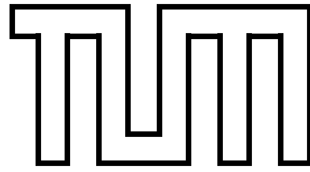


FAKULTÄT FÜR INFORMATIK

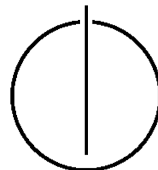
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

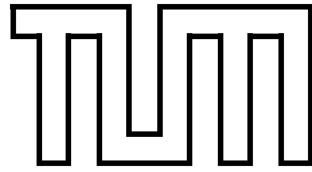
Dissertation

Automatic Online Tuning of HPC Applications

Robert Mijaković

Technische Universität München





FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Rechnertechnik und Rechnerorganisation

Automatic Online Tuning of HPC Applications

Robert Mijaković

Vollständiger Abdruck der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Thomas Huckle

Prüfer der Dissertation:

1. Prof. Dr. Hans Michael Gerndt
2. Prof. Dr. Siegfried Benkner, Universität Wien

Die Dissertation wurde am 20.11.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 06.12.2017 angenommen.

Ich versichere, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15.11.2017 Robert Mijaković

Abstract

Performance tuning of scientific codes often requires tuning many different aspects like vectorization, OpenMP synchronization, MPI communication, load balancing. The Periscope Tuning Framework (PTF), an online automatic tuning framework, relies on a flexible plugin mechanism providing tuning plugins for different tuning aspects.

To reach the exascale level within given energy constraints, system vendors rely on the wider range of more energy-efficient accelerators and manycores. This work focuses on enabling tuning plugins for such state-of-the-art HPC architectures and on their combination by automatically selecting plugins based on the prediction of their tuning potential. Two special tuning plugins were designed and implemented in this thesis that supports tuning of OpenCL codes based on selecting the best combination of flags for the offline compilation of kernels as well as on the investigation of application level tuning parameters specified by the user.

The thesis also presents the concept of meta-plugins that combines individual tuning plugins. Since each plugin can take considerable execution time for testing the various combination of the tuning parameters, it is desirable to automatically predict the tuning potential of plugins for programs before their application. We developed a generic automatic prediction mechanism based on machine learning for this purpose. This thesis demonstrates this technique in the context of the Compiler Flags Selection and the MPI Parameters plugin, which tune the flags of the compiler and the parameters of the MPI library, respectively. When the tuning process finishes, each tuning plugin generates a tuning advice that is automatically applied either before and during the runtime of the application.

Acknowledgments

Coming so far would not be possible without people who were there at right moments of my life. They provided me with support, love, and guidance.

First of all, I wish to thank my supervisor, Prof. Dr. Michael Gerndt, for his guidance, the advice, support, and remarkable patience he gave me throughout my PhD. Without his guidance and the chance he gave me by working at the chair this work would not be possible. His humbleness combined with exceptional intellectual power and immense knowledge was always a great motivation to continue my work.

Besides my supervisor, I would also like to tank my second supervisor, Prof. Dr. Siegfried Benkner, for accepting to review my thesis and giving insightful comments and suggestions.

Moreover, I would also like to thank Prof. Dr. Arndt Bode, the leader of the Chair of Computer Architecture (LRR) at Technische Universität München for the great working environment he created and the funding support in the final part of my work.

I would also like to thank my office colleague, Isaías Alberto Comprés Ureña, and Andreas Wilhelm for interesting discussions. Furthermore, I would like to express my gratefulness to my friends and colleagues at LRR for great moments I had on the chair.

Also, I would like to express the deepest gratitude to Elvira and Stjepan Vrečko for interesting moments of practicing math and the support.

Last but not least, I wish to thank my family for all the support, encouragement, sacrifices, and patience.

Thank you all.

Robert Mijaković
Munich, Germany
2017

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Contribution of This Work . . . 3

1.3 Structure of This Thesis . . . 5

2 Related Work **7**

2.1 Introduction . . . 7

2.2 Comparative Methods for Application Characterization . . . 8

2.2.1 White-Box Signatures . . . 9

2.2.2 Black-Box Signatures . . . 12

2.2.3 Applications of the Comparative Methods . . . 14

2.3 Tuning Tools . . . 17

2.3.1 Domain-Specific Automatic Tuning Tools . . . 20

2.3.2 Iterative Compilation . . . 21

2.3.3 Application-level Parameter Optimizations . . . 22

2.3.4 Combined Optimizations Approach . . . 23

2.4 Summary . . . 24

3 The Periscope Tuning Framework **27**

3.1 Introduction . . . 27

3.2 Performance Analysis . . . 30

3.2.1 Single Core Analysis . . . 30

3.2.2 OpenMP Analysis . . . 31

3.2.3 MPI Analysis . . . 31

3.2.4 Configurable Analysis . . . 31
3.3 Tuning Concepts . . . 32
3.3.1 Terminology . . . 32
3.3.2 Tuning Model . . . 32
3.3.3 Tuning Objectives . . . 34
3.3.4 Tuning Scenario . . . 35
3.3.5 Tuning Actions . . . 39
3.4 Search Algorithms . . . 40
3.4.1 Exhaustive Search . . . 41
3.4.2 Individual Search . . . 41
3.5 Scenario Execution Infrastructure . . . 43
3.5.1 Tuning Plugin Interface (TPI) Sequence . . . 43
3.5.2 Tuning Step Pre-Analysis . . . 46
3.5.3 Scenario Analysis . . . 47
3.6 Results Pools . . . 47
3.7 Advice Format . . . 48
3.8 Summary and Current Limitations . . . 50

4 Tuning Plugins **51**
4.1 Compiler Flag Selection . . . 51
4.1.1 Introduction . . . 51
4.1.2 Analysis . . . 52
4.1.3 Tuning . . . 54
4.1.4 Complete Tuning Flow . . . 57
4.1.5 Summary . . . 61
4.2 Compiler Flags Selection for OpenCL . . . 61
4.2.1 Introduction . . . 61
4.2.2 Analysis . . . 62
4.2.3 Tuning . . . 65
4.2.4 Complete Tuning Flow . . . 68
4.3 MPI Parameters . . . 70
4.3.1 Introduction . . . 70

4.3.2 Analysis 71
4.3.3 Tuning 73
4.3.4 Complete Tuning Flow 77
4.3.5 Summary 79
4.4 User Defined Parameters 79
4.4.1 Introduction 79
4.4.2 Analysis 80
4.4.3 Tuning 80
4.4.4 Complete Tuning Flow 81

5 Meta-plugins **83**
5.1 Introduction 83
5.2 Fixed Sequence 85
5.2.1 Introduction 85
5.2.2 Tuning 85
5.2.3 Complete Tuning Flow 85
5.3 Adaptive Sequence with Analytical Model 87
5.3.1 Introduction 87
5.3.2 Analysis 88
5.3.3 Tuning 89
5.3.4 Complete Tuning Flow 89
5.4 Adaptive Sequence with Predictive Model 92
5.4.1 Overview 92
5.4.2 Design 93
5.4.3 Implementation 104
5.4.4 Tuning 111
5.4.5 Complete Tuning Flow 113

6 Automatically Applied Tuning Advice **117**
6.1 SCORE-P Filter File 118

7 Evaluation **119**

7.1 Compiler Flags Selection for OpenCL 119
7.1.1 Experimental Setup 119
7.1.2 Rodinia Benchmark Suite 119
7.2 Meta-plugins 123
7.2.1 Experimental Setup 124
7.2.2 Benchmark Suites and Proxy Applications 124
7.2.3 Kernels and Parameter Settings 127
7.2.4 Evaluation Methodology 128
7.2.5 Compiler Flags Selection 131
7.2.6 User's Time Saved by Tuning Runs for Meta-plugin 145
7.2.7 MPI Parameters Plugin 147
7.2.8 User's Time Saved by Tuning Runs for Meta-plugin 160

8 Summary and Outlook **163**

8.1 Future Work 166
8.1.1 Central Optimization Repository 166

List of Figures

2.1 Cycle of the Tuning Process. 20

3.1 Architecture of the Periscope Tuning Framework. 29
3.2 The tuning model of PTF. 33
3.3 Execution of tuning scenarios. 37
3.4 The TPI call sequence. 44
3.5 Specification of the Tuning Plugin Interface functions. 45
3.6 Example of a PTF advice for the MPI Parameters plugin in the XML advice format. 49

4.1 Example CFS configuration file. 55
4.2 The machine learning-based search strategy uses historical performance data to guide the selection of good candidate scenarios. Historical performance data in the tuning database which is then used to configure probabilistic random search with probability functions for each tuning parameter. 58
4.3 Tuning approach of the CFS plugin. 58
4.4 Example CFS for OpenCL configuration file. 66
4.5 Offline compilation flow. 67
4.6 Execution time of FSSIM for different values of the eager limit and memory buffers parameters. 75
4.7 Hypothetical frequencies of occurring in certain ranges of message sizes from the `EagerLimitDependency` property. 76
4.8 Example MPI Parameters configuration file. 77
4.9 Tuning approach of the MPI Parameters plugin. 78
4.10 Example of scenario reordering scheme. 81
4.11 Example User plugin configuration file. 82

5.1 The tuning flow of the Adaptive Sequence meta-plugins. 84
5.2 Example Fixed Sequence configuration file. 85
5.3 The tuning flow of the Fixed Sequence meta-plugin. 86
5.4 Example ASAM configuration file. 89
5.5 The tuning flow of the ASAM meta-plugin. 90
5.6 The simplified overview of the approach. 94
5.7 The explained variance with clusters with respect to the number of clusters. 99
5.8 Classification of an object with k-Nearest Neighbors classifier with two classes where k is set to 5. 102
5.9 SVM classifier with two classes. 103
5.10 Example Adaptive Sequence with Predictive Model configuration file. 113
5.11 The tuning flow of the meta-plugin. 114

6.1 Example CFS pre-execution tuning advice. 117
6.2 Example Score-P filter file. 118

7.1 The execution time of LavaMD on different devices using three grids: (a) 5, (b) 10, (c) 20. 121
7.2 The execution time of PathFinder on different devices using three grids with different widths: (a) 100K, (b) 200K, (c) 400K. 122
7.3 The execution time of HotSpot on Intel CPU using three grids: (a) 64 x 64, (b) 512 x 512, (c) 1024 x 1024. 123
7.4 The confusion matrix reports the number of true positives, true negatives, false positives, and true negatives. 130
7.5 The speedup achieved on various benchmarks with CFS plugin relative to scenario 24. 133
7.6 The speedup achieved on various benchmarks with CFS plugin relative to scenario 96. 134
7.7 The speedup achieved on various benchmarks with CFS plugin relative to scenario 6. 135
7.8 The cost as the function of the number of signature scenarios represented as clusters. 138
7.9 The cost as the function of the number of signature scenarios. 138
7.10 The execution time of various benchmarks with MPI Parameters plugin. . 149
7.11 The execution time of various benchmarks with MPI Parameters plugin. . 150

7.12 The execution time of various benchmarks with MPI Parameters plugin. . 151

List of Tables

7.1 Compiler flags combinations used in figs. 7.1 to 7.3 120

7.2 Kernel parameter settings and their ranges 128

7.3 Intel 16 compiler parameters and their respective values used for evaluation 131

7.4 The speedup achieved with Compiler Flags Selection plugin on dataset relative to scenario 96. 137

7.5 Relation between the number of scenarios and classification quality measures for speedup relative to scenario 6 for the linear SVM kernel with threshold 1.23941. Signature determined with k-medoids. 139

7.6 Settings used by the meta-plugin for CFS plugin and the distribution of expected meta-plugin's decisions (don't tune, tune) for benchmarks. 139

7.7 Comparison of SVM kernels used for classification with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with information gain. Values represent minimum/average/maximum. 140

7.8 Comparison of SVM kernels used for classification with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum. 141

7.9 Comparison of information gain and k-medoids for various SVM kernels used for classification with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with two thresholds (6/46, 26/26). . . . 142

7.10 Comparison of SVM kernels used for regression with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with information gain. Values represent minimum/average/maximum. 143

7.11 Comparison of SVM kernels used for regression with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum. 144

7.12 Comparison of information gain and k-medoids for various SVM kernels used for regression with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with two thresholds (6/46, 26/26). 145

7.13 Comparison of classification and regression-based classification for various SVM kernels with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with thresholds (6/46, 26/26). 146

7.14 Settings used by the meta-plugin, the best kernel, saved time, wasted time, wrongly and correctly predicted time. Total time used to execute all scenarios with all programs takes 759.36 h. 147

7.15 MPI parameters and their respective values used for evaluation 147

7.16 The speedup achieved with Compiler Flags Selection plugin on dataset relative to scenario 7. 152

7.17 Settings used by the meta-plugin for MPI Parameters plugin and the distribution of expected meta-plugin's decisions (don't tune, tune) for benchmarks. 153

7.18 Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with information gain. Values represent minimum/average/-maximum. 154

7.19 Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum. 156

7.20 Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum. 156

7.21 Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with information gain. Values represent minimum/average/-maximum. 158

7.22 Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum. 159

7.23 Comparison of information gain and k-medoids for various SVM kernels with 300 scenarios where the speedup is calculated relative to three different scenarios with two thresholds (17/18, 28/7). 160

7.24 Comparison of classification and regression-based classification for various SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with two thresholds (17/18, 28/7). 160

7.25 Settings used by the meta-plugin, the best kernel, saved time, wasted time, wrongly and correctly predicted time. Total time used to execute all scenarios with all programs takes 332.39 h. 161

8.1 Comparison of Classification and Regression-based Classification performance for both tuning plugins. 164

8.2 Comparison of Regression performance for both tuning plugins. 165

Chapter 1

Introduction

1.1 Motivation

Large-scale machines help to solve some of the greatest challenge humanity faces today in science and engineering. The next grand challenge for large-scale is to reach exascale-level performance in the 2019-2022 timeframe. Exascale computing will enable the solution of vastly more accurate predictive models and the analysis of massive quantities of data, production quantum advances in areas of science and technology. To reach the exascale level three challenges to be resolved are, reducing power requirements, coping with runtime errors and exploiting massive parallelism. Based on current technology, scaling today's systems to an exascale level would consume more than 100 megawatts of power.In the most optimistic scenario, based on Green500, reaching exascale-level performance in the given timeframe with only homogeneous machines seems not to be possible. As of June 2017, 88 systems in Top500 systems [10] are heterogeneous. They account for 268 (rpeak - rmax = 169) petaflops, over one-fifth of the list's total FLOPS. NVIDIA Tesla GPU-based supercomputer comprises most (71) of these systems. Additionally, homogeneous systems driven by manycores seems to emerge. Most of manycore systems are driven by Intel Xeon Phi, but others are also emerging.

Writing an HPC application requires extensive knowledge of particular scientific subjects and good knowledge of the underlying system. To use system resources efficiently the application has to be carefully written and tuned which is time-consuming. Having a well-optimized application is a win-win situation for both, the scientist writing the application and the system provider. The scientist can have more simulation results or more detailed simulation results within the reasonable time. The system provider can have more efficient resource usage which results in a higher utilization of the system and cost savings.

Performance tuning often relies on an empirical evaluation of performance on an HPC system. Taking into account the parallelism, heterogeneity and the complexity of the supercomputers, the amount of information the scientist has to be aware becomes over-

whelming. To support the effort of both, reduce the time necessary for performance tuning and improve tuning results *tools-aided performance analysis and tuning* is a must. Tools-aided performance analysis and tuning allow the scientist to concentrate more on the scientific subjects.

Performance analysis and tuning typically involve instrumentation, measurement collection, analysis, tuning and applying tuning recommendations. The application being analyzed is augmented by inserted probe-functions during the instrumentation capable of registering relevant execution events. During the runtime, the events generated by the probe-functions are processed by the measurement tool (monitoring library) which results in raw performance measurements. The knowledge about optimization potential is then obtained during analysis phase either by interpretation of the results by the user based on visualization or through an automatic evaluation. Tools capable of automatic tuning interpret the analysis results and selects a set of potentially fruitful optimizations. Selected optimizations are automatically evaluated during tuning phase and the user manually applies tuning recommendations.

In this thesis, we present concepts and their implementation in the context of the Periscope Tuning Framework (PTF) allowing valuable insight in performance tuning of heterogeneous machines and at the same time mitigation of the following challenges. The challenges addressed in this thesis are (1) tuning of different aspects of heterogeneous systems, (2) automatic tuning of many heterogeneous aspects and (3) automatic aspect tuning decision with expert knowledge criteria and machine learning.

The context of the work is PTF, an automatic tuning framework, developed at the Technische Universität München during the Autotune project. The framework extends Periscope, an online performance analysis tool, with fully automatic tuning support and brings several tuning plugins which tune individual aspects of HPC applications. Tuning plugins provide codified expert knowledge about the tuning of many different aspects of the system like single-core, MPI, accelerator, memory access.

To follow the trends in HPC and support automatic tuning of heterogeneous code we have developed a set of PTF plugins:

- Compiler Flags Selection
- MPI Parameters
- MPI Power Capping
- MPI Tool Interface
- MPI-Input/Output
- OpenMP Power Capping
- Parallel Patterns (Pipeline Coordination Layer)

- Dynamic-Voltage And Frequency Scaling
- User Definable Parameter Tuning

Getting the best performance of an HPC application requires tuning many aspects in consecutive order. The user starts the coding process by selecting the right algorithm for the problem the application solves and optimize its work on the system. Once the application is written, the developer starts with the tuning process by setting a set of hypotheses for performance problems in his code. The problem hypotheses are usually problems like the memory access pattern or the optimal combination of compiler optimizations, load-imbalances, mapping of tasks/threads onto cores, interprocess communication, etc. The user now instruments his code, runs a set of experiments, collects the runtime performance data and proves some of his hypotheses. Then, he refines his hypothesis and repeats the tuning process until he is satisfied with the results.

To replicate such approach we have developed a concept of meta-plugins that automates aspect optimization. The meta-plugin combines multiple component tuning plugins into a single plugin and acts as an intermediate between the frontend and the component plugins. It loads the component plugins, decides upon which plugins to run, in which order and drives their execution.

Optimizing most of these aspects often does not pay-off and performance improves only slightly. Therefore, we have developed automatic tuning decision of individual tuning aspects. Automatic tuning decision is done with two approaches, one where an expert is required to define criteria for each tuning aspect and one based on machine learning that does not require such knowledge. The machine learning decision is agnostic of tuning aspect and in general portable across systems.

1.2 Contribution of This Work

The work described in this thesis focus on the automatic tuning of single and multiple aspects of multicore- and manycore-based parallel systems ranging from parallel desktop systems with accelerators to petascale and future exascale HPC architectures.

Contributions presented in this thesis are:

1. New plugins targeting optimization of compilation process and user definable tuning
2. Meta-plugins that combine individual plugins
3. Automatic prediction of tuning benefit based on machine learning
4. Automatic application of tuning advice on production runs

Two tuning plugins were developed/extended to target the compilation process and user definable tuning of heterogeneous/manycore systems. The first one, Compiler Flags Selection, was extended to support tuning OpenCL kernels with the help of compiler flags presented to an OpenCL offline compiler. The second one, User Defined Parameters, allows the user to specify tuning parameters and the search algorithm finds the optimal combination of parameters. Although this plugin provides no codified expert knowledge it allows the user to cover multiple aspects or to evaluate ideas about new plugins before implementing them. Also, three metaplugins which allow several aspects to be tuned were developed. Individual aspects are executed as a user specified fixed sequence or as an adaptive sequence. The adaptive sequence is controlled in two ways based on performance properties of the program or by program signature evaluation.

The main contribution of this thesis is the characterization of the search space of a plugin and automatic decision about the necessity to tune a new program based on the characterization. The characterization is tuning aspect, e.g., compiler flags, and feature, e.g., cache size, agnostic. Therefore, the decision is made from the sensitivity of the program to the search space signature. The program signature identification and the tuning benefit prediction is based on machine learning. The program signature is extracted by evaluating a small set of experiments, signature scenarios, representing points in the search space. The signature scenarios characterize the application search space, reduces tuning time and concentrates it just to the aspects that are more fruitful.

After the tuning of codes is done a tuning advice gets generated in the sense of scripts to prepare the run and a runtime library. The scripts are used to compile the code with optimal compile-time parameters, to start the application with the optimal configuration of pre-execution parameters, e.g., the number of processes. The runtime library takes care that individual regions are run with optimal runtime parameters, e.g., the number of threads, optimal frequency or algorithm implementation. The optimal parameter configuration discovery is done by tuning plugins that tune individual aspects. The special care is taken on minimizing the overhead of the execution in production runs. For that purpose, we eliminate all the instrumentation not necessary for the run and consequently overhead.

In the context of this thesis these plugins were extended or written:

- Compiler Flags Selection for OpenCL
- User Defined Parameters
- Fixed Sequence
- Adaptive Sequence with Analytical Model
- Adaptive Sequence with Predictive Model

1.3 Structure of This Thesis

The rest of the thesis is composed as follows.

Chapter 2 presents the related work. Prior work on the characterization and performance modeling of mostly serial programs is discussed. Then work related to performance tuning is reviewed and the use of machine learning techniques for performance tuning.

Chapter 3 overviews the context of the study, the Periscope Tuning Framework. Here we explain important concepts of tuning with the framework and overall design.

Chapter 4 introduces new tuning plugins tuning many aspects of a heterogeneous system and the infrastructure extension necessary to support them.

Chapter 5 presents the design of a new plugin type which allows combining multiple tuning plugins into a single plugin. The plugins employed by this plugin are described in Chapter 4. In this chapter, we present the approach of automatic plugin decision, theoretical background necessary to understand the approach in-depth and the infrastructure extensions necessary to support the new plugin type.

Chapter 6 presents the automatically applied tuning advice generated for production runs. In production runs the application is started with the optimal combination of tuning parameters and their values. It configures the application before execution, for instance, compiles it with the best combination of compiler flags, but also during the runtime, e.g., changes the frequency of the CPU.

Chapter 7 gives the evaluation of the plugins developed in the context of this thesis. It covers a wide range of benchmark suites and proxy applications like Rodinia Benchmark Suite, CESAR benchmarks, CORAL, lanl-proxy-apps, llnl-proxy-apps, Mantevo and NPB suites.

Chapter 8 finally concludes this thesis, summarizes the contributions and discusses future work outlook.

Chapter 2

Related Work

2.1 Introduction

This chapter gives a short overview of the research relevant to this thesis. The first section describes the research done in the extraction of the program signature, search space characterization and performance modeling of programs. Section 2.3 provides an overview of tools that automate the search for the best combination of execution environment parameters and/or code transformations. Finally, Section 2.4 summarizes the research done and its connection to work done in this thesis.

The optimization engineer often considers a large number of parameters believed to influence the performance of program execution on an architecture. Each parameter brings a new dimension to the size of the search space which is the set of all possible points of an optimization problem and the search space grows exponentially. Evaluation of individual points/configurations might require a substantial amount of time as the execution of one time step of a simulation may take 5 to 10 minutes. Therefore, in the most practical cases exploration of such a huge search space by evaluating all points is impossible to be done exhaustively.

To evade the aforementioned problem scientists have developed methods that are capable of predicting the search space from results of its partial evaluation. The developed methods could be classified into two groups, analytic and comparative. Analytic methods are built from the observations made by the computer scientist to predict the performance of a program or a computer system with respect to various program or system parameters, e.g., static code structure or memory hierarchy, and system workloads, e.g., program inputs. Based on these observations, causality between the parameters and performance can be determined and expressed as an analytical model. Comparative methods analyze the programs, extract key characteristics from them and based on these characteristics group similar programs to make predictions for new programs belonging to the groups.

These methods are used for different purposes which partially overlap, while analytic

methods are used in (a) Performance tuning (b) Adaptive performance control (c) System design, the comparative evaluation methods are used for (a) System procurement (b) System classification/pruning (c) Benchmark suite reduction (d) Design space exploration.

2.2 Comparative Methods for Application Characterization

The main idea of comparative methods is to make predictions for new programs based on their similarity with other programs. To quantify the similarity between programs, properties relevant to program's execution and the system where the program is executed have to be identified. In performance engineering, a property of a program; e.g., different performance counters, work set size; or of a system, e.g., CPI (Clocks Per Instruction), memory hierarchy is a feature of the program and/or the system. Features of the program and/or the system form an n-dimensional vector of features represented with real numbers called signature. The similarity between two signatures is based on similarity measures, e.g., Euclidean distance between signatures. Programs whose signatures are similar form a group of similar programs in the process called program characterization.

The signature is considered homogeneous if its features measure the same quantity or heterogeneous if they measure different quantities. When the value of a feature cannot be directly associated with any property of a program or of a system it is considered to be agnostic of the program and/or system properties. Black-box signatures consists of features that does not expose interaction between inner components of the micro-architecture, the network and the program. On the other hand, if the signature consists of features directly associated with the property of a program or of a system it is considered to be white-box. For example, black-box signatures consist of quantities like execution times or energy-delay products, while white-box consists of measures like communication time, a number of cache misses, synchronization time, etc. Both, white-box and black-box signatures have their advantages and disadvantages. White-box signatures provide more in-depth knowledge about the characteristics of a program or of a system.

Depending when features are collected they can be grouped into two groups, static and dynamic. Features collected before the program is run are considered static, while features collected during the runtime are considered dynamic. A common source of static features are compilers while dynamic features are collected from the runtime system. These signatures can be gathered or extracted) from micro-architecture aware and/or program-inherent features.

The work on comparative methods started when Saavedra and Smith [75] proposed an analytic model which estimates execution time based on similar programs. To express similarity between programs they computed the Euclidean distance between their signatures. In the proposed model, program signatures are built from dynamic program-inherent fea-

tures of Fortran programs. The dynamic features used for characterization are operation mix, the number of function calls, the number of address computations, etc. The information collected about similarity is later used to identify the representative benchmark suite, i.e., benchmark suite reduction. Such a reduced benchmark suite requires much less time to evaluate a new architecture without introducing significant error to the performance estimation.

Eeckhout et al. [34] extended that work and proposed usage of micro-architecture aware features combined with program-inherent features. The micro-architecture aware features used for this characterization are cache-related events, branch prediction efficiency, and ILP. This approach was used to identify similarities across program-input pairs and to make performance predictions. Programs with similar signatures are grouped by cluster analysis, i.e., K-Nearest Neighbors (kNN) from raw features processed with the Principal Component Analysis (PCA). PCA is used to reduce the number of features that compose the signature without introducing significant error into predictions.

More detailed description of research done for comparative methods can be found in Section 2.2.3.

The next subsections provides an overview of research done for predictors that employ white-box and black-box signatures with their advantages and disadvantages.

2.2.1 White-Box Signatures

White-box signature consists of features that expose interaction between inner components of the micro-architecture, the network and the program. Such signatures allow building models which are easy to understand. White-box signatures are gathered from micro-architecture or program-inherent features, before, i.e., statically, or during, i.e., dynamically, the runtime of the application. Micro-architecture aware features are gathered statically from hardware design properties or dynamically from hardware performance counters. Program-inherent features are gathered from information derived by the compiler or during the execution of programs. Gathering both types of features is not free of costs and limitations but they are more informative compared to features that are agnostic of program or system properties. The cost come from the fact that programs from which they are gathered have to be instrumented.

Micro-architecture Features

Hardware design properties can be extracted from Register Transfer Level (RTL) descriptions and contain information like number, type, size and organization of hardware components [60, 32, 64]. Such features might be especially useful for prediction of performance of a new program on a new architecture. Hardware performance counters are built in modern CPUs to count hardware-related events. The events are accessed through

various programming interfaces, e.g., PAPI [66] or PERF [86].

Features collected from hardware performance counters suffer from many disadvantages. Hardware performance counters provide a small set, usually 4 to 8, of configurable counters which can be programmed to count a larger number, up to hundreds, of events. The Intel Haswell processors used for evaluation can measure 276 events with 11 counters (3 fixed function counters, 4 general purpose counters, and 4 RAPL counters) [26]. Additionally, measuring some events might make measurements of other events during the same run not possible. Therefore, measuring a larger number of features or mutually exclusive events might require multiple runs of the program. Additionally, hardware performance counters are not standardized and therefore micro-architectures from different manufacturers and even different microarchitecture revisions offer significantly different events.

Several research studies have proven that measurements should be repeated several times to reduce measurement inaccuracies [63, 67, 91]. Some counters count events for shared resources and it is hard to attribute them to individual resources [87, 43, 68]. There are counters stored in 32-bit MSR registers which require special permissions and are keen to overflow for some events, e.g., RAPL counters [68]. Model-based performance counters, e.g., RAPL counters, are updated in not very accurate intervals (jitter) with some minimal granularity which also makes them not very accurate [43]. Signatures built of architecture specific features suffer from not being portable across system configurations and have to be determined for every new architecture/search space which is tedious. Configuring and gathering performance counter measurements requires instrumentation which affects program's instruction mix which correlates in gathered hardware performance metrics [62]. Signature extracted from hardware performance counters is proposed by research [11, 22, 92, 16, 78, 89, 90, 65, 82].

Program-Inherent Features

Program-Inherent features are architecture-independent and include knowledge of programs and their execution. The execution of programs is not always possible, e.g., the system is not available, therefore scientists tried to identify useful static features that can give good predictions. Compilers are very attractive source of features as their collection is automated and they provide various statistics/meta-data about the source code, like the number of instructions of a certain type (instruction mix), static code features (lines of code) and dependencies between instructions but also non-standard features like Abstract Syntax Tree (AST) or Control Flow Graph (CFG) [71].

On the other hand, when the system is available and programs can be executed, more informative dynamic features can be collected. For instance, before the program is actually executed or simulated and the input is known we have no means to discover some of its features, e.g., dynamic call context. The dynamically collected features include working set size, dynamic call context, information like number of calls. Features collected from program-inherent metrics comes with the execution overhead and/or necessity for instru-

mentation [49, 59, 39, 41]. The disadvantage is that program-inherent features does not take into account effects that come from software layers like compiler, libraries, operating system, optimization drivers and run-time libraries that can potentially influence performance [28]. Research that employ signature extracted from program-inherent metrics [58, 49, 60, 39, 41, 71, 84].

Signature Identification

Selecting right features for the signature is an open problem and often very subjective. In the previous research, researchers have proposed different white-box signatures that consist from 1 [76] to 47 [49] different features. The number of potential features is very large, e.g., hardware performance counters offer up to several hundreds of features. Many of these features are considered irrelevant like a number of comments, redundant or might not perform well with the machine learning algorithm [54]. Therefore, some researchers have proposed automatic feature extraction [59] based on multivariate statistical techniques.

The most commonly used feature extraction technique is Principal Component Analysis (PCA) [34, 41] which converts a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. The resulting principal components, represented as linear combination of observation variables, are sorted by the amount of variance they contain, i.e., the first component contain the most with every succeeding containing less. Such property of the transformation provides means to reduce number of features to only those responsible for most of the variability.

The next popular technique helpful for understanding the complex nature of multivariate relationship is clustering analysis and assessing dimensionality. The analysis relies on similarities or distances between data points. Many clustering techniques have been proposed in the past like hierarchical methods and non-hierarchical methods. The hierarchical methods provide users with dendrograms to present overall cluster structure of the data. On the other hand, the non-hierarchical methods, e.g., K-means provides statistical technique to explain the importance of every feature by using F-ratio which can help when the number of clusters is not obvious. More information on clustering analysis techniques can be found in 5.4.2.

Factor Analysis is a method that identifies complex relationships between features without a priori assumptions about their relationships. The analysis groups highly correlated metrics into one group while separating uncorrelated ones into the other group. Both methods are based on second-order statistics (covariance).

Scatterplot/Correlation Matrix is used to visualize the relationship between different feature's variance. For k features, it consists of k rows and k columns describing relations between of feature i versus feature j for row and column.

Dong H. Ahn et al. [11] in their work compare several scalable multivariate analysis

techniques to identify white-box signature from three applications. The techniques they employ are clustering analysis, factor analysis, PCA and scatterplot/correlation matrix.

Lee et al. [60] built a regression modeling approach for microarchitectural performance and power prediction for various applications executing on any microprocessor configuration in a large microarchitectural design space. The signature is identified with cluster analysis and only one feature per cluster is selected to remove potentially redundant features. They have considered a wide-range of static architecture-aware features such as number of registers, number of reservation stations, different cache sizes, memory latencies, but also static program-inherent features such as cache access patterns, and branch patterns. Two predictors were built, one for prediction of architecture behavior and one for program behavior.

Yoo et al. [90] 2012 used decision tree algorithm, random forest, to identify performance problems based on a dynamic white-box signature in benchmarks. The system called Automated Diagnosis of Performance pathology system, analyzes and tries to discover patterns of hardware events and identify these patterns in applications. The existance or non-existance of 40 common performance problems is tested with 10 micro-benchmarks and three working sets. From these runs performance problems are identified and their dynamic micro-architecture signature is extracted. The signature is identified from a set of features with correlation-based feature selection technique. The decision tree algorithm, random forest, is then trained to identify performance problems from these signatures. Based on the detected performance problem the user is directed to the code location, e.g., function, and provided with the advice how to resolve it. The evaluation of the proposed system is done with SPEC CPU2006 and the PARSEC benchmarks.

2.2.2 Black-Box Signatures

Although, the traditional micro-architecture or program-inherent features are believed to be more informative [56] they are hard to identify, not portable, perturb execution, occasionally hard or impossible to gather. The features which have no such limitations form black-box signatures which capture only the causal relationship between input and output. Unlike micro-architecture aware signatures they are architecture independent and therefore portable between architectures. Characterization with black-box features is suitable for search spaces independent of what they represent.

The research of black-box signatures started with Dujmovic [33] who proposed benchmark suites that cover the program space well and do not have to be updated. He heavily criticized white-box signatures for being hard to identify, not portable and hard to be gathered. Some of these issues were addressed by interfaces like PAPI, but some are impossible to be solved, e.g., portability. To overcome limitations of white-box signatures he proposes black-box signatures, like throughput or execution time that characterize the benchmark suite on a given micro-architecture.

Piccart et al. [72] predict the performance of an application of interest based on its performance similarities with the industry-standard benchmarks on a limited number of predictive machines. To make predictions, tha application of interest does not have to be executed on the target machine. The approach is used for different purposes like system procurement, design space exploration and affinity toward task-to-device or task-to-core mapping.

Haneda et al. [44, 45, 46] proposed a statistical technique to prune the search space of compiler optimizations. The focus of their work is on the quantification of the interaction between compiler optimizations for different programs [44, 45] from the SPEC benchmark. In the later work [46], they focused also on the impact of compiler optimizations on different architectures and different input datasets. The signature they propose is black-box and derived from the effect of compiler optimizations on the program that is above some defined threshold. They initially developed a multi-step algorithm [44] but later employed non-parametric inferential statistics [45, 46], i.e., the Mann-Whitney test. Their algorithm, in the first step, identifies a set of optimizations that have large effect alone and sets of pairs of options that positively interact. In the next step, they iteratively add individual optimizations to identified sets as long as they perceive the positive effect of optimizations. In the last step, they run identified optimization sets on SPEC benchmark suite and select the settings with the best average improvement.

Park et al. [71] compared signatures extracted from architecture-aware features, program-inherent features and two black-box features, i.e., reactions and graph-based intermediate representation (IR), to predict the optimal combination of compiler optimizations. The big novelty of their works is the usage of the static signature which is not a feature vector but rather a topology, i.e., Control Flow Graph (CFG). The evaluation was done on two systems with two compilers, ICC and GCC, using iterative and non-iterative scenario with 30 applications from the PolyBench benchmark suite. This research has shown that for compiler optimizations static graph-based signature have outperformed static MilePost's program-inherent features, dynamic micro-architecture signature, and dynamic black-box reactions. Moreover, contrary to [56], micro-architecture aware signature on average slightly outperforms two and performs equally well for two configurations.

Cammarota [15] proposed the weighted predictor which combines predictions of two predictors. The first predictor predicts the performance of a program on a new system from performance achieved by the programs on other systems (program signature). The second one predicts the performance of a program on the existing system from the performance of other programs on that system (system signature). For both predictors, he used black-box signature and predicts the performance of sequential and parallel programs, SPEC CPU2006, SPEC OMP2012 and NAS Parallel Benchmarks, through similarity. For the construction of prediction models, they compare regression techniques like Linear Regression and Support Vector Regression (SVR). To validate the proposed predictor he uses k-fold cross-validation.

2.2.3 Applications of the Comparative Methods

Based on the performance study this approaches can be grouped into:

- System procurement [72, 84, 12]
- Tuning of compiler heuristics [16, 39],
- Estimation of power consumption [78, 22, 60], performance [60] or area [64]
- Identify performance problems [90] or potential faults [89]
- Tuning affinity toward task-to-device [41, 83] or task-to-core mapping [65, 92]
- Benchmark suite reduction [11, 30]
- Design space exploration [31, 32, 49]
- Adaptive performance control [82]

Ardalani et al. [12], in their work on **system procurement**, built a model called XAPP which predicts GPU performance of individual code sections from its CPU performance. The signature is identified from a wider set of features with logistic regression in an iterative approach. It iterates through the list of features to find the feature with the highest performance improvement of the predictor. This feature is added and the process is repeated until the improvement is lower than the defined threshold. The prediction is based on ensemble prediction technique called bootstrap aggregating which averages predictions from individual predictors. The evaluation was done on 24 benchmarks and the average relative error was 22.4%.

John Cavazos et al. [16], in their work on the **tuning of compiler heuristics**, built a logistic regression model called PCModel. The model classifies compiler optimizations, based on the dynamic micro-architecture aware signature, as good or bad and associate a probability with the decision. The training of PCModel is done offline with benchmarks from SPEC FP and MiBench by applying the same 500 random compiler optimizations to each program. Their speedup is stored together with the signature from 60 performance counters normalized with the total executed instructions. The model is evaluated with leave-one-out cross-validation. The signatures build from static program-inherent features achieved a speedup of 1.01 compared to 1.08 for dynamic micro-architecture aware features. Compared to Combined Elimination (CE), PC Model on average obtained higher speedup, 1.17 vs 1.09, with much fewer evaluations, 3 instead of 250. To reduce the number of features forming the signature they employ feature extraction technique that maximizes the Gaussian approximation which reduced signature to 15 relevant performance counters.

Fursin et al. [39] used experiences collected into a central database from many users continuously to select the most profitable compiler optimizations. The optimizations

are selected from statistically combined profile information generated by the gnu profiler like time spent in a function, the number of calls, etc. The correctness of optimized code is checked by comparing the output of the optimized and unoptimized code. The predictors are built based on the amount of information available for the program. For the programs that were often tuned, in the paper called stage 3, the prediction of the optimal combination of compiler optimizations is based on program-specific probability distribution. This predictor is capable of predicting behavior for different datasets. The prediction for known programs, but with few runs the predictions are made from averaged distribution of similar programs from stage 3. The predictions for an unknown program is based on unweighted averages of all predictions from stage 3. The similarity of programs is based on program reaction to optimization, i.e., speedup. The selection of the stages is based on scoring mechanism driven by the accuracy of prediction from all three stages. The tuning approach is built as a plugin to GCC that instruments the program and generates binaries with multiple optimized function clones.

Stockman et al. [78] in their work on **estimation of power consumption** from memory related performance events evaluated different machine learning techniques. The employed techniques are Artificial Neural Networks (ANNs), Support Vector Machines for Regression (SVR), and Genetic Algorithm (GA). The NN and SVR showed the very high accuracy of Mean Squared Error (MSE) 0.047 and 0.063 while GA had high accuracy with MSE 1.1.

Contreras et al. [22] developed a linear power model that estimates CPU and memory power consumption from dynamic micro-architecture aware signatures. The signature used for CPU power prediction consists of five features with high correlation to the power consumption.The power predictions of a CPU are the sum of static CPU power and feature values multiplied with their individual weights, extracted during the training phase. On the other side, memory power prediction signature consists of just two features due to lack of features related to memory. Accuracy is within 4% of the measured average CPU power consumption while memory power consumption error is up to 70%.

Hoste et al. [49] predicted the performance of programs based on program similarity from program-inherent characteristics. The raw dataset of collected characteristics is transformed using normalization, principal component analysis and genetic algorithm into what they call benchmark space. In the benchmarks space, the relative distance is a measure of the relative performance differences. The performance of the application of interest is predicted from weighted performance numbers of few neighboring applications. Characteristics of the applications are independent of micro-architecture but specific to a given ISA and a given compiler (instruction mix, ILP, register traffic, working set size, data stream strides, branch predictability, etc.).

Meeuws et al. [64] presented a Quipu modeling approach which predicts area and latency of the design written in HLL(C) from the entire design. They have evaluated various machine learning approaches like linear regression (LR), artificial neural networks(ANNs) and statistical analysis. The models are trained from software complexity metrics (SCM),

extracted from the ANSI-C sources, and hardware performance data, extracted from synthesized hardware, for different kernels in the training set.

Piccart et al. [72] predict the performance of an application of interest based on its performance similarities with the industry-standard benchmarks on a limited number of predictive machines. They employ two machine learning techniques, linear regression, and neural networks.

Yilmaz [89] work on **identification of program faults (bugs)** in programs used a machine learning technique, Local outlier factor (LOF). To identify faults he employed dynamic architecture-aware signatures (performance counter patterns).

Grewe et al. [41] in their work on **tuning affinity toward task-to-device mapping** developed a static task partitioning approach for Heterogeneous Systems based on predictive models and program features. The scheduler partitions predict the optimal partitioning factor of OpenCL application to get executed on CPU and GPU. Predictions are done by a hierarchical classifier based on SVM (Support Vector Machine) from static program-inherent features. The reduction of features and normalization of the data is done with PCA. The predictor is split into two levels, the first which decides about mapping only to CPU or GPU, and the second which partitions to both in some ratio.

Moore et al. [65] in their AutoFinity used a signature composed of multiple samples of performance counters to predict affinity policy. The whole approach is driven by machine learning approach called RIPPER which ignore counters and/or counter values that are statistically insignificant. They limit collected data with CfsSubsetEval feature selection algorithm and identify limited (4) counters which form signature.

Zhang et al. [92] in their work on the selection of loop schedulers on HT SMPs (HyperThreaded Symmetric Multi-Processors) have developed a Hardware-Counter Directed Scheduler (HCS) that decides on the scheduler algorithm based on loop signature (cache miss rate, the number of floating point operations, load imbalance, etc.). The optimal scheduler predictor is based on decision trees. The rules for the tree are generated from the offline profile data by clustering software.

Van Craeynest et al. [84] used performance property metrics; based on hardware counters, architecture specific information (reorder buffer size, dispatch instructions per cycle, etc.) and dynamic program information; to predict to which cores to map application tasks on heterogeneous multi-cores.

Ukidave et al [83] in their Mystic framework developed a scheduler which targets servers and cloud for GPU workloads that dispatch applications based on limited profile signature. The signature is created from hardware counters and used by analytic techniques and machine learning, i.e., SVD-based collaborative filtering, to predict compatibility for co-scheduling with another application. The selection of SVD-based collaborative filtering is motivated by the need to fill the gaps of missing values.

Dubach [30] in his thesis employed benchmark characterization technique for **benchmark**

suite reduction. In the thesis, an architecture-centric predictor is proposed, where similarly to [16], it classifies good and bad architecture configurations and tries to identify the optimal one. The training of the linear regressor predictor is done with input data transformed with PCA, collected from SPEC CPU 2000. The evaluation was done with leave-one-out cross-validation.

Dubach et al. [31] in his work on **design space exploration** proposed an approach which combines program-specific predictors with architecture-centric regressor to characterize each new program on a new architecture. The program-specific predictor is built as a neural network and trained offline on a number of simulations from the training programs. Program-specific predictions are combined based on weights estimated by architecture-centric regressor from a small signature. The signature is extracted from the design space of the new program. The model predicts cycles, energy and energy-delay product for a program.

Dubach et al. [32] proposed a dynamic micro-architecture runtime resource management scheme that predicts the best configuration for any phase of a program which maximizes energy efficiency and execution time. The micro-architecture would configure itself by the soft-max algorithm which predicts the best level of resource adaptation based on hardware counters (signatures).

Turakhia et al. [82] in their work on **adaptive performance control** predict micro-architecture configuration, which satisfies defined power-constraints, from micro-architecture aware features. The predictor is split into two levels, the first predictor predicts the CPI from micro-architecture configuration, and the second one predicts power from CPI. Based on predicted power new configuration which satisfies power-constraint is selected.

2.3 Tuning Tools

Reaching good program performance is a very difficult task which often requires extraordinary intuition and knowledge due to the immense complexity of interrelated aspects. Therefore, having tools that employ a well-defined methodology is a must. Most of these tools follow the cyclic performance tuning methodology depicted in figure 2.1.The tuning cycle consists of several steps that answer one or more questions:

Objective Step Before starting a new tuning iteration the user asks himself: "What do I want to achieve with tuning?"

To answer this question the user comes with a set of objectives that he wants to improve, like energy consumption, execution time or increase utilization. The next question he should ask himself is: "How do I know that I reached my objective?" The user now defines a set of tests that can prove or disprove correctness of the

decision to modify the system or program parameters. Once the tuning objectives are met the user decides whether to continue with the tuning process.

Preparation Step What has to be done before I can do experiments?

Before starting with the experiments the program has to be prepared. Preparation includes code instrumentation, compilation, identification of parameters believed to influence performance, e.g., CPU frequency, and configuring measurements believed to provide insight in application behavior, e.g., cache misses. Automation of the program preparation for analysis is often asisted by various tools like instrumenters and various tools that prune the search space by eliminating parameters and reducing their range.

Experiment Step How to collect knowledge necessary to improve my program?

The knowledge is available from various micro-architecture properties, program-inherent properties, and their runtime interactions. To gather this values various sources were developed or exposed in the past with tools like compilers, performance counters, library wrappers, source code instrumenters, etc. Automation of experiment execution and collection of performance events is commonly done with performance analysis tools.

Analysis Step What are problems of my program and how to detect them?

To answer this question, the user searches for potential performance problems or bottlenecks. Detection of performance problems is considered to be the hardest as it requires a good knowledge of the program being analyzed, the micro-architecture it runs on, the network topology and their interactions. During this step, the user considers what causes a bottleneck in his program. He comes with hypotheses about the performance problem, e.g., load imbalances. The user or the tool specifies when he considers that such a problem exists, e.g., load balance exists when the relative difference between the minimum and the maximum execution time is more than 30%. After identifying that the problem exists he wants to attribute which resource, e.g., transfer operation, is responsible for performance problems. The usual code consists of distinct phases of execution, e.g., initialization, computation, and finalization, for which the behavior of the program can change radically. Therefore the user wants to identify which phase of program execution is responsible. Performance analysis tools provide wide-range of automation ranging from tools that provide just the plain list of metrics with their values, tools that visualize them to tools that provide high-level performance properties.

Optimization Step How to improve performance?

Based on knowledge collected from previous steps about performance problems the user can decide which system or application parameter's value to alter in order to improve the performance of the objective. The selection of parameters is not

trivial and requires knowledge of many interrelated interactions between different parameters. Depending on the tuning aspect the user might change the combination of compiler optimizations, change the CPU frequency, select a different algorithm, etc.

Verification Step Did optimization improve performance?

Once the runtime parameter value was altered the user wants to percieve the effect it had on his program's performance. The change of a runtime parameter value may besides the effects on performance have negative effects on correctness. An example of such behavior are incorrectly generated binary due to incompatible compiler optimizations. When the verification step is done the user commonly wants to have information about the objective, detected performance problems and which optimizations he tried. From this information, he can later collect experience, see what went wrong in the tuning process, etc.

Production Phase After the tuning process has finished the user implements the best performing optimizations into his code or the runtime environment.

This section presents tools proposed or developed by researchers that automate the tuning process with little or no user intervention.

