. It uses distributed sensors to extract qualitative and quantitative performance data from the executing applications. This data is processed by distributed actuators and the preliminary performance benchmark is reported to the application developer.

Recent trends in ongoing tuning projects show an interest in refining the search space and search approach for auto-tuning. The X-TUNE project (X-TUNE http://ctop.cs.utah.edu/x-tune/) aims at seamlessly integrating programmer-directed and compiler-directed auto-tuning. Domain specific knowledge should herewith also be considered. Results from the SUPER project (Balaprakash et al. 2013) show that multi-objective tuning is a feasible approach to trade off power, performance, energy, and resilience.

With most related approaches mentioned above, performance analysis and tuning of different performance problems is usually done with separate tools. In contrast, AutoTune aims at bridging this gap and integrating support for both steps, analysis and tuning, in a single framework. PTF supports a wide set of different performance aspects targeting different programming models and parallel architectures. The plugin-based structure enables the user to experiment with different configurations of tuning plugins (e.g., using different search strategies) and makes PTF easily extensible to other performance aspects.

3 PTF architecture and infrastructure

Figure 1 provides a high-level overview of the PTF infrastructure. On the top level, there is Periscope, the online performance analysis tool (Benedict et al. 2010). Its main functional

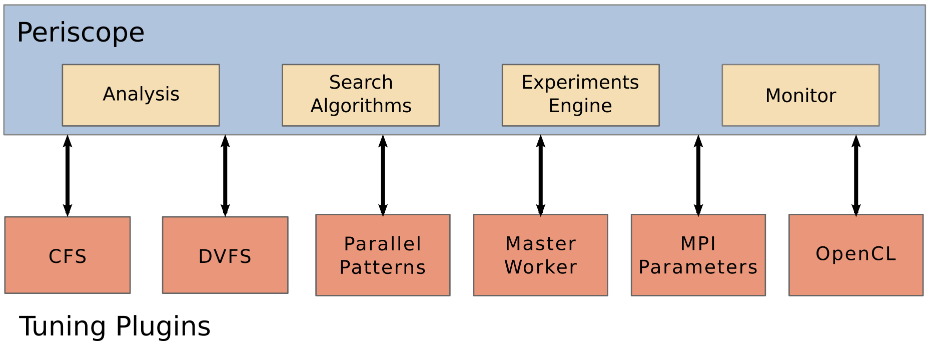


Fig. 1 The plugin infrastructure of PTF

components with respect to tuning are *Analysis*, *Search Algorithms*, *Experiments Engine*, and *Monitor* (see Section 3.2 for detailed descriptions). On the bottom level, there are the tuning plugins (described further in Section 4). Only the plugins developed within the AutoTune project are represented here, but the set of plugins is extensible.

The bidirectional arrows between the plugins and Periscope represent the two aspects of their interaction: on the one hand, Periscope is responsible for steering the tuning process and thus controls the execution flow of the plugins; on the other hand, the plugins define the tuning strategy, based on which they request specific performance analysis and tuning actions to be gathered and performed by Periscope.

The entire infrastructure was designed to cover a wide range of tuning strategies. In order to support this flexibility, the tool components exhibit rather complex functionalities and rich interaction interfaces. The AutoTune book (Gerndt et al. 2015) presents all this matter in great detail. Here, we will only focus on the main aspects of the auto-tuning approach. Particular usage examples are then illustrated by the plugins covered in the sections hereafter.

3.1 Tuning model

Every common tuning process generally follows the same algorithm, as presented in Fig. 2. The tuning model of the PTF tuning infrastructure is an implementation of this algorithm, enriched with support for extensions and specializations characteristic to performance optimization of applications. The correct flow of the process is rendered by Periscope and is exposed via the *Tuning Plugin Interface (TPI)* to the plugins. Any needed tuning information is exchanged between Periscope and the plugins by means of specialized data structures, which are communicated within the TPI calls. Figure 3 presents a diagram of the tuning model within PTF.

The first step in the PTF tuning model covers steps 1–3 of the general tuning algorithm. Each plugin optimizes an application for a specific tuning aspect, thus implicitly defining the *tuning objective*. Most common tuning objectives are execution time and energy consumption.

Plugins also enclose the expert knowledge with respect to the tuning mechanisms they utilize of and the appropriate strategy to follow—steps 2 and 3 of the general algorithm. The particular adjustments or tweaks handled by the plugins are called *tuning parameters*

```
General tuning algorithm:
1. choose a tuning objective or aspect and its acceptable threshold;
2. identify mechanism and tweaks that influence the objective;
3. decide upon a tuning approach, the sequence of tuning steps to be performed;
4. for each tuning step:
   (a) threshold_achieved = False;
   (b) while not threshold_achieved:
         i. define a set of changes to be applied, the tuning scenario;
        ii. apply the changes - perform the tuning action;
       iii. threshold_achieved = check whether the threshold was achieved;
5. the result of the tuning (tuning advice) is given by the set of changes that have to be
   applied in order to achieve the best evaluation of the tuning objective;
```

Fig. 2 Tuning algorithm within common tuning processes

and are defined along with their appropriate values. The set of all acceptable combinations of tuning parameters for a given tuning process is called the *tuning space* and is created at the initialization of the plugin.

The *Tuning Step*, marked in green in Fig. 3, covers step 4 of the algorithm, including all its substeps, 4a and 4b. The first two boxes here are extensions specific to performance tuning with PTF. In the *Pre-Analysis* step, Periscope can execute particular performance analysis of applications, as required by some plugins. This step is managed by the analysis component (see Fig. 1).

The *variant space* is a subspace of the tuning space, defined by a subset of the tuning parameters and corresponding subsets values. When generating the variant space, the plugins can use the information from the pre-analysis step in order to filter out the appropriate tuning parameters and values.

The *Search Step*, marked in gray in Fig. 3, is repeated several times within one tuning step. Due to the automatic nature of tuning, if the *tuning scenarios* are independent from each other, then it is more suitable to create an entire pool of tuning scenarios within one single search step. The creation of scenarios is handled by the Search Algorithms component (see Fig. 1), using instructions from the plugins.

Specialized for application tuning, one distinguishes in step 4(b)ii two main types of *tuning actions*: pre-execution and runtime. For example, altering the command line that starts the application is a pre-execution action and changing the CPU frequency while the application is executing is a runtime action. The first type of actions is covered in the *Prepare Scenarios* step, while the second one in *Run Experiment*.

Since PTF is tuning the performance of applications, specific measurements and analysis are carried out when running the experiments. Three components are involved within this task: Analysis, Experiments Engine and Monitor (see Fig. 1).

After having looped through all necessary tuning scenarios and tuning steps, the tuning model concludes with the *tuning advice* corresponding to the last step in the general algorithm (step 5). The advice is in XML format and provides the tuning scenario that achieved the best evaluation of the tuning objective.

3.2 PTF infrastructure components

In the following, we provide a brief overview of the common infrastructure components available to all the different tuning plugins.

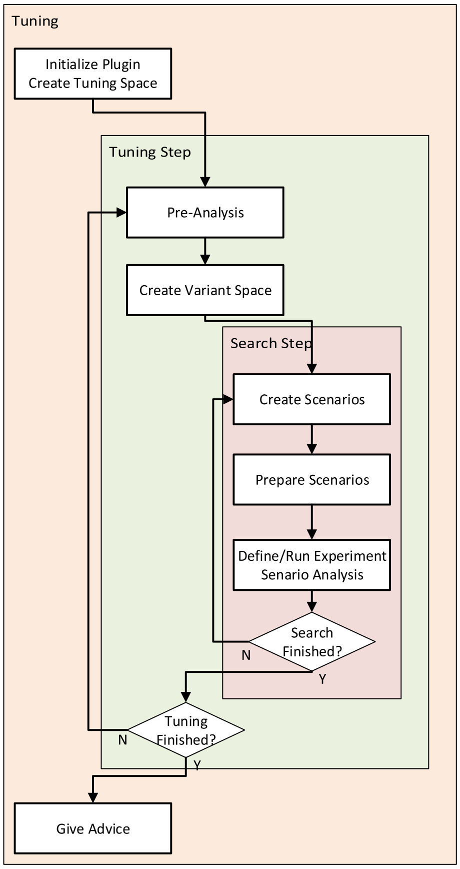


Fig. 3 The tuning model of the PTF tuning infrastructure

3.2.1 Analysis

The analysis component of Periscope is used in the pre-analysis and scenario analysis steps of the tuning model. Its functionality is based on the *performance property* concept used by Periscope to characterize the performance behavior of applications. The plugins can request a list of properties to be evaluated within the specially designed *configurable analysis* strategy of Periscope. Some examples of performance properties are `ExecTimeImportance` (see Section 4.1), for determining the most time consuming routines of an application; `ExecTime` (see Section 4.1), for determining the execution time of the whole application; and `EagerLimitDependent` (see Section 4.2), for determining if the execution time of the application could be significantly affected by the value of the MPI eager limit parameter.

