Technische Universität München
Department of Informatics

Master's Thesis in Computational Science and Engineering

Machine Learning-based tuning of HPC applications

Author:	Miklós Homolya
1st Examiner:	Prof. Dr. Michael Gerndt
2nd Examiner:	Prof. Dr. Arndt Bode
Submission date:	September 15, 2014

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, _____ _____

Miklós Homolya

Contents

Chapter 1

Introduction

Run-time performance is crucial in scientific computing, high-performance computing, data centers, mobile and embedded devices. An optimized software may allow more accurate simulations, finer resolution, reduced energy consumption, and – in case of mobile – longer battery life.

Serial optimization should generally be the first choice when optimizing a piece of software, since it can often reduce execution time and energy consumption at the same time. Parallelization reduces execution time, but not the overall work to be done, and therefore cannot directly[1] reduce energy consumption. For the most challenging applications, every aspect of optimization should be exploited.

On modern computers, highest performance can only be achieved by targeting all the features and peculiarities of the specific computing architecture. However, manual tuning is often infeasible due to the high number of parameters to consider and the variety of target hardware. There is a need for automatic tuning and optimization.

Periscope is an automatic online and distributed performance analysis tool developed by Technische Universität München. AutoTune, a project funded by the European Union, extends Periscope with automatic online tuning plugins for performance and energy efficiency tuning. The project expects energy savings of up to 10% and performance improvements for high-performance applications. AutoTune features plugins tuning compiler flags, CPU frequency, MPI communication, GPGPU patterns, and others. This Master's thesis focuses primarily on the Compiler Flag Selection (CFS) plugin, aiming to improve performance by finding the best set of compiler optimizations for the given application.

Some program transformations, such as *dead code elimination* and *constant folding* are

[1] Running on more cores with less clock frequency reduces energy consumption, although the energy savings come from the reduced clock frequency, while parallelization compensates for the increase in execution time.

universally beneficial, which is good, but they still leave much room for improvement. Other program transformations may also degrade performance in certain cases – they may depend on parameters which are hard to guess at compilation time, or their effect may be platform-dependent.

Automatic tuning of compiler optimizations is nevertheless challenging. Modern optimizing compilers feature a large number of program transformations, each parametrized with at least two possible values (on, off), which expand a huge search space. These transformations *may* or *may not* affect each other. The order of transformations *might* be significant as well, even though we cannot control it through compiler flags. Different code segments may also require different tuning parameters, therefore maximum performance may require fine granularity of tuning sections.

Our goal is to use *machine learning* techniques to speed up the search for optimal tuning parameters, specifically compiler flags. We extend Periscope with a machine learning framework which can support both the CFS and other plugins. We plan to use *performance counters* as the signatures of applications for learning, and implement or integrate several machine learning techniques, such as *k-nearest neighbors*, or *Support Vector Machines* (SVM). For evaluation, we use the NAS Parallel Benchmarks (NPB) [1]. We also plan to create infrastructure for a collaborative extension of the training, such as *Collective Mind* (cM) [2].

Chapter 2

Related Work

In several cases, conventional high-performance languages such as C and Fortran are too low-level to conveniently to express certain optimizations and tuning opportunities.

FFTW [3] is an adaptive, self-tuning fast Fourier transform library, whose primary focus is not the reduction of floating-point operations (hardware independent), but of execution time, which is also affected by pipelining and memory hierarchy. FFT is a *divide and conquer* algorithm, therefore two kinds of *codelets* are generated: *normal* codelets directly compute DFT for short sequences, and *twiddle* codelets, which combine N_1 transforms of size N_2 to get a transform of N_1N_2. These codelets computing transforms of different sizes are generated by a program written in Caml Light, which also applies algebraic simplifications to them to produce optimal codelets. To compute a transform of size N, N must be factorized into twiddle and normal codelet sizes – such a factorization is called a *plan*. Dynamic programming is applied to learn optimal factorizations.

ATLAS [4] also uses code generation and search based tuning to portably and automatically tune and optimize *Basic Linear Algebra Subprograms* (BLAS). Linear algebra problems often offer a great opportunity to rearrange operations by blocked implementation and loop vectorization, which may lead to better cache performance and pipeline utilization. There are also conflicts in optimization targets, e.g. loop vectorization helps to fill the pipeline by reducing data dependencies, but also needs more registers, which might cause *register spilling*. ATLAS tries to estimate several hardware parameters (L1 cache size, pipeline length, number of floating-point registers) by running experiments. This information helps tuning the parameters for code generation.

SPIRAL [5] is a generator for platform-adapted libraries of DSP transforms. Its architecture consists of three modules: the *Formula Generator* generates fast algorithms for the transform ("fast" here refers to operation count) in the signal processing language (SPL); the *Formula*

Translator translates SPL into C or Fortran code, with certain platform-specific parameters such as degree of loop unrolling; and the *Search Engine* controls the generation of configuration settings for the former modules to find an optimal configuration. Various search methods are implemented and evaluated: exhaustive search, dynamic programming, random search, hill climbing search, and STEER, a stochastic, evolutionary search approach, similar to genetic algorithms, but it represents exploration space as ruletrees instead of bit vectors.

OSKI [6] tunes sparse matrix kernels at run-time, thus it can find the best representation for the given matrix. Matrices can only be accessed through handles, which implies that the matrix representation is transparent from the API. Nevertheless, each of the works above tunes only certain specific routines, while we primarily look for ways to improve any application.

Triantafyllis et al. [7] propose Optimization-Space Exploration (OSE), an *iterative compilation* approach making *a posteriori* optimization decisions. OSE applies aggressive pruning on the *configuration space*. Traditional *predictive heuristics* are used to create seed configurations, then the configuration space is explored by alternating *expansion* and *selection* steps. The expansion step constructs all configurations differing from one of the seeds in only one parameter, while the selection step reduces the set of considered configurations by selecting a few best ones. To explore the configuration space more effectively, OSE takes into account the correlation of configurations. Certain configurations perform well with certain program features, thus finding a configuration that gives good performance suggests other configurations which may also perform well. OSE also features a *static performance estimator* to speed up the evaluation of each configuration, i.e. the application is not run, but configurations are evaluated based on the generated machine code, a *simplified machine model* and profile data. Its simplified machine model considers ideal cycle count, data cache performance, instruction cache performance and also estimates branch misprediction.

Haneda et al. [8] tune compiler optimization flags based on *inferential statistics*. Test configurations are based on *orthogonal arrays* (OA). An orthogonal array is expressed as an $N \times k$ binary matrix, where columns are interpreted as compiler options, and each row defines a compiler settings. Such a matrix represents an OA if two arbitrary columns contain the patterns 00, 01, 10, 11 equally often. This implies that each option is turned on and off equally often. Having an orthogonal array generated for a set of compiler options, the program is compiled and run for each row. Statistically significant options are selected based on the Mann-Whitney test, and turned on or off depending on their effect. These options are then removed from the set of considered compiler options, and the process is repeated with a smaller OA, until there are no statistically significant options.

Pan et al. [9] introduce faster *search algorithms* for tuning compiler flags: *Batch Elimination* (BE), *Iterative Elimination* (IE), and their combination, *Combined Elimination* (CE). Batch Elimination starts with all optimizations *on* as baseline, then each optimization is separately turned *off* and evaluated whether the optimization is harmful. Optimizations, which are not harmful with respect to the baseline, form the result of the search. Batch Elimination is fast, and gives good results when optimizations are independent. Iterative Elimination is similar to BE, but it disables only the most negative optimization at once, then the process is repeated with the new configuration as "baseline" until there are no negative optimizations. Combined Elimination improves the speed of Iterative Elimination using a greedy approach: once the effect of all optimizations are measured individually with respect to the baseline, each negative optimization is tried in decreasing order of their negativity, and disabled immediately if still negative. In other words, the most negative optimization is disabled immediately, just as in Iterative Elimination, but then the second most negative optimization is tried in addition to the already disabled first one. If its effect is still negative, it is also disabled immediately (updating the "baseline"). Then the third most negative option is tried in addition to the disabled first and the potentially disabled second, and so on. Once all negative optimizations are tested cumulatively, a new iteration begins. (A generalized version of CE, which can handle not only binary, but also multi-value options, is described in [10].) Batch Elimination, Iterative Elimination, Combined Elimination, *exhaustive search*, OSE [7], and *Statistical Selection* [11] are experimentally evaluated on a set of benchmark programs. CE is claimed to achieve equal or better performance with less tuning time than the other alternatives.

Agakov et al. [12] use machine learning to automatically focus search on those areas likely to contain the best configuration. They speed up *random search* and *genetic algorithm* by up to an order of magnitude on large spaces, not counting the cost of training which is shared across programs. Two kinds of predictive models are evaluated: an *independent, identically distributed* (IID) model, which assumes that the goodness of each program transformation is independent, and a *Markov model*, which can capture simple interactions between program transformations. Both produce a probability distribution of transformations, suggesting that certain transformations are more likely to be beneficial than others. This probability distribution is then sampled for random search and for the initial population of the genetic algorithm. These predictive models are learnt by machine learning, using *k-nearest neighbors* and *static program features*, whose dimensionality is first reduced using *Principal Components Analysis* (PCA). The methods are evaluated on the UTDSP benchmark suite [13].

Cavazos et al. [14] use *performance counters* instead of static program features, and achieve

10% average speedup improvement over the highest optimization level of the PathScale EKOPath 2.3.1 optimizing compiler. Their modelling technique is based on the independent model and *logistic regression*, and the dimensionality of the feature vector is reduced by selecting a subset of performance counters which maximize the *mutual information* between that subset and good optimization sequences. The proposed technique is shown to find very good optimization sequences much faster than Combined Elimination.

Dubach et al. [15] suggest a machine learning based technique to automatically predict the performance of a modified program without actually having to run it. Certain *code features* are taken from the SUIF representation of transformed programs, and they form a *training set* with their corresponding performance values. (SUIF [16] is restructuring compiler infrastructure offering various source-to-source transformations.) Two feature-based predictors were used: a linear model and artificial neural networks (ANN). In an ANN, every input corresponds to a neuron. In order to keep the size of the ANN, and thus the number of inputs small, PCA (Principal Components Analysis) is applied to the code features. This technique is also able to estimate the performance of program transformations unseen in the training set. The authors also emphasize that such an accurate estimator can make hardware-software co-design possible. This approach is orthogonal to search algorithms.

Leather et al. [17] argue that such a technique only changes the problem of finding good program transformations into finding good code features, and proposes a machine learning technique to learn the code features themselves, described as *feature grammars*. The search component is based on *genetic programming* and *grammatical evolution*. This technique is evaluated on the problem of optimizing loop unrolling.

Fursin et al. [18] introduce Milepost GCC, the first publicly-available open-source machine learning-based compiler. Milepost GCC aims at multi-objective optimization, reducing execution time, compilation time and code size at the same time. This goal requires excessive amount of training – the authors report "a few weeks training" –, however, the idea is "to mitigate the need for *per-program* iterative compilation and learn optimizations *across programs* based on their features". This version of Milepost GCC uses *static program features* only. Two machine learning techniques are presented. The *probabilistic model* maps program features to distributions over configurations. For a test program, a good configuration can be estimated by maximization of conditional likelihood, or by *k-nearest neigbors*. The other machine learning technique is a *transductive model*, where optimization configurations and program features together form a feature vector for training. Many learning algorithms can accompany this representation; this paper uses decision a tree model. The probabilistic model performs much better

in benchmark experiments, but the transductive model produces a decision tree which is much easier to analyze.

This thesis focuses primarily on improving the Compiler Flag Selection (CFS) plugin in AutoTune to improve the execution time and energy efficiency of high-performance applications. An initial implementation of the CFS plugin was provided before the start of this work. This initial implementation had two search algorithms: exhaustive search, and "individual search". Individual search tries to optimize each compiler option individually. Starting with an empty configuration, every possible value of the first option is evaluated, and the k best ones form the next set of configurations. Then each of the at most k configurations of the current set is combined with each possible value of the next compiler option; they are all evaluated, and again the best k of them are kept. Once all specified options are processed, the best configuration of the last set is the "solution". To reduce compilation time, the user can provide the set of files containing hot code – recompilation is then restricted to these files only. A script is provided which, given profiling data, can automatically find these files. To further speed up tuning, execution time can be reduced for iterative algorithms using manually guided instrumentation. The user has to create a region around the body of the main loop, so the driver can terminate execution after a limited number of iterations. The granularity of tuning sections – at the beginning of this thesis – is the whole application, although it would not be too hard to specify different set of compiler flags for each file.

Chapter 3

Tuning with Periscope

Periscope was initially a performance measurement and analysis toolkit, later extended with automatic tuning capabilities. It is designed to operate in supercomputing environments, such as the SuperMUC.

Periscope is able to collect different kinds of performance and profiling information, called *properties*, such as execution time, energy consumption, hardware counters via PAPI, MPI-specific or GPU-specific profiling data. If it is not possible to collect all the required properties in a single run, then Periscope will automatically group these properties in a way that each group of profiling data can be collected during the same run, and Periscope will *restart* the application to collect profiling data for every group.

Periscope can profile both *instrumented* and *not instrumented* applications. Instrumentation is *manual* and *intrusive*. The user must define regions in the source code, marking their beginning and end, and the application must be compiled with the help of Periscope's instrumentation wrapper, `psc_instrument`. A typical approach for instrumenting iterative algorithms is to create a `USER` region around the body of the application's main loop. Despite the manual work, instrumentation is preferred, since it offers several benefits. Instrumented measurements are *more accurate*, and we can define more accurately what we wish to measure. Periscope offers a way to limit how many times the main loop is run, which can significantly speed up the collection of profiling information. Periscope can *stop* and *resume* instrumented applications, and resume can replace restarts in cases when several runs are necessary to collect all required information.

Figure 3.1 represents the tuning architecture in Periscope. Users can select, what aspect of the application they want to optimize (compiler flags, MPI parameters, CPU frequency *et cetera*). The corresponding tuning *plugin* is loaded, which may request an optional *analysis phase* to collect profiling information which may be relevant for the tuning. The plugin then

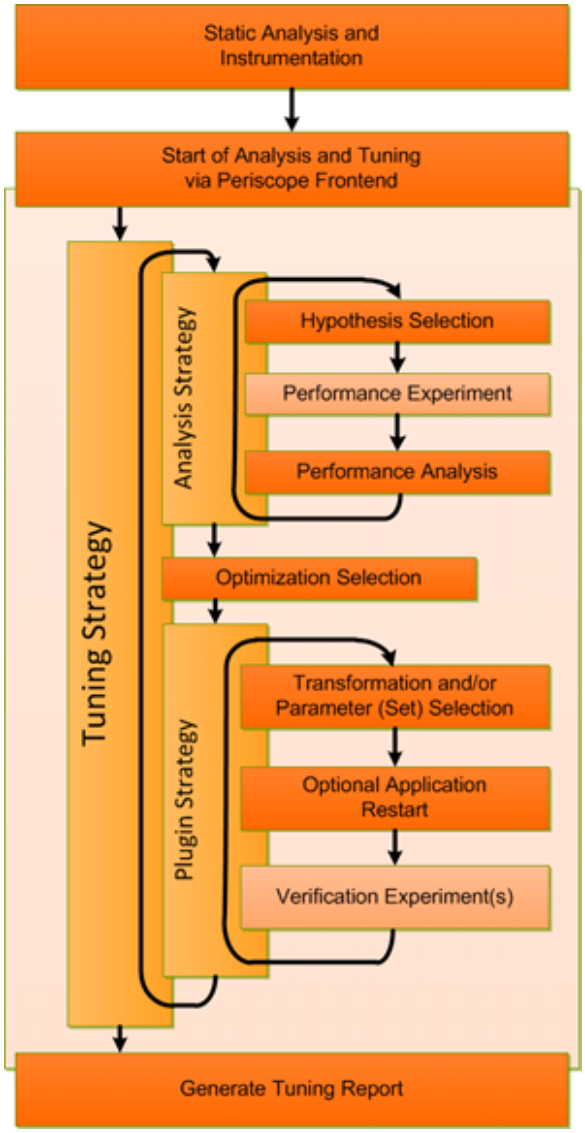


Figure 3.1: Tuning architecture of Periscope.

creates a *search space*, i.e. a space of possible configurations. A *search algorithm* guides the exploration of the search space, generating *scenarios*, which are executed by Periscope. If the tuning process is not finished after the search phase is finished, another (optional) analysis and search phase may begin, as it is denoted by the arrow in Figure 3.1. When the tuning is finally over, a *tuning report* is generated, which contains the suggested configuration.

