KernTune: Self-tuning Linux Kernel Performance Using Support Vector Machines

Long Yi
Department of Computer Science, University of the Western Cape
Bellville, South Africa
2475600@uwc.ac.za

James Connan
Department of Computer Science, University of the Western Cape
Bellville, South Africa
jconnan@uwc.ac.za

ABSTRACT

Self-tuning has been an elusive goal for operating systems and is becoming a pressing issue for modern operating systems. Well-trained system administrators are able to tune an operating system to achieve better system performance for a specific system class. Unfortunately, the system class can change when the running applications change. Our model for self-tuning operating system is based on a monitor-classify-adjust loop. The idea of this loop is to continuously monitor certain performance metrics, and whenever these change, the system determines the new system class and dynamically adjusts tuning parameters for this new class. This paper describes KernTune, a prototype tool that identifies the system class and improves system performance automatically. A key aspect of KernTune is the notion of Artificial Intelligence (AI) oriented performance tuning. It uses a support vector machine (SVM) to identify the system class, and tunes the operating system for that specific system class. This paper presents design and implementation details for KernTune. It shows how KernTune identifies a system class and tunes the operating system for improved performance.

Keywords

support vector machine, Linux kernel, operating system, optimisation, performance, benchmark, machine learning, workload, open source, system profiler

1. INTRODUCTION

KernTune is a combination of performance tuning and SVM technology that provides a practical tool for operating system optimisation needed to support continued improvements in system performance. A major obstacle to operating system optimisation is obtaining high-quality system information that represents the current system usage. Ideally, such information should be representative of how an operating system is used on a machine. Given the current operating system architecture, it is not practical for end-users to obtain and use such information, as typical users do not know how to generate the information and would not be able to optimise their systems if they did. The KernTune tool addresses this problem, using the *monitor-classify-adjust* cycle to collect, process, and apply information to optimise a system automatically.

This paper describes our implementation of KernTune for GNU/Linux systems. Our implementation is composed of three components: the *Monitor*, the *Classifier* and the *Adjustor*. *The monitor* is a monitoring component that implements continuous, low-overhead collection and system information monitoring. *The classifier* is a classifying component that translates system information and classifies the system class, including computing the best possible system class from the classification results. *The adjustor* is an adjusting component that implements optimisation, choosing kernel parameters and adjusting system settings to the specific system class. In this paper we focus on the first two components, *the monitor* and *the classifier*. Although some discussion of *the adjustor* is needed to provide context for the rest of the work, a detailed description of specific system optimisations is outside the scope of this paper.

2. RELATED WORK

There are a number of tools for automatic Linux kernel optimisation. In this section we review some relevant tools and explain how they differ from KernTune.

KernTune enables systems to evolve with changes in usage. This functionality is currently not provided by any other tool. KernTune builds on technology originally developed for classification. Several existing tools adjust kernel parameters for the purpose of improving performance. These tools differ from KernTune in two important ways. First, though these tools are based on the designer's knowledge and experience of adjusting the kernel parameters for improving performance, the users of the tools need to know the purpose of the system (the system class). In KernTune, we employ SVM technology to classify the system class. A further difference between KernTune and earlier tuning tools is that KernTune is designed specifically for automatic tuning and optimisation. It runs as a daemon in the background and keeps track of usage changes.

Powertweak-Linux [9] is a kernel tuning tool that improves Linux system performance. It uses the `/proc` file-system, and `hdparm` tool to tweak systems. Like KernTune, this tool also performs optimisations on systems by known tun-

ing rules. Powertweak-Linux was designed for Linux systems and can either apply manual optimisation or tuning rules. KernTune uses machine learning, parameter-driven and system-class-specific optimisations to improve system performance.

The most recent tool for tuning the Linux kernel is the kernel patches developed by Jake Moilanen [5]. He implements an automatic optimisation module in the Linux kernel. It supports continuous tuning of the kernel by applying a Genetic Algorithm. There are substantial similarities between Moilanen's patches and KernTune. Both Moilanen's patches and KernTune use a Machine Learning method to tune systems while causing very little overhead. The primary difference between the two tools is how they are applied. In KernTune we have focused on classification, whereas the emphasis of Moilanen's patch has been focused on chromosomes. Another difference is that Moilanen's patch works in the kernel as a module and KernTune runs out of the kernel as a daemon.