3.2.2 Search algorithms

The core of every tuning technique is represented by some type of a search algorithm. For generating the scenarios and the corresponding search path, PTF provides different search algorithms including Exhaustive search, Probabilistic random search, Individual search (Bajrovic et al. 2016), GDE3 multi-objective genetic search (Kukkonen and Lampinen 2005), Active Harmony's Nelder-Mead Simplex algorithm (Ţăpuş et al. 2002), and Machine Learning Based Random Search (see Section 4.1). A single run of a search algorithm can perform one or more search steps. In each search step, a number of new scenarios is created. If there are multiple search steps, then the newly created scenarios usually depend on the results obtained in the previous search steps. PTF supports the introduction of new search algorithms, either as binaries loaded in the PTF tuning infrastructure, or as implementations in the plugins. For a discussion of the available search algorithms, we refer the reader to (Gerndt et al. 2015).

While heuristic search strategies constitute one way of coping with large search spaces, the other approach is to include expert knowledge in the plugin and to use pre-analysis strategies.

3.2.3 Experiments engine

The Experiments Engine component is mainly responsible for handling the tuning actions, both in pre-execution and at runtime. It manages the execution of the application, suspending or restarting if necessary, all in tight collaboration with the Monitor component.

3.2.4 Monitor

The Monitor component is configurable via the *Monitor Request Interface (MRI)* and is the component closest to the tuned application. It attaches to the application process(es) and gathers performance data, as configured via the MRI. It then communicates the results to the other Periscope components.

For performance measurements, applications are usually first *instrumented* with specific annotations and then they are built and linked against the measurement library. Most important and frequently used instrumentation in PTF is the declaration of code *regions*. The measurements, analysis and tuning are carried out with respect to code regions, files, or the entire application.

4 Tuning plugins

4.1 Compiler flag selection plugin

The Compiler Flag Selection (CFS) plugin searches automatically for the best combination of compiler flags to be used for a given application.

In practice, manually choosing the right flags for the compilation process turns out to be very complicated. On the one hand, there is a very large number of flags to choose from. On the other hand, one needs solid background knowledge to be able to reason about which compiler operations would benefit the performance of a given application. Moreover, the effects on application performance of one compiler option or the other are not easy foreseeable, not even for compiler experts. The application developer is thus only left the option to test some flags combinations and choose the best among those.

The CFS plugin provides an automated process that evaluates several compiler flag combinations based on a given search strategy. The assessment of the effects each combination has on the performance of the given application is supported through the integration into PTF. For each tested combination, the application is automatically recompiled and executed within the PTF tuning flow, and its performance is measured and analyzed by Periscope.

Figure 4 gives an overview of the integration of the CFS plugin within PTF.

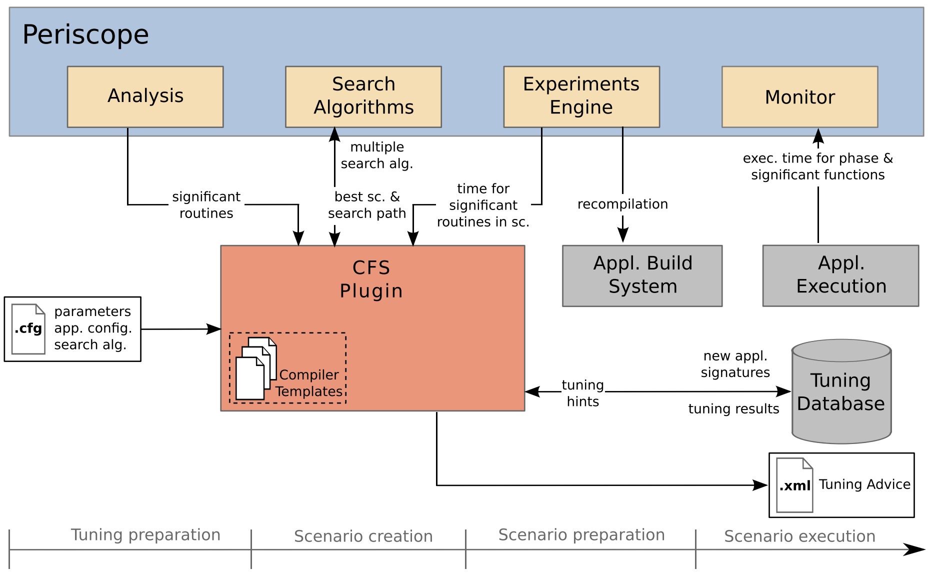


Fig. 4 Integration of the CFS plugin into PTF and specific aspects of the tuning flow

4.1.1 *Compiler templates and plugin settings*

The *tuning objective* addressed by the CFS plugin is the execution time of applications. The mechanism through which the execution time is reduced is through compiler optimization guided by selected compiler flags. These compiler flags thus represent the *tuning parameters*, and the *tuning space* is given by all possible flag combinations.

The *compiler templates* are pre-defined sets of compiler flags provided as template files in the standard installation of the CFS plugin. These templates can be loaded in any tuning process and they offer the advantage that the set of tuning parameters and corresponding values they define are known to work best for the target compiler.

Custom *tuning parameters* can also be defined in the configuration file, as represented in the tuning preparation step in Fig. 4. Other settings that influence the execution of the tuning flow are the search algorithm to be used, as well as the selective build feature addressed next.

4.1.2 *Significant routines and selective build*

The analysis component in Fig. 4 provides *significant routines* to the CFS plugin in the *Tuning preparation* step. More precisely, this is the pre-analysis step of the tuning model. The CFS plugin uses this step to refine the compilation process of the given application. It uses the *Importance* performance analysis strategy of Periscope to determine, based on the `ExecTimeImportance` performance property, which of the routines of the application have an execution time above a certain percentage of the total considered execution time. Only these files are then considered and recompiled during the tuning process.

It is often the case for real-world applications that they have a considerable large compilation time. As the CFS plugin needs to recompile the application for each tested combination of compiler flags, reducing the number of files to be recompiled also reduces the total time needed for tuning.

The list of files for the selective build process can be either determined in the pre-analysis step or it can be provided directly by the plugin user.

4.1.3 Search refinements

The CFS plugin relies on the Search Algorithm component of PTF to generate the compiler flags combinations in the *Scenario creation* step. The plugin feeds the component with the tuning parameters, i.e., the compiler flags, and the Search Algorithm component generates the tuning scenarios, i.e., the desired combinations of flags. The search algorithm also guides the search path and delivers in the end the best scenario along with the traversed search path.

There are two characteristics specific to the optimization based on compiler flags and which are exploited by the CFS plugin in the search process. First of all, some compiler flags are expected to have a higher impact on the performance of applications than other flags. Hence, sorting the list of tuning flags based on their expected impact and using the greedy-like algorithm *Individual Search* has the potential to deliver close to optimal results while also speeding up the entire tuning time. The second characteristic is that the compiler flags tend to be consistent with the *program signature*. The program signature is a collection of Performance Application Programming Interface (PAPI) counter values measured for the given application (see (Gerndt et al. 2015) for details). The consistency between compiler flags and program signature makes it possible for the CFS plugin to use the machine learning-based search strategy, also provided by PTF. In Fig. 4, one can see the Tuning Database, which is fed with new program signatures and tuning results. In response, appropriate flags combinations are delivered to the plugin.

4.1.4 Application compilation and execution

The last steps of the tuning model refer to applying the *tuning actions* and evaluating their impact. The CFS plugin has a *pre-execution* type of tuning action, namely the recompilation of the application using the currently tested combination of compiler flags. The Experiment Engine first triggers recompilation and then starts the execution of the application. During execution the Monitor measures the `ExecTime` performance property, needed for evaluating the effects of the scenario. The execution time for the tuned region and, if selected, the significant routines are then delivered back to the CFS plugin. The plugin analyzes the returned execution times and recommends a globally best flag combination as well as individual best flag combinations for the selected significant routines.

4.1.5 Extensions for tuning OpenCL kernels

Performance of OpenCL kernels can be significantly influenced by selecting proper optimizations for the backend compiler. The CFS plugin also supports the tuning of OpenCL kernels for various target accelerators based on compiler flags for offline compilers. Figure 5 outlines these extensions. The plugin comes with scripts for offline compilation of OpenCL kernels.

