AUTO-TUNING MULTI-TIERED APPLICATIONS FOR PERFORMANCE

Vimuth Fernando

158017T

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Thesis/Dissertation submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science and Engineering

Department of Computer Science & Engineering

University of Moratuwa

Sri Lanka

July 2016

DECLARATION

I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

The above candidate has carried out research for the Masters thesis/Dissertation under my supervision.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Signature of the Supervisor: Date:

ACKNOWLEDGEMENTS

I am sincerely grateful for the advice and guidance of my supervisor Prof. Sanath Jayasena. Without his help and encouragement this project would not have been completed. I would like to thank him for taking time out of his busy schedule to be available anytime that was needed with help and advice.

I would also like to thank my progress review committee, Dr. Ajith Pasqual and Dr. Dilum Bandara. Their valuable insights and guidance helped me immensely. My thanks also goes to Prof. Saman Amarasinghe for his help and advice.

I would like to thank the entire staff of the Department of Computer Science and Engineering, Both academic and non-academic for all their help during the course of this work and for providing me with the resources necessary to conduct my research.

This work was partially funded by the LK Domain Registry through Prof. V.K. Samaranayake top-up grant.

Finally, I would like to express my gratitude to my family and all my friends for their support.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

ABSTRACT

Auto-Tuning Multi Tiered Applications for Performance

With the widespread use of cluster-based environments, getting the maximum possible performance from multi-tiered web applications becomes an important task. The large numbers of configurable parameters in such environments and applications, however, makes manual performance tuning (i.e., searching for and identifying key parameters that affect performance and optimal values for those parameters) extremely difficult, if not virtually impossible. The problem becomes further complicated because the key parameters and/or their optimal values will vary across environments, applications and workloads.

In this work, we explore the autotuning approach, which is generally used to automatically tune the performance of programs in traditional HPC settings, to tune multi-tiered web applications. Our approach is based on OpenTuner, a framework used to build auto-tuners to search through a configuration space for an optimal configuration. Even for this autotuning approach, the wide variations and the dynamic nature in the runtime environment, such as network congestion, variations in demand, possible node failures and changes in workloads pose a significant challenge. In this work, we explore offline and online tuning techniques to overcome the challenges of autotuning multi-tiered applications.

We present results of offline autotuning experiments that tuned benchmark applications for multiple performance goals. We show that 20% - 25% improvements in response time and throughput can be achieved through our offline autotuning approach. We present a way of reducing the tuning time by pruning the configuration space. We identify the parameters in web servers that contribute most to performance. We also show that different performance goals can lead to differences in configurations and discuss the shortcomings of offline autotuning methods. We also take a look at online tuning methods and show that online tuning of multi-tiered applications is feasible.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Keywords: Performance autotuning; Multi-tiered applications; Opentuner; Autotuning;

LIST OF FIGURES

Figure 1.1	A simple multi-tiered application	2
Figure 3.1	Major components of Opentuner	11
Figure 3.2	Offline autotuner	13
Figure 5.1	Percentage performance gains achieved from tuning	16
Figure 5.2	RUBiS benchmark tuning results	17
Figure 5.3	TPC-W benchmark tuning results	18
Figure 5.4	Tuning the RUBiS benchmark for response time.	19
Figure 5.5	Process of pruning the configuration space	21
Figure 5.6	Tuning the Benchmarks with differently sized configuration spaces	23
Figure 6.1	Tuning the RUBiS benchmark to maximize throughput with online autotuning	29
Figure 6.2	Online autotuner based on Sibling Revelry	31
Figure 6.3	Online tuning using the Sibling Revelry inspired method	32

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

LIST OF TABLES

Table 3.1 Parameters used in the tuning process 12

Table 5.1 Parameters that contribute most to the performance. Ranked according to their contribution 24

Table 5.2 Gains in response time for TPC-W benchmark from tuning individual tiers. 25

Table 6.1 Comparison of online tuning algorithms 30

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

LIST OF ABBREVIATIONS

API	Application Programming Interface
JVM	Java Virtual Machine
SLA	Service Level Agreement
WIRT	Web Interaction Response Time
WIPS	Web Interactions Per Second
QOS	Quality of Service

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

TABLE OF CONTENTS

Declaration of the Candidate & Supervisor i
Ackowledgement ii
Abstract iii
List of Figures iv
List of Tables v
List of Abbreviations vi
Table of Contents vii
1 Introduction 1
1.1 Problem 1
1.2 Proposed Solution 3
1.3 Contributions 3
1.4 Organization 3
2 Literature Survey 5
2.1 Auto-tuning 5
2.1.1 Offline auto-tuning 6
2.1.2 Online auto-tuning 7
2.1.3 Hybrid approaches 7
2.2 Performance tuning of multi-tiered applications 8
2.2.1 Model based approaches 8
2.2.2 Learning based approaches 9
3 Methodology 11
4 Experimental Setup 14
4.1 RUBiS Benchmark 14
4.2 TPCW Benchmark 14
4.3 Deployment Environment 15
5 Offline Tuning Results 16
5.1 RUBiS results 17
5.2 TPC-W results 18

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

5.3 Tuning time 19
5.4 Offline Tuning Discussion 23
6 Online Autotuning of Multi Tiered Applications 27
6.1 Simple online autotuning 28
6.2 Sibling Revelry based methodology 30
6.3 Discussion 32
7 Conclusions and Recommendations 34
References 35

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Chapter 1

INTRODUCTION

Web applications are a very important aspect of the Internet today. These applications are commonly used around the world for all kinds of purposes including business, security, education and communication. These applications rely on performing to their maximum ability to cater to the rapidly growing demand. A lot of work has been done in this are to ensure that these applications yield their best performance. Getting the maximum performance out of such applications also allow them to be deployed with fewer resources thereby lowering costs, and to provide a better user experience.

These are one of the many cases where multi tiered applications are used. We define multi-tiered applications as those whose functionality is divided across multiple physically separated groups of computers also called *tiers*. In addition to web services other applications also use this architecture to ensure good performance and reliability including high performance computing applications used by researchers around the globe.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 1.1 shows one such multi-tiered application using the common tier breakdown used by web application. In this case the functionality of the application is broken down into three *tiers*. One tier handle the presentation related functionality(Ex: generating web pages), one tier handles the business logic of the application, and the final tier handles the database related functionality.

Performance tuning programs that uses a multi-tiered architecture is a difficult task. Any method of generating an increase in performance of these application can have far reaching implications.

1.1 Problem

Each tier in a multi-tiered application consists of a set of computers executing some program. These programs run in environments that can be configured

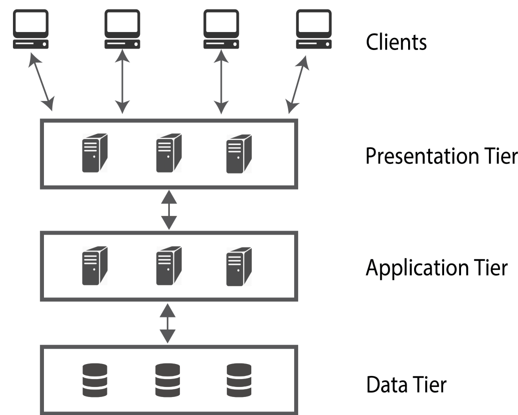


Figure 1.1: A simple multi-tiered application

in multiple ways to change their performance characteristics. For example web services run on server applications that provide them with required functionalities. These applications can be configured in multiple ways by setting the value of a set of configurations.

This provides a large number of configurable parameters that can be used to change the behavior of the servers. Enterprise level server software usually used in similar situations such as Apache Server, Tomcat Server and MySQL Database server each exposes hundreds of configurable parameters. By tuning these parameters user are able to get the best possible performance out of a system.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Usually this is handled by system administrators based on their expertise and using trial and error experiments. This is made harder due to factors such as the wide variety in application behavior, differences in available technology, and variations in load and user behavior. So achieving good performance out of these systems is a hard process.

Significant amount of work has been done to simplify this process. Some people have focused on developing mathematical models that can be used for performance prediction, configuration management, etc. But these methods require adapting the model to specific scenarios through model parameters that need to be identified. Other tuning methodologies have been proposed that focus on a very small number of parameters. But this ignores a large portion of tunable

parameters that can yield improvements in performance.

1.2 Proposed Solution

Autotuning mechanisms have been used in similar situations previously and they have been shown to provide increases in performance with no changes being made to the original application. Autotuning methods provide ways of identifying optimal configurations by searching through possible values automatically.