SarCheck [2] is another performance analysis and tuning tool for Linux systems. It produces recommendations and explanations for tuning the kernel with supporting graphs and tables. The KernTune does not produce recommendations and Its goal is to tune real systems. Though the goal of SarCheck is different from KernTune, the two tools share many similarities: both tools continuously collect system information and use the information to guide further operations. The main differences between the two tools are that SarCheck produces recommendations and suggestions for tuning the kernel, it collects system information, and produces the optimisation recommendations, rather than applying the optimisation as in KernTune. KernTune also supports automatic optimisation.

3. SVM CLASSIFICATION

In the last few years, there has been a surge of interest in Support Vector Machines (SVMs), a new generation learning system based on recent advances in statistical learning theory. SVMs have empirically been shown to deliver good generalisation performance in real-world applications such as text categorisation [4], hand-written character recognition [8], image classification [6], object detection [7], speaker identification [12], etc. SVMs show a competitive performance on problems where data is sparse (few data) and noisy (many features). SVMs were first introduced in the early 1990s by Vapnik [11] as a binary classification tool and are rapidly growing in popularity due to many attractive features, and promising empirical performance.

We first briefly introduce basic ideas behind SVM. Then we discuss SVM classification. Next, we present LibSVM, a popular open source tool for SVM classification and regression.

3.1 Review of SVMs

Support Vector Machines (SVMs) are a set of related supervised learning methods used for classification and regression. Their common factor is the use of a technique known as the "kernel trick" to apply linear classification techniques to non-linear classification problems.

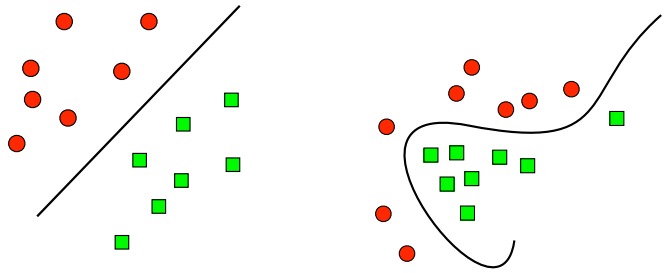

Figure 1: (a) A linear classification (b) A non-linear classification

Suppose we want to classify some data points into two classes. We are interested in whether we can separate them by a *hyperplane*. A hyperplane is a geometrical concept. It is a generalisation of the concept of a plane. We also want to choose a hyperplane that separates the data points clearly, with maximum distance to the closest data point from both classes, this distance is called the *margin*. We desire this margin as large as possible since we can more accurately classify a new point when the separation between the two classes is greater. If such a hyperplane exists, the hyperplane is clearly of interest and is known as the *maximum-margin hyperplane* or the *optimal hyperplane*. The vectors that are closest to this hyperplane are called the *support vectors*. The basic idea behind SVMs is to learn a decision hyperplane that separates the data points with maximum margin. In linear classification cases, the algorithm aims to find a linear decision hyperplane that can separate the data points with maximum margin. In non-linear cases, the algorithm maps the data points into a higher dimensional space and thus finds a decision hyperplane that can separate the data points linearly. Figure 1 (a) shows a classic example for linear classification.

In this example, the objects: circles and squares, belong either to class A—circles or B—squares. The separating line defines a boundary between class A and B. Any new object added to this example would be classified as class A or class B. The above is the simplest example. Unfortunately, most real-world problems are not that simple. Most problems involve non-linear separable data for which there does not exist a hyperplane that can successfully separate one class from another. More complex structures are needed to make an optimal hyperplane. This situation is illustrated by Figure 1 (b).

Compared to Figure 1 (a), it is clear that a curve instead of a line, forms the separation between circles and squares. The curve is more complex than the straight line. SVM is particularly suitable to solve such problems. Rather than drawing a curve between the two classes, SVM maps the data into a higher dimensional space by using a kernel function and then draws a separating hyperplane there. Figure 2 shows the idea behind SVM non-linear classification.