The CFS plugin supports target devices of two OpenCL vendors, Intel and NVIDIA. Intel provides a kernel builder and an LLVM-based optimizer (The LLVM Compiler Infrastructure http://llvm.org/) while NVIDIA does not provide an offline compiler. The script

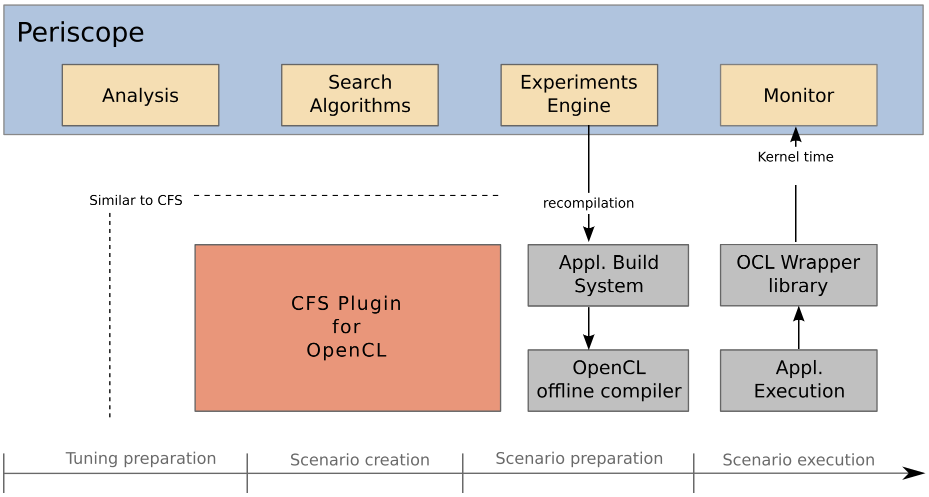


Fig. 5 Integration of the CFS plugin for OpenCL tuning into PTF and specific aspects of the tuning flow

provided with the CFS plugin for Intel targets, i.e., standard processors and Xeon Phi, first generates the *Standard Portable Intermediate Representation (SPIR)* file with the `ioc` tool for the selected OpenCL compiler flags. This intermediate version is input to the `OCLOpt` optimizer, which generates an optimized version based on a large number of standard compiler optimization flags. `ioc/ioc64` (OpenCL SDK User-guide https://software.intel.com/sites/default/files/m/e/7/0/3/1/33857-Intel_28R_29_OpenCL_SDK_User_Guide.eps) is an offline compiler for OpenCL available in Intel OpenCL SDK, while `OCLOpt` is an LLVM-based optimizer that provide more than 250 optimization options.

The script developed for offline compilation for NVIDIA GPUs wraps NVIDIA's online compiler and uses besides the OpenCL standard optimizations additional optimizations defined by NVIDIA.

OpenCL tuning with the CFS plugin depends on the MRI monitor to measure the kernel execution time. The monitor performs the measurements for each of the kernels with the help of an OCL Wrapper library. The kernel times are reported back in the form of performance properties that are then used in the plugin to evaluate the different tuning scenarios with respect to the execution time objective. The plugin reports for each kernel the best flags combination.

4.1.6 Evaluation

Automatic tuning with the CFS plugin was applied to a wide range of applications on the SuperMUC[2] system at Munich: nine applications from the AutoTune application repository, three applications from CESAR (https://cesar.mcs.anl.gov/content/software) and two applications from the CORAL (https://asc.llnl.gov/CORAL-benchmarks) benchmarks suite. The results are very much dependent on the applications themselves. Nevertheless, CFS was

[2]The software environment on SuperMUC comprised the Intel Compiler 14, Parallel Environment 1.3, and OS SLE11 SP3. Details on SuperMUC can be found at: https://www.lrz.de/services/compute/supermuc/systemdescription

Table 1 Tuning parameters used for evaluation of compiler flag selection plugin

```
tp "TP_IFORT_OPT" = "-" ["O2", "O3", "O4"];
tp "TP_IFORT_XHOST"  = " " ["-xhost", " "];
tp "TP_IFORT_UNROLL" = " " ["-unroll", " "];
tp "TP_IFORT_VERSION" = " " ["-opt-multi-version-aggressive", " "];
tp "TP_IFORT_FMA" = " " ["-fma", " "];
tp "TP_IFORT_INLINE" = " " ["-finline-functions","-fno-inline-functions"];
tp "TP_IFORT_PREFETCH" = "-opt-prefetch=" [1,4,1];
tp "TP_IFORT_UNROLL" = "-unroll" [1,16,4];
tp "TP_IFORT_OPTBLOCK" = "-opt-block-factor=" [1,3,1];
tp "TP_IFORT_STREAM" = " " ["-opt-streaming-stores always",
          "-opt-streaming-stores never", "-opt-streaming-stores auto"];
tp "TP_IFORT_IP" = " " ["-ip", " "];
```

able to provide speedups ranging from 1–4% for codes from the AutoTune repository like Convolution, Primes and HydroC, to 6–15% for NPB benchmarks BT, LU, and SP, and even 16–38% for SeisSol and Fssim.

The OpenCL extension of the CFS plugin was evaluated with three different benchmarks applications on Intel Xeon CPUs and Intel Xeon Phi. The best flags combinations are highly dependent on the target architecture and improvements from 11 to 56% were achieved. This indicates that the Intel OpenCL compiler did not yet achieve the maturity of the FORTRAN and C++ compilers as well as the importance for OpenCL tuning via carefully choosing the compiler flags.

4.1.7 Evaluation of search strategies

In order to evaluate different search strategies, we experimented with three proxy exascale applications (MOCFE, NEKBONE and RSBench) from CESAR (cesar.mcs.anl.gov) (CESAR https://cesar.mcs.anl.gov/content/software), which are designed to explore the design space of exascale machines and resembling future exascale applications.

The following search strategies have been evaluated:[3] (RS) Random Search where the plugin randomly selects combinations of flags; (IS) Individual Search, which includes the compiler flags one after the other from the tuning parameter list in the order used in the specification file; (GS) Genetic Search, which searches for the combination of compiler flags based on the GDE3 genetic algorithm; and (MLRS) Machine learning-based Random Search, which uses the Tuning Database of PTF with a collection of CFS tuning results for the NAS parallel benchmarks to model and predict suitable compiler flags for the application to tune.

In the following, we present an evaluation of these search strategies for each of the three proxy applications using the compiler flags and possible values according to Table 1. PTF exploits that these codes have a progress loop where the same computation is executed in each iteration. Each iteration is called a phase and the loop body is the phase region. Due to the similarities of the phases, the measurements are performed for a single phase only. Figures 6, 7, 8, 9, and 10 thus show the phase execution times by different search algorithms after a given number of experiments.

[3]Due to the large number of flags, exhaustive search has not been used. It would have required over 27000 experiments.

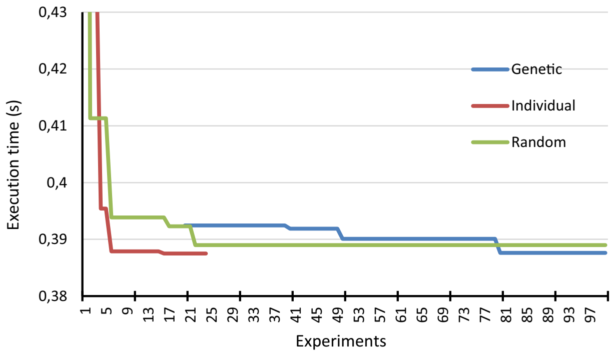


Fig. 6 Comparison of IS, GS, and RS for NEKBONE

NEKBONE The NEKBONE proxy application is a thermal hydraulics simulation code for reactor simulations that solves Poisson, Helmholtz, and other equations. The computational domain is partitioned into high-order quadrilateral elements. This is an MPI-based FORTRAN application consisting of 13 files. It was executed with 8 processes and 50 elements per process, a polynomial order of 10, and without the multi-grid preconditioner, i.e., Example 3 in the source code provided at CESAR (https://cesar.mcs.anl.gov/content/software).

Figure 6 compares the different search algorithms for the NEKBONE application. The x-axis represents the experiments executed for the different configurations and the y-axis the execution time of a phase. All three search algorithms converged to almost the same execution time of 0.38 seconds. While IS found the best execution time already after 5 experiments, RS took 22 experiments and GS 80 experiments. The line for GS only starts at 20 experiments since the initial population is 20 configurations, and a result is only available after the whole population was evaluated. RS was configured to execute 100 experiments, while GS took overall 200 experiments until the search algorithm stopped due to convergence detection. The figure also confirms that the order of the flags evaluated by IS as given by the configuration relates very well to the potential improvement.

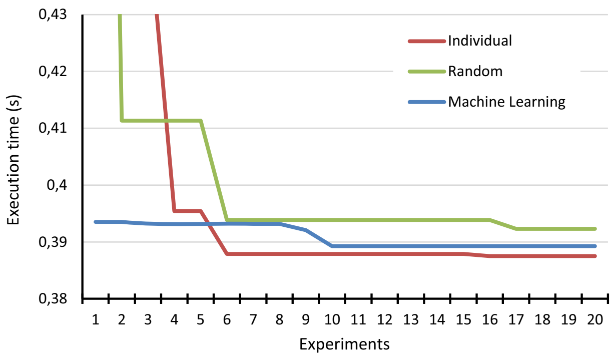


Fig. 7 Comparison of IS, RS, and MLRS for NEKBONE

As a further experiment, we executed MLRS based on the tuning results for the NAS Parallel Benchmarks. MLRS was configured to use only 20 experiments. The results are shown in Fig. 7. It can be seen that random search guided by machine learning converged much faster than pure RS and even IS. On the other hand, IS found a slightly better solution.

Figure 8 presents the quality of the whole population of 10 individuals created by GS. Since the GDE3 algorithm is suited for multi-objective tuning (Xiujuan and Zhongke 2004), it keeps a pre-configured number of best individuals in the populations determining the Pareto curve. The figure presents the range of the quality of individuals in the current population. The x-axis is the population number and the y-axis is the phase execution time. This diagram also explains why GS took 20 populations. The result improved over these populations continuously with respect to the overall quality of the individuals.