In this work we are exploring the use of OpenTuner[1], an autotuning framework developed by Ansel et al, towards the tuning of applications deployed in a tiered setup. Opentuner provides the basic building blocks needed to develop a domain specific tuning system.

We propose an Opentuner based autotuner to search through the possible configurations to identify a better performing configuration.

1.3 Contributions

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

We make the following contributions in this thesis:

- A framework for developing offline tuners that can generate significant performance gains in multi-tiered applications
- A methodology to reduce the size of a configuration space automatically thereby reducing tuning time
- Analysis of parameters and their contribution to performance in a common multi-tiered application setup
- An online autotuner to tune multi-tiered applications and a study of its effectiveness.

1.4 Organization

The rest of this document is organized as follows. Chapter 2 presents the past work done in the area of autotuning and performance tuning of multi-tiered ap-

plications. Chapter 3 presents the methodology we used in our offline autotuner. Chapter 4 describes the experimental testbed used by us in out tests. Chapter 5 discusses the results of offline autotuning and methods of reducing tuning time. Then we look at online autotuning. We conclude this document with a discussion on the results obtained.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Chapter 2

LITERATURE SURVEY

In this section we take a look at the prior work done in the area of tuning multi tier applications. First we look at common performance tuning methodologies and frameworks. Then, we look at how these methods and other are used in the context of Multi-tiered applications.

2.1 Auto-tuning

Autotuning is a research area that has been gaining popularity in the recent years. Especially in the areas of high performance computing, computer vision and other performance critical areas.

Performance tuning is used to identify the best possible implementation for an program. Possible implementations of the program have to be searched to identify the optimal one. This used to be done manually by developers working to optimize programs for new platforms. This becomes a very hard task with the vast amount of platforms available today, possible resource limitations, and other factors that affect performance. Auto-tuning methods can take away the burden of performance tuning from developers and allow applications to perform at their best potential.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Autotuners usually consists of a method of representing all possible implementations of a program and a set of methods to optimally search through the implementations looking for the best.

Auto-tuning methods can be broadly broken down into two possible methodologies,*Offline auto-tuning* and *Online auto-tuning*. The distinction among the two methodologies comes down to the adaptability of tuning methods and the positioning of the tuning process in the application's life cycle.

2.1.1 Offline auto-tuning

Offline auto-tuning methods are distinguished by their distinct tuning phase that comes up before an application is deployed. Before the actual use of the application a separate tuning phase is used to tune the application and identify the best possible configuration for deploying the application.

These autotuning methods works by either using code variant tuning methods or parameter tuning methods. Code variant tuning work by using code transformations that changes the final compiled version of the program. These methods can be developed using advanced compiler techniques. Parameter based methods work by finding the optimized set of values to the parameters being tuned. The difference being that it doesnt affect the actual code of the program it self.

ATLAS[1] is a linear algebra library that is autotuned. The tuning process described here consists of two approaches. The first is *parameterized adaptation* used to tune parameter values such as block sizes in distributed algorithms. The other approach is *source code adaptation* where the compiler changes the code itself with methods such as loop unrolling. Several versions of *performance critical routines* are developed and the best is identified by searching through them.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

PHiPAC[2] presents another way of generating hardware optimized linear algebra libraries using a optimizing compiler. Using the results of ANSI C compilers, they derive rules that can be used to specifically tune matrix multiplication scripts for multiple platforms. Offline tuning methods are very popular and widely used.

Orio [3] is another autotuning system that works through code transformations. Orio takes annotated C source code as input, generates many versions of the code, and searches for the best performing version of the program by going through these versions.

Major downside to these approaches is the lack of adaptability. If the program environment changes after the tuning process is completed the whole tuning needs to be done again to re-optimize the program. This is not possible in dynamic rapidly changing environments.

2.1.2 Online auto-tuning

Online autotuning is also known as adaptive/dynamic autotuning. These methods work alongside the program being tuned and does not require a dedicated tuning phase. These methods consists of either a linear model of the program that is used to test performance characteristics or some machine learning method that uses feedback from the program to do the tuning.

The lack of a separate autotuning phase allows online tuning systems to adapt with changes in the environment at program runtime, providing better performance. But the tuning program itself can negatively effect the performance of the tuned program and can cause disruptions to the program's normal flow.

PowerDial[4] is one such system. It transform static application configuration parameters into dynamic control variables called *dynamic knobs*. These *knobs* are then tuned at runtime to minimize system resource usage. It uses a framework that provides system performance information at given periods to monitor the system and make the necessary changes.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Ansel et al. presented a way of using using local competition to performance tuning in SiblingRevelry[5]. This approach separates the available resources into two groups, safe and experimental. Experimental resources are used to search for better performing implementations. If found, the better performing version is copied to the safe subsystem. This ensures reliability of the system at the cost of resource waste. Half the resources are used for tuning making the identified good implementations only valid for half the available resources.

2.1.3 Hybrid approaches

Hybrid approaches to autotuning have also been proposed that include a initial offline training phase and an adaptive online phase. The offline phase is used to model the performance characteristics of the system. This model is then used to adaptively tune the system at runtime.

Ding et al. [6] presents a system where the system is tuned to perform the best for the given input. They achieve this by breaking the input space to clusters

in an initial offline phase. The for each identified cluster the best configuration is identified. When the program is running and it receives a new input, fist the relevant cluster is identified and the previously recognized configuration is used to maximize performance.

In AROMA[7] a similar method is introduced to tune MapReduce queries for performance. It works by the assumption that querries with similar resource usage have similar performance characteristics. So in an initial phase a set of example queries are clustered based on their resource usage patters. For new requests the relevant cluster is identified and the identified configuration is used.

2.2 Performance tuning of multi-tiered applications

With the popularity and importance of Multi-tiered applications on the Internet, their performance is a major concern. So, a lot of research has been performed in this area from around the world. Several people have worked on performance modeling and tuning implementing both offline and online autotuning systems.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

2.2.1 Model based approaches

One major method of tackling performance in multi-tiered applications is the use of model based approaches. Many people have worked on modeling multi-tiered applications. Usually the tiers in a multi-tiered application is modeled as a set of queues. This allows for queuing theory knowledge to be used to make the necessary calculations required.

For example, Menascé et al. presents one such approach[8]. In their proposed approach they model a multi-tiered application as a set of queues. For queuing theory calculations some values such as visit ratios and service times needs to be estimated. These values are estimated by monitoring the system. The parameters that needs to be tuned are also incorporated into the model itself(Ex: Number of threads in a server affects the service time). This allows the to use a simple hill climbing method with the model to search for the best configuration without affecting the online application.

Many other models such as the above have been presented[9][10][11]. These approaches vary by the factors they have captured in their model and the approach they use to predict performance.

Model based approaches allow for quick estimation of performance for a given configuration and there by faster performance tuning. But they do come with a some drawbacks. Modeling how configuration parameters affect performance is not a trivial task. Some parameters such as garbage collection parameters in JVMs provide huge challenges to modeling in a queuing model. This make tuning large set of parameters very difficult.

2.2.2 Learning based approaches

A common approach that have been used in performance autotuning for multi-tiered application is the use of machine learning algorithms to try configurations and use their feedback to search through the possible configuration space.

Zheng et al. presents one such approach[12]. They use a tuning framework called Active Harmony to autotune a set of configurations in multi-tiered applications. They tune a set of 20 parameters using Nelder-Mead method [13]. It is a simplex method for finding a local minimum of a function. This approach has the same problem of not scaling well with the number of tuning parameters.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Bu et al. [14] uses reinforcement learning to tune 8 parameters in a multi-tiered setting. Reinforcement learning allows for very fast searching through the configuration space but suffers when the configuration space grows.

In [12], Zheng et al. uses a parameter dependency graph to reduce the number of parameters that needs to be tuned in another tuning system using the Nelder-Mead method. They use an online system of monitoring a multi-tiered application. When a change in the environment is detected they use the parameter dependency graph to identify the parameters that need to be adjusted. Then those selected parameters are tuned. This a one way of overcoming the size of possible configuration space.

Other approaches have also been proposed to handle performance tuning[15][16][17].

These approaches also follow similar methodologies varying only by the learning methods used and other optimizations.

These approaches have the benefit of being black box approaches that do not require changes to the program or server applications from developers. No modeling of resources or additional work from the developers is needed to apply these learning based methods as opposed to the model based approaches.