This higher dimensional space is called the feature space and the original training set is called the input space. With an appropriately chosen feature space, any input space can be separated linearly. In Figure 2, the original objects in the input space have been mapped into the feature space by

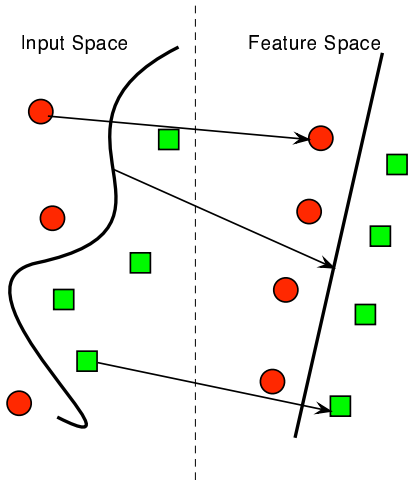

Figure 2: Mapping from input space to feature space

using a set of known functions as the *kernel* which maps the input data into a different space—the feature space—where a hyperplane can be used to do the separation. Note that in the feature space, the mapped objects are linearly separable. Thus, instead of drawing a complex curve in the *input space* to separate the circles and squares, we find an optimal line in the *feature space* that separates the two classes linearly. A more detailed introduction on SVM can be found at [11].

3.2 SVM Classification

The SVM is a supervised classification technique for creating a *classifier* from training data. The training data consist of pairs of input objects typically *vectors*, and *desired outputs*. The output can be a continuous value—called *regression*, or can be class labels of the input object—called *classification*. Since the primary interest of our research is SVM classification, we will ignore SVM regression and focus on SVM classification.

SVM classification usually involves two kinds of data, training data and testing data. The training data includes some training examples. Each of those examples contains one *label* and several *attributes*. The testing data includes testing examples, which each of them containing several *attributes*. Using the training set, the SVM classifier would distinguish between the members and non-members of a given class. Having learned the features of the class, the SVM can predict new objects as members or as non-members of the class. The goal of SVM classification is to predict the *label* of testing instances which are only given by *attributes* in the testing set by learning from the training instances in the training set. To achieve this, the SVM learns a separating hyperplane from the training data by using the kernel function to map the data point into a higher dimensional space.

3.3 LibSVM

LibSVM (Library for Support Vector Machines) [1] is a simple and easy-to-use open source implementation for SVM. It was developed by Chang and Li and contains C-support vector classification (C-SVC) [3], v-support vector classification (v-SVC) [8], e-support vector regression (e-SVR) [8], and v-support vector regression (v-SVR) [10]. It supports multi-class classification, weighted SVM for unbalanced data, cross-validation and automatic model selection. It has interfaces for Python, R, Splus, MATLAB, Perl, Ruby, and LabVIEW. Users can easily link it with their own programs. It includes four kernels: Linear, Polynomial, Radial Basis Function (RBF), and Sigmoid. The goal of LibSVM is to help users to easily use SVM as a tool. We use the C-support classification and RBF kernel in our KernTune.

4. DESIGN AND IMPLEMENTATION

We designed our KernTune tool to meet a number of requirements that we believe are necessary for mainstream systems:

- Optimisation should happen on the machine where the application or service is running. For complex systems, there can be substantial variation in how the operating system is used for different requirements. Given these variations between system classes, automatic optimisation will be most effective if it can process information specific to the system and the machine.
- Optimisation should not happen only once. In a real computing environment, system usage changes according to various requirements. The application or service running in the morning might not be used by end-users in the evening. The end-users could require another service instead of the morning one. Re-optimisation for this requirement is reasonable, but it is not practical for end-users to keep optimising a dynamically changing system. To enable optimisation continuously, we assume that the system information represents the class of system and base all optimisations on this system class. In practice, a user-level daemon is usually used for this.
- Optimisation must be transparent to the user. We assume that the people who use computers do not have much background in the field of computers or system optimisation. Optimisation should be transparent in that the user does not need to understand or participate in the optimisation process. Transparency also implies that the optimisations must achieve consistent performance benefits with no negative performance impact.