Depending on when the optimization takes place, tools can be split into offline, i.e., optimizations take place after execution or online, i.e., optimizations take place during the execution. Online optimization tools often exploit repetitive behavior of programs, phases, to reduce time necessary for the tuning process. The repetitive behavior of the program would be the tendency for a section of code to be executed multiple times, i.e., a timestep of a simulation. Depending on the performance stability between phase iterations, tools can be split into ones that assume changes in performance dynamics or tools without such assumption. Tools that assume stable performance dynamics are assumed to be static in contrast to dynamic tools. The reason for performance dynamics comes from the fact that programs take conditional branches depending on the iteration or the workload changes between iterations, e.g. the simulation employs adaptive mesh refinement.

The explored tuning tools can be grouped based on techniques they employ into the following categories:

- Automatic tuning of HPC applications started in context of self-tuning linear algebra and signal processing libraries. Representatives are: ATLAS [88], FLAME [42], OSKI [85], FFTW [37], SPIRAL [73], Brainy [52].
- Tools that automatically analyze alternative compiler optimization and search for their optimal combination; [81, 45, 70, 59, 38]. To cope with the large search space these tools use either iterative search techniques or machine learning techniques.

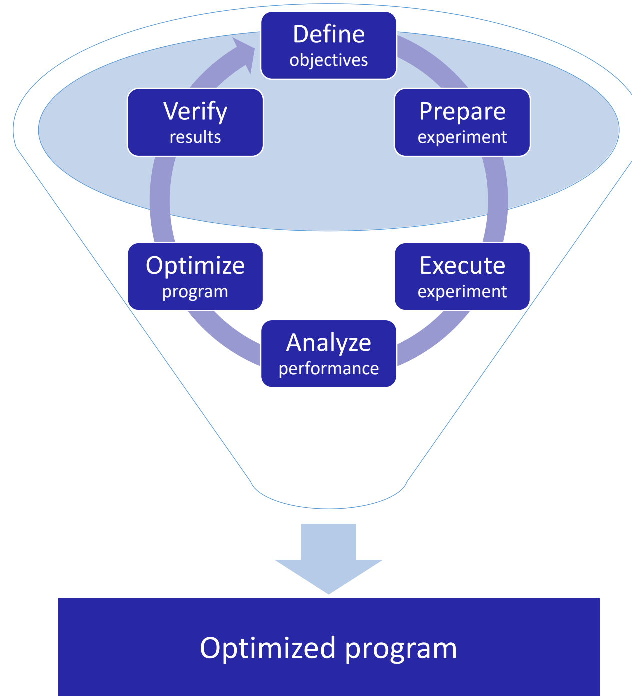


Figure 2.1: Cycle of the Tuning Process.

- Auto-tuners that search a space of application-level parameters that are believed to impact the performance of an application [20, 69, 27, 21, 53]; and
- Frameworks that try to combine ideas from all the other groups. [80, 74, 51, 35]

2.3.1 Domain-Specific Automatic Tuning Tools

The initial research of automatic tuning of HPC applications started with self-tuning linear algebra and signal processing libraries. One of such libraries, ATLAS [88], an offline self-tuning library, tunes the standard basic dense linear algebra kernels from BLAS library. During install time, it runs micro benchmarks to stress different aspects of the architecture such as the memory hierarchy or floating-point units. During micro-benchmark runs it empirically searches over the parameter space of a function, multiple implementation and code generation. The optimized library is then generated with the best configurations for different problem sizes on the target machine discovered during the benchmark phase.

Similar in the context is FLAME [42], which also tunes the dense linear algebra kernels. The optimized library is generated from a high-level specification by searching over implementations of different algorithms to select the best performing. OSKI [85] tunes sparse linear algebra methods like sparse matrix-vector multiply (SpMV), sparse triangular solve. The library combines offline benchmarking, performance modeling and run-time analysis

for tuning available in SPARSITY framework.

The Fastest Fourier Transform in the West (FFTW) [37] tunes signal processing method, discrete Fourier transforms (DFT). The optimal algorithm used for decomposition of transform and calculation is selected from the estimated or measured execution of available algorithms. It provides a wide range of decomposition algorithms like Cooley-Tukey variants and prime-factor FFT algorithms (Rader's, Bluestein's, etc.). When decomposition is done the DFT is computed using compile time generated hard-coded unrolled kernels. The measurements are collected during subsequent executions and stored into knowledge files.

SPIRAL [73] generates digital signal processing (DSP) algorithms and other numerical kernels from high-level specifications (domain-specific knowledge). The high-level specification contains implementation details specific to the system. To further improve its performance the library does feedback driven iterative compilation. The generated library can contain platform-tuned software or hardware implementations of transforms such as DFT, discrete cosine transform (DCT), etc.

Jung et al. developed Brainy [52], a performance analysis tool, that selects the best performing STL datastructure. The estimation of best performing datastructure for is done with Artificial Neural Networks (ANNs) through similarities of white-box signatures, i.e., a combination of architecture-aware features with program-inherent features.

2.3.2 Iterative Compilation

A very popular optimization strategy for automatic tuning are tools that automatically analyze compiler optimizations and search for their optimal combination. The strategy repeatedly evaluates a large set of optimizations and based on information collected during runtime selects other optimizations until their optimal combination is found. Approaches employed by these research could be split into iterative search tools that search through the search space and machine learning approaches that focus on characterizing optimizations.

The iterative search tools were pioneered by Bodin et al. [13] who searches for optimal loop tiling and loop unrolling factor for the matrix multiplication problem. Later, Triantafyllis et al. [81] have developed Optimization-Space Exploration (OSE) algorithm which uses iterative compiler technique to evaluate many combinations of optimization options and to select the best performing combination. The approach employs heuristics to prune the search space and static performance estimations to reduce compilation overhead. The search is guided by the feedbacks from previous estimations. Haneda et al. [45] propose non-parametric inferential statistic method, the Mann-Whitney test, to identify the state of individual compiler optimization options and finally their optimal combination. Pan et al. [70] have developed Combined Elimination (CE) algorithm which identifies options with the negative effect on performance and eliminates them one-by-one, if optimization interacts, or in all-at-one, if there is no interaction. The optimization is

guided by relative improvement in performance of disabled option compared to disabled one. The CE approach requires fewer evaluations compared to OSE.

The next sub-class of iterative compilation tools employ platform specific models to predict optimal combination of compiler optimizations from the white-box signature. Leather et al. [59] employ machine learning techniques to assist the process of searching for the optimal combination of compiler flags. The search is guided by the signature based on which the prediction is made. The feedback is used to evaluate the quality of the signature and update the signature. The update of the signature is used by the algorithm which is a hybrid between Grammatical Evolution and Genetic Programming. The Milepost project investigated machine learning based compilation for GCC for Intel, AMD, and ARC CPUs [38]. Tuning results are forwarded to a public tuning database to improve machine learning results across multiple codes. The developed GCC machine learning plugins are based on probabilistic and transductive approaches to predict good combinations of over 100 optimization flags. To summarize, these tools either employ iterative search techniques or machine learning techniques to effectively deal with the large search space.

2.3.3 Application-level Parameter Optimizations

Another group of auto-tuners are tools that search a space of application-level parameters. Chung et al. [20] in their Active Harmony infrastructure discover the relative importance of the parameters in advance from historical performance data and focus on performance critical parameters first. The infrastructure is capable runtime tuning of parameters like which algorithm is being used to different buffer sizes. The search algorithm they employ is the Nedler-Mead modified simplex method which finds a minimum value of the tuning objective. The critical parameter's sensitivity is evaluated with the sensitivity test. The sensitivity test tries different values for one of parameters while the rest of the parameters are fixed with the default value. The process is repeated until sensitivity of all parameters is tested.

Nelson et. al. [69] propose the model-guided empirical optimization, a hybrid approach which combines the models and empirical techniques. The tool provides three high-level models capacity parameters, balance parameters, and functional specification parameters. Capacity parameters affect storage structures like cache, registers, and local memory. Balance parameters which trade-off the use of multiple resources, e.g., the effect of loop unroll on registers, and floating point units. The last model is built with the help of the programmer and expresses the impact of parameter values on programs' performance. Such approach helps to prune the search space. The intention of the tool is to do the design-time tuning.

Costa [27] developed Monitoring, Automatic and Tuning Environment (MATE), an automatic dynamic tuning framework (architecture) for parallel applications. MATE implemented high-level models built-in tunelets which employ actuators, to modify runtime

parameter values, and sensors, to request and gather metrics, to drive the tuning process. Sensors also perform basic operations to reduce metric communication, like summarize, average and threshold based metric generation. The heuristic performance model applied is applied to Master-Worker pattern to find the optimal number of workers and the best grain size without affecting execution time. The tunelets decide what should be dynamically instrumented and measurements produced from instrumentation are transferred back to the tunelets.

Collings et al. [21] developed MaSiF, an autotuner, which employs program characterization from the statically extracted signature to prune the search space. In the offline phase, MaSiF collects features which comprise the signature and find per-program a set of near optimal parameter values. In the online phase, it extracts the signature and selects the set of near optimal parameter values from the most similar program. Now, the pruned search space is transformed with PCA into a set of values of linearly uncorrelated variables, i.e., principal components and only first two are independently searched. Such search strategy reduces complexity from $\mathcal{O}(mn)$ to $\mathcal{O}(m+n)$. Katagiri et al. [53] developed ppOpen-AT framework, based on ABCLibScript and the FIBER framework, which supports install-time, before execute-time and runtime autotuning. ppOpen-AT provides directives which allow the user to define code selection that has to be transformed, define variable and function parameters with their ranges. Supported transformations are loop unrolling, loop split, loop collapse, loop reordering and combinations. The search is done with exhausting search.

2.3.4 Combined Optimizations Approach

The last group of autotuners are frameworks that combine ideas from all other groups. Tiwari et al. [80] developed Parallel Active Harmony which combines Active Harmony's powerful parallel search backend with CHiLL compiler transformation framework's automated transformation and code generation engine. The Parallel Active Harmony is a search based tool which employs simple heuristics based on architectural parameters to reduce tuning parameter ranges, i.e., to prune the search space. During the tuning process Active Harmony's search backend requests CHiLL's code-generator to generate variants with given sets of parameters for loop transformations. Based on this requests, CHiLL compiler framework does the necessary loop transformations and generates necessary code which is then evaluated. The tool supports two modes of work, as automated compiler optimizations and as user-directed tuning. The best candidate is integrated into the application.

Ribler et al. [74] developed Autopilot, an integrated toolkit for instrumentation, performance monitoring and online application tuning for the heterogeneous computational grid. The toolkit consists of configurable distributed sensors distributed actuators. The sensors capture quantitative application and system performance measurements and generate a qualitative description of resource demand and qualitative performance measure-

ments. The actuators can modify runtime tuning parameter values. Local, i.e., processes, and global decisions, i.e., application-wide, are done with the help of fuzzy logic and neural network classifications from performance measurements and application resource demands.

Jordan et al. [51] developed Insieme framework a multi-objective dynamic tuning framework. The framework consists of the analyzer, the optimizer, the backend and the runtime system. The analyzer identifies code region transformation skeletons, their parameters and based on the target platform their parameter constraints. The optimizer determines variants that are evaluated for each of the regions. Now, the backend generates and compiles multi-versioned binaries which are evaluated on the runtime system. Measurements for all regions are obtained in a single run of the program. From the measurement results, the optimizer derives a Pareto set for each of the selected regions. The backend now generates a set of specialized code versions, one code version for each solution in the Pareto set. The runtime system now can dynamically select among the code versions. To further reduce the search time, Insieme employs powerful parallel evaluation and search engine similar to Parallel Active Harmony.

Fialho et al. [35] extended PerfExpert a performance measurement and optimization framework, developed by the Texas Advanced Computing Center, with automatic tuning features. The framework combines knowledge of the micro-architecture, the system software, and the compiler technology. The analysis tool derives performance bottlenecks from raw performance measurements gathered by HPCToolkit. The focus is on performance bottlenecks that come from data memory access, instruction memory access, floating-point operations, branch instructions, data Translation Lookaside Buffer (TLB) access, and instruction TLB access. Based on detected performance bottlenecks the AutoSCOPE tool [77] generates a list of possible optimizations. The tuning process in guided by the recommendation system which based on historical performance data recommends possible optimizations. The optimization process is closed loop, first, the performance data is collected and performance bottlenecks are identified. Then, the recommendation system recommends possible optimizations and they are evaluated with given constraints through static analysis of the input code. The recommendations are then applied to the code and it is recompiled and run. The process is repeated until no more valid recommendations exist.

2.4 Summary

From research done by Park et al. [71], contrary to popular belief, it was proven that black-box signatures can perform equally well, if not better, compared to white-box signatures. Therefore, as one of the contributions, we propose a performance model for program optimization which employs black-box characterization of the search space. We employ plugin signature, plugin performance across programs. The purpose of the model

is to predict tuning potential of a plugin based on similarity with other programs tuned with the plugin. The model is agnostic to the aspect that the plugin tune and the architecture where it executes. More details about the implementation of the performance model are given in Chapter 5. Such a characterization technique is shown to be effective on a large number of serial and parallel benchmark suites and suitable plugins from PTF. Our work differs from previous research due to fundamental aspects:

1. We present a tuning potential prediction model which requires small number of runs.
2. Tuning aspect independent.
3. Signature that we use is black-box.

The evaluation of the signature quality can only be found by making predictions with the machine learning and see how good the predictions are.

Due to the special constraint of our approach, we employed methods like information gain and cluster analysis.

Chapter 3

The Periscope Tuning Framework

3.1 Introduction

This chapter explains the context of this work, the Periscope Tuning Framework (PTF), is a tuning framework developed in the Autotune project [4] at the Technische Universität München. Understanding how the framework works and its concepts is necessary to understand the challenges and contributions addressed in this thesis.

The main idea behind the framework is to close the gap between performance analysis and tuning. Therefore, Periscope [40], an online performance analysis tool, was extended with fully automatic tuning support. The framework brings several tuning plugins which tune individual aspects of HPC applications, e.g., MPI. With such an approach, the tuning plugins have insight in the application's performance information. The gathered information is presented to the plugin in the form of performance properties, e.g., load imbalance. Performance properties are used by the plugin to perceive the effect of the setting on application performance, to shrink the search space and to increase the efficiency of the tuning plugins.

Modern HPC systems are highly parallel machines which consist of a high number of NUMA nodes hierarchically connected with tree-like fast interconnect. To satisfy needs of modern HPC systems and modern highly parallel applications PTF is built around three concepts. The first concept is the distribution of its components across the system. Such an approach distributes many important tool activities and effectively reduces data analysis time and communication between the back-end components and the frontend. The second concept is the automation of analysis and tuning which requires little or no intervention from the user. The third concept is to do analysis and tuning during the program run which reduces the necessity for the complete run to produce valuable results or can collect more information within one run.

The Figure 3.1 depicts the high-level overview of the PTF architecture. The architecture of Periscope consists of distributed components, namely, the user interface, the frontend,

analysis agent hierarchy and the monitor linked to the application processes. The user interface is developed as a plugin for Eclipse and allows the user to inspect found performance properties. The frontend triggers performance analysis strategies and executes the predefined tuning model. The analysis strategies are executed by the analysis agents. The monitor gathers performance data from the running application necessary to identify its performance properties.

The novelty of PTF is support for the implementation of tuning plugins. For that purpose, PTF provides a rich toolbox. Tuning plugins in PTF follow the predefined tuning model. The tuning model consists of operations which every plugin is required to implement. The operations are defined by the Tuning Plugin Interface (TPI). More details about TPI are given in Section 3.5. The frontend calls individual operations of the TPI and interacts with plugins and other components of Periscope but the tuning process is controlled by the plugin. The tuning plugins can be distributed in source or binary (shared libraries) form. Before and during the runtime, Periscope is able to extract static information and dynamic information from the code.

During the tuning process, plugins investigate a number of variants to identify optimizations. For that purpose PTF allows tuning plugins to request a performance analysis strategy whose results can help tuning plugins to reduce the search space. When requested, the analysis strategy gets executed and its results in the form of performance properties are reported to the plugin. The reported properties provide the plugin with dynamic information. Additionally, tuning plugins have access to static information, e.g., the available program regions which can be used for the tuning purpose.

The evaluation of an experiment requires its optimizations to be applied to the tuning region and to measure its effect on the tuning objective. The tuning region can be either the entire application or a program region. This region, called the phase region, typically corresponds to one iteration of the main progress loop, e.g. the time step of a simulation. Repetitive phase regions allow the execution of consecutive experiments without restarting the application, with the assumption that different iterations of the progress loop have the same behavior.

An optimization is executed by tuning actions, while the effect on the tuning objective is represented as Periscope properties. The tuning actions can be executed during runtime or before it, e.g., change the CPU frequency. The plugin requests them and they are propagated through the hierarchy to the monitor. The monitor receives them and executes them at the requested state of program execution.

To support tuning plugins search algorithms were developed. The search algorithm generates variants, i.e., points in the search space. Similar to tuning plugins these plugins are also dynamically loaded when requested and can be distributed in the source or binary form.

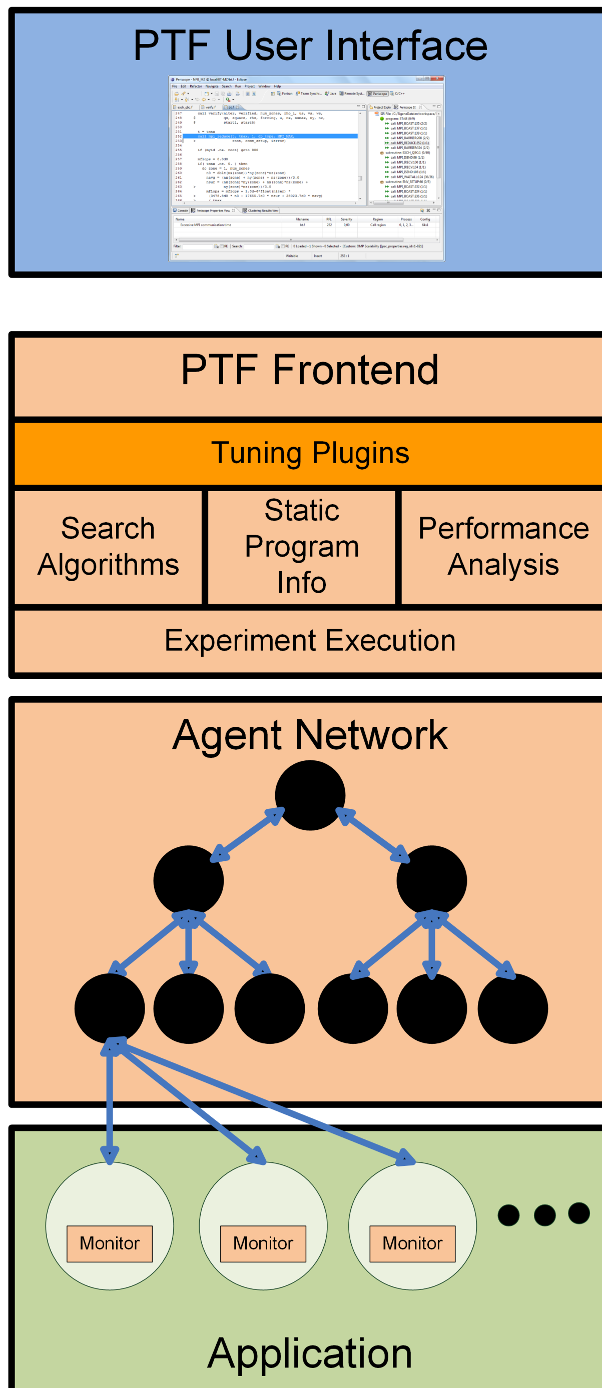


Figure 3.1: Architecture of the Periscope Tuning Framework.

3.2 Performance Analysis

The search for performance inefficiencies is triggered by the frontend but the real analysis is done by the analysis agent hierarchy. The analysis agents automate performance analysis by following the expert knowledge codified as search strategies. The performance inefficiencies in Periscope are formalized as performance properties, e.g., cache misses, instruction starvation, branch mispredictions, load imbalances etc.

Application analysis in Periscope is done as an iterative online process. The search strategies at first create a set of performance hypotheses for a specific context in the form of performance properties. The presence of performance properties can be proven by evaluating its condition from collected performance metrics. The monitor gets configured by the analysis agent and during the runtime, it collects necessary metrics and reports them to the search strategy. The performance properties are proven with the confidence value between 0 and 1. The confidence value of 1 proves the hypotheses and 0 disprove it. Additionally, a performance property becomes a performance problem if and only if its severity is above the predefined threshold. The performance problem with the highest severity is called the performance bottleneck. The analsysis strategies can be configured to report all properties through the `pedantic` flag. The context contains information about static location, e.g., file name and line number as well as dynamic location, e.g., MPI rank, OMP threads, etc. The search strategy might refine the search process based on proven hypothesis. The refinement is repeated until the search process is done. Individual leaf analysis agents perform analysis on their subset of MPI processes. The proven properties are propagated through the analysis agent hierarchy, aggregated by higher level agents and finally reported to the frontend. PTF supports several analysis strategies, such as single core analysis, OpenMP analysis, MPI analysis, and configurable analysis.

Depending on how many experiments the search strategy requires to gather all performance data necessary to identify all performance problems, strategies can be categorized into single- or multi-step search strategies.

3.2.1 Single Core Analysis

The *Single Core Analysis* bundles several analysis strategies as it is system specific. The focus of these strategies is to find performance problems that affect utilization of a single core. The monitoring relies on hardware performance counters built-in the CPU accessed through the Performance Application Programming Interface (PAPI). Typical performance problems are various stalls in the processor pipeline related to memory access stalls or branch mispredictions.

3.2.2 OpenMP Analysis

The *OpenMP analysis* strategy focuses on finding performance problems related to the OpenMP execution. It focuses on performance problems that affect resource utilization like load balancing, task distribution, overheads of parallel regions, unnecessary synchronizations. The monitoring relies on source level instrumentation of OpenMP constructs. Typical performance problems are `LoadImbalanceOMPRegion`, `OverheadDueToTaskCreation` or `FrequentAtomic`.

3.2.3 MPI Analysis

The *MPI Analysis* strategy focuses on finding performance problems related to the MPI execution. It focuses on performance problems that affect resource utilization like idle resources and excessive communication. The monitoring relies on library wrapping instrumentation of MPI.The profiling information of MPI operations is available via the MPI profiling interface and contains the execution time of MPI routines, waiting times, the frequency of call sites and the data transfer volume. The strategy is considered to be single-step as it requires only one experiment to gather all performance data and identify performance problems. Typical performance problems are `ManySmallMessages` or `LateReciever`.

3.2.4 Configurable Analysis

The *Configurable Analysis* strategy was developed to meet the requirements for additional information necessary during the tuning process. This acquired knowledge can be used to shrink the search space or to measure the effect of tuning actions on different regions. The analysis is configured via property requests which specify the:

- list of properties to be evaluated, and
- a list of contexts for which the properties will be evaluated

This allows the plugin to request arbitrary properties for arbitrary contexts. Let's consider how the Master-Worker plugin reduces the search space. The plugin implements a model which calculates the optimal partition factor but requires additional information retrieved by the configurable strategy. The optimal partition factor leads to the most balanced execution with the smallest communication overhead. Without this additional information, the plugin would have no means to calculate the model and reduce the search space.

3.3 Tuning Concepts

3.3.1 Terminology

[1] This section introduces the PTF terminology necessary to understand the tuning concepts and the work explained in this thesis.

The goal of tuning plugins is to improve the performance of an application by modifying parameters, which are believed to considerably influence the performance of an application, called *tuning parameters*. The tuning parameter is defined as $TP = \{v_1, v_2, ..., v_n\}$ where, TP denotes the tuning parameter, and v_i denotes one of the values it can take. One of such tuning parameters is the CPU frequency which can take different clock frequencies. This parameter influences the execution time and the energy consumption.

A list of tuning parameters assigned to one of its values is called a *tuning variant*. The cross-product of all tuning parameters with its values compose the *tuning space*. The tuning space is defined as $TS = TP_1 \times TP_2 \times ... \times TP_k$, where TS denotes the tuning space and TP_i denotes one of the tuning parameters. A set of such variants is called a *variant space* and is a subspace of the tuning space.

Each tuning plugin selects a variant space, which will be empirically evaluated. It defines a *search space* that is the tuple between the variant space (VS) and the *variant context* (VC). The variant context is the location to which the variant is applied. $SS = VS \times VC$. The location to which the variant is applied can be either a program, a list of regions of the program or a file list, but not necessary the tuned region. For example, such flexibility allows the Master-Worker plugin to modify the chunk in the master while tuning the performance of workers.

The search space exploration is guided by a *search algorithm* to optimize for certain objectives. A *tuning objective* is a function $obj : REG \times \ SS \rightarrow \Re$ which maps an element of the search space and a tuned region to an objective value.

The tuning plugin can optimize for a single or multiple objectives. It creates a sequence of *tuning scenarios* executed by the Periscope which evaluate one or more objectives. Each scenario defines the variant to be executed, the context for which objectives are to be measured and the objectives. More information on tuning scenarios can be found in Section 3.3.4.

3.3.2 Tuning Model

The *tuning model* of PTF, depicted on figure 3.2, defines the sequence of operations that every tuning plugin must implement. The tuning process starts with the plugin initializa-

[1]This section contains definitions from the 2nd Chapter of Automatic Tuning of HPC Applications: The Periscope Tuning Framework, Shaker Verlag 2015, ISBN, 978-3-84403-517-9

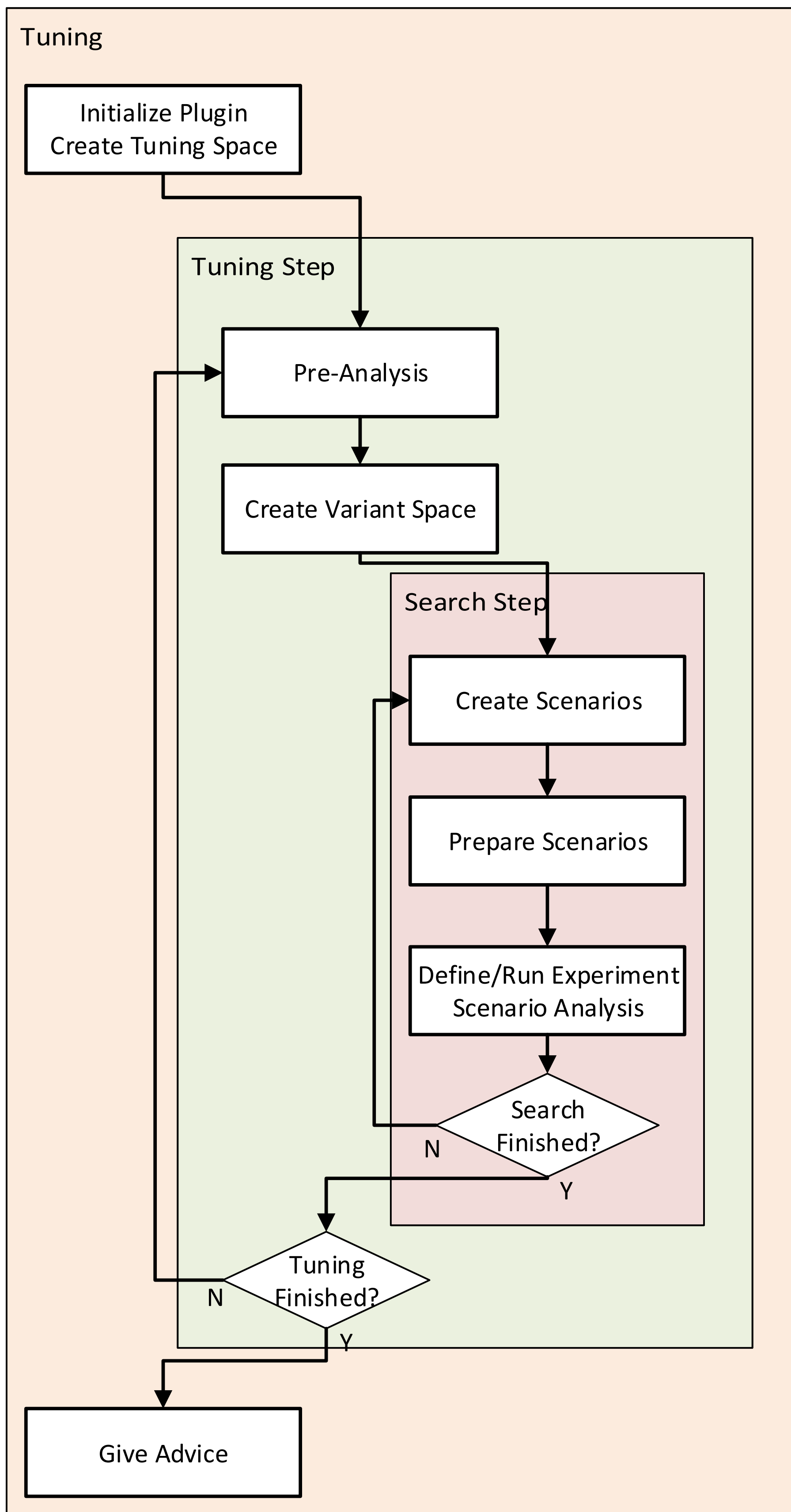


Figure 3.2: The tuning model of PTF.

tion during which the plugin usually prepares the tuning parameters and initializes one of PTF's search algorithms. The tuning parameters are either predefined by the tuning plugin or loaded from the configuration file. The tuning process consists of either one or more tuning steps. Each tuning step begins with the pre-analysis where the plugin selects and configures one of the available PTF analysis strategies. From performance properties proven during the analysis, the plugin populates the variant space. The variant space is combined with the variant context to determine the search space. The properties gathered during the performance analysis can be used to determine the precomputed model.

The search algorithm searches for the variant which optimizes the objective function. PTF provide single- or multi-step search algorithms. The plugin can select PTF's search algorithms or plugin-specific search algorithms. In a search step, the search algorithm generates a set of variants, represented as scenarios (Section 3.3.4). Before scenarios are evaluated they might require additional preparation, like application recompilation or prepare the execution environment (MPI). Scenarios are then combined into experiments and evaluated. During the experiment evaluation, the tuning plugin can request any PTF analysis strategies to get detailed information about the effect of the variant on the application performance.

When all the search steps are executed the plugin can decide about additional tuning steps, select a new variant space based on results from previous tuning steps and, possibly, the results from the pre-analysis. Once all the tuning steps are executed the tuning plugin generates the tuning advice for the user.

3.3.3 Tuning Objectives

The goal of performance tuning is to find the variant that optimizes the *tuning objective*. Tuning objectives values in PTF are represented as severities of Periscope properties. The Periscope analysis capability is employed to obtain the objective value. The plugin defines the property that represents the objective and the search algorithm generates the variant represented as scenarios. Each scenario holds the objective property which is propagated to the analysis agent, the analysis agent requests raw measurements defined in property specification from the monitor necessary to evaluate it. Evaluated properties for the scenario are propagated from the analysis agent to the frontend and stored in so-called scenario pools (Section 3.5).

When all scenarios are evaluated, the search algorithm identifies the best scenario by applying the objective function on a pool of results. For example, the user might want to minimize execution time or maximize the software pipeline throughput. PTF provides common objective functions which minimize or maximize these values but the user also can define its own objective functions. The user might implement more complex objective functions which could combine multiple properties. One such use case is the optimization of the energy-delay product where execution time is combined with energy consumption. To optimize the execution time of the tuned region Periscope provides `ExecTime` property,

while to optimize the energy consumption provides `EnergyConsumption` property. More advanced users can implement their own performance properties and use them to optimize applications for desired objectives.

3.3.4 Tuning Scenario

The central concept of the tuning in PTF are scenarios. They provide a detailed specification necessary to evaluate the tuning objective in the experiment. Each scenario in its specification defines the variant to be executed, the context for which objectives are to be measured and the objective.

Due to generic nature of the framework the scenario specification is very flexible and covers:

- Tuned region
- Tuning specification
- Objective property request
- Objective values
- Scenario Id

Lets consider a complex scenario for the pipeline programming model with the following specification:

- The tuned region is the whole pipeline region executed by the rank 0.
- The tuning objectives are execution time and energy consumption measured for the whole pipeline region executed by the rank 0.
- In the first pipeline stage, the replication factor is set to 4.
- In all other pipeline stages, the replication factor is set to 1.
- In the first pipeline stage, the buffer size is set 32.
- In other pipeline stages, the buffer size is set to 16.
- In the whole pipeline region, the number of CPUs is set to 12.
- In the whole pipeline region, the number of GPUs is set to 4.
- In the whole pipeline region, the scheduling policy was set to HEFT (Heterogeneous Earliest Finish Time).

Its scenario specification could be:

```
tuned region:  PIPELINE
tuning spec 1: ((REPL_FACTOR, 4), PIPESTAGE1, ALL)
tuning spec 2: ((REPL_FACTOR, 1), (PIPESTAGE2,...,PIPESTAGEN), ALL)
tuning spec 3: ((BUFFER_SIZE, 32), PIPESTAGE1, ALL)
tuning spec 4: ((BUFFER_SIZE, 16), (PIPESTAGE2, ..., PIPESTAGEN), ALL)
tuning spec 5: ((NCPUS, 12), PIPELINE, ALL)
tuning spec 6: ((NGPUS, 4), PIPELINE, ALL)
tuning spec 7: ((SCHEDULER, HEFT), PIPELINE, ALL)
prop request:  (PIPEEXECTIME, ENERGY\_CONSUMPTION, 0)
scenario id:   1
obj values:
```

Scenarios are created for a search step by the search algorithm from search spaces, as well as the objective properties and the tuned region. Created scenarios are prepared by the plugin for execution, e.g., the program is compiled. The tuning parameters which are configured before the application executes, e.g., the compiler optimization flags, are set to their values according to the tuning specification. The application is now restarted if the plugin requests it and waits for the requests.

The tuning specification represents the search space point in the search space and contains information about the variant configuration like:

- Variant - The variant to be set. A list of tuples (tuning parameters and their values).
- Variant Context - The static context, the region where the variant configuration is applied.
- Ranks - The dynamic context, the ranks for which the variant configuration is applied.

The scenarios created by the plugin go through several steps of preparation and evaluation in different components of the PTF hierarchy depicted on Figure 3.3. After scenarios are prepared they are combined into an experiment and propagated to analysis agents from the frontend. In the analysis agent, the scenarios are split into the variant to be executed and objective property requests. The variant is used to configure the execution runtime, while the objective property requests are propagated to the requested analysis strategy. The tuning parameters with its values specified in the variant are propagated to the monitor which updates tuning parameters values by applying tuning actions.

For our example the following tuning parameters would be set by the monitor:

- PIPELINE, ALL ranks: NCPUS is set to 12

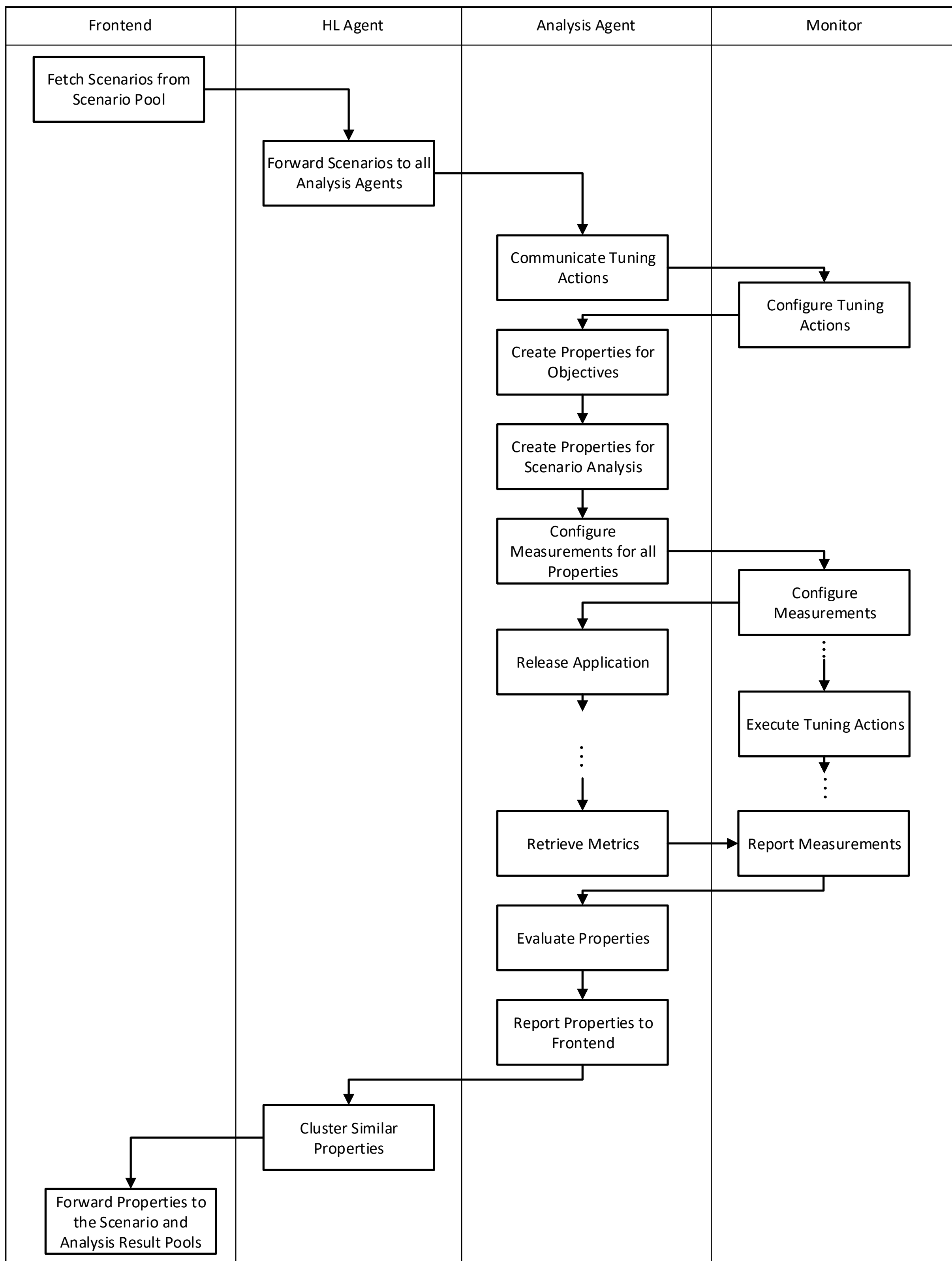


Figure 3.3: Execution of tuning scenarios.

- PIPELINE, ALL ranks: NGPUS is set to 12
- PIPELINE, ALL ranks: SCHEDULER is set to HEFT
- PIPESTAGE1, ALL ranks: REPLICATION_FACTOR is set to 4
- PIPESTAGE1, ALL ranks: BUFFER_SIZE is set to 32
- PIPESTAGE2, ..., PIPESTAGEN, ALL ranks: REPLICATION_FACTOR is set to 1
- PIPESTAGE2, ..., PIPESTAGEN, ALL ranks: BUFFER_SIZE is set to 16

On the other hand, the requested analysis strategy creates a set of objective properties. These properties are combined with properties created from the analysis strategy attached to the experiment. The metric requests are generated from property requests and propagated metric requests. The propagated metric requests are used to configure the monitor.

The objective property contains relevant information to which properties are evaluated and their execution context:

- Property Ids - The properties that should be evaluated.
- Regions - The region for which the property is evaluated.
- Ranks - The ranks for which the property is evaluated.

For our example the following metric requests would be set by the monitor:

- PIPELINE, ALL ranks: measure execution time and energy consumption

As the monitor is now configured the application is released. During the application runtime, the tuning parameters are set to their values and the measurements are collected. After the experiment has finished, the collected measurements are reported to the analysis agent and used for evaluation of properties. Proven properties are reported back to the frontend. The objective values are attached to the scenario. To reduce the communication overhead the agent hierarchy might also cluster similar properties. The collected properties are stored in scenario and analysis result pools.

For our example the collected execution time and energy consumption:

```
objective values: (PIPEEXECTIME, 0, 15), (ENERGY_CONSUMPTION, 0, 750)
```

The experiment can consist of one or more scenarios evaluated in parallel on different processes. The most common distribution is where one scenario is executed in all processes. Besides evaluation of multiple scenarios on different processes, multiple scenarios might be evaluated in a single process if variants are applied to different regions. The specification of regions where experiments can be executed:

- Ranks - ALL, <integer>, or a list of ranges, e.g., 1-4, 6-8

3.3.5 Tuning Actions

The variant contains a list of tuning parameters with values to which they have to be set. Tuning parameters are specified by its name and a list of values or a range of numeric values they can take. The list of values a tuning parameter can hold are numerical or string values. The action in which the tuning parameter is set to the predefined value is called the *tuning action*.

Depending when the tuning action takes place we distinguish *pre-execution* and *runtime tuning actions*. Examples of pre-execution tuning actions are setting the compiler flags for the compilation process or setting the MPI runtime parameters, e.g., Modular Component Architecture (MCA), exposed through the MPI starter. The pre-execution tuning actions take place before the scenario is executed. To support such tuning actions PTF allows the plugin to apply them before the scenario is started in a step called prepare scenarios. During the scenario preparations, the tuning plugin can specify the additional command line arguments or export environment variables. Based on the type of the tuning action, the plugin can decide upon the restart of the application and modify the startup command line.

On the other hand, the runtime tuning actions are, for example, assigning the number of OpenMP threads or selecting the algorithm during runtime. The runtime tuning actions can be divided into two special kinds:

- *Variable tuning actions*
- *Function tuning actions*

These two groups differ how they apply the tuning action. While the variable tuning action applies the parameter value to a variable the function tuning actions call a function with the parameter value. To expose the runtime tuning parameters the user have to map the variable or a function through the mapping functions and assign it to the parameter name. Therefore, PTF introduces the mapping function, PSC_MAP_TP_VAR(...). The analysis agent configures the monitor from the scenario specifications. The requests are stored by the monitor and executed on the region specified in the specification. To define what happens with the tuning parameter value after the region is exited the user enables or disable the restore parameter of the mapping function.

The extension to support the tuning actions in the monitor is done in the context of this thesis. The work done will be elaborated in Section 4.2 of Tuning Plugins Chapter.

3.4 Search Algorithms

The creation of the search space and the decision on which points of the space are going to be evaluated is taken by the search algorithm. Similarly to the tuning plugins, the search algorithms can be written by the user and distributed in the source or binary form. The tuning plugin can decide whether it will implement its own search algorithm or employ one of PTF's predefined search algorithms.

Search algorithms are used to guide the search space exploration by selecting scenarios that the plugin will evaluate. PTF provides a set of predefined search algorithms:

- Exhaustive search
- Probabilistic random search
- Individual search
- GDE3 multi-objective genetic search
- Active Harmony's Nelder-Mead Simplex algorithm

Alongside the predefined search algorithms, PTF allows plugin developers to implement their own search algorithms. A search algorithm can be implemented through the predefined plugin interface. The predefined interface allows the plugin to add the search space, request new scenarios, determine when the search has finished, and to get the search path and the optimum.

PTF supports multiple search steps. In each tuning step, the search algorithm creates a new set of scenarios perhaps based on the results acquired in the previous search step.

The input of the search algorithm corresponds to one or more search spaces, as well as the tuned region, the objective properties and the objective function. A *search space* is a tuple between the variant space and the variant context. The variant context represents the program location to which the variant is applied which can be either a program, a list of regions or a file list.

The product of the search algorithm are scenarios, introduced in Section 3.3.4.

The size of the variant space is determined by the number of parameters and their values. Therefore, its size grows exponentially with every new parameter which makes it impractical for search. To circumvent the problem of huge variant spaces, the algorithms often trade optimality for speed. Based on the trade-off, the search algorithm follows one of two main approaches, *exact* and *heuristic*, to walk through the given search space.

The exact algorithms always find the optimal combination of tuning optimizations but are not always possible. Therefore, the heuristic algorithms are employed which finds a close-to-optimal combination of tuning optimizations.

3.4.1 Exhaustive Search

The simplest form of the search algorithm is the exhaustive search. The algorithm simply creates all possible scenarios in a single search step. Each represented variant contains all tuning parameters with one of their values. This results in the search space that is the cross-product of all tuning parameters and their values.

Due to its exact approach, this algorithm is guaranteed to find the optimal combination of tuning optimizations but it is not possible to use it in most practical cases.

The variant space consists of n tuning parameters each with m_i values and based on that the complexity of the algorithm can be expressed as:

$$\mathcal{O}(m^n) \quad where \quad m = \max_i m_i, \tag{3.1}$$

3.4.2 Individual Search

[2]

The individual search assumes that the list of tuning parameters (TPs) is ordered according to their importance. It iterates through the list of parameters and incrementally adds a new tuning parameter to the list of already explored ones for every tuning step. When the new tuning parameter is added its whole range is evaluated and the tuning parameter is narrowed to only the configurable number of best values. In the successive steps, only the best values for a tuning parameter will be considered.

The individual search algorithm (see Algorithm 1) is implemented on top of the existing exhaustive search. It first creates a new internal list of TPs and then adds the next not yet processed tuning parameter to it. It then calls exhaustive search to create the scenarios defined by all TPs in the list. These scenarios are executed and then sorted by their objective value. The algorithm keeps track of the current best scenario. If the new TP leads to a better scenario, it selects `k` values of this TP leading to the best results in the previous experiments. These `k` best values are then marked down for the current TP. The vector containing the set of all possible values for the TP is now restricted to the chosen `k` values. This operation is called *vector restriction* and results in shrinking the search space. When a new set of scenarios is created with the exhaustive search on the next search steps, only those `k` best values of the TP are going to be used.

[2]This subsection is a minor revision of the work published in Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS) (2016), pp. 5752-5761, Kohol, HI, USA, Jan 2016.

Algorithm 1 Individual search strategy.

```
TP                                          ▷ set of input flags
TP' ← ∅                                     ▷ set of explored flags
while (TP ≠ ∅) do
    select tp ∈ TP
    TP ← TP − tp
    TP' ← TP' + tp
    exhaustive search for TP'
    sort scenarios by fitness
    if (better scenario exists) then
        determine k best values of tp
        restrict the range of tp to selected values
    else
        TP' ← TP' − tp
output best scenario
```

If the new TP only led to scenarios that are not better than the current best scenario, this TP is deleted from the internal list. The algorithm repeats this step until all the given TPs are processed. It then returns the best found configuration.

If and only if the algorithm assumption is true the algorithm finds the optimal variant, otherwise the algorithm finds close-to-optimal variant. The quality of the final result depends on the order of the given sequence of TPs. TPs with higher performance impact should be listed at the beginning of the sequence.

The k coefficient introduces a relaxation of the algorithm, giving the opportunity to cover cases where there are dependencies between the TPs. For example, the value of the current TP associated with the second best scenario might deliver better results when combined with the next TP, than when combining the value of the current TP in the best scenario with the next TP. In this case, setting $k = 2$ also covers the better scenario.

The complexity of the algorithm can be expressed as:

$$\mathcal{O}(k^n \cdot m \cdot n), \tag{3.2}$$

where

- k is the *keep factor* - the maximal number of values for one TP tested on subsequent steps, with $k < m$,
- m is the maximal number of values for any given TP,
- n is the number of TPs.

In the most restrictive version, it only evaluates $\sum_i^n m_i$ scenarios.

Compared to exhaustive the speed is improved as the number of evaluated scenarios was reduced from m^n to k^n. Although the complexity remained exponential $k < m$ and the keep factor is usually very small, in the range of 1 or 2 compared to m which could have any value. Secondly, the individual search heuristics employs the greedy algorithm which is considered to have a good trade-off between the speed and accuracy. If parameters are well ordered by their importance, there is a high probability that scenarios with worse objective values are pruned from the search space. Also, it is worth noticing that if the tuning objective is execution time the removal of worse scenarios has a positive effect on the search time.