All these learning based methods have the same drawbacks. Because they use the feedback from configurations to generate better ones, bad configurations have to be tried which can cause disruptions to the tuned program. In addition reinforcement learning and other similar learning techniques do not scale well with large numbers of tuning parameters.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Chapter 3

METHODOLOGY

We used OpenTuner [18] to build the autotuners for Multi-tiered applications. Opentuner is a open source framework for creating domain specific autotuning applications. It consists of a method of representing a complex configuration space and methods of manipulating the configurations. In addition it also provides a large number of search methods that are used to go through the configuration space looking for the optimal configuration.

The structure of the framework is shown in figure 3.1. Three main components are required to create a autotuner using Opentuner. First we need a definition of the configuration space describing the parameters used in the tuning and the possible values they can take. Then we need a way of manipulating and setting the configurations in the system. Finally a measurement driver is required to monitor the system and report the results to Opentuner.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

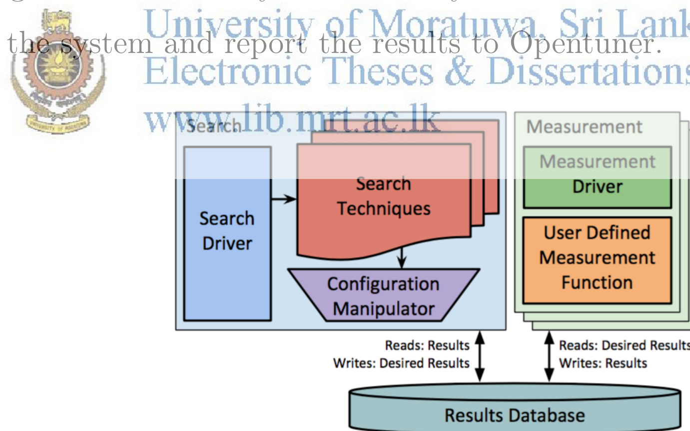


Figure 3.1: Major components of Opentuner

So we needed to identify a set of parameters that can define the configuration space. We did this by looking at prior work on the field and by consulting the documentation of the server applications. Each server application in the system exposes hundreds of tunable parameters. From these we selected a set that can be used to prove the effectiveness of the tuning process. The selected parameters are shown in table 3.1.

Server	Parameters
Apache	MaxConnectionsPerChild, StartServers, MinSpareThreads, MaxSpareThreads, ThreadsPerChild, ThreadLimit, MaxRequestWorkers
Tomcat Server	maxThreads, acceptCount, acceptorThreadCount, minSpareThreads, maxConnections
	Java Virtual Machine parameters
MySQL Server	table_cache, query_cache_size, sort_buffer_size, thread_stack, query_cache_limit, read_buffer_size, max_connections, thread_cache_size, key_buffer_size, innodb_buffer_pool_size

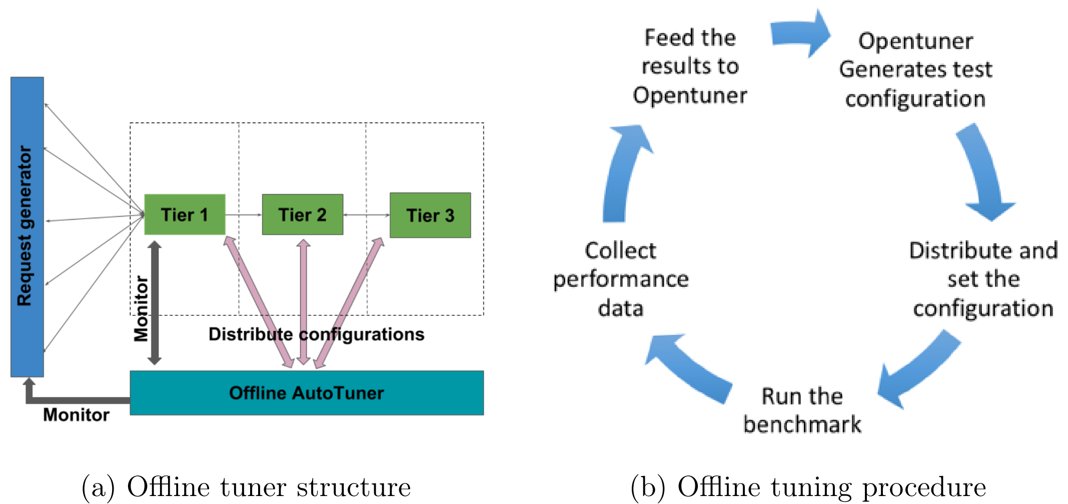
Table 3.1: Parameters used in the tuning process

In addition to the server parameters we also focused on the tunable parameters of the underlying Java Virtual Machines(JVMs). Tuning this JVM allows us to get bigger performance gains. Opentuner allows us to handle the large number of additional tunable parameters that are added here but the tuning can be slow due to having a bigger configuration space to look through. To overcome these issues we used the prior work done in JATT[19]. Their work focused on developing a hierarchical structure to represent the possible configuration space in JVMs. By using this *flag structure* they were able to avoid erroneous flag configurations and speedup the tuning process.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The offline autotuner we developed consists of a single node monitoring the system and distributing the testing configurations. The structure is shown in figure 3.2a Opentuner generates sample configurations that need to be tested to identify their performance. The autotuner then distributes these configurations, runs the benchmark application, monitors the system to identify the required performance metrics and then reporting the feedback to Opentuner. The results are used in the generation of new test configurations that need to go through the same procedure. the procedure is shown in Figure 3.2b

Using this procedure allows us to be very flexible in our autotuner. We are able to tune programs for any performance goal that we can set. We are also able to handle systems deployed in any architecture by making simple changes to the scripts used to set the configurations.



(a) Offline tuner structure

(b) Offline tuning procedure

Figure 3.2: Offline autotuner

We tested our tool is a experimental setup using two benchmarks and tuned the benchmarks for multiple performance goals. In each of the tuning experiments the tuner was run for 24 hours. This is an arbitrarily defined time limitation and can be changed. We selected this limit to allow the autotuner enough time to identify a better configuration compared to the default. Further discussions about the tuning time can be found in section 5.3

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Chapter 4

EXPERIMENTAL SETUP

The experiments on the autotuners were conducted using two popular benchmark web applications. These benchmarks consist of a benchmark application that can be deployed in our system and a request generator that is used to get the benchmark application working in a setting similar to an actual web application.

4.1 RUBiS Benchmark

RUBiS[20](**R**ice **U**niversity **Bi**dding **S**ystem) is benchmark specification for an auction prototype developed at Rice University. The benchmark specifies the structure of a auction website and the possible interactions with users. In this research we used a Java Servlet implementation of the benchmark. This implementation provides a client application that can be used to simulate the behavior of a given number of potential customers.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

4.2 TPCW Benchmark

A transactional web e-Commerce benchmark based on the TPC-W specification. We used a Java Servlets version of the benchmark[21] developed at University of Wisconsin - Madison. The benchmark also provides a client simulator with multiple behavioral patterns. We are able to generate requests from a given number of simulated clients with a pre defined behavior pattern. Two main metrics of performance are considered by this benchmark.

- WIPS - Web Interactions Per Second
- WIRT - Web Interaction Response time

4.3 Deployment Environment

These two applications were deployed in a three tier setup. Each tier in the application was deployed in a separate server with the following configuration

```
Intel Core i7 CPU @3.40Ghz (4 cores)
16 GB RAM
Ubuntu 12.04 LTS
OpenJDK 7 (HotSpot JVM) update 55
```

The Autotuner and the client emulator was run on a seperate server with the following configuration.

```
Intel Xeon E7-4820v2 CPU @2.00 GHz (32 cores)
64GB RAM
Ubuntu 14.04 LTS
OpenJDK 7 (HotSpot VM) update 55
```

Following server applications were selected for the system.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

- Presentation tier - Apache 2.4.7
- Application tier - Apache Tomcat 7.0.47
- Data Tier - MySQL Server 5.5.46

All these software are free and open-source. Together they makes up the most popular web server configuration used by users. This setup was used as a simple testbed that is easy to understand and experiment on. But the auto-tuner can be easily extended to support any other configurable server application.