RSBENCH This proxy application models the multi-pole cross section look up algorithm. It is implemented in C and uses OpenMP. For the evaluation, we executed it with 8 threads using the following application arguments:

$$-\text{t } 8 - \text{s small} - \text{l } 1000000 - \text{p } 1000 - \text{w } 100$$

where, -t represents the number of OpenMP threads, -s represents the size of the benchmark to run, -l represents the number of cross section lookups, -p represents the average number of poles per nuclide, and -w represents the number of windows per nuclide. The cross-section parallel lookup simulation part of the application, which is iterative in nature, is instrumented as the phase region for PTF timing evaluation.

Figure 9 shows the results of the different search algorithms. The best value is found by GS after 80 experiments but it took GS up to 129 experiments to detect convergence. The second best result was found by IS after 29 experiments. We can also see from this graph that MLRS again improved on the convergence of RS, although better solutions are finally found by RS after 30 more evaluations. The reason is probably that the training for machine learning did not include all of the flags used for during this experiment.

MOCFE MOCFE-Bone is a mini-application that simulates the main procedures in a 3D method of characteristics (MOC) code for the numerical solution of the steady state neutron

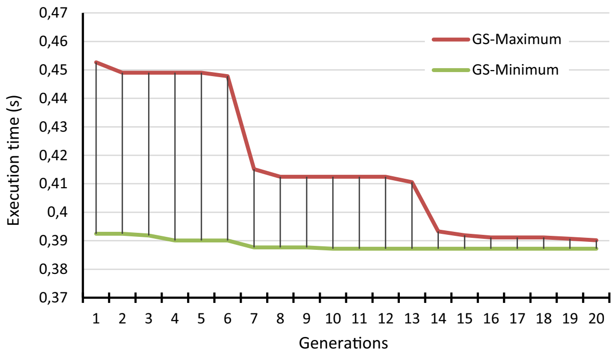


Fig. 8 Quality of the whole population of GS with 10 individuals for NEKBONE

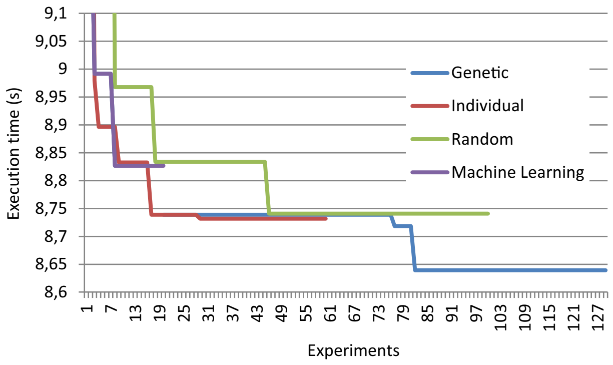


Fig. 9 Comparison of IS, GS, RS, and MLRS for RSBENCH

transport equation. 3D-MOC features heterogeneous geometry capability, high degree of accuracy, and potential for scalability. It is a FORTRAN 90 MPI application consisting of 36 files.

We have measured the phase, i.e., the inner loop of `Method_FGMRES.F90` of the application. The application was executed with 16 processes and a Krylov iteration size of 60.

Figure 10 shows the results of the different search algorithms. For this application, IS, GS, and RS found the same best execution time. While IS was again faster than GS and RS, GS found the same value almost as fast as IS but went through many more experiments to detect convergence. In contrast to RSBENCH and NEKBONE, RS already starts with good values in this experiment. MLRS started with worse results than RS but found the best execution much faster than RS.

Overall, it can be seen for these three proxy applications that, in most cases, GS found the best combination of compiler flags and MLRS converged within a few number of flag combinations, which is much faster than RS. Nevertheless, in all cases, IS was able to find a near optimal solution, which is much faster than all other algorithms. In addition, it can also be seen that tuning the compiler flags is leading in all these cases to an improvement in the applications execution time.

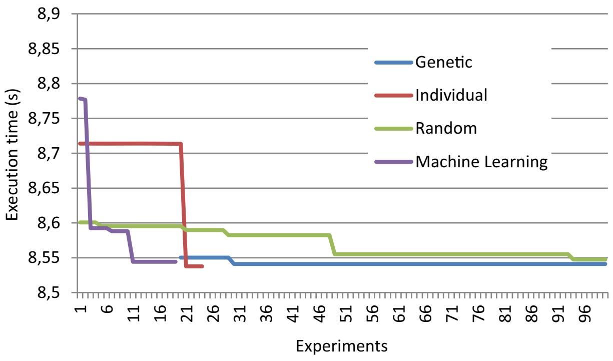


Fig. 10 Comparison of IS, GS, RS, and MLRS for MOCFE

4.2 MPI parameters plugin

MPI is the “de facto” standard for interprocess communication in distributed-memory environments and thus the optimization of MPI aspects represents a key factor in the optimization of large-scale parallel applications. However, the setups of libraries are usually specific to the environment where the libraries are installed, and often they do not achieve the same performance in another environment. In order to overcome this portability issue, most MPI implementations provide multiple runtime parameters that can be changed by the user. For example, IBM MPI, Intel MPI, and OpenMPI provide 50–150 runtime parameters from which ten to several tens can significantly influence the application performance.

The MPI Parameters plugin aims to automatically optimize the values of a user selected subset of MPI runtime parameters. The integration with PTF (see Fig. 11) provides the plugin with on-line measurements in the form of high-level properties, allowing it to make tuning decisions based on the actual performance of the application. The plugin generates the scenarios to represent specific MPI configurations in the form of tuples of parameter-value pairs (i.e., specific combinations of values for the selected subset of MPI parameters). These scenarios are executed as experiments via PTF and evaluated using the resulting properties.

4.2.1 *Parameter templates for MPI flavors*

In the tuning preparation step, a set of MPI parameters and their possible values are obtained from a configuration file. These are the *tuning parameters* and the *search space* is then generated with one scenario for each possible combination of parameter-value pairs.

The plugin provides three default configuration files, for IBM MPI, Intel MPI and Open-MPI, the MPI flavors that are directly supported. Each such template file includes the

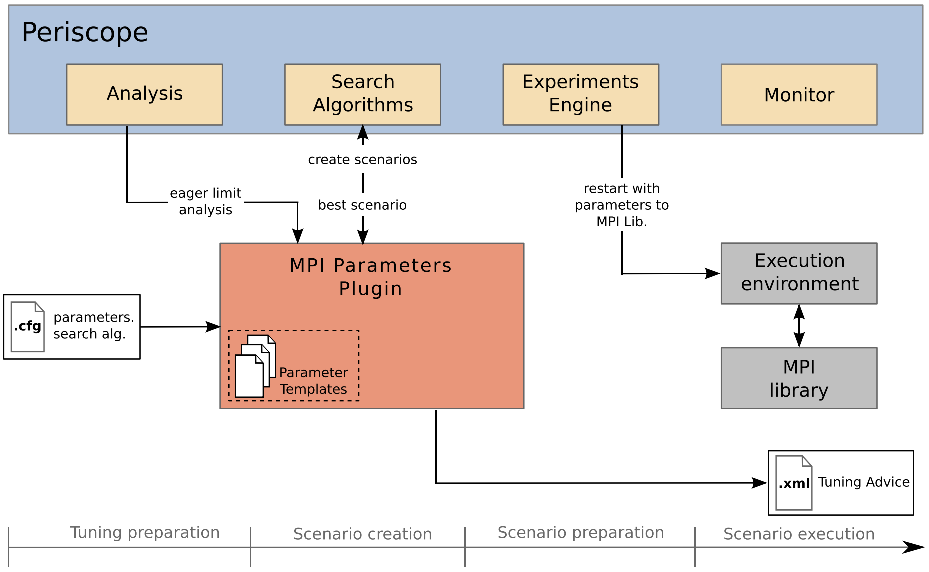


Fig. 11 Integration of the MPI parameters plugin into PTF and specific aspects of the tuning flow

parameters that usually have more impact on performance for each implementation, and it can be the starting point for the automatic tuning process.

4.2.2 Model-based search space reduction

Depending on the number of parameters to be tuned and the range of values they can take, the potential size of the search space can be huge. In addition, given the number of MPI parameters that can influence the performance of the applications, and the dependence of the parameters on the library implementations, it is extremely difficult to find general models to guide the tuning.

The MPI parameter plugin provides a prediction model only for a reduced set of specific parameters, consisting of the *eager_limit* and the *buffer_mem* parameters. The two parameters are especially relevant because of their high potential influence on the performance of applications.

The goal is thus to shrink the search space for the tuning process. In order to do so, one pre-analysis step is required. The plugin uses the *configurable analysis strategy* of Periscope, asking for the EagerLimitDependent property. This property returns the proportion of point to point messages in the valid range of the eager limit, over the total number of messages. It also classifies these messages in buckets within the valid range of the eager limit.

With the information acquired in the pre-analysis step, the plugin first decides if it is worth tuning the eager limit parameter. The parameter will be included in the tuning space only if the proportion of bytes in the valid range of the eager limit is more than 30% of the total number of bytes sent by the application. If this is the case, the plugin then analyzes the number of messages in each bucket and sets the search space according to the limits of the smallest group of consecutive buckets that includes a major proportion of the total number of messages.