3.5 Scenario Execution Infrastructure

3

PTF defines the *Tuning Plugin Interface* (TPI) which every tuning plugin has to implement. Execution of individual functions of TPI and their order is enforced by the state machine in the frontend.

For illustration purposes, a version of PTF's flow is presented in Figure 3.4. The flow illustrates tuning process from scenario creation, processing of the scenarios until their evaluation is finished.

To manage the scenario evaluation flow the set of scenario pools is used. The scenario pools are created by the frontend and passed to the plugin through TPI. The movement of scenarios between scenario pools is illustrated with dashed lines. The scenarios created by the search algorithm, which are created for the tuning step, are stored in the *Created Scenario Pool (CSP).* Scenarios from the Created Scenario Pool are popped from the pool, consumed to execute pre-execution tuning actions, prepared and stored into the *Prepared Scenario Pool (PSP).* From this pool, the plugin decides which scenarios can be executed in a single experiment and moves them into the *Experiment Scenario Pool (ESP).* As soon as, the scenarios are evaluated they are forwarded by the frontend to the *Finished Scenario Pool (FSP).*

3.5.1 Tuning Plugin Interface (TPI) Sequence

TPI functions are triggered by the frontend in the following order:

initialize: initializes the plugin. The frontend provides the plugin with a context object

[3]This section is a minor revision of the work published in Proceedings of the International Conference on Parallel Computing, ParCo 2013, Advances in Parallel Computing 25, IOS Press 2014, ISBN 978-1-61499-380-3, pp. 123-132, Garching (near Munich), Germany, 10-13 September 2013.

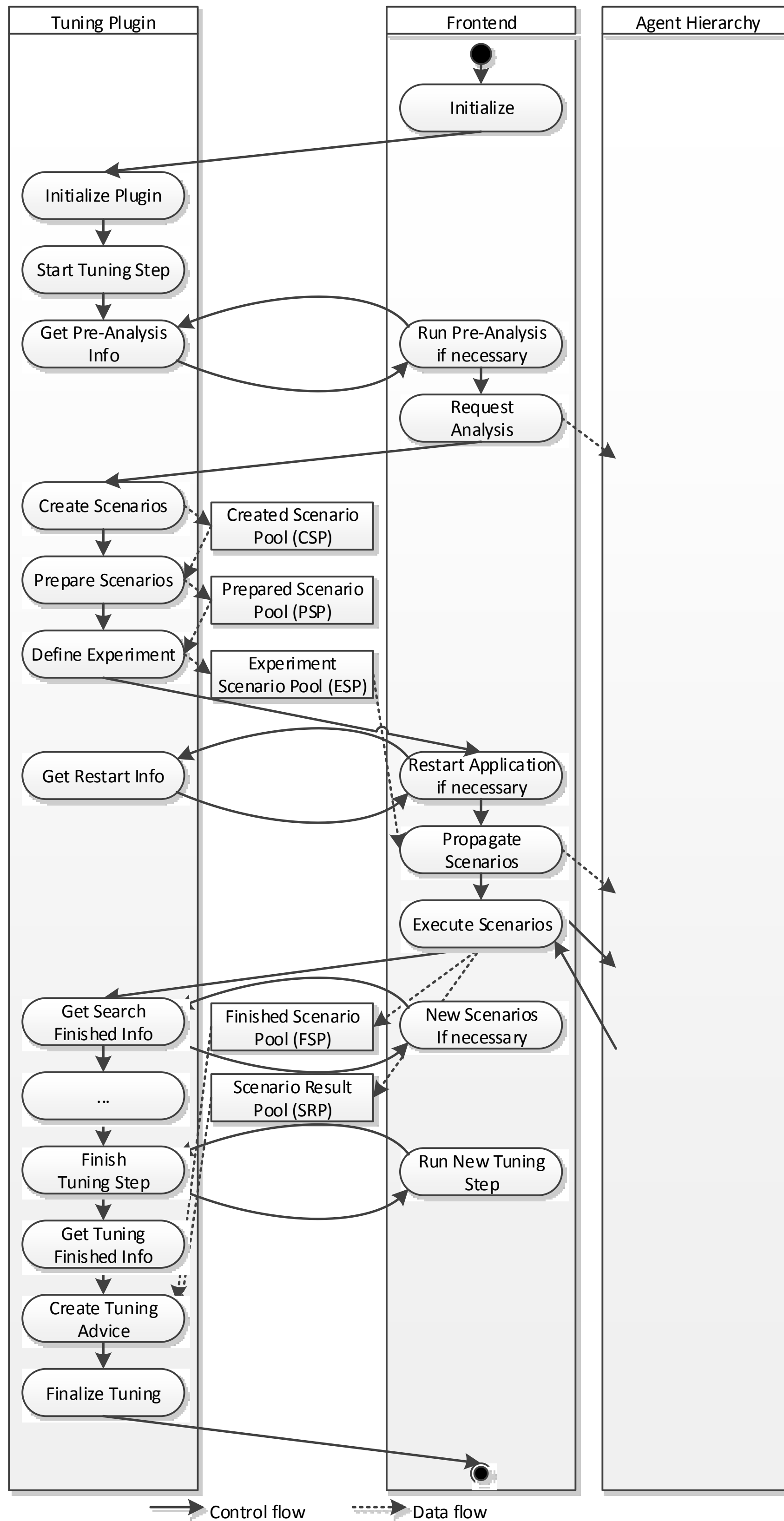


Figure 3.4: The TPI call sequence.

```
virtual void initialize( DriverContext   *context,
                         ScenarioPoolSet *pool_set ) = 0;
virtual void startTuningStep( void ) = 0;
virtual bool analysisRequired( StrategyRequest **strategy) = 0;
virtual void createScenarios( void ) = 0;
virtual void prepareScenarios( void ) = 0;
virtual void defineExperiment( int               numprocs,
                               bool             *analysisRequired,
                               StrategyRequest **strategy ) = 0;
virtual bool restartRequired( string *env,
                              int    *np,
                              string *cmd,
                              bool   *instrumented )  = 0;
virtual bool searchFinished( void ) = 0;
virtual void finishTuningStep( void ) = 0;
virtual bool tuningFinished( void ) = 0;
virtual Advice *getAdvice( void ) = 0;
virtual void finalize( void ) = 0;
virtual void terminate( void ) = 0;
```

Figure 3.5: Specification of the Tuning Plugin Interface functions.

and a set of scenario pools. The context object provides basic information on the tuned program and a timer. This step is often used to parse the plugin specific options, read the tuning parameters from a configuration file or load the search algorithm, e.g., exhaustive search.

startTuningStep: starts a new tuning step. It can be used to create the variant space from the tuning parameters.

analysisRequired: notifies PTF whether a pre-analysis is required. It can be used to determine regions that account for significant amount of time and therefore are worth tuning according to Amdhal's law.

createScenarios: creates a set of scenarios, usually done by the search algorithm from the search space. For example, the tuning parameters and regions, possibly processed based on the results of the analysis, are passed to exhaustive search that creates the possible scenarios for evaluation.

prepareScenarios: prepares or executes pre-execution tuning actions. For example, the additional command line arguments are prepared to test the MPI command line parameter selected in the scenario.

defineExperiment: selects scenarios for the next experiment and possibly requests scenario analysis. The strategy analyzes the effect of tuning actions on different regions. For example, the effect of the experiment on energy consumption of other regions.

restartRequired: notifies PTF whether the application must be restarted for pre-execution tuning actions to be applied. During restart, the plugin can request a different number of MPI processes, change runtime command line and export environment variables.

searchFinished: notifies PTF whether the search step is finished or a new set of scenarios is to be created. For example, machine learning search algorithms like Nelder-Mead Simplex algorithm go through a sequence of generations to find a close-to-optimal tuning optimization.

finishTuningStep: finalizes the tuning step. It can be used to process the results collected in the tuning step before entering a new tuning step. For example, Master-Worker plugin uses this information to fix the partition factor.

tuningFinished: notifies PTF whether it requires another tuning step. For example, the Master-Worker plugin uses two tuning steps, the first to find the optimal partition factor, and the second, to find the number of workers.

getAdvice: reports PTF with the tuning advice based on the findings collected during the tuning process of the plugin. Each plugin generates the optimal variants and PTF stores it into the specified XML file.

finalize: finalizes the plugin, releases the resources used like the search plugin.

terminate: terminates the plugin.

The frontend triggers `prepareScenarios` until all scenarios from CSP are evaluated, while `defineExperiments` is triggered until PSP is empty. Once all the created scenarios are evaluated, the frontend calls `searchFinished` to determine whether the plugin needs to create additional scenarios.

The plugin search is controlled by the search algorithm selected by the plugin. However, the user can decide about the amount of time he is willing to spend to tune the application. The time budget is propagated to the plugin as a timer through the context object. The plugin can assign the function that is called when the timer expires. This functionality can be used to ensure that the plugin terminates after it expired without evaluating all the scenarios generated by the search algorithm.

3.5.2 Tuning Step Pre-Analysis

PTF provides means to execute a performance analysis before each tuning step called pre-analysis, or during the execution of a scenario called scenario analysis. The plugin

can decide to run one of the available analysis strategies or its own analysis strategy. For pre-analysis, based on the resulting properties the plugin collects the signature used by the model to predict optimal parameter values or narrow parameter ranges and thus effectively prune the search space. For example, the DVFS Plugin employed Configurable analysis to collect the white-box features necessary for the model that predicts the optimal frequency.

The plugin can request the pre-analysis strategy during the `restartRequired` call of the TPI. At the end of the call, the plugin returns true if the pre-analysis is requested. The properties proven during the analysis are stored in the dedicated *property pool* described in Section 3.6. From there the plugin can request them for inspection.

PTF allows enforcing a baseline variant before the pre-analysis is executed. This is often required for the machine learning models, which is trained for a certain variant. For example, the DVFS plugin sets the CPU frequency before it collects the model signature. This is possible through so-called `tune` strategy requests. The `tune` strategy request consists of the scenario and the analysis strategy requests. The analysis request for the pre-analysis consists of a request for the tune strategy which is configured with a scenario and a sub-strategy request. The `tune` strategy is implicitly executed by the analysis agent during the tuning process and here explicitly requested for running the pre-analysis.

3.5.3 Scenario Analysis

Due to the complex interaction of hardware and software, the positive effect on the performance of the variant in one region can have negative effects on performance of other regions. Therefore, PTF allows the plugin to request the analysis during the execution of the application. Such an approach allows combining the effect of the execution variant on the tuning objective with an in-depth analysis of the application behavior. For example, the DVFS plugin uses this extension to gather the execution time of significant routines during an experiment with a certain CPU frequency. From this information, region-specific recommendations are returned to the user in the tuning advice. The scenario analysis configures the analysis strategy to report all properties through the pedantic flag. The plugin can request the scenario analysis strategy through the `defineExperiment` function interface of the TPI. The plugin requests it by configuring the `strategy` object and setting the `analysisRequired` flag, which enables it.

3.6 Results Pools

During the tuning process, the tuning plugin generates various requests that result in properties. Depending on the purpose of the evaluated properties, they fall into two groups, *analysis properties* and *objective properties*. Those requests are propagated through the hierarchy to the leaf analysis agents for evaluation. Once they are evaluated, the anal-

ysis agent marks them with a flag indicating their purpose and propagates them to the frontend. Based on the flags, the frontend automatically distributes them into specialized result pools. The analysis properties are stored in the *Analysis Results Pool (ARP)*, while the objective properties are stored in the *Scenario Results Pool (SRP)*.

The SRP maintains two lists of scenarios based on a scenario id and a search step for which the property is evaluated. The scenarios created by the plugin for a search step are stored with the search step information. The plugin can later easily retrieve those properties based on the scenario id or the search step and make new decisions or generate the advice.

The APR also maintains two lists, one based on an experiment id and a tuning step of the plugin. The plugin can later quickly retrieve those properties and make decisions during the pre-analysis or the scenario analysis.

3.7 Advice Format

Once the tuning process has finished, the user often wants to save a full backlog of all previous steps and decisions. This information can be used as a starting point for the future tuning. Therefore, PTF provides support for automatic generation of XML-based tuning advice. The XML format allows easy human inspection but also easily interpretable by other tools.

The tuning advice contains the best variant as well as the search path. Figure 3.6 shows an advice of the MPI Parameters plugin. The structure of the tuning advice is the following:

- Name of the plugin, e.g., MPI Parameters Plugin
- The list of best scenarios
- The search path

The list of best scenario consist of a `ScenarioResult` which specifies the XML representation of the scenario (Section 3.3.4) and the list of objective results, represented as a `Scenario` and as a `Result`, respectively. Each `ScenarioResult` specifies the scenario id, the description, the tuned region, and the tuning specification. The tuning specification, `TuningSpecification`, consists of the tuning variant (`Variant`), the variant context (`VariantContext`) and ranks (`Ranks`). The variant defines a list of tuples between the parameter name and its value. Several results can be attached to each scenario. For example, this is used in the CFS plugin to store measurements gathered during the experiment for individual routines.

The list of `ScenarioResult` is also used to specify the search path.

One of the contributions of this thesis is the generation of the tuning advice that is automatically applied in the production environment.

```
<Advices>
    <Advice>
        <PluginName>MPI Parameter Plugin</PluginName>
        <BestScenarios>
            <ScenarioResult>
                <Scenario>
                    <ID>0</ID>
                    <Description/>
                    <Region>
                        <FileName>bt.f</FileName>
                        <FirstLine>189</FirstLine>
                        <LastLine>191</LastLine>
                    </Region>
                    <TuningSpecification>
                        <Variant>
                            <TuningParameter>
                                <Name>eager_limit</Name>
                                <Value>4096</Value>
                            </TuningParameter>
                            <TuningParameter>
                                <Name>buffer_mem</Name>
                                <Value>8388608</Value>
                            </TuningParameter>
                        </Variant>
                        <VariantContext>
                            <Type>REGION_LIST</Type>
                            <Region>
                                <FileName>bt.f</FileName>
                                <FirstLine>189</FirstLine>
                                <LastLine>191</LastLine>
                            </Region>
                        </VariantContext>
                        <Ranks>
                            <Type>ALL</Type>
                        </Ranks>
                    </TuningSpecification>
                </Scenario>
            </ScenarioResult>
        </BestScenarios>
        <SearchPath>
           <ScenarioResult> ... </ScenarioResult>
           ...
        </SearchPath>
    </Advice>
</Advices>
```

Figure 3.6: Example of a PTF advice for the MPI Parameters plugin in the XML advice format.

3.8 Summary and Current Limitations

The current design of PTF provides very flexible and powerful tuning framework with easy to write tuning plugins. The framework addresses most of the challenges in the performance tuning of current HPC applications. It also covers a wide range of relevant application aspects. However, PTF does not give enough attention to knowledge available in performance data gathered from previous executions. Moreover, the original design does not provide means to store this data permanently for later processing. Additionally, plugins concentrate on the tuning of just individual aspects completely ignoring tuning of multiple aspects. Furthermore, as tuning is very time consuming and requires execution of many experiments until the close-to-optimal combination of tuning parameter values are found, the user would benefit from the prediction of potential tuning benefit. This makes PTF an interesting context for further development and testing of new technologies making intelligent tuning of HPC applications with multiple aspects possible.

Chapter 4

Tuning Plugins

This chapter describes several tuning plugins which are used, developed or extended in the course of this thesis. The selection of individual plugins combined by the meta-plugin is influenced by the size of their search space. Therefore, only plugins with huge search spaces like the Compiler Flags Selection and the MPI Parameters plugins are employed by meta-plugins.

4.1 Compiler Flag Selection

4.1.1 Introduction

Compiler technology plays an important role in computer science since its invention. Besides its role to translate the high-level code into the correct lower-level code, it also plays important role in generating the program which executes more efficiently or faster. To support that role, today compilers offer a plethora of compiler transformations like loop unrolling, software pipelining, automatic parallelization, data prefetching, inlining, vectorization etc. The selection of optimizations is not trivial and requires knowledge of many interrelated interactions between the program and the underlying hardware with often unclear benefits to the performance or correctness. Therefore, a lot of effort was spent on automatic analysis of compiler optimizations and search for their optimal combination as described in Section 2.3.2.

Due to a large number of compiler optimizations exposed through compiler flags and the required good knowledge of the underlying transformations and their interactions with the system, it is very difficult for the programmer to select the best combination of flags.

To help the user in the evaluation of compiler optimizations PTF provides the Compiler Flag Selection (CFS) plugin. The plugin automates the vast majority of the actions the user usually does. The user is only required to provide the configuration file with the list

of compiler flags with their values which have to be taken into account. Using PTF, the CFS plugin selects different combinations of compiler flags, recompiles the application and evaluates its effects on the runtime. The best performing combination of optimizations is tracked with the search path, exposed to the user and stored in the advice.

The CFS plugin supports tuning of serial and parallel codes. It provides a flexible configuration system which allows the use of different target compilers. If the user does not specify the configuration file, the CFS provides its compiler-specific default configurations. The plugin also features per-file recommendation for significant files, based on PTF scenario execution analysis, and automatic partial recompilation of only significant files including both SPMD[1] and MPMD[2] parallel codes.

The user can select between one of the search algorithms distributed with PTF, such as exhaustive, individual and genetic search, but also plugin-specific search algorithm based on machine learning.

4.1.2 Analysis

The CFS plugin relies on the analysis capabilities of Periscope for both, a pre-analysis and a scenario analysis of each experiment. The pre-analysis is employed to identify files that are responsible for a significant share of the execution time. The scenario analysis is employed to perceive the effect of the experiment on the significant routines. Based on the information collected during the analysis, the plugin can provide the advice with recommended compiler optimizations for individual files instead of the entire application.

Significant Region Analysis

During the tuning process, the CFS plugin requires a recompilation of the code for each experiment. Code compilation includes many steps like lexical analysis, preprocessing parsing, semantic analysis, code generation, and code optimizations. This process can be quite time-consuming and as most code sections account for a small portion of execution bring negligible improvement of the complete execution time, according to Amdahl's law. Therefore, concentrating to more time-consuming sections of the code can be of significant interest.

To support such tuning approach the plugin executes a new analysis strategy called *Importance* analysis strategy before the search for the optimal combination of compiler flags. The strategy identifies the routines that have an exclusive execution time above a certain threshold with a default value of 10% of the phase execution time. For that purpose, the performance property called `ExecTimeImportance` is introduced.

[1]Single-Program Multiple-Data
[2]Multiple-Program Multiple-Data

$$Severity_s = \frac{T(s) - \sum\limits_{s' \in calls} T(s')}{phaseCycles} * 100 \quad (4.1)$$

Equation 4.1 specifies the severity of the property for a given subroutine s. The severity is computed as a percentage of exclusive execution time spent in the phase execution time. The exclusive execution time is computed by subtracting the execution time spent in called subroutines from the routines execution time. The condition of the property checks whether the severity is larger than the defined threshold, e.g., 10%.

The individual strategy is a single-step search strategy. The search strategy creates an `ExecTimeImportance` candidate property for each subroutine and each process. The property is disassembled into metric requests collected in a single experiment. The properties are evaluated and contain the execution time of each subroutine and of all call sites of that routine.

Many subroutines of an application are relatively small compared to the time necessary to make the measurement and therefore the measurement infrastructure brings the huge overhead to the measurement. Therefore, Periscope offers the user to control performance intrusion made by the monitor via *selective instrumentation*. Small subroutines can be excluded from instrumentation via the instrumentation configuration file and instrumenter command line options. The instrumenter used by the MRI monitor relies on the instrumentation configuration file *psc_inst_config*. The user controls instrumentation of individual files by specifying the region types to be instrumented. While instrumenting with Score-P instrumenter, the user can define which region types should be instrumented through the command line but also by specifying the filter file. This approach does not work with files that contain several routines, of which some are not small. In that cases, selective instrumentation can be combined with one of Periscope's *automatic reinstrumentation strategies*.

Three automatic reinstrumentation strategies are provided with Periscope: `overhead`, `all_overhead`, `analysis`. The `overhead` strategy estimates the instrumentation overhead, while `all_overhead` strategy measures the overhead introduced by the instrumentation. The `analysis` instrumentation strategy combines both strategies with a performance analysis strategy. The CFS plugin can employ the `all_overhead` instrumentation strategy with the `Importance` analysis strategy. In the first run, the importance strategy determines all regions whose instrumentation overhead is higher than 5%. In the second run, the instrumentation strategy disables instrumentation of those regions and the application is recompiled. After the instrumentation is refined, the `Importance` analysis strategy is executed and the overhead of instrumentation is limited by the predefined percentage for the automatic reinstrumentation strategy.

The plugin uses the properties proven during the pre-analysis to identify the significant files that are to be recompiled for each experiment.

Additionally, to the new analysis strategy, the CFS plugin is capable of using the profil-

ing support, called Profile-Guided Optimization(PGO), built into Intel's C and Fortran compilers. Due to its sampling approach, the PGO function profiling provides the same information but with low overhead. The CFS plugin is capable of extracting the significant files from PGO's output. However, contrary to the `Importance` strategy this information is not used to guide the selection of file-specific tuning flags combinations.

Scenario Analysis

During the tuning process, the tuning plugin requests the objective property to perceive the effect of the scenario on the execution time of the tuned region, i.e., the phase region. The property that captures the execution time for a region is the `ExecTime` property. Besides that, to perceive the effect of the compiler flags on other regions the plugin requests the configurable analysis strategy. The configurable analysis is configured to collect the `ExecTime` property for the significant regions, which were determined during the pre-analysis. To ensure that all properties are reported, the analysis strategy is configured with the pedantic flag.

The severity of the reported properties represents the percentage of the region execution time in the phase region execution time. Besides the severity, the property also provides the additional information in the form of the real execution time, represented as a floating point value.

4.1.3 Tuning

The CFS plugin provides powerful configuration system to express the tuning parameters representing the individual compiler flags. The configurations are provided through configuration files. The plugin is distributed with the default configuration file for commonly used compilers like Intel C and Fortran compilers. Besides the default configuration file, the compiler also supports user defined configuration files. Through the configuration file, the user can also specify the search algorithm and algorithm-specific configuration used to explore the search space.

Tuning Parameters

The compiler flags are represented in the CFS as tuning parameters. The tuning parameters and the values they take can be represented with various data types like string or numbers. Their values can be represented as lists of strings or numbers or as ranges of values.

```
1 tp "TP_IFORT_OPT" = "-O" ["0", "1", "2", "3"];
2 tp "TP_IFORT_XHOST"  = " " ["-xhost", " "];
3 tp "TP_IFORT_PREFETCH" = " " ["-opt-prefetch", " "];
4 tp "TP_IFORT_UNROLL" = "-unroll=" [5,20,5];
5
6 search_algorithm="individual";
7 individual_keep=1;
```

Figure 4.1: Example CFS configuration file.

Configuration File

To parse the configuration file, the CFS plugin defines its own lexical analyzer and the parser. The parser is capable of configuring the plugin based on the configurations provided by the user in the form of the configuration file. The configuration file specifies information on how to issue the build of the application, the compiler flags and their possible values, and the search algorithm and its configuration. Through the configuration file, the user can also indicate a compiler-specific default configuration file. The plugin distributes several compiler-specific default configuration files with a list of important compiler flags worth exploring. The selection of compiler flags in these configuration files is based on experiences of the developers of the CFS plugin. These files can be a good starting point for the user, lead to improved tuning time and the quality of the final advice.

Let's consider an Example of the Intel Fortran configuration file depicted in Figure 4.1. In lines 1-4 the tuning parameters are defined with its values. The first, `TP_IFORT_OPT` has four possible values defined as a list of values, namely, -O0, -O1, -O2, and -O3. Parameters `TP_IFORT_XHOST` and `TP_IFORT_PREFETCH` switch -xhost and -opt-prefetch on and off. The tuning parameter `TP_IFORT_UNROLL` is defined as a range from 5 to 20 with step size 5. The last two lines select the search algorithm and configure it. More detailed description of the search algorithm is given later in this section.

Recompilation of the Application

During the tuning process, the CFS plugin often recompiles the application with a new combination of compiler flags to evaluate their effects on its performance. Therefore, the special care has to be taken on how to recompile it and how to transfer and apply the new combination of flags. The CFS plugin supports a well-known *make* build automation tool. As PTF lacks the native support for automatic recompilation the plugin provides its own support.

The support relies on the configuration file to provide the list of compiler flags which represent the tuning parameters. The tuning parameters are used to build the variant

space. The variant space is given to the search algorithm with the tuned region. The search algorithm creates the scenarios and they are provided for the preparation. In the preparation step, the variant is extracted from the scenario. According to the variant and the configuration from the file, the compile line is prepared. For each tuning parameter, the flag prefix and its value are concatenated into a single line of compiler flags. For the example from the previous subsection, the compile line could be: "-O2 -host -opt-prefetch -unroll=15".

The application makefile may require little modification from the user in the sense that the user inserts the optimization variable to the proper place and provide its name in the configuration file. This optimization variable is used to enforce the optimization in the build process.

The compile line is transferred through the optimization variable specified on the make command line. For our example that could be: "make OPT_FLAGS="-O2 -host -opt-prefetch -unroll=15"".

The build automation tool is informed about the files selected for the recompilation. To apply the pre-execution runtime action the application is now recompiled on the local or on a remote system. The reason to compile on the remote system comes from the fact that the local compute nodes where PTF runs often have stripped down version of the operating system which makes recompilation impossible. Therefore, PTF compiles the application on the login nodes which are accessed remotely through ssh. The configuration file provides details for the remote or the local make command execution.

Now, the application is recompiled and the experiment is executed.

In the configuration file, the user can enable the selective recompilation introduced in Subsection 4.1.2. The selective recompilation reduces the time necessary to recompile the application with minimum effects on the potential tuning benefit according to the Amdahl's law. The user can specify the significant files in the configuration file, or they can be determined during the runtime by the plugin through the significant region pre-analysis. As the order of optimization passes might affect the generated code, the compiler flags from the configuration file are kept in the order.

CFS Search Strategies

The CFS plugin supports all standard PTF search strategies described in Section 3.4. Two search strategies were developed especially for the CFS plugin, the individual strategy and the probabilistic random strategy. The individual strategy is a standard PTF search strategy based on the observation that some compiler flags have a higher impact on the performance of the application than other compiler flags. In our example, the optimization levels, i.e., -O2 or -O3, will have a higher impact on the performance than -xhost or -opt-prefetch [61]. Consequently, one could order the corresponding tuning parameters based on their estimated impact on performance. Based on the presumption that ordering is

done properly, the individual search could substantially speed up the tuning process and still find close-to-optimal configuration.

The probabilistic random search strategy is a search algorithm based on machine learning [48]. An overview of the integration of the search algorithm into the CFS plugin is depicted in Figure 4.2. The strategy assumes that tuning parameters are independent and that individual probability function can be determined.

The search algorithm relies on the *tuning database* to store the data collected from the previous tunings with the CFS plugin. The collected data consists of the unique application identifier, the dynamic white-box program signature, and the objective value. The unique identifier is determined from the hash of the source code.

The program signature consists of various events gathered through PAPI like caching, TLB, branching, different types of instructions, etc. It is collected during the pre-analysis step with the configurable analysis strategy. The purpose of the signature is to characterize the application similar to methods presented in Chapter 2. This method 2.2 compares the application signature with other signatures and makes predictions based on similar programs.

The selection of individual scenarios from the search space is controlled by the probabilistic random search. During the initialization of the search algorithm, for each program, several best performing scenarios are selected from the database to guide the search. Based on the selected scenarios, the probability model decides which tuning parameter values are selected. The selection of the probability model is guided by one of the classification techniques, 1-nearest neighbor or support vector machines [47]. For example, the 1-nearest neighbor technique determines the most similar application and computes probability mass function for each tuning parameter.

Once the search strategy is configured, it generates a fixed number of scenarios. These scenarios are evaluated and their results are stored in the tuning database for future use.

The configuration file allows configuring the budget to limit the search time.

4.1.4 Complete Tuning Flow

Figure 4.3 gives an overview of the complete tuning flow of the CFS plugin.

Plugin initialization: During the initialization, the plugin reads the configuration file and populates the tuning space with the tuning parameters from the specified compiler flags. After the tuning space is populated the requested search algorithm is loaded, initialized and configured. Here also the tuning time budget is defined.

Start tuning step: The tuning process of the CFS plugin requires a single tuning step. However, the tuning step consists of two nested loops.

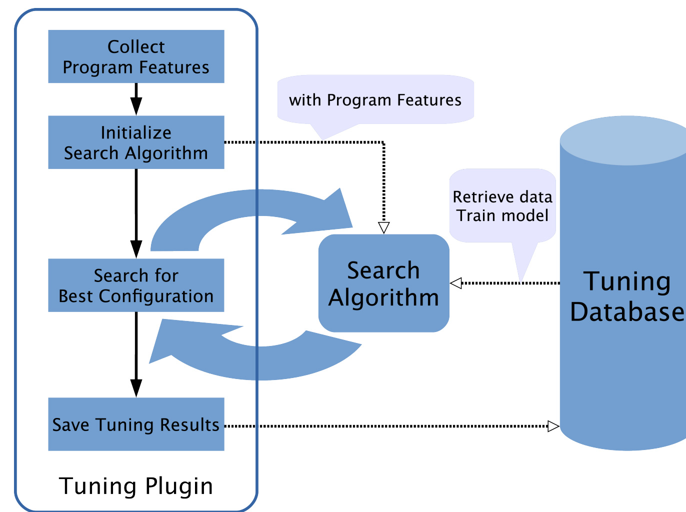


Figure 4.2: The machine learning-based search strategy uses historical performance data to guide the selection of good candidate scenarios. Historical performance data in the tuning database which is then used to configure probabilistic random search with probability functions for each tuning parameter.

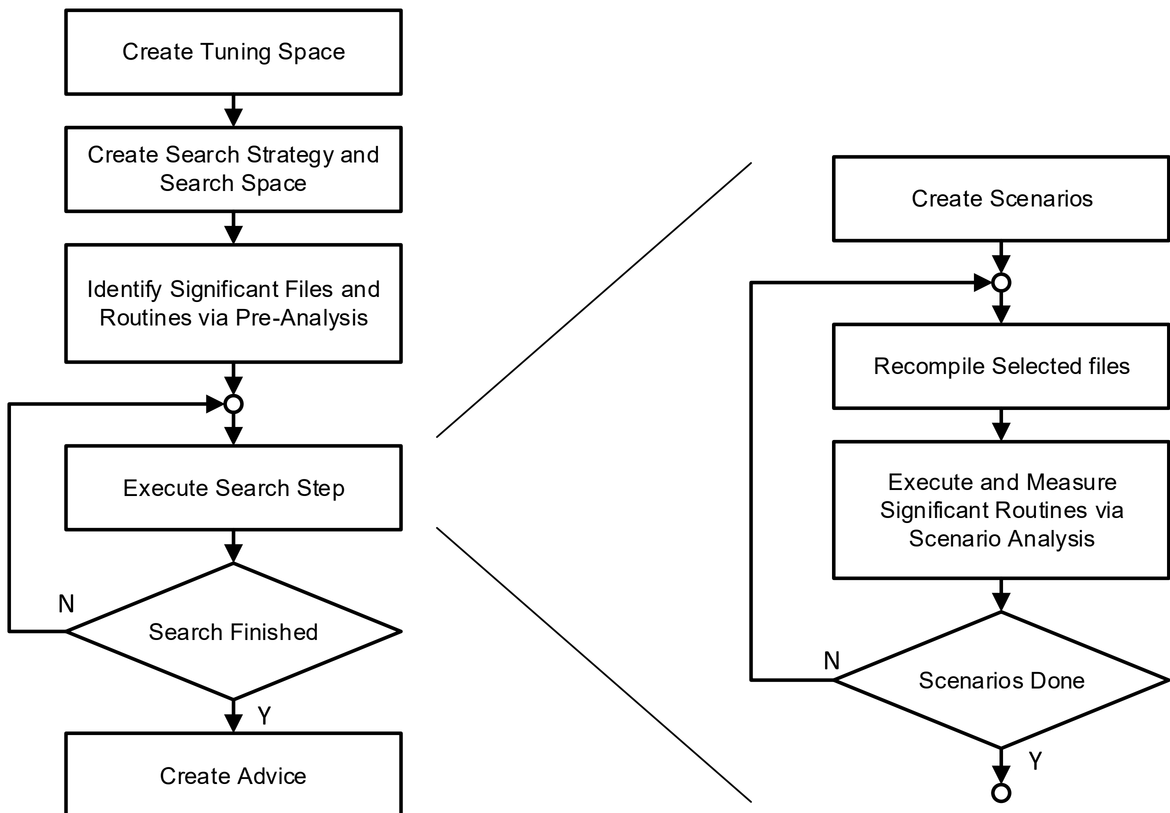


Figure 4.3: Tuning approach of the CFS plugin.

The number of iterations of the outer loop depends on the number of search steps of the search algorithm. For example, feedback driven search algorithms like iterative search require multiple search steps as the algorithm evaluates a single parameter per search step. When it finds the optimum for that parameter it proceeds with the next search step and a new parameter. On the other hand, search algorithms like the exhaustive search and machine learning search require only one iteration to generate all scenarios to be evaluated.

The inner loop controls the execution of scenarios generated by the current search step. The number of scenarios generated again depends on the search algorithm. The exhaustive search generates all possible scenarios from the tuning space, i.e., the cross-product of all tuning parameters with its values, while for others like individual generate the number of scenarios equal to the cross product of the number of values chosen from the previous and the current tuning parameter. For the machine learning search, the probabilistic random search generates a fixed number of scenarios which are predicted to be the most promising.

Pre-analysis: The pre-analysis is used to identify the significant regions. For that purpose, the CFS plugin employs the Intel profiling feature or the *Importance Analysis Strategy* (Section 4.1.2).

The pre-analysis strategy can be requested only once per tuning step. Therefore, the machine learning uses the pre-analysis to determine the baseline execution time used to normalize the data used by the probabilistic random search. It recompiles the application with the baseline compiler flags. Moreover, it determines the unique application identifier and the signature for the phase.

Create scenarios: In the create scenarios, the plugin identifies significant files based on the results from the pre-analysis and prepares them for recompilation. The resulting properties are created for each region and each process. Each property contains information to which file the region belongs. The files with the significant execution time are identified and stored in a list. From the list, the duplicates are eliminated. If the file contains at least one significant region executed by any process, the file is marked as significant and prepared for recompilation. With such approach, MPMD applications are equally well treated.

From each significant file, the most significant routine is identified and stored in a list. The list is later used to decide which regions are going to be evaluated by the scenario analysis strategy.

In the machine learning mode, the signature is extracted from the collected properties. Now, the plugin provides the search algorithm with the search space and for the machine learning strategy the signature. The search algorithm generates the scenarios that are to be evaluated during the tuning step. Depending on the search strategy, all scenarios might be generated or just a limited number of them. For the exhaustive search, all the scenarios are evaluated in a single search step. Other

strategies like the individual strategy generate only a starting set of scenarios. The machine learning strategy generates a fixed number of scenarios which are highly probable good ones.

Prepare and execute scenarios: In these steps, the compiler flags are extracted from the scenario tuning specification and the application is recompiled.

With the experiment, the plugin attaches the importance analysis strategy to perceive the effect of the experiment on the most significant regions from the file. The results of the experiment and its effect on the significant regions are stored in the result pools for the later inspection during the advice generation.

Give advice: Once the tuning has finished the plugin generates the tuning advice. The tuning advice is generated using the built-in support in PTF. The advice consists of two parts:

1. The global best configuration. The selection of the configuration for the phase region is done with the help of the tuning objective function.
2. The best configuration per-file. The selection of these configurations is done through the scenario analysis of significant regions per-file. For the significant regions, the analysis strategy evaluates the `ExecTime` property. For each file, the best combination of compiler flags is reported in the file.
3. The search path.

The global best configuration offers the single best combination of compiler flags for the whole application. Such configuration can always be applied during compilation.

On the other hand, the best configuration per-file provides the best combination of compiler flags for individual files. However, deploying the optimal combination of flags to individual files might require changes in the application build system, e.g., makefiles. For large applications, this might not always be possible.

Additionally to best configurations, the plugin dumps the search path into the advice file. The advice file contains entries with each individual combination of evaluated compiler flags, their overall execution time, and the execution time of their significant regions.

To simplify the future analysis of the results and enable the creation of reports with tools like Excel, the plugin outputs the search path information as a CSV file.

Plugin finalization: Stores the program signature and the tuning case which consists of the unique application identifier and the variant.

4.1.5 Summary

The presented plugin is used by the meta-plugin as it provides user configurable search space. The search space is large and therefore the search process may take a considerable amount of time. Therefore, it is advised that the plugin's tuning potential is determined in advance by machine learning that tuning process is not ran if little or no improvement is expected.

4.2 Compiler Flags Selection for OpenCL

[3]

4.2.1 Introduction

The next grand challenge for large-scale computing is to reach exascale-level performance in the 2019-2022 timeframe within the 20 MW power constraint. To drive the improvement in performance and keep the continuing constraint on power have led the vendors to utilize the classical multi-core CPUs with accelerators, which form a heterogeneous system. Writing applications for the heterogeneous system require using one of available heterogeneous software platforms. OpenCL, one of the platforms, is a cross-platform programming interface for parallel applications on heterogeneous systems. Applications written in OpenCL [9] support various types of standard processors and accelerators like multi-core CPUs, many-core GPUs, FPGAs, and many other application-specific hardware. Although OpenCL provides functional portability of codes across different multicore and manycore platforms, it does not guarantee performance portability. It has been shown, that the performance of OpenCL codes may vary significantly even between different generations from a single GPU vendor, let alone different vendors. Also, due to the very different architectural characteristics of CPUs, GPUs, and co-processors like the Intel Xeon Phi, it is even more challenging to achieve performance portability of OpenCL-based applications.

Optimizing OpenCL applications requires carefully tuning many aspects. An important role in the performance of an OpenCL application is tuning of kernels. Kernels can be tuned by the choice of the algorithms, on the implementation level, work distribution level, but also importantly on the compiler level. In OpenCL, a kernel can be compiled either online or offline. When compiled online, the kernel is built from the source during runtime, just-in-time, using the OpenCL runtime library. When compiled offline, the kernel is pre-build using an OpenCL compiler, and the generated binary gets loaded using the OpenCL API. By the standard online compilation can be influenced by a set of

[3] This is a minor revision part of the work published in 49th Hawaii International Conference on System Sciences - HICSS 2016, Koloa, HI, USA, January 5-8, 2016

compiler optimization options. These options, however, trade performance for correctness. OpenCL by the standard supports vendor specific extensions, which were available only for NVIDIA compiler. However, during testing, they have proven not to have any influence on the kernel. On another hand, offline compilation supported by the external compiler, from vendors like Intel, provide a wide range of compiler options. Therefore, we have decided to support Intel Xeon Phi through it.

To support this very important trend in computing in the context of this thesis we have developed a support for OpenCL compiler flags. As OpenCL compiler flags share many similarities with the existing Compiler Flags Selection plugin we have extended the CFS plugin with OpenCL support into the resulting Compiler Flags Selection for OpenCL plugin (CFS for OpenCL).

The CFS for OpenCL plugin supports tuning OpenCL kernels with the help of compilation flags presented to an OpenCL offline compiler. The plugin automatically searches for the best combination of OpenCL compiler flags used to build an OpenCL kernel. The user is required to provide a predetermined list of compiler flags in a compiler-specific configuration file. This configuration file can be enhanced with application specific flags and build information to guide the execution of the plugin.

The plugin constructs flag combinations that are evaluated by recompiling the application's kernels and automatically executing the application. The recompilation is based on the application's makefile that exposes an environment variable that takes the selected flag combination. After the plugin has selected a flag combination it touches the kernel source files and triggers the make process. After the build is finished, the application is automatically restarted by PTF and execution time is measured for the OpenCL kernels. When the evaluation of the selected combinations has finished, the plugin analyzes the measured execution times and outputs the best flag combination per kernel.

Like the regular CFS plugin, the extended plugin is able to use one of PTF's generic search strategies.

4.2.2 Analysis

The CFS for OpenCL plugin relies on the analysis capabilities of Periscope for both, a pre-analysis and a scenario analysis of each experiment. The pre-analysis is employed to identify files that are responsible for a significant share of the execution time. The scenario analysis is employed to perceive the performance impact of the compiler optimization used in the experiment on the kernels. The plugin executes a new analysis strategy, developed for that purpose, called *OpenCL* analysis strategy. Based on the information collected during the analysis, the plugin can provide the advice with recommended compiler optimizations for individual kernels instead of the entire application.

Significant Region Analysis

During the tuning process, the CFS plugin requires a recompilation of the code for each experiment. Code compilation includes many steps like lexical analysis, preprocessing parsing, semantic analysis, code generation, and code optimizations. This process can be quite time-consuming and as most code sections account for a small portion of execution bring negligible improvement of the complete execution time, according to Amdahl's law. Therefore, concentrating on more time-consuming sections of the code can be of significant interest.

To support such tuning approach the plugin executes a new analysis strategy called *Importance* analysis strategy before the search for the optimal combination of compiler flags. As the CFS for OpenCL plugin is an extension of the CFS plugin, the strategy is introduced in Section 4.1.2.

Scenario Analysis

During the tuning process, the tuning plugin requests the objective property to perceive the effect of the scenario on the execution time of the tuned region, i.e., the phase region. Besides, the effect on the tuned region the user is often interested in perceiving the effect of the scenario on the individual OpenCL kernels. Therefore, the OpenCL tuning strategy was implemented in the Autotune project. The analysis strategy evaluates the `KernelExecutionTime` property for the kernels executed inside the phase region. The property reports the kernel name and its execution time.

The requested analysis strategy is propagated to the tuning strategy as a sub-strategy. The strategy is triggered implicitly during the experiment by the analysis agents. The agents instruct the monitor, which is linked to the application to measure the execution of the kernels. Once the kernel has been started, the monitor waits for its execution and retrieves the execution time and then retrieves the execution time. At the end of the experiment, which can either be the end of the application or the end of the phase region, the aggregated execution time of each kernel is returned to the analysis agent. The analysis agent creates a performance property for each kernel and propagates it through the hierarchy to the plugin. To ensure that `KernelExecutionTime` properties are reported for all kernels, the analysis strategy is configured with the pedantic flag.

For each tested flags combination the execution time is available for all the kernels and thus, the plugin can determine the best combination for each kernel.

Monitoring

Evaluation of OpenCL kernels requires insight into the execution information about the OpenCL kernel execution, like kernel execution time. To better understand requirements to gather such information, Figure gives an overview of the OpenCL kernels execution.

Before OpenCL can execute the kernel, one or more devices has to be associated with an OpenCL context. Contexts are used by the OpenCL runtime for managing command-queues. When the context is created, a command-queue for a specific device from the context has to be created. During the command-queue creation, it is possible to enable profiling. This is done by providing a command-queue property. When a command-queue is created, the memory buffers have to be allocated on the device and attached to the context. Actions, e.g., create memory buffers, relevant to the device used by the context are stored in the command-queue as commands. When the buffer is created, a program from the source or precompiled program has to be created by providing a source code or binary buffer and its size. When the program is created, it is built and the program object gets created. From the created program a kernel is created. As one of the parameters of kernel creation routine, the kernel name is provided. When the kernel is created, the kernel argument values have to be provided. To launch the kernel, besides the kernel object, the number of dimensions used to specify the global work-items and work-items in the work-group has to be provided. Work-items are threads in terms of its control flow and its memory model. Work-groups are a set of work-items that must be able to make progress in the presence of barriers executed on a single compute unit, e.g., a CPU core. With the kernel launch, an event object could be specified which communicate the status of commands in OpenCL. Event objects can be used to synchronize operations in a context but also more important for tools community to track the status of a command and its profiling information. It can collect timestamps when the command was enqueued in the command-queue by the host, when it was submitted by the host to the associated device, when it was started on the device and when it finished its execution. After the device has executed all kernels on data is transferred back to the host.

OpenCL interposition library

The profiling of OpenCL is based on a library interposition technique. Library interposition is a powerful technique which allows interception of calls to arbitrary OpenCL run-time functions. The approach is based on creating a wrapper function around the OpenCL API function. The wrapper library calls OpenCL functions it wraps but also stores information necessary for profiling.

The interposition takes place during the automatic instrumentation through the techniques called link-time interposition. During the compilation of the OpenCL program, the instrumenter links the wrapper library first followed by the OpenCL library.

Based on a wrapper library of the OpenCL API functions, the monitor uses a given OpenCL event object or creates a new event object to measure the kernel execution time.

During the creation of the command-queue, the wrapper enables OpenCL profiling. When the kernel gets created, the wrapper stores the kernel and its name for later use. During the kernel launch call, the wrapper creates a new event object or uses a given event object, launches the kernel, synchronizes the kernel, calculates the execution time and a number of

kernel execution. The event object allows the monitor to calculate the execution time from the OpenCL event timestamps, i.e., when the kernel was started on the device and when it finished its execution. When the program or the phase region ends, the information collected from the wrapper gets reported to the analysis agent which propagates it through the hierarchy to the CFS for OpenCL plugin.

The following functions are relevant for profiling:

clCreateCommandQueue: creates the command queue. During the creation of the command queue, the monitor enables profiling.

clCreateKernel: creates the kernel from the program object with the specified name. The monitor stores the kernel name to identify the kernel. This name is attached to the property and allows the plugin to identify for which kernel it is collected.

clEnqueueNDRangeKernel: launches the kernel. With this call, the event object is provided. The event object is later used to retrieve the execution time.

clGetEventProfiling: retrieves the profiling information for the event command like the timestamp of kernel start or end.

4.2.3 Tuning

Due to a large number of tuning flags, the exhaustive analysis might not be possible. Therefore, the CFS for OpenCL plugin provides several search algorithms to reduce the number of evaluations. Besides the individual search, which is described in Section 3.4.2, it can apply also a machine learning based random search, as well as the GDE3 genetic search [57].

The CFS for OpenCL plugin provides powerful configuration system to express the tuning parameters representing the individual compiler flags. The configurations are provided through configuration files. The plugin is distributed with the default configuration file for commonly used optimizers/compilers like Intel oclopt and NVIDIA. Besides the default configuration file, the compiler also supports user defined configuration files. Through the configuration file, the user can also specify the search algorithm and algorithm-specific configuration used to explore the search space.

Tuning Parameters

The compiler flags are represented in the CFS for OpenCL plugin as tuning parameters. The tuning parameters and the values they take can be represented with various data types like string or numbers. Their values can be represented as lists of strings or numbers or as ranges of values.

```
1 opencl_tuning;
2 tp "TP_OCLOPT_OPT" = "-O" ["1", "2", "3"];
3 tp "TP_OCLOPT_VECTORIZE" = " " ["-loop-vectorize", " "];
4 tp "TP_OCLOPT_PREFETCH" = " " ["-prefetch", " "];
5
6 search_algorithm="individual";
7 individual_keep=1;
```

Figure 4.4: Example CFS for OpenCL configuration file.