Chapter 5

OFFLINE TUNING RESULTS

Both benchmarks were tuned for multiple performance goals. We were able to generate significant performance gains in all cases. The performance metrics we used were,

1. Average Response Time : Average of the time taken to complete a request sent by a user in a given time period. measured from the client's perspective.
2. Average Throughput : Average of the number of requests correctly handled from the program in a given time. the throughput is calculated per minute and the average value at the end of monitoring is used.
3. 99th percentile average response time : When dealing with large number of requests in a web application some of them tend to fail or take longer. These outliers affect the average response time even though they are unavoidable. So this metric is calculated by removing these outliers.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

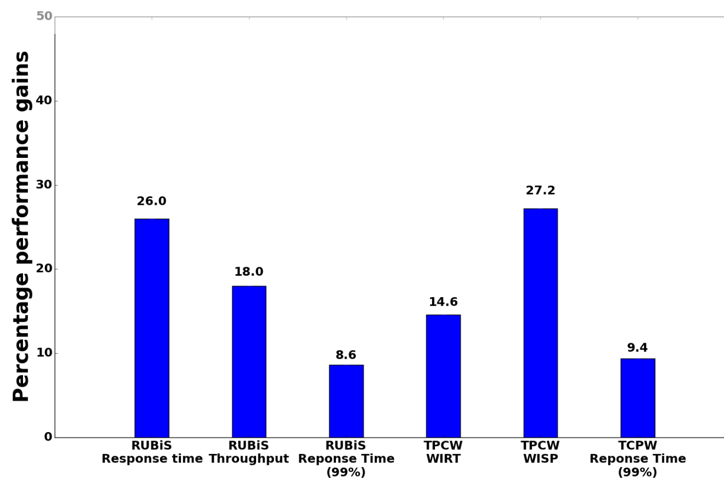
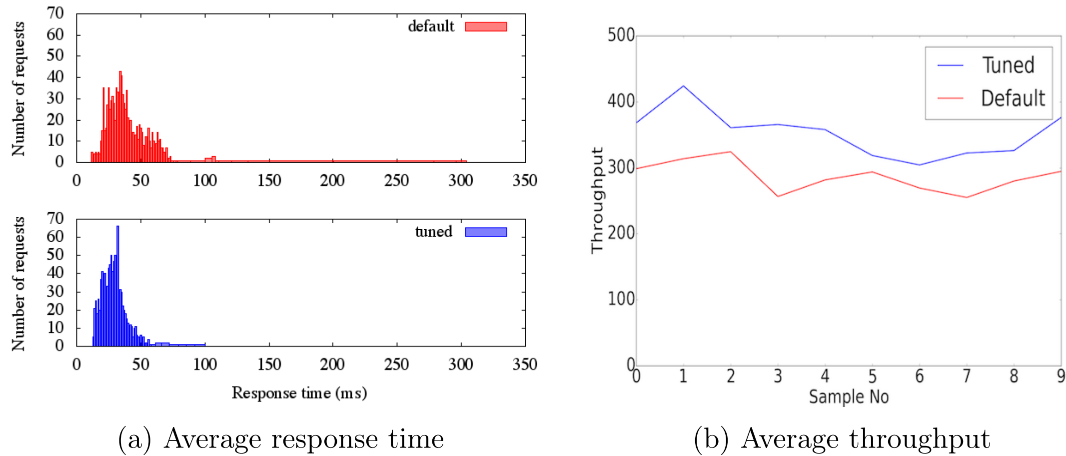


Figure 5.1: Percentage performance gains achieved from tuning

The performance gains are summarized in Figure 5.1. These gains show the improvement of the performance against the default configuration available in the system.



(a) Average response time

(b) Average throughput

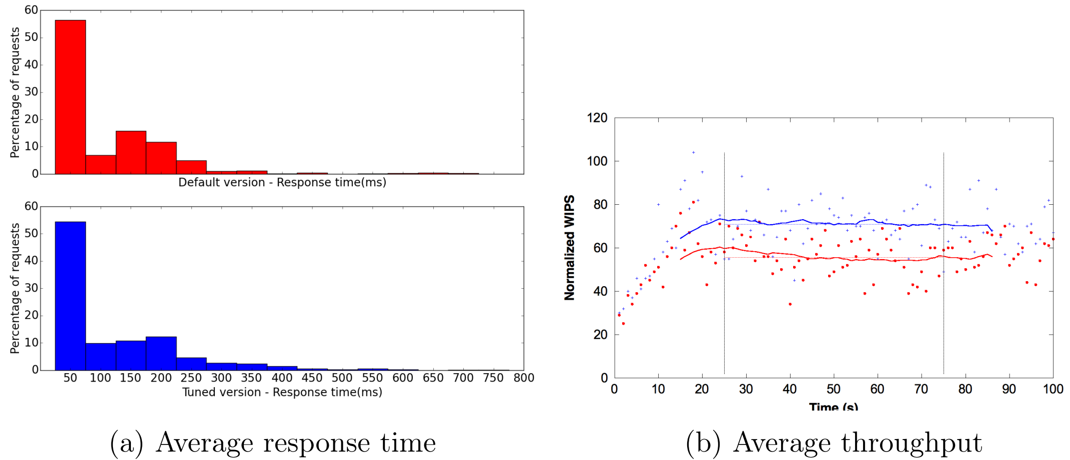
Figure 5.2: RUBiS benchmark tuning results

5.1 RUBiS results

We were able to observe significant performance gains in the RUBiS benchmark for all benchmarks after the tuning process. For the tuning we used the emulated clients provided with benchmark. We used a set of 500 clients to generate the requests concurrently. The system was then tuned to maximize/minimize certain performance goals.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

1. Average Response Time : We got an 26% reduction in average response time as shown in Figure 5.2a. This figure shows the comparison between the response time we get from the default configuration against that of the tuned system. We can also see that significant reduction in the tail of the response time histogram was achieved by the autotuner. This was achieved by improving the concurrency in the program. Being able to handle more concurrent results allowed a reduction in average response time.
2. Average Throughput : We were able to observe a 18% improvement in average throughput. Figure 5.2b shows that through multiple measurements the throughput of the tuned program outperformed the default program.
3. 99th percentile average response time : We observed 8.8% reduction in average response time when the outliers with long response times were ignored. This is expected as we saw when tuning for average response time, reduction in the tail end is not enough to increase the overall average response



(a) Average response time (b) Average throughput

Figure 5.3: TPC-W benchmark tuning results

time because the number of outliers are somewhat limited.

5.2 TPC-W results

Similar to the RUBiS benchmark, we used client scripts to emulate the behavior of actual clients in the system. We used 500 clients to generate the results. We then used 50 seconds to ramp up the system, monitored the system and ramped down as per the benchmark specification. The performance metrics were collected in the monitoring phase after the ramp-up period. We were able to observe good gains in performance for multiple performance goals.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

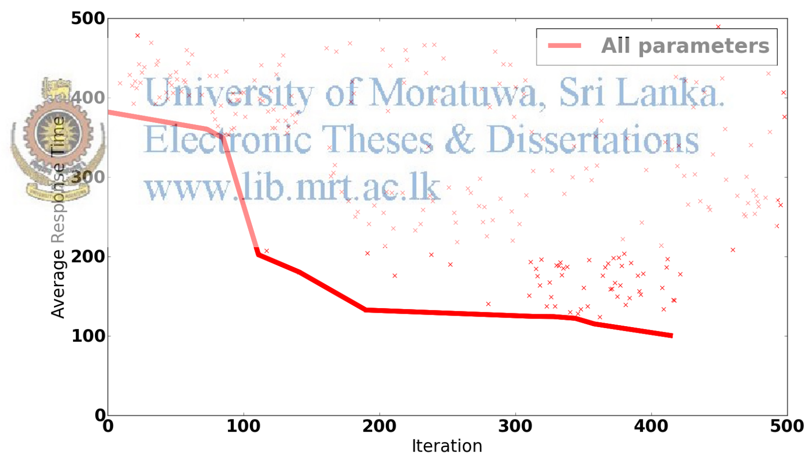
1. Average Response Time : In the TPC-W benchmark the response time is calculated as the **W**eb **I**nteraction **R**esponse **T**ime. This measures the time taken to successfully complete an interaction between the client and the server. We got an 14.6% reduction in WIRT as shown in Figure 5.3a.
2. Average Throughput : In the same way, throughput is measured in **W**eb **I**nteractions **P**er **S**econd. This is a measure of the rate of interactions the user can make with the program. We were able to observe a 27.2% improvement in average throughput. Figure 5.2b shows that through multiple measurements the throughput of the tuned program outperformed the default program.

3. 99th percentile average response time : We observed 9.4% reduction in average response time when the outliers with long response times were ignored. This observation is similar to what we saw in the other benchmark due to similar reasons.

5.3 TUNING TIME

As mentioned earlier, the autotuner in each of the above experiments were run for 24hrs. OpenTuner uses many search algorithms to go through the configuration space and these need to test many configurations before identifying good configurations that can improve our performance metrics.