Chapter 4

Machine Learning

This chapter first introduces a few important machine learning techniques, then – based on related work – we summarize the approaches on how to speed up tuning with machine learning, concerning topics relevant to the Periscope Tuning Framework.

4.1 Basic Concepts and Techniques

For purpose of applying machine learning to automatic tuning, we first briefly review *supervised learning*. Supervised learning aims to learn a function $\mathbf{x} \mapsto y$ from a set of *training data*: $(\mathbf{x}^{(1)}, y^{(1)})$, $(\mathbf{x}^{(2)}, y^{(2)})$, ..., $(\mathbf{x}^{(m)}, y^{(m)})$. The trained model is able make a prediction y^* for a value $\mathbf{x}^*$, which has not been seen yet.

If the range of the function is discrete and finite, we talk about a *classification* problem. If it is a real number, then we call it a *regression* problem. Periscope is designed in a way that each tuning parameter is defined to have a finite number of possible values – this is required by search algorithms, such as exhaustive search or individual search, which try all possible values for each parameter. Therefore classification methods can be applied for all cases, although some problems – e.g. predicting optimal CPU frequency – might be better formulated as regression problems.

4.1.1 k-Nearest Neighbors algorithm

The *k-nearest neighbors* algorithm estimates y^* by finding those k vectors in $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(m)}$, which are the nearest to $\mathbf{x}^*$, and "averaging" (aggregating) their corresponding y labels.

In case of $k = 1$, no aggregation is necessary, thus the method boils down to finding the i index which minimizes $\|\mathbf{x}^{(i)} - \mathbf{x}^*\|$, and returning $y^{(i)}$ as y^*.

4.1.2 Linear regression

Linear regression learns a linear function of the form

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \tag{4.1}$$

which minimizes the cost function

$$J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} \left(h_w(\mathbf{x}^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} w_i^2 \tag{4.2}$$

where λ is the regularization parameter. $\lambda = 0$ means no regularization. A common convention is to set $x_0 = 1$, so that the linear function can be written in the compact $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$ form.

4.1.3 Logistic regression

Despite the regression in the name, *logistic regression* is actually a *binary classifier*. It learns the function

$$h_{\mathbf{w}}(\mathbf{x}) = g(w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n) = g(\mathbf{w}^T \mathbf{x}) \tag{4.3}$$

where g is the *logistic* or *sigmoid* function (see Figure 4.1):

$$g(z) = \frac{1}{1 + e^{-z}} \tag{4.4}$$

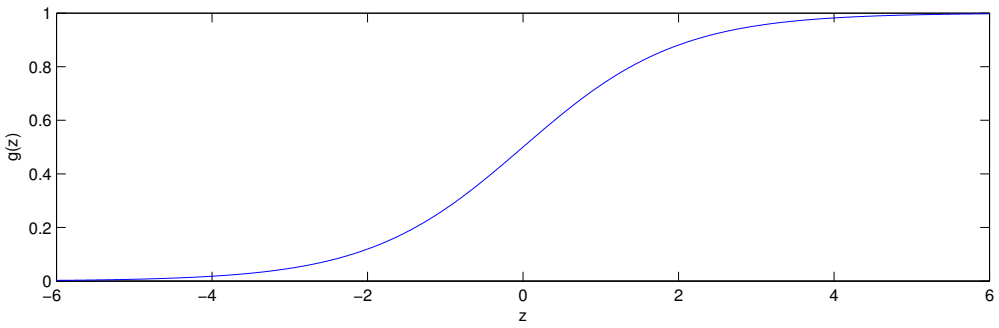


Figure 4.1: The logistic function.

One could use a cost function of the same form as in (4.2), but that cost function would not be convex when $h_{\mathbf{w}}(\mathbf{x})$ is defined as in (4.3). This would make optimization hard, therefore usually the following (convex) cost function is used:

$$J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} \left[-y^{(i)} \log h_{\mathbf{w}}(\mathbf{x}^{(i)}) - (1 - y^{(i)}) \log(1 - h_{\mathbf{w}}(\mathbf{x}^{(i)})) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} w_i^2 \tag{4.5}$$

Gradient descent can find the $\mathbf{w}$ which minimizes the cost function. There are also some more advanced optimization algorithms, such as *conjugate gradient*, BFGS, L-BFGS.

Once $\mathbf{w}$ is trained, the *positive class* is predicted when $h_{\mathbf{w}}(\mathbf{x}) \geqslant 0$, and the *negative class* is predicted when $h_{\mathbf{w}}(\mathbf{x}) < 0$. $\mathbf{w}$ basically defines the *separation line* in the hyperspace of input features.

4.1.4 Support Vector Machines (SVM)

There are several different versions of *Support Vector Machines*. Their simplest form implements a *binary classifier*, similar to logistic regression, but with a different cost function. We could rewrite (4.5) in the following form:

$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\, cost_1(\mathbf{w}^T\mathbf{x}^{(i)}) + (1-y^{(i)})\, cost_0(\mathbf{w}^T\mathbf{x}^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n} w_i^2 \tag{4.6}$$

where $cost_1(z) = -\log\frac{1}{1+e^{-z}}$ and $cost_0(z) = -\log\left(1-\frac{1}{1+e^{-z}}\right)$ for logistic regression. SVM approximates these cost functions with *piecewise linear functions* which offer more efficient optimization opportunities:

$$cost_1(z) = \begin{cases} c\cdot(1-z) & \text{if } z < 1 \\ 0 & \text{if } z \geq 1 \end{cases} \tag{4.7}$$

$$cost_0(z) = \begin{cases} 0 & \text{if } z \leq -1 \\ c\cdot(1+z) & \text{if } z > -1 \end{cases} \tag{4.8}$$

where c is a positive constant which approximates well the former cost functions.

An important consequence of this choice is that SVM without regularization implements a *maximum-margin classifier* (see Figure 4.2) on a separable dataset, i.e. the minimal distance of any data point from the hyperplane is maximized among the separating hyperplanes.

Instead of the regularization parameter λ, the conventional formulation of SVM features the parameter C, which is similar to $1/\lambda$. Thus the conventional formulation of the cost function of SVM is:

$$J(\mathbf{w}) = C\sum_{i=1}^{m}\left[y^{(i)}\, cost_1(\mathbf{w}^T\mathbf{x}^{(i)}) + (1-y^{(i)})\, cost_0(\mathbf{w}^T\mathbf{x}^{(i)})\right] + \frac{1}{2}\sum_{j=1}^{n} w_i^2 \tag{4.9}$$

With the help of a *non-linear* transformation φ, SVM can also classify datasets where the separating manifold is not a hyperplane. SVM applies *linear separation* in the transformed space, but due to the non-linear transformation, it will correspond to a non-linear separator in the original space. Mapping $\mathbf{w}$ and $\mathbf{x}^{(i)}$ replaces the dot product $\mathbf{w}^T\mathbf{x}^{(i)} = \left\langle \mathbf{w}, \mathbf{x}^{(i)} \right\rangle$ with $\left\langle \varphi(\mathbf{w}), \varphi(\mathbf{x}^{(i)}) \right\rangle$. Since we need to evaluate only expressions of the form $\langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle$, instead of using φ directly, often a *kernel function* K is used, where $K(\mathbf{x}, \mathbf{x}') = \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle$. There are several kinds of kernels used, the following two being the most often used:

Linear kernel (no kernel)

$$K(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle$$

Radial basis function (RBF) kernel (with parameter γ)

$$K(\mathbf{x}, \mathbf{x}') = e^{-\gamma\|\mathbf{x}-\mathbf{x}'\|}$$

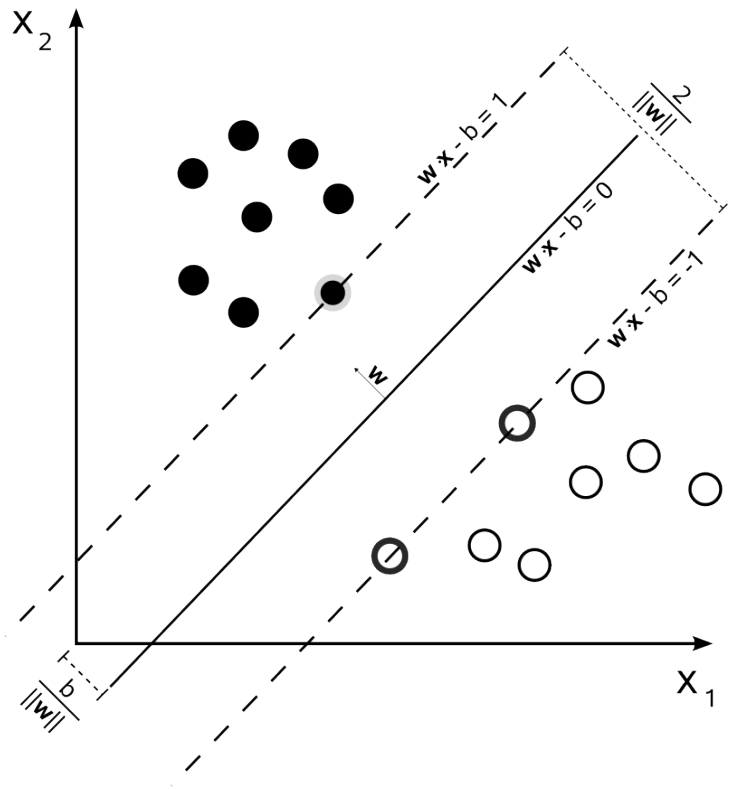


Figure 4.2: A hyperplane separating two datasets with maximum margin. Data points on the margin are called *support vectors*. This is where the name *Support Vector Machine* comes from, since SVM finds such a separating hyperplane.

4.2 Machine Learning for Automatic Tuning

Before the contributions of this thesis, the only way Periscope could tackle the problem of huge search spaces was "smart" search algorithms, which use heuristics not to explore the whole search space, but still find the best or a reasonably good configuration. However, search was started over for each new program, even if a program were indeed very similar to a previously encountered one. The idea is to use *machine learning* to exploit this potential speedup opportunity. If we have already tuned a program which is similar to the program we are currently tuning, then we may reuse the results from that previous tuning to make tuning faster.

The first ingredient to applying machine learning to the tuning problem, is to define this "similarity" of programs. To do that, we describe programs by a vector of *program features*, $\mathbf{p}$. We chose to use *PAPI performance counters* as program features. To prevent differences due to shorter or longer running times, we divide each event counter value by the total number of instructions (`PAPI_TOT_INS`). Two programs, p_1 and p_2 are considered "similar", if $\|\mathbf{p_1} - \mathbf{p_2}\|$ is not far from zero.

Experience shows that while some tuning parameters may be very influential on performance, others may have little effect. That means that several different configurations can be within the measurement accuracy of the best configuration. In order not to lose valuable information, we save our *training data* as $(\mathbf{p}, \mathbf{t}, s)$ tuples, where $\mathbf{p}$ denotes the *program features*, $\mathbf{t}$ denotes the *tuning parameter values*, and s means *speedup*. This way we are able to either train a $(\mathbf{p}, \mathbf{t}) \rightarrow s$ predictor, or consider only the best or a few "near best" configurations and train a $\mathbf{p} \rightarrow \mathbf{t}$ predictor. The former approach does not change how the search space is explored, but may speed up the exploration by using a prediction instead of actually executing the program. This is still no help with huge search spaces, where simply iterating through all the possible configurations is infeasible. Therefore we are going to stick to the latter approach, which has two variants in the related works:

Focused search

Predict a model M which "focuses" the search for the best configuration. This approach requires search algorithms, which can be "guided" by a model predicted from training data, for example:

Random sampling search randomly samples the search space, and thus can use a probability model to focus sampling.

Genetic algorithms randomly sample an initial set of configurations, and thus can also benefit from a probability model.

Two probability models are suggested in [12]:

Independent model assumes no interaction between tuning parameters, i.e. the effect of the choice of each tuning parameter value is independent.

Markov model is able to capture interactions between tuning parameters.

Direct prediction

Suggested configuration is directly predicted, and no search is involved. From a probability model, for example, we can simply predict the configuration which is the most likely to be the best.

Periscope has already had a genetic algorithm, GDE3 [19]. Random sampling search is added to Periscope in the context of this thesis.

In following we are going to focus on the prediction of a probability model from training data. *K-nearest neighbors* can be used with both probability models. We look for the k nearest programs in the training data, take their best configurations, and build the model from those

configurations. In case of the independent model, we can also train per tuning parameter *classifiers* or *regressors*.

Although we know that the assumption about the independence of tuning parameters is false, since tuning parameters – especially program transformations controlled by compiler flags – do affect each other, we focus more on the independent model, since the Markov model has much higher degree of freedom, and therefore requires a lot more training data according to [12] to outperform the independent model.

Chapter 5

Design and Architecture

This chapter details the design and architecture of machine learning support for automatic tuning Periscope. Periscope had no support for machine learning before this thesis, nor did it have a tuning data storage persistent across different runs, and several problems had to be solved which were not really problems before the requirement to have machine learning support. We discuss here the architecture of the new software components added to Periscope, their design considerations and the goals we wanted to achieve, and occasionally we also have to explain existing design pieces with which our new components interact.

Section 5.1 gives a "big picture" view of how tuning works in Periscope, and how it is extended with machine learning capabilities. Section 5.2 explains the *input features* of machine learning, their representation, and the way they can be collected in Periscope. Section 5.3 describes the representation of *tuning configurations*, going into details about how Periscope represents and handles them internally, and simplifies that representations to the bare needs of machine learning in order to achieve the flexibility we need. Section 5.4 describes the functions of the tuning database and the interface through which it is used. Section 5.5 discusses the machine capability of *search algorithms*, and in particular discusses the *Random Sampling Search*, a search algorithm supporting machine learning which was added to Periscope in the context of this thesis. Section 5.6 discusses the machine learning approaches we use to predict probability models to focus search algorithms such as Random Sampling Search.

5.1 Overview

This section gives a brief overview of the major components involved in making machine learning-based automatic tuning in Periscope possible. First, let us have a look at how tuning works in Periscope without machine learning. Figure 5.1 shows the tuning approach in a very

simplified manner. In fact, it shows only those details of the tuning process which are affected by adding machine learning. Searching for a good configuration is done through the help of a *search algorithm* (which may simply be exhaustive search or a more sophisticated one). First, the plugin initializes the search algorithm by defining the *search space* of configurations. The search algorithm will then guide the exploration of the search space, and tuning experiments will be carried out through an interaction between the search algorithm and the plugin. Finally, when the search is over, the plugin reports the tuning results to the user, and recommends a configuration (the best performing one).