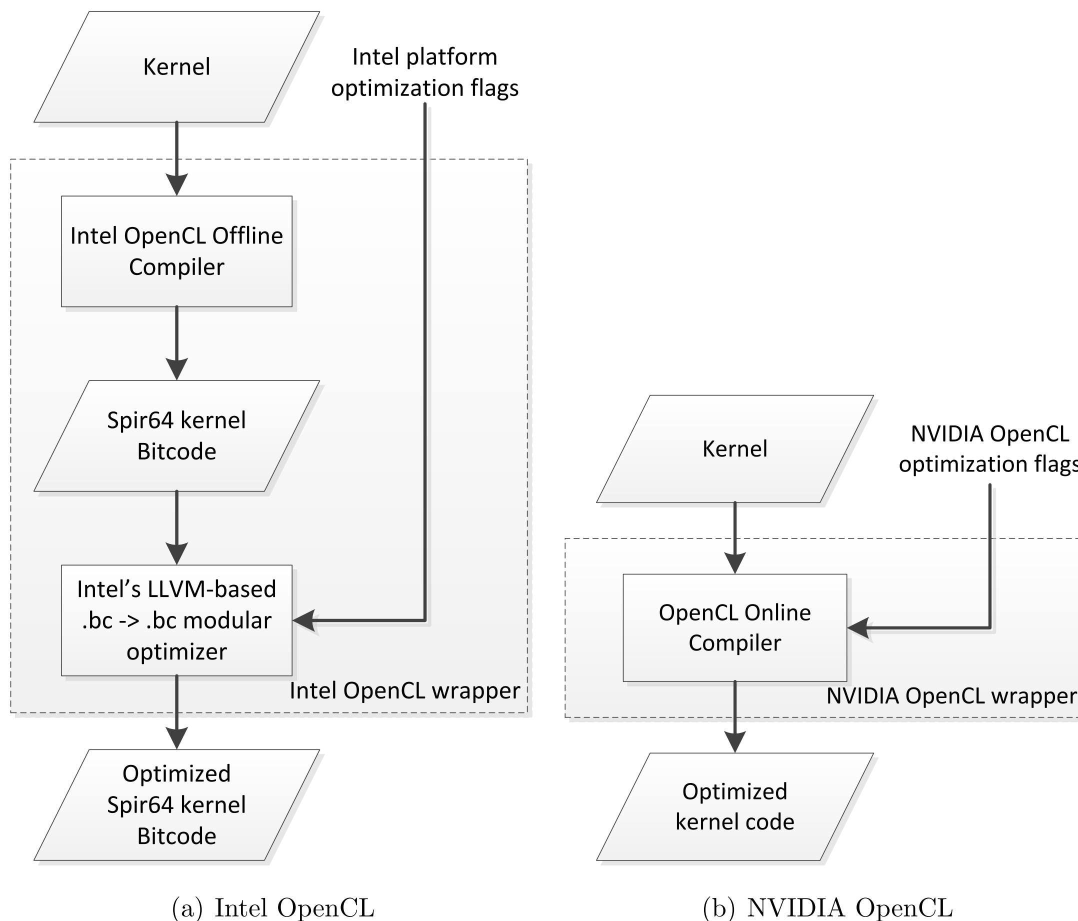
Configuration File

The user is required to provide a predetermined list of compiler flags in a compiler-specific configuration file. This configuration file can be enhanced with application specific flags and build information to guide the execution of the plugin. To parse the configuration file, the CFS for OpenCL plugin extends the lexical analyzer and the parser of the CFS plugin. The parser is capable of configuring the plugin based on the configurations provided by the user in the form of the configuration file. The configuration file specifies information on how to issue the build of the application, the compiler flags and their possible values, and the search algorithm and its configuration. The OpenCL support in the CFS for OpenCL plugin is enabled with the flag in the configuration file. Through the configuration file, the user can also indicate a compiler-specific default configuration file. The plugin distributes several compiler-specific default configuration files with a list of important compiler flags worth exploring. The selection of compiler flags in these configuration files is based on experiences of the developers of the CFS plugin. These files can be a good starting point for the user, lead to improved tuning time and the quality of the final advice.

Let's consider the example depicted in Figure 4.4. The example represents the Intel OpenCL compiler optimizer configuration file. In line 1, the OpenCL kernel offline compilation is enabled for the CFS for OpenCL plugin. In lines 2-4 the tuning parameters are defined with its values. The first, `TP_CLOPT_OPT` has three possible values defined as a list of values, namely, -O1, -O2, -O3. Parameters `TP_CLOPT_VECTORIZE` and `TP_CLOPT_PREFETCH` switch -loop-vectorize and -prefetch on and off. The last two lines select the search algorithm and configure it. More detailed description of the search algorithm is given later in this section.

Recompilation of the Application

Before a scenario is executed, the kernels that are to be tuned have to be recompiled with the specified flags. Recompilation of a kernel in PTF is specific to each OpenCL vendor and is therefore performed by a script that is executed by the plugin. PTF currently supports two OpenCL vendors, Intel and NVIDIA, and their target devices. Intel provides



(a) Intel OpenCL

(b) NVIDIA OpenCL

Figure 4.5: Offline compilation flow.

a kernel builder and an LLVM-based optimizer while NVIDIA does not provide an offline compiler. The compilation of a kernel with the Intel toolchain is depicted in 4.5(a). First the `ioc` tool [24] generates the *SPIR (Standard Portable Intermediate Representation)* file, according to selected OpenCL compiler flags. This intermediate version is input to the `oclopt` optimizer, which generates on optimized code version in form of a new SPIR file. The optimization flags defined by the OpenCL standard relax mathematical precision and thus trade correctness for speed. The `oclopt` optimizer provides a large number of standard compiler optimization flags. To perform the optimizations it executes a number of passes including specific analyses and optimizations.

As shown in Figure 4.5(b), for NVIDIA we have developed an offline compiler which wraps the online compiler and uses besides the OpenCL standard optimizations two additional optimizations defined by NVIDIA, i.e., general optimization levels and the maximum register count. The selected optimizations are propagated to NVIDIA's `ptxas` optimizing backend compiler. In addition, PTF supports loop unrolling based on NVIDIA's loop

unroll pragma extension. The programmer can insert the unroll pragma into the code and direct the unroll factor with a tuning parameter given to the plugin via the configuration file.

4.2.4 Complete Tuning Flow

Plugin initialization: During the initialization, the plugin reads the configuration file and populates the tuning space with the tuning parameters from the specified compiler flags. After the tuning space is populated the requested search algorithm is loaded, initialized and configured. Here also the tuning time budget is defined.

Start tuning step: The tuning process of the CFS for OpenCL plugin requires a single tuning step. However, the tuning step consists of two nested loops.

The number of iterations of the outer loop depends on the number of search steps of the search algorithm. For example, feedback driven search algorithms like iterative search require multiple search steps as the algorithm evaluates a single parameter per search step. When it finds the optimum for that parameter it proceeds with the next search step and a new parameter. On the other hand, search algorithms like the exhaustive search and machine learning search require only one iteration to generate all scenarios to be evaluated.

The inner loop controls the execution of scenarios generated by the current search step. The number of scenarios generated again depends on the search algorithm. The exhaustive search generates all possible scenarios from the tuning space, i.e., the cross-product of all tuning parameters with its values, while for others like individual generate the number of scenarios equal to the cross product of the number of values chosen from the previous and the current tuning parameter. For the machine learning search, the probabilistic random search generates a fixed number of scenarios which are predicted to be the most promising.

Pre-analysis: The pre-analysis is used to identify the significant regions. For that purpose, the CFS for OpenCL plugin employs the *Importance Analysis Strategy* (Section 4.1.2).

The pre-analysis strategy can be requested only once per tuning step.

Create scenarios: In the create scenarios, the plugin identifies significant files based on the results from the pre-analysis and prepares them for recompilation. The resulting properties are created for each region and each process. Each property contains information to which file the kernel belongs. The files with the significant execution time are identified and stored in a list. From the list, the duplicates are eliminated. If the file contains at least one significant kernel executed by any process, the file is marked as significant and prepared for recompilation. With such approach, MPMD applications are equally well treated.

From each significant file, the most significant routine is identified and stored in a list. The list is later used to decide which regions are going to be evaluated by the scenario analysis strategy.

Now, the plugin provides the search algorithm with the search space and for the machine learning strategy the signature. The search algorithm generates the scenarios that are to be evaluated during the tuning step. Depending on the search strategy, all scenarios might be generated or just a limited number of them. For the exhaustive search, all the scenario are evaluated in a single search step. Other strategies like the individual strategy generate only a starting set of scenarios.

Prepare and execute scenarios: In these steps, the compiler flags are extracted from the scenario tuning specification and kernels are recompiled.

With the experiment, the plugin attaches the OpenCL analysis strategy to perceive the effect of the experiment on kernels inside the phase region. The results of the experiment and its effect on kernels are stored in the result pools for the later inspection during the advice generation.

Give advice: Once the tuning has finished the plugin generates the tuning advice. The tuning advice is generated using the built-in support in PTF. The advice consists of two parts:

1. The global best configuration. The selection of the configuration for the phase region is done with the help of the tuning objective function.
2. The best configuration per-kernel. The selection of these configurations is done through the scenario analysis of kernels. The analysis strategy evaluates the `KernelExecutionTime` property for each kernel. The best combination of compiler flags is reported for each kernel in the advice file.
3. The search path. The configurations in order in which they were evaluated by the search algorithm.

The global best configuration offers the single best combination of compiler flags for the whole application. Such configuration can always be applied during compilation.

On the other hand, the best configuration per-kernel provides the best combination of flags for individual kernels. However, deploying the optimal combination of flags to individual kernels might require changes in the application build system, e.g., makefiles. For large applications, this might not always be possible.

Additionally to best configurations, the plugin dumps the search path into the advice file. The advice file contains entries with each individual combination of evaluated compiler flags, their overall execution time, and the execution time of their significant regions.

To simplify the future analysis of the results and enable the creation of reports with tools like Excel, the plugin outputs the search path information as a CSV file.

Plugin finalization: Stores the program signature and the tuning case which consists of the unique application identifier and the variant.

4.3 MPI Parameters

4.3.1 Introduction

Today's HPC systems are dominantly clusters with a fast interconnect. Therefore, many HPC applications rely on interprocess communication for data sharing on these systems. MPI [36] is the "de facto" standard for interprocess communication in distributed parallel applications. Thus a well optimized MPI implementation is one of the critical factors to achieve good performance and scalability. However, for portability reasons, the MPI developers do not optimize their implementation for the specific system. Therefore, the implementation might not perform optimally on each system environment and might require fine-tuning. For that purpose, most MPI implementations provide a set of configuration parameters that allows customization of the MPI runtime system to each system needs. These fine-tuning parameters are usually set by system vendors but also by the users to optimize their MPI application. The popular MPI implementations like OpenMPI, IBM MPI and Intel MPI offer between 50 and 150 of these parameters and many of them influence performance. Access to these parameters is available through MPICH's parameter interface, OpenMPI's MCA (Modular Component Architecture), or since MPI 3.0 with the standardized MPI-T interface. Due to a large number of MPI parameters and their unclear interaction with the application and the system a deep knowledge is a must. Even with a deep knowledge of MPI parameters, manual tuning is not recommended as it is very time-consuming.

The importance of the MPI parameters on the application performance has been demonstrated by the tuning tools such as mpitune [23] and OPTO [17].

Due to the immense importance of MPI on HPC applications and its effect on their performance, Universitat Autónoma de Barcelona has developed the MPI Parameters plugin in the context of the Autotune Project. The goal of the MPI parameters plugin is to automatically find the optimal values of the user defined subset of MPI parameters. From the provided MPI parameters, the plugin generates a set of scenarios evaluated during the application run. The results of the evaluation are presented as Periscope properties proven from the measurements collected on-line during the application run. Based on these results, the plugin makes tuning decisions and anticipates (predicts) which scenarios to evaluate.

The available MPI configuration parameters depend on the implementation of the MPI library and therefore it is impossible to find a general model to guide the search. To make things worse, the number of tuning parameters that the user can select and their often unrestricted range of values create the enormous variant space. Despite that, it is possible

to reduce the parameter ranges and hence prune the variant space. Therefore, the plugin implements the model based on collected message distribution that reduces the range of the eager limit and the memory buffer size parameters and effectively shrinks the variant space.

To guide the search the user can select one of the search algorithms distributed with PTF, such as exhaustive, individual, and genetic search.

4.3.2 Analysis

The MPI Parameters plugin relies on the analysis capabilities of Periscope for a pre-analysis. The pre-analysis is employed to reduce the range of the important MPI parameters, i.e., the eager limit and the memory buffer size and thus effectively reduce the search space. The analysis strategy depends on the distribution of the message sizes to analyze these two important MPI parameters.

Analysis Background

The main motivation to use the analysis is to reduce the search space and search the rest of it efficiently.

Let's consider the case where the user wants to tune an MPI application by finding the optimal combination of 5 MPI parameters using IBM MPI on the SuperMUC, a system installed in Leibniz Supercomputing Centre.

The parameters might be:

`eager_limit`: ranges from few bytes to 64 KiB.

`buffer_mem`: from 4 KiB to 2 GiB.

`pe_affinity`: yes/no

`task_affinity`: core/cpu.

`polling_interval`: from 1×10^6 to $2 \times 10^6 \mu s$.

As the search space size is defined as the cross-product of parameters and their value ranges, the search space could easily consist of 1024000 (32x80x2x2x100) possible scenarios. Evaluating such a huge number of scenarios exhaustively is almost impossible even for programs whose execution take only a few seconds. Therefore, in the most practical cases exhaustively finding the optimal combination of MPI parameters that handle the communication among processes in parallel MPI applications is not feasible due to huge search space.

In such cases, scientists try to build an analytic model that predicts the behavior of each configuration in the search space. Due to a large number of MPI parameters that the user can select and that available parameters depend on the implementation, this is non-trivial.

Therefore, the plugin provides a specific strategy that reduces the range of parameter values and hence prunes the search space. The strategy estimates the reduced parameter ranges for two parameters, the *eager limit* in combination with the *memory buffer*. The selection of the parameters is based on their potential influence on the applications' performance.

Once the search space is pruned it can be searched with the heuristic search algorithms, which provide a good trade-off between necessary time and the final results.

Eager Limit Strategy

The importance of the eager limit parameter and its availability in most MPI implementations makes it a good candidate for tuning. This parameter allows users to define a threshold, defined as the maximum size in bytes, at which the eager protocol is used as the message passing protocol. The parameter often has an upper bound defined by the MPI implementation. For example, the IBM MPI implementation from the example above bounds the parameter from few bytes up to 64 KiB, making 65536 possible values for the parameter alone.

The eager protocol can significantly affect the point-to-point communications, e.g., `MPI_Send`. This type of communication is affected by the message passing protocol. The selected protocol has effects on how the messages are buffered, the necessity of handshaking protocol with the receiver before transferring data, etc. The eager protocol assumes that the receiver has allocated a buffer large enough to store the message if it is sent. Therefore, the sender just sends the message as it does not have to ask for the acknowledgment from the receiver. This reduces the synchronization delays but does not scale as it requires that the receiver has dedicated buffers for each sender. When this assumption can not be made, or simply the eager limit is exceeded the MPI implementation decides upon the other transfer protocol like rendezvous. Such protocols require some type of handshaking protocol but use buffers only when needed making them more efficiently utilized.

To decide upon the performance impact of the eager protocol we defined a performance property called `EagerLimitDependent`. The `EagerLimitDependent` property is requested through the configurable pre-analysis strategy. This performance property collects 8 metrics relevant to decide about the upper and lower bounds of the eager limit.

The following metrics are used:

`MPI_MSG_P2P_THR`: corresponds to the total number of bytes transferred near the eager limit threshold (currently between 1 KiB and 64 KiB)

MPI_MSG_P2P_TOT: corresponds to the total number of bytes transferred using the MPI point to point communication operations.

MPI_MSG_FREQ_<2K-64K>: corresponds to the frequencies of occurring in certain ranges of message sizes. The ranges of message sizes start with 2K which corresponds for sizes between 1 byte and 2 KiB while the next one is between 2 KiB + 1 to 4 KiB.

The property request is propagated through the hierarchy to the analysis agent which configures the metric requests in the MRI monitor. The corresponding events for metrics are computed by the MPI wrapper library for the running application, from MPI DataType sizes used and the occurrence of certain ranges of message sizes. Once the metrics are computed and the agent requests, the summary metric data is propagated to the agent and used to generate the requested property. The reported property is used to detect the sensitivity of the point-to-point communications to the eager limit configuration parameter.

Equation 4.2 specifies the severity of the property. The severity of the EagerLimitDependent property is computed as a percentage of the total MPI point-to-point traffic near valid eager limit setting. Where $T_{threshold}$ stands for the MPI traffic near eager limit setting, while T_{total} stands for the total MPI point-to-point traffic.

$$Severity = \frac{T_{threshold}}{T_{total}} \cdot 100 \tag{4.2}$$

The resulting property is used to reduce the range of the eager limit parameter, but also the memory buffer size and thus prune the search space.

4.3.3 Tuning

The MPI Parameters plugin assumes that tuned applications are SPMD and therefore all application processes are optimized in the same way.

The plugin is built around the user-defined configuration file. In the configuration file, the user specifies the search algorithm and lists the tuning parameters and their ranges of potential values. During the tuning process, the MPI Parameters plugin often restarts the application with a new combination of MPI parameters to evaluate their effect on its performance. The MPI tuning parameters specified by the user may be set to their values using environment variables or mpirun options (flags) depending on the MPI runtime. The application has to be restarted before each experiment as all MPI runtime parameters need to be known during the initialization. To keep track of the improvement of the objective, the execution time of the phase region is measured.

Due to its huge search space, the plugin should be combined with one of the PTF's heuristic based generic search strategies. One of these strategies is *Generalized Differential*

Evolution 3 (GDE3) which implements the genetic search algorithm. The strategy starts by randomly generating and evaluating a predefined set of scenarios, ten by default. In the subsequent iterative process called generation, the n best-performing scenarios are selected for the next generation and their configuration goes through the process of mutations. In the process of mutation, each parameter's values are mutated with a fixed probability, which is defined by the user. The user can define the number of repetitive generations, but, generally, a close-to-optimal configuration is found with less than 30 generations.

For the example introduced in Section 4.3.2, this means that only 300 scenarios have to be evaluated instead of 1024000. That makes, otherwise, an impossible task to be finished within several hours or maybe few days depending on the application's execution time. Nevertheless, for many applications that might be reasonable tuning time compared to the tuning benefit.

To improve tuning time, the plugin eliminates scenarios with associated parameters that does not make any effect. For example, if IBM MPI pe_affinity flag is set to no the associated task_affinity parameter has no effect. Therefore, for each group of scenarios which have pe_affinity set to no only one scenario is kept while others are removed. To improve tuning time even further, the user can enable the automatic eager strategy developed for the plugin that reduces the range of parameters through a specific model. The tuning model shrinks the parameter range for the eager limit and consequently the memory buffer size. The eager limit value is determined with the help of PTF's analysis capability and the MRI monitor which collects detailed information about message lengths from the MPI library wrapper.

If the use of automatic eager strategy is not enabled, the parameter is used with the full specified range.

Model for Tuning the Eager Limit Parameter

The tuning parameter explained in Section 4.3.2 allows users to define the maximum size of messages sent using the eager protocol. The parameter is available in most MPI implementations and results in which transfer protocol is employed by the MPI implementation. It has a significant effect on the performance of point-to-point communications. However, the eager limit and the buffer used by the eager protocol results in an explosion of the search space size.

The influence of the parameter on the performance of an individual-oriented simulation of fish schools application (FSSIM) can be seen in Figure 4.6. The application simulates the population of 256K individuals executed on 64 cores in the SuperMUC. IBM MPI runtime available on the system sets the eager limit parameter by default to 32 KiB, while the optimal setting for this simulation of 1 KiB results in approximately 30% better performance. The Figure also shows the effect of different buffer sizes on the performance. The buffer memory is allocated by the communication library for storing the received message.

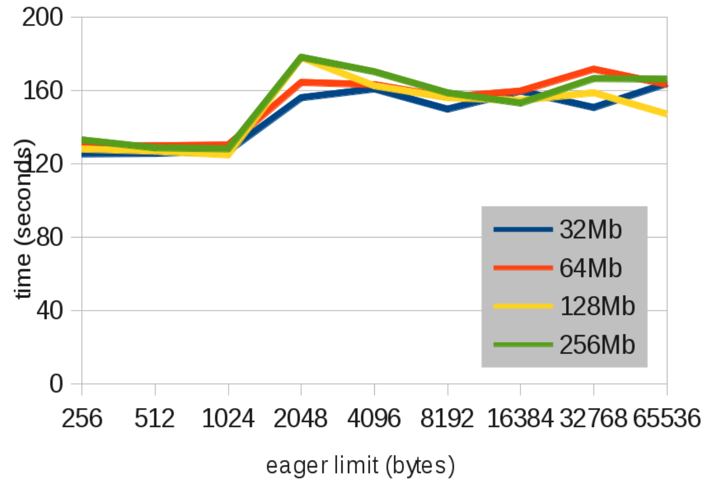


Figure 4.6: Execution time of FSSIM for different values of the eager limit and memory buffers parameters.

Allocation of big, mostly unused, buffers can have negative effects on the performance. As can be seen, the variations in the execution time for different memory buffer sizes are very small.

During the pre-analysis step the plugin calls a configurable analysis strategy to obtain the `EagerLimitDependent` property. The property collects the frequencies of occurring in certain ranges of message sizes and the percentage of the total MPI point-to-point traffic near valid eager limit setting. If the percentage of the MPI point-to-point traffic is more than 30% of the total number of messages sent by the application, the parameter is considered to be worth tuning.

$$mem_buff_low = 2n \cdot max(eager_limit_low, 64) \tag{4.3}$$

Once the eager limit is determined to be worth tuning, the plugin decides about the range of the parameter. The range is determined from the frequencies of occurring in certain ranges of message sizes by selecting the smallest group of consecutive ranges that include the major percentage of the total number of messages. For example, Figure 4.7 shows the frequencies of occurrence of message sizes where a major percentage of messages are in two ranges 4 KiB to 8 KiB and 8 KiB to 16 KiB. Based on this, the plugin selects the parameter range from 4 KiB to 16 KiB with a step of 1 KiB. Now, from the eager limit ranges and the step, the plugin employs Equation 4.3 to calculate the range and the step of the memory buffer size.

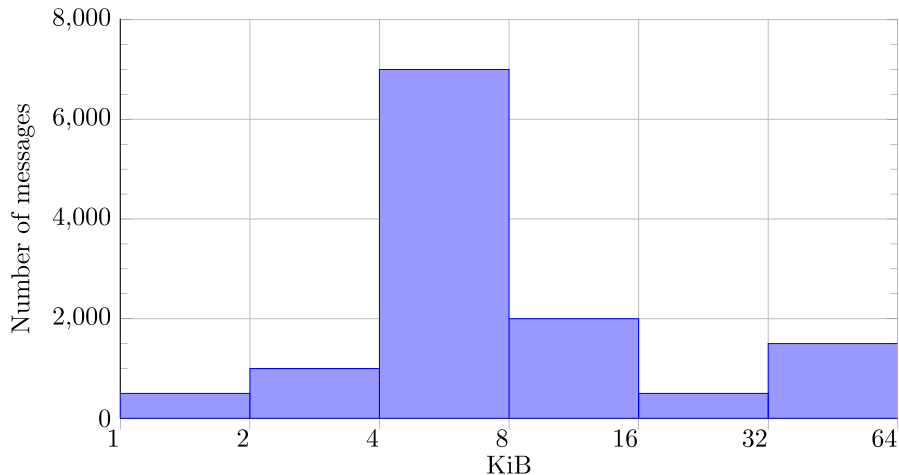


Figure 4.7: Hypothetical frequencies of occurring in certain ranges of message sizes from the `EagerLimitDependency` property.

Configuration File

The MPI parameters plugin is built around the configuration file which allows the user to configure the parameters used during the tuning process. To parse the configuration file, the plugin defines its own lexical analyzer and the parser. The tuning parameters and the values they take are represented with various data types like string or numbers. Their values can be represented as lists of strings or numbers or as ranges of values.

Let's consider an Example depicted in Figure 4.8 which represents the IBM MPI configuration file. The user defines the parameters and their possible values, the search algorithm and enables the automatic eager strategy. The header and the footer of the file specify that the configuration file is written for IBM MPI. In lines 2-6 the user has specified the tuning parameters. The tuning parameters `eager_limit`, `buffer_mem`, and `polling_interval` are ranges of values. The `eager_limit` parameter is set between 4 KiB and 64 KiB in steps of 2 KiB, the `buffer_mem` parameter is set between 8 MiB and 128 MiB in steps of 2 MiB while the `polling_interval` parameter is set between 100 ms and 1000 ms in steps of 10 ms. The tuning parameters `pe_affinity` and `task_affinity` contains bool values and strings. The `pe_affinity` can be enabled and disabled, with yes and no respectively. The `task_affinity` can be set to core or cpu.

In line 7 the user specified the GDE3 search algorithm to walk through the search space. The user also selected, in the last line, that the automatic eager limit strategy should be used to determine the optimal parameter ranges for `eager_limit` and `buffer_mem` parameters.

```
1  MPI_PIPO BEGIN MPI_IBM
2  eager_limit = 4096:2048:65536
3  buffer_mem = 8388608:2097152:134217728
4  pe_affinity = yes, no
5  task_affinity = core, cpu
6  polling_interval = 100000:10000:1000000
7  SEARCH = GDE3
8  AUTOMATIC_EAGER = eager_limit, buffer_mem
9  MPI_PIPO END
```

Figure 4.8: Example MPI Parameters configuration file.

4.3.4 Complete Tuning Flow

The complete tuning flow of the MPI parameters plugin is depicted in Figure 4.9.

Plugin initialization: The plugin loads the tuning parameters from the configuration file and creates the tuning space. The configuration file contains specifications of the parameters specific to the MPI implementation, the search algorithm and could request the automatic eager limit strategy. The automatic eager limit strategy is also enabled if the user has requested it, otherwise, the full parameter range is provided to the search algorithm. The specified search algorithm is loaded, initialized and configured with the predefined configuration.

Start tuning step: The tuning process of the MPI Parameters plugin requires a single tuning step. However, it contains two implicit loops, the outer that controls the generation of scenarios and the inner that controls their evaluation. The generation of scenarios depends on the search algorithm. While exhaustive search generates all scenarios in a single step, the GDE3 generates a predefined set of scenarios and in next steps, it does selection by mutating the best resulting scenarios from the step. The inner loop just provides scenarios that are to be evaluated.

Pre-analysis: In the configuration file, the user can request the automatic eager limit strategy. When it is enabled, the plugin uses the configurable analysis to evaluate the `EagerLimitDependent` property on the phase region to automatically determine the eager limit.

Create scenarios: When the automatic eager limit strategy is enabled, the eager limit analysis reduces ranges of the eager limit and the memory buffer parameters. The criteria to reduce the parameters ranges is that 30 % of the total message sizes are near the eager protocol threshold. If the criterium is satisfied the plugin selects the group of consecutive ranges that include the major percentage of the total messages. The determined ranges of the eager limit is then used to determine the appropriate

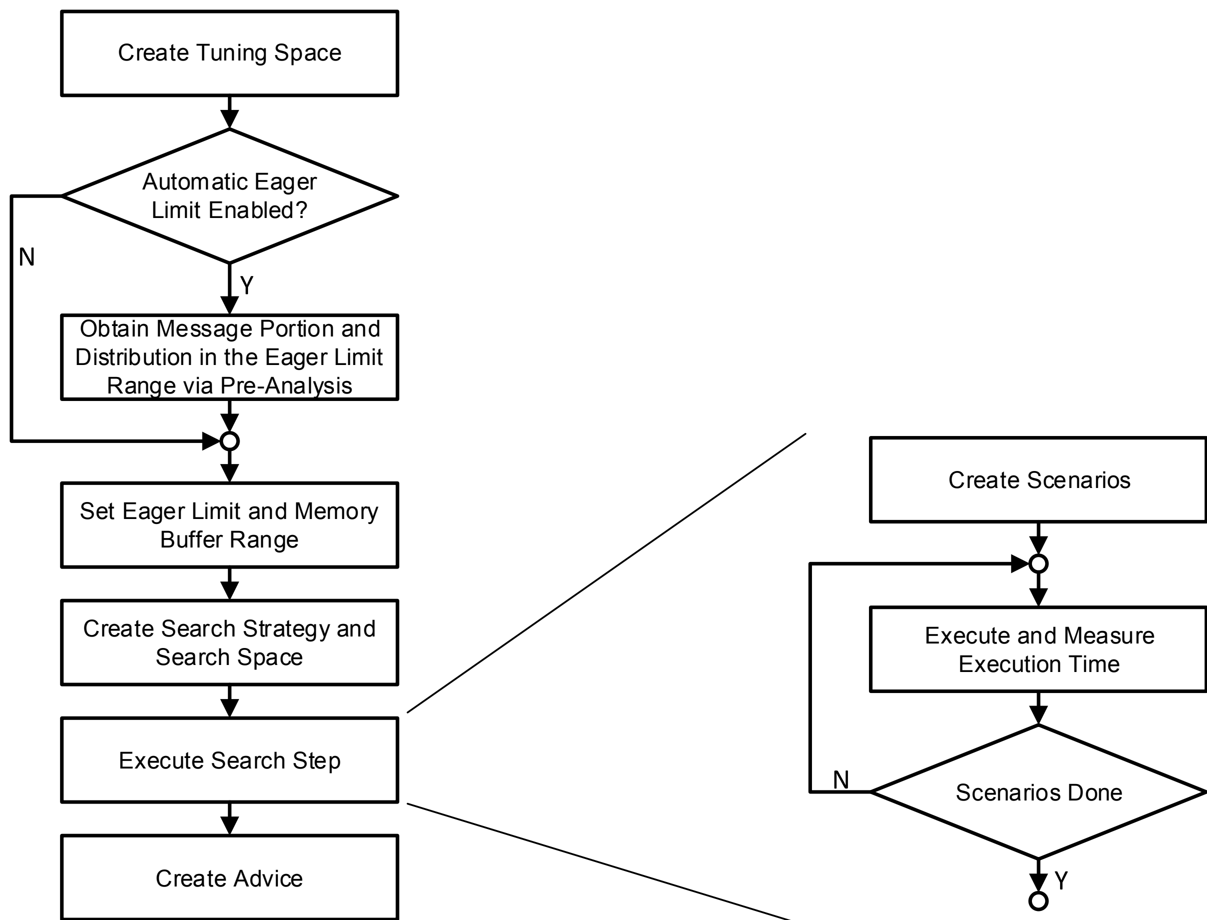


Figure 4.9: Tuning approach of the MPI Parameters plugin.

range and step of the memory buffer size parameter. Once the parameter list is refined, the selected search algorithm creates scenario that are evaluated.

Prepare and execute scenarios: Now, the scenario is prepared depending on the MPI runtime and the scenario tuning specification. For some MPI runtimes that mean exporting environment variables, e.g., Intel, while for others that mean providing flags during the application start. To evaluate the new configuration of the MPI parameters the application must be restarted as their change is not supported during the application execution. In this step, some scenarios are eliminated if they provide scenarios with the same configuration due to parameters with no effects. The application is now re-started and the new configuration is applied.

Give advice: The best scenario and the search path are shown to the user and stored in the advice file for later inspection. This configuration can later be applied by the user to the MPI runtime used for tuning. For example, the eager limit can be applied for IBM MPI by export MP_EAGER_LIMIT = 16384 or by passing the value as a flag to the mpiexec command, mpiexec –eager_limit 16384.

Plugin finalization: Unloads the search algorithm and cleans the plugin.

4.3.5 Summary

The presented plugin is used by the meta-plugin as it provides user configurable search space. The search space is large and therefore the search process may take considerable amount of time. Therefore, it is advised that the plugin's tuning potential is determined in advance by machine learning that tuning process is not ran if little or no improvement is expected.

4.4 User Defined Parameters

4.4.1 Introduction

The user sometimes would like to tune an aspect which is not supported by tuning plugins in PTF. Therefore, in the context of this thesis, we have developed the User Defined Parameters (UDP) plugin. The plugin tunes the application from the parameters defined by the user in the configuration file. From the experience collected by tuning with the User Defined Parameters plugin, the user might decide to develop his own PTF plugin.

4.4.2 Analysis

The User Defined Parameters plugin relies on the analysis capabilities of Periscope for both, a pre-analysis and a scenario analysis of each experiment. The pre-analysis is employed to identify files whose execution time significant enough to be tuned. Regions that count for at least 5% of the phase execution time are considered to be significant. The *Importance* strategy employed for the identification of significant regions is described in Section 4.1.2. The scenario analysis is employed to perceive the effect of the experiment on the significant regions. The effect of the experiment is captured by the `ExecTime` property from the regions identified as significant during the pre-analysis. Based on the information collected during the analysis, the plugin can provide the advice with recommended optimizations for individual regions instead of the entire application.

4.4.3 Tuning

The User defined parameters plugin is based on tuning parameters specified by the user through the configuration file. The configuration file can specify also the search algorithm. The search strategy controls the evaluation of scenarios. However, the tuning plugin tries to reduce tuning time by minimizing the necessary restarts by reordering scenarios.

Tuning Parameters

Five types of tuning parameters are supported:

Runtime variable: the runtime tuning parameter responsible to modify the variable during the application execution.

Runtime function: the runtime tuning parameter responsible to call a function with parameter value during the application execution.

Command line: the pre-execution tuning parameter responsible to define the command line. This can be used to modify the MPI parameters executed during the runtime.

Environment variable: the pre-execution tuning parameter responsible to set the environment variable. This can be used to modify the number of threads before the application is executed.

Process number: the pre-execution tuning parameter responsible for the number of MPI processes.

Based on the tuning parameter type, the plugin can automatically decide when the tuned application has to be restarted and change the number of MPI processes necessary for execution.

```
Scenario Id RTParam PEParam => Scenario Id RTParam PEParam
          0       0       a =>           0       0       a
          1       0       b =>           2       1       a
          2       1       a =>           1       0       b
          3       1       b =>           3       1       b
```

Figure 4.10: Example of scenario reordering scheme.

Each tuning parameter whose value is updated with the pre-execution tuning action requires the application to be restarted. During the tuning process, the plugin goes through multiple steps of scenario preparations and evaluations. Therefore, the prepared scenarios are reordered in such a fashion that two consecutive scenarios to be evaluated only modify pre-execution tuning parameters if there are no scenarios that only modify runtime tuning parameters. This minimizes the necessary restarts during the tuning process and consequently results in quicker tuning. Once the optimal execution of scenarios is determined the scenarios are selected to be evaluated in an experiment.

Figure 4.10 provides an example of the scenario reordering scheme. The search algorithm has generated four scenarios. In the case on the left, if scenarios are executed in order how they are issued four application restarts are necessary while in the case on the right only two restarts are necessary. The benefit might be much higher in cases with many tuning parameters.

Configuration File

The configuration file is the central concept of the plugin.

Let's consider an Example depicted in Figure 4.11 which represents the user defined configuration file. Line 1 defines the command line argument used to propagate tuning parameter of the command line type. Line 2 defines the shell command used to set the environment variable. Lines 3-7 are used to define the tuning parameters. The configuration file defines two types to specify values which the tuning parameter can be assigned to. The supported types are lists of numbers and strings and ranges of numbers.

4.4.4 Complete Tuning Flow

Plugin initialization: The plugin parses the configuration file and creates the tuning parameters from the user specification. Then, based on the user specification the search algorithm is loaded and initialized. From the tuning parameter's specifications, the variant space is created and provided to the search algorithm.

Start tuning step: The plugin requires a single tuning step.

```
1 command_prefix="-genv";
2 environment_prefix="export";
3 param name="rtvariable", type=RUNTIME_VARIABLE, range=[1,10,1];
4 param name="rtfunction", type=RUNTIME_FUNCTION, range=[11,20,1];
5 param name="variable", type=COMMAND_LINE, list=["1", "2", "3"];
6 param name="PTF_TEST", type=ENVIRONMENT_VARIABLE, range=[1,2,3];
7 param name="processes", type=PROCESSES_NUMBER, list=[1,2,4,8,16,32];
```

Figure 4.11: Example User plugin configuration file.

Pre-analysis: The plugin configures the pre-analysis led by the *Importance Analysis Strategy* described in Section 4.1.2. The analysis strategy determines for each process the regions responsible for at least 5% of the overall execution time. These regions are used by the scenario analysis to determine the effect of the configuration of significant regions.

Create scenarios: The plugin processes the results of the pre-analysis and creates the list of significant regions. The significant regions are used to generate the filter file which reduces the instrumentation overhead.

Prepare and execute scenarios: The plugin now goes over the tuning specifications and reorders the scenarios to minimize tuning time while evaluating all scenarios. One of the scenarios gets selected and according to the user request the plugin prepares the new number of MPI processes, the new command line, and the new environment variable. In the case when the scenario changes the value of any pre-execution tuning parameters from the previous experiment the application is restarted. Before the restart of the application, the prepared MPI processes, the command line, and the environment variables are executed. The evaluation of scenarios is repeated until the search algorithm finishes all search steps.

Give advice: Once the tuning has finished the plugin generates the tuning advice. The tuning advice is generated using the built-in support in PTF. The advice consists of two parts:

1. The global best configuration. The selection of the configuration for the phase region is done with the help of the tuning objective function.
2. The best configuration per-file. The selection of these configurations is done through the scenario analysis of significant regions per-file. For the significant regions, the analysis strategy evaluates the `ExecTime` property. For each file, the best combination of tuning parameters and its values is reported in the file.
3. The search path.

Plugin finalization: Unloads the search algorithm and cleans the plugin.

Chapter 5

Meta-plugins

5.1 Introduction

Getting the best performance of an HPC application requires selecting the right algorithm for the problem the application solves and optimize its work on the system. When the application is written, the developer usually optimizes the performance of individual single-core aspects like the memory access pattern or finding the optimal combination of compiler optimizations. After the single-core optimizations, the developer concentrates on parallel optimizations to improve the application scalability by reducing load-imbalances, find the optimal mapping of tasks/threads onto cores, and searches for the optimal combination of parameters that improve interprocess communication. As it is explained, to reach good performance the user often tunes many different aspects exposed in PTF's tuning plugins. Therefore, PTF allows combining multiple tuning plugins into a single plugin with meta-plugins. A meta-plugin acts as an intermediate between the frontend and the component plugins. The meta-plugin loads the component plugins, decides upon which plugins to run, in which order and drives their execution.

In the context of this thesis, three meta-plugins were developed: Fixed Sequence, Adaptive Sequence with Analytical Model and Adaptive Sequence with Predictive Model.

The simplest form of a meta-plugin is the Fixed Sequence which loads a list of tuning plugins and runs them one after another. Once the tuning process of individual component plugins has finished, the meta-plugin creates a tuning advice for all component plugins and finishes its execution.

The more advanced approach is built into Adaptive Sequence meta-plugins, shown in Figure 5.1. The Adaptive Sequence meta-plugins identify suitable component plugins and once they are known tune only with them. The Adaptive Sequence is implemented for two plugins, Adaptive Sequence with Analytical Model and Adaptive Sequence with Predictive Model. The Adaptive Sequence with Analytical Model (ASAM) decides upon which component plugins to run based on a performance property, analytical model, criteria

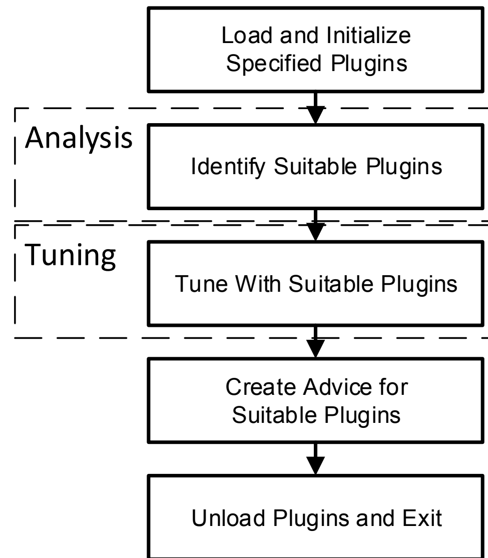


Figure 5.1: The tuning flow of the Adaptive Sequence meta-plugins.

defined by an expert. To achieve high accuracy it requires in-depth knowledge of the microarchitecture and/or the tuning aspect. This approach is component plugin specific and requires the plugin developer to derive a model by hand. As individual component plugins tune various programs, the model has to be generic and portable across programs. Additionally, each new feature added to the tuning plugin or the microarchitecture might require changes to the model. Moreover, with more complex architectures and tuning plugins, it becomes increasingly difficult to define the accurate criteria.

To evade the aforementioned problems we have developed ASAM which takes a completely different approach. The Adaptive Sequence with Predictive Model plugin decides upon which component plugins to run independently of the architecture or the tuning aspects the plugin tunes. Although it requires no intervention from the developer it still requires knowledge about the previous tuning with the component plugin. Its approach is characterized by a small black-box signature evaluated for every plugin extracted from the tested application. The signature is evaluated by executing a small number of signature scenarios. Signature scenarios are identified as the most representative scenarios for the search space based on historical performance data. These scenarios characterize the search space. Once the signature is known, the meta-plugin predicts the expected improvement of the tuning objective for the component plugin and decides about the plugin's suitability.

```
1 plugins = "compilerflags", "mpiparameters";
```

Figure 5.2: Example Fixed Sequence configuration file.

5.2 Fixed Sequence

5.2.1 Introduction

As already explained, the user often tunes many different aspects exposed in PTF's tuning plugins to reach good performance. Therefore, PTF allows combining multiple tuning plugins into a single plugin with meta-plugins. A meta-plugin acts as an intermediate between the frontend and the component plugins. Its simplest form, the Fixed Sequence plugin, loads the component plugins specified by the user in the configuration and executes them one after another. Due to its simplicity, the Fixed Sequence plugin does not need the analysis capabilities of Periscope.

5.2.2 Tuning

The list of component plugins can be defined through the configuration file or as a command line argument when the tuning process is started. If the user does not specify the list of plugins the meta-plugin loads the default list of component plugins.

Configuration file

The Fixed Sequence can be configured through the configuration file or through the command line optins. The configuration file has very simple syntax. The user defines only a list of component plugins separated with commas.

In the Example depicted in Figure 5.2, the user specified two component plugins, the Compiler Flags Selection plugin and the MPI Parameters plugin.

5.2.3 Complete Tuning Flow

Figure 5.3 gives an overview of the complete tuning flow of the Fixed Sequence plugin.

Plugin initialization: The meta-plugin first reads the configuration file or parses the command line. This is followed with the initialization of requested component plugins one after another. The first component plugin is selected.

Start tuning step: Before the call to start the tuning step method of the selected component plugin, the meta-plugin copies the analysis results pool (ARP) from the

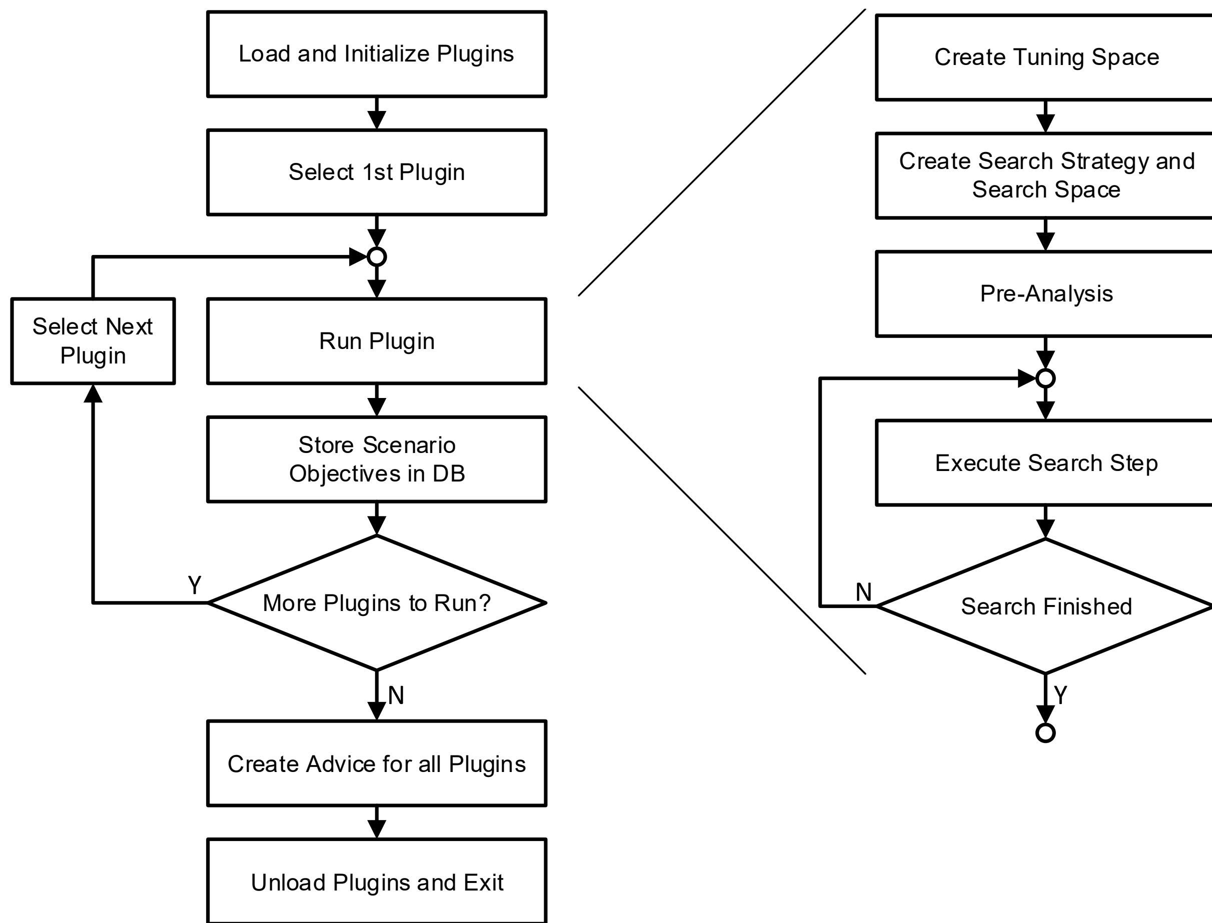


Figure 5.3: The tuning flow of the Fixed Sequence meta-plugin.

meta-plugin into the selected component plugin's ARP to synchronize the progress of the component plugin.

Pre-analysis: Propagates the call from the selected component plugin and copies the CSP from the selected component plugin into the meta-plugin.

Create scenarios: Propagates the call from the selected component plugin and copies the created scenarios from plugin's CSP into the meta-plugin's CSP.

Prepare and execute scenarios: Propagates calls to prepare and execute scenarios from the statemachine into the selected component plugin. Before executing scenarios the selected component plugin is contacted signal if the number of processes has to be changed, the new command line or new environment variable has to be issued. The selected component plugin can also request the application restart. Meta-plugin's scenario pools are cleared of scenarios and then populated with scenario pools from the component plugin. When the search has finished the scenario results pool is copied from the meta-plugin's scenario result pool into the selected component plugin. The finish tuning step call of the selected component plugin is called from meta-plugin's call. When the selected component plugin has finished tuning process the next plugin specified by the user is selected from the list and the flag from the previous component plugin is propagated to the frontend.

Give advice: Now the meta-plugin iterates through the list of tuning advices of component plugins and stores them in a single tuning advice file.

Plugin finalization: The meta-plugin finalizes requested component plugins one after another. The first component plugin is selected.

5.3 Adaptive Sequence with Analytical Model

5.3.1 Introduction

The tuning process of individual component plugins can potentially be very expensive. Therefore it is of immense importance for the meta-plugins to focus only on component plugins which are worth tuning. The decision is usually based on the expected improvement of the tuning objective and has to be done automatically without the user intervention. The user is only required to define the minimum improvement of the tuning objective for which he is willing to tune the tuning aspect. For that purpose, we have developed two automatic decision approaches, one based on the analytical model and one based on the comparative method.

This section introduces the approach driven by the analytical model. The analytical model derived by hand requires in-depth knowledge of the architecture and/or the tuning

aspect. It is expressed as a Periscope performance property for individual component plugins executed by the Adaptive Sequence with Analytical Model (ASAM) plugin. The model has to be generic enough to allow tuning of various programs. This approach is sensible to changes in the component plugin or in the execution environment. For instance, if the model uses information that is not available on a new system, e.g., a hardware performance counter, the model has to be redefined. Moreover, with more complex architectures and tuning plugins, it becomes increasingly difficult to define the accurate model. Compared to the comparative approach introduced in the coming section the meta-plugin only requires a single step pre-analysis while the comparative approach requires analysis which consists of several scenarios evaluation.