Consider the tuning run represented in Figure 5.4. This shows the result of one tuning run that was used to tune the RUBiS benchmark to minimize response time.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 5.4: Tuning the RUBiS benchmark for response time.

In the tuning process, Opentuner generates test configurations that needs to be run and the result recorded. Each of the red dots in the graph represents one such recorded instance of response time. This graph represents 500 such test configurations and the resultant response time. And the red line connecting them connects the best possible configurations identified.

So, we can see that we have to go through a large number of test configurations before arriving at a result that gives a significant improvement in performance.

For each of these test configurations we have to set the configuration in the system, run the benchmark and record the performance metric. This results in the tuning time growing bigger delaying the program deployment.

One major cause of this increase in the number of test configurations is the size of the configuration space it self. Due to the large number of parameters we use in the tuning process the configuration space is growing exponentially. Reducing the size of the configuration space by removing some parameters from being considered for tuning can lead to the number of configurations that need to be tested being reduced.

But reducing the number of tuning parameters need to be done carefully. Not considering a performance critical parameter in the tuning stage can lead to us loosing a potentially large performance gain. So identifying and removing parameters that don't have a significant impact in performance is very important.

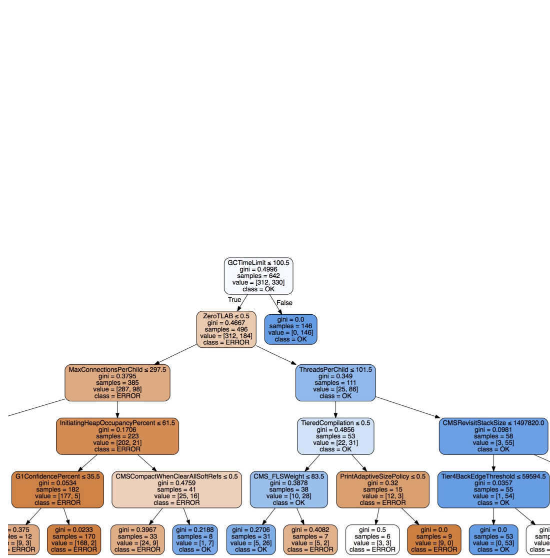
Manually going through each of the 250+ parameters to look at its contribution to performance is not a trivial task. We also need to have a system that can be generically applied. So adding new parameters to be tuned should be easy and the user should not have to do the same analysis for every new parameter that needs to be tuned. So, an automated method is very important.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

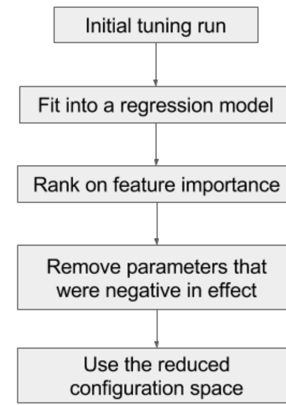
The method proposed by us is to use the results of a initial offline tuning phase to prune the configuration space. This initial tuning phase consists of running a set of test parameters and collecting the performance record for each of those test configurations. This data can be then used to model how changing each parameter affect the performance and how much a parameter contribute to performance.

First the performance data is modeled as a Random Forest Regression[22]. Random Forest Classifiers and Regressors are a very common ensemble method of modeling large data sets foe prediction and analysis. They use a set of decision trees that are trained with a dataset. Then the mean of their output is considered for future prediction needs. we used the implementation of Random Forests available with the scikit-learn[23] API.

This allows us to calculate the *feature importance*. This is a representation



(a) Example decision tree from a Random Forest used in parameter pruning.



(b) Steps in the parameter pruning process

Figure 5.5: Process of pruning the configuration space

of how changing each parameter can affect performance. identify features which have lower importance and therefore does not contribute to a change in performance. We can then remove these parameters as they are not required for tuning. This prunes the configuration space.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

This method works as follows. The random forest consists of a set of decision trees similar to the one in Figure 5.5a. For each such tree in the random forest go to each internal node, calculate the error reduction by that node using the gini index and add the values to get *feature importance* from a single tree. Average the values over all the trees in the forest to get the overall *feature importance*

But this only considers the effect on the performance metric but not the quality of the effect. Consider for example an optimization method that is turned on by default. Disabling the optimization leads to poor performance. But, since this parameter has a big effect on performance it is given a higher importance. Tuning such parameters is futile because it is already at the best position in default.

To identify such parameters we used a simple heuristic of comparing the average value of a parameter in the best performing configurations with its default

value. For parameters with higher *importance* if the values are similar then we can estimate that changing the value does not affect performance positively. While this is not the perfect solution this provides enough insight to suit our tuning needs.

The full process of parameter pruning is given in Figure 5.5b. To test the effectiveness of the process we ran experiments by reducing the configuration space, first to 50 parameters and then to 25 parameters, for both the benchmarks. Figure 5.6 shows the results of the tuning runs.

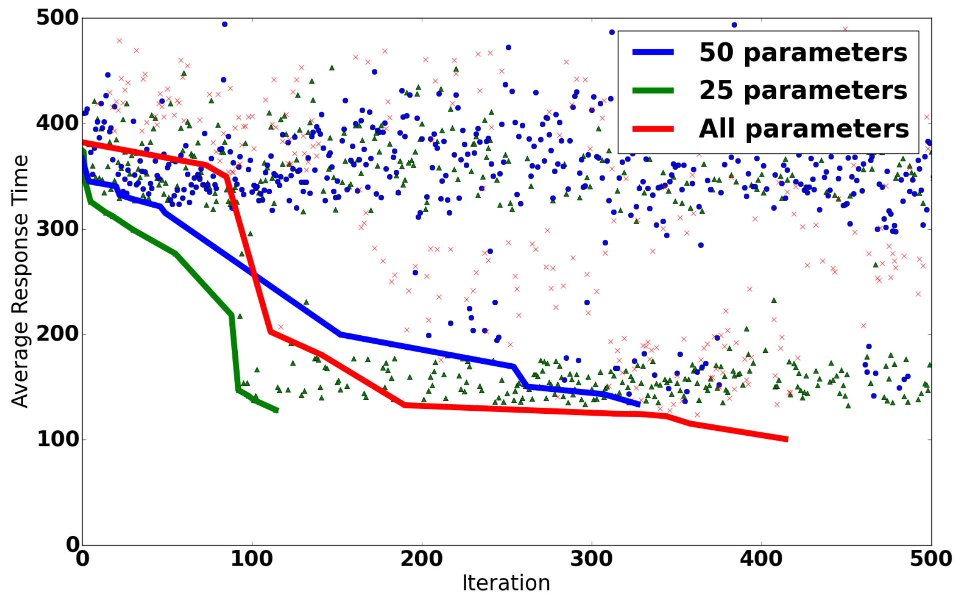
Consider Figure 5.6a as an example. This graph shows the result for tuning the RUBiS benchmark to minimize throughput. The red graph shows the results of tuning the program with all the parameters available(Same as in Figure 5.4). The blue line represents the tuning done with 50 parameters selected using the method discussed above. The green graph is with 25 parameters.

As can be seen clearly the smaller configuration space leads to faster tuning times. We were able to generate significant performance gains within a very small time span. But even though they are fast, we don't see the full performance gains that can be achieved when all the parameters are used.

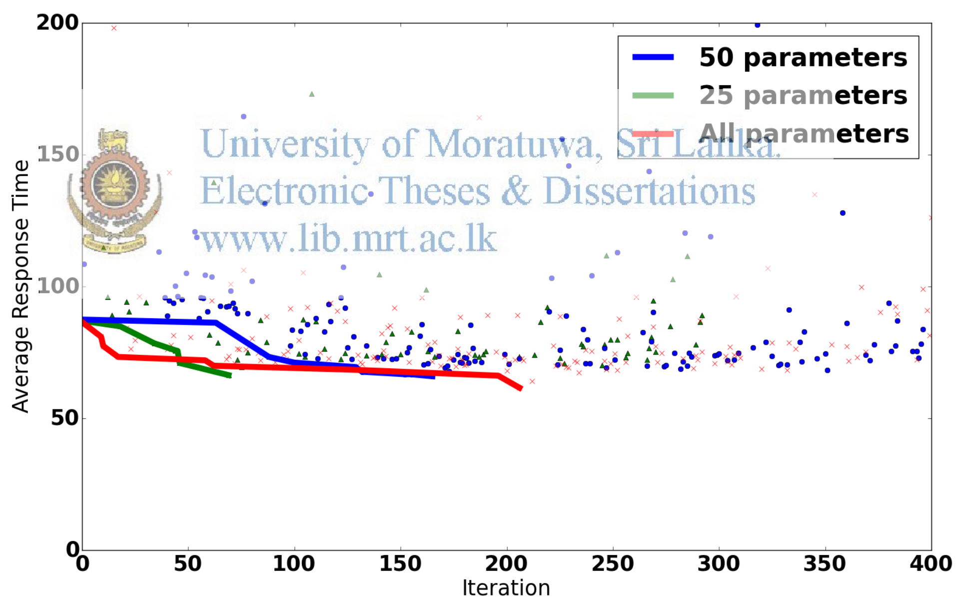
University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

The graph here is limited to 500 iterations of the tuning process to ensure clarity. But we observed that given enough time the full configuration space yields far better performance gains. The same can be set of the 50 parameter configuration space in that it is able to generate better performance compared to the smaller 25 parameters. This is expected and presents us with a trade-off between the tuning time and the performance gains.