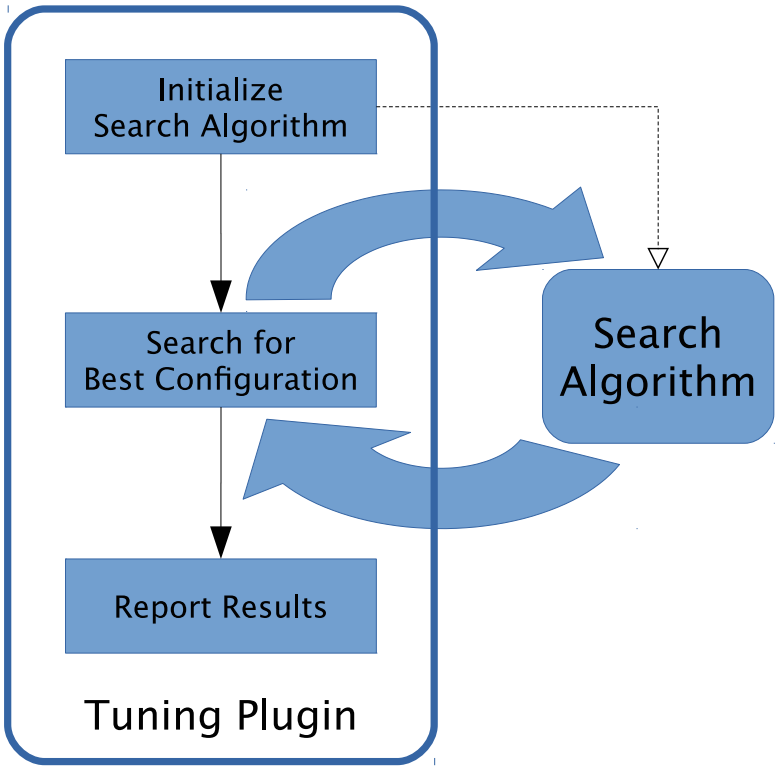


Figure 5.1: Simplified representation of tuning with Periscope without machine learning.

Before discussing the architecture of full support for machine learning, let us first examine what shall we do to collect tuning data for training purposes. This requires machine learning support from the plugin, but not from the search algorithm. Figure 5.2 shows the extended architecture. We see a new component, the *tuning database*, which serves the purpose of storing tuning data for later retrieval. Instead of simply reporting tuning results to the user, these results are also stored in the tuning database. There is another important change in the plugin: the collection of *program features*, which serve as input features of the machine learning process. Program features are submitted to the tuning database together with the tuning results.

Finally, Figure 5.3 demonstrates the use of machine learning to speed up automatic tuning. This requires both the plugin and the search algorithm to support machine learning. Compared

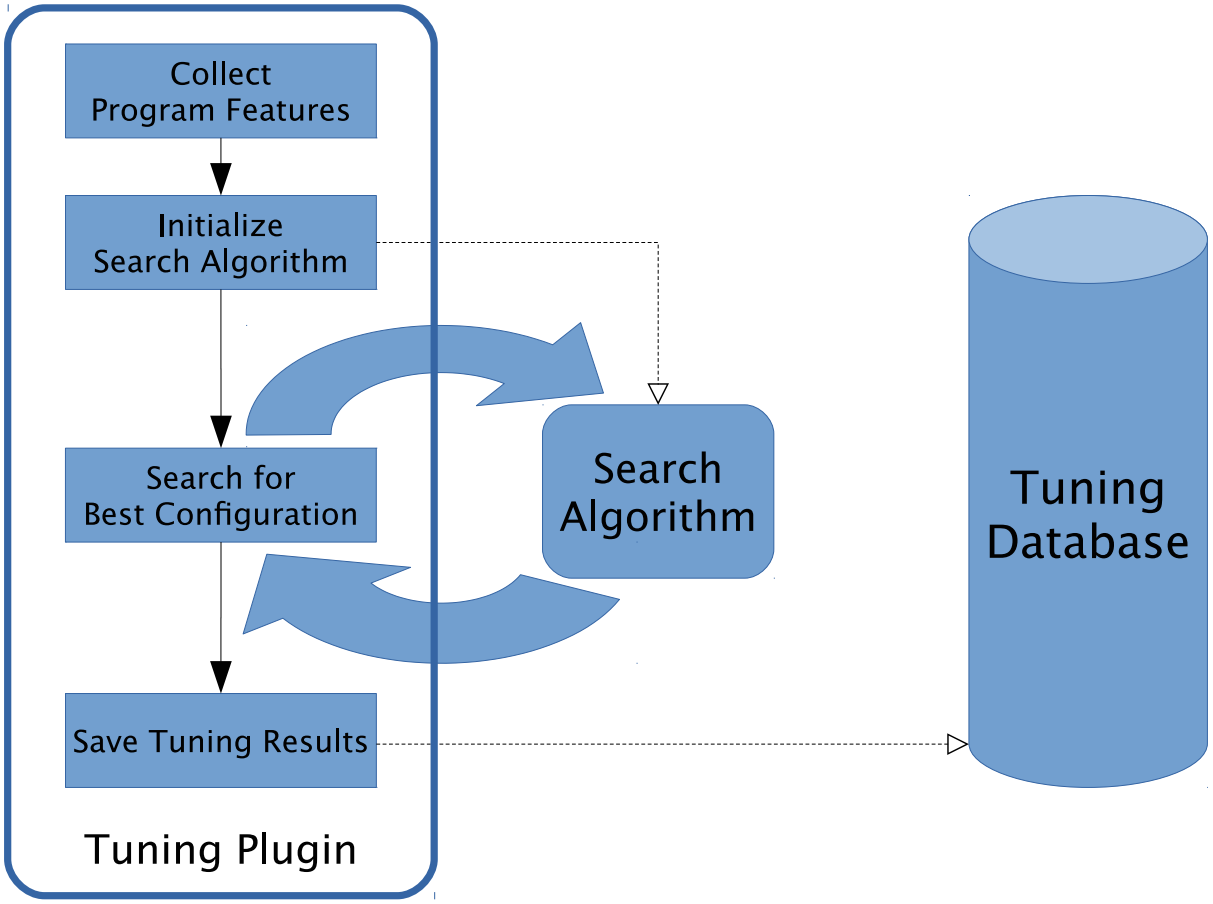


Figure 5.2: Collection of tuning data in a plugin supporting machine learning.

to the previous diagram, now the plugin also passes the program features to the search algorithm in addition to the search spaces. This way the search algorithm will have all the information that it needs for machine learning. The search algorithm can retrieve tuning data from the tuning database, train a model from the data, and make predictions from the program features (input features) to speed up finding a good configuration.

5.2 Program Features

We disclosed in section 4.2 that we chose to use PAPI performance counters as *input features* for machine learning. In this section, we briefly introduce their representation, and then explain how they can be collected in Periscope.

The representation of program features is indeed very simple. `ProgramSignature`, a new class wraps a `std::map<std::string, INT64>`, where the key is the name of the PAPI event counter, and the value is its value. The accessor method, `operator[]` is defined in a way to automatically normalize each event counter by the number of total instructions, `PAPI_TOT_INS`, thus returning a floating-point number.

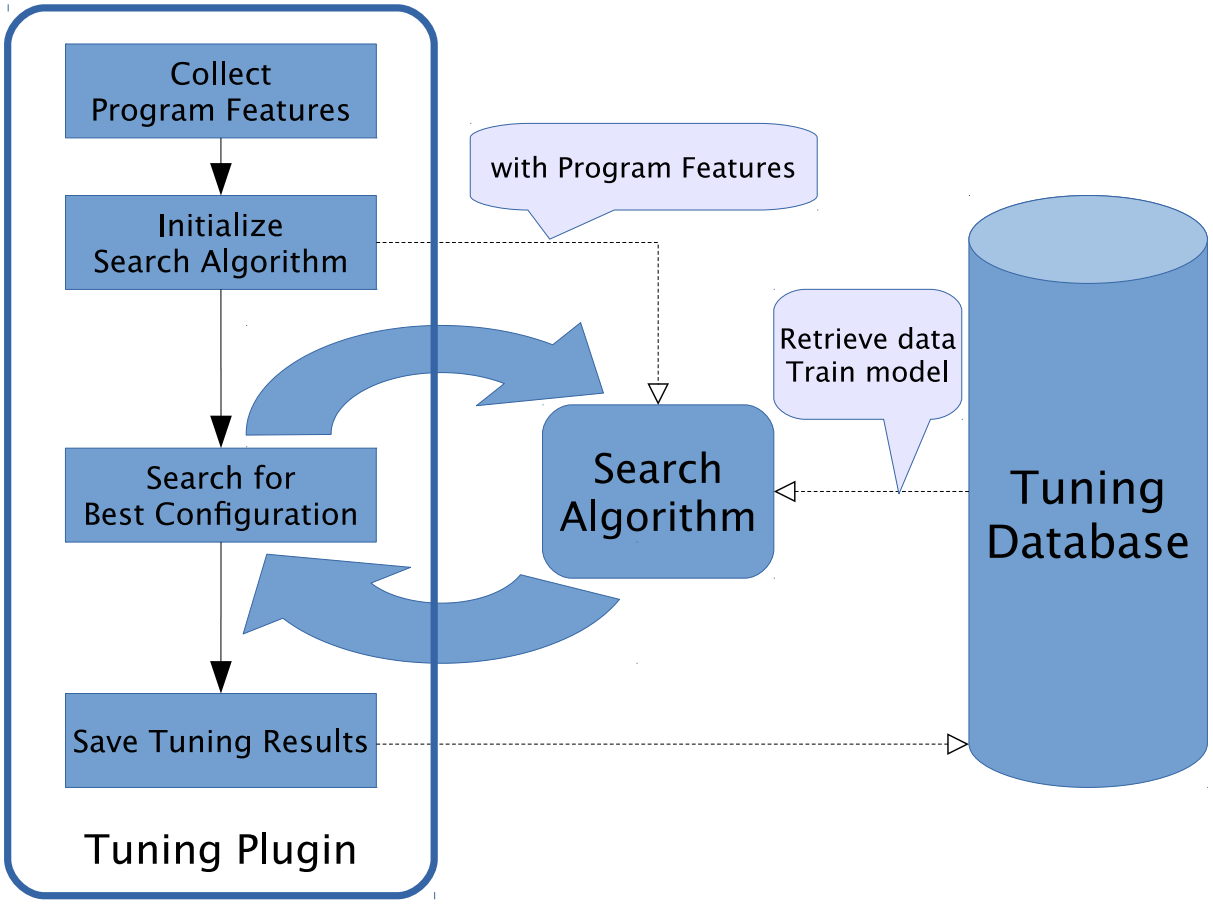


Figure 5.3: Re-use of formerly saved tuning data to speed up tuning. Both the plugin and the search algorithm must support machine learning.

Collecting PAPI event counters, however, involves a more intimate knowledge of Periscope's architecture. Figure 5.4 shows the state machine and the flowchart of how tuning is carried out. Most of these boxes are implemented in the *tuning plugins*, which derive from the `IPlugin` interface[1]. In other words, many of the boxes in the flowchart correspond to virtual functions in the `IPlugin` interface, which are then implemented in a derived class. As machine learning was added to the *Compiler Flag Selection* plugin in the context of this thesis, our discussion of collecting program features will closely follow how machine learning support has been implemented in CFS.

In Periscope, run-time information about program execution is defined in *properties* which are collected during *experiments*. The CFS plugin, for example, creates an experiment for each scenario generated by the search algorithm, and requests a property measuring the *execution time*. Similar experiments can be carried out before the *search phase* in the optional *analysis*

[1] A C++ class defining only public pure virtual member functions, and usually a virtual destructor with an empty body.

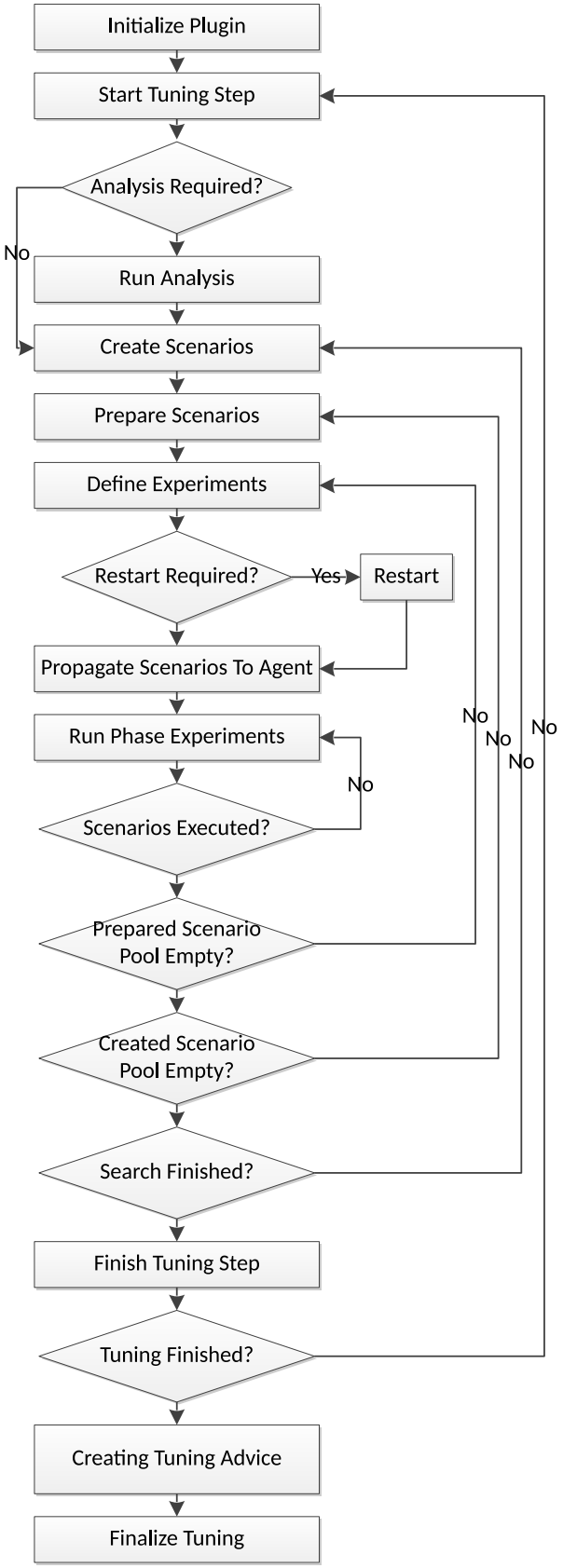


Figure 5.4: Plugin flowchart of Periscope.

phase (see Figure 5.4). The CFS plugin, for example, may use the analysis phase to determine which source files give most of the execution time – only those file will be re-compiled during the search phase.

A *property* contains several pieces of information, such as: **severity** specifies the importance of the property; **confidence** describes the degree of confidence about the existence of the performance property; **condition** tells if the performance property is going to be reported. Properties are defined as subclasses of the `Property` class. Properties, which are to be reported, are first converted to XML, then the XML is parsed into a `MetaProperty` object, which is accessible from the *analysis results pool* or the *scenario results pool*, depending on whether the property was requested in the analyis phase or in the search phase. This architecture also offers a way to extend a property with further information in addition to the predefined ones (condition, confidence, severity). By overriding the `toXMLExtra()` virtual member function in `Property`, we can squeeze extra information into the XML, as it is created from the property object. This extra information is then accessible from `MetaProperty` via `getExtraInfo()` as an `addInfoType` (alias to `std::map<std::string, std::string>`).

In order to collect the PAPI hardware event counter we need as program features, we created a new property, named `PerformanceCounters`. This property requests the collection of all PAPI event counters already defined in Periscope. Its severity is the execution time, but it forwards all collected event counter values as *extra information*. As explained above, this extra information is accessible from the `MetaProperty`, and `ProgramSignature` has a constructor which converts from an `addInfoType`. This way we can gather our program features in Periscope.