The flow of the ASAM plugin is split into the analysis part and the tuning part.

5.3.2 Analysis

The ASAM plugin relies on the analysis capabilities of Periscope for a pre-analysis. The pre-analysis is used to identify component plugins that are worth tuning with.

Tuning Decision Analysis

To decide upon the necessity to tune with individual component plugins we defined a performance property called `HPCConditional`. The `HPCConditional` property is requested through the configurable pre-analysis strategy. The severity of the reported property represents the average execution time of the phase region. Besides the severity, the property also provides the additional information in the form of the value of three important derived metrics (subproperties). These three metrics are used by the analytical model for the automatic tuning decision of individual component plugins. As the plugin can tune various programs, the criteria have to be generic and portable across programs.

For that purpose, the meta-plugin uses the Instructions per Cycle (IPC) defined in Equation 5.1 to decide about tuning with the Compiler Flags Selection (CFS) plugin introduced in Section 4.1. In the case that IPC of phase p is lower than the user defined threshold $IPC_{threshold}$ the plugin is used for tuning the program, otherwise not.

$$IPC_p = \frac{N_{INSTRUCTION}(p)}{N_{CYCLES}}, IPC_p < IPC_{threshold} \tag{5.1}$$

The Dynamic Voltage and Frequency Scaling (DVFS) plugin uses Instructions per Joule, expressed in Equation 5.2. The plugin was developed in the context of the Autotune project by Leibniz-Rechenzentrum and tunes the energy consumption of an application by finding the optimal frequency for individual program regions. In the case that IPJ of phase p is lower than the user defined threshold $IPJ_{threshold}$ the plugin is used for tuning the program, otherwise not.

```
1 plugins = "compilerflags";
2 threshold = 1;
```

Figure 5.4: Example ASAM configuration file.

$$IPJ_p = \frac{N_{INSTRUCTION}(p)}{E(p)}, IPJ_s < IPJ_{threshold} \tag{5.2}$$

The MPI Parameters plugin, introduced in Section 4.1, uses the fraction of MPI time in complete execution time of phase p, expressed in Equation 5.3. In the case that the fraction of MPI time is higher than the user defined threshold $f_{threshold}$ the plugin is used for tuning the program.

$$f_p = \frac{T_{MPI}(p)}{T(p)}, f_s > f_{threshold} \tag{5.3}$$

5.3.3 Tuning

The list of component plugins and the threshold for component plugins can be defined through the configuration file or as a command line argument when the tuning process is started. If the user does not specify the list of plugins and/or thresholds, the meta-plugin loads the default settings. The default settings are defined for the CFS plugin, the MPI parameters plugin, and the DVFS plugin.

Configuration file

ASAM can be configured through the configuration file or through the command line options. The configuration file has a concise syntax. In the configuration file, the user defines only a list of component plugins that should be used in the tuning process separated with commas and the thresholds for component plugins when tuning should take place.

Let's consider an Example of the ASAM plugin configuration file depicted in Figure 5.4. For simplicity, the configuration file contains a configuration for only one plugin. Line 1 declares which tuning plugins to be used by the meta-plugin. Line 2 defines the criteria for the Compiler Flags Selection plugin to trigger tuning. The CFS plugin is used only if the average number of instructions executed per each cycle is lower than 5.2.

5.3.4 Complete Tuning Flow

Figure 5.5 gives an overview of the complete tuning flow of the ASAM plugin.

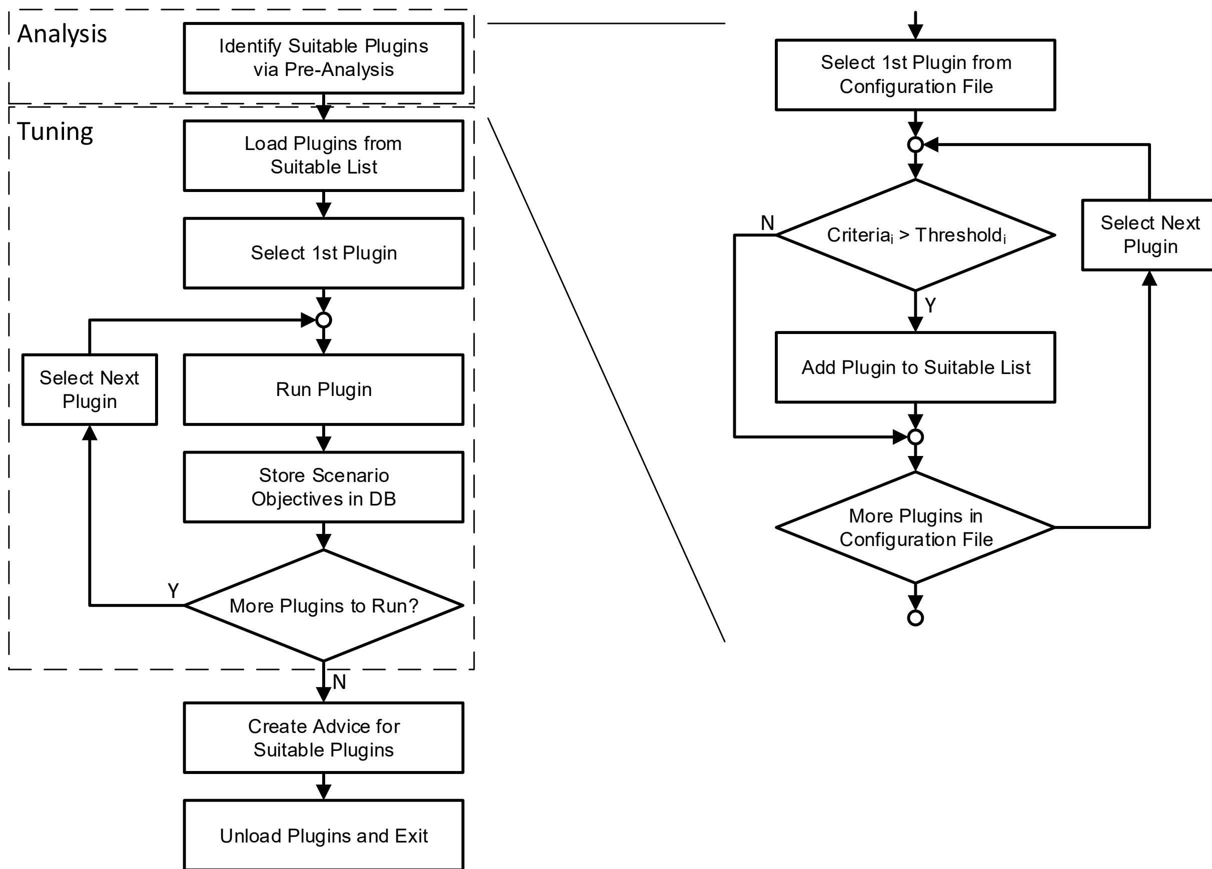


Figure 5.5: The tuning flow of the ASAM meta-plugin.

Plugin initialization: The meta-plugin initializes requested component plugins one after another.

The automatic plugin decision part:

Start tuning step: The tuning flow of the ASAM plugin can be split into two parts executed in different tuning steps. The first part controls the execution of the automatic plugin decision of individual component plugins. While the second part handles the execution of individual plugins.

Pre-analysis: In the analysis part, the meta-plugin configures the configurable analysis and requests the `HPCConditional` property. From the hardcoded metric-to-plugin rules the meta-plugin decides should the individual component plugin be tuned.

Create scenarios: The meta-plugin analyzes the reported `HPCConditional` property and compares its derived metrics with the condition for individual component plugins. The metric used for individual component plugins is hardcoded into the meta-plugins. The component plugins that do not meet the condition to be tuned with are finalized and removed from the list of component plugins. The first plugin that satisfies the tuning condition is selected.

Prepare and execute scenarios: Not relevant for the analysis part of the meta-plugin.

The component plugin execution part:

Start tuning step: Before the call to the start tuning step method of the selected component plugin, the meta-plugin copies the analysis results pool (ARP) from the meta-plugin into the selected component plugin's ARP.

Pre-analysis: Propagates the call from the selected component plugin and copies the CSP from the selected component plugin into the meta-plugin.

Create scenarios: Propagates the call from the selected component plugin and copies the created scenarios from plugin's CSP into the meta-plugin's CSP.

Prepare and execute scenarios: Propagates calls to prepare and execute scenarios from the statemachine into the selected component plugin. Before executing scenarios the selected component plugin is contacted to provide if it requests the number of processes has to be changed, the new command line or new environment variable has to be issued. The selected component plugin can also request the application restart. Meta-plugin's scenario pools are cleared of scenarios and then populated with scenario pools from the component plugin. When the search has finished the scenario results pool is copied from meta-plugin's scenario result pool into the selected component plugin. The finish tuning step call of the selected component

plugin is called from meta-plugin's call. When the selected component plugin has finished tuning process the next plugin specified by the user is selected from the list and the flag from the previous component plugin is propagated to the frontend.

Give advice: Now the meta-plugin iterates through the list of tuning advices of component plugins and stores them in a single tuning advice file.

Plugin finalization: The meta-plugin finalizes requested component plugins one after another. The first component plugin is selected.

5.4 Adaptive Sequence with Predictive Model

5.4.1 Overview

In the previous Section, we have introduced the Adaptive Sequence with Analytical Model plugin that employs an analytical model to focus the tuning process on component plugins worth tuning. This approach requires expert knowledge of the underlying execution environment and the component plugin. Moreover, with more complex architectures and tuning plugins, it becomes increasingly difficult to define the accurate model. Additionally, changes in the component plugin or in the execution environment might require rethinking the model.

Therefore, we have developed the comparative method in the Adaptive Sequence with Predictive Model (ASPM) plugin which evades the aforementioned problems. The main idea behind ASPM is to develop a meta-plugin that can decide upon which component plugins to use for tuning without any knowledge about the system and component plugins.

This work significantly differ from the previous research. In the previous work on comparative methods, presented in the Chapter 2.2, we have explained various problems with the gathering of features, the selection of right features for the signature, and the signature portability across architectures and tuning aspects. Gathering of dynamic features perturbs execution and is often not possible to be done within a single application run, e.g., PAPI counters. On the other hand, selecting right features for the signature is an open problem and often very subjective. The number of potential features is very large, while most of these features are tuning aspect specific. Researchers have tried to identify important features with various approaches such as multivariate statistical techniques, Principle Component Analysis (PCA), cluster analysis, factor analysis, and visualization techniques. In order to identify important features, the user has to collect many different features and then process the collected features with different, independent, tools. Once the features are collected, the user uses various independent tools to extract the signature and integrates them in its own tool. Changes in the system for which the tuning takes place in the most cases invalidates previously identified signature and requires signature reevaluation. Also, each of these approaches has to be repeated for each individual tuning

aspect. To circumvent all previously mentioned problems, PTF takes completely different approach. First, contrary to previous approaches, features composing a signature are sensitivities of a program to the search space. As features used for the signature does not depend on any aspect specific feature, it is portable accross a wide range of tuning aspects, i.e., plugins. Second, it is the first tool that fully integrates automatic signature detection into the tuning framework.

The meta-plugin starts its execution by loading the individual component plugins. For each of the component plugins it first checks the tuning potential of the plugin for the given application. To do this, it triggers a function provided by the plugin.

The component plugin then determines first the sensitivity of the application for its tuning approach. This sensitivity is computed by evaluating the tuning result of a predefined set of signature scenarios. This is done by executing the signature scenarios and measuring the objective function. The ratio between the objective function values for the signature scenario and a default scenario determines the improvement obtained. The vector of all improvements for the signature scenarios is called the signature, i.e, the sensitivity of the program for the tuning plugin. The computation of the signature is not costly as it requires evaluation of very few scenarios.

This sensitivity is then used to predict the tuning potential of the plugin for the given application. The plugin applies a classifier to the signature that determines whether the tuning potential of the application is above or below a given threshold.

Section 5.4.2 describes how the set of signature scenarios is computed for a plugin and the classifier is learned with supervised learning.

Once the tuning potential is returned by the component plugin, the meta-plugin triggers the component plugin only if the indicated tuning potential is high according to the threshold used in the classifier. After the execution of the component plugin, the scenarios on the search path and their objective values are inserted into the historical performance database for dynamically improving the knowledge learned. The meta-plugin continues executing the component plugins until all have been processed and end with unloading the plugins.

The next Section describes the overall design of the meta-plugin and the necessary teoretical background to understand the techniques used for identification of signature scenarios and predict the tuning outcome.

5.4.2 Design

To reach the goal of having a meta-plugin which is portable across systems and independent of the component plugins, we have developed the approach depicted in Figure 5.6.

Before the meta-plugin can predict tuning potential of a program for any component

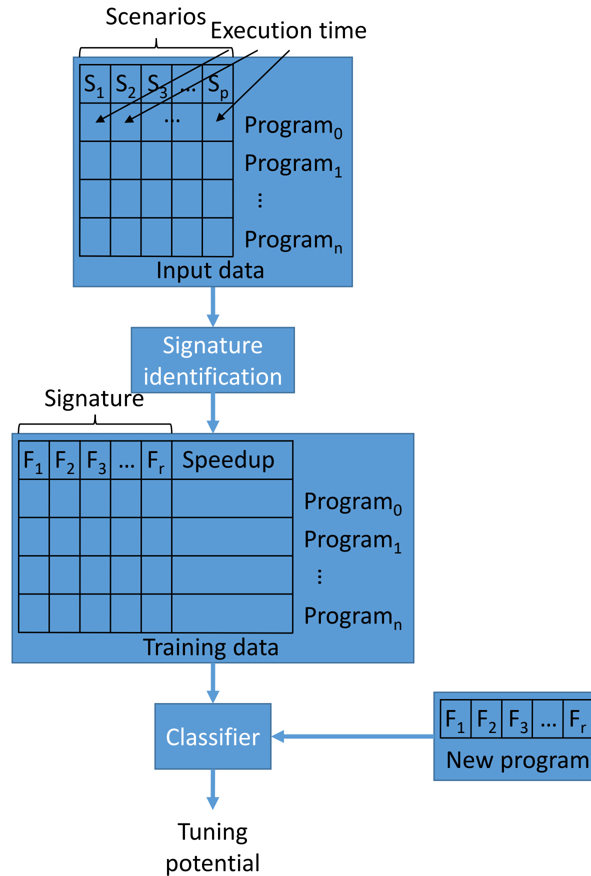


Figure 5.6: The simplified overview of the approach.

plugin it requires that data from several executions of the plugin is stored in the historical performance database. When started, the meta-plugin checks does it have more historical performance data than the user specified in the configuration file to be sufficient to train the predictive model and make predictions. Once enough data is available, the meta-plugin identifies scenarios 3.3.4 that characterize the search space called signature scenarios. The signature scenarios and their objective values, called the signatature, are used by the machine learning to train the predictor, a classifier or a regressor. The training process takes into account the improvement in the execution the user considers to be significant enough to justify the tuning time.

From the predictive model the meta-plugin makes predictions for a new program. To make predictions the meta-plugin first needs to evaluate the signature scenarios for a plugin on the new program a form the signature. Once the signature is available, the predictive model classifies the program to be worth tuning or not, or make predictions about the expected improvement of the tuning objective.

The design of the meta-plugin is split into three steps:

1. Identification of signature scenarios for a plugin
2. Collecting signature scenarios for a program
3. Predicting of the tuning outcome

In the following subsections we will give a short explanation on the steps and the required background to understand each of these steps.

Identification of signature scenarios

The approach is based on the idea of comparative methods explained in Section 2.2. The idea behind the approach is to make predictions of the tuning potential of new programs based on their similarity with other programs tuned by the component plugin. To quantify the similarity between programs we have identified the signature which consists of features that are sensitivity of the application to optimizations expressed in the search space, i.e., the search space characterization. Such features are architecture independent and therefore, portable across systems, and plugin independent.

The input data are performance properties that represent the tuning objective collected for scenarios executed by component plugins during the tuning process. Common tuning objectives for plugins described in the previous Chapter are execution time, energy or energy-delay product. The tuning objective properties are collected for many programs tuned by the plugin and stored in the historical performance database. They are retrieved by the meta-plugin and processed by the signature identification engine. The signature identification engine identifies the most representative scenarios for the search space, i.e., the signature. The signature objective values are used to train the regression and classification models. The regression predicts which tuning benefit the user should experience while the classification predicts that the tuning outcome is above some user predefined threshold. The signature scenarios are evaluated by the meta-plugin for a new program and their tuning objective properties are used by the predictor as a signature to make predictions of the tuning outcome or the tuning potential of the plugin for the program.

Besides the presented advantages, such approach also does not require modification of component plugins by the tuning aspect expert.

Background

This section introduces machine learning techniques used to implement the machine learning approach developed for this thesis. However, it does not provide a complete description of the techniques, more detailed information can be found in [47]. For signature identification, we have used two different techniques, information gain and clustering. The tuning potential is predicted from the identified signature with supervised learning techniques. The prediction of the tuning outcome is done in two modes, one to predict

the expected improvement of the tuning objective for the application with and another to predict whether the expected improvement is above the user defined threshold. The prediction of the expected improvement is done using the supervised machine learning technique called regression. While the prediction that the expected improvement is above the user defined threshold is done using the technique called classification.

Entropy and Information Gain

Finding a signature of the program requires identifying a subset of features that are relevant for the problem we are solving. For that purpose, we can use different approaches. One approach would be to identify features that carry the most of disorder or chaos in a system. The measure for disorder or chaos in physics is called entropy. To determine the expected reduction in entropy caused by partitioning a dataset on a given attribute the information gain is used. The entropy is calculated as follows:

$$entropy\left(Set\right) = I\left(Set\right) = -\sum_{i=1}^{k} P\left(value_i\right) \cdot log_2\left(P\left(value_i\right)\right) \tag{5.4}$$

Where $P(value_i)$ is the probability of getting the ith value when randomly selecting one from the set. For the set R={a,a,a,b,b,b,b,b} the entropy is calculated as in Equation 5.5.

$$entropy\left(R\right) = I\left(R\right) = -\left[\left(\frac{3}{8}\right) \cdot log_2\left(\frac{3}{8}\right) + \left(\frac{5}{8}\right) \cdot log_2\left(\frac{5}{8}\right)\right] \tag{5.5}$$

The algorithm used to calculate the information gain can be split into steps:

1. Calculate entropy of the target.
2. The dataset is split on the individual features.
3. The entropy for the individual feature is calculated.
4. The entropy for the individual feature is added proportionally, to get total entropy for the split.
5. The resulting entropy is subtracted from the entropy before the split.
6. The result is the Information Gain or decrease in entropy.

Constructing a signature scenarios is about finding features that return the highest information gain. The resulting signature scenarios consists of 5 to 20 features (scenarios) with the highest information gain.

Clustering

Another useful technique used to identify a signature is clustering. It is the machine learning task that describes hidden structure from unlabeled data, i.e., unsupervised learning technique. Clustering groups a set of features in such a way that objects of the same group are as similar as possible to each other than to those in other groups. Clustering algorithms can be split into three distinct types [47]:

1. Combinatorial algorithms which work directly on the observed data with no direct reference to an underlying probability model.
2. Mixture modeling which assumes that the data is an independent and identically distributed sample from some population described by a probability density function.
3. Mode seekers which take a nonparametric perspective, attempting to directly estimate distinct modes of the probability density function.

Similarity measure

The main characteristic of the distance based clustering techniques is that objects from the same groups have a minimum distance while the objects belonging to a different cluster have a maximum distance. To quantify the distance between objects that form the cluster the similarity measures are used, e.g., Euclidean distance between signatures. The selection of the appropriate similarity measure requires the knowledge about the domain. Scenarios used for the signature are usually objective values like execution time, the consumed energy or combination of both. Due to the nature of elements, to quantify the similarity between two scenarios we use the Euclidian distance.

$$d\left(x_i, x_i'\right) = \sum_{j=1}^{p} \left(x_{ij} - x_{i'j}\right)^2 = \|x_i - x_{i'}\|^2$$

K-medoids

K-medoids is one of the most popular iterative descent clustering methods. It is a combinatorial algorithm and therefore it assigns each feature to a cluster without regard to a probability model describing the data. This algorithm is intended to be applied to situations in which all variables are of the quantitative type, and squared Euclidian distance is chosen as the dissimilarity measure.

The results of applying k-medoids depend on the choice of the number of clusters to be searched and a starting configuration assignment.

Algorithm 2 K-medoids Clustering

1. For a given cluster assignment C find the scenario (objects) in the cluster minimizing total distance to other points in that cluster:

$$i_k^* = \underset{\{i:C(i)=k\}}{\operatorname{argmin}} \sum_{C(i')=k} D\left(x_i, x_{i'}\right)$$

Then $m_k = x_{i_k^*}$, k=1,2,...,K are the current estimates of the cluster centers.

2. Given a current set of cluster centers $m_1, m_2, ..., m_k$, minimize the total error by assigning each scenario to the closest (current) cluster center:

$$C\left(i\right) = \underset{i \leq k}{\operatorname{argmin}}$$

3. Iterate steps 1 and 2 until the assignments do not change.

Therefore, it is recommended to randomize starting configuration assignment and pick the one with the lowest cost.

To make the k-medoids parameterless we did 10-fold cross validation to determine the optimum value for parameter k.

The selection of the optimal parameter k is done with the elbow method [79]. The method is based on the observation about the correlation between the number of clusters and the total error. The total error, expressed in Equation 5.6, drops as the number of clusters increases. Figure 5.7 shows the total variance explained with clusters related to the number of clusters. The total explained variance is expressed as the total summed square error (SSE). At first, the reduction of SSE is significant, however as the number of clusters (k) grow and with certain k the reduction of SSE will be marginal and at that point, the k is considered to be optimal.

$$SSE = \sum_{i=1}^{K} \sum_{x \in C_i} (c_i - x)^2 \tag{5.6}$$

As this method is not guaranteed to unambiguously identify the optimal number of k we limit the range of k. The optimal value of k is in the range between 5 and 20 and is the number of features forming the signature.

Algorithm 3 K-medoids Clustering.

function DETERMINELOWESTERRORFORK(k)
  k ▷ Number of clusters
  $error_{lowest} \leftarrow \infty$
  $error \leftarrow \infty$
  for $(i \leftarrow 1, 10)$ **do**
    $randomCenters \leftarrow$ RANDOMCLUSTERCENTERS(k)
    $error \leftarrow$ KMEDOIDSCLUSTER($randomCenters$)
    if $(error < error_{lowest})$ **then**
      $error_{lowest} \leftarrow error$
      $randomCenters_{lowest} \leftarrow randomCenters$
  return $randomCenters_{lowest}$, $error_{lowest}$

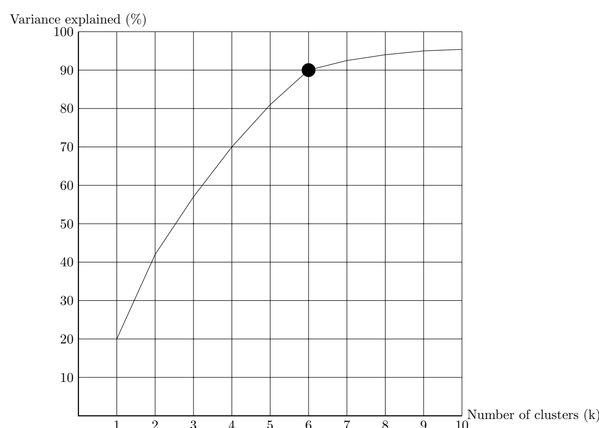


Figure 5.7: The explained variance with clusters with respect to the number of clusters.

Hierarchical clustering

The idea behind the hierarchical clustering is to produce clusters where each level of clusters is created by merging clusters from the previous level. The algorithms can be divided into two groups depending on the strategy they take to construct clusters, as agglomerative (bottom-up) and divisive (top-down). Agglomerative strategies start from n clusters represented by n objects and recursively merge selected pairs of clusters into a single cluster. The selected pair of clusters has the smallest distance, i.e., highest similarity. Divisive strategies start from a single cluster which contains all objects and recursively splits one of the clusters into two clusters until n clusters are created with 1 object. The cluster is split into two clusters which have the highest distance, i.e., lowest similarity. This method does not require the user to specify the number of clusters but rather the termination condition, i.e., the cutoff distance. However, finding the right and large enough cutoff distance for all input data is hard and might result in clusters split into two parts. Single linkage (SL) agglomerative clustering takes the intergroup dissimilarity

to be that of the closest (least dissimilar) pair.

$$d_{SL}(G,H) = \min_{i \in G i' \in H} d_{ii'}$$

Density-based methods

Density Based Spatial Clustering of Applications with Noise (DBSCAN) is a clustering algorithm that group objects based on their density. Objects with high density are part of clusters while objects with low density are considered to be noise. For that purpose, the user defines two global parameters ϵ and minPts. minPts stands for the minimum number of objects required to form a dense region while ϵ is the radius in which objects have to be. The algorithm starts with an arbitrary point and retrieves all points reachable from that point with respect to ϵ and minPts. The selection of parameters ϵ and minPts requires the knowledge of the domain and might not be trivial. The algorithm is developed with the requirement to be tolerable to noise and therefore it is robust to outliers. Unlike the k-medoids, this method does not require that the number of clusters is known a priori. The number of clusters depends on the data and its density.

Collecting signature

To enable meta-plugin decision based on machine learning the tuning history has to be stored and therefore the performance database was introduced. The database is designed around the scenario pool concept. The database stores measurements with the context of the execution from the given scenario pool. The context comprises of the program, the plugin that was used, the system, the region to which tuning configurations were applied, and which objective property is used. Due to its flexibility and simplicity, the SQLite relational database is selected.

The tuning history is stored into the database when the frontend is instructed to do so. Once the tuning history is enabled the plugin collects performance data during the tuning process in scenario pools and the frontend stores it into the database. The tuning history can later be retrieved by the meta-plugin and used to predict the tuning potential of a new program from its own signature and this data.

Predicting tuning outcome

In previous sections, we have presented techniques used to identify scenarios that form the signature. From the signature, we would like to predict that the plugin sensitivity is significant enough that it is worth tuning the aspect. The user specifies the speedup he expects to justify the cost of the tuning process. The expected speedup represents the radius of an N-dimensional tetartosphere, i.e., one-fourth of a sphere. Once the radius is known the classifier can predict which programs of the space are outside and therefore

satisfy the user constraint. For that purpose, we have employed two supervised learning techniques. The supervised learning learns a function $x \mapsto y$ from a set of training data: $(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n)$. The trained model can make a prediction y^* from a new value $\vec{x}^*$. Depending on the range of the function, i.e., the set of all possible output, we distinguish two predicting approaches, i.e., classification and regression. The range of the classification can be discrete and finite or a real number. For the discrete and finite range, we talk about the classification which assigns an input in one of the possible classes. On the other hand, if the range is a real number we talk about regression which predicts the output for a given input.The regression and classification models are trained from the signature objective values presented in this section. The purpose of the regression is to predict the tuning benefit the user should experience while the classification predicts that the tuning outcome will be above some user predefined threshold. The signature scenarios are evaluated by the meta-plugin for a new program and their tuning objective properties are used by the classifier/regressor as a signature to make predictions of the tuning outcome or the tuning potential of the plugin for the program. For that purpose, we have assessed two well known supervised learning classification and regression algorithms.

k-Nearest Neighbors algorithm

The k-Nearest-Neighbor is an algorithm used for classification and regression. The classifier selects k objects in the neighborhood of a new object and assigns it to the class of the majority. The regressor selects k objects in the neighborhood and assigns their average value to a new object. Sometimes, weighting scheme is introduced which favors neighbors based on some metric, e.g., the distance between points.

Figure 5.8 depicts an example of classification where k is set to 5. In the example, the majority of 5 objects in the surroundings of the classified objects belong to the blue class and therefore it is assigned to the blue class.

Support Vector Machines

Support Vector Machines (SVM) are a class of supervised learning models which can be used for classification and regression.

The SVM classifier can be written as in Expression 5.7:

$$\min_{\theta} C \sum_{i=1}^{m} \left[y^{(i)} cost_1(\theta^T x^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T x^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^{n} \theta_j^2. \tag{5.7}$$

where C>0, is the penalty parameter of the error term while $cost_0$ and $cost_1$ are the cost functions.

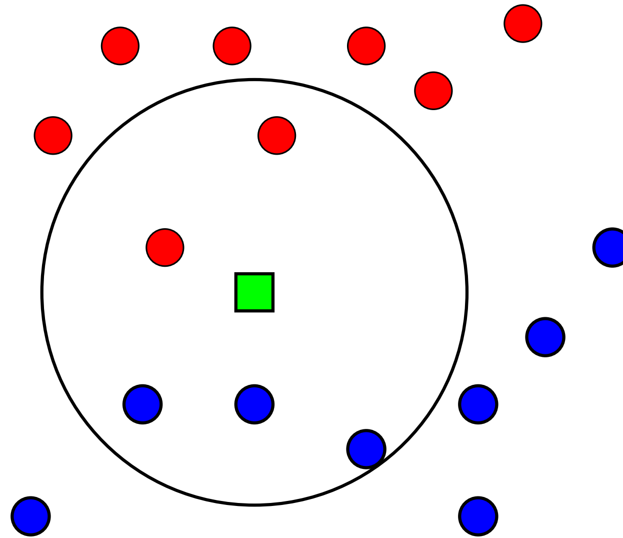


Figure 5.8: Classification of an object with k-Nearest Neighbors classifier with two classes where k is set to 5.

The cost functions are defined as `piecewise linear functions`:

$$cost_1(z) = \begin{cases} c \cdot (1 - z), & \text{if } z < 1 \\ 0, & \text{otherwise} \end{cases}$$

$$cost_0(z) = \begin{cases} 0, & \text{if } z \leq -1 \\ c \cdot (1 + z), & \text{otherwise} \end{cases}$$

The selection of parameter C affects the performance of the classifier:

large C: Classifier has a lower bias toward a class, but higher variance.

small C: Classifier has a higher bias toward a class, but lower variance.

Evaluating a machine learning techniques usually involves separating the dataset into two, the training and testing sets. If C is very large, the margin is going to separate the training dataset with a lower boundary but completely correctly which might result in overfitting and poor prediction on the testing data. On the other hand, when C is very small, the classifier might not be able to capture the trend of the data which results in underfitting. Therefore, to give the fair predictions for the training and the testing sets, the parameter C must be tuned to a value which is not too large so the classifier might allow some outliers but also not too small.

Figure 5.9 depicts the linear SVM classifier of two classes where $C = \infty$. As the penalty of outliers is extremely big, the two classes are separated by a maximum margin hyperplane.

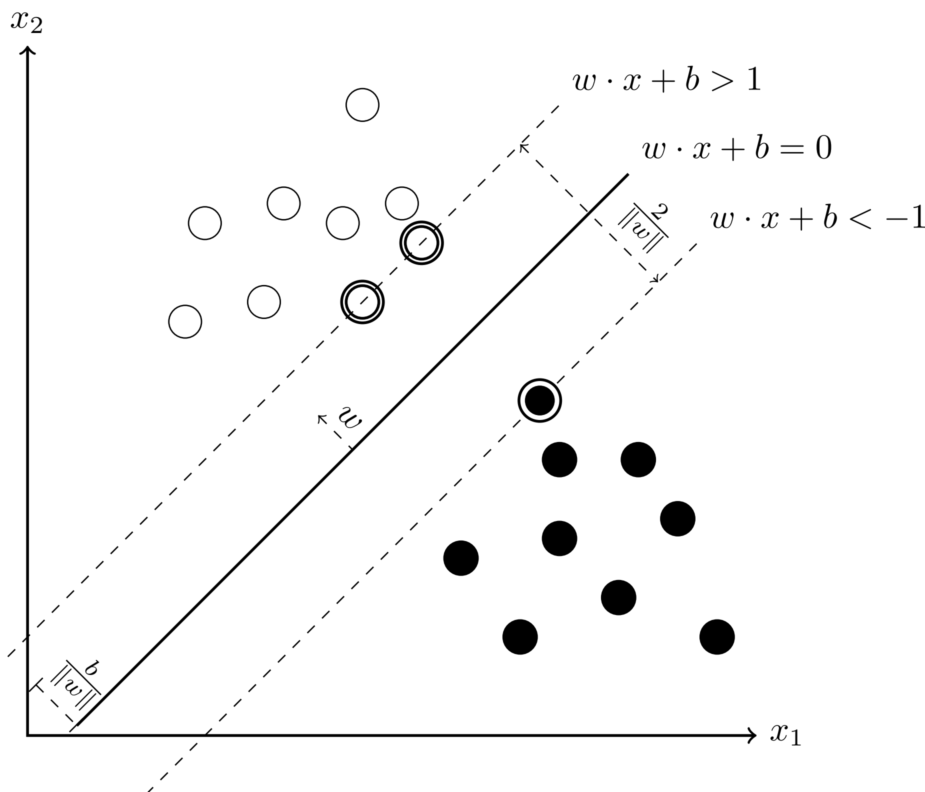


Figure 5.9: SVM classifier with two classes.

Therefore, such a classifier is called *maximum-margin classifier*. The objects on the margin are called the support vectors.

The SVM, in general, applies the linear separation. However, with a kernel trick, it is possible to do the nonlinear separation. Space is transformed using the nonlinear transformation ϕ in which the objects are linearly separable.

For that purpose various kernels are available. The most commonly used kernels are:

Linear (no kernel): $K(x_i, x_j) = x_i^T x_j$.

Polynomial: $K(x_i, x_j) = (\gamma x_i^T x_j + r)^d, \gamma > 0$.

Radial basis function (RBF): $K(x_i, x_j) = e^{-\gamma \|x_i - x_j\|^2}, \gamma > 0$.

Sigmoid: $K(x_i, x_j) = tanh(\gamma x_i^T x_j + r)$.

The presented kernels contain parameters, like γ, r and d which also influence the result of the classification or regression and therefore have to be tuned for the training dataset.

The selection of SVM is based on the fact that it generally performs better than other algorithms when the features are continuous and collinearity is present. SVM can also

deal with a case where the relationship between input and output feature is nonlinear, i.e., data are not linearly separable [55, 29].

5.4.3 Implementation

The goal of the analysis in the meta-plugin is to correctly predict the tuning potential of a new program based on its similarity with other programs already tuned by the component plugin. To reach the goal we employ techniques introduced in the previous Section.

Before the meta-plugin can quantify similarities between programs it has to identify scenarios that form signature scenarios. The signature scenarios are then executed and their objective values form what is called a signature. The signature consists of features that are sensitivity of the program to optimizations from the scenario expressed in the search space.

Extracting the signature scenarios necessary for ASPM requires a well defined experimental methodology. The methodology consists of running the same program with the same dataset on the same platform optimized with the same set of predefined scenarios. The platform has to provide the same execution environment like the same operating system, the same compiler, the same architecture, distributed system runtime, etc.

Search Space Characterization

As already discussed in Section 2.2.1 finding suitable signature is an open problem and often very subjective. In general, the goal is to identify the smallest set of signature scenarios that consist of scenarios that well describe the search space and not the large number of scenarios that allow to separate applications with different sensitivity best. A large number of signature scenarios increases the signature evaluation time. On the other hand, signatures with a long signature vector and small dataset usually have a high variance which is keen in overfitting due to bias-variance tradeoff. Collecting big dataset is very time consuming and not always possible. Therefore, finding a signature of a program requires identifying a small set of scenarios which well explains the search space is a must. While the Principal Component Analysis (PCA) presented in Chapter 2 is often used, its property to transform correlated variables into a set of uncorrelated variables makes its application to signature identification very hard. The principal components would be a linear combination of possibly all scenarios. Consequently, it would not reduce the number of scenarios we need to evaluate and therefore is useless. To overcome these limitations, we have employed the clustering and the information gain techniques.

Three clustering algorithms were evaluated: k-medoids, hierarchical clustering, density based methods (DBSCAN). Due to the nature of the data and potentially uneven distribution of the data due to the user configuration, the DBSCAN was nonoptimal and only k-medoids is used in the PTF implementation.

The identification process starts in the signature identification engine with a set of candidate scenarios for a given plugin. These scenarios are evaluated for a set of benchmark programs. We then determine from the large set of candidate scenarios a small set of signature scenarios that are not giving redundant information with the two presented techniques. Based on the used techniques we identify scenarios with the smallest set of scenarios with the highest information gain or the number of scenarios with the significant reduction in the total error of clusters.

The approaches used to identify the signature scenarios require the data to be normalized. For example, different programs do different computations and hence will have different execution times which might create a bias toward applications with longer execution times. Therefore, all the input dataset is normalized by the baseline objective value, e.g., execution time. The baseline is expressed as a baseline scenario of the plugin determined by the user.

From the search space signature, the plugin can characterize the search space and make predictions with much fewer experiments. Therefore, the search space signature can focus tuning just on fruitful plugins and consequently improve the time necessary for tuning. The size of the signature is limited between 5 and 20.

Extracting the signature scenarios requires to evaluate the same set of scenarios across a large number of program runs with the same dataset and environment, e.g., Operating System, Compiler, Architecture, and MPI. To determine the set of candidate scenarios we employ the uniform random sampling.

Uniform Random Sampling

The search space can be huge and cannot be evaluated in realistic time and therefore it has to be represented by the subset. To ensure unbiased selection of scenarios the uniform random sampling is selected. From the candidate set, the meta-plugin identifies then the signature scenarios.

Uniform random sampling is an algorithm which selects scenarios uniformly at random from a set of scenarios S if and only if it outputs a scenario $s \in S$ with probability $\frac{1}{|S|}$.

Search space pruning

Uniform random sampling selects scenarios uniformly at random from a set of scenarios. Many of these scenarios have the identical effect on the runtime due to disabled parameters. For example, MPI has a parameter use_bulk_xfer which enables bulk transfer (RDMA) and bulk_min_msg_size which controls the minimum size of the message to use the bulk protocol. It is clear that if bulk transfer protocol is not used that the bulk_min_msg_size has no effect. There are numerous cases of such interconnected parameters. To circumvent evaluation of meaningless scenarios we have developed an approach

that searches for such scenarios and removes them from the pools. In this thesis we call parameters, such as use_bulk_xfer, dependers since parameters such as bulk_min_msg_size, called dependee, depend on their value.

The search space pruning algorithm is shown in Algorithm 4.

The algorithm maintains:

- a set of depender parameters such as use_bulk_xfer (DP);
- a set of input/output scenarios (SS). At the beginning this list contains all scenarios while at the end this list is pruned and contain only scenarios that have unique effect on the execution.;
- a set of depender scenarios (DS).
- a set of independent scenarios (IS).

The algorithm initially loads a set of depender parameters (DP) and a set of input scenarios (SS). It also maintains two sets, one for scenarios which depend on the parameter, called depender scenariosDS, and the another which do not depend on the parameter, called independent scenarios IP. Now the algorithm selects the first DP and searches for scenarios in SS for which this parameter is disabled, e.g., it is set to no. If the parameter is disabled the scenario is transferred into DS otherwise it is transferred into IS. Now, the scenarios in the DS list are sorted and searched for duplicate scenarios ignoring the disabled parameter. The duplicate scenario with disabled depender parameter is removed from the DS list since it has no effect on the runtime. The algorithm repeats until all DPs are processed. In the final step the scenarios from DS and IS are merged into SS and present the output scenario set. Once all DPs are evaluated the pruning process is over and the output scenario set contains no meaningless scenarios.

Step to Determine the Signature Scenarios

Next, the component plugin determines the sensitivity of a new program for its tuning approach. This is done by evaluating the signature scenarios and measuring the objective value. The ratio between the objective function values for the signature scenario and a default scenario determines the improvement obtained. Now, the vector of all improvements for the signature scenarios called the signature is available, i.e, the sensitivity of the program for the tuning plugin. The computation of the signature is not costly as it requires only the evaluation of the signature scenarios.

Decision Making

Once the signature is determined, the ASPM meta-plugin must be able to predict the expected tuning benefit from the given signature. The user can configure the meta-plugin

Algorithm 4 Search space pruning.

```
DP                                        ▷ Set of depender parameters
SS                                        ▷ Set of input/output scenarios
DS ← ∅                                    ▷ Set of depender scenarios
IS ← ∅                                    ▷ Set of independent scenarios
while (DP ≠ ∅) do
    select dp ∈ DP
    DP ← DP − dp
    while (SS ≠ ∅) do
        select sc ∈ SS
        SS ← SS − sc
        if sc(dp) = disabled then
            DS ← DS + sc
        else
            IS ← IS + sc
    Expand DS parameters and values as strings
    Sort DS scenarios w.r.t. value
    Remove duplicates from DS
    Remove depender parameters from DS
    SS ← DS ∪ IS
    DS ← ∅
    IS ← ∅
Output Scenarios (SS)
```

to work as a regressor or as a classifier. We have evaluated two different predictors, k-Nearest Neighbors and Support Vector Machine algorithms. Due to the nature of the data and potentially uneven distribution of the data due to the user configuration the k-Nearest Neighbors was less useful than Support Vector Machine algorithm. Therefore, it is used in the PTF implementation.

Depending on the mode of the predictor, the plugin applies the regressor or the classifier. The regressor predicts the expected improvement of the objective by tuning with the plugin. The classifier, on the other hand, determines whether the tuning potential of the application is above or below a given threshold. Once the tuning potential is returned by the component plugin, the meta-plugin triggers the component plugin only if the indicated tuning potential is high according to the threshold used in the classifier/regressor.

Our implementation of regression supports two modes of execution, as a pure regression which predicts the improvement of the objective or as a regression-based classifier.

For instance, it may be only suitable to tune the application given by a particular user, if it can provide at least a reduction of 10% in the execution time of the program using optimized parameters suggested by the component plugin as compared to the base execution

time for the aspect the user is tuning.

The model is trained with libSVM [18].

Improving the Learned Knowledge

While the tuning plugin is used by application programmers for tuning their programs, additional entries collecting the signature, the scenario, and the improvement are stored in a tuning database. This information can improve the learned knowledge, i.e., the set of signature scenarios and the classifier. Therefore, the signature scenario set and the classifier should be retrained periodically by online incremental learning. From the retained data the plugin periodically repeats the procedure of the signature identification and training of the predictor. With the assumption that from more data the plugin can identify higher quality signature and train the better classifier the plugin can consequently make better tuning decisions.

Infrastructure Extensions

This section provides details on the infrastructure extensions that allows the meta-plugin to store the historical performance data and later learn from it. In the previous work [48] Periscope was extended with a simple database. The database was capable of storing execution time and performance counters used as a white-box signature. In order to use the database, the plugin had to be extended and explicitly store the required data. This implementation was now extended and the performance data is stored in a more complex database which is universal and does not require the plugin developer intervention.

Here we discuss the architecture of the new components added to Periscope, their design and the goals we wanted to achieve.

Storing of Historical Performance Data

To enable meta-plugin decision based on machine learning the tuning history has to be stored and therefore the performance database was introduced. The database is designed around the scenario pool concept. The database stores measurements with the context of the execution from the given scenario pool. The context comprises of the program, the plugin that was used, the system, the region to which tuning configurations were applied, and which objective property is used. The database is implemented as SQLite relational database.

The frontend is instructed to collect the tuning history and to store it into the database when the frontend is started. During the tuning process, the plugin collects performance data and stores it in the pools (finished scenario pool, analysis result pool and scenario result pools). When the plugin informs the meta-plugin that it finished the tuning process

the frontend creates the tuning advice and triggers historical performance data store process. Now the frontend searches in the database for the program identity in the table of tuned programs. If the application was not found the new program is added into the table of tuned programs.

Tables used to store the historical performance data:

System: Unique identification of the system where tuning took place.

Plugin: Unique identification of the plugin which was used for tuning.

Program: Unique identification of the tuned program.

Region: The tuned region is a region to which the variant has to be applied.

Tuning Specification: The tuning specification represents the search space point in the search space and contains information about the variant configuration.

Objective Property Request: Defines properties which have to be evaluated for the scenario objective.

Value: The value of the tuning objective calculated by the objective function from objective properties.

Plugin Identity

PTF brings several tuning plugins which tune individual aspects of HPC applications, e.g., MPI. Therefore, the tuning database must be able to uniquely identify each tuning plugin. The plugin is uniquely identified with information built into the plugin like the name of the plugin and the plugin version.

Program Identity

Users can tune many applications with a plugin on a system and therefore the meta-plugin has to be able to differentiate for which application were measurements retrieved. Therefore when we store historical performance data we must be able to uniquely identify the tuned program. Such approach should be as easy as possible for the user but also portable across plugins.

One of the approaches solving this problem, proposed by [48], employs the cryptographic hash function, e.g., SHA1 on the source code of the application. This solution is characterized by the following advantages and disadvantages:

Not portable across plugins: the location of the source code of the application might not be known or has to be specified by the user.

Discriminate too much: Small and irrelevant changes in the source code creates completely different hash code while the resulting binaries can be identical. Also, the proposed approach does not work for programs whose source code is generated and can contain a timestamp. For example, NPB generates the source files and inserts the timestamp into one of the files.

Discriminate too little: Depending on the runtime options or the input dataset the program can have completely different behavior. The runtime options and the input dataset of the program are completely ignored by this approach.

Therefore, we have decided to employ a different approach. The program identity is determined from the program name and the parameters.This information is always available as the user has to specify them before starting the tuning process.

This solution is characterized by the following advantages and disadvantages

Portable across plugin and automatic: the program name and the application parameters are always available since the user has to provide them before he starts the tuning process.

May discriminate too much: since the order of the program options might or might not affect the program execution we distinguish them. This might discriminate applications whose options order does not affect the execution as they are considered to be different runs.

May discriminate too little: the approach is unable to capture if the program input is read from the file. In such case, the identification is left to the user.

System Identity

PTF runs on various Linux clusters and can collect the historical performance data from various systems. Therefore, it is necessary to be able to uniquely identify the system on which the performance data is collected. Once the central historical performance database is available, PTF could store data and make predictions of the plugin behavior on different systems from the performance data collected.

To uniquely identify the system we use the following information:

- System name
- Operating system
- Operating system version
- CPU

- RAM amount
- Compiler
- Compiler Version
- MPI Implementation
- MPI Version

The information is supplied partially by the user in PTF's configuration file, partially collected by the frontend during the runtime, and partially provided by the tuning plugin. PTF's configuration file provides the system name. The frontend collects the operating system information, CPU information and RAM with kernel level calls. The remaining information is plugin specific and provided by individual plugins. Such information could be a compiler for the CFS plugin, MPI implementation and its version for the MPI parameters plugin.

Improved Tuning Model

The tuning model of PTF, introduced in Section 3.3.2, defines a sequence of calls that each plugin must implement. The sequence of calls is controlled by the state-machine. To store the historical performance data the state-machine was extended with a new call. The execution of this call is controlled by the user through the command line parameter which can be enabled or disabled.

5.4.4 Tuning

Configuration File

The Adaptive Sequence with Predictive Model plugins can be configured through the configuration file or through the command line options. The configuration file is split into three parts. The first part configures the general settings of the meta-plugin such as the used component plugins, when prediction takes place and similar. The second part relates to the signature scenario extraction algorithm and the approach specific configuration. The last part relates to the prediction algorithm and its specific configuration.