The KernTune tool is designed to address all these requirements. It automatically collects and analyses system information and uses that to optimise the kernel. Our prototype implementation of KernTune supports automatic optimisation for Intel x86-based computers running the Linux kernel. We built several custom system tools to collect performance data, test results and made small modifications to LibSVM to support automatic collection and classification. The main component of KernTune is based on the Library for Support Vector Machines (LibSVM) [1]. We have extended this library to support constant system-specific optimisation.

4.1 Overview of KernTune

Our initial design targets a single server environment. KernTune provides a *monitor-classify-adjust* cycle control for SVM-based and system-specific optimisation. An important

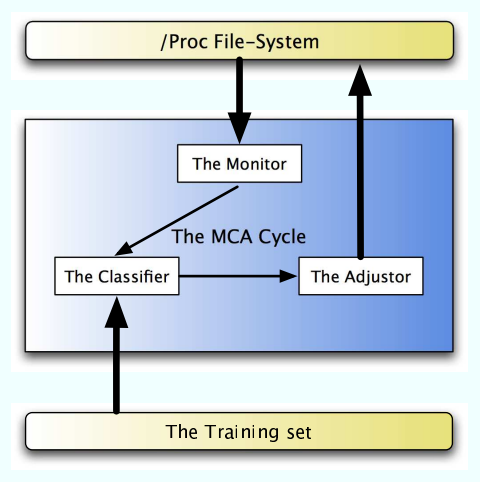

Figure 3: Overview of KernTune

feature of KernTune is that optimisation occurs on the system where the applications are running. The optimisation occurs only when the applications or services have already changed.

The major components of KernTune and their relationships are represented schematically in Figure 3. Optimisations in KernTune are applied without the use of Linux kernel source code. To achieve this goal we explore the `/proc` file-system that presents the state of a running GNU/Linux system. Prior work on Linux kernel tuning was implemented by Jake Moilane [5]. His goals were to modify the kernel source and applying a Genetic Algorithm (GA) to kernel optimisation. An important aspect of our design is that the *monitor-classify-adjust* cycle is machine and operating system independent. Though we are aware of prior work on automatic kernel optimisation, we have chosen not to modify the Linux kernel, focusing instead on the problem of using SVM-based optimisations to improve system performance. The traditional method for providing a viable tool for self-optimisation is to define a series of optimisation rules that build a knowledge-based database to support various system classes. Our method is to apply the SVM technology to build a system tool (KernTune) that permits automatic adjustment of system settings.

KernTune has three major software components: *monitor*, *classifier* and *adjustor*. *The monitor* collects system information for use by *the classifier*. *The monitor* collects system information with very low overhead, leaving subsequent processing to *the classifier*. *The classifier* is a user-level daemon that provides system settings management as well as information analysis to classify system classes and decide when to invoke *the adjustor*. *The adjustor* implements the optimisations provided by KernTune. The current version of our Adjustor supports three different system-class optimisations: web server class, ftp server class and database server class.

4.2 Data Collection: *The monitor*

All the optimisations require system-specific information, including the active application information and performance counters. *The monitor* automatically collects this information. The data collection is implemented in *the monitor*, the output of *the monitor* is the input of *the classifier*. One key goal of *the monitor* is low system overhead. Our monitor achieves this goal by using a subset of the performance counters and the active application information, rather than collecting all performance and application data. A straightforward approach for system data collection is real-time collection. Although this approach has the advantage that it is well understood, the system overhead incurred during *the monitor* execution would be substantial. If too little data is collected, it would be difficult to ensure that the resulting data was representative of the system class. By collecting performance counters and the active application information on the system continuously, we assure that the data accurately reflects how the system is used.

After collecting the system data from the `/proc` file-system, we combine the individual performance counters from each performance object to generate a single basic output file. *The monitor* sums the counters from each performance object for each kernel module to create an aggregate file. This file includes the active application name, its open port number, and other system performance counters such as processor time, memory usage, packets sent/sec, etc. (See Section 4.3). The selection of performance counters is a general problem for performance tools and is not unique to our methodology. In conventional performance tools, complete system information is collected. It is a large data collection and results in system overhead. By contrast KernTune only collects a very small amount of important performance counters and the active application information. We believe that our method is appropriate, even though complete data collection can give better classification results.