In case of the CFS plugin, we collect program features in the *analysis phase*. The "Analysis Required?" branch of Figure 5.4 is implemented by the `analysisRequired` virtual function of `IPlugin`. In the implementation of this function, we can file a *strategy request*, which we use to request the collection of the `PerformanceCounters` property. The analysis results will be available only once we reach "Create Scenarios" (corresponding to virtual function `createScenarios`). We have to be cautious with *multi-step* search algorithms, as they will make `createScenarios` be called multiple times, while processing the analysis results again may be unwanted. See section 6.6.2 about this issue in CFS and its solution.

5.3 Configuration Representation

In section 4.2 we discussed that the training data for machine learning should be $(\mathbf{p}, \mathbf{t}, s)$ tuples, where $\mathbf{p}$ denotes the program features, $\mathbf{t}$ denotes the tuning parameter values, and s means speedup. In this section we discuss the detailed representation of $\mathbf{t}$ and the basic architecture

of its implementing C++ class in Periscope, `TuningConfiguration`. Further implementation considerations about `TuningConfiguration` are in section 6.1, and about the representation of **t** in the database in section 6.4.

5.3.1 From `Scenarios` to `Variants`

In Periscope, *search algorithms* get `SearchSpaces`, which define the space of possible configurations to explore, and [search algorithms] create `Scenarios`, which define the concrete measurement to carry out, including the selected configuration from the space of possible configurations. In the following we discuss how information propagates from `SearchSpaces` to `Scenarios` with the mediation of search algorithms, and we give a simplified representation of a configuration, implemented by the C++ class `TuningConfiguration`.

A `SearchSpace` consists of `Regions` and a `VariantSpace`. A `VariantSpace` consists of `TuningParameters` and `VariantSpaceConstraints`. The extracts of these definitions from the source code of Periscope are the following:

```
class SearchSpace {
  vector<Region*> regions;
  VariantSpace* variantSpace;
};

class VariantSpace {
  vector<TuningParameter*> tuningParameters;
  list<VariantSpaceConstraint> vsConstraints;
};
```

`TuningParameters` define their set of possible values, and we discuss them in detail later in the next subsection. Variant space constraints are *ignored*. In summary, a `SearchSpace` has a set of `TuningParameters` (with their set of values), which define the *configuration space*, and a set of *code regions*, on which the selected configuration has to be applied. By passing several `SearchSpaces` to a search algorithm, the user can define a different configuration space for each set of code regions.

A `Scenario` has `TuningSpecifications`, and other data fields controlling the creation and execution of experiments – those can be ignored here. A `TuningSpecification` has a `Variant`, a `VariantContext`, and `Ranks`. A `Variant` is an assignment of values to `TuningParameter` objects. A `VariantContext` is a *tagged union* of code regions or source files. We ignore `Ranks`, since search algorithms do so as well. Relevant source code extracts are the following:

```
class Scenario {
  list<TuningSpecification*> *ts;
  // other data members are not used by search algorithms
};

class TuningSpecification {
  Variant *variant;
  VariantContext context;
  Ranks ranks;
};

class Variant {
  map<TuningParameter*, int> value;
};

typedef struct VariantContext_t {
  int type;
  ContextUnion context_union;
} VariantContext;

typedef union ContextUnion_t {
  list<Region*> *region_list;
  list<string> *file_list;
} ContextUnion;
```

Search algorithms create `Scenario`s by assigning values to the `TuningParameter`s in the `SearchSpace`s. Since a search algorithm can be given multiple search spaces, every created scenario shall have a separate `TuningSpecification` instance for each `SearchSpace`. Regions are simply propagated from the `SearchSpace` to the `VariantContext` of the `TuningSpecification`.

To reduce complexity and improve flexibility, we decided to use **machine learning on a *per* search space basis.** This imposes the following requirement on the plugins wishing to support machine learning:

- *Program features* must be gathered for (the code regions of) each search space *separately.* Search spaces shall be passed to the search algorithm together with the program features gathered on the corresponding regions of the program.

To support this requirement, the `ISearchAlgorithm` interface has been extended, for further details see section 5.5.

With this architecture, the $\mathbf{t}$ in our $(\mathbf{p}, \mathbf{t}, s)$ training data tuples should contain information equivalent to that of a `Variant`. In the next section we discuss `TuningConfiguration`, a C++ class designed to be a substitute for `Variant` for the purposes of machine learning.

5.3.2 From `Variants` to `TuningConfigurations`

The `Variant` class in Periscope was designed a long time before considering the extension of Periscope with machine learning capabilities. It contains a lot of information besides specifying the *tuning configuration* to run the program with. Some of these information may change if *plugin configuration* changes. These are not problems when tuning is done in a single run of Periscope. However, machine learning requires gathering training data in different runs, and later reusing them in another run to make tuning faster. `Variant` simply *lacks the flexibility* required by machine learning. We provide a new class, `TuningConfiguration`, which (with respect to search space exploration and machine learning) contains the same information as `Variant`, but lacks all the "cruft" which contribute to `Variant`'s inflexibility.

Although the discussion of design considerations below is mostly at the C++ level as it is found in Periscope's source code, the final design of the `TuningConfiguration` class is general enough to provide a basis for the design of database representations.

Let's recall that the `Variant` class essentially wraps a `std::map<TuningParameter *, int>`, which assigns values to tuning parameters. A `TuningParameter`, however, contains quite some information:

```
class TuningParameter {
  long ID;
  tPlugin pluginType;
  runtimeTuningActionType runtimeActionType;
  std::string name;
  int from, to, step;
  Restriction *restriction;
};
```

These data members can be grouped according to the purpose they serve:

Tuning parameter identification: `ID`, `pluginType` and `name` identify *what* is being set. `pluginType` identifies which tuning plugin uses the parameter, while `name` and `ID` can

distinguish between the tuning parameters of the same plugin. We found that `name` works better for this purpose across different runs of Periscope, because for some plugins (e.g. MPI, CFS) `ID` depends on the order in which the parameters appear in the configuration file of the plugin.

Control data: `runtimeActionType` controls how the tuning parameter should be processed when executing an experiment. `restriction` can do several things, such as limiting the code regions on which the parameter shall have an effect. These are necessary for the execution of experiments, but have little to do with machine learning.

Value range: `from`, `to` and `step` define the range of possible values. `restriction` may impose a further constraint on the set of allowed values. These are absolutely *superfluous* in the `Variant`, as the `Variant` has one single value assigned already to each tuning parameter.

A `TuningParameter` → *value* mapping can also be represented as a set of (`TuningParameter`, *value*) pairs, which can be decomposed into (*tuning parameter identification*, *control data*, *value range*, *value*) tuples. Considering that *control data* and *value range* are irrelevant for our purposes, we can simplify a `Variant` into a set of (*tuning parameter identification*, *value*) pairs. We introduce `TuningValue` to represent these pairs. `TuningConfiguration`, a substitute to `Variant`, is basically a `std::set<TuningValue>`. Thus the core ingredients of a `TuningValue` are the following:

How intrusive is this?

```
class TuningValue {
  // tuning parameter identification
  tPlugin pluginType;
  std::string name;
  // tuning value
  int value;
};
```

For the purpose of machine learning, it is essential that `TuningValue` have an *equality operator* (`==` and `!=` in C++) defined. However, for efficiency reasons, we chose to equip `TuningValue` with a *three-way comparison* method. For further discussion about implications and alternative options, see section 6.1.

To make `TuningValue` interoperable with the rest of Periscope, there has to be a way to coerce between `TuningParameter` and `TuningValue`. (We remind again that a `TuningValue` does *not* correspond to a mere `TuningParameter`, but to a `TuningParameter` *and* an assigned

value.) For this purpose, we extended `TuningParameter` with a new virtual function which – given an assigned value – produces a `TuningValue`:

```
boost::optional<TuningValue>
TuningParameter::getTuningValue(int value) const {
  return TuningValue(pluginType, name, 0, value);
}
```

`TuningValue`s are stored in the tuning database, and are compared against each other when doing machine learning. We will see later in section 5.6 that there is no need to restore a `TuningParameter` from a `TuningValue`.

5.3.3 `TuningParameter` subclasses and `TuningValue` extensions

In the previous section we have shown how a `TuningParameter` and its assigned value are compacted into a new type, `TuningValue`, to obtain the flexibility necessary for machine learning. However, in some cases `TuningParameter` fails to provide enough data to accurately describe a tuning parameter, and a subclass is introduced to extend `TuningParameter` with the required information. At the time of writing this thesis, the only example of such subclassing in Periscope is `CFSTuningParameter`, a class describing compiler flags as tuning parameters.

We added an extension infrastructure which lets `TuningValue` be augmented with additional data members to be able to represent a tuning parameter value precisely in all cases. We achieve this through adding `TuningValue` a pointer to extensions:

```
class TuningValue {
  // tuning parameter identification
  tPlugin pluginType;
  std::string name;
  // extensions
  TuningValueExtension *extension;
  // tuning value
  int value;
};
```

The reason for choosing `extension` to be a pointer is twofold: *a*) it can be `NULL`, since in most cases extensions are not required; *b*) it can be polymorphic, i.e. plugins can define their own way of extending `TuningValue`.

`TuningValueExtension` is an *interface* that defines virtual functions for the operations

required on `TuningValue` (e.g. comparison). See section 6.1 for further implementation details about `TuningValue`, and section 6.4.2 for further concerns about extensions.

5.4 Tuning Database

The purpose of a *tuning database* is to store tuning information as *training data*, and later retrieve them for training *models* which help to speed up the tuning of future programs. To our best knowledge, this is the first time that Periscope needs to save data to be reused in future runs, therefore we had to design and create a new piece of software architecture in Periscope to support this mode of behavior.

We have experimented with several ways of implementing a tuning database, and the different implementations are detailed in section 6.4. These implementations, however, are hidden behind an interface, represented by the C++ class `TuningDatabase`. In this section we describe the common design of data representation manifested in this interface, explain the functions of the interface and their pre- and post-conditions which are not obvious from the C++ code of the interface.

In section 4.2 we discussed that *training data* should be stored as $(\mathbf{p}, \mathbf{t}, s)$ tuples, where $\mathbf{p}$ denotes the *program features*, $\mathbf{t}$ denotes the *tuning parameter values*, and s means *speedup*. However, we wish to support *incremental* collection of training data, i.e. if we tune the same program two or more times, then we desire to know that the explored tuning configurations belong to the same program, thus each further tuning run potentially extends the explored part of the configuration space, instead of pretending to be a separate program with very similar program features and few training configurations. This is why we introduce the concept of **program identity**, a piece of information used to determine if two benchmark programs are indeed the same, or different. Using program identity, we decompose our $(\mathbf{p}, \mathbf{t}, s)$ tuples into two kinds of tuples:

- (*program identity*, *program features*),
- (*program identity*, *tuning configuration*, *execution time*).

For simplicity, we use execution time instead of speedup in the database. We also assume that if for a *program identity* there is an entry with tuning configuration and execution time, then there is also an entry with program features.

`TuningDatabase` has the following *virtual* functions for **storing** tuning information:

- `void saveSignature(ProgramID id, ProgramSignature signature);`

  Saves *program features* of a program identified by a *program identity*. Implementations can decide what to do exactly if that *program identity* already has *program features* assigned in the database.

- `void saveTuningCase(ProgramID id, TuningCase tc);`

  Saves a *tuning configuration* and the resulting *execution time* of a program identified by a *program identity*. Implementations can decide what to do exactly if that *program identity* and *tuning configuration* already have *execution time* assigned in the database. `TuningCase` is an alias (`typedef`) to `std::pair<TuningConfiguration, double>`.

- `void commit();`

  `saveSignature` and `saveTuningCase` should not change the state of the database immediately, but they are supposed to queue the changes to be *atomically* applied on the database when `commit()` is called.

`TuningDatabase` has the following *virtual* functions for **retrieving** tuning information:

- `Iterator<int> *queryPrograms();`

  Returns an iterator (see section 6.2) to *program identities* present in the database.

  What happens if we have different signatures for the same program

- `ProgramSignature querySignature(int id);` — different programs

  Returns the *program features* of the program identified by the given *program identity*.

  If the *program identity* was returned by a `queryPrograms()` call, then the user shall assume, that this function will succeed, i.e. *program features* were saved at least once for the specified program. If *program features* were saved multiple times, the database implementation should aggregate them in a sane way, but the exact way is left unspecified.

- `Iterator<TuningCase> *queryCases(int id);`

  Returns an iterator (see section 6.2) to saved *tuning cases* of the specified program in **increasing order** of execution time, i.e. best case comes first. The database implementation should take care not to return the same tuning configuration multiple times. If the same program with the same tuning configuration was measured multiple times, the implementation should aggregate the corresponding execution times in a sane way, but the exact way is left unspecified.

  `TuningCase` is an alias (`typedef`) to `std::pair<TuningConfiguration, double>`.

We also provide *helper* functions for two common patterns of **retrieval**. These functions are implemented using the *virtual* functions discussed above, therefore these functions need not

be implemented by the concrete implementations of the `TuningDatabase` interface.

- `TuningConfiguration queryBestConfiguration(int id);`
  Returns the best tuning configuration stored in the database for the given program, i.e. the tuning configuration with the smallest execution time.

- `std::vector<TuningConfiguration> queryConfigurationsByRatio(int id, double r);`
  Returns all tuning configurations whose corresponding execution time is not greater than the smallest execution time of the given program times the ratio `r`.
  For example, if the smallest execution time is 3.88 seconds, and `r` is 1.02 (2%), then those tuning configurations are returned whose corresponding execution time is at most 3.9576 seconds.

We must note that *program identity* representation is different in the functions *retrieving* data and in the functions *storing* data. For retrieval, we simply use integers to identify programs. For storage, however, we use a `ProgramID` object, wrapping some kind of identifying information which can be reproduced by Periscope. (For such an approach, see section 6.3.) `ProgramID` is determined by Periscope during tuning, and used by the database implementation to determine if it already has tuning information for that program. It is then the database implementation's responsibility to merge tuning information gathered in different runs of Periscope.

To provide access to the tuning database from anywhere in Periscope, we declare a pointer to a global tuning database instance in `TuningDatabase.h`:

```
extern TuningDatabase *tdb;
```

This way the above functions will be accessible from anywhere, and independently from the concrete implementations of the tuning database. One can, for example, get the list of programs in the tuning database by simply calling `tdb->queryPrograms()`.

5.5 Search Algorithms

This section summarizes the changes to *search algorithms*, and then goes into detailed discussion about *random sampling search*, a search algorithm we added to Periscope which is able to benefit from machine learning.

As we have already seen in Figure 5.3, a search algorithm must be able to receive *program features* together with the *search space* to benefit from machine learning. To support this requirement, the `ISearchAlgorithm` interface has been extended with a new virtual function:

```
virtual void addSearchSpaceWithSignature(SearchSpace *ss,
```

```
      ProgramSignature const& ignored) {
    addSearchSpace(ss);
}
```

This virtual function is supposed to be *overridden* by search algorithms which are able to benefit from machine learning. A default implementation is provided to prevent breaking existing search algorithms. The default implementation ignores the program features, and falls back on simply calling `addSearchSpace`.

5.5.1 Random Search – uniform sampling

Let $\mathcal{S}$ be a search space, consisting of tuning parameters $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$. Each tuning parameter $\mathcal{T}_i$ defines its set of possible values T_i. (T_i is always finite in Periscope.) Then $\mathcal{S}$ contains the following configurations:

$$S = T_1 \times T_2 \times T_3 \times \cdots \times T_k$$

where $\times$ denotes the Cartesian product of two sets, and the cardinality of S is:

$$|S| = |T_1| \cdot |T_2| \cdot |T_3| \cdot \cdots \cdot |T_k|$$

If k, the number of tuning parameters is large (e.g. often the case for compiler flags), $|S|$ can easily become astronomically large, making it practically impossible to run search algorithms such as *exhaustive search*.

Random Search samples the seach space N times, where N is a pre-configured constant. During every sampling step, the probability that each configuration $\mathbf{t} = (t_1, t_2, t_3, \ldots, t_k) \in S$ is chosen is exactly $\frac{1}{|S|}$.