The trade-off is clearer when the configuration space is further pruned. Removing more tunable parameters makes the performance gains less significant. We believe that for these two benchmarks and this system 25 presents the best of both performance gains and faster tuning. But this automatic process can be used with other systems to identify the best trade-off for them.



(a) RUBiS Benchmark



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

(b) TPC-W Benchmark

Figure 5.6: Tuning the Benchmarks with differently sized configuration spaces

5.4 OFFLINE TUNING DISCUSSION

As we saw we were able to generate good performance gains for both benchmarks for multiple performance goals. We wanted to see if there is a optimal configura-

Parameter	**Tier**
ThreadsPerChild	Apache
max_allowed_packet	MySQL
AdaptiveSizePolicyCollectionCostMargin	Tomcat
BaseFootPrintEstimate	Tomcat
ThreadLimit	Apache
thread_cache_size	MySQL
YoungPLABSize	Tomcat
ParallelGCBufferWastePct	Tomcat
max_connections	MySQL
GCTimeLimit	Tomcat

Table 5.1: Parameters that contribute most to the performance. Ranked according to their contribution

tion that can generate performance gains for any application. What we observed is that better performing configurations vary a lot between the two benchmarks and between the performance goals. So, there is no *silver bullet configuration*. This further emphasizes the need for a autotuning system. As there is no one perfect configuration, each time a new program is deployed the optimal configuration can change. The best way to overcome this burden on tuning is to use an auto tuner.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Even though there is no optimal configuration that works in any scenario we identified a set of parameters that seem to contribute the most to performance. We used the method described in the configuration space pruning to rank the parameters according to their contribution to changing the performance. The top parameters we identified are presented in Table 5.1. We see a lot of parameters that directly contribute to the amount of concurrent requests a system can handle in the top parameters. This confirms our intuition that concurrency is the biggest bottleneck for performance in these cases.

We also looked at other methods of tuning a multi-tiered system. One such scenario we looked at is tuning the components of the system separately and combining the partial configurations we get to get the full optimized configuration required. As we expected this method did not perform as well as when we considered the system as a whole. This is not very surprising as treating the tiers as

Experiment	Performance Gain
Apache server	13.4%
Tomcat server	8.75%
MySQL server	10%
Combining the separately identified configurations	10.84%
tuning the system as a whole	14.6%

Table 5.2: Gains in response time for TPC-W benchmark from tuning individual tiers.

separate fails to take into account the interactions between configurations across the tiers. The results of this experiment is shown in Table 5.2. In addition we can also see that the parameters we iden- tified as the most important towards performance are spread across the tiers. This further confirms the need of tuning the system as a whole rather than as individual elements.

Even though we see good performance gains from the offline autotuning we also can see some drawbacks to the tuning system.

- As there is a separate tuning process before the application is deployed the autotuner needs to know about the environment the program will be deployed in. With these benchmarks we used a set of simulated clients and only had to estimate the load that will be used in deployment. But for real world usage this presents some difficulty.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

- The tuning process we use here is going to cause disruptions in a deployed program during the tuning phase. most of the tunable parameters available in the server applications used by us requires restarting the application for changes to take place. Since we are using test configurations generated by Opentuner they can be invalid or very poor performing configurations.

- In addition, this tuning process is an off-line process. So it is unable to adapt to changes in the environment an application is deployed in. Therefore a configuration identified as optimal can become sub-optimal with changes in the system. Modern methods of deploying web applications such as auto-scaling systems also makes off-line tuning methods less accurate as the runtime system is very hard to replicate in an offline setting.

It is difficult to overcome these issues in any offline tuning mechanism. So the solution would be to move into an online tuning system. But this introduces a lot of difficulties that needs to be overcome. The next chapter looks into the work done by us in online autotuning.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

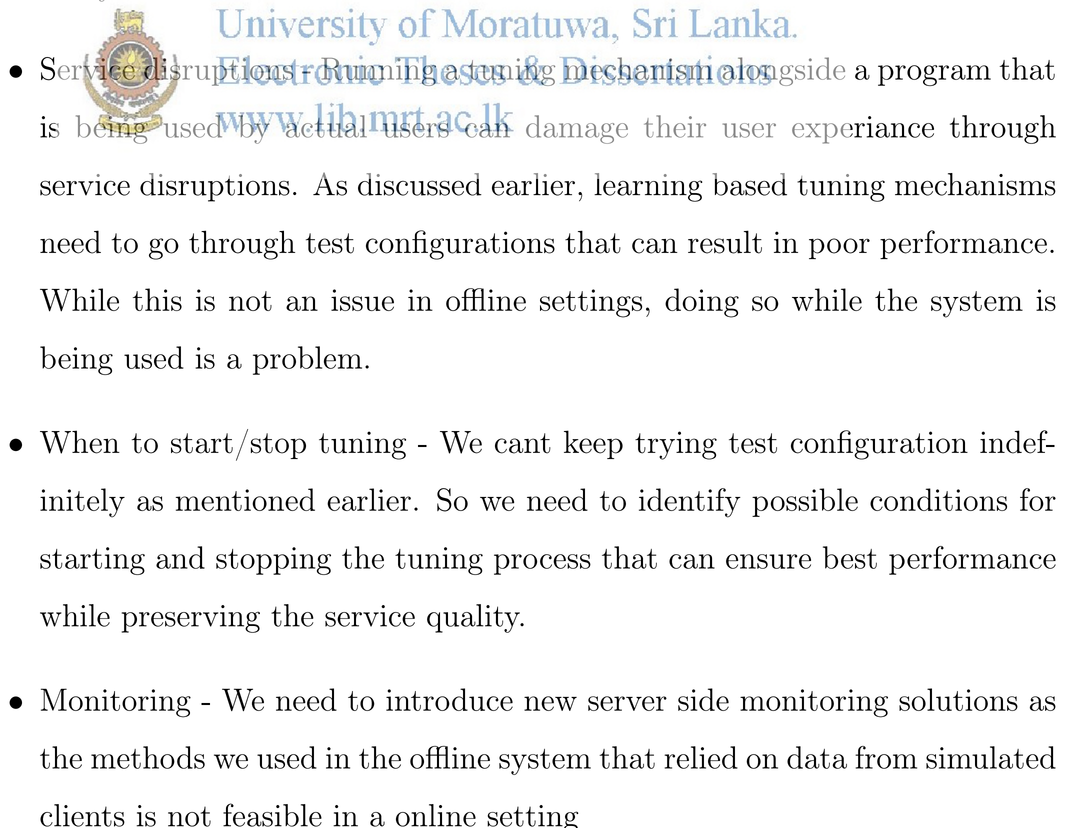
Chapter 6

ONLINE AUTOTUNING MULTI-TIERED APPLICATIONS

The major drawback of offline tuning methodologies is the lack of adaptability. Once a optimized configuration is found through the offline tuning phase it doesn't change with changes in the environment. This lack of adaptability is major problem for real life scenarios in modern web services where the environment goes through rapid changes due to factors such as scaling, changes in user behavior, shared resource usage, etc.

The solution to overcoming these issues is to introduce a tuning system that constantly monitors and adapt to changes in the environment. These types of tuning systems are known as online tuning mechanisms.

But online tuning systems have a lot problems that needs to be overcome before they become feasible.



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

- Service disruptions - Running a tuning mechanism alongside a program that is being used by actual users can damage their user experiance through service disruptions. As discussed earlier, learning based tuning mechanisms need to go through test configurations that can result in poor performance. While this is not an issue in offline settings, doing so while the system is being used is a problem.