The overhead of our current monitor satisfies our main subjective performance goal: it is unnoticeable. Measurements of our current monitor implementation show the overhead of system profiling is typically below 1% of CPU usage. Given this level of performance, we have chosen to defer further work on reducing monitor overhead and focus on the other functionality required for automatic optimisation.

4.3 Data Processing: *The classifier*

The classifier is implemented as a user-level daemon that periodically reads and processes the raw data samples generated by *the monitor*. Whereas *the monitor* is responsible for data collection, *the classifier* performs data translation and processing. The goals of this processing are twofold: to transform the raw data samples into a compact form that can be used directly by LibSVM and to decide when to invoke an optimisation. In our work to date, we have primarily focused on transforming raw data for use by LibSVM.

The classifier translates *the monitor* output file into the LibSVM file format and identifies the system class in the translated file by applying LibSVM. In the conversion from raw data to the LibSVM file format, each sample has to be translated into numeric values that the LibSVM library can recognise. The translated data provides the information needed by LibSVM to identify the system class. LibSVM maps the translated data to an intermediate form that the LibSVM engine can process.

From the optimisation point of view, operating systems are logically composed of multiple modules, typically with CPU, memory, file-system, disk I/O and networking. The number of performance counters used results in increasing the classification accuracy. We use the term sample set to refer to the output from *the monitor*. When processing the sample set, *the classifier* invokes the `svm_predict` interface of LibSVM and outputs a result file containing the class of system being recognised. A sample set may not include enough information to perform an accurate classification. This makes it desirable for *the classifier* to process multiple sample sets from *the monitor* and combine the results into a single, more accurate, result for *the adjustor*. *The classifier* sums all the results and concludes the best possible system class.

4.4 System Tuning: *The adjustor*

The adjustor applies optimisations using the system class from *the classifier*. For this paper, *the adjustor* implemented three different system-class optimisations, namely the Web Server, Ftp Server and Database Server classes. The input to *the adjustor* is the system class which has already been identified by *the classifier*. Using a set of suggested kernel values, *the adjustor* performed optimisations through the `sysctl` system call.

When *the classifier* gives us a clear recommendation on which parameter should be adjusted and by how much, we merely have to change the parameter. However, this simple adjustment may cause technical problems: dynamically changing parameters could cause system instability while the system is performing a high workload. To address this problem, an adaptable mechanism is necessary for the high-workload condition. We provide an extra workload check before adjusting the parameters, to make sure the adjustment is under a low-workload. The optimisations do not have to include all tuning parameters since the parameters are used only to direct optimisation toward the most important parts of the system. Our KernTune Adjustor is currently implemented as a composition of tuning rules which optimise the most important kernel components for the system class. Considering system overhead and complexity, we have chosen to adjust the most important parameters that impact on the performance rather than adjusting every parameter in the kernel.

Since these optimisations make some system components run faster at the expense of others, a potential pitfall of the optimisation is that the system might perform worse when running more interactive applications on the system. This problem could be avoided by introducing more system characteristics which represent more system classes into the training set, for example, adding the port numbers of applications or services to the sample set. This scheme is used by *the monitor* and *the classifier*.

5. EXPERIMENTAL RESULTS

This section describes our experiments to evaluate automatic SVM-based optimisation with KernTune. After describing the experimental system and workloads, we present two sets of results. One set of experiments documents the correctness of classification in KernTune. The second set of experiments quantifies the effectiveness of optimisation in the KernTune Adjustor. We report the results, both in terms of correctness of classification and effectiveness of optimisation. Our experimental results show that the correctness of classification using SVM is as high as 90% and the optimisations applied by KernTune achieve substantial performance improvements for our test workloads. The main criteria upon which KernTune should be evaluated is its correctness for SVM classification. This section also discusses issues relating to making KernTune more practical.