In the first part of the configuration file, the user can specify general settings of the meta-plugin such as the used component plugins. The user can configure the prediction approach used to decide which component plugins to use for tuning. The allowed values are "Classification" and "Regression". Here, the user can also configure limits when the user considers that the quality of predictions made by the predictive model is good enough. The limit is defined as the number of applications in the historical database. Once

the historical database contains enough applications the meta-plugin can reconsider the predictive model by reevaluating the signature and retraining the predictor as described in Section 5.4.3. The user can specify the threshold as the percentage of new programs in the historical database since the last reconsideration took place. Since the user is interested in the improvement of the tuning objective, e.g., speedup, and not the absolute values, he also defines the default scenario over which all scenario values are normalized. The objective setting is the objective which the meta-plugin predicts and can be set to Minimum, Quartile1, Median, Quartile3, and Maximum. Once the prediction takes place the user declares the expected improvement in the execution, i.e., the speedup, he considers to be significant enough to justify the tuning time.

The second part of the configuration file defines the algorithm used for signature scenarios identification. Currently, the meta-plugin supports two signature scenarios identification approaches, but others are also planned in the future. The first approach, based on clustering, is described in Section 5.4.2 while the second approach, based on information gain, is described in Section 5.4.2. Here, the user also can define algorithm specific configuration. Limiting the maximum number of signature size might make sense for long running applications. The max_signature_size setting determines the maximum size of the signature. The next two settings are relevant only for the k-medoids signature algorithm. The first setting, max_refinements, defines the number of cluster refinements before the final signature is determined. If clusters centers do not stabilize before the maximum refinements are reached, the centers from the max_refinements are reported. Due to the k-Medoid randomness of starting clusters centers, we define max_retries setting which specifies the number of random starting centers. From all these retries we select only the best performing one with respect to the minimum total error of clustering.

The third part of the configuration file defines the algorithm used for prediction. For now, the prediction is only possible with the SVM algorithm. This algorithm can be configured for various prediction algorithms, i.e., classification, regression, and distribution estimation. Besides the prediction algorithm selection, the user can specify which kernel should be used for the prediction. The kernel type can be set to one of the kernels; linear, polynomial, radial basis function (RBF) and sigmoid; presented in Section 5.4.2. We have also assessed the k-Nearest Neighbors algorithm but due to its bias towards majority class, it was not implemented in PTF. In the future, we plan to add new classification algorithms.

An example, presented in Figure 5.10, is the configuration file of the ASPM plugin. The configuration is used for the evaluation of one of the meta-plugin. For simplicity, line 1 in the configuration file, we have specified only a single component plugin, i.e., compilerflags, to be used by the meta-plugin. Line 2, benefit_approach, declares that the algorithm used for the prediction is classification. The benefit_limit setting determines the minimum number of programs before the decision making can take place. If the number of programs in the database is higher than this number the signature is going to be identified and from this signature, the prediction algorithm is trained and makes predictions.

```
1  plugins = "compilerflags";
2  benefit_approach = "Classification";
3  benefit_limit = 20;
4  benefit_reevaluation = 20;
5  default_scenario = 3;
6  objective = "Maximum";
7  threshold "compilerflags" = 1.14339;
8  signature_algorithm = "KMeans";
9  max_signature_size = 20;
10 max_refinements = 20;
11 max_retries = 20;
12 prediction_algorithm = "SVM";
13 svm_type = "C-SVC";
14 kernel_type = "polynomial";
```

Figure 5.10: Example Adaptive Sequence with Predictive Model configuration file.

The re-evaluation criteria in line 4 define that increase of 20% in the number of tuned applications is necessary to identify the signature again and retrain the predictor of the tuning potential. The default_scenario setting defines that scenario 3 is used to normalize values and calculate the improvement of the objective. In this case, the objective is the improvement of the execution time. Therefore, the meta-plugin predicts the maximum expected speedup of the application execution and the objective setting is set to "Maximum". Line 7 defines that the user considers 14.339 % improvement of the execution time is significant enough to justify the tuning time.

The signature algorithm defines k-medoids to be used for the signature identification. The user specifies that the signature scenarios should not be larger than 20 scenarios. The meta-plugin is allowed to do maximum 20 refinements of clusters before the final signature is determined. If their centers do not stabilize within 20 refinements, the centers from the last refinement are reported. Due to k-Medoid randomness, the user also specifies that he is willing to wait for 20 randomly selected cluster centers.

In lines 12, 13, and 14, the user specifies that he wants predictions to be done with clustering by the SVM algorithm with polynomial kernel.

5.4.5 Complete Tuning Flow

Figure 5.11 gives an overview of the complete tuning flow of ASPM plugin.

Plugin initialization: During the initialization, the meta-plugin first loads the configuration file if the user specified it. In the case that the user did not specify the configuration file, the default settings are loaded. Now from the settings, the component

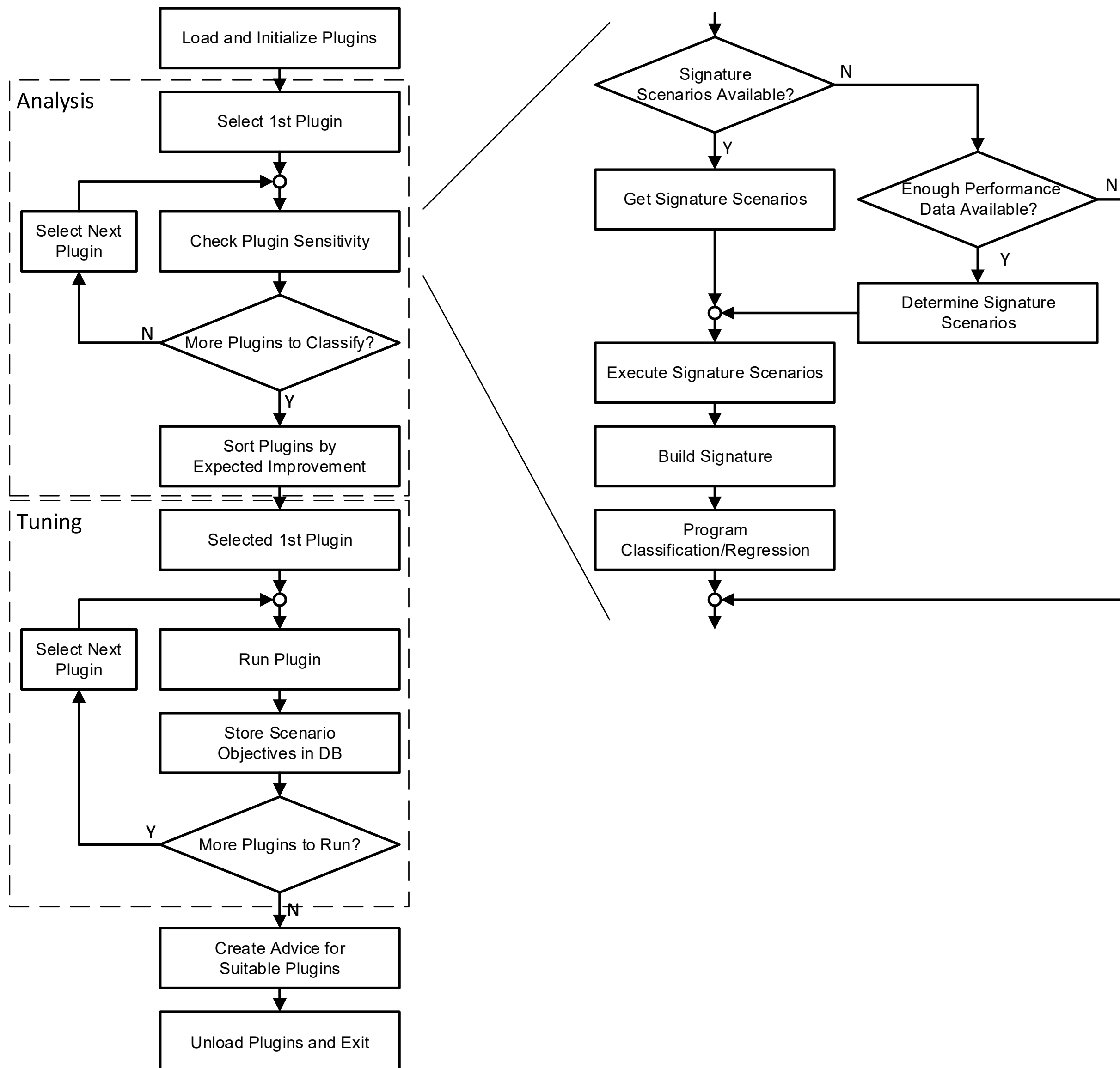


Figure 5.11: The tuning flow of the meta-plugin.

plugins are loaded. In the next step the machine learning techniques necessary for the signature extraction and the prediction of the tuning potential of the plugin is loaded. First, the signature extraction strategy is chosen and the corresponding signature extraction algorithm is loaded. This is followed by the selection of the classification/regression strategy for which the classification/regression algorithm is loaded. The first component scenario is selected for execution.

The automatic plugin decision part:

Start tuning step: The tuning flow of the ASPM plugin can be split into two parts executed in different tuning steps. The first part controls the execution of the automatic plugin decision of individual component plugins while the second part handles the execution of individual plugins.

Pre-analysis: The pre-analysis is not used by the meta-plugin to predict the tuning outcome of individual plugins.

Create scenarios: The meta-plugin first evaluates the signature scenarios if they exist or can be identified from the existing data in the database. The selected component plugin is called to create scenarios and prepare them for evaluation. The scenarios are now transferred from the ESP of the component plugin into the pool set of the meta-plugin. The meta-plugin now checks the application sensitivity. It starts by checking, whether the database contains signature scenarios. In the case where there are no signature scenarios in the database or the threshold for incremental learning was reached, the meta-plugin checks whether the database contain enough data to determine n new set of signature scenarios. If that is not the case the meta-plugin restarts the component plugin and runs the plugin with its complete search space. This will fill the database with more results and finally it will generate the set of signature scenarios. Otherwise, the meta-plugin determines the signature scenarios or fetches them from the database and stores them in the created scenario pool (CSP).

Prepare and execute scenarios: The meta-plugin now prepares and execute the signature scenarios stored in the CSP. Preparation and execution of scenarios is repeated until the created scenario pool is empty. When all scenarios are executed the meta-plugin selects the next plugin for the evaluation.

Start tuning step: The meta-plugin calls the start tuning step method of the component plugin.

Pre-analysis: The meta-plugin configures the pre-analysis if the component plugin requested it.

Create scenarios: The meta-plugin first builds the signature from the evaluated scenarios. The built signature is normalized with the baseline scenario identified by the user. If the signature scenarios were not available in the database the meta-plugin trains the classifier/regressor and stores it in the file. Then the trained model is fed with the signature collected during the first part of the tuning step. The trained model predicts the plugin sensitivity on the tuned program. From the prediction made, the meta-plugin decides about tuning with the plugin. Depending on the outcome of the prediction the plugin might tune with the complete search space or the next component plugin is selected. In the case that the predicted outcome is not to tune the next component plugin is selected and the procedure from the first part is repeated on it.

Prepare and execute scenarios: The meta-plugin now prepares and execute the signature generated for the complete search space from the CSP. Preparation and execution of scenarios is repeated until the created scenario pool is empty. When all scenarios are executed the meta-plugin selects the next plugin for the evaluation.

Give advice: The meta-plugin iterates across the component plugins and collects their tuning advices. The tuning advices are then stored into a single tuning advice file.

Plugin finalization: During the finalization, the meta-plugin finalizes all component plugins and frees the occupied resources.

Chapter 6

Automatically Applied Tuning Advice

Once the tuning process has finished the user implements the best performing optimizations into his code or the runtime environment. To automate this process we have extended PTF with automatically generated tuning advices. The two tuning advices are generated, the runtime tuning advice and the pre-execution tuning advice. The runtime tuning advice consists of the configuration file interpreted and applied by Score-P monitor during the application runtime. The pre-execution tuning advice is a bash script which prepares the application for runtime and configures it. In the future, we plan to extend the generated script for various resource managers.

The plugin configures the automatic tuning advice with the command that is used to assign command line parameters, to assign the environment variable, and prepare the application for the run. Based on this specification the automatic tuning advice is generated. The generated tuning advice expresses the tuning variant through the environment variable, or as an argument of the MPI execution command. The MPI execution command can specify options like the number of processes, the runtime parameter values, e.g., eager limit. Once the variant is exposed through the environment variable, the application is prepared through the configured command. Now, the application is executed and possibly configured through the MPI execution command.

One such pre-execution tuning advice for the CFS plugin is presented in Figure 6.1. The generated tuning advice consists of the suggested variant exported as an environment

```
1 export CFLAGS=-O2
2 make
3 mpiexec -n 2 ./add.exe -a
```

Figure 6.1: Example CFS pre-execution tuning advice.

```
1 SCOREP_REGION_NAMES_BEGIN
2   EXCLUDE *
3   INCLUDE mainRegion
4 SCOREP_REGION_NAMES_END
```

Figure 6.2: Example Score-P filter file.

variable. Since the program was already tuned with the CFS plugin, it has a prepared makefile for such a variable. This is followed by the preparation step defined by the plugin. Once the tuning variant is prepared the application is executed.

6.1 SCORE-P Filter File

The tuning process with PTF requires that many program regions are instrumented. As tuning progresses the plugin often reduces the number of instrumented regions to reduce the measurement overhead as presented in Section 4.1.2. To apply the tuning advice the number of instrumented regions can be reduced even further and restricted only to regions where the runtime tuning advice has to be applied. Therefore, we disable instrumentation for all regions except for regions from the runtime tuning advice. With this approach, we minimize the perturbation by the monitor.

To partially disable the instrumentation during the runtime we employ the Score-P filtering capability. The Score-P filtering can disable the compiler instrumented regions, user instrumented regions, OPARI2 instrumentation, and CUDA device and host activities. Therefore in the pre-execution tuning advice, the filtering file is exposed. The filtering file is shown in Figure 6.2. Regions are filtered based on their name. It is also possible to filter regions based on the source file.

The filtering file can be applied during compile-time or runtime of the application. The compile-time filtering is more efficient as Score-P completely removes the instrumentation from the generated binary but is not always a viable option. In cases when the user can't or doesn't want to recompile the application the instrumentation filtering has to be applied during runtime of the application. In this case, the filter file is read by Score-P during initialization and instrumented regions are not processed. This does not remove the overhead completely but most of it.

The compile-time filtering requires the user to augment the makefile of the application. This is done by adding ADVICE_FILTERING environment variable as one of the options of the scorep instrumenter call. This completely eliminates the perturbation made by the monitor for regions not necessary to apply the tuning advice.

Chapter 7

Evaluation

In this chapter, we demonstrate the techniques presented in the two previous chapters. The evaluation chapter is split into two sections where each section targets evaluation of one contribution. The applications were manually instrumented and tuned with the plugins from the previous chapters.

7.1 Compiler Flags Selection for OpenCL

[1] Evaluation of the Compiler Flags Selection for OpenCL tuning plugin has been performed with several benchmarks from the Rodinia Benchmark Suite [19].

7.1.1 Experimental Setup

The experiments were performed on different architectures including a traditional GPU (NVIDIA K20m, 2496 CUDA cores), an Intel Xeon Phi (5110P, 1.053 GHz, 60 cores), and a CPU (dual Intel Xeon E5-2650, 2.00 GHz, 8 cores).

7.1.2 Rodinia Benchmark Suite

Three benchmarks from Rodinia Benchmark Suite were used:

- LavaMD
- Pathfinder

[1] This is a minor revision of the work published in 49th Hawaii International Conference on System Sciences - HICSS 2016, Koloa, HI, USA, January 5-8, 2016

Flag comb.	1	2	3	4	5	6	7	8	9	10	11	12
-O1	x	x	x	x								
-O2					x	x	x	x				
-O3									x	x	x	x
-loop-vectorize	x	x			x	x			x	x		
-prefetch	x		x		x		x		x		x	

Table 7.1: Compiler flags combinations used in figs. 7.1 to 7.3

- Hotspot

In the following we present evaluation results[2] for the Compiler Flags Selection of the OpenCL tuning plugin for three different benchmarks from the Rodinia benchmark suite[19].
The evaluation is performed using mainly the exhaustive strategy to investigate the complete search space.

The flag combinations explored by the Compiler Flags Selection of the OpenCL tuning plugin are shown in Table 7.1 below.

LavaMD

The LavaMD benchmark calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual cluster nodes. The large box at each node is further divided into cubes, called boxes. 26 neighbor boxes surround each box (the home box). Home boxes at the boundaries of the particle space have fewer neighbors. Particles only interact with those other particles that are within a cutoff radius since ones at larger distances exert negligible forces. Thus the box size is chosen so that the cutoff radius does not span beyond any neighbor box for any particle in a home box, thus limiting the reference space to a finite number of boxes.

Figure 7.1 shows the measured kernel execution times for experiments with three different grid sizes, (a) 5, (b) 10, (c) 20 on the Intel Xeon CPU and the Xeon Phi. There is only a significant impact for sizes (c).

The figure shows that the best combination of flags mainly depends on the type of device, and to a lesser extent on the data set. The best combination of flags on the Intel Xeon Phi is O3 with loop vectorization and prefetching for dataset (a), O2 for dataset (b) and O1 with prefetching for dataset (c). The global best combination for the Xeon Phi, independent of the dataset, is 2 with loop vectorization and prefetching. On the other hand, the best combination of flags on the Intel CPU for dataset size (a) is O2 with

[2]We applied to Compiler Flags Selection of the OpenCL plugin also on NVIDIA K20m, but since no performance differences could be observed for different compiler flags combinations we omit the results.

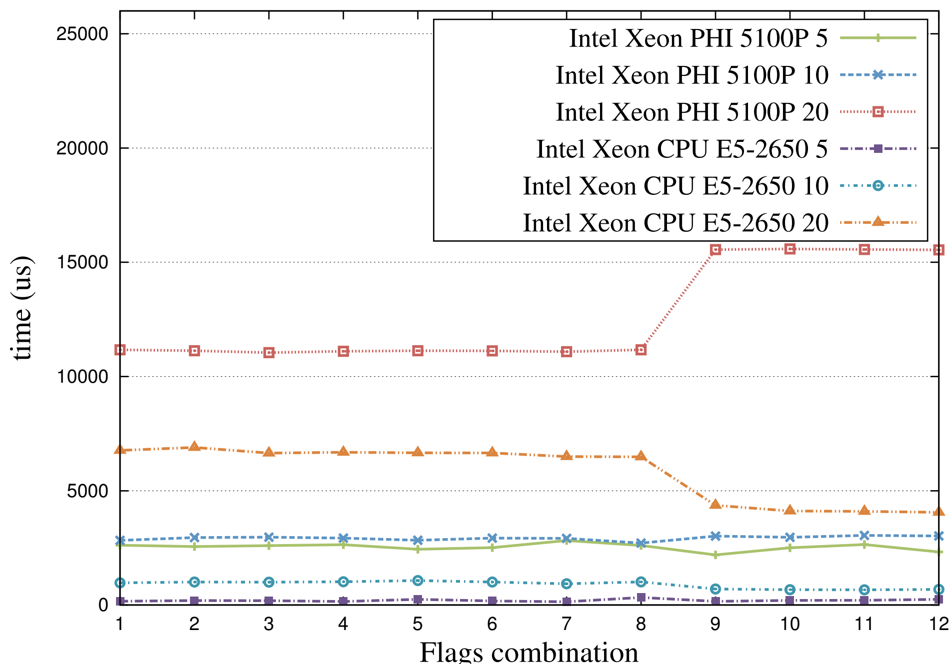


Figure 7.1: The execution time of LavaMD on different devices using three grids: (a) 5, (b) 10, (c) 20.

prefetching, for (b) it is O3 with prefetching and for dataset (c) it is O3. It is worth mentioning that selecting the flags combination highly depends on the device selected. The execution time improved on Xeon Phi between 11% and 29%, while on the CPU the improvement is much bigger, between 38% and 56%, depending on the dataset.

PathFinder

The PathFinder benchmark uses a dynamic programming algorithm to find a path on a 2-D grid from the bottom row to the top row with the smallest accumulated weights, where each step of the path moves straight ahead or diagonally ahead. It iterates row by row, each node picks a neighboring node in the previous row that has the smallest accumulated weight, and adds its own weight to the sum.

Figure 7.2 shows the kernel execution times for three grids with different widths, (a) 100K, (b) 200K, (c) 400K on the Intel Xeon CPU and the Xeon Phi. The figure shows that again the best combination of flags weakly depends on the dataset size but more on the type of device. The best combination of flags on the Xeon Phi is O1 depending on the dataset size with prefetching, (a) and (c), or with loop vectorization, (b). The global best combination for the Xeon Phi, independent of the dataset, is O1 with prefetching. On the other hand, the best combination of flags on the Xeon CPU is again O3 but depending on the dataset size with loop vectorization and prefetching, (a), with loop vectorization, (b), and with

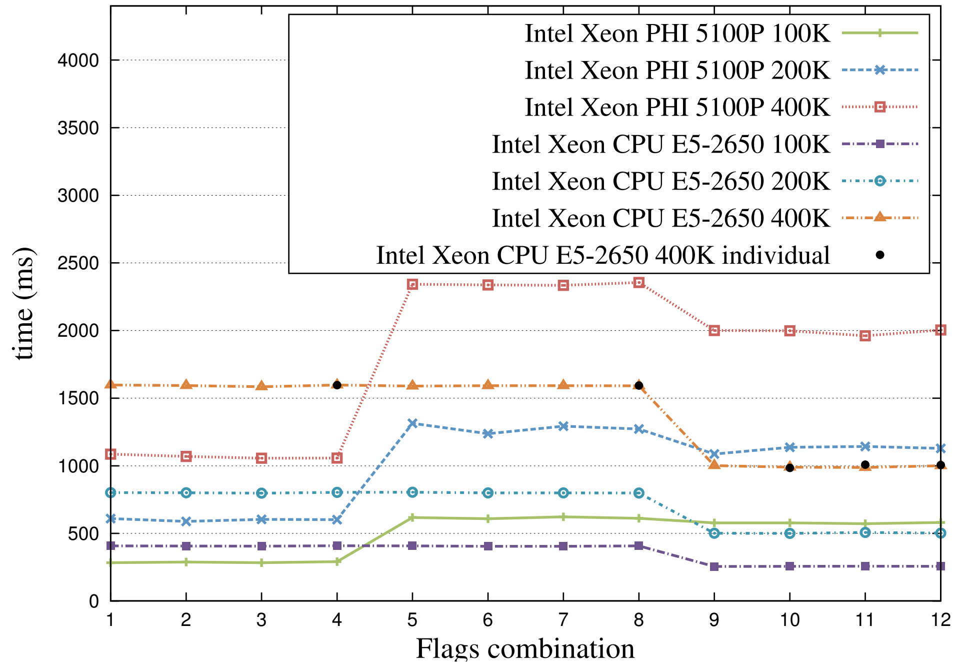


Figure 7.2: The execution time of PathFinder on different devices using three grids with different widths: (a) 100K, (b) 200K, (c) 400K.

prefetching, (c). The global best combination for the Xeon CPU, independent of the dataset, is O3 with loop vectorization and prefetching. It is worth mentioning that again the flag combination highly depends on the device. The execution time improvement, between the slowest and the fastest performing code, on Xeon Phi is around 55%, while on a CPU it is around 37%, depending on the dataset. It is also visible that the biggest performance improvement comes from the optimization levels (O flags), rather than other flags used in the experiment. Other flags caused maximum variation in performance of 6%, compared to 54% of the optimization levels. Experiments with the individual strategy on the CPU with the dataset (c) show that the best flag combination can be found with 60% less evaluated scenarios.

HotSpot

The HotSpot benchmark estimates processor temperature based on an architectural floorplan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip.

Training and evaluation of the automatic prediction of tuning benefit is done with several benchmark suites, namely, CESAR benchmarks, CORAL benchmark codes, ASP Proxy Applications and NAS Parallel Benchmarks.

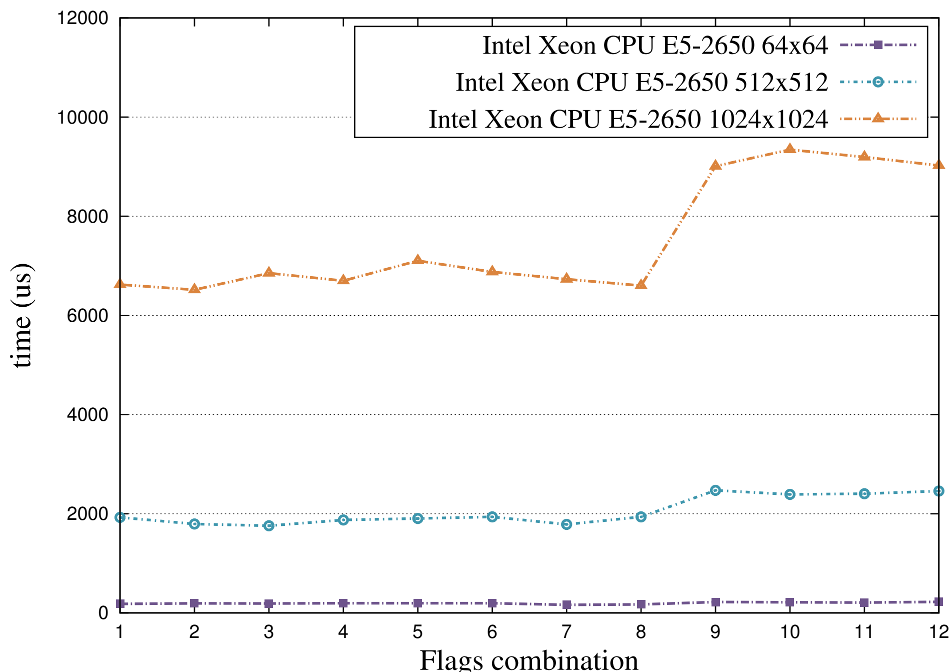


Figure 7.3: The execution time of HotSpot on Intel CPU using three grids: (a) 64 x 64, (b) 512 x 512, (c) 1024 x 1024.

Figure 7.3 shows the kernel execution times for three different grid sizes, (a) 64 x 64, (b) 512 x 512, (c) 1024 x 1024. The experiments were performed only for the Intel Xeon CPU, as on the Xeon Phi the kernel did not run. The figure shows that the best combination of flags slightly depends on the dataset size. The best combination of flags for dataset size (a) is O2 with prefetching, for (b) it is O1 with prefetching, and for (c) it is O1 with loop vectorization. The global best combination of flags is O2 with prefetching, as it results in only 1.6% of lost performance on average, compared to the best result per dataset. It is worth mentioning that selecting the most aggressive optimization results in almost the slowest performing code. The difference between the slowest and the fastest version is between 26% and 30% depending on the dataset.

7.2 Meta-plugins

Evaluation of the meta-plugins is done on benchmarks and proxy applications developed by the US labs and agencies. Proxy applications are applications with simplified characteristics of real applications. Please note that some applications are represented in more than one benchmark/proxy apps suites but are evaluated only once.

7.2.1 Experimental Setup

For the tuning experiments SuperMUC Phase 2, a petascale system hosted by the Leibniz-Rechenzentrum (LRZ) was used. The system has Linpack Performance of 2814 TFlop/s which ranked it as the 21st system in the world by June 2015 TOP500.org list. It consists of 3072 Lenovo NeXtScale dx360M5 nodes with 86016 cores in total and 194 TByte of memory. The individual nodes are built with Haswell Xeon 14 core CPUs with 2 sockets and have 64 GB of RAM. The system consists of 6 thin node islands interconnected with Infiniband FDR14 interconnect with non-blocking tree intra-island topology and pruned tree 4:1 inter-island topology.

Applications were compiled with Intel compiler v16.0 together with IBM's Parallel Environment (IBM MPI). Score-P 4.0 with a tuning substrate plugin and PTF 2.0 were used to respectively instrument, monitor, analyze and tune the test applications.

7.2.2 Benchmark Suites and Proxy Applications

NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [8] are a small set of scientific computational problems formulated to evaluate the performance of parallel supercomputers. The codes are derived from computational fluid dynamics (CFD). The suite consists of five kernels, three pseudo-applications, unstructured computation and two data movement codes. Kernels are:

- IS - Integer Sort, random memory access
- EP - Embarrassingly Parallel
- CG - Conjugate Gradient, irregular memory access and communication
- MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
- FT - discrete 3D fast Fourier Transform, all-to-all communication

Pseudo applications are:

- BT - Block Tri-diagonal solver
- SP - Scalar Penta-diagonal solver
- LU - Lower-Upper Gauss-Seidel solver

Benchmarks for unstructured computation and data movement:

- UA - Unstructured Adaptive mesh, dynamic and irregular memory access
- DC - Data Cube
- DT - Data Traffic

CESAR Proxy Apps

Center for Exascale Simulation of Advanced Reactors (CESAR) [2] from Argonne National Laboratory develops The CESAR Proxy-apps. The applications are split into three classes, each class representing one of the three major components of the overall application. Each class has its own specific communication and major access patterns, and other ways in which it interacts with the hardware. Classes of the Proxy-apps are:

- Thermal Hydraulics - Fluid codes: NEKBONE, NEKBONE KERNEL
- Neutronics: MOCFE, MCCK, XSBench
- Coupling and Data Analytics - data intensive tasks: CIAN

CORAL Benchmark Codes

The CORAL benchmark codes [3] represent the broad range applications expected to dominate the science in the near future. The benchmarks represent envisioned system workloads and provide specific insight into features of the proposed systems. They can be split into five categories:

- Scalable Science Benchmarks are full applications expected to scale to a large fraction of a supercomputer. They are typically single physics applications designed for areas such as material science and combustion.
- Throughput Benchmarks represents particular subsets of applications that are expected to be used as part of the everyday workload of science application with priority on minimizing the overall throughput time.
- Data Centric Benchmarks reproduce the data intensive patterns like integer operations, instruction throughput, and indirect addressing capabilities.
- Skeleton Benchmarks reproduce the memory or communication patterns of a physics application or package.
- Micro Benchmarks are small compute expensive portions of applications, i.e., kernels.

The applications from the suite were updated to the latest versions.

ASC Co-Design Proxy Applications

ASC Co-Design Proxy Applications [1] were developed by three DOE National Labs in Livermore, Sandia and Los Alamos that belong to National Nuclear Security Administration (NNSA). They are used in the co-design process and capture a subset of the characteristics typical in real applications such as:

- Algorithms
- Data structures and memory layout
- Parallelism and I/O models
- Languages and coding styles

They consist of different types of proxy apps:

- Kernels are small compute intensive code fragments or data layouts that are used extensively by the applications.
- Skeleton apps
- Mini apps

LLNL Proxy Apps [6] consists of:

- AMG2013 is a C, MPI, OpenMP algebraic multi grid (subset of hypre library) application.
- Kripke is a C++, MPI, OpenMP sweep based, structured mesh deterministic transport application.
- Lassen is a C++, MPI highly load-imbalanced front-tracking through a 2D mesh application.
- LCALS is a C++, OpenMP suite of kernels in a unified framework for testing compilers, SIMD and threading.
- LULESH is a C++, MPI, OpenMP explicit lagrangian shock hydrodynamics on unstructured mesh representation application.
- MCB is a C++, MPI, OpenMP Monte Carlo Particle Transport application.

LANL Proxy Apps [5] consists of:

- SNAP is a F90/95, MPI, OpenMP SN (Discrete Ordinates) Application Proxy.

- PENNANT is a C++, MPI, OpenMP Unstructured Mesh Hydrodynamics application.
- CLAMR is a C++, MPI, OpenMP, OpenCL Cell-Based Adaptive Mesh Refinement application.
- NuT is a C++ Monte Carlo code for Neutrino Transport application.

SNL Mantevo Proxy Apps [7] consists of:

- CloverLeaf is a F90, C, MPI, OpenMP compressible Euler equations on a Cartesian grid application.
- CoMD is a C, MPI, OpenMP Molecular Dynamics with array-of-structures and structure-or-arrays data layout application.
- MiniAero is a C++, MPI unstructured finite element application.
- MiniAMR is a C, MPI 3D stencil calculation with Adaptive Mesh Refinement application.
- MiniFE is a C++, MPI, OpenMP unstructured implicit finite element codes application.
- MiniMD is a C++ force computation in a typical Molecular Dynamics application.
- MiniGhost is a F90, C, MPI difference stencil of a homogeneous 3D domain application.
- MiniXyce is a C++, MPI electrical modeling Xyce application.
- Pathfinder is a C, OpeMP signature search application.

7.2.3 Kernels and Parameter Settings

Finding the optimal combination of the parameters for SVM and its kernels 5.4.2 is not an easy task. For that purpose we employ, grid search. The method explores the search space of various parameters shown in Figure 7.2 by visiting points specified in the grid. Once the optimal range of points is found we refine the search closer to that point.

Table 7.2: Kernel parameter settings and their ranges

	Kernel			
Kernel	Linear	Poly	RBF	Sigmoid
Gamma (γ)	$\emptyset$	$1e-2, 1e-1, 1, 1e1, 1e2$		
Epsilon (ϵ)	$1e-5, 1e-4, 1e-3$			
Cost (C)	1, 3, ..., 3000, 10000			
Degrees (d)	$\emptyset$	1, 2, 3	$\emptyset$	$\emptyset$
Coefficient (coef0)	$\emptyset$	0, 1		
Search space size	27	810	270	270

7.2.4 Evaluation Methodology

Evaluation of the meta-plugin's predictor for CFS plugin was done on 52 benchmarks while MPI parameters plugin was done on 35 benchmarks from various benchmark suites presented in the previous section. This gives us a great dataset compared to datasets presented in Section 2. To effectively use this dataset, circumvent the problem with limitations of the dataset size, and give the best possible evaluation of the predictor's quality of predictions we employ the cross-validation techniques.

The cross-validation techniques can, in general, be split into two classes, exhaustive and non-exhaustive. We have considered two non-exhaustive techniques, K-fold cross-validation and Holdout method. K-fold cross-validation randomly splits the dataset into k subsets. From the k subsets, one subset is used for the testing while k-1 subsets are used for training. The process is repeated k times (folds) until all k subsets were used once for testing. The resulting k validation results are averaged to produce a single estimation. Holdout method randomly splits the dataset into one for training and one for validation. However, this has a huge disadvantage for a small dataset since the training set might not be representative for the validation dataset which might result in a huge variance. All these methods are approximations of leave-p-out cross-validation.

Leave-one-out Cross-validation

Although the necessary effort to collect this dataset is significant, from the perspective of machine learning its size is considered to be small, below 100 programs (dimensions). Therefore, we have decided to use the exhaustive validation technique called leave-one-out cross-validation. Leave-one-out cross-validation splits the dataset of n elements into individual elements. It selects n-1 elements for training and one element for testing. The process is repeated until all n elements were used once for testing. Once n estimations of accuracy were made, the final estimation is computed as average. This testing method ensures that predictions for a specific program has never been trained with and also gives fair evaluation accuracy.

The dataset is collected by evaluating many different combinations of tuning parameters (scenarios) with a large number of program runs with the same dataset and the environment (architecture, OS, compiler). In the case of the CFS plugin, the search space of 1075200 scenarios is reduced to a set of 500 candidate scenarios which are evaluated on a set of 52 programs from the described benchmark suites. Therefore, the dataset consists of 52 x 500 scenarios = 26000 scenarios. Tuning of programs and collecting their tuning objectives for dataset requires circa 50 days.

Mean Error of Regression Prediction

To measure the performance of the prediction of the tuning potential we have employed *relative mean absolute error* (RMAE) and *root mean squared error* (RMSE). Both methods measure average errors of the predicted value $\hat{y}_i$ and the real value y_i for all points in the test set. They don't take into account the error direction. With such an approach, positive and negative errors do not cancel out. RMAE is expressed in Equation 7.1 while RMSE is expressed in Equation 7.2. To simplify the comparison between different datasets values of both scoring methods are normalized. Because RMSE squares values before they are averaged it is more sensible to large errors compared to RMAE.

$$RMAE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{7.1}$$

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{7.2}$$

Measurements of Classification Performance

A confusion matrix is commonly used in the analysis of classification performance. Figure 7.4 provides relation between predicted outcome and actual values of the binary classifier. The binary classifier predicts two classes, positive and negative. Predicted values can of positive class, p, and negative class, n. The total number of the positive and the negative predicted class members is labeled with P and N, respectively; while the total number of actual class members is labeled with P′ and N′, respectively. Also, actual values can be of positive class, p′, and negative class, n′. The classifier can predict these values correctly or incorrectly. The matrix allows more detailed analysis of the prediction quality. Thus, for two classes it defines four possible outcomes of the classification. For example, if the classifier predicts that the value is positive (p) but the actual value is negative (n′) we call this false positive (FP). Analogous, we define a true positive (TP), true negative (TN), and false negative (FN). The number of true positive, true negative, false positive, and false negative, is labeled with N_{TP}, N_{TN}, N_{FP}, N_{FN}, respectively. From

Prediction outcome

		p	**n**	**total**
Actual value	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Figure 7.4: The confusion matrix reports the number of true positives, true negatives, false positives, and true negatives.

the provided classification outcomes we can express accuracy, a well-known measure of classifier performance, as in Expression 7.3.

Due to the accuracy paradox, accuracy is not a very good measure of classifier quality if the dataset consists of the dominant class. Let's imagine a binary classifier which always predicts positive value. If in the test dataset we have 95% of positive values it might appear that this is a good classifier. To circumvent the well-known problems of accuracy with a dominant class datasets we have decided to employ F1-score of test's accuracy. Therefore, we have used precision and recall. Equation 7.4 defines precision which measures the fraction of true predictions for the positive class. Equation 7.5 defines recall, sometimes called true positive rate, which measures the fraction of predicted positives of all positives in the dataset. A similar measure, called true negative rate 7.6, measures the fraction of predicted negatives of all negatives in the dataset. Recall or true positive rate is a good metric in the case the user does not want to miss tuning opportunities while he might be willing to sacrifice some more time by tuning applications with negligible tuning potential. On the other hand, if the user is more willing to sacrifice tuning opportunities instead of time he could consider true negative rate to be a better metric.

$$Accuracy = \frac{N_{TP} + N_{TN}}{N_{TP} + N_{TN} + N_{FP} + N_{FN}} \tag{7.3}$$

$$Precision = \frac{N_{TP}}{N_{TP} + N_{FP}} \tag{7.4}$$

$$Recall = \frac{N_{TP}}{N_{TP} + N_{FN}} \tag{7.5}$$

Table 7.3: Intel 16 compiler parameters and their respective values used for evaluation

Parameter	Values
Optimization level	-O0, -O1, -O2, -O3
Function inlining	-finline-functions, -fno-inline-functions
Force inlining	-inline-forceinline, ∅
Disable standard library inline	-nolib-inline, ∅
Aggressive loop unrolling	-unroll-aggressive, -no-unroll-aggressive
Unroll all loops	-funroll-all-loops, ∅
Align function byte boundary	-fno-align-functions, -falign-functions=16
Enable/disable ANSI aliasing	-ansi-alias, -no-ansi-alias
Enable streaming stores	-qopt-streaming-stores=[always, never, auto]
Register alloc. region strategy	-qopt-ra-region-strategy=[rout., bl., trace, loop, def.]
Interprocedural optimization	-ip, -no-ip
Interprocedural opt b/w files	-ipo[0, 1, 2, 4, 8, 16], -no-ipo
Prefetch	-qopt-prefetch=[0,1,2,3,4]
Loop blocking factor	-qopt-block-factor=[2,16]

$$\text{True negative rate} = \frac{N_{TN}}{N_{TN} + N_{FP}} \tag{7.6}$$

$$F1 - score = 2 \cdot \frac{Precision \cdot Recall}{(Precision + Recall)} \tag{7.7}$$

These two measures easily identify a bogus classifier. In the case that all predictions by the classifier are positives recall is equal to 1, but precision will be very bad. To express and balance both measures as equally important a single value F1-score is used. F1-score is defined as in Expression 7.7.

7.2.5 Compiler Flags Selection

The selection of flags is done based on the Intel Compiler and Reference Guide [25] and the compiler flags study from CERN openlab [14]. We have selected 14 program transformations such as inlining, unrolling, etc., among many others and which are known to influence performance.

The general optimization level: Enables groups of optimizations. Every higher level contains optimizations from previous levels with some new optimizations.

Functions inlining: Enables function inlining for single file compilation.

Force inlining: Instructs the compiler to force inlining of functions suggested for inlining whenever the compiler is capable doing so.

Disable standard library inline: Disables inline expansion of standard library or intrinsic functions.

Aggressive loop unrolling: More aggressive loop unrolling for certain loops.

Unroll all loops: Unroll all loops even if the number of iterations is uncertain when the loop is entered.

Align function byte boundary: Align functions on an optimal byte boundary.

Enable/Disable ANSI aliasing: Enables or disables usage of ANSI aliasing rules in optimizations.

Enable streaming stores: Enables generation of streaming stores for optimization. Streaming stores can be generated always, never or auto (decided by the compiler).

Register allocation region strategy: Selects the method that the register allocator uses to partition each routine into regions. Possible strategies are routine, block, trace, loop, default.

Interprocedural optimization: Determines whether additional interprocedural optimizations for single-file compilation are enabled.

Interprocedural optimization between files: Determined whether interprocedural optimization is enabled between files. n specifies the number of object files the compiler should create.

Prefetch: Enables or disables software prefetch insertion optimization. 0 stands for disabling software prefetch, 1-4 enables different levels of software prefetching.

Loop blocking factor: Specification of a loop blocking factor.

Measurements

Figures 7.5, 7.6, and 7.7 show the sample space's distribution of speedups or slowdowns for each program across the compiler flags configurations with the calculated speedup relative to scenarios 24, 96, and 6. Each program is represented by the box and whisker plot which summarize the distribution of speedups for that program. The box is represented by the 1st quartile (25th percentile) and 3rd quartile (75th percentile). The line inside of the box represents the median speedup. While the whiskers represent the extremes, i.e., the minimum and the maximum speedups or slowdowns.

Two important observations can be made from the presented figures. First, the effect of optimization flags can be very significant, and second, the scenario used for the speedup

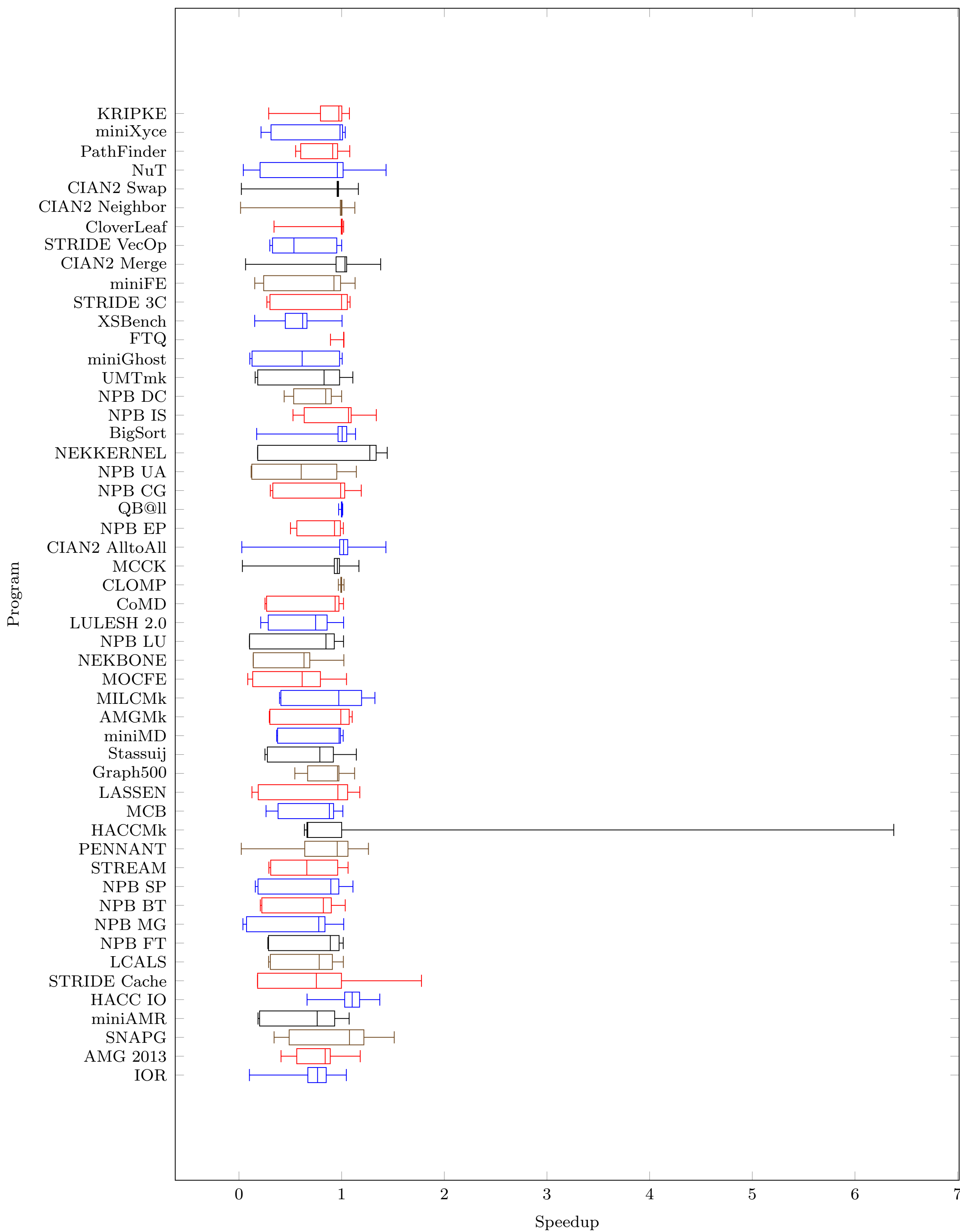


Figure 7.5: The speedup achieved on various benchmarks with CFS plugin relative to scenario 24.

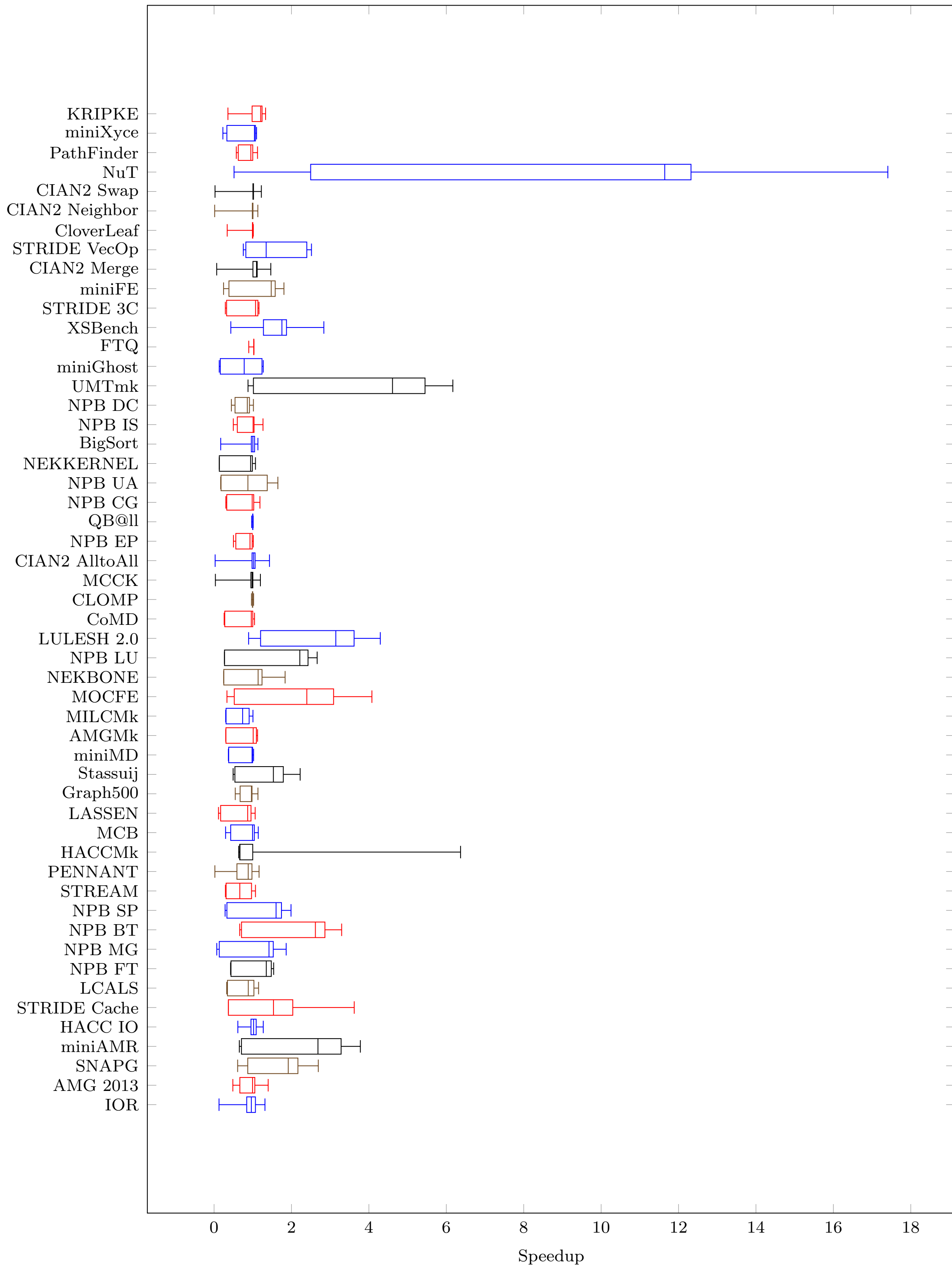


Figure 7.6: The speedup achieved on various benchmarks with CFS plugin relative to scenario 96.