- When to start/stop tuning - We cant keep trying test configuration indefinitely as mentioned earlier. So we need to identify possible conditions for starting and stopping the tuning process that can ensure best performance while preserving the service quality.

- Monitoring - We need to introduce new server side monitoring solutions as the methods we used in the offline system that relied on data from simulated clients is not feasible in a online setting

- Fast tuning - To be able to adapt to changes in the environment we need our tuning algorithms to yield fast results that can be used in time. Slower methods can produce more system disruptions due to longer tuning times and makes the tuning system less responsive.

We experimented on some methods of performing online autotuning that try to overcome these issues.

6.1 Simple online autotuning

One approach we attempted is the simple application of Opentuner to online autotuning ignoring the service disruption caused by the tuning process. We setup a autotuner similar to the one used for offline tuning. We used a simplified configuration space identified by the method described earlier.

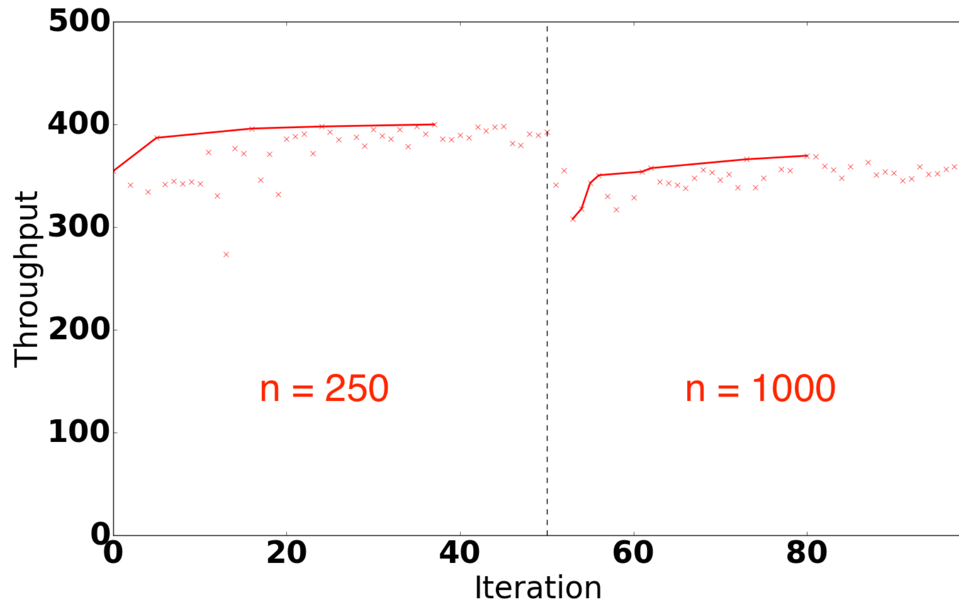
We used a threshold value of 10% to define the rules for starting and stopping the tuning process. So at the start the autotuner will tune the program until a 10% improvement in tuning is achieved. Once it is done the autotuner will go into the monitoring phase and monitor the systems performance. If due to any change in the environment the performance of the system go up or down by the same margin the tuner will start again and will attempt to autotune the program again.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

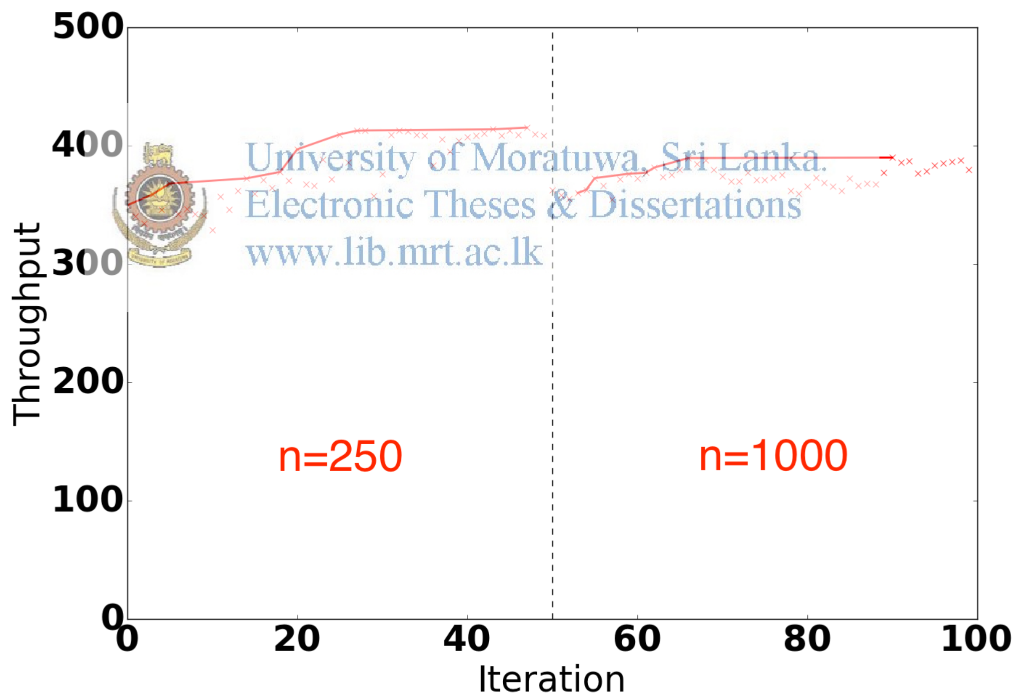
We looked at using two search algorithms for this. Opentuner by default uses evolutionary algorithm based search methods to go through the configuration space. This leads to large amounts of illegal configurations as the result of the randomized nature of the algorithms. In addition no guarantees can be provided about the tuning time again due to their random nature.

So in addition to the default methods used by Opentuner we implemented our own simple hill climbing method that can go through the configuration space. This search technique helps us to reduce the amount of invalid configurations that would have to be tested in our system.

Figure 6.1 presents the results of tuning the RUBiS benchmark to maximize throughput using the two methods. To simulate a change in the environment



(a) Using the default search methods on Opentuner



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

(b) Using a simple hill climbing search algorithm

Figure 6.1: Tuning the RUBiS benchmark to maximize throughput with online autotuning

after 50 iterations of the tuner the number of request generators were increased to 1000 in both cases. Each dot represents a measurement of throughput gathered

Tuning Methodology	Performance Gain	Invalid configuration Rate
Default Algorithms (Evolutionary)	5.4%	8 per 50
Hill Climbing method	6.5%	0

Table 6.1: Comparison of online tuning algorithms

after monitoring the system for 1 minute.

Both the search algorithms were able to identify better configurations compared to the default that can provide up to 10% in performance gains. In terms of tuning performance both algorithms performed similarly.

The main difference is in the variability of the performance. As can be seen from the graph the evolutionary algorithms tended to generate configurations with degraded performance more regularly compared to the simple hill climbing method. The number of invalid configurations fell dramatically too. In experiments run by us the default evolutionary algorithm used by Opentuner tended to generate around 8 invalid configurations per fifty tied. The hill climbing method was able to avoid generating invalid configuration altogether in most runs.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

We also wanted to see how the tuning process affected the overall performance. At the end of all the tuning we were able to observe overall performance gains in both search methods despite the overheads and the restarting times involved. These information is presented in Table 6.1

6.2 Sibling Revelry based methodology

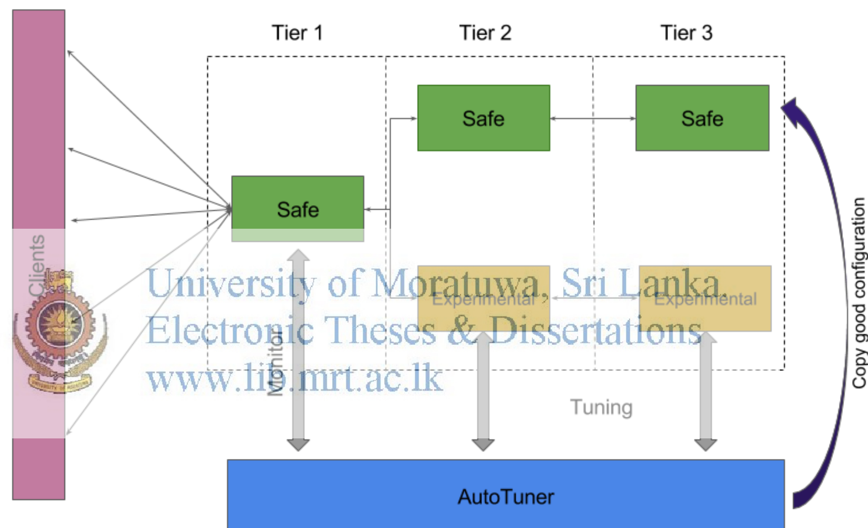
One major drawback of the simple online autotuning methodology is the service disruptions cause by either invalid or poor performing configurations. These cause the entire service to be stopped multiple times in the tuning process. In addition setting the configuration in the servers requires a restart that again disrupts the service.