Finally, the memory buffer parameter is set based on the values obtained for the eager limit parameter. A detailed description of the approach for computing the parameter values can be found in Gerndt et al. (2015).

4.2.3 Heuristic search

As already mentioned, the total number of possible tuning parameters remains very large and thus, in most cases, an exhaustive search over the entire search space would be impractical. The plugin gives users the possibility of using one of the PTF's heuristic algorithms (GDE3, Individual, or Random) to guide the search, drastically reducing the number of experiments to be executed. The appropriate search strategy can be chosen in the scenario creation step.

4.2.4 Execution of the tuning experiments

In the last step of the tuning flow, the plugin starts to experiment with the scenarios according to the selected search strategy. It evaluates each of them using execution time as the *tuning objective*.

The application must be restarted for each set of options, because MPI parameters cannot be changed during the execution of an application. For each experiment, the plugin builds the new command-line string to re-launch the application with the new configuration. This is then executed by the Experiments Engine component as a pre-execution *tuning action*.

The plugin finishes when all the selected scenarios are explored or a time limit is reached, and the scenario that yields the best performance (lower execution time) is used as the *tuning advice* to the user.

4.2.5 Evaluation

The plugin was evaluated on five applications from the AutoTune applications repository (Fssim, SeisSol, NPB, modp and Pmatmul) and one extra application developed at the UAB group (XFire), using different MPI flavors (IBM MPI and Intel MPI) and different search strategies (exhaustive, GDE3, Individual, and Random), as well as automatically determining the range for the *eager_limit* and *buffer_mem* parameters. As expected, results depend on the application, some applications were almost unaffected because the weight of communication is not relevant for them. For others, moderate improvements of up to 10% were obtained. For applications with significant communication activity, an improvement of up to 60% was achieved, e.g., for the Fssim application, which is a biological simulator that models the behavior of large fish schools.

Figures 12 and 13 show the results obtained by PTF for Fssim on the SuperMUC using a medium workload of 64K individuals and 64 MPI processes, distributed over 4 SuperMUC Phase 1 thin nodes (16 cores per node). Again reported execution times (seconds) refer to a phase. In the first case (Fig. 12), an exhaustive search was performed on five IBM MPI parameters (including the *eager_limit* one) obtaining a speedup of 1.56 (2.77 seconds for the best scenario vs. 4.32 seconds for the execution with the parameters' default values). However, this search took almost 7.5 hours for running 1024 scenarios. In the second case (Fig. 13), an individual search is performed on ten IBM MPI parameters (including the *eager_limit* and *buffer_mem* ones), obtaining a speedup of 1.51 (2.86 seconds vs. 4.32 seconds) testing only 119 scenarios in 52 minutes. These results show the advantage of using a heuristic instead of exhaustive search.

In addition, in both cases, the *eager_limit* parameter had the highest impact on the application performance, which is reinforced when applying the model-based search space reduction strategy explained in Section 4.2.2. Using this strategy, the plugin determines that

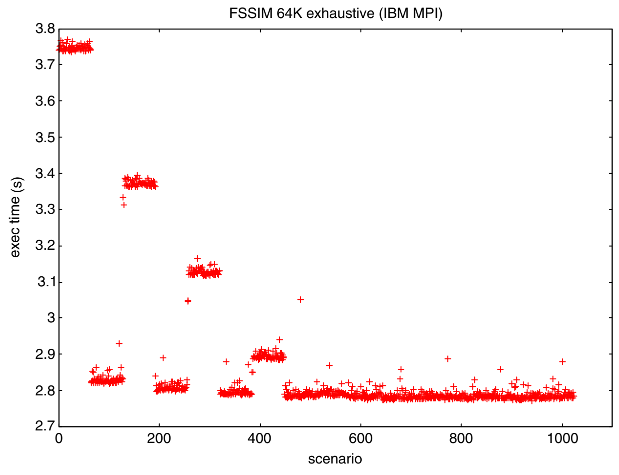


Fig. 12 Execution of Fssim using the MPI parameters plugin with exhaustive search

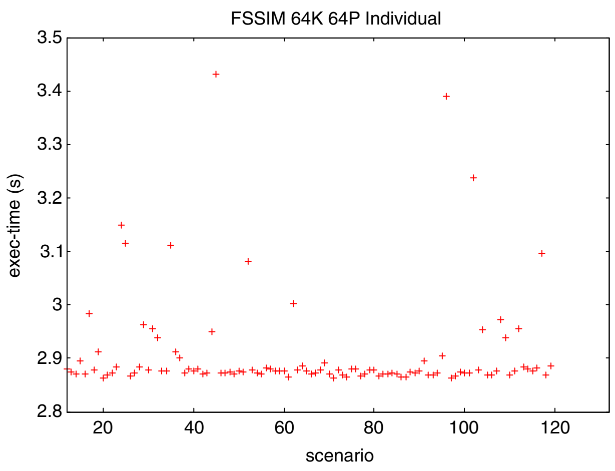


Fig. 13 Execution of Fssim using the MPI parameters plugin with individual search

more than 90% of the bytes communicated by the application were sent in messages of less than 64KB (the upper limit for the IBM MPI *eager_limit* parameter) and that most of these messages were concentrated in the 16 to 32 KB range.

4.3 DVFS plugin

Energy and power related costs dominate the expenditures needed to run HPC systems. Typical power requirements of large HPC systems are in the range of several megawatts, which means several million euros each year. The main goal of the DVFS plugin is to tune HPC applications with respect to energy and power related metrics.

The tuning mechanism is based on the dynamic voltage frequency scaling (DVFS) feature of modern processors. The plugin currently supports Intel Sandy Bridge-EP processors and it uses the *userspace* governor[4] to ensure that a given CPU frequency is not altered by the operating system unless explicitly requested by the user. The DVFS plugin depends on the architecture because it uses Running Average Power Limit (RAPL) hardware counters and a model with coefficients calculated for a particular architecture. Consequently, for using the plugin on a different type of processor, the names of the RAPL counters should be changed in the appropriated header files and the model coefficients recalculated for the new architecture. Figure 14 shows the main components of the plugin as well as the main workflow.

4.3.1 Specialized tuning objectives

The DVFS plugin is unique in that it focuses on different tuning objectives: energy consumption, power capping, total cost of ownership (TCO), and energy delay product (EDP). Besides these four tuning objectives, the DVFS plugin also provides three policies for energy and power tuning with respect to the allowed performance degradation.

[4]Governors are processor policies to change frequency.

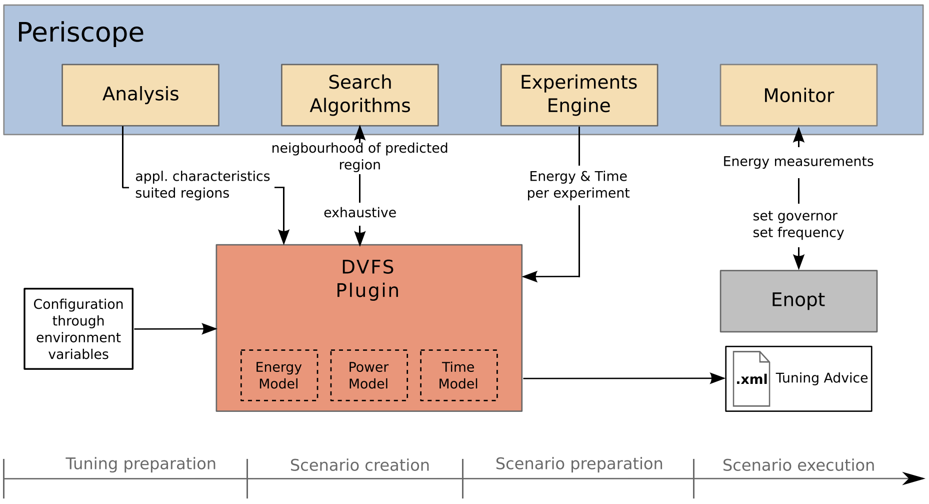


Fig. 14 Integration of the DVFS plugin into PTF and specific aspects of the tuning flow

The energy consumption objective deals with energy savings, which have a direct impact on the costs for running HPC systems. Power capping strives to keep power demand below a given limit, in order to avoid penalties imposed when power spikes occur. High power oscillations can also damage the infrastructure, making costs rise unexpectedly. EDP and TCO take into account the costs associated with time and thus address the productivity of HPC centers.

Within one run of the plugin, the application is tuned with respect to only one of these tuning objectives, which needs to be specified by means of an environment variable.

4.3.2 *Region-level tuning*

Another particularity of the DVFS plugin is the fine-grained tuning approach. While there is only one *tuning parameter*, the CPU frequency, the plugin does not tune the entire application with only one value of the tuning parameter, but it optimizes specific code regions. The code regions eligible for tuning, called *suited regions*, are determined in the pre-analysis step by executing the application at a *nominal frequency*.

Moreover, the frequency can be set either at the core level or at the node level. In HPC systems, one computing node usually consists of one or more multicore CPUs. By default, the plugin sets the frequency from every task to the core where the task is running. The DVFS plugin works with MPI, OpenMP, and hybrid codes and the frequencies are changed on the cores where the threads or the processes run. Alternatively, the frequency can also be set from a master task to the entire node.