5.1 Experimental Details

Our KernTune prototype tool used the Linux kernel 2.4.29. We ran our experiments on two Intel x86-based PCs. One machine represented a server from one of our chosen system classes. The other machine was a workload generator to simulate a real computing environment. The two systems both include a 512 KB second-level cache and 256 MB of main memory and they have exactly the same hardware configuration. The target experimental system is based on Gentoo Linux version 2004.2. The workload-generator system is based on SUSE Linux 10.0. We used three workloads: a web server workload, an ftp server workload and a database workload for the three system classes in this study.

Table 1 gives a description of each workload generator. All experiments for this paper were run in single-system-class mode. Two factors limited our choice of workloads. First, the system workload must be easily simulated by existing tools. As a research performance tool, KernTune does not support a full set of system classes and workloads. Second, a benchmark tool for the workload must be readily available. As a result we were unable to simulate workloads for many popular system classes. We also excluded similar training samples within a class, as they will not affect the SVM processing in *the classifier* significantly.

5.2 Classification Results

Table 2 shows our classification experimental results for the three system-class workloads. It gives the number of testing attempts and the system classes identified by *the classifier*. The accuracy is the ratio of correct classification against the total number of test attempts. There are two accuracy results in the table. One is calculated from a training set including only 40 samples. The other is calculated from a bigger training set with 400 samples. Our comparative experiment results demonstrate that increasing effective training samples improves the classification results of the SVM significantly. Classification using SVM technology achieves a high accuracy, depending on system class. However, more training samples will need more CPU time to analyse and calculate, this would lead to a high overhead of the system. Our experience with SVM suggests that an effective and small training set is needed to obtain better accuracy of classification. To determine the trade-off between the appropriate number of training samples and system overhead is important. Our experiments show that using up to 2000 samples causes tolerable overhead. Table 3 lists the overhead in the 400-sample case. Figure 4 plots the accuracy of classification for each class as the number of samples are varied. Figure 5 plots the overhead as the number of samples are varied. We developed `perfmon` for monitoring system performance and it is also a part of *the monitor*.

5.3 Performance Results

Table 1: Workload Simulation

System Class	Workload Generator
web server	`httperf` runs multiple http fetches in parallel, to simulate a web server workload. It can also be considered a web server benchmark tool. Example: `./httperf --wsess=1000,50,0 --rate=200 --server=xxx.xxx.xxx.xxx --uri=/index.html`
ftp server	`dkftpbench` is an ftp benchmark tool. It can be used as a ftp server workload generator. Example: `./dkftpbench -n1 -hxxx.xxx.xxx.xxx -t30 -v -uftp -pftp -fbigfile`
database server	The `mysql test suit` is a database benchmark suite for mysql. It includes test_insert, test_select, test_connect and test_create tools. They can be found in the mysql package. They can also be used as sql simulation generators to simulate database workloads. Example: `./test_insert;./test_select`

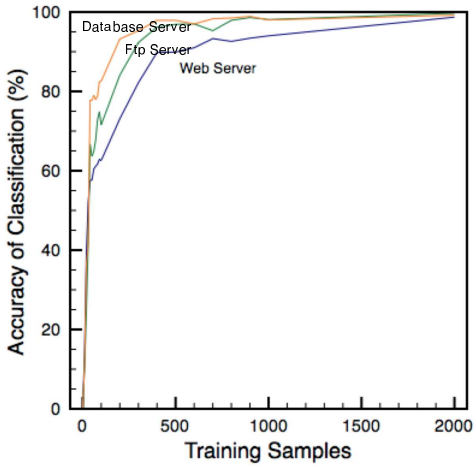

Figure 4: Accuracy of Classification

Table 2: Classification Results

System Class	Test Tool	Attempts	40	400
Web Server	httperf	1000	57.8%	89.8%
Ftp Server	dkftpbench	1000	66.7%	96.2%
DB Server	test_insert	1000	77.8%	97.9%