Random Search with uniform sampling can be used to explore huge search spaces, which other methods fail to handle. With a relatively big N, such as $N = 1000$, it can be used to collect training data for configurations with many parameters. However, the real reason that we focus on this seach algorithm is that it can be configured to sample different *probability models*, not just a uniform distribution. Thus Random Search can benefit from machine learning, by predicting a probability model with the help of machine learning.

5.5.2 Random Search – sampling a Probability Model

Random Search can be parameterized by a *probability model*, or more precisely a *discrete probability distribution*, which can be defined, for example, by a *probability mass function*. $p : S \to [0, 1]$ is a probability mass function, if and only if $\sum_{\mathbf{t} \in S} p(\mathbf{t}) = 1$. Given such a distribution, Random Search samples a configuration $\mathbf{t}$ with $p(\mathbf{t})$ probability.

For the purpose of finding tuning configurations, we discuss the following types of probability models:

Uniform distribution

$p(\mathbf{t}) \equiv \frac{1}{|S|}$, for any $\mathbf{t}$. It assumes no knowledge of the search space. It is basically the same as what we discussed previously.

Implementation class: `UniformDistributionModel`

Independent parameters model

$p(\mathbf{t}) = p(t_1, t_2, t_3, \ldots, t_k) = p_1(t_1) \cdot p_2(t_2) \cdot p_3(t_3) \cdot \cdots \cdot p_k(t_k)$

where p_1, p_2, $\ldots$, p_k are probability mass functions. This model assumes that the effect of different parameters is independent. We know that this assumption does not hold in general, although this model could still be useful.

Implementation class: `IndependentParametersModel`

Markov model

We refer here to the model described in [12], which is able to capture simple interactions between different parameters. However, according to [12], this model achieves better results than the independent model only when a *lot* of training data is available, since the Markov model has a much higher degree of freedom, and thus it is more prone to have *high variance*.

(For n binary parameters, the independent model has $O(n)$ degree of freedom, while the Markov model has $O(n^2)$ degree of freedom.)

From the software architecture point of view, `RandomSearch` was designed in a way that most of the "logic" is delegated to the probability model, which is accessed via the `ProbabilityModel` interface:

```
class ProbabilityModel {
public:
  virtual ~ProbabilityModel() {}
  virtual TuningSpecification *sample() const = 0;
};
```

An instance of a subclass of `ProbabilityModel` shall be created from a `SearchSpace` (or equivalent) and potentially additional information specifying the probability distribution. As we discussed earlier, machine learning is applied on a *per* search space basis. That choice implies here, that for each search space we need one probability model instance. In case of

multiple search spaces, each probability model is sampled for every sample, and the returned `TuningSpecification`s are assembled into a single `Scenario` by `RandomSearch`. `RandomSearch` also takes care to remove duplicate tuning configurations. For details about duplication elimination, see section 6.5.

These ideas about probability models could apply to any search algorithm involving randomness. For example, choice of the *initial seed* for a *genetic algorithm* could also be "focused" this way, making the genetic algorithm benefit from machine learning.

5.6 Prediction of Probability Models

Search algorithms involving *randomness*, such as random sampling search, can use a *probability model* for sampling which can focus its search on certain areas of the *search space*. Our approach to speed up tuning is to use machine learning to predict probability models, which could help such search algorithms to find good configurations sooner. In this section, we present two ways of predicting probability models using machine learning.

why not to compute?

5.6.1 k-Nearest Neighbors

For a general introduction into the *k-nearest neighbors* algorithm, see section 4.1.1.

To predict a probability model, we look for the program in the tuning database which is the most similar to program being tuned (1-nearest neighbor). Then we retrieve the "good" tuning configurations for that program from the database, and use them to predict an *independent model*. Here we consider a tuning configuration "good" if its corresponding execution time at most 2% more than the execution time of the best configuration recorded in the tuning database. (This is a hard-coded, but easy to change definition.)

Since we use an independent model, we estimate the *probability mass function* for each tuning parameter individually. For each `TuningParameter`, we iterate through the range of values, and generate `TuningValue`s for each value in the range. For each `TuningValue` we count how many of the "good" `TuningConfiguration`s have that `TuningValue`. This way we get a *sample count* for each value of the tuning parameter. Then we add *smoothing* (each sample count is added the same value, by default 1), and after normalization we get the probability mass function for the tuning parameter. Repeating this process for each tuning parameter in the search space, we finally get the independent probability model.

A `TuningParameter` may fail to return a `TuningValue` object for a value in its range (see section 5.3.2). In this case we assign all those tuning configurations to this value, which do not

contain any of the remaining `TuningValues` generated by the `TuningParameter`. We assume that a `TuningParameter` fails to produce a `TuningValue` for at most one value only.

5.6.2 Linear SVM

For a general introduction into *Support Vector Machines* (SVM), see section 4.1.4.

This approach also predicts an *independent model*, but uses *per*-parameter *linear SVMs* to predict probability mass functions. To train a linear SVM, we need to set the *regularization parameter* C. To find a good value for C, we apply *cross-validation* (details below). If cross-validation finds out that SVM performs poorly for all values of C, then we fall back on using a *uniform* probability mass function for that tuning parameter.

As *training data* we use all programs with their best tuning configuration from the tuning database. Since we use SVMs on a *per*-parameter level, we extract from every tuning configuration the assigned value of the tuning parameter being learned. This may not be possible, if the `TuningParameter` never fails to generate a `TuningValue`, and none of the generated `TuningValues` appear in the tuning configuration. In this case we ignore that training program for learning this tuning parameter.

Cross validation. The approaches for *cross-validation* are very similar to the approaches for *testing*, but the goals are different. Cross-validation is used to tune the parameters for training (such as C for the linear SVM), while testing is the final evaluation about the goodness of the machine learning method. Ideally we need three separate datasets: a *training set*, a *cross-validation set* and a *testing set*. The training set and some *parameters* are used for training a model, but the parameters are tuned by evaluating these models on the cross-validation set. Once the parameters are fixed, final evaluation is done on the testing set. Often the cross-validation set is made by taking away elements from the (original) training set.

We use *leave-one-out* cross-validation for our linear SVM probability model prediction. That is we train SVM from all but one of the training examples, then we check what is the probability of correctly predicting the missing example. Once we have this probability value for each training example, we take their *arithmetic mean*[2]. This process is repeated for different values of C; we try values from the set $\{1, 3, 10, 30, 100, 300, 1000, 3000, 10000\}$. Then we choose C with the highest average probability of correct prediction. If none of the average probabilities is greater than $1/k$, k being the cardinality of the value range of the tuning parameter, then we abandon the idea of using SVM for the given tuning parameter, and fall back on using a *uniform* probability mass function, which correctly predicts with exactly $1/k$ probability.

[2]See section 9.1 for a discussion about alternatives.

Chapter 6

Implementation

This chapter serves to augment the previous chapter, *Design and Architecture*. Here all those ideas, discussions and decisions are included, which deserve to be documented, but do not belong to the *big picture*. Therefore its sections rather connect to the sections of the previous chapter than to each other.

6.1 Representation of tuning configurations – efficiency concerns

A `TuningValue` object represents a *tuning value*, i.e. a *tuning parameter* together with a certain assigned *value*. We gave a rationale for the introduction of `TuningValue`s in section 5.3.2; in this section we discuss an important practical concern about its implementation.

For machine learning, we often have to answer the question whether a `TuningConfiguration` has a certain `TuningValue`. While technically we only need an *equality operator* on `TuningValue` to answer this question, this is a frequent operation, and having merely an equality operator would not let us answer this question quickly, as we would need to use *linear search*, which has $O(n)$ expected running time. We considered two more efficient approaches:

- Provide a **hash function** on `TuningValue`. Let `TuningConfiguration` be a *hash table*, so checking whether it contains a certain `TuningValue` requires $O(1)$ running time in expected case, although the worst case might be as bad as $O(n)$.
- Provide a **comparison operator** on `TuningValue`. Let `TuningConfiguration` be a *balanced search tree* (`std::set` is usually implemented as a *red-black tree*), so checking whether it contains a certain `TuningValue` requires $O(\ln n)$ running time in worst case.

Since hash functions are harder to make right than comparison operators, especially concern-

ing the extension capability which `TuningValue` possesses, and balanced search trees provide *good enough* performance in most cases, we decided to implement a comparison operator on `TuningValue`, and to implement `TuningConfiguration` as `std::set<TuningValue>`.

Although `std::set` requires only a `<` operator, for simplicity and consistency we decided to implement a *three-way comparison*, and then to implement the full set of comparison operators in C++ (`==`, `!=`, `<`, `<=`, `>`, and `>=`) on top of the three-way comparison. A three-way comparison takes two values A and B belonging to a type with a total order and determines whether $A < B$, $A = B$, or $A > B$ in a single operation. For the `TuningValue` class, the `compare` method implements the three-way comparison, and returns a negative value for *less*, zero for *equal*, and a positive value for *greater*.

The `compare` method in `TuningValue` implements a *lexicographic order*, and members are compared in the following order: `pluginType`, `name`, `extension`, and finally `value`. However, `extension` is *polymorphic* and can also be `NULL`. In this case, for comparing extensions the following rules apply:

1. Two `NULL`s are considered *equal*, and `NULL` is always *less* than non-`NULL`.

2. If the pointed objects have different types, then their types are compared with the `before` method of the `type_info` class.

3. If the pointed objects are of the same type, then we use the *virtual* `compare` method of `TuningValueExtension`. This implies that the creator of a tuning value extension has to define a three-way comparison for the extension data.

6.2 Polymorphic Iterators

The *generic interface* `Iterator<T>` was introduced as the interface class of the *Iterator* object-oriented design pattern [20]. The C++ Standard Template Library defines several kinds of iterators, but they are all *template*-based iterators, while the one defined here is *polymorphic*[1], which implies the following advantages and disadvantages:

- *Worse run-time performance*, since every iteration step requires calling virtual functions.

- *Better flexibility* when modules are loaded dynamically. These iterators can transparently support traversing containers whose implementation is loaded dynamically during run-time, which is an integral part of Periscope's architecture.

[1] We usually mean *subtype polymorphism*, when we simply say polymorphic. Templates can be considered an example of *parametric polymorphism*.

The `Iterator<T>` interface defines the following virtual functions:

- `bool hasNext() const;`

  Returns true if there are further elements in the container.

- `T next();`

  Returns the next element, and advances the iterator to its next position. Undefined behavior if there are no more elements.

In most cases, one needs to apply the following code pattern to iterate through a container with these polymorphic iterators:

```
boost::scoped_ptr< Iterator<T> > it(pointer to iterator instance);
while (it->hasNext()) {
  T value = it->next();
  do something with value
}
```

Wrapping the iterator in a *scoped pointer* will automatically take care of the release of the iterator, since we have to allocate the iterator on the heap and access through a pointer because of polymorphism.

6.3 Program Identification

In section 5.4 we introduced the concept of *program identity* when we designed our database representation. In this section we are going to propose an approach to automatically recognize identical programs. To achieve that, we deterministically produce an identifier which does not change between different tuning runs for the same program, but is different for different programs.

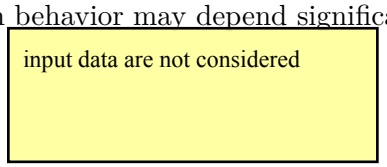
The approach we implemented is to use a *cryptographic hash function* (e.g. SHA1) on the source code of the application. Since we added machine learning only to the Compiler Flag Selection plugin, some of the considerations for this choice are CFS-specific:

- CFS requires the source code to be available, otherwise it makes no sense to tune compiler flags. Thus we can also rely on the presence of the source code for program identification.
- CFS has to know where the source files are to be able to recompile them. We use the same assumptions and same configurations about the location of source code as CFS does. (Except that CFS may use selective compilation, while for program identification we always use all of the source code.)

Calculating the SHA1 of all source files was also implemented in a similar manner to that of recompilation in CFS: a tiny shell script (`cfs_sha1.sh`) calculates the SHA1 checksum, and we simply read the standard output of this shell script.

Finally, let us provide a discussion of the advantages and disadvantages of this approach:

- *Automatic*, since there is no need to manually specify an identifier for the program.
- *Does not depend on compiler versions or compiler options*, since the checksum is calculated on the source code.
- *May discriminate too much*, since changes in whitespace or comments, or renaming variables result in identical (machine) code, but affect the checksum.
- *May discriminate too little*, since the program behavior may depend significantly on runtime options and other forms of input.



6.4 Tuning Database implementations

We introduced the *tuning database interface* in section 5.4, which also made coarse-grained assumptions about the database schema. To recall, we need to have at least the following "tables":

- (*program identity*, *program features*),
- (*program identity*, *tuning configuration*, *execution time*).

We also had two representations of *program identity*, integer numbers for retrieving tuning data, and `ProgramID` objects for saving tuning data. In the following sections we discuss different tuning database implementations which were created during the course of this thesis.

6.4.1 SQLite3 database back-end

SQLite3 [21] is an embedded *relational database management system* (RDBMS). An *embedded database* is available as a *software library*, and does not require running a separate server process. SQLite3 supports conventional relational database schemas, fully ACID-compliant, and implements most of the SQL standard. It features a C language API directly accessible from C and C++, and there are also bindings for other languages. There is also a command-line utility `sqlite3`, which is useful to manually examine and modify databases using the SQL *query language*. An SQLite3 database is stored in a single file.

We discuss first the database schema we use to store tuning data, and then we briefly explain a few details about the `SQLite3TuningDatabase` class, an implementation of the *tuning database interface*. We defined the following tables:

```
CREATE TABLE benchmarks (
    id INTEGER PRIMARY KEY,
    text TEXT UNIQUE
);
```

The `benchmarks` table is primarily used to assign IDs to any *benchmark program* we encounter. The `text` column is added to save the content of a `ProgramID` object, basically the SHA1 checksum (see section 6.3). Thus when we save tuning data, we look for the `text` first to check whether we already have an ID for the program. If yes, we use that ID, otherwise we create a new entry with a unique ID and we also save the SHA1 checksum to the `text` column.

```
CREATE TABLE counters (
    id INTEGER NOT NULL,
    name TEXT NOT NULL,
    value INTEGER NOT NULL,
    FOREIGN KEY(id) REFERENCES benchmarks(id)
);
```

The `counters` table stores the PAPI hardware event counters, which we use as input features for machine learning. `id` is a *foreign key* which references the *primary key* in the *benchmarks* table. Basically it identifies which benchmark program the entry belongs to. `name` is the PAPI name of the event counter, such as `PAPI_FP_OPS`, and `value` is the corresponding counter value. We can have several entries with different `value`s but the same `id` and `name`. This happens when we tune the same program multiple times, and the program features are collected and saved during each tuning run. We will take care to aggregate `value`s when retrieving data.

```
CREATE TABLE measurements (
    id INTEGER PRIMARY KEY,
    bid INTEGER NOT NULL,
    time REAL NOT NULL,
    FOREIGN KEY(bid) REFERENCES benchmarks(id)
);
```

The `measurements` table contains informations about each tuning measurement. Since a tuning configuration may contain arbitrary number of tuning values, they are delegated into the

`configurations` table. Here we only assign an `id` to each measurement, which will be referred from `configurations`. `bid` is the *benchmark ID*, identifying the benchmark program itself, and `time` is the execution time.

```
CREATE TABLE configurations (
    mid INTEGER NOT NULL,
    plugintype INTEGER NOT NULL,
    name TEXT,
    value INTEGER,
    FOREIGN KEY(mid) REFERENCES measurements(id),
    UNIQUE(mid, plugintype, name)
);
```

The `configurations` table augments the `measurements` table, containing the tuning configurations, one tuning value *per* row. `plugintype`, `name` and `value` describe the tuning value (without extensions, see section 5.3.2). `mid`, or *measurement ID*, identifies the measurement from the `measurements` table. It is okay to have multiple measurements of a benchmark program with the same configuration; we will aggregate the running times during retrieval.