4.3.3 *Model-based search space reduction*

The DVFS plugin relies on three main models, an energy model, a time model, and power model in order to reduce the search space. All tuning objectives use either one of the three main models, or/and some derived models to predict the corresponding best frequency setting (see Gerndt et al. (2015) for more information).

In the pre-analysis step, along with the suited regions, the application performance characteristics are determined and used as an input to predict the value of the tuning objective for each available frequency. Based on the value of the tuning objective, the *global frequency* is then set. The global frequency represents the best frequency to be used for the main code region. What exactly *best* frequency means, depends on the chosen tuning objective. For example, the best frequency when tuning energy consumption is at the minimum of this metric. For power capping, the best frequency is the highest frequency to not surpass a given power limit. We then construct a limited search space comprising the global frequency and the next lower and the next higher frequency and explore this search space by means of exhaustive search.

4.3.4 Kernel-level tuning actions

After the experiments engine creates and dispatches all the needed experiments, there are two tuning actions that need to be performed at runtime: setting the governor and setting the frequency. Since these are privileged operations that cannot be executed by common processes, PTF uses a specialized library, the enopt library (Navarette et al. 2014), to access the energy measurements and to set the frequencies of the cores on which the application runs.

The same library has to be used also afterwards, when applying the tuning advice provided by the DVFS plugin. The application needs to be linked to the enopt library and the proper frequency settings have to be provided by means of an environment variable.

4.3.5 Evaluation

The DVFS plugin was applied to a wide range of applications with several tuning objectives. As much as 20 automatic tunings were performed in less than a month that optimized six applications for energy, power capping, TCO, and/or EDP. Figure 15 shows a summary of these results. The energy savings for the energy tuning objective are also shown.

4.4 OpenCL plugin

OpenCL is an explicitly parallel programming language designed for achieving portability across different kinds of multicore and many-core architectures. OpenCL requires the

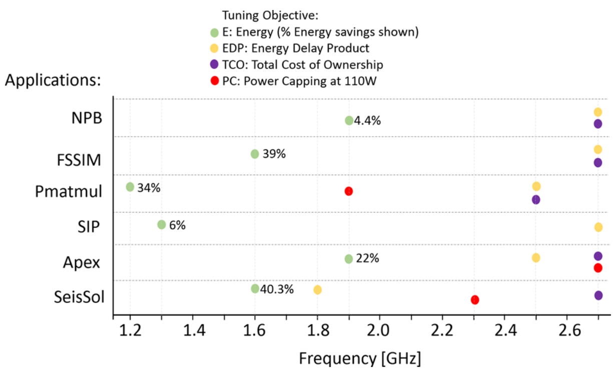


Fig. 15 Results of DVFS pluging for different applications and tuning objectives

programmer to specify how the parallel execution of computational kernels should be organized by defining a so-called NDRange, which defines how work items are to be executed in parallel by multiple threads organized into threads groups. The NDRange configuration has a profound influence on performance and the best configuration usually depends on the parallelism available in a computational kernel as well as on low-level architectural details of the target devices and may widely vary between CPUs, GPUs and other types of accelerators. The OpenCL plugin targets automatic optimization of the NDRange parameter for kernel execution for a specific application on a specific parallel architecture such that overall execution time is minimized.

The NDRange parameter controls the number of work items to be processed by the kernel and the way these work items are divided into work groups. It specifies the number of times the kernel is executed (in parallel), processing a different work item each time, but also how parallel execution is organized and how threads are mapped to the available compute units. Each work item corresponds to a point either in 1D, 2D, or 3D space. When launching the kernel, it is necessary to specify a range in each dimension, which defines the space that is to be processed by the kernel. The work items are partitioned into smaller chunks—the 1, 2, or 3-dimensional space is divided into equally sized 1, 2, or 3-dimensional *work groups*. The size of the work groups is also defined by the NDRange parameter and it is called *local work size*. The global work size in any dimension must be divisible by the local work size in that dimension.

Usually, the global work size is defined by the problem size, with small adjustments due to issues like padding or alignment. The local work size may also be dictated by the algorithm and the problem size, but in many cases, the local size can be selected from a range of options. The exact range of options may depend on the actual kernel and the device. The local work size may also affect the amount of resources (local memory, registers, etc.) needed to process a work group. In some cases, the kernel is written in such a way that it can run (it adapts) with any valid (remember that it has to evenly divide the global work size) local work size or it is completely independent of the local work size. In these cases, the local work size can still significantly affect the performance of the kernel.

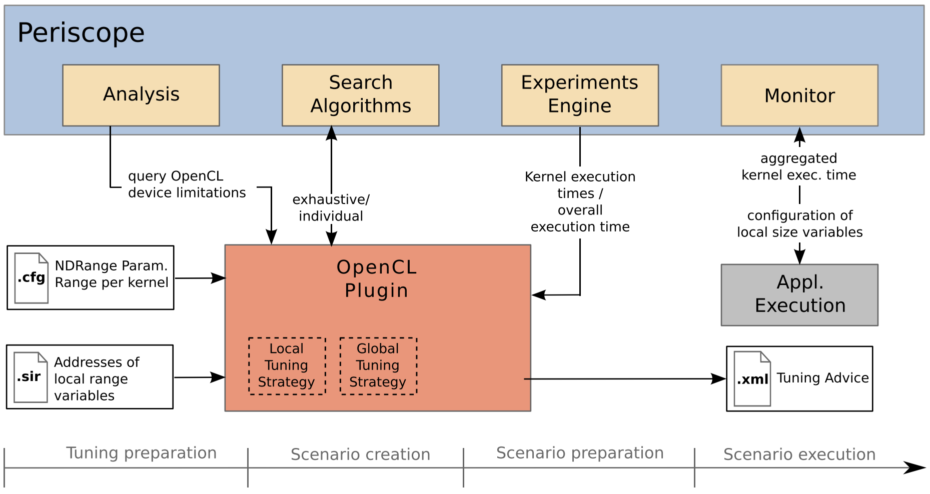


Fig. 16 Integration of the OpenCL plugin into PTF and specific aspects of the tuning flow

The OpenCL plugin automatically searches for the best local work size for each kernel in an OpenCL application so that the application or kernel execution time is minimized.

4.4.1 PTF integration and tuning specification

The integration of the OpenCL plugin, along with the general tuning flow is shown in Fig. 16. Each OpenCL kernel may have up to three tuning parameters, i.e., one for each dimension of the NDRange. The plugin requires an XML configuration file containing a user-provided NDRange tuning specification of feasible local work sizes for all OpenCL kernels that should be tuned. Feasible local work sizes may be specified either via a range expression or explicitly enumerated. Furthermore, the user can choose between a local and a global tuning strategy (see below). The plugin relies on PTF's OpenCL performance analysis facilities to obtain the execution time of the individual kernels and the whole application. The OpenCL API is used in the pre-analysis phase of the plugin to query device limitations and limits for a specific kernel, which may result in the removal of some values from the configuration file.

4.4.2 Search space and tuning strategy

The OpenCL plugin processes the user provided NDRange tuning specification and constructs a search space. The way how the search space is constructed and explored depends on whether the local or global tuning strategy is used.

With the local tuning strategy, each OpenCL kernel in the application is tuned separately. For each kernel, a kernel-specific search space is constructed according to the tuning specification and then explored with a search strategy in order to find the NDRange configuration that minimizes the execution time of the kernel. The same process is repeated for all other kernels in the application.

With the global tuning strategy, a global search space is determined by the cross product of all kernel-specific search spaces. This global search space is then explored by a search strategy with the objective of minimizing overall application execution time. Provided that the user specified tuning specification covers all feasible NDRange configurations for all kernels, the global tuning strategy in combination with exhaustive search will find the scenario with minimal execution time. However, if the application contains many kernels, the search space might become prohibitively large. With the local search strategy, the number

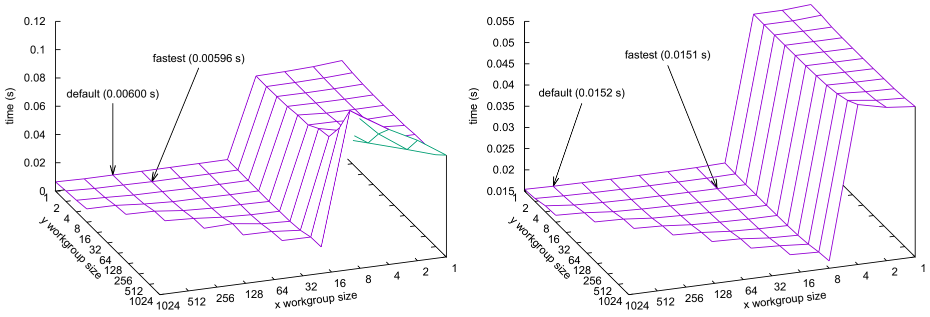


Fig. 17 Execution times of Coulomb summation benchmark on Xeon Phi 5110P (left) and Xeon E5-2650 (right) for different NDRange configurations as explored by the tuning plugin

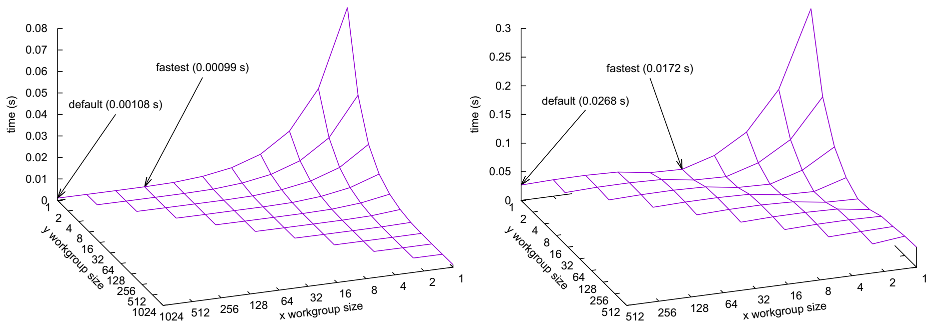


Fig. 18 Execution times of Coulomb summation benchmark on NVIDIA K20m (*left*) and Intel HD 4000 (*right*) for different NDRange configurations as explored by the tuning plugin

of scenarios to explore is given by the sum of kernel-specific scenarios, which in most cases will be much smaller. If the kernels in an application are independent of each other, the local tuning strategy usually will deliver the same results as the global strategy.