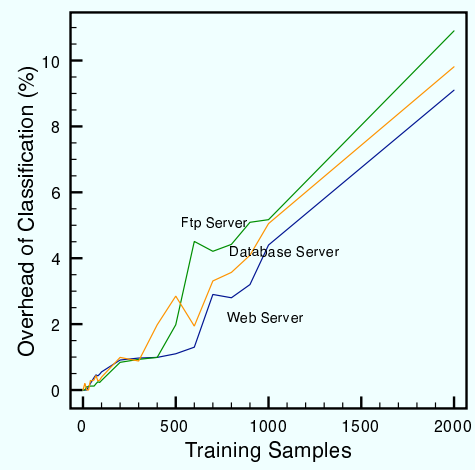

Figure 5: Overhead of Classification

Table 3: Overhead of KernTune

System	KernTune Off	KernTune On	Overhead
Web Server	0.00% CPU	0.99% CPU	0.99%
Ftp Server	1.98% CPU	2.97% CPU	0.99%
DB Server	85.15% CPU	87.13% CPU	1.98%

Table 4: Performance Results

System Class	Test Tool	Improve
Web server	time, httperf	6.16%
Ftp server	time, dkftpbench	2.19%
Database server	time, test-insert	8.04%

Table 5: Web Server Results

Tool	KernTune Off	KernTune On	Diff
`httperf`	50.956(s)	47.283(s)	7.2%
`httperf`	50.993(s)	47.043(s)	7.7%
`httperf`	50.453(s)	47.843(s)	5.2%
`httperf`	50.879(s)	47.409(s)	6.0%
`httperf`	51.334(s)	48.943(s)	4.7%
`httperf`	50.339(s)	47.463(s)	5.7%
`httperf`	50.421(s)	47.579(s)	5.6%
`httperf`	50.842(s)	47.085(s)	7.4%
`httperf`	50.563(s)	47.467(s)	6.1%
`httperf`	50.675(s)	47.636(s)	6.0%

KernTune optimised the system, and then we benchmarked the optimised system. Table 4 shows optimisation results of the three system classes. Each optimisation improvement given in Table 4 are computed from the average of ten tests in Tables 5, 6 and 7. The `time` program is used to compute the CPU time. The results in "KernTune Off" and "KernTune On" columns are the time of CPU costs, smaller is better. Table 4 shows that in each case the optimised system performs better that the original system.

6. CONCLUSIONS

Our study has demonstrated that automatic optimisation based on SVM technology is both efficient and effective. In our experience of developing KernTune, we found some issues that need to be resolved to make KernTune more practical. Our current KernTune uses a training set to train the SVM classifier. Finding and testing a good training set with minimal samples for all different system classes is very difficult, especially in a complex networking environment. The samples must be able to represent the different system usage concisely to reduce the time and overhead of SVM processing. As more system classes evolve in the future, this would lead to even more samples. It is not possible to collect those training samples manually. A tool should be developed to collect the samples automatically. Our current KernTune uses the SVM method for classifying system classes. We achieved high classification accuracy with a small training set. We did not optimise the SVM classification procedure and the training set in this study. We believe that providing optimisation for the SVM process with carefully tested training samples can achieve more accurate classification. Optimising SVM also includes choosing better feature vectors for training sets, scaling training data, choosing a better kernel function, etc. As more system classes should be considered in the real computing world, this would bring a bigger training set and more complex training samples to the SVM processing. The continuous collection of system information and SVM processing can lead to overhead issues with KernTune. This issue must be fully resolved if the overhead approaches the performance improvement, since KernTune is a tuning tool to make a system run faster, and not to slow down the system. KernTune must achieve a balance between good classification accuracy and the load caused by the SVM processing. The KernTune tool has a major practical drawback, however, as it requires new training samples when applying it to a new system class. Another possibility is to develop a tool to generate samples for new classes.

Table 6: Ftp Server Results

Tool	KernTune Off	KernTune On	Diff
`dkftpbench`	62.070(s)	60.093(s)	3.2%
`dkftpbench`	61.278(s)	60.061(s)	2.0%
`dkftpbench`	61.630(s)	60.092(s)	2.5%
`dkftpbench`	60.929(s)	60.034(s)	1.5%
`dkftpbench`	61.422(s)	60.053(s)	2.2%
`dkftpbench`	60.708(s)	60.016(s)	1.1%
`dkftpbench`	60.776(s)	60.011(s)	1.3%
`dkftpbench`	61.653(s)	60.082(s)	2.5%
`dkftpbench`	61.756(s)	60.025(s)	2.8%
`dkftpbench`	61.755(s)	60.021(s)	2.8%