Saving tuning data is initiated through the `saveSignature` and the `saveTuningCase` virtual functions. The following points apply to both:

- Calls `disableAutocommit()` which checks whether the database is in auto-commit mode, and if yes, it starts a transaction with the SQL command `BEGIN`.
  The transaction ends when `commit()` is called, which issues the SQL command `COMMIT`.
- Receives a `ProgramID` which must be converted to the *benchmark ID* internally used by the database, potentially inserting a new row into the `benchmarks` table. (`selectBenchmark` and `insertBenchmark` serve as helper functions.)

Let us have a look at the unique parts now:

- `void saveSignature(ProgramID id, ProgramSignature signature);`
  Inserts each PAPI counter from `signature` one by one using the `insertCountersEntry` helper function.
- `void saveTuningCase(ProgramID id, TuningCase tc);`
  First inserts a new row into `measurements` using `insertMeasurementEntry`, then inserts each tuning value from `tc.first` into `configurations` using `insertTuningValueEntry`.

Retrieving tuning data is a bit more complicated, since the aggregation of repeated measurements occurs during retrieval. For retrieval, we use the same identifiers as the `id` column in the `benchmarks` table. Let us discuss the retrieving functions one by one:

- `Iterator<int> *queryPrograms();`

  Prepares a query selecting the `id` column from `benchmarks`, then constructs and returns a `BenchmarkIterator` (implements `Iterator<int>`), which iterates through the rows of the query result, returning `id`s.

- `ProgramSignature querySignature(int id);`

  Executes the following query:

  ```
  SELECT name, CAST(AVG(value) AS INTEGER)
  FROM counters
  WHERE id = ?
  GROUP BY name;
  ```

  then assembles the `ProgramSignature` object to return from the query result. Aggregation is handled by SQL with the `GROUP BY` clause and the `AVG` (average) aggregating function.

- `Iterator<TuningCase> *queryCases(int id);`

  Similar to `queryPrograms()`, it also prepares a query and constructs a `CaseIterator` (implements `Iterator<TuningCase>`) which iterates through tuning configurations in *increasing order* of execution time (requirement of the tuning database interface, see section 5.4). Ordering is handled by SQL with the `ORDER BY` clause. However, aggregation of different measurements with the same tuning configuration is handled at the C++ side. For each tuning configuration we consider the measurement with the best execution time only. (In other words, the aggregation function is the *minimum* function.) This is implemented by the iterator class `RemoveDuplicates` (implements `Iterator<TuningCase>`) which decorates the `CaseIterator` by skipping those tuning cases whose tuning configuration has already been encountered. Since `CaseIterator` gives tuning cases in the increasing order of execution time, we end up keeping just the best execution time for each distinct tuning configuration.

6.4.2 JSON dump and loading from JSON

In this section we discuss saving tuning data to regular files in a JSON format, and loading tuning data from files in a JSON format. This is *not a full tuning database* implementation, but serves *interoperability* purposes with external processing tools.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is similar to XML in that respect that it defines a *notation* only, and users can decide themselves how they want to structure their data. Compared to XML, JSON has less *syntactic noise*, making it easier for humans to read and write, and JSON is *native* to one programming language (JavaScript), while XML is native to *none*.

JSON features atomic values, such as numbers (`-12`, `3.14`), strings (`"foo"`), logical values (`true`, `false`) and `null`, and compound values: *arrays* and *objects*. JSON arrays are a comma-separated list of JSON values in square brackets, e.g. `[12, "foo", true]`. JSON objects are basically a collection of name-value pairs. Syntactically they are a comma-separated list of name-value pairs in curly braces; each name-value pair is a string (name), a colon, and a JSON value, in order. For example: `{"spam": 3.16, "egg": true}`.

Let us introduce now the structure of the input and output formats. Our description uses *production rules*, similar to the *Backus-Naur Form*. *Non-terminal* nodes are surrounded with angle brackets, and `::=` denotes a rule definition. Type names prefixed with `@` refer to atomic JSON values of that type; `null` is included if the type name has a `?` suffix added. When `...` follows a reference to a non-terminal, that means a comma-separated list of that non-terminal. With this notation, the structure of the output format can be defined as follows:

```
<output-format> ::= {
  "signatures": [ <signature-record> ... ],
  "tuningcases": [ <tuning-case-record> ... ]
}
<signature-record> ::= {
  "id": @string,
  "signature": { <signature-value> ... }
}
<signature-value> ::= @string: @integer
<tuning-case-record> ::= {
  "id": @string,
  "configuration": [ <tuning-value> ... ],
  "exectime": @real
}
<tuning-value> ::= {
  "type": "generic",
  "plugintype": @integer,
```

```
  "name": @string,
  "value": @integer?
}
```

We provide an example output below to help understanding. The listing of PAPI counters and values is shortened to preserve space. Two tuning cases were measured: execution time is 1.22 seconds when the application is compiled with `-O2`, and 1.19 seconds when it is compiled with `-O3`.

```
{
  "signatures": [
    "id": "da39a3ee5e6b4b0d3255bfef95601890afd80709",
    "signature": {
      "PAPI_BR_CN": 708524110,
      "PAPI_BR_INS": 783021936,
      "PAPI_BR_MSP": 711976,
      ...
      "PAPI_TOT_CYC": 14693050458,
      "PAPI_TOT_INS": 32672706096
    }
  ],
  "tuningcases": [
    {
      "id": "da39a3ee5e6b4b0d3255bfef95601890afd80709",
      "configuration": [
        {
          "type": "generic",
          "plugintype": 1,
          "name": "-O2",
          "value": null
        }
      ],
      "exectime": 1.22
    },
    {
      "id": "da39a3ee5e6b4b0d3255bfef95601890afd80709",
```

```
      "configuration": [
        {
          "type": "generic",
          "plugintype": 1,
          "name": "-O2",
          "value": null
        }
      ],
      "exectime": 1.19
    }
  ]
}
```

We now define the input format, which is very similar to the output format, but lacks the SHA1 checksums and tightly couples the PAPI counters to the tuning cases.

```
<input-format> ::= [ <input-record> ... ]
<input-record> ::= {
  "signature": { <signature-value> ... },
  "tuningcases": [ <tuning-case> ... ]
}
<tuning-case> ::= {
  "configuration": [ <tuning-value> ... ],
  "exectime": @real
}
```

Here we provide an example for the input format. This example has the same data as the example for the output format. In practice, several different output files (tuning measurements) should be processed and merged into a single input file (tuning database).

```
[
  {
    "signature": {
      "PAPI_BR_CN": 708524110,
      "PAPI_BR_INS": 783021936,
      "PAPI_BR_MSP": 711976,
```

```
      ...
      "PAPI_TOT_CYC": 14693050458,
      "PAPI_TOT_INS": 32672706096
    },
    "tuningcases": [
      {
        "configuration": [
          {
            "type": "generic",
            "plugintype": 1,
            "name": "-O2",
            "value": null
          }
        ],
        "exectime": 1.22
      },
      {
        "configuration": [
          {
            "type": "generic",
            "plugintype": 1,
            "name": "-O2",
            "value": null
          }
        ],
        "exectime": 1.19
      }
    ]
  }
]
```

Since the JSON file formats closely follow the tuning database interface, the implementation is fairly straightforward. The implementing class is `JsonTuningDatabase`. We use *property trees* from Boost to read and write JSON formatted files. We added `toPtree` and `fromPtree` methods to affected class, such as `TuningValue`, `TuningConfiguration` and `ProgramSignature`.

These methods are, however, tricky in the case of `TuningValue`, which may have a polymorphic extension. To create property trees, we added a virtual `toPtree` method to the `TuningValueExtension` interface. This virtual method is called by the `toPtree` method of `TuningValue`, so the extension values will be added to the property tree.

Parsing property trees into `TuningValue`s with polymorphic extensions, however, is more complicated, since we cannot know ahead the type of the extension. This is why we introduced the `type` field in `<tuning-value>`; `type` shall be set to `"generic"`, when there are no extensions, and each extension type shall define its unique identifier, and set it from the `toPtree` method of its class. For parsing, the parser functions shall be registered with their corresponding `type` string, using the `registerTuningValueParser` function. For easier registration, we also provide the helper type `TuningValueParserRegistrator` which call the registration function from its constructor. Therefore, if we have a `CFSTuningValueExtension` concrete type implementing the `TuningValueExtension` interface, we can implement a parser and register it with the type identifier string `"compilerflag"` as follows:

```
TuningValueExtension *
CFSTuningValueExtensionFromPtree(ptree const& pt) {
  FooTuningValueExtension *result = new CFSTuningValueExtension;
  /* read extension values from pt to result */
  return result;
}

static TuningValueParserRegistrator
CFSTuningValueRegistrator("compilerflag",
                          &CFSTuningValueExtensionFromPtree);
```

When we parse a `TuningValue`, we simply check whether `type` is `"generic"`. If it is not, we lookup the specific registered parser, and call that to parse the extension fields.

6.4.3 Dummy back-end

The *dummy back-end* is the earliest tuning database "implementation". It does not really implement a tuning database, but it satisfies the `TuningDatabase` interface and it can be used as a dummy substitute to test other components of the machine learning architecture (see Figure 5.3), such as adding machine learning support for a plugin, or the usage of fake tuning data in a search algorithm.

For "saving" tuning data, the current version relies on the architecture created for the JSON

tuning database (section 6.4.2). `saveSignature` and `saveTuningCase` dump data directly to the *standard error* stream in the JSON format. `commit` does nothing.

The query functions simply return hard-coded fake tuning data. The current version has an *empty* database hard-coded.

6.5 Duplicate Elimination in `RandomSearch`

`RandomSearch` samples the probability model N times – we call N the *sample count*. There is a chance that some of these samples yield the exact same configuration. Running a measurement with the same configuration over and over is undesirable, as it provides little new information, but may require significant amount of computational resources. Therefore, after the sampling of each list of `TuningSpecification`s, we check whether we have already sampled an equivalent list of `TuningSpecification`s. If not, we add the new scenario.

Merely applying duplication elimination, however, can have undesired consequences. We asked `RandomSearch` to generate N scenarios, but it might end up generating less than N scenarios. We might use a while-loop to generate scenarios until we have N unique ones, but that approach would also have problems. First, it would not terminate if N is greater than $|S|$, the size of the search space. Even if we included a special case to fall back on *exhaustive search* is $N > |S|$, scenario generation time could still blow up. Let assume that the search space is huge, and N is close to, but not greater than $|S|$. In this case the last few scenarios would take really long to generate, since at that point most scenarios in S would have already been generated, thus there is a very low probability that a randomly sampled scenario is new.

Considering all these approaches and consequences, we decided to adopt the scheme: the probability models are sampled at most $3N$ times, but we immediately stop once we have N unique scenarios. This reduces the chance that we end up with less than N scenarios, but keeps the scenario generation time predictable. Although we may still end up with a number of scenarios which is less than both N and the size of the search space, we chose to keep the implementation simple.

6.6 Compiler Flag Selection plugin related changes

This section discusses the details about the changes made to the Compiler Flag Selection plugin. Some of these changes were already discussed from a general perspective, we will just briefly mention them here. Other changes, or at least some details of them, however, are rather CFS-specific – they are thoroughly discussed here.

6.6.1 `CFSTuningParameter` and its mapping to `TuningValue`

In this section we discuss the changes made to, or related to `CFSTuningParameter`, a subclass of `TuningParameter` which is used to describe *compiler flag* parameter options.

`CFSTuningParameter` extends `TuningParameter` with two data members: `flagString` – prefix part of the compiler flag; and `valueStrings` – vector of values for the compiler flag. How these additional data and the data inherited from `TuningParameter` are used, branches into two cases, depending on whether the values of the tuning parameters are strings or integers. CFS tuning parameters (compiler flags) are defined in the CFS configuration file. Let us explain two examples:

- `tp "TP_IFORT_OPT" = "-" ["O2", "O3", "O4"];`

  The `tp` keyword denotes a *tuning parameter* definition in the configuration file. Its first argument is the *name* of the tuning parameter – this string is written to the `name` field inherited from `TuningParameter`. The equal sign which follows it, is simply a syntactical element. Then comes the *prefix part* of the compiler flag – this goes to `flagString`. The last argument before the semicolon is the *list of values*. In this example, we have got a list of *string values* – they go to `valueStrings`.

  The `ID` field (inherited from `TuningParameter`) is assigned integers *sequentially* by the CFS plugin, starting from 0, in the order as they appear in the configuration file. Therefore the `ID` can discriminate between different compiler flags in a single tuning run, but it is quite useless across different runs, as it depends heavily on the configuration file.

  `from`, `to` and `step` (all inherited from `TuningParameter`) are set to 1, n and 1, respectively, where n is the number of values defined. This means, that the allowed values for this tuning parameter are 1, 2 and 3, and they correspond to the compiler flags `-O2`, `-O3` and `-O4`, respectively.

- `tp "TP_UNROLLN" = "-unroll=" [5,10,5];`

  For most part, this example is the same as the previous. The difference is in the last argument: in this case we have got an *integer range* as a list of values, while in the previous example we had a list of strings. These values are written to `from`, `to` and `step`, while `valueStrings` remains an *empty vector*. The third number (step) can be omitted – in that case `step` will get value 1.

  So, for this example, the allowed values are 5 and 10, and they correspond to the compiler flags `-unroll=5` and `-unroll=10`.

The `getAFLAGS` function of the `CompilerFlagsPlugin` had originally contained the logic

to turn `TuningParameter` and *value* pairs into compiler flag strings. This functionality was then refactored into the `CFSTuningParameter` class, adding a new method `getFlagWithValue`, which receives the *value* assigned to the tuning parameter as an argument, and returns a string representing the compiler flag. This way only the `CFSTuningParameter` class has to know that it has two cases (string values or integers values).

For the purpose of machine learning, we need to generate a `TuningValue` object from a `CFSTuningParameter` object and its assigned value. We can see here a pattern similar to that of section 5.3.2. `CFSTuningParameter` includes the whole range of possible values, while what we need in a `TuningValue` is just the selected value. Depending on whether the compiler flag has integer or string values, we represent compiler flags as `TuningValue` objects as follows:

String values

First, from `valueStrings` we need only the selected value. Second, we can readily concatenate the *prefix part* and the selected *value string*. This gives us more flexibility for machine learning, as it provides equivalent set of tuning values for the following CFS tuning parameter definitions:

- `tp "TP_IFORT_OPT" = "-" ["O2", "O3", "O4"];`
- `tp "TP_IFORT_OPT" = "-O" ["2", "3", "4"];`

Third, we also trim (remove leading and trailing whitespace from) the resulting *compiler flag string*. We might need an extension to store this compiler flag string.
`TuningValue` however has an integer `value` field, which must have a value, and that value matters during comparison. For this purpose, we introduced `TuningValue::NULL_VALUE`, a value used to note that `value` has no value. (`TuningValue::NULL_VALUE` is chosen to be `INT_MIN`. We know no cases of a `TuningParameter` having negative values in its range, so this choice should be safe.)
CFS tuning parameters are often defined to either include a certain compiler flag or have no effect, for example:

- `tp "TP_IFORT_XHOST" = " " ["-xHost", " "];`

This tuning parameter either adds the `-xHost` compiler flag, or adds no compiler flags. If after concatenation and trimming we end up with and *empty string* – this happens for the second value of the above example –, then we will fail to return a `TuningValue`, returning `boost::none` from `getTuningValue`. (In fact, we made the return value of `getTuningValue` optional in the `TuningValue` class exactly to support this case here.)