Both strategies rely on Periscope's OpenCL performance analysis facilities to obtain the execution time of the individual kernels and the whole application. The OpenCL NDRange plugin currently can use either the exhaustive search strategy or the individual search strategy.

4.4.3 Evaluation

Evaluation of the NDRange tuning with the OpenCL plugin has been performed with the Needleman-Wunsch benchmark from Rodinia Benchmark Suite (using the dataset with 2880 element-long sequences) and a Direct Coulomb summation benchmark (with grid size 1024 × 1024). The plugin was tested using the exhaustive search as well as the individual search strategy.

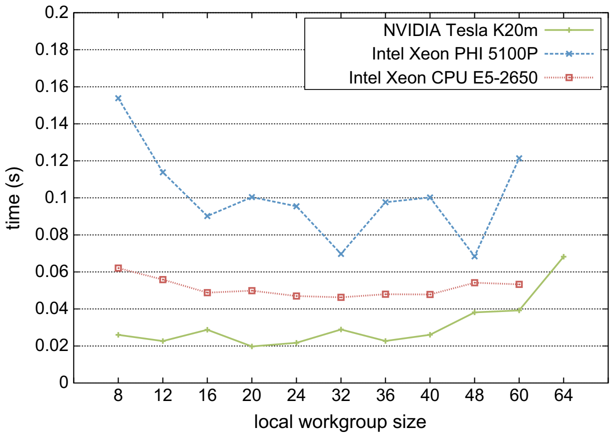


Fig. 19 Execution times of the Needleman-Wunsch application on different OpenCL devices with different local workgroup sizes

The execution times obtained for the two benchmarks on different architectures with different NDRange configurations are shown in Figs. 17, 18 and 19. The results show a significant performance impact of the NDRange configuration and a clear dependence on the target architecture (Bajrovic et al. 2016). We observed improvements of up to 56% for the Direct Coulomb summation, and up to 30% for the Needleman-Wunsch application with respect to default configurations.

5 Conclusion and future work

The main goal of the European AutoTune project has been to simplify the software optimization process by automating the tuning of software for HPC systems. The project has developed the Periscope Tuning Framework (PTF), an online automatic tuning framework, which combines both performance analysis and performance tuning with respect to the myriad of tuning parameters available to today's software developer on modern HPC systems.

PTF is an open and extensible framework that supports tuning of different performance aspects of parallel applications through a set of tuning plugins. To facilitate the development of tuning plugins, the framework provides common infrastructure components (*Analysis*, *Search Algorithms*, *Experiments Engine* and *Monitor*) and defines a flexible tuning model for steering the execution of individual plugins. Each of the associated tuning plugins focuses on one or more specific performance aspects and relies on codified expert knowledge to steer automatic performance analysis and tuning.

In this paper, we presented the Compiler Flag Selection (CFS) plugin for finding the best selection of compiler flags including support for OpenCL compilers, the MPI parameters plugin for tuning parameters of the MPI library, the DVFS plugin for optimizing power and energy consumption of applications through selective dynamic voltage and frequency scaling, and the OpenCL plugin for optimizing the organization of multi-threaded execution of OpenCL-based applications on heterogeneous manycore architectures.

As reported in this paper, all of the applications tuned with PTF exhibit improved performance as well as improved energy efficiency. Evaluation of PTF also demonstrates that using the tuning plugin achieves not only significant performance improvements of the applications but also a significant increase of the productivity of application tuning as compared to manual tuning.

Future work on PTF includes the development of meta-plugins that combine individual plugins (Gerndt et al. 2015). Currently two meta-plugins are available. The first one is a static sequence of plugins while the second one is adaptive. The current adaptive version takes simple characteristics of the application to decide whether to apply a plugin. A future version will rely on a machine learning approach based on a sensitivity analysis of an application for a certain plugin.

In addition, this work is currently being extended towards dynamic tuning of the energy efficiency of applications in the European Horizon 2020 project READEX (Runtime Exploitation of application Dynamism for energy-effcient eXascale computing - www.readex.eu). PTF is used to precompute a tuning model that is then forwarded to a runtime tuning library that switches hardware configurations dynamically (Oleynik et al. 2015).

Acknowledgments This work was supported by the European Commission FP7 project AutoTune under grant no. 288038.

References

Bajrovic, E., Mijakovic, R., Dokulil, J., Benkner, S., & Gerndt, M. (2016). Tuning OpenCL applications with the periscope tuning framework. In *Hawaii international conference on system sciences*. IEEE.

Balaprakash, P., Tiwari, A., & Wild, S.M. (2013). Multi-objective optimization of hpc kernels for performance, power, and energy. In *4th international workshop on performance modeling, benchmarking, and simulation of HPC systems (PMBS12), 11/2013*.

Benedict, S., Petkov, V., & Gerndt, M. (2010). Periscope: An online-based distributed performance analysis tool. In Müller, M.S., Resch, M.M., Schulz, A., & Nagel, W.E. (Eds.) *Tools for high performance computing 2009* (pp. 1–16). Berlin Heidelberg: Springer.

Bruel, P., Gonzalez, M., & Goldman, A. (2015). Autotuning gpu compiler parameters using opentuner. *XXII Symposium of Systems of High Performance Computing*.

Buck, B., & Hollingsworth, J.K. (2000). An api for runtime code patching. *International Journal of High Performance Computing Applications*, *14*(4), 317–329.

CESAR. Proxy-apps. https://cesar.mcs.anl.gov/content/software.

Chen, C., Chame, J., & Hall, M. (2008). Chill: A framework for composing high-level loop transformations. Technical report University of Southern California.

Chung, I.-H., & Hollingsworth, J.K. (2004). Using information from prior runs to improve automated tuning systems. In *Proceedings of the 2004 ACM/IEEE conference on supercomputing, SC '04* (p. 30). Washington: IEEE Computer Society.

CORAL. benchmarks. https://asc.llnl.gov/coral-benchmarks.

Costa, G., Jorba, J., Morajko, A., Margalef, T., & Luque, E. (2008). Performance models for dynamic tuning of parallel applications on computational grids. In *2008 IEEE international conference on cluster computing* (pp. 376–385).

Costa, G., Sikora, A., Jorba, J., & Gmate, T.M. (2014). Dynamic tuning of parallel applications in grid environment. *Journal of Grid Computing*, *12*(2), 371–398.

Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., & Yelick, K. (2008). Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on supercomputing, SC '08* (pp. 4:1–4:12). Piscataway: IEEE Press.

Demmel, J., Dongarra, J., Eijkhout, V., Fuentes, E., Petitet, A., Vuduc, R., Whaley, R.C., & Yelick, K. (2005). Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, *93*(2), 293–312.

Frigo, M., & Johnson, S.G. (1998). Fftw: an adaptive software architecture for the fft. In *Proceedings of the 1998 IEEE international conference on acoustics, speech and signal processing, 1998* (Vol. 3, pp. 1381–1384).

Frigo, M., & Johnson, S.G. (2005). The design and implementation of fftw3. *Proceedings of the IEEE*, *93*(2), 216–231.

Fursin, G., Kashnikov, Y., Memon, A.W., Chamski, 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., & O'Boyle, M. (2011). Milepost gcc Machine learning enabled self-tuning compiler. *International Journal of Parallel Programming*, *39*(3), 296–327.

Gerndt, M., César, E., & Benkner, S. (eds.) (2015). *Automatic tuning of HPC applications - the periscope tuning framework*. Shaker Verlag.

Haneda, M., Knijnenburg, P.M.W., & Wijshoff, H.A.G. (2005). Automatic selection of compiler options using non-parametric inferential statistics. In *14th International conference on parallel architectures and compilation techniques, 2005. PACT 2005* (pp. 123–132).

Kukkonen, S., & Lampinen, J. (2005). Gde3: The third evolution step of generalized differential evolution. In *The 2005 IEEE congress on evolutionary computation, 2005* (Vol. 1, pp. 443–450). IEEE.