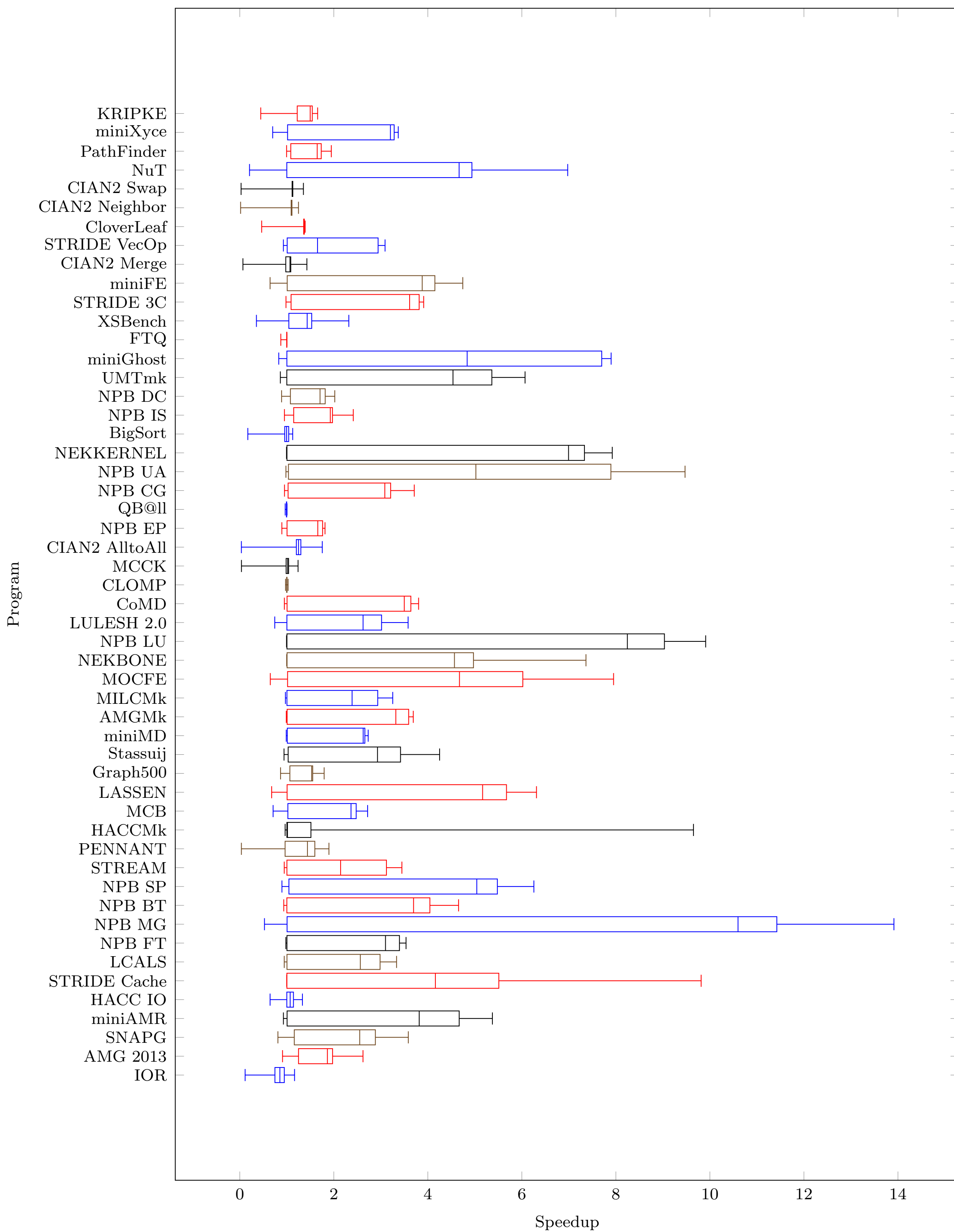


Figure 7.7: The speedup achieved on various benchmarks with CFS plugin relative to scenario 6.

calculation plays an important role in values that have to be predicted. For some benchmark programs, the optimization flags can have a significant effect. Some programs, such as NuT, can achieve up to 18x speedup depending on the scenario used for the speedup calculation. The selection of the scenario used for optimization can have a significant effect on the performance of the predictor which can be seen in the coming sections.

Table 7.4 shows the expected speedup, the predicted speedup, and the signature used to predict such speedup. Signature scenarios are selected using the information gain method. The scenario with id 96 is used to calculate the speedup.

Signature

The very important step that has a significant impact on the quality of predictions is the selection of the proper signature. For that purpose, we have employed two approaches, clustering and information gain. The selection of the optimal number of signature scenarios, e.g., the number of clusters, is based on the Elbow method, explained in Section 2. This method is applied to both approaches.

Figure 7.8 shows the dependence between the cost of clusters and the number of clusters of the first approach. The cost is calculated as the Euclidean distance from the center of the cluster. The method finds the highest reduction of the cost of clusters with the minimum number of clusters, represented by the black dot. The line above shows the distance between the expected and the real reduction in cost. Based on the shape of the graph, we can see that the cost of clusters can even grow with the higher number of clusters. This is normal since the starting cluster centers of the k-Medoid algorithm are selected randomly. We have generated their centers 20 times and selected clustering with the smallest error but the fluctuations still exist.

Figure 7.9 shows the dependence between the gained information and the number of signature scenarios. Unlike the clustering based signature scenario selection, the reduction of the growth of information gain is much slower and not very visible. However, information gain is completely deterministic and as we will see, in the coming sections, the predictors on average give better predictions from signature scenarios determined with it.

Table 7.5 shows the relation between the number of scenarios collected by the component plugin during the regular tuning and the quality of predictions the meta-plugin makes from this data. As we can see the quality of predictions seems to improve with the higher number of scenarios available for a plugin even though the signature size is lower than in the most cases.

Classification

In this section, we present the prediction quality obtained for two methods used to determine the signature scenarios and four supported SVM kernels. To give a better view on

Table 7.4: The speedup achieved with Compiler Flags Selection plugin on dataset relative to scenario 96.

Outcome		Signature								
Real	Pred	96	314	157	480	270	180	282	430	422
1.31	1.40	1.00	0.91	1.16	1.11	0.78	1.08	0.86	1.11	1.20
1.40	1.51	1.00	1.29	1.09	1.01	1.02	1.18	1.23	1.13	1.15
2.69	2.71	1.00	2.28	2.43	2.34	2.12	2.14	2.31	2.32	2.33
3.78	4.23	1.00	3.45	3.71	3.43	3.33	3.76	3.42	3.19	3.32
1.27	1.35	1.00	1.26	1.04	1.07	1.21	1.15	1.17	1.02	1.18
3.62	3.11	1.00	3.57	1.94	3.54	2.05	3.54	2.05	3.50	2.04
1.15	1.54	1.00	1.03	0.99	1.03	1.08	1.13	1.14	1.01	1.14
1.54	1.69	1.00	1.48	1.46	1.50	1.49	1.50	1.51	1.46	1.50
1.86	1.91	1.00	1.81	1.77	1.79	1.52	1.80	1.81	1.54	1.50
3.30	3.48	1.00	3.14	3.13	3.23	2.87	3.20	3.16	2.76	2.84
1.99	2.12	1.00	1.75	1.84	1.81	1.66	1.82	1.65	1.84	1.67
1.07	0.86	1.00	1.00	0.66	0.97	0.71	1.02	0.70	0.64	0.71
1.16	1.69	1.00	1.08	1.00	0.94	1.02	0.14	0.91	0.99	1.04
6.37	0.95	1.00	0.67	0.85	0.67	1.18	1.00	1.18	0.67	1.00
1.14	1.18	1.00	1.00	1.01	1.00	1.11	1.02	1.01	0.98	0.99
1.06	1.22	1.00	0.97	0.97	0.98	0.91	1.01	0.99	1.00	1.01
1.13	1.06	1.00	0.96	0.96	0.98	0.98	0.95	0.97	0.97	0.98
2.22	2.04	1.00	1.80	1.71	1.81	1.72	1.83	1.97	1.83	1.91
1.02	1.08	1.00	0.99	0.98	0.96	1.02	0.98	0.99	1.00	0.99
1.12	1.18	1.00	1.04	1.01	0.94	1.10	1.03	1.01	0.99	1.08
1.00	0.62	1.00	0.71	0.91	0.71	0.99	0.75	1.00	0.71	0.72
4.08	3.43	1.00	3.98	3.84	3.99	3.11	4.02	3.76	3.02	2.81
1.84	1.48	1.00	1.73	1.76	1.81	1.23	1.71	1.72	1.23	1.22
2.66	2.85	1.00	2.59	2.65	2.62	2.39	2.59	2.66	2.37	2.36
4.29	5.00	1.00	3.72	3.73	3.67	4.07	4.12	4.03	3.46	3.77
1.05	1.16	1.00	1.03	0.98	0.99	0.94	1.03	0.99	1.02	1.01
1.02	1.14	1.00	1.01	1.00	1.00	1.00	0.99	0.99	0.99	1.00
1.20	1.06	1.00	0.93	1.03	0.86	1.02	0.97	0.96	1.01	0.97
1.43	1.14	1.00	1.04	1.04	1.08	1.08	1.08	0.99	1.05	0.97
1.01	1.09	1.00	0.95	0.95	0.91	1.00	1.00	1.00	0.95	1.00
1.01	1.15	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.01	1.01
1.19	1.21	1.00	1.00	1.03	1.03	1.08	0.98	0.92	1.09	1.08
1.65	1.66	1.00	1.41	1.44	1.65	1.32	1.49	1.43	1.40	1.34
1.07	0.97	1.00	0.90	0.98	0.92	1.05	0.95	0.98	1.01	0.83
1.13	1.09	1.00	0.85	1.01	0.97	0.96	0.99	0.91	1.08	1.01
1.26	1.18	1.00	1.04	1.03	1.03	0.99	1.04	1.06	1.01	1.07
1.02	1.05	1.00	0.83	0.99	0.91	0.86	0.94	0.83	0.90	0.90
6.17	7.28	1.00	5.78	5.19	5.88	5.61	5.16	5.18	5.74	5.64
1.27	1.29	1.00	1.15	1.24	1.21	1.25	1.25	1.26	1.27	1.25
1.03	1.17	1.00	1.03	1.03	1.03	1.03	1.03	1.03	1.03	1.03
2.84	2.30	1.00	1.84	2.19	1.87	1.88	1.84	2.18	1.86	2.78
1.16	1.13	1.00	0.99	1.09	0.98	1.14	0.99	1.00	1.01	1.07
1.81	1.69	1.00	1.55	1.71	1.48	1.78	1.55	1.61	1.50	1.58
1.47	1.24	1.00	1.10	1.10	1.11	1.19	1.18	1.06	1.11	1.00
2.52	2.85	1.00	2.46	2.51	2.46	2.39	2.46	2.51	2.39	2.51
1.01	1.17	1.00	0.99	0.99	0.70	0.99	1.00	1.00	1.00	1.00
1.13	1.14	1.00	1.01	1.00	1.01	1.00	1.00	1.06	1.00	1.00
1.22	1.16	1.00	1.01	1.02	1.02	1.01	1.01	1.02	1.03	1.04
17.41	14.51	1.00	11.87	11.80	11.84	11.84	12.32	11.70	11.91	12.28
1.12	1.15	1.00	1.02	1.01	0.95	0.93	0.98	1.05	1.03	1.00
1.10	1.24	1.00	1.09	1.08	1.07	1.03	1.09	1.08	1.04	1.06
1.33	1.51	1.00	1.23	1.22	1.25	1.22	1.21	1.24	1.16	1.24

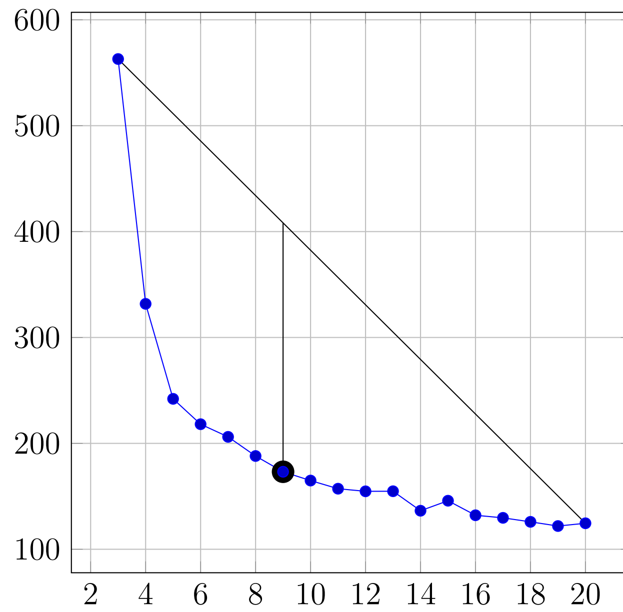


Figure 7.8: The cost as the function of the number of signature scenarios represented as clusters.

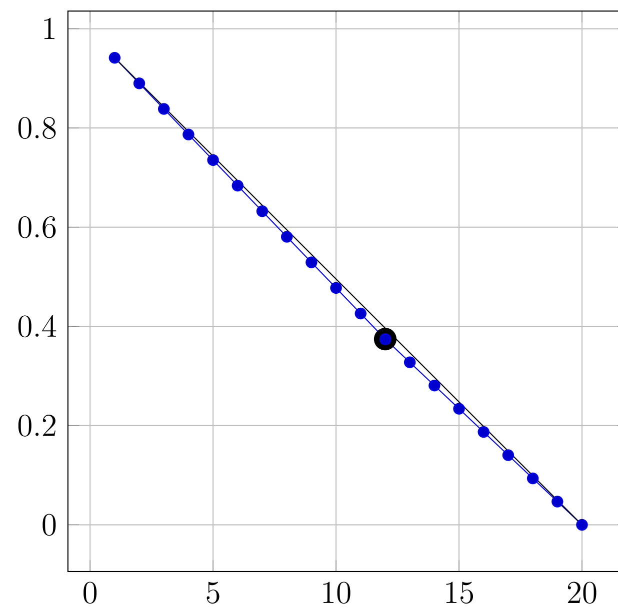


Figure 7.9: The cost as the function of the number of signature scenarios.

Table 7.5: Relation between the number of scenarios and classification quality measures for speedup relative to scenario 6 for the linear SVM kernel with threshold 1.23941. Signature determined with k-medoids.

	Number of scenarios				
Measure	100	200	300	400	500
Signature size	11	9	13	13	10
Precision	1.000000	1.000000	1.000000	1.000000	1.000000
Recall	0.956522	0.956522	0.978261	0.956522	1.000000
True negative rate	1.000000	1.000000	1.000000	1.000000	1.000000
Accuracy	0.961538	0.961538	0.980769	0.961538	1.000000
F1-score	0.977778	0.977778	0.989011	0.977778	1.000000
C	3	10	10	1	10
Epsilon (ϵ)	0.000010	0.000010	0.000010	0.000010	0.000010

Table 7.6: Settings used by the meta-plugin for CFS plugin and the distribution of expected meta-plugin's decisions (don't tune, tune) for benchmarks.

Scenario	Threshold	Don't tune	Tune
24	1.01181	6	46
24	1.07915	26	26
96	1.02140	6	46
96	1.26419	26	26
6	1.23941	6	46
6	3.33472	26	26

the predictions quality, we have calculated the speedup relative to three different scenarios; i.e., 24, 96, and 6; and different thresholds. The first threshold setting is determined to split benchmarks into 6 benchmarks with significant tuning potential and 46 with insignificant potential. The second threshold setting is determined to split benchmarks into 26 benchmarks with significant tuning potential and 26 with insignificant. This gives 6 different settings used to evaluate the prediction quality. Table 7.6 shows all the settings used by the meta-plugin for CFS plugin, settings differ depending on the scenario used to calculate the speedup, presented in Figures 7.5, 7.6, and 7.7.

The quality of classification is presented in tables for both signature scenarios identification methods, information gain and k-medoids. The optimal kernel is selected by the meta-plugin based on the F1-score metric presented in Section 7.2.4 in Equation 7.7. For each kernel's results presented in the next two sections; we have compared minimum, average, and maximum of precision, recall, true negative rate, accuracy and F1-score metrics. These metrics are presented in this chapter for completeness and to provide more insights into the quality of predictions.

Table 7.7: Comparison of SVM kernels used for classification with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with information gain. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	4/9.5/14	4/9.5/14	4/9.5/14	4/9.5/14
Precision	0.88/0.93/1.00	0.88/0.95/1.00	0.83/0.95/1.00	0.37/0.83/1.00
Recall	0.62/0.90/1.00	0.81/0.93/1.00	0.88/0.95/1.00	0.38/0.80/1.00
True negative rate	0.00/0.64/1.00	0.00/0.76/1.00	0.50/0.80/1.00	0.00/0.41/1.00
Accuracy	0.79/0.90/1.00	0.85/0.93/1.00	0.87/0.94/1.00	0.37/0.79/0.90
F1-score	0.74/0.91/1.00	0.84/0.94/1.00	0.87/0.95/1.00	0.38/0.81/0.95

Information Gain

Table 7.7 compares quality of predictions of various kernels used for classification from signature scenarios identified by information gain. Due to information gain's deterministic nature, the number of scenarios and scenarios that comprise the signature is equal for all kernels.

Besides the prediction quality, we have also compared the mean training time on the per-program basis. As our training set consists of 52 programs, the complete training time is divided by 52. The linear kernel has reasonable training time of under 1 minute and F1-score between 74.42% and 100%, with the average value of 90.67% for each program. The polynomial kernel has very big variations of training time from under 1 minute to an hour, due to numerical instabilities, which should be taken into account when selecting it. Its F1-score is between 84.00% and 100%, with the average value of 93.78%. RBF has reasonable training time of under 1 minute and F1-score between 87.27% and 100%, with the average value of 94.93%. Sigmoid has reasonable training time of under 1 minute and F1-score between very low 37.74% and 94.85%, with the average value of 80.76%.

From the information provided in the table, we can conclude that the best prediction quality is achieved with the RBF kernel. This kernel also provided very short training time, usually less than a minute per program. On the other hand, the sigmoid kernel achieved the worst prediction quality with F1-score as low as 37.74%. Also, the polynomial kernel seems impractical due to its unpredictable long training time as a consequence of numerical instability which ranged from less than a minute up to an hour, 36 hours in total. However, its prediction quality is quite good with respect to the performance of precision metric which is the best. The last remaining kernel, provided reasonable training time, but the quality of predictions is relatively low. The most concerning fact for all other kernels, except RBF, is that the resulting true negative rate is 0.00. This happened in the case where the speedup is calculated relative to scenario 96 with the threshold set to 1.02140 or 2.14% of improvement.

Table 7.8: Comparison of SVM kernels used for classification with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	8/9.67/11	8/10/13	8/10/13	8/10.5/15
Precision	0.80/0.90/1.00	0.83/0.92/1.00	0.69/0.90/0.98	0.27/0.76/0.92
Recall	0.62/0.89/1.00	0.58/0.90/1.00	0.69/0.91/1.00	0.23/0.77/1.00
True negative rate	0.00/0.64/1.00	0.00/0.63/1.00	0.00/0.63/0.96	0.00/0.40/0.92
Accuracy	0.73/0.88/1.00	0.73/0.90/1.00	0.69/0.89/0.98	0.31/0.76/0.92
F1-score	0.70/0.89/1.00	0.68/0.91/1.00	0.69/0.90/0.99	0.25/0.76/0.96

K-medoids

Table 7.8 compares quality of predictions of various kernels used for classification from signature scenarios identified by k-medoids. Due to k-medoid's non-deterministic nature, the number of scenarios that comprise the signature varies for all kernels.

From the information provided in the table, the polynomial kernel provided the best prediction quality, ranges between 68.18% and 100%, on average 90.69%. However, it suffers from long training time due to numeric instabilities which range from less than a minute to more than an hour. The RBF kernel provided slightly worse prediction quality, however, its predictable training time is its biggest advantage. The F1-score ranges between 69.23% and 98.92%, on average 90.45%. The linear kernel, again, provided fairly good prediction quality with F1-score ranging from 69.57% to 100%, with on average 89.42%. The sigmoid kernel is again problematic and its lowest F1-score is between 25% and 95.83%, with the average value is 76.09%.

From the signature scenarios determined by the k-medoids algorithm, none of the compared kernels could determine programs that provide insignificant tuning potential for the speedup calculated relative to scenario 24 with the threshold set to 1.01181 or 1.18% of improvement.

Comparison of Information Gain and K-medoids

Table 7.9 compares information gain and k-medoids methods for a predictor configured as classifier. The values present the relative difference between the information gain and k-medoids average values for the speedup calculated relative to scenarios 24, 96, and 6 according to Table 7.6. It can be seen that on average k-medoids is inferior to information gain in every measure for classification. The information gain method provided signature scenarios which contain fewer features (scenarios) compared to the k-medoids method. From fewer signature scenarios, information gain makes better predictions than

Table 7.9: Comparison of information gain and k-medoids for various SVM kernels used for classification with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with two thresholds (6/46, 26/26).

	Kernel				
Measure	Linear	Poly	RBF	Sigmoid	Average
Signature size	0.982759	0.950000	0.950000	0.904762	0.943760
Precision	1.030061	1.034109	1.057340	1.091945	1.061003
Recall	1.007194	1.024360	1.041209	1.041667	1.024430
True negative rate	0.996678	1.195286	1.271187	1.026596	1.011637
Accuracy	1.014493	1.032143	1.054152	1.042373	1.028433
F1-score	1.013920	1.032890	1.049450	1.061330	1.037627

k-medoids. This is especially true for the minimum expected prediction quality which on average contains 0.54 fewer scenarios while having on average 3.76% higher F1-score, 6.10% accuracy, 2.44% true negative rate, 1.16% recall, and 2.84% precision. The only higher measure for k-medoids is the true negative rate for the linear kernel which is 0.30% higher than information gain equivalent.

What is interesting to notice is that good selection of much smaller signature can bring much better prediction quality as it can be seen for predictions made by the information gain method where the speedup is calculated relative to scenario 24 with the threshold set to 1.01181 or 1.18% of improvement.

Regression

In this section, we present the regression prediction quality obtained for two methods used to determine the signature scenarios and four supported SVM kernels. To give a better view on the predictions quality, we have calculated the speedup relative to three different scenarios; i.e., 24, 96, and 6; and different thresholds. The first threshold setting is determined to split benchmarks into 6 benchmarks with significant tuning potential and 46 with insignificant potential. The second threshold setting is determined to split benchmarks into 26 benchmarks with significant tuning potential and 26 with insignificant. Table 7.6 shows all the settings used by the meta-plugin for CFS plugin, settings differ depending on the scenario used to calculate the speedup, presented in Figures 7.5, 7.6, and 7.7.

The results of the quality of regression are presented in tables for both signature scenarios identification methods, information gain and k-medoids. The optimal kernel is selected by the meta-plugin based on the RMSE metric presented in Section 7.2.4 in Equation 7.7. For each kernel's results presented in the next two sections, we have compared minimum, average and maximum of precision, recall, true negative rate, accuracy, F1-score, RMSE, and RMAE metrics. These metrics are presented in this chapter for completeness and to

Table 7.10: Comparison of SVM kernels used for regression with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with information gain. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	4/9.5/14	4/9.5/14	4/9.5/14	4/9.5/14
Precision	0.83/0.93/1.00	0.69/0.91/1.00	0.92/0.96/1.00	0.26/0.79/1.00
Recall	0.73/0.92/1.00	0.77/0.93/1.00	0.88/0.96/1.00	0.19/0.79/1.00
True negative rate	0.17/0.66/1.00	0.17/0.71/1.00	0.33/0.79/1.00	0.00/0.48/1.00
Accuracy	0.79/0.91/0.96	0.71/0.90/1.00	0.90/0.95/1.00	0.33/0.79/0.94
F1-score	0.78/0.92/0.98	0.73/0.92/1.00	0.90/0.96/1.00	0.22/0.79/0.96
RMSE	0.17/0.75/1.18	0.10/0.43/1.08	0.10/0.79/1.49	0.32/1.95/3.02
RMAE	0.11/0.29/0.43	0.08/0.21/0.42	0.08/0.47/1.49	0.14/1.03/2.18

provide more insights into the quality of predictions.

The average speedup across all programs is 2.402 with the standard deviation of a prediction made by the best kernel equal to 0.116.

Due to an additional parameter used to train the regression requires approximately 25x more time compared to classification. This affected the linear and the polynomial kernel. However, the effect on the linear kernel is much smaller due to its smaller search space of 225 variants compared to 4050 of the polynomial kernel. As a consequence, the polynomial kernel requires a significant amount of time to be trained.

Information Gain

Table 7.10 compares quality of predictions of various kernels used for regression from signature scenarios identified by the information gain. Due to information gain's deterministic nature, the number of scenarios and scenarios that comprise the signature is equal for all kernels.

The best prediction quality is made with the RBF kernel for almost all presented metrics used by the regression-based classification. However, the regression capability of the RBF kernel expressed by RMSE and RMEA metrics is between only 0.10 to significant 1.49 with the average value of 0.79 and between only 0.08 to very significant 1.49 with the average value of 0.47, respectively. Contrary, the polynomial kernel has not that good performance as a classifier, but its RMSE and RMAE are the best among kernels. RMSE and RMAE metrics are in the range between 0.10 and 1.08 with the average value of 0.43, and between insignificant 0.08 and quite low 0.42 with the average value of 0.21, respectively. Again, the worst prediction quality for all presented metrics comes from the sigmoid kernel.

Table 7.11: Comparison of SVM kernels used for regression with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	8/9.67/11	9/11/14	8/10/13	7/9.33/12
Precision	0.50/0.85/1.00	0.69/0.92/1.00	0.69/0.91/1.00	0.50/0.79/1.00
Recall	0.92/0.97/1.00	0.77/0.94/1.00	0.77/0.95/1.00	0.42/0.87/1.00
True negative rate	0.00/0.47/1.00	0.00/0.66/1.00	0.17/0.69/1.00	0.00/0.42/1.00
Accuracy	0.50/0.85/1.00	0.71/0.91/1.00	0.71/0.91/1.00	0.50/0.77/0.92
F1-score	0.67/0.90/1.00	0.73/0.92/1.00	0.73/0.93/1.00	0.47/0.81/0.94
RMSE	0.33/0.61/0.88	0.15/0.37/0.64	0.22/0.73/1.40	0.58/1.87/3.02
RMAE	0.19/0.29/0.44	0.10/0.23/0.41	0.12/0.30/0.52	0.20/0.96/2.18

K-medoids

Table 7.11 compares quality of predictions of various kernels used for regression from signature scenarios identified by the k-medoids. Due to k-medoid's non-deterministic nature, the number of scenarios that comprise the signature varies for all kernels.

The best prediction quality is made with the RBF kernel for almost all presented metrics used by the regression-based classification. The RBF kernel has RMSE of between only 0.22 to significant 1.40. Contrary, the polynomial kernel has not that good performance as a classifier, but its RMSE and RMAE are the best among kernels and range between 0.15 and 0.64 with the average value of 0.37, and between 0.10 and 0.41 with the average value of 0.23, respectively. Again, the worst prediction quality for all presented metrics comes from the sigmoid kernel.

Again, the performance of true negative rate can be very bad and the resulting true negative rate is 0% for all except RBF. Thus, it always predicts to tune and cannot identify those scenarios where its not worth tuning. Even the best performing RBF had the true negative rate of only 17% or only one of 6 benchmarks is correctly predicted.

Comparison of Information Gain and K-medoids

Table 7.12 compares information gain and k-medoids methods for predictor configured for regression. The values present the relative difference between the information gain and k-medoids average values for the speedup calculated relative to scenarios 24, 96, and 6 according to Table 7.6. It can be seen that on average k-medoids is inferior to information gain in most measures for regression as well as regression-based classification. The information gain method provided signature scenarios which contain fewer features (scenarios) compared to the k-medoids method. From fewer signature scenarios, information

Table 7.12: Comparison of information gain and k-medoids for various SVM kernels used for regression with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with two thresholds (6/46, 26/26).

	Kernel			
Measure	Linear	RBF	Sigmoid	Average
Signature size	0.96734	0.927078	0.986905	0.960441
Precision	1.131891	1.069068	1.011938	1.070966
Recall	0.947882	1.011667	0.870354	0.943301
True negative rate	0.930555	1.325294	0.611111	0.955653
Accuracy	1.103784	1.048861	1.028777	1.060474
F1-score	1.034699	1.039716	0.937387	1.003934
RMSE	1.365536	1.022433	0.952970	1.113647
RMAE	1.054681	1.293667	1.041130	1.129826

gain makes better predictions than k-medoids. This is especially true for the minimum expected prediction quality which on average contains 0.42 fewer scenarios while having on average 0.39% higher F1-score, 6.05% accuracy, and 7.10% precision while having 5.67% lower recall, and 4.43% true negative rate. However, they also had 11.36% higher RMSE and 12.98% RMAE which means predictions made for regression with Information Gain determined signature scenarios were on average up to 11.36% worse than those of signature scenarios determined by k-medoids.

What is interesting to notice is that good selection of much smaller signature can bring much better prediction quality as it can be seen for predictions made by the information gain method by the information gain method where the speedup is calculated relative to scenario 24 with the threshold set to 1.01181 or 1.18% improvement.

Comparison of Classification and Regression-based Classification

Table 7.13 compares classification and regression-based classification predictor. The values present the relative difference between the classifier and regression-based classifier average values for the speedup calculated relative to scenarios 24, 96, and 6 according to Table 7.6. It can be seen that on average regression-based classifier used 1.14% fewer signature scenarios while achieving on average 4.05% higher F1-score, 2.09% lower accuracy, 23.67% higher true negative rate, 9.62% higher recall, 0.73% lower precision. Therefore, it seems that the regression-based classification can make slightly better classification predictions.

7.2.6 User's Time Saved by Tuning Runs for Meta-plugin

Tuning with the meta-plugin eliminated many applications whose speedup would not satisfy the user and therefore would not justify spent time. Therefore, this section provides

Table 7.13: Comparison of classification and regression-based classification for various SVM kernels with 500 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with thresholds (6/46, 26/26).

	Kernel			
Measure	Linear	RBF	Sigmoid	Average
Signature size	1.015939	1.024724	0.916767	0.985810
Precision	0.910036	0.989030	1.079064	0.992710
Recall	1.062574	1.029201	1.196832	1.096202
True negative rate	1.071057	0.959174	1.679885	1.236705
Accuracy	0.919105	1.005044	1.013215	0.979122
F1-score	0.979916	1.009363	1.132227	1.040502

the information about time saved from not tuning applications with insignificant tuning potential, but also of the time wasted by tuning such applications. Table 7.14 provides information about different settings used by the meta-plugin and the resulting saved time, wasted time and time necessary to tune the application which is wrongly predicted to have insignificant tuning potential. Please note that the exact time saved by the predictor is higher as time presented in this table does not include overheads introduced by PTF like fetching data from the historical performance database, compiling the application, and restarting the application after recompilation. Data presented by the table includes only the time required to execute the phase regions of programs. In the table, we see the best performing kernel per-setting and per-threshold used to predict the tuning potential.

$$t_{saved} = tt_{TN} - ts_{TN} \tag{7.8}$$

$$t_{wasted} = tt_{FP} + ts_{FN} \tag{7.9}$$

$$t_{missed} = tt_{FN} \tag{7.10}$$

Total time tt and time to evaluate signature ts in equations index, specifies correctness of the predictor where TN, TP, FN, FP stands for true negative, true positive, false negative, and false positive, respectively.

Saved time presented in Equation 7.11 is calculated as the time that would be necessary to tune applications with insufficient tuning potential tt_{TN} minus the time necessary to evaluate signature scenarios and determine the signature for those applications ts_{TN}. Wasted time presented in Equation 7.12 is time used to tune applications with insufficient tuning potential due to the predictor's misprediction, tt_{FP}, minus the time necessary to evaluate signature scenarios and determine the signature for those applications, ts_{FN}. The missed opportunity presented in Equation 7.13 is the time that would be used by

Table 7.14: Settings used by the meta-plugin, the best kernel, saved time, wasted time, wrongly and correctly predicted time. Total time used to execute all scenarios with all programs takes 759.36 h.

Scenario	Threshold	Kernel	Signature	Mode	Saved t.	Wasted t.	Missed opp.
24	1.01181	Poly	Info Gain	Cl.	211.06184	14.65771	0.48768
24	1.07915	RBF	Info Gain	Reg.	515.23650	3.65527	3.20386
96	1.02140	RBF	Info Gain	Reg.	69.05517	30.28942	0.00000
96	1.26419	RBF	k-medoids	Reg.	398.99743	0.00003	0.00177
6	1.23941	RBF	Info Gain	Reg.	75.70159	0.00000	0.00000
6	3.33472	RBF	Info Gain	Reg.	520.24470	0.33227	12.39625

Table 7.15: MPI parameters and their respective values used for evaluation

Parameter	Values
eager_limit	256, 512, 1k, 2k, 4k, 8k, 16k, 32k, 64k
buffer_mem	1M, 2M, 4M, 8M, 16M, 32M, 64M, 128M, 256M
use_bulk_xfer	yes, no
bulk_min_msg_size	4K, 8K, 16K, 32K, ..., 256M, 512M, 1G, 2G
wait_mode	nopoll, poll, sleep, yield
css_interrupt	yes, no
task_affinity	CORE, MCM
pe_affinity	yes, no
polling_interval	100000, 200000, 300000, ..., 800000, 900000, 1000000
cc_scratch_buf	yes, no

the plugin to tune the application if the predictor did not mispredict insufficient tuning potential tt_{FN}.

The presented results show that significant amount of time is saved, which ranges from more than 69 hours up to 520 hours which account for between 9.09% to 68.51% of the time.

7.2.7 MPI Parameters Plugin

The selection of MPI Parameters is done based on documentation of IBM MPI [50].

eager_limit: Specifies the upper boundary for which the eager protocol is used instead of the rendezvous protocol. The parameter range is from 0 to 64 KiB (default 64 KiB).

buffer_mem: Specifies the buffer size used for eager protocol by each process. The range of the parameter is from 0 to 256 MiB (default depends on the value of eager limit).

use_bulk_xfer: Specifies that portions of the user's virtual address space can be mapped to a communication adapter. Then the communication protocol can use Remote Direct Memory Access (RDMA) to copy data from the send buffer to the receive buffer as part of the MPI receive. The parameter can be set to yes or no (default yes).

bulk_min_msg_size: Determines the minimum message size to be used with bulk transfer mode instead of packet mode. The parameter range is from 4 KiB to 2 GiB (default value 64 KiB).

wait_mode: Specifies how a thread or task behaves when it discovered it is blocked, waiting for a message to arrive. The parameter can be set to poll or nopoll (default nopoll).

css_interrupt: Specifies whether or not arriving packets generate interrupts. Certain applications may achieve better performance if packets generate interrupts. The parameter can be set to yes or no (default no).

task_affinity: Specifies how tasks are attached to system cpusets. Possible values are core - single core, core:N - several cores, CPU - a whole processor, CPU:n - several processors, MCM - a socket in a node in the round-robin fashion (default CORE).

pe_affinity: Specifies whether Load Leveler or the MPI environment determine the task scheduling. The parameter can be set to yes or no (default yes).

polling_interval: Specifies a fixed period of time to interrupt the nopoll wait. The parameter range is from 0 to 2 billion μs (default 400000).

cc_scratch_buf: Use the fastest collective communication algorithm even if that algorithm requires allocation of more scratch buffer space. The parameter can be set to yes or no (default yes).

Measurements

Figures 7.10, 7.11, and 7.12 show the sample space's distribution of speedups or slowdowns for each program across the MPI parameters configurations. Each program is represented by the box and whisker plot which summarize the distribution of speedups for that program. The box is represented by the 1st quartile (25th percentile) and 3rd quartile (75th percentile). The line inside of the box represents the median speedup. While the whiskers represent the extremes.

Classification

In this section, we present the prediction quality obtained for two methods used to determine the signature scenarios and four supported SVM kernels. To give a better view on

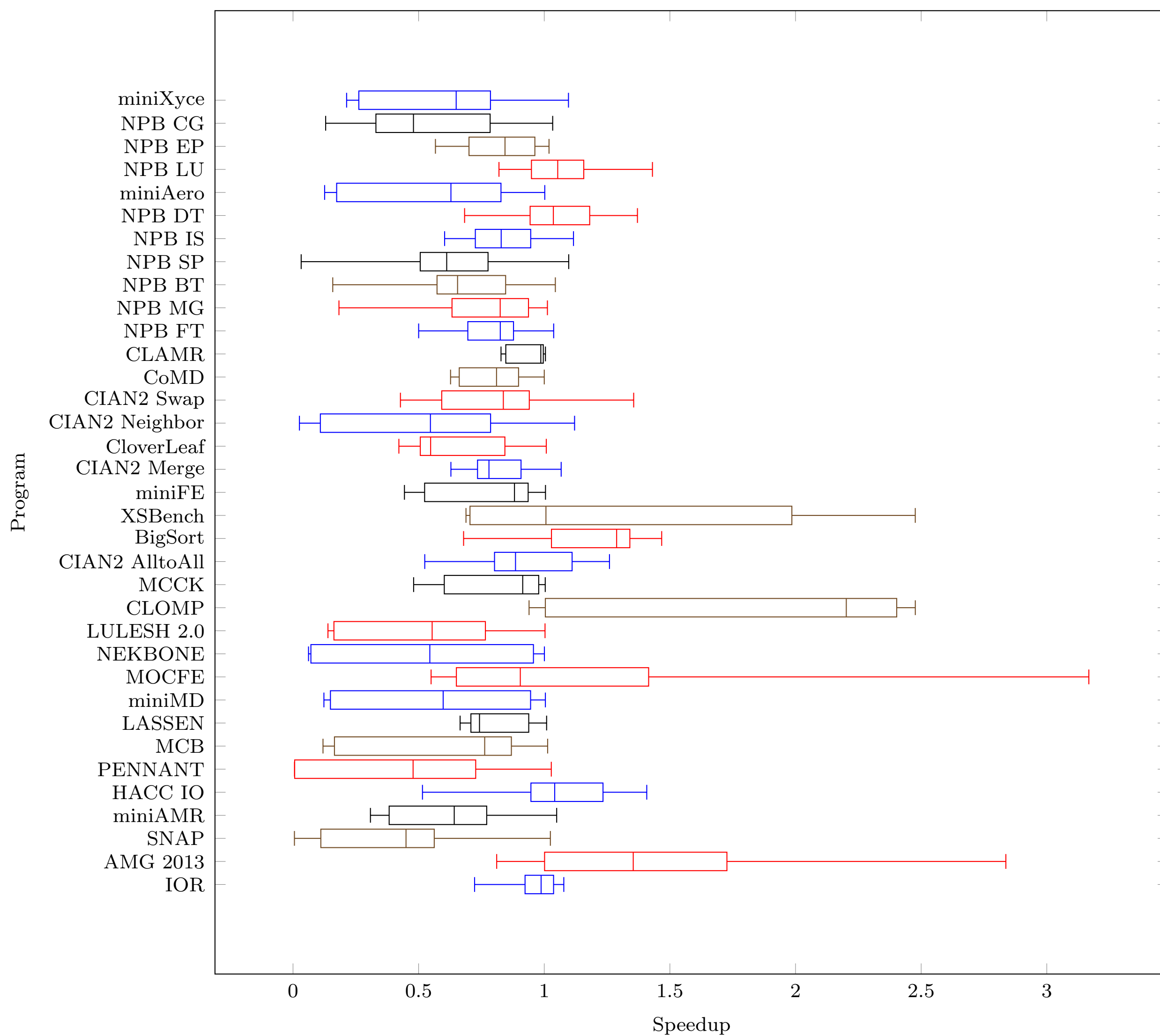


Figure 7.10: The execution time of various benchmarks with MPI Parameters plugin.

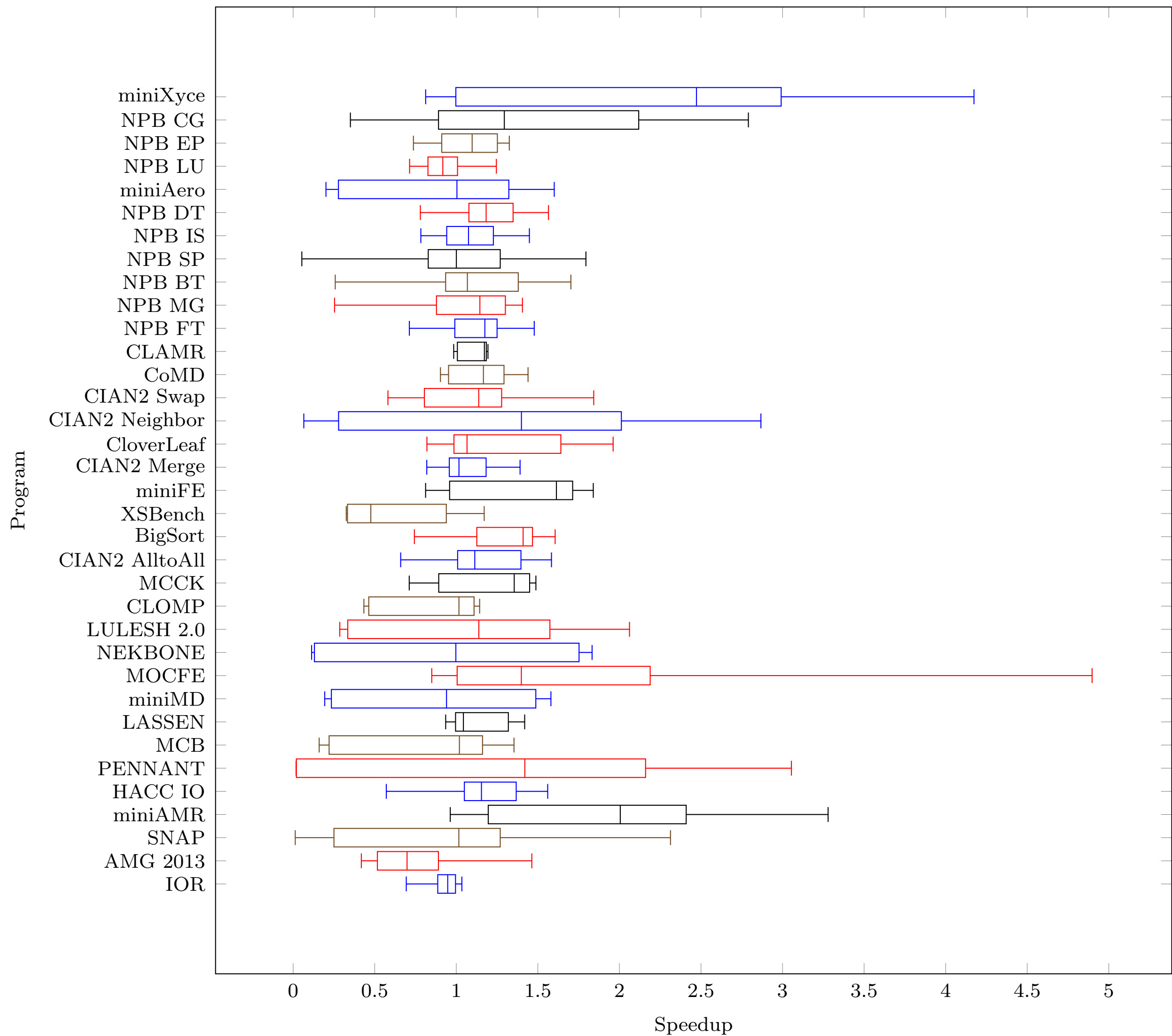


Figure 7.11: The execution time of various benchmarks with MPI Parameters plugin.

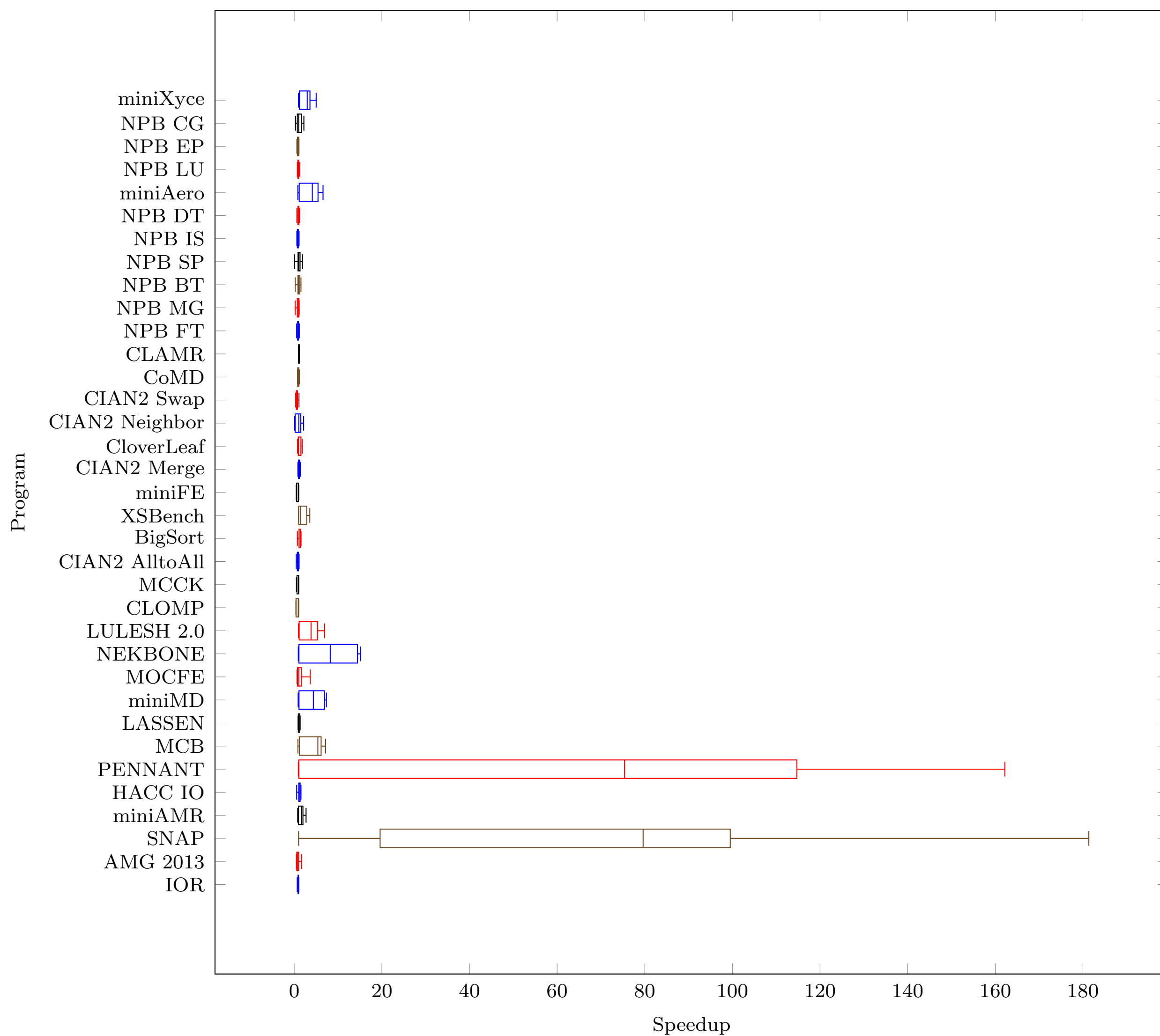


Figure 7.12: The execution time of various benchmarks with MPI Parameters plugin.