One possible solution to overcome this issue is to add additional resources for autotuning that can save the entire system from having to be stopped regularly. We propose an approach based on Sibling Revelry[5]. In this approach the in-

coming requests from users are load-balanced among two sets of servers *safe* and *experimental*.

The *experimental* servers are used in the autotuning if a good configuration is found in the *experimental* servers they are copied to the *safe* servers.

In this case the *safe* servers ensure service continuation despite the tuning process. Even though we are testing poor configurations in the *experimental* servers the service will not get completely disrupted. While this does not fully eliminate failed requests this can reduce them. Even if a very bad configuration is being tested at least some of the requests will get good performance. The structure of this autotuner is available in Figure 6.2



University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Figure 6.2: Online autotuner based on Sibling Revelry

One major issue with this approach is the lack of tuning done on the front facing tier. Due to load balancing and monitoring reasons we have to remove one entire tier from the consideration for performance tuning.

This limits the possible performance gains achievable through this method. The fornt facing tier is a major factor in the amount of simultaneous requests a system can handle. So it is a major factor in the throughput of a system. Not changing the configuration at that tier causes us to miss out on possible performance gains.

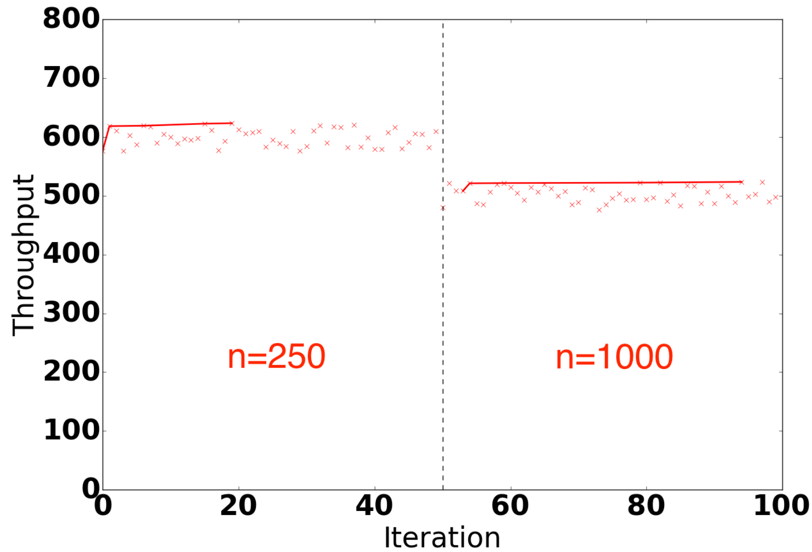


Figure 6.3: Online tuning using the Sibling Revelry inspired method

During our experiment similar to the ones done for other online tuning methods, where we tune RUBiS benchmark to maximize throughput, we were only able to generate 5.4% performance gains on average. This gains are less than what we got with the simple online tuning methods.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

But the major advantage of using this method is the lack of downtime. Since the *safe* set of of servers always run using a configuration that is better than or equally performing to the default configuration we are able to observe stable performance even while the tuner is being run. This is important as the system is being tuned whilst it is being used by actual users.

6.3 Discussion

We looked at three methods of tuning multi-tiered applications online. All methods are able to achieve some level of performance gains and the difference is in the level of reliability that is required in the service being tuned.

The simple online tuning methods cause disruptions to the service at each iteration as the configurations need to be set in the servers. The Sibling revelry based method avoids that by only updating the configuration once a good configuration is found. So the simple methods get better results but at the cost of

reliability.

We chose a simple threshold of 10% for starting/stopping the tuning process. this is not the ideal solution as it is a arbitrarily selected threshold. More complex methods of starting/stopping the tuning process could lead to better results.

All the tuning here were done with very limited configuration spaces being considered. Improving the search techniques used here can enable us to tune more parameters thereby improving the performance gains that can be achieved.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

Chapter 7

CONCLUSIONS AND RECOMMENDATIONS

We have shown that the Opentuner framework can be successfully used to develop autotuners for multi-tiered applications. Our autotuners were able to generate significant performance gains for multiple benchmark applications. The autotuners we introduced are very flexible and can be adapted to suit the needs of any multi-tiered setup. Our approach also allows the tuning of a large number of parameters space thereby attempting to achieve the best possible performance gains.

In addition we have also introduced an automated method of identifying performance critical configuration parameters. This allows us to focus our tuning in these parameters and increase our efficiency.

We also looked at online autotuning and the challenges that online tuning poses. we were able to generate good performance gains through the online tuning methodologies.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

There are multiple ways that this work can be improved on. Our online tuning methodologies still suffer from many service disruptions that needs to be avoided if the system is to become more useful. In addition, We are not focusing on modern trends in the development of multi-tiered systems such as auto-scaling. We can also look at other uses on multi-tiered applications such as the ones in High Performance Computing use cases.

Overall, good progress have been made on autotuning multi-tiered applications and with further improvements we can make even further improvements in the performance of these applications.

References

[1] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.

[2] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[3] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using orio. In *Proceedings of the 2009 ieee international symposium on parallel&distributed processing*, pages 1–11. IEEE Computer Society, 2009.

[4] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ACM SIGPLAN Notices*, volume 46, pages 199–212. ACM, 2011.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

[5] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, pages 91–100. ACM, 2012.

[6] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May OâĂŹReilly, and Saman Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Notices*, volume 50, pages 379–390. ACM, 2015.

[7] Palden Lama and Xiaobo Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 63–72, New York, NY, USA, 2012. ACM.

[8] Daniel A Menascé, Daniel Barbará, and Ronald Dodge. Preserving qos of e-commerce sites through self-tuning: a performance model approach. In *Proceedings of the 3rd ACM conference on Electronic Commerce*, pages 224–234. ACM, 2001.

[9] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. *SIGMETRICS Perform. Eval. Rev.*, 33(1):291–302, June 2005.

[10] Wes Lloyd, Shrideep Pallickara, Olaf David, Jim Lyon, Mazdak Arabi, and Ken Rojas. Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 73–80. IEEE, 2012.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

[11] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 289–302, New York, NY, USA, 2007. ACM.

[12] Wei Zheng, Ricardo Bianchini, and Thu D Nguyen. Automatic configuration of internet services. *ACM SIGOPS Operating Systems Review*, 41(3):219–229, 2007.

[13] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[14] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A reinforcement learning approach to online web systems auto-configuration. *Distributed Computing Systems*, 9:29, 2009.

[15] Palden Lama and Xiaobo Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 151–160. IEEE, 2010.

[16] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *2006 IEEE International Conference on Autonomic Computing*, pages 65–73. IEEE, 2006.

[17] Yixin Diao, Frank Eskesen, Steven Froehlich, Joseph L Hellerstein, Lisa F Spainhower, and Maheswaran Surendra. Generic online optimization of multiple configuration parameters with application to a database server. In *International Workshop on Distributed Systems: Operations and Management*, pages 3–15. Springer, 2003.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk

[18] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: an extensible framework for program autotuning. In *14). ACM, New York, NY, USA,. DOI=10.1145/2628071.2628092*, pages 303–316. Proceedings of the 23rd international conference on Parallel architectures and compilation (PACT, 2014.

[19] Sanath Jayasena, Milinda Fernando, Tharindu Rusira, Chalitha Perera, and Chamara Philips. Auto-tuning the java virtual machine. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pages 1261–1270. IEEE, 2015.

[20] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware archi-

tectures for generating dynamic web content. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 242–261. Springer-Verlag New York, Inc., 2003.

[21] Harold W Cain, Ravi Rajwar, Morris Marden, and Mikko H Lipasti. An architectural evaluation of java tpc-w. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 229–240. IEEE, 2001.

[22] Andy Liaw and Matthew Wiener. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

University of Moratuwa, Sri Lanka.
Electronic Theses & Dissertations
www.lib.mrt.ac.lk