Integer values

When a CFS tuning parameter has integer values, then we use the `value` of `TuningValue` to store the assigned value as usual. We store the *trimmed* prefix part in the same field where we store the whole compiler flag string in case of string values. This makes it easy to recognize whether a `TuningValue` encodes a CFS tuning value with string or integer values – one just has to check whether `value` is equal to `TuningValue::NULL_VALUE`.

One might argue that it would be more flexible to convert the integer value to a string, and append it to the prefix part as in the case of string values. After all, that would generate the same set of `TuningValue`s for the following two CFS parameter definitions (which indeed behave the same way):

- `tp "TP_IFORT_OPT" = "-" ["O2", "O3", "O4"];`
- `tp "TP_IFORT_OPT" = "-O" [2, 4];`

On the other hand, we might wish to model the prediction of compiler flags with integer values as *regression problems*, and we need to keep these two cases separate to be able to do that. (Currently we use *multi-class classification* in all cases, but *regression* could be worthwhile to try for this case in the future.)

Initially we created a tuning value extension to store the string data of the CFS tuning value (the whole compiler flag string for tuning parameters with string values, or the trimmed prefix part for tuning parameters with integer values). However, to achieve more flexibility and to reduce dependency on the CFS configuration file, we decided to set the `name` field of `TuningValue` to empty string, irrespective of what was written in the configuration file. Since we had had an unused slot for storing strings, we decided to drop the extension, and put the string data of the CFS tuning value directly into the `name` field.

Finally, we summarize the behavior of `getTuningValue` of the `CFSTuningParameter` class in the following points:

- If both the prefix part and the value part is whitespace, no `TuningValue` is returned (returns `boost::none`).
- Otherwise, we return a `TuningValue` object with `pluginType` set to `CFS`, and `extension` set to `NULL`.
- For tuning parameters *with string values*, `name` is the trimmed concatenation of the prefix part and the value string, while `value` is `TuningValue::NULL_VALUE`.

- For tuning parameters *with integer values*, `name` is the trimmed prefix part, and `value` is simply the assigned value.

6.6.2 Machine Learning support

In this section we discuss the implementation details of adding machine learning support to the Compiler Flag Selection plugin. This includes the following changes:

- **Collecting *program features***, chosen as PAPI hardware event counters in section 4.2, collecting them as described in general case in section 5.2. Further details about refactoring and conflicting analysis requirements are discussed below.

- **Determining *program identification* information**, as anticipated in section 5.4 and discussed in section 6.3.

- **Passing *program features* to the *search algorithm*** to be able to benefit from machine learning, using the new method in `ISearchAlgorithm`, `addSearchSpaceWithSignature` (see section 5.5).

- **Saving new tuning data to the *tuning database*** when search is finished, calling the save functions of `TuningDatabase` (section 5.4) in step "Finalize Tuning" (see Figure 5.4, implemented by the function `finalize`).

Multi-step search algorithm issue

We already noted in section 5.2, that program features are available only once we have reached the "Create Scenarios" step, but "Create Scenarios" is executed multiple times when a multi-step search algorithm is used, such as *individual search*. Originally the search space was added to the search algorithm in "Start Tuning Step". Since we need to add the search space together with the program features, we had to delay this operation until "Create Scenarios", however, this led to erroneous behavior with the individual search, as search spaces were added again and again in each step during the search.

We solved this issue by refactoring the CFS plugin, creating a new private member function `processAnalysisResult`. We moved that code into this function, which processes the analysis result and adds the search space to the search algorithm. This function is then called from `createScenarios`, but using a helper variable we make sure `processAnalysisResult` is called only when `createScenarios` is called for the first time.

Conflicting pre-analysis request

In some cases the CFS plugin requests a *conflicting pre-analysis*, which cannot be done or requested together with the collection of PAPI event counters. CFS supports *selective compilation*, that is compiling only those files which contain *hot code*. The purpose of this functionality is to reduce compilation time, and thus speed up tuning. When selective compilation is enabled in the CFS configuration file, but the hot files are not specified, CFS uses a pre-analysis step to determine hot functions and hot files. Although the conflict could have been solved by using multiple *tuning steps* – each one might have its own pre-analysis step –, this approach would have added significant complexity to the plugin implementation. Therefore we decided to *silently disable* selective compilation *if* it would require an pre-analysis step *and* machine learning is enabled.

Compiler flags for collecting PAPI event counters

Since we are tuning compiler flags, it is an interesting question how do they affect our program features, and what compiler flags shall we use for collecting the program features. Since compiler flags affect the code generation, they also affect the PAPI event counter values, which could mislead our learning process which uses them as the input features for machine learning. Consequently, every time we collect PAPI counters, we first recompile the program with a predefined configuration. Cavazos et al. [14] use `-O0` for this purpose. Our intuition suggests to apply all universally beneficial and hardware-independent optimizations (not to differentiate among practically equivalent programs), but apply none of the hardware-dependent optimizations (not to make our input features hardware-dependent). We ran a few experiments comparing the PAPI counters for the same program, but compiled with different optimization levels. We found that `-O0` significantly changes the characteristics of the program by not applying any optimizations, while the other optimizations levels yield indeed quite similar numbers. *We picked `-O1`* for collecting PAPI counters.

Chapter 7

Usage

This section explains how to use machine learning with Periscope. First section briefly discusses the installation of Periscope, with emphasis on the requirements for machine learning. For the general usage of the CFS plugin, please refer to the document *"PTF CFS Plugin – User's Guide"*. Second section explains how to use machine learning with the CFS plugin.

7.1 Installation Requirements

Installation of the Periscope Tuning Framework is briefly summarized in three steps below. We include this section here, because there are *build time requirements* for machine learning.

1. If you use a *development version* of Periscope from the Mercurial repository, you first need to bootstrap the `configure` script. If you use a *release distribution*, skip to step 2.
   To bootstrap, you need `autoconf`, `automake`, `m4` and `libtool`. At the time of writing this thesis, the following versions are known to work: `autoconf 2.69`, `automake 1.12.4`, `m4 1.4.17`, `libtool 2.4.2`. Change directory to the root of Periscope's source tree, and then issue:
   `./bootstrap`

2. Before you configure, make sure you have all dependencies installed. Periscope has the following *required* dependencies: ACE, Boost, Xerces-C++. For machine learning, the PAPI library is also *strictly required*. If you wish to use the SQLite3 database back-end, then you will also need the SQLite3 library. Configure with:
   `./configure --enable-papi --enable-sqlite3`
   You may skip SQLite3, and then you will not have that tuning database back-end. You may also add `--prefix=`*location*, if you wish to install Periscope to a location other than

the default (`/usr/local`). You can also call the `configure` script from a different directory if you want an *out-of-source* build. Make sure that everything has gone well by looking for the following snippets in `configure`'s output:

```
PAPI enabled: yes
PAPI support: yes
PAPI compile flags:
PAPI link flags: -lpapi -lpfm
```

. . . and also:

```
SQLite3 enabled: yes
SQLite3 support: yes
SQLite3 compile flags:
SQLite3 link flags: -lsqlite3
```

3. To build, issue:
   `make`
   To install, issue:
   `make install`
   To run the test suites, issue:
   `make check`

Summary: Make sure to enable PAPI during configuration. To be able to use the SQLite3 database backend, you need the SQLite3 library as a dependency, and you need to explicitly enable it during configuration.

7.2 Machine Learning with the CFS plugin

For general instructions about the usage of the CFS plugin, please refer to *"PTF CFS Plugin – User's Guide"*. This section assumes basic understanding of the usage of the CFS plugin, and discusses only the usage of machine learning.

To enable machine learning, the following line must be added to the configuration file:

```
machine_learning="true";
```

Using `"false"` instead of `"true"` will explicitly disable machine learning, although that is the default behavior. Given that Periscope was installed with PAPI enabled, CFS will first recompile the application with `-O1`, then collect the PAPI performance counters. When tuning is finished, CFS saves the tuning data through the *tuning database* interface. Depending on the actual implementation behind the interface, the following things could happen:

- If Periscope was built *with* SQLite3 support, then it is hard-coded to use the *SQLite3-based tuning database* (class `Sqlite3TuningDatabase`). It is also hard-coded that the database is the `tuning.db` file (in the current directory). If this file does not exist, then Periscope will automatically create an empty database with this name and with the correct schema. After the tuning session, the user can investigate the content of the database file with the `sqlite3` command-line utility.

- If Periscope was built *without* SQLite3 support, then it is hard-coded to use the *JSON dumper and loader* (class `JsonTuningDatabase`). It is also hard-coded that previous tuning is loaded from the `database.json` file (in the current directory), or Periscope assumes an empty database if this file does not exist. Again, it is hard-coded that tuning data from the current tuning session are saved into the file `measurements.`*time*`.json`, where *time* is the return value of the `time` function, that is the number of seconds since the Epoch (beginning of 1970 in UTC). *time* is added to the filename to prevent overwriting previous data. The time resolution of 1 second should be satisfactory, since Periscope never finishes tuning successfully in less than a second.

- Periscope also contains a `DummyTuningDatabase` class, which is never used. If it were used, it would dump tuning data in a partially JSON format directly to the screen.

Now that we have the training data, we must make it accessible to Periscope before we could use it for machine learning. Depending on the database implementation, this means the following:

- **SQLite3 back-end:** Nothing to do, except for maybe moving the `tuning.db` file to make sure it is still in the current directory.

- **JSON dumper and loader:** The individual `measurements.`*time*`.json` files must be processed and merged into a single `database.json` file. For the format of these files, see section 6.4.2. Also see section 5.4 which contains the assumptions which the JSON database must fulfill as well, primarily about aggregation of different measurements of the same thing. This processing and merging requires external tools, which are not provided.

Once training data is accessible through the tuning database interface, the next step is to use a search algorithm which supports machine learning. At the time of writing this thesis, this choice is easy: *random sampling search* is the only search algorithm in Periscope with machine learning support. In case of the CFS plugin, random sampling search can be selected by having the following line in the configuration file:

```
search_algorithm="random";
```

We probably wish to set how many samples we would like to measure while searching. This is possible via the following CFS configuration file option:

```
sample_count=k;
```

where k is the desired number of samples. (The number of actual samples might be less, see section 6.5 for details.)

From this point on, there is nothing else to do. It is hard-coded into `RandomSearch` to use linear SVMs to predict an independent model (see section 5.6).

7.3 Further notes

If you wish to use both *machine learning* and *selective compilation*, you need to give the list of files to recompile in the CFS configuration file. Since machine learning "occupies" the pre-analysis stage, the profiling analysis to determine the hot files is not run when machine learning is enabled. If the analysis is necessary for selective compilation, then selective compilation will be silently disabled.

Chapter 8

Evaluation

In this chapter we experimentally evaluate our contribution to the Periscope Tuning Framework. First we consider the accuracy and precision of our measurements, then we carry out several experiments to gain insight into the benefits of the implemented machine learning approaches on tuning speed and result.

8.1 Measurement Precision – SuperMUC

Before we have a look at measurement data from actual evaluation experiments, we must have a clue about how accurate our measurements are. For this purpose, we prepared an experiment and executed it over and over again. In a perfect world with absolutely accurate and precise measurements, we shall get exactly the same result every single time for this experiment. However, in the real world, there will be some variance, and the statistical evaluation of the data helps us understand the nature of measurement errors.

Figure 8.1 shows the graph of 256 measurements of the `BT` benchmark program from the *NAS Parallel Benchmarks* [1]. Many measurement values lie on the same line, but there are also a lot of "spikes", apparently outlier values. Statistical evaluation shows a wide value range and high variance for both configurations. However, if we eliminate outliers, re-evaluate the remaining data, and we repeat these two steps until convergence, then we finally get the following results:

- 23% of measurement values is considered *outlier*, which is a fairly high rate.

- The mean and standard deviation of the remaining data are $3.7969 \pm 0.11\%$ seconds for `-O2 -xHost`, and $3.7981 \pm 0.10\%$ seconds for `-O3`. Note that after outlier elimination there is *very low standard deviation*.

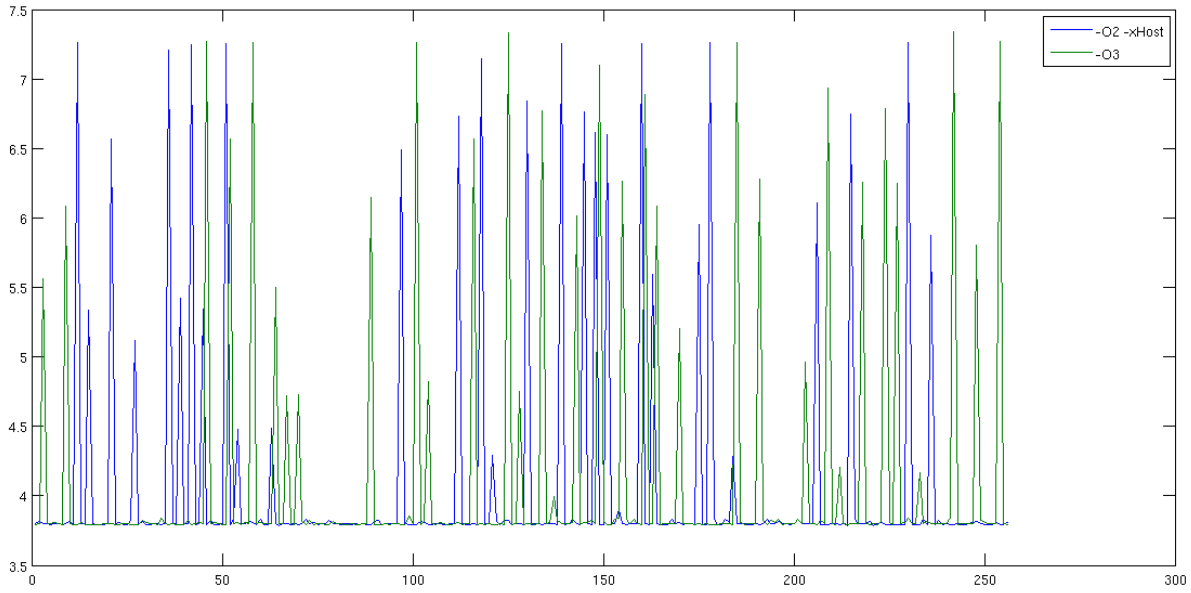


Figure 8.1: `BT` benchmark evaluated with `-O2 -xHost` and `-O3` compiler flags, each measured 256 times. *Vertical axis:* execution time in seconds. *Horizontal axis:* the individual measurements.

8.2 Experimental Results

For evaluation, we used 9 benchmark programs from the *NAS Parallel Benchmarks* [1]: `BT`, `CG`, `FT`, `IS`, `LU`, `LU-HP`, `MG`, `SP`, and `UA`. To collect training data, we ran *exhaustive search* on all of these with following the configuration:

```
tp "TP_IFORT_OPT" = "-" ["O2", "O3", "O4"];
tp "TP_IFORT_XHOST"  = " " ["-xhost", " "];
tp "TP_IFORT_UNROLL" = " " ["-unroll", " "];
tp "TP_IFORT_PREFETCH" = " " ["-opt-prefetch", " "];
tp "TP_IFORT_IP" = " " ["-ip -ipo", " "];
tp "Unroll" = "-" ["unroll-aggressive", "no-unroll-aggressive"];
tp "Unrolln" = "-unroll=" [5,10,5];
```

How are outliers handled here

This sums up to $9 \cdot (3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2) = 9 \cdot 192 = 1728$ measurements.

In order to have a reference case for a machine learning predicted probability models, we first tested *random sampling search* on these programs with *uniform distribution*, without any machine learning. Results are shown on Figure 8.2. We see that for `IS` the first random guess outperforms exhaustive search by more than 10% – a theoretically impossible result, which is a pretty sure sign of a measurement error. Although initially the execution time is often quite far

from the optimal, after 8 samples (of the 192 cases) all but one are within 3% of the optimal execution time – `FT` does not go below 10%.