Table 7: Database Server Results

Tool	KernTune Off	KernTune On	Diff
`test-insert`	36.263(s)	33.656(s)	7.2%
`test-insert`	36.871(s)	33.491(s)	9.2%
`test-insert`	36.123(s)	33.131(s)	8.3%
`test-insert`	36.443(s)	33.154(s)	9.0%
`test-insert`	36.754(s)	33.212(s)	9.6%
`test-insert`	36.278(s)	33.512(s)	7.6%
`test-insert`	36.218(s)	33.625(s)	7.2%
`test-insert`	36.389(s)	33.643(s)	7.5%
`test-insert`	36.261(s)	33.712(s)	7.0%
`test-insert`	36.411(s)	33.589(s)	7.8%

This paper describes a tool for continuous low-overhead monitoring, classifying and adjusting to meet the requirements of automatic optimisation. We achieve these low overheads through SVM-based statistical sampling and by deferring SVM processing to available idle CPU cycles. Our results show that continuous optimisation can be achieved with very low overhead and that the resulting optimisations are effective. As modern operating systems become increasingly complex, the importance of these automatic optimisations will increase. The research combines aspects of both operating systems and machine learning. Our KernTune demonstrates how a practical tool for automatic optimisation can be implemented for GNU/Linux systems. It brings a new application to the SVM area and a new approach to oper-

ating system automatic optimisation.

There are three areas of future work we plan to pursue, SVM classification, system optimisation and benchmark skills. We plan to: 1. Tune the SVM classification and make the SVM processing faster; 2. Scale and tune training sets to train SVM; 3. Introduce more system classes to KernTune and generate data for the classes; 4. Optimise the SVM library—LIBSVM—built into KernTune; 5. Discover and incorporate more system optimisation techniques into KernTune; 6. Improve benchmark skills by introducing industry-standard test suits; 7. Add the Linux kernel-2.6.x support to KernTune and bring KernTune to other operating systems; 8. Optimise KernTune itself and reduce its overhead to the system.

7. REFERENCES

[1] C. Chang and C. Lin. LibSVM: a library for support vector machines. 2001. Available at `http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.ps.gz`.

[2] A. Corporation. *SarCheck*, 1996. Available at `http://www.sarcheck.com`.

[3] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995. Available at `http://citeseer.ist.psu.edu/cortes95supportvector.html`.

[4] T. Joachims. Transductive inference for text classification using support vector machines. In I. Bratko and S. Dzeroski, editors, *Proceedings of ICML-99, 16th International Conference on Machine Learning*, pages 200–209, Bled, SL, 1999. Morgan Kaufmann Publishers, San Francisco, US. Available at `http://citeseer.ist.psu.edu/joachims99transductive.html`.

[5] J. Moilanen. Linux: Tuning the kernel with a genetic algorithm. Jan 2005. Available at `http://kerneltrap.org/node/4493`.

[6] E. Osuna, R. Freund, and F. Girosi. Training support vector machines:an application to face detection. *Computer Vision and Pattern Recognition*, pages 130–136, 1997. Available at `http://citeseer.ist.psu.edu/osuna97training.html`.

[7] M. Pontil and A. Verri. Support vector machines for 3d object recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(6):637–646, 1998. Available at `http://citeseer.ist.psu.edu/pontil98support.html`.

[8] B. S.-K. C. F. P. T.Poggio and V.Vapnik. Comparing support vector machines with gaussian kernels to radial basis function classifiers. *IEEE Trans*, 45:2758–2765, Nov 1997.

[9] A. van de Ven. *Powertweak Linux*, 2003. Available at `http://powertweak .sourceforge.net`.

[10] V. N. Vapnik. *Statistical Learning Theory*. Adaptive and Learning Systems for Sigal Processing. Wiley and Sons, Sep 1998.

[11] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, Nov 1999.

[12] V. Wan and W. Campbell. Support vector machines for speaker verification and identification. pages 775–784, 2000. Available at `http://citeseer.ist.psu.edu/ wan00support.html`.