Leather, H., Bonilla, E., & O'Boyle, M. (2009). Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th Annual IEEE/ACM international symposium on code generation and optimization, CGO '09* (pp. 81–91). Washington: IEEE Computer Society.

Morajko, A., Caymes-Scutari, P., Margalef, T., & Mate, E.Luque. (2007). Monitoring, analysis and tuning environment for parallel/distributed applications. *Concurrency and Computation: Practice and Experience*, *19*(11), 1517–1531.

Morajko, A., César, E., Caymes-Scutari, P., Margalef, T., Sorribes, J., & Luque, E. (2005). Automatic tuning of Master/Worker applications. In *Proceedings of Euro-Par 2005 parallel processing: 11th international euro-par conference* (pp. 95–103).

Navarette, C., Guillen, C., Hesse, W., & Brehm, M. (2014). Autotuning the energy consumption. In Bader, M. et al. (Eds.) *Parallel computing accelerating computational science and engineering*. IOS Press.

Nelson, Y.L., Bansal, B., Hall, M., Nakano, A., & Lerman, K. (2008). Model-guided performance tuning of parameter values A case study with molecular dynamics visualization. In *IEEE international symposium on parallel and distributed processing, 2008. IPDPS 2008* (pp. 1–8).

Oleynik, Y., Gerndt, M., Schuchart, J., Kjeldsberg, P.G., & Nagel, W.E. (2015). Run-time exploitation of application dynamism for energy-efficient exascale computing (READEX). In *IEEE 18th international conference on computational science and engineering (CSE), 2015* (pp. 347–350). IEEE.

OpenCL SDK User-guide. https://software.intel.com/sites/default/files/m/e/7/0/3/1/33857-Intel_28R_29_OpenCL_SDK_User_Guide.eps.

Pan, Z., & Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the international symposium on code generation and optimization, CGO '06* (pp. 319–332). Washington: IEEE Computer Society.

Püschel, M., Moura, J.M.F., Singer, B., Xiong, J., Johnson, J., Padua, D., Veloso, M., & Johnson, R.W. (2004). Spiral: a generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, *18*(1), 21–45.

Ravipati, G., Bernat, A.R., Miller, B.P., & Hollingsworth, J.K. (2007). *Towards the deconstruction of dyninst*. Technical report. University of Wisconsin.

Ribler, R.L., Simitci, H., & Reed, D.A. (2001). The autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, *18*(1), 175–187.

Ribler, R.L., Vetter, J.S., Simitci, H., & Reed, D.A. (1998). Autopilot: adaptive control of distributed applications. In *Proceedings of the 7th international symposium on high performance distributed computing, 1998* (pp. 172–179).

Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.-K., & Leiserson, C.E. (2011). The pochoir stencil compiler. In *Proceedings of the 23rd annual ACM symposium on parallelism in algorithms and architectures, SPAA '11* (pp. 117–128). New York: ACM.

Tiwari, A., Chen, C., Chame, J., Hall, M., & Hollingsworth, J.K. (2009). A scalable auto-tuning framework for compiler optimization. In *IEEE International symposium on parallel distributed processing, 2009. IPDPS 2009* (pp. 1–12).

Tiwari, A., & Hollingsworth, J.K. (2011). Online adaptive code generation and tuning. In *2011 IEEE international parallel distributed processing symposium (IPDPS)* (pp. 879–892).

Ţăpuş, C., Chung, I.-H., & Hollingsworth, J.K. (2002). Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on supercomputing, SC '02* (pp. 1–11). Los Alamitos: IEEE Computer Society Press.

The LLVM Compiler Infrastructure. http://llvm.org/.

Triantafyllis, S., Vachharajani, M., Vachharajani, N., & August, D.I. (2003). Compiler optimization-space exploration. In *Proceedings of the international symposium on code generation and optimization: feedback-directed and runtime optimization, CGO '03* (pp. 204–215). Washington: IEEE Computer Society.

Vuduc, R., Demmel, J.W., & Yelick, K.A. (2005). Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, *16*(1), 521.

Whaley, R.C., Petitet, A., & Dongarra, J.J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, *27*(12), 3–35. New Trends in High Performance Computing.

Williams, S., Waterman, A., & Patterson, D. (2009). Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, *52*(4), 65–76.

X-TUNE. Autotuning for exascale: self-tuning software to manage heterogeneity. http://ctop.cs.utah.edu/x-tune/.

Xiujuan, L., & Zhongke, S. (2004). Overview of multi-objective optimization methods. *Journal of Systems Engineering and Electronics*, *15*(2), 142–146.

Michael Gerndt He received his Ph.D. in Computer Science in 1989 from the University of Bonn. He developed SUPERB the first automatic parallelizer for distributed memory parallel machines. For 2 years, in 1990 and 1991, he held a postdoc position at the University of Vienna and joined Research Centre Juelich in 1992 where he concentrated on programming and implementation issues of shared virtual memory systems. This research led to his habilitation in 1998 at the Technische Universität München (TUM). Since 2000, he is a professor for architecture of parallel and distributed systems at TUM. His current research focuses on programming models and tools for scalable parallel architectures.

Siegfried Benkner He is a professor of Computer Science and the head of the Scientific Computing Research Group at the University of Vienna, Austria. His research interests include parallel and distributed computing, service-oriented software architectures, Grid and Cloud computing as well as languages, compilers and runtime systems for parallel and distributed systems. A major current research focus is on programming and runtime support for heterogeneous many-core systems.

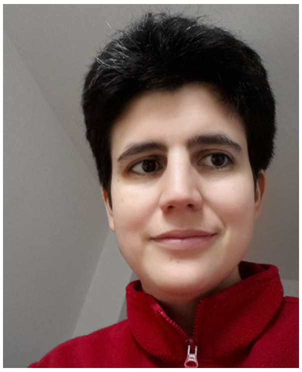
He coordinated the European PEPPHER Project (Programmability and Performance Portability of Heterogeneous Many-Core Systems) and was involved in the European Autotune project (Automatic Online Tuning). He helped organizing several international conferences and was program chair of ICPP 2009 and EuroMPI 2012. Siegfried Benkner has published more than 100 peer-reviewed publications and is a member of the ACM, the IEEE, and the HiPEAC Network of Excellence.

Eduardo César He received his BS degree in computer science in 1992 from Universidad Simón Bolívar. He received his MSc in computer science in 1994, and in 2006 his Ph.D. in computer science, both from Universitat Autònoma de Barcelona. Since 1998, his research has been related to parallel and distributed computing. He is currently involved in national projects in Spain and participated in the European project Autotune. His main interests focus on parallel applications and automatic performance analysis and tuning. He has been involved in the definition of performance models for automatic and dynamic performance tuning in cluster environments.

Carmen Navarrete She studied Computer Science and Engineering and received her Ph.D. in Computing and Telecommunication engineering in 2011 with her dissertation about automatic reconfiguration of farms of computers in HPC. Currently, she is working at the Leibniz Computer Center (LRZ) of the Bavarian Academy of Science (BAdW) as a researcher in the application support group. Her research topics include energy mathematical models, energy efficiency of applications in HPC and performance monitoring of applications.

Enes Bajrovic He received his B.Sc. and M.Sc. degrees in computer science from the University of Vienna, Austria. He is currently with the Research Group Scientific Computing at the University of Vienna. His recent research activities involve programming models and runtime systems for parallel and heterogeneous computing, as well as automatic performance tuning.

Jiri Dokulil He is a postdoctoral researcher at the University of Vienna, Austria. He received his M.Sc. and Ph.D. in computer science from the Charles University in Prague, Czech Republic. His research interests are in practical aspects of parallel and distributed computing, focusing on task-based runtime systems, programming models, and dynamic adaptation.



Carla Guillén She is part of the Application Support Group at the Leibniz Supercomputing Centre of the Bavarian Academy of Sciences. Her focus is on application and system-wide performance of high performance computers. Carla received a Doctor rer. Nat. in informatics from the Technische Universität München. Her areas of interest include energy optimization, performance monitoring, performance analysis and tuning in the context of high performance computing.

Robert Mijakovic He received the diploma engineer degree (Dipl.-Ing) in Electrical Engineering in 2008 from the University of Osijek, Osijek, Croatia. Robert's diploma thesis was carried out at the Faculty of Electrical Engineering, University of Osijek. His thesis was on Parallel Image Processing using FPGA systems.

In 2011, he joined the Institute for Informatics, Technische Universität München, Munich, Germany, as a research assistant and a Ph.D. student. His current research focuses on automatic performance analysis and tuning tools, programming models and machine learning with application to automatic analysis and tuning.

Anna Sikora She received her BS degree in computer science in 1999 from Technical University of Wroclaw (Poland). She received her MSc in computer science in 2001 and in 2004 the Ph.D. in computer science, both from Universitat Autònoma de Barcelona (Spain). Since 1999, her research has been related to parallel and distributed computing. She is currently involved in national projects in Spain, and participated in the European project Autotune. Her main interests focus on high performance parallel applications, automatic performance analysis and dynamic tuning. She has been involved in programming tools for automatic and dynamic performance tuning in cluster and Grid environments.