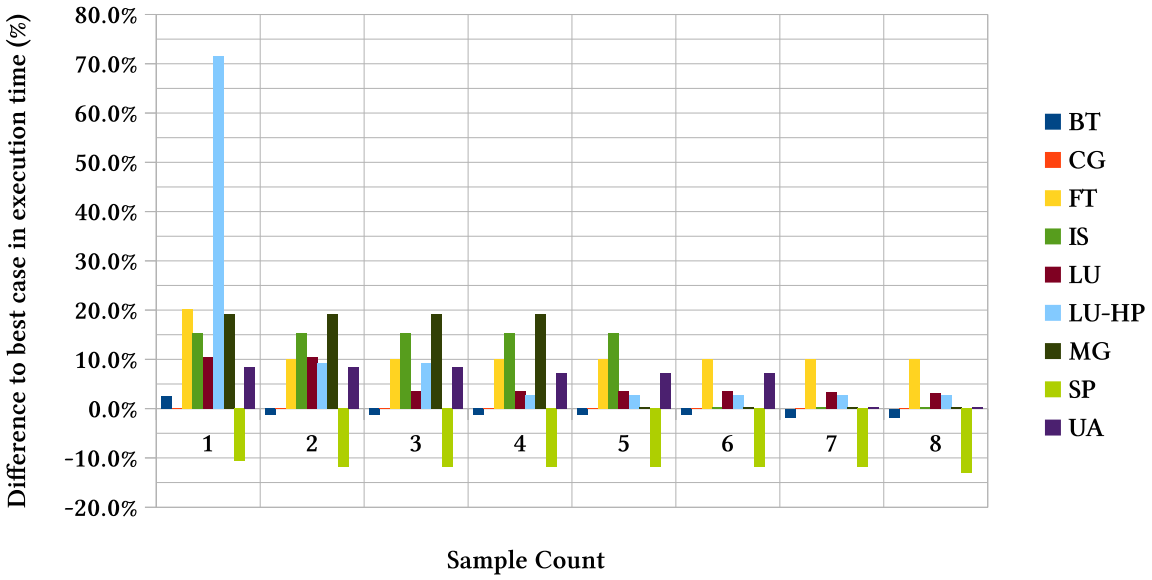


Figure 8.2: Execution time of *uniform sampling* relative to the minimum execution time found by exhaustive search. Bars show the *cumulative minimum* along the horizontal axis, i.e. the ith set of bars shows the best execution time found after i samples.

Figure 8.3 shows the evaluation of *1-nearest neighbor* on the *training data*. Given that the collection of PAPI hardware event counters is reliable, the nearest neighbor will always be the same program, so this approach could basically retrieve the previously found solution from the database. Therefore it rather serves to check whether the implementation suffers from serious defects, than to evaluate the method itself. Since it learns parameters independently from the tuning cases in database within 2% of the best execution time, it may not found the best solution in one shot, but it is never very far from it, and eventually gets very close.

We did the same experiment for the *linear SVM* predictor, see Figure 8.4. Although several benchmark programs have very good execution even for the first try, for a few it is not as good as it was with *1-nearest neighbor*. That is probably because with SVM we use *regularization*, which gives up accuracy on the training set to achieve better accuracy on *unseen* data.

Figure 8.5 finally shows an evaluation of the goodness of our machine learning approach. Here we have a test set which is separate from the training set. Due to the smallness of our dataset, we use a *leave-one-out* approach, i.e. the test set has only one element, while all the other elements form the training set. This is repeated such that each benchmark program is

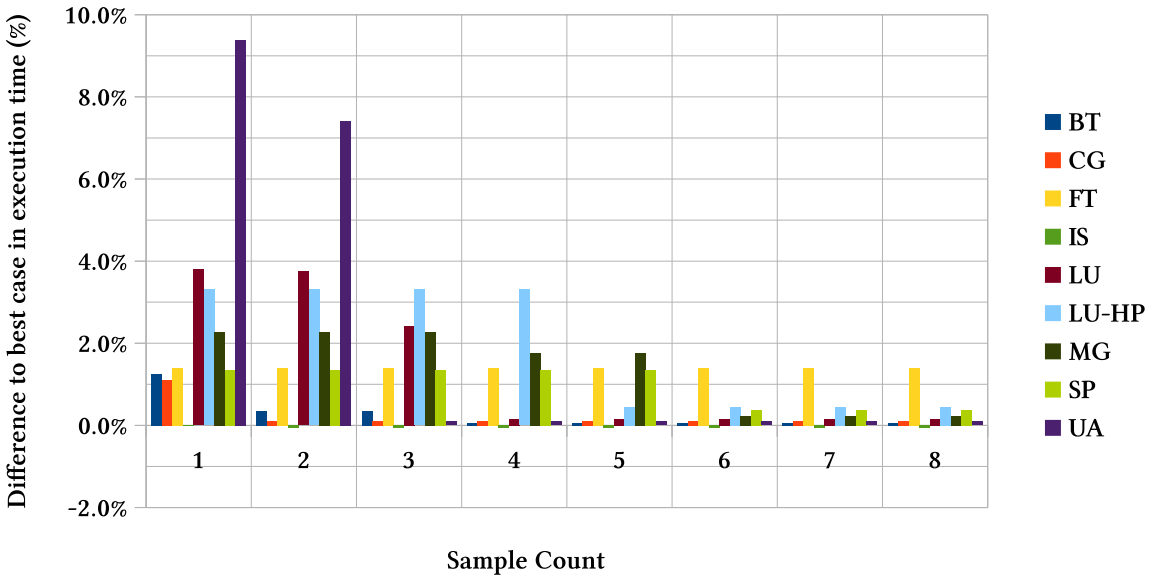


Figure 8.3: Execution time of *1-nearest neighbor* relative to the minimum execution time found by exhaustive search. *Test set* is the same as *training set* for this figure.

once selected to form the test set. This way we always test our learning algorithm on unseen benchmark programs.

The results are pretty good. For most benchmark programs, the first guess is already within 5% of the best case. The execution time for `UA` is not so good initially, but then it eventually finds the optimal solution in 7 samples. Out of the 9 benchmarks, it is only the `IS` benchmark which this approach apparently cannot solve satisfactorily in 8 samples.

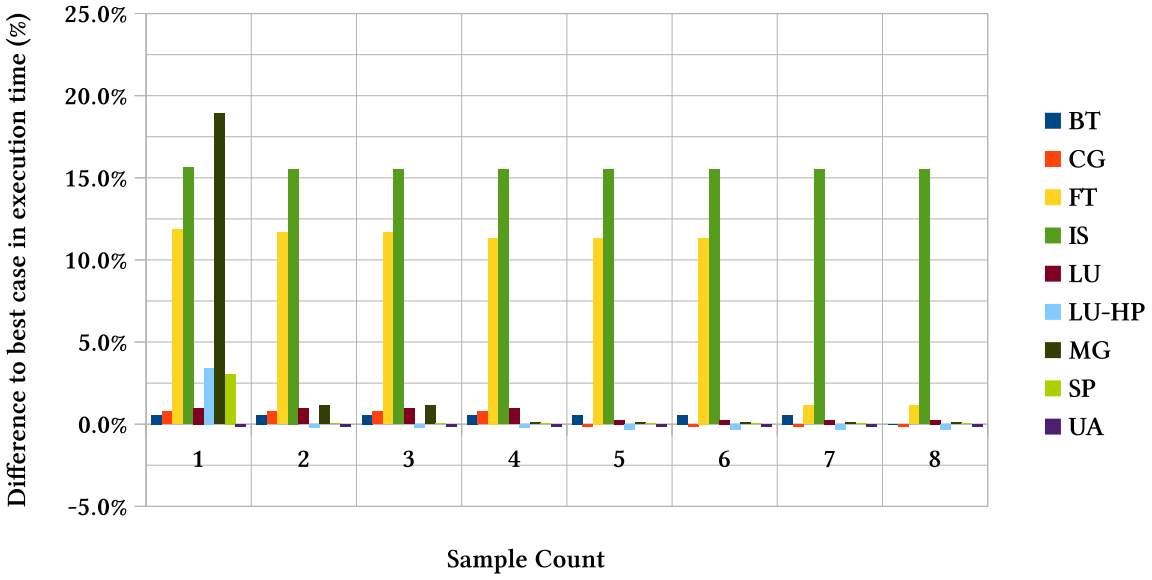


Figure 8.4: Execution time of *linear SVM* relative to the minimum execution time found by exhaustive search. *Test set* is the same as *training set* for this figure.

Figure 8.5: Execution time of *linear SVM* relative to the minimum execution time found by exhaustive search. Here the *training set* and the *test set* are in a *leave-one-out* manner, that is to *test* tuning `BT`, we use all benchmarks but `BT` as training data.

Chapter 9

Conclusion and Further Work

Optimization and tuning can give a competitive advantage in many areas which demand high-performance computations. We have discussed the challenges of tuning and optimization, reviewed approaches and ideas in the scientific literature, and described how Periscope tries to tackle these problems.

Our goal was to improve Periscope, to make tuning with it faster and better. We found that machine learning could speed up tuning, because it can reuse previous tuning experience instead of starting the tuning of every application with no prior knowledge.

Periscope had no machine learning before the this work. This thesis contributes the *design*, *implementation* and *testing* of machine learning support in Periscope, based on *state-of-the-art* techniques. We also provided a discussion of the design choices, alternative ideas, and many details about the implementation, so that it can serve as a documentation for further work and improvements.

We evaluated this contribution experimentally on a benchmark suite. Our results show that in most cases we are able to find a configuration within a few percent of the best configuration in one single guess, on previously unseen examples.

9.1 Further Work

In this section we share a few notes, ideas and suggestions about potential further work in the future:

- `HAVE_SQLITE3` macro is defined when SQLite3 is *found*, not when it is enabled. This means that the `--enable-sqlite3` configure option is practically ignored. What only matters is whether the SQLite3 library can be found.

- The choice of *tuning database implementation* (SQLite3 tuning database, when SQLite3 is available at build time, otherwise JSON dumper and loader) is hard-coded. It would be nice the make it configurable.

- Tuning database implementations use hard-coded *filenames*. (SQLite3 tuning database: `tuning.db`; JSON dumper and loader: `measurements.`*time*`.json` and `database.json`.) It would be nice to make these configurable.

- *Random sampling search* uses *linear SVM* prediction in a hard-coded manner, although *k-nearest neighbors* prediction is also implemented, and switching requires only a single line change in the source code. It would be nice to make it configurable without changing the source code.

- Machine learning techniques are *tightly coupled* to random sampling search – they are all in the same dynamically loaded library, for example. This is okay as long as random sampling search is the only probability model based search algorithm supporting machine learning. However, once probability model based machine learning is added to another search algorithm (e.g. the GDE3 genetic algorithm), it will be desirable to move the `ProbabilityModel` interface, its implementing classes `UniformDistributionModel` and `IndependentParametersModel`, and the prediction techniques (k-nearest neighbors and linear SVM) into a module separate from random sampling search.

- Related to the previous point, it is *unclear* which component should contact the *tuning database* directly: the *search algorithm* or the *machine learning technique*. The former approach is used with k-nearest neighbors, but the latter approach with linear SVM. Considering the separation suggested in the previous point, probably a unification with the latter approach would be the right thing to do.

- It would not be hard to use SVM with *radial basis functions* (RBF):

  - The parameter γ should be tuned simultaneously with the parameter C during cross-validation.
  - SVM with RBF is more likely to have *high variance*, but less likely to have *high bias*, i.e. it likely performs worse than linear SVM with few training programs, but better when there are plenty of training programs.

- Once there will be many training programs, several performance concerns might arise around training:

  - *Leave-one-out* cross-validation might need to be replaced with *k-fold* cross-validation, where k is a fixed integer.
  - *Dimensionality reduction* of input feature vectors with *Principal Component Analysis* (PCA) can lead to significant performance improvement when training SVMs.

- It might be interesting to try using *geometric mean* when averaging probabilities during cross-validation for SVM parameter tuning, as that could help us predict the expected number of samples required to find the best configuration.

  Using geometric mean could give a balanced performance on all programs, although that could also mean worse performance than arithmetic mean in the majority of cases. An interesting example is `-xHost`, which is part of the best tuning configuration for all but one benchmark program we tested. During the cross-validation, at one point we select the program missing the `-xHost`, and try to predict that from the other programs. But all other programs have `-xHost` in their best tuning configuration, so there is no way to predict the lack of `-xHost`. Therefore we have correct prediction with 0 probability in that case. That would mean 0 geometric mean, which implies fall back on uniform distribution. Arithmetic mean, however, would be willing to predict `-xHost` with high probability – a good idea for most programs, and a very bad idea for that single program which does not need it.

Acknowledgements

I would like to offer my special thanks to Prof. Michael Gerndt for his supervision and guidance throughout this thesis work.

I also greatly appreciate the assistance given by Isaías Alberto Comprés Ureña and Robert Mijakov, for their help regarding installation, build system and also other issues.

Bibliography

[1] D. H. Bailey, *The NAS Parallel Benchmarks*. Springer, 2011.

[2] G. Fursin, "Collective Mind: cleaning up the research and experimentation mess in computer engineering using crowdsourcing, big data and machine learning," *CoRR*, vol. abs/1308.2410, 2013.

[3] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3, pp. 1381–1384 vol.3, May 1998.

[4] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-–2, pp. 3–35, 2001. New Trends in High Performance Computing.

[5] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, R. W. Johnson, M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms," *Journal of High Performance Computing and Applications*, vol. 18, pp. 21–45, 2004.

[6] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Institute of Physics Publishing*, 2005.

[7] S. Triantafyllis, M. Vachharajani, and D. I. August, "Compiler Optimization-Space Exploration," in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pp. 204–215, IEEE, 2003.

[8] M. Haneda, P. M. Knijnenburg, and H. A. Wijshoff, "Automatic Selection of Compiler Options Using Non-parametric Inferential Statistics," in *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 123–132, IEEE, 2005.

[9] Z. Pan and R. Eigenmann, "Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning," in *Code Generation and Optimization, 2006. CGO 2006. International Symposium on*, pp. 12–, IEEE, 2006.

[10] Z. Pan and R. Eigenmann, "PEAK–a fast and effective performance tuning system via compiler optimization orchestration," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 3, p. 17, 2008.

[11] R. P. Pinkers, P. M. Knijnenburg, M. Haneda, and H. A. Wijshoff, "Statistical selection of compiler options," in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pp. 494–501, IEEE, 2004.

[12] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using Machine Learning to Focus Iterative Optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, (Washington, DC, USA), pp. 295–305, IEEE Computer Society, 2006.

[13] C. Lee and M. Stoodley, "UTDSP benchmark suite," 1998.

[14] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pp. 185–197, IEEE, 2007.

[15] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *Proceedings of the 4th international conference on Computing frontiers*, pp. 131–142, ACM, 2007.

[16] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *Computer*, vol. 29, pp. 84–89, Dec. 1996.

[17] H. Leather, E. Bonilla, and M. O'Boyle, "Automatic Feature Generation for Machine Learning Based Optimizing Compilation," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, (Washington, DC, USA), pp. 81–91, IEEE Computer Society, 2009.

[18] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thom-

son, C. Williams, and M. O'Boyle, "Milepost GCC: Machine Learning Enabled Self-tuning Compiler," *International Journal of Parallel Programming*, vol. 39, no. 3, pp. 296–327, 2011.

[19] S. Kukkonen and J. Lampinen, "GDE3: The third evolution step of generalized differential evolution," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 1, pp. 443–450, IEEE, 2005.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[21] "SQLite website." `http://www.sqlite.org/`. Accessed: 2014-09-04.