Table 7.16: The speedup achieved with Compiler Flags Selection plugin on dataset relative to scenario 7.

Outcome		Signature					
Real	Pred	7	221	116	155	220	105
1.03	1.21	1.00	0.90	0.95	0.94	0.88	1.00
1.46	0.80	1.00	0.52	0.52	0.52	0.52	0.52
2.31	2.21	1.00	1.78	2.26	2.17	1.84	1.81
3.28	3.10	1.00	3.11	3.11	3.22	3.26	3.28
1.56	1.32	1.00	0.62	0.91	1.17	0.66	1.26
3.05	2.98	1.00	3.03	3.02	3.00	3.04	3.00
1.35	1.40	1.00	1.12	1.35	1.33	1.14	1.34
1.42	1.57	1.00	1.38	1.38	1.39	1.39	1.38
1.58	1.71	1.00	1.58	1.49	1.55	1.54	1.55
4.90	4.95	1.00	1.54	4.89	4.88	1.54	4.89
1.83	1.96	1.00	1.81	1.82	1.82	1.81	1.79
2.06	2.07	1.00	2.04	2.05	1.76	1.77	1.76
1.14	0.74	1.00	0.46	0.46	0.46	0.46	0.43
1.49	1.65	1.00	1.49	1.48	1.46	1.46	1.46
1.58	1.41	1.00	1.26	1.25	1.14	1.14	1.14
1.61	1.68	1.00	1.55	1.51	1.49	1.53	1.49
1.17	0.77	1.00	0.48	0.47	0.47	0.48	0.47
1.84	1.83	1.00	1.78	1.69	1.68	1.74	1.67
1.39	1.50	1.00	1.39	1.29	1.27	1.35	1.27
1.96	2.06	1.00	1.94	1.94	1.89	1.92	1.93
2.87	2.63	1.00	2.81	2.75	2.53	2.53	2.41
1.84	1.48	1.00	1.30	1.30	1.24	1.23	1.23
1.44	1.54	1.00	1.43	1.36	1.35	1.28	1.30
1.19	1.33	1.00	1.18	1.19	1.19	1.18	1.19
1.48	1.45	1.00	1.31	1.27	1.20	1.24	1.20
1.41	1.54	1.00	1.33	1.35	1.32	1.31	1.31
1.70	1.70	1.00	1.58	1.52	1.57	1.53	1.53
1.79	1.81	1.00	1.63	1.69	1.60	1.58	1.56
1.45	1.34	1.00	1.23	1.15	1.05	1.09	1.05
1.57	1.34	1.00	1.12	1.10	1.33	1.40	1.40
1.60	1.75	1.00	1.60	1.59	1.54	1.54	1.54
1.58	1.38	1.00	1.53	1.23	1.20	1.52	1.20
1.33	1.43	1.00	1.30	1.30	1.30	1.30	0.90
2.79	2.65	1.00	2.67	2.76	2.44	2.42	2.43
4.17	4.00	1.00	4.09	4.15	3.99	4.13	4.17

Table 7.17: Settings used by the meta-plugin for MPI Parameters plugin and the distribution of expected meta-plugin's decisions (don't tune, tune) for benchmarks.

Scenario	Threshold	Don't tune	Tune
81	1.035510	17	18
81	1.363345	28	7
7	1.577600	17	18
7	2.187495	28	7
16	1.541965	17	18
16	5.792625	28	7

the predictions quality, we have calculated the speedup relative to three different scenarios; i.e., 81, 7, and 16; and different thresholds. The first threshold setting is determined to split benchmarks into 17 benchmarks with significant tuning potential and 18 with insignificant potential. The second threshold setting is determined to split benchmarks into 28 benchmarks with significant tuning potential and 7 with insignificant. Table 7.6 shows all the settings used by the meta-plugin for MPI Parameters plugin, settings differ depending on the scenario used to calculate the speedup, presented in Figures 7.10, 7.11, and 7.12.

The quality of classification is presented in tables for both signature scenarios identification methods, information gain and k-medoids. The optimal kernel is selected by the meta-plugin based on the F1-score metric presented in Section 7.2.4 in Equation 7.7. For each kernel's results presented in next two sections, we have compared minimum, average, and maximum of precision, recall, true negative rate, accuracy and F1-score metrics. These metrics are presented in this chapter for completeness and to provide more insights into the quality of predictions.

Information Gain

Table 7.18 compares quality of predictions of various kernels used for classification from signature scenarios identified by information gain. Due to information gain's deterministic nature, the number of scenarios and scenarios that comprise the signature is equal for all kernels.

Besides the prediction quality, we have also compared the mean training time on the per-program basis. As our training set consists of 35 programs, the complete training time is divided by 35. The linear kernel has reasonable training time of under 1 minute and F1-score between 57.14% and 100% for each program, with the average value of 88.19%. The polynomial kernel has very big variations of training time from under 1 minute to an hour which should be taken into account when selecting it. Its F1-score is between 57.14% and 100%, with the average value of 89.67%. RBF has reasonable training time of under 1 minute and F1-score between 72.73% and 100%, with the average value of

Table 7.18: Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with information gain. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	5/11/17	5/11/17	5/11/17	5/11/17
Precision	0.56/0.91/1.00	0.56/0.92/1.00	0.75/0.94/1.00	0.51/0.85/1.00
Recall	0.59/0.86/1.00	0.59/0.88/1.00	0.71/0.90/1.00	0.76/0.90/1.00
True negative rate	0.56/0.91/1.00	0.56/0.92/1.00	0.78/0.94/1.00	0.00/0.75/1.00
Accuracy	0.57/0.89/1.00	0.57/0.90/1.00	0.74/0.93/1.00	0.51/0.84/1.00
F1-score	0.57/0.88/1.00	0.57/0.90/1.00	0.73/0.92/1.00	0.68/0.86/1.00

91.18%. Sigmoid has reasonable training time of under 1 minute and F1-score between 67.92% and 100%, with the average value of 86.09%. However, please note that for this kernel; Scenario 16, Threshold 5.79; the F1-score could not be calculated with the last setting due to non-existing precision (nan) as the kernel failed to predict true positives. The values were therefore calculated without the last setting.

From the information provided in the table, we can conclude that the best prediction quality is achieved with the RBF kernel. This kernel had 2.14% higher average value with 15.58% higher lowest value combined with a very short training time, usually less than a minute per program. The second best is the polynomial kernel with 1.48% higher average value compared to the linear kernel and the same minimum and maximum values. However, the polynomial kernel seems impractical due to its unpredictable long training time which, as a consequence of numerical instability, ranged from less than a minute up to an hour, 36 hours in total. The linear kernel has reasonable prediction quality combined slightly worse than the polynomial kernel. The sigmoid kernel is the worst performing due to the fact that it could not calculate F1-score and it seems to be problematic with high variations of the predicting value. This happens with for both the MPI parameters plugin and the CFS plugin. Speedups calculated relative to scenario 16 of MPI Parameters plugin and scenario 6 of CFS plugin with the threshold which separates classes in approximately half, 26/26 and 17/18. However, without such behavior the would provide reasonable performance with on average 2.10% lower F1-score than the linear kernel and 4.80% higher minimum F1-score than the best performing RBF. Interesting, again the true negative rate is high compared to CFS plugin which is for some threshold/scenario pairs as low as 0%. This might be explained by the fact that CFS plugin had very low threshold as there were only 6 applications with insignificant tuning potential. For MPI Parameters plugin the threshold resulted in a different split among classes. As part of the future work, this discrepancy in performance between CFS and MPI Parameters plugin should be investigated.

K-medoids

Table 7.19 compares quality of predictions of various kernels used for classification from signature scenarios identified by k-medoids. Due to k-medoid's non-deterministic nature, the number of scenarios that comprise the signature varies for all kernels.

From the information provided in the table, the linear kernel provides the best prediction quality (F1-score). The linear kernel's F1-score ranges between 66.67% to 100%, with the average value of 87.62%. The second best is the RBF kernel with F1-score which ranges between 72.73% and 100%, while the average value is 87.13%. This kernel provides the highest minimum F1-score, 3.76% higher than the polynomial kernel. The polynomial kernel provided slightly worse prediction quality, with F1-score ranging between 68.97% and 100%, on average 86.12%. Similar to other training done with CFS plugins, the polynomial kernel suffers from very long training time, as a consequence of numerical instability, which ranges between a minute and more than an hour per program. This limits its practical usability. The sigmoid kernel provided the worst prediction quality, which ranges between 67.92% and 92.31%, on average 80.54%. However, please note that for this kernel; Scenario 16, Threshold 5.79; the F1-score could not be calculated with the last setting due to non-existing precision (nan) as the kernel failed to predict true positives. The values were therefore calculated without the last setting.

Similar to CFS plugin, the RBF has the best performance and the sigmoid kernel again suffers from the bad prediction quality. The bad prediction quality of the sigmoid kernel happens when speedups are calculated relative to scenario 16 of MPI Parameters plugin and scenario 6 of CFS plugin with the threshold which separates classes in approximately half, 26/26 and 17/18. Speedups calculated for programs relative to those scenarios in both cases results in the highest standard deviation which seems to be problematic for the sigmoid kernel. Interesting, the true negative rate of the sigmoid kernel is again high compared to CFS plugin which is for some threshold/scenario pairs as low as 0%. This might be explained by the fact that CFS plugin had very low threshold as there were only 6 applications with insignificant tuning potential. For MPI Parameters plugin the threshold resulted in a different split among classes. As part of the future work, this discrepancy in performance between CFS and MPI Parameters plugin should be investigated.

Comparison of Information Gain and K-medoids

Table 7.22 compares information gain and k-medoids methods for a predictor configured as classifier. The values present the relative difference between the information gain and k-medoids average values for the speedup calculated relative to scenarios 81, 7, and 16 according to Table 7.17. It can be seen that on average k-medoids is inferior to information gain in every measure for classification.

The information gain method provided signature scenarios which contain 0.33 (2.89%) fewer features (scenarios) compared to the k-medoids method. From those signature

Table 7.19: Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	8/11/14	8/11.67/14	9/11.5/14	8/11.17/13
Precision	0.90/0.97/1.00	0.83/0.96/1.00	0.72/0.94/1.00	0.51/0.69/1.00
Recall	0.53/0.81/1.00	0.59/0.79/1.00	0.57/0.82/1.00	0.71/0.82/1.00
True negative rate	0.94/0.98/1.00	0.89/0.97/1.00	0.72/0.94/1.00	0.00/0.73/1.00
Accuracy	0.74/0.91/1.00	0.74/0.90/1.00	0.74/0.90/1.00	0.51/0.81/1.00
F1-score	0.67/0.88/1.00	0.69/0.86/1.00	0.73/0.87/1.00	0.68/0.81/0.92

Table 7.20: Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum.

	Kernel				
Measure	Linear	Poly	RBF	Sigmoid	Average
Signature size	1.000000	0.942857	0.956522	0.985075	0.971113
Precision	0.931583	0.986197	1.007526	1.052509	0.994454
Recall	1.064429	1.144455	1.102670	1.110758	1.105578
True negative rate	0.923931	0.980942	1.009232	1.047652	0.990439
Accuracy	0.979058	1.037037	1.036842	1.051136	1.026018
F1-score	1.006380	1.071400	1.065420	1.090540	1.058436

scenarios, k-medoids makes better predictions than information gain. This is especially true for the minimum expected prediction quality which on average has 5.84% higher F1-score, 2.6% higher accuracy, 10.56% higher recall, 0.96% lower true negative rate and 0.55% lower precision. This exactly the opposite of what happened with the CFS plugin. It might be worth investigating why this happens.

Regression

In this section, we present the regression prediction quality obtained for two methods used to determine the signature scenarios and four supported SVM kernels. To give a better view on the predictions quality, we have calculated the speedup relative to three different scenarios; i.e., 81, 7, and 16; and different thresholds. The first threshold setting is determined to split benchmarks into 17 benchmarks with significant tuning potential and 18 with insignificant potential. The second threshold setting is determined to split benchmarks into 28 benchmarks with significant tuning potential and 7 with insignificant. Table 7.17 shows all the settings used by the meta-plugin for MPI Parameters plugin,

settings differ depending on the scenario used to calculate the speedup, presented in Figures 7.10, 7.11, and 7.12.

The quality of regression is presented in tables for both signature scenarios identification methods, information gain and k-medoids. The optimal kernel is selected by the meta-plugin based on the RMSE metric presented in Section 7.2.4 in Equation 7.7. For each kernel's results presented in next two sections, we have compared minimum, average and maximum of precision, recall, true negative rate, accuracy, F1-score, RMSE, and RMAE metrics. These metrics are presented in this chapter for completeness and to provide more insights into the quality of predictions.

The average speedup across all programs is 5.196 with the standard deviation of a prediction made by the best kernel equal to 0.203.

Due to an additional parameter used to train the regression requires approximately 25x more time compared to classification. This affected the linear and the polynomial kernel. However, the effect on the linear kernel is much smaller due to its smaller search space of 225 variants compared to 4050 of the polynomial kernel. As a consequence, the polynomial kernel requires a significant amount of time to be trained.

Information Gain

Table 7.21 compares quality of predictions of various kernels used for Regression from signature scenarios identified by the information gain. Due to information gain's deterministic nature, the number of scenarios and scenarios that comprise the signature is equal for all kernels.

Using the linear kernel is recommended for the regression, however, its training time can be quite long. Also, its regression-based classification performance is not the best one but still not very bad. Using the RBF kernel is highly recommended for the regression-based classification because of its performance and short training time. On the other hand, using this kernel for regression is not recommended because of its manyfold worse performance compared to linear or polynomial kernels.

K-medoids

Table 7.22 compares quality of predictions of various kernels used for Regression from signature scenarios identified by the k-medoids. Due to k-medoid's non-deterministic nature, the number of scenarios that comprise the signature varies for all kernels.

The best prediction quality is made with the polynomial kernel for all presented metrics used by the regression-based classification. Its F1-score is between 80% and 100%, with the average of 96.19%. Its regression performance is the second best with RMSE and RMAE; between 0.26 and 2.29, with the average value of 0.87; and between 0.17 and 0.79, with the average of 0.37; respectively. However, its practical usability is limited

Table 7.21: Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with information gain. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	5/11/17	5/11/17	5/11/17	5/11.2/17
Precision	0.63/0.91/1.00	0.76/0.96/1.00	0.79/0.96/1.00	0.53/0.80/1.00
Recall	0.88/0.95/1.00	0.76/0.93/1.00	0.88/0.95/1.00	0.88/0.95/1.00
True negative rate	0.50/0.90/1.00	0.78/0.96/1.00	0.78/0.95/1.00	0.06/0.68/1.00
Accuracy	0.69/0.92/1.00	0.77/0.95/1.00	0.83/0.95/1.00	0.54/0.82/1.00
F1-score	0.73/0.93/1.00	0.76/0.95/1.00	0.83/0.95/1.00	0.69/0.86/1.00
RMSE	0.20/0.46/1.11	0.15/0.40/1.00	0.15/10.6/31.9	0.33/8.43/40.7
RMAE	0.15/0.24/0.45	0.12/0.21/0.42	0.1/3.25/10.01	0.20/2.43/11.2

by its extremely long training time. The linear kernel has the best regression prediction quality with RMSE and RMAE; between 0.26 and 2.03, with the average value of 0.82; and RMAE between 0.21 and 0.68, with the average value of 0.36. Its regression-based classification performance is not among the best ones but still decent and ranges between 75.56% and 100%, with the average of 90.48%. Also, this kernel suffers from long training time of few minutes per program. On the other hand, the RBF kernel is the best one with respect to the regression-based classification performance with values between 75.56% and 100%, with the average value of 92.23%. However, its regression performance is very poor due to the case where the speedup is relative to scenario 16. RMSE ranges from very low 0.22 to very bad 31.82, with the average value of 10.75. RMAE is the lowest 0.12 and very high 31.82. It remains to be seen why the case with big variations had a bad effect on RBF while devastating on the performance of the sigmoid kernel. In this case, the sigmoid kernel could not even classify a single class 1 sample. The quality of regression-based classification was also affected by this problem. Please note that for this kernel; Scenario 16, Threshold 5.79; the F1-score could not be calculated with the last setting due to non-existing precision (nan) as the kernel failed to predict true positives. The values were therefore calculated without the last setting.

Using the linear kernel is recommended for the regression, however, its training time can be quite long. Also, its regression-based classification performance is not the best one but still not very bad. Using the RBF kernel is highly recommended for the regression-based classification because of its performance and short training time. On the other hand, using this kernel for regression is not recommended because of its manyfold worse performance compared to linear or polynomial kernels.

Table 7.22: Comparison of SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 81, 7, and 16 with various thresholds. Signature determined with k-medoids. Values represent minimum/average/maximum.

	Kernel			
Measure	Linear	Poly	RBF	Sigmoid
Signature size	8/10.5/13	9/11.83/13	8/11/14	9/11/13
Precision	0.61/0.88/1.00	0.70/0.94/1.00	0.61/0.92/1.00	0.51/0.79/1.00
Recall	0.83/0.95/1.00	0.89/0.96/1.00	0.86/0.94/1.00	0.83/0.94/1.00
True negative rate	0.39/0.87/1.00	0.61/0.93/1.00	0.39/0.89/1.00	0.00/0.64/1.00
Accuracy	0.69/0.90/1.00	0.77/0.94/1.00	0.69/0.92/1.00	0.51/0.80/1.00
F1-score	0.76/0.90/1.00	0.80/0.95/1.00	0.76/0.92/1.00	0.68/0.84/1.00
RMSE	0.26/0.82/2.03	0.21/0.83/2.29	0.22/10.8/31.8	0.32/6.67/40.7
RMAE	0.21/0.36/0.68	0.17/0.35/0.79	0.12/3.34/10.0	0.19/2.45/11.2

Comparison of Information Gain and K-medoids

Table 7.22 compares information gain and k-medoids methods for a predictor configured as classifier. The values present the relative difference between the information gain and k-medoids average values for the speedup calculated relative to scenarios 81, 7, and 16 according to Table 7.17. It can be seen that on average k-medoids is inferior to information gain in every measure for classification.

The information gain method provided signature scenarios which contain 0.11 (0.91%) fewer features (scenarios) compared to the k-medoids method. From those signature scenarios, k-medoids makes better somewhat predictions for regression-based classification than information gain. This is true for the expected prediction quality which on average has 1.61% higher F1-score, 1.72% higher accuracy, 4.90% higher true negative rate, 0.46% lower recall, and 2.37% higher precision. This exactly the opposite of what happened with the CFS plugin. It might be worth investigating why this happens. On the other hand, the performance of regression is much better with information gain than with k-medoids, RMSE for 18.51% while RMAE for 20.18%.

Comparison of Classification and Regression-based Classification

Table 7.24 compares classification and regression-based classification predictors. The values present the relative difference between the classifier and regression-based classifier average values where the speedup is calculated relative to three different scenarios according to Table 7.17. It can be seen that on average Regression-based Classifier used 3.75% fewer signature scenarios while achieving on average 4.21% higher F1-score, 0.95% higher accuracy, and 11.20% higher recall while Classifier achieved higher precision (2.83%) and true negative rate (5.60%). Therefore, it seems that the regression-based classification

Table 7.23: Comparison of information gain and k-medoids for various SVM kernels with 300 scenarios where the speedup is calculated relative to three different scenarios with two thresholds (17/18, 28/7).

	Kernel				
Measure	Linear	Poly	RBF	Sigmoid	Average
Signature size	1.047619	0.970588	1.000000	1.018182	1.009097
Precision	1.033016	1.012124	1.033451	1.016066	1.023664
Recall	0.998858	0.951138	1.014333	1.017219	0.995387
True negative rate	1.039549	1.029703	1.072961	1.053917	1.049033
Accuracy	1.021053	0.990050	1.036269	1.021429	1.017200
F1-score	1.022450	0.983692	1.033403	1.024879	1.016106
RMSE	0.555192	0.456660	0.981025	1.266547	0.814856
RMAE	0.677195	0.549062	0.972947	0.993647	0.798213

Table 7.24: Comparison of classification and regression-based classification for various SVM kernels with 300 scenarios where the speedup is calculated relative to scenarios 24, 96, and 6 with two thresholds (17/18, 28/7).

	Kernel				
Measure	Linear	Poly	RBF	Sigmoid	Average
Signature size	0.954545	0.971429	0.956522	0.967484	0.962495
Precision	0.901809	0.954545	0.971429	0.956522	0.971743
Recall	1.065646	0.901809	0.974383	0.974914	1.111985
True negative rate	0.888781	1.065646	1.203248	1.087089	0.944022
Accuracy	0.958871	0.888781	0.952645	0.940605	1.009510
F1-score	0.984287	0.958871	1.047460	1.000553	1.042124

can make slightly if not much better classification predictions.

7.2.8 User's Time Saved by Tuning Runs for Meta-plugin

Tuning with the meta-plugin eliminated many applications whose speedup would not satisfy the user and therefore would not justify spent time. Therefore, this section provides the information about time saved from not tuning applications with insignificant tuning potential, but also of the time wasted by tuning such applications. Table 7.25 provides information about different settings used by the meta-plugin and the resulting saved time, wasted time and time necessary to tune the application which is wrongly predicted to have insignificant tuning potential. Please note that the exact time saved by the predictor is higher as time presented in this table does not include overheads introduced by PTF like fetching data from the historical performance database, compiling the application, and restarting the application after recompilation. Data presented by the table includes only

Table 7.25: Settings used by the meta-plugin, the best kernel, saved time, wasted time, wrongly and correctly predicted time. Total time used to execute all scenarios with all programs takes 332.39 h.

Scenario	Threshold	Kernel	Signature	Mode	Saved t.	Wasted t.	Missed opp.
81	1.035510	Sigmoid	Info Gain	Cl.	151.3885	28.01162	0.02168
81	1.363345	Linear	Info Gain	Cl.	217.29243	0	0
7	1.577600	RBF	Info Gain	Reg.	72.62797	0	0.78798
7	2.187495	RBF	Info Gain	Reg.	191.3994	0	0
16	1.541965	Linear	Info Gain	Reg.	77.99707	0	0.35787
16	5.792625	Linear	Info Gain	Reg.	181.45791	0	0

the time required to execute the phase regions of programs. In the table, we see the best performing kernel per-setting and per-threshold used to predict the tuning potential.

$$t_{saved} = tt_{TN} - ts_{TN} \tag{7.11}$$

$$t_{wasted} = tt_{FP} + ts_{FN} \tag{7.12}$$

$$t_{missed} = tt_{FN} \tag{7.13}$$

Total time tt and time to evaluate signature ts in equations index, specifies correctness of the predictor where TN, TP, FN, FP stands for true negative, true positive, false negative, and false positive, respectively.

Saved time presented in Equation 7.11 is calculated as the time that would be necessary to tune applications with insufficient tuning potential tt_{TN} minus the time necessary to evaluate signature scenarios and determine the signature for those applications ts_{TN}. Wasted time presented in Equation 7.12 is time used to tune applications with insufficient tuning potential due to the predictor's misprediction, tt_{FP}, minus the time necessary to evaluate signature scenarios and determine the signature for those applications, ts_{FN}. The missed opportunity presented in Equation 7.13 is the time that would be used by the plugin to tune the application if the predictor did not mispredict insufficient tuning potential tt_{FN}.

The presented results in Table 7.25 show that significant amount of time is saved, which ranges from more than 72 hours up to 217 hours which account for between 21.85% to 65.37% of the time.

Chapter 8

Summary and Outlook

This thesis has presented different techniques that focus on more efficient tuning process, which requires fewer interventions from the user and results in lower overhead. Furthermore, we have developed a set of plugins to support user's tuning effort. The plugins target the compilation process and user-definable tuning of heterogeneous/manycore systems. The first one, Compiler Flags Selection of OpenCL extends the Compiler Flags Selection plugin with support to tune OpenCL kernels with the help of compiler flags available in the OpenCL offline compiler of Intel and the online compiler of nVidia. The CFS for OpenCL plugin was tested on Intel Xeon Phi 5100P and E5-2650 with three applications from Rodinia benchmark suite, LavaMD, HotSpot, and Pathfinder, respectively. Tuning was done with a small set of optimization flags with three different datasets. The achieved speedup for the CPU was up to 56% while Xeon Phi achieved up to 55% between the slowest and the fastest performing kernel.

The second one, User Defined Parameters allows the user to specify tuning parameters and the search algorithm. Once the user specifies necessary parameters the plugin automatically finds the optimal combination of parameters and their values. Although the plugin itself provides no codified expert knowledge, it allows the user to explore multiple tuning aspects before implementing a new tuning plugin.

The most important contribution and the central part of this thesis are meta-plugins. Meta-plugins combine several of PTF's tuning plugins into a single plugin making the tuning process more efficient with fewer interventions from the user. From PTF perspective they act as an intermediate between the frontend and component plugins. The meta-plugin loads component plugins, decide upon which plugins to run, in which order and drives their execution. Three meta-plugins were developed in the course of this thesis, Fixed Sequence, Adaptive Sequence with Analytical Model, Adaptive Sequence with Predictive Model. The Fixed Sequence plugin is the simplest form of meta-plugins which only executes plugins one after another.

On the other hand, we have adaptive meta-plugins which intelligently decide about which

Table 8.1: Comparison of Classification and Regression-based Classification performance for both tuning plugins.

	CFS		MPIParams	
	Classification	Regression	Classification	Regression
Measure	RBF	RBF	RBF	RBF
Signature size	4/9.5/14	4/5/6	5/11/17	5/11/17
Precision	0.83/0.95/1.00	0.92/0.96/1.00	0.75/0.94/1.00	0.79/0.95/1.00
Recall	0.88/0.95/1.00	0.88/0.96/1.00	0.71/0.90/1.00	0.88/0.94/1.00
True negative rate	0.50/0.80/1.00	0.33/0.79/1.00	0.78/0.94/1.00	0.78/0.94/1.00
Accuracy	0.87/0.94/1.00	0.90/0.95/1.00	0.74/0.93/1.00	0.83/0.94/1.00
F1-score	0.87/0.95/1.00	0.90/0.96/1.00	0.73/0.92/1.00	0.83/0.94/1.00

plugins to use to tune the selected program. One of such meta-plugins is the Adaptive Sequence with Analytical Model plugin which decides plugin execution based on special Periscope analysis before tuning starts. The analysis strategy generates a property per plugin that is used as criteria to tune with that particular plugin. The user decides the threshold that severity of plugin's property has to reach for the plugin to be selected for tuning.

The Adaptive Sequence with Predictive Model plugin analyses the previous runs of the plugin on several different applications. Based on their success in tuning programs, the meta-plugin characterizes the search space of a plugin and can predict program's sensibility to the plugin. From the predicted sensibility of the plugin and the user preference, the meta-plugin can decide to tune the program with the plugin or not to. The characterization is based on a signature which is agnostic of tuning aspects and therefore universally applicable to any tuning plugin. The signature is extracted from the search space and consists of individual tuning scenarios. Such approach allows the user to concentrate tuning only on plugins which are predicted to provide significant improvement of the tuning objective. This reduces the tuning time and provides very good results. The evaluation of the meta-plugin was done with two plugins, the Compiler Flags Selection and the MPI Parameters, in two predictor modes, classification and regression. In total 4 SVM kernels were used in the predictor. Table 8.1 presents the best performing kernel for both predictors in the classification mode, the classifier and the regression-based classifier. Signature results presented in the table are determined by the information gain method because the best performing results were better than those determined by k-medoids. Predictions for both plugins show similar results with F-score between 72.73% and 100%, with the average value of 84.75%. The best kernel for both predictors is RBF kernel.

Table 8.2 presents the best performing kernels for both plugins in the regression mode with signature determined by information gain and k-medoids. The best performing kernels in the regression mode are linear and polynomial for both plugins. CFS plugin's linear kernel whose signature scenarios were identified by k-medoids has slightly better RMSE,

Table 8.2: Comparison of Regression performance for both tuning plugins.

	CFS		MPIParams	
	Information Gain	K-medoids	Information Gain	K-medoids
Measure	Poly	Poly	Poly	Linear
Signature size	4/9.5/14	9/11/14	5/11/17	5/11/17
RMSE	0.10/0.43/1.08	0.15/0.37/0.64	0.15/0.40/1.00	0.26/0.82/2.03
RMAE	0.08/0.21/0.42	0.10/0.23/0.41	0.12/0.21/0.42	0.21/0.36/0.68

and slightly worse RMAE. On the other hand, MPI Parameters plugin's polynomial kernel whose signature scenarios were identified by information gain has much better RMSE and RMAE compared to its k-medoids equivalent. The best performing kernel of CFS plugin has the average value of 2.402 and the average error of 0.44 or 18.23% which is still decent. Also, the best performing kernel of MPI Parameters plugin has the average value of 5.196 and the average error of 0.40 or 7.79% which is a reasonable error for such predictions.

The slight concern comes from the fact that training time of these two kernels is quite long and ranges from several minutes per program in the case of the linear kernel to more than an hour in the case of the polynomial kernel. This significantly reduces the usability of regression as the prediction method for the meta-plugin. Alternative kernels such as RBF and sigmoid suffer from very poor F1-score but comes with very quick training time, less than a minute per program.

The goal of this work was to develop a generic approach for any PTF tuning plugin/aspect for predicting its tuning potential. The developed method circumvents the identification of aspect specific program signatures. In general, identification of the signature components is a difficult and non-intuitive process. Many signatures also require static program information such as information about the vectorization and the program structure, that cannot be easily determined. The predictors developed for the method have very high F1-score which shows the high quality of predictions. The goal is met with and we have shown practical applicability of the developed method.

Once tuning of the program is done the user puts his program in the production. For that purpose, we generate an automatic tuning advice. The tuning advice consists of the script used to start the program and a configuration file for a runtime library. The script is used to select the optimal combination of compiler flags, prepare the application for execution and configure the runtime used to execute the application. For that purpose, the script configures the optimal number of processes or the optimal combination of MPI parameters. Once the application is started Score-P monitor configures the optimal combination of runtime parameters provided in the configuration file. These parameters might include the optimal number of threads, the optimal frequency or the best algorithm implementation. Special care was taken to reduce the overhead in the production run and therefore as part of the tuning advice, Score-P's filter file is used. The generated filter configuration file disables instrumentation of regions that are not used to apply tuning

actions. The filter can be applied to the program that is recompiled if recompilation is requested during the tuning process or in the worse case during the runtime. Filters that are applied during the recompilation keeps the instrumentation to its absolute minimum, compared to runtime where all instrumentation exists but events are ignored and therefore most of the overhead is gone.

8.1 Future Work

This thesis presented new plugins that follow trends from the Top500 list. However, based on the observation from the list there are opportunities to add new promising tuning aspects for emerging heterogeneous and manycore systems. New plugins could include:

- CUDA plugin. Most of accelerator-based systems in Top500 are NVIDIA based and therefore CUDA plugins could be where future work should concentrate.
- MPI mapping plugin. The plugin searches the MPI process mapping that results in the fastest executing code and/or most energy efficient one.
- MPICAP plugin. The plugin searches the most energy efficient number of MPI processes.

Besides the new plugins, this thesis also presented novel approaches to characterize the search space applicable to various tuning aspects (plugins). However, this is just a first step towards a complete characterization. There are many improvement opportunities like using the central optimization repository to predict behavior in different situations like a new architecture, exploring better prediction models, predicting on different granularity levels, e.g., use machine learning to predict scenarios that the search algorithms should evaluate, etc.

8.1.1 Central Optimization Repository

The techniques presented in this thesis are a foundation for further work in the area of machine learning applied to performance tuning. The basis of the future work in machine learning with PTF is in the historical performance database which allows storing past tuning. Since, computer facilities, like LRZ, often do not allow access to a remote central database but rather store performance data into a local database, the range of collected data could be extended by adding a merge tool that collects performance data from various machines and stores it in the central optimization repository (historical performance database). The generic design of the database allows stored data to be processed later and new estimation models to be build which predict various behaviors. This enables:

- Search algorithm based on recommendation system, e.g., collaborative filtering. The search algorithm predicts the tuning objective improvement from tuning objective improvement of other scenarios of a plugin. The collaborative filtering is trained with tuning objective improvement from other programs tuned with the same plugin. The signature consists of improvements of other evaluated scenarios. At first, the search algorithm selects several random scenarios and they are evaluated by the tuning plugin. Once the random scenarios are evaluated the recommendation system is trained. The search algorithm now predicts the improvement of the tuning objective for the complete search space. Scenarios are sorted according to their predicted improvements and the first batch of scenarios are selected for the evaluation by the plugin. Once the first batch is evaluated, the search algorithm retrains the recommendation system with these new scenarios and the process is repeated until given time budget expires.
- Search algorithms that predicts improvement of scenario's tuning objective values on a new architecture from scenario's tuning objective values on other architectures.
- Meta-plugins that predicts tuning potential of the tuning plugin on a new architecture from tuning potential of this plugin on other architectures.

Current limitations to overcome are:

- Evaluation of scenarios generated by a tuning plugin for a search space might result in errors during preparation or execution of scenarios, e.g., runtime violations due to wrongly generated binaries for CFS plugin. Currently, PTF is not tolerant of such events and often reacts unpredictable, e.g., crashes. Unsuccessfully evaluated scenarios do not have entries in the central optimization repository.
- The current implementation of the machine learning algorithm does not handle situations where there are some objective values missing. PTF could be extended to be error tolerant and then the machine learning support of the Adaptive Sequence with Predictive Model plugin could be extended to support missing objective values by predicting the missing objective value entry from the remaining objective values for the program. The prediction of the missing values could be done with Singular Value Decomposition (SVD). Currently, if one of the programs from the benchmarks we tested was sensitive to some points in the search space it was excluded from the suite.

Such information would give us the opportunity to build new prediction models that predict behaviors of the new application on a new architecture. Next, we could build a model that predicts the plugin behavior based on a plugin signature for a new program. A set of plugins could be used on a set of benchmarks. From the set of benchmarks, we could identify the set of signature applications that are representative of plugins. Then

we tune with a set of signature applications with a new plugin and capture its signature. From this signature, we can predict the behavior of the plugin on a new application.

Tuning effect with respect to different input sets. Prediction of the application behavior with different input sets.

Bibliography

[1] ASC Proxy Applications. http://www.lanl.gov/projects/codesign/proxy-apps/index.php.

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

[3] CORAL Benchmark Codes. https://asc.llnl.gov/CORAL-benchmarks/.

[4] The European AutoTune Project. http://www.autotune-project.eu/.

[5] LANL Proxy Applications. http://www.lanl.gov/projects/codesign/proxy-apps/lanl/index.php.

[6] LLNL Proxies. https://codesign.llnl.gov/proxy-apps.php.

[7] Mantevo Project. https://mantevo.org/.

[8] NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html.

[9] OpenCL. https://www.khronos.org/opencl/.

[10] The TOP500 Project. https://www.top500.org/.

[11] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–16, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[12] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 725–737, New York, NY, USA, 2015. ACM.

[13] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike O' Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In *Workshop on Profile and Feedback-Directed Compilation*, Paris, France, October 1998.

[14] Mirela-Madalina Botezatu. A study on compiler flags and performance events. Technical report, CERN openlab, 2012.

[15] Rosario Cammarota, Laleh Aghababaie Beni, Alexandru Nicolau, and Alexander V. Veidenbaum. Optimizing program performance via similarity, using a feature-agnostic approach. In Chenggang Wu and Albert Cohen, editors, *APPT*, volume 8299 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2013.

[16] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P OBoyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *In Proceedings of the 5th Annual International Symposium on Code Generation and Optimization (CGO)*, pages 185–197, 2007.

[17] Mohamad Chaarawi, Jeffrey M. Squyres, Edgar Gabriel, and Saber Feki. A tool for optimizing runtime parameters of open mpi. In Alexey L. Lastovetsky, M. Tahar Kechadi, and Jack Dongarra, editors, *PVM/MPI*, volume 5205 of *Lecture Notes in Computer Science*, pages 210–217. Springer, 2008.

[18] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[19] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[20] I-H. Chung and J.K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 30–, Washington, DC, USA, 2004. IEEE Computer Society.

[21] A. Collins, C. Fensch, and H. Leather. Masif: machine learning guided auto-tuning of parallel skeletons. In P.-C. Yew, S. Cho, L. DeRose, and D.J. Lilja, editors, *PACT*, pages 437–438. ACM, 2012.

[22] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale®processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 221–226, New York, NY, USA, 2005. ACM.

[23] Intel Corporation. Intel mpi library. developer reference for linux* os, 2003-2017.

[24] Intel Corporation. Intel opencl sdk user's guide. `https://software.intel.com/sites/default/files/m/e/7/0/3/1/33857-Intel_28R_29_OpenCL_SDK_User_Guide.pdf`, 2010-2011.

[25] Intel Corporation. Intel c++ compiler 16.0 user and reference guide, 08 2015. Online.

[26] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, September 2016. Online.

[27] Genaro Fernandes De Carvalho Costa. *Dynamic tuning of parallel/distributed applications on computational grids.* PhD thesis, Universitat Autnoma de Barcelona, 2009.

[28] Frederica Darema. *Performance Evaluation with Real Applications*, pages 8–18. The MIT Press, Cambridge, Massachusetts, 2001.

[29] Carsten F. Dormann, Jane Elith, Sven Bacher, Carsten Buchmann, Gudrun Carl, Gabriel Carr, Jaime R. Garca Marquz, Bernd Gruber, Bruno Lafourcade, Pedro J. Leito, Tamara Mnkemller, Colin McClean, Patrick E. Osborne, Bjrn Reineking, Boris Schrder, Andrew K. Skidmore, Damaris Zurell, and Sven Lautenbach. Collinearity: a review of methods to deal with it and a simulation study evaluating their performance. *Ecography*, 36(1):027–046, 1 2013.

[30] Christophe Dubach. Using machine-learning to efficiently explore the architecture/-compiler co-design space. *PhD Thesis*, 2009.

[31] Christophe Dubach, Timothy Jones, and Michael O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 262–271, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. A predictive model for dynamic microarchitectural adaptivity control. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 485–496, Washington, DC, USA, 2010. IEEE Computer Society.

[33] Jozo J. Dujmović. Universal benchmark suites. In *MASCOTS*, 1999.

[34] Lieven Eeckhout, Hans Vandierendonck, and Koenraad De Bosschere. Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 83–94, Washington, DC, USA, 2002. IEEE Computer Society.

[35] Leonardo Fialho and James Browne. Framework and modular infrastructure for automation of architectural adaptation and performance optimization for hpc systems. In *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ISC 2014, pages 261–277, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[36] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, Sep. 2012. Chapter author for Collective Communication and Process Topologies, and One Sided Communications.

[37] M. Frigo and S.G. Johnson. Fftw: An adaptive software architecture for the fft. pages 1381–1384. IEEE, 1998.

[38] G. Fursin, Y. Kashnikov, A. Wahid, M. Z. Chamski, O. Temam, M. Namolaru, E. Yom-tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, and C.K.I. Williams. Milepost gcc: machine learning enabled self-tuning compiler, 2009.

[39] Grigori Fursin and Olivier Temam. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, 7(4):20:1–20:29, Dec 2010.

[40] M. Gerndt and M. Ott. Automatic performance analysis with Periscope. *Concurrency and Computation: Practice & Experience*, 22(6):736–748, 2010.

[41] Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.

[42] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001.

[43] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, Jan 2012.

[44] M. Haneda, P. M. W. Knijnenburg, and H. A. G. Wijshoff. Optimizing general purpose compiler optimization. In *Proceedings of the 2Nd Conference on Computing Frontiers*, CF '05, pages 180–188, New York, NY, USA, 2005. ACM.

[45] M. Haneda, P.M.W. Knijnenburg, and H.A.G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. *International Conference on Parallel Architectures and Compilation Techniques*, 0:123–132, 2005.

[46] Masayo Haneda, Peter M. W. Knijnenburg, and Harry A. G. Wijshoff. On the impact of data input sets on statistical compiler tuning. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006.

[47] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2009.

[48] Miklós Homolya. Machine Learning Based Tuning of HPC Applications. Master's thesis, Technische Universität München,`http://www.autotune-project.eu/system/files/Thesis_Homolya.pdf`, Germany, 2014.

[49] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 114–122, New York, NY, USA, 2006. ACM.

[50] IBM. Ibm mpi: An mpi implementation for supermuc, 2011-2017.

[51] H. Jordan, P. Thoman, J.J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. A multi-objective auto-tuning framework for parallel codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 10:1–10:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[52] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[53] Takahiro Katagiri, Masaharu Matsumoto, and Satoshi Ohshima. Auto-tuning of hybrid mpi/openmp execution with code selection by ppopen-at. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 00:1488–1495, 2016.

[54] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, December 1997.

[55] Sotiris B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica (Slovenia)*, 31(3):249–268, 2007.

[56] Umesh Krishnaswamy and Isaac D. Scherson. A framework for computer performance evaluation using benchmark sets. *IEEE Trans. Comput.*, 49(12):1325–1338, Dec 2000.

[57] Saku Kukkonen and Jouni Lampinen. Gde3: The third evolution step of generalized differential evolution. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pages 443–450. IEEE, 2005.

[58] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Proceedings of the IEEE International*

Symposium on Performance Analysis of Systems and Software, 2005, ISPASS '05, pages 236–247, Washington, DC, USA, 2005. IEEE Computer Society.

[59] H. Leather and E. Bonilla. Automatic feature generation for machine learning based optimizing compilation. In *Code Generation and Optimization (CGO)*, pages 81–91, 2009.

[60] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 185–194, New York, NY, USA, 2006. ACM.

[61] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, Mar 2012.

[62] Jan-Patrick Lehr. Impact of compiler instrumentation on hardware performance counters. *Master Thesis*, 2017.

[63] Abdelhafid Mazouz, Sid Ahmed Ali Touati, and Denis Barthou. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In Waleed W. Smari and John P. McIntire, editors, *HPCS*, pages 273–279. IEEE, 2011.

[64] Roel Meeuws, S. Arash Ostadzadeh, Carlo Galuzzi, Vlad Mihai Sima, Razvan Nane, and Koen Bertels. Quipu: A statistical model for predicting hardware resources. *ACM Trans. Reconfigurable Technol. Syst.*, 6(1):3:1–3:25, May 2013.

[65] Ryan W. Moore and Bruce R. Childers. Automatic generation of program affinity policies using machine learning. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 184–203, Berlin, Heidelberg, 2013. Springer-Verlag.

[66] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[67] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[68] Carmen B. Navarrete, Carla Guillén, Wolfram Hesse, and Matthias Brehm. Autotuning the energy consumption. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, pages 668–677, 2013.

[69] Y.L. Nelson, B. Bansal, M. Hall, A. Nakano, and K. Lerman. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2008.

[70] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 319–332, 2006.

[71] Eunjung Park, John Cavazos, and Marco A. Alvarez. Using graph-based program characterization for predictive modeling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 196–206, New York, NY, USA, 2012. ACM.

[72] Beau Piccart, Andy Georges, Hendrik Blockeel, and Lieven Eeckhout. Ranking commercial machines through data transposition. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization, IISWC 2011, Austin, TX, USA, November 6-8, 2011*, pages 3–14, 2011.

[73] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, Robert W. Johnson, Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications*, 18:21–45, 2004.

[74] R. Ribler, J. Vetter, H. Simitci, and D.A. Reed. Autopilot: Adaptive control of distributed applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, pages 172–179, 1998.

[75] Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, Nov 1996.

[76] J. E. Smith. Characterizing computer performance with a single number. *Commun. ACM*, 31(10):1202–1206, October 1988.

[77] Olalekan A. Sopeju, Martin Burtscher, Ashay Rane, and James Browne. Autoscope: Automatic suggestions for code optimizations using perfexpert. In *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 8488*, ISC 2011, pages 19–25. CSREA Press, 2011.

[78] Mel Stockman, Mariette Awad, Raul Khanna, Christian Le, Howard David, Eugene Gorbatow, and Ulf Hanebutte. A novel approach to memory power estimation using machine learning. *International Conference on Energy Aware Computing (ICEAC)*, 2010.

[79] Robert L. Thorndike. Who belongs in the family. *Psychometrika*, pages 267–276, 1953.

[80] A. Tiwari, C. Chen, J. Chame, M. Hall, and J.K. Hollingsworth. A scalable autotuning framework for compiler optimization. *Parallel and Distributed Processing Symposium, International*, 0:1–12, 2009.

[81] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization*, pages 204–215. IEEE Computer Society, 2003.

[82] Yatish Turakhia, Guangshuo Liu, Siddharth Garg, and Diana Marculescu. Thread progress equalization: Dynamically adaptive power-constrained performance optimization of multi-threaded applications. *IEEE Trans. Computers*, 66(4):731–744, 2017.

[83] Yash Ukidave, Xiangyu Li, and David Kaeli. Mystic: Predictive scheduling for gpu based cloud servers using machine learning. In *The 30th IEEE International Symposium on Parallel and Distributed Processing*, 2016.

[84] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 213–224, Washington, DC, USA, 2012. IEEE Computer Society.

[85] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[86] Vincent M. Weaver. Linux perf event features and overhead. In *Proceedings of the 2014 3rd International Workshop on Performance Analysis of Workload Optimized Systems*, March 2013.

[87] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Dan Terpstra, and Shirley Moore. Measuring energy and power with papi. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*, ICPPW '12, pages 262–268, Washington, DC, USA, 2012. IEEE Computer Society.

[88] C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimization of software and the atlas project. *PARALLEL COMPUTING*, 27:2001, 2000.

[89] Cemal Yilmaz. Using hardware performance counters for fault localization. In *Proceedings of the 2010 Second International Conference on Advances in System Testing*

and Validation Lifecycle, VALID '10, pages 87–92, Washington, DC, USA, 2010. IEEE Computer Society.

[90] Wucherl Yoo, Kevin Larson, Lee Baugh, Sangkyum Kim, and Roy H. Campbell. Adp: Automated diagnosis of performance pathologies using hardware events. *SIGMETRICS Perform. Eval. Rev.*, 40(1):283–294, Jun 2012.

[91] Dmitrijs Zaparanuks, Milan Jović, and Matthias Hauswirth. Accuracy of performance counter measurements. In *ISPASS*, pages 23–32. IEEE Computer Society, 2009.

[92] Yun Zhang and Michael Voss. Runtime empirical selection of loop schedulers on hyperthreaded smps. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 44.2–, Washington, DC, USA, 2005. IEEE Computer Society.