Automatically Tuning Database Server Multiprogramming Level

by

Mohammed Abouzour

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

© Mohammed Abouzour 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Optimizing database systems to achieve the maximum attainable throughput of the underlying hardware is one of the many difficult tasks that face Database Administrators. With the increased use of database systems in many environments, this task has even become more difficult. One of the parameters that needs to be configured is the number of worker tasks that the database server uses (the multiprogramming level). This thesis will focus on how to automatically adjust the number of database server worker tasks to achieve maximum throughput under varying workload characteristics. The underlying intuition is that every workload has an optimal multiprogramming level that can achieve the best throughput given the workload characteristic.

Acknowledgements

This thesis would not have been possible without the help of God, and second without the continuous support and encouragement of my family who gave me the strength and will to succeed. A special thank to my wife for her continued support, patience, and sacrifices for me.

I have to thank my supervisor, Kenneth Salem, for his support, advice, and guidance during my thesis work. His continues guidance helped me complete my work successfully. I would also like to thank my thesis readers Peter Bumbulis and Ashraf Aboulnaga.

Finally, I am grateful to my colleagues at iAnywhere Solutions: Glenn Paulley, Mark Culp, John Smirnios, Ian McHardy, Bruce Hay, and Ivan Bowman who answered my questions about experimental results or internal SQL Anywhere database server issues.

Contents

1 Introduction and Motivation **1**

1.1 Thesis Statement . . . 1

1.2 Motivation . . . 1

1.3 Thesis Organization . . . 2

1.4 Major Thesis Contributions . . . 2

2 Background and Related Work **3**

2.1 Background . . . 3

2.2 Related Work . . . 6

3 Design and Implementation **8**

3.1 Controller Architecture . . . 8

3.2 Auto-tuning Algorithms . . . 9

3.2.1 Hill Climbing . . . 11

3.2.2 Global Parabola Approximation . . . 12

3.2.3 Local Parabola Approximation . . . 14

4 Workloads and Test Methodology **16**

4.1 Database Server Software . . . 16

4.2 Hardware Configuration . . . 16

4.3 Workloads . . . 17

4.3.1 TPC-C . . . 17

4.3.2 DS2 . . . 19

4.3.3 Summary . . . 20

4.4 Methodology . . . 20

5 Workload Characterization **22**
5.1 TPCC1-BIGMEM 22
5.2 TPCC1-SMALLMEM 24
5.3 TPCC2-BIGMEM 26
5.4 DS2 27

6 Algorithm Parameter Sensitivity and Tuning Experiments **30**
6.1 Hill Climbing 30
6.1.1 Varying C 30
6.1.2 Varying Δ 31
6.2 Global Parabola Approximation 33
6.3 Local Parabola Approximation 35

7 Evaluation and Results **37**
7.1 Auto-tuning Experiments 37
7.1.1 TPCC1-BIGMEM 37
7.1.2 TPCC1-SMALLMEM 39
7.1.3 TPCC2-BIGMEM 40
7.1.4 DS2 41
7.2 Changing Workload Experiments 43
7.2.1 TPCC1-BIGMEM to TPCC1-SMALLMEM 43
7.2.2 TPCC1-SMALLMEM to TPCC1-BIGMEM 45
7.2.3 TPCC1-BIGMEM to TPCC2-BIGMEM 47
7.2.4 TPCC2-BIGMEM to TPCC1-BIGMEM 49
7.3 Summary 50
7.3.1 Hill Climbing 51
7.3.2 Global Parabola Approximation 51

8 Conclusion and Future Work **53**

List of Tables

3.1 Standard notation used by the different algorithms 10
3.2 Summary of parameters for tuning algorithms 15

4.1 Standard Transaction mix of the TPC-C workload 17
4.2 TPCC2 workload Transaction mix 19
4.3 Workloads configurations 20

5.1 Characteristics of the TPCC1-BIGMEM workload 24
5.2 Characteristics of the TPCC1-SMALLMEM workload 26
5.3 Characteristics of the TPCC2-BIGMEM workload 27
5.4 Characteristics of the DS2 workload 29

6.1 Various throughput levels with fixed Δ of 10 31
6.2 Various throughput levels with fixed C value of 0.300 32
6.3 Various throughput levels with fixed C value of 0.5 33
6.4 Effect of Γ value on the throughput and number of threads for the Global Parabola Approximation 34
6.5 Number of times branches of the Global Parabola are taken for various Γ values 34
6.6 Effect of Γ value on the throughput and number of threads for the Local Parabola Approximation 35
6.7 Number of times branches of the Local Parabola are taken for various Γ values 36

List of Figures

2.1 Worker-per-connection architecture 4
2.2 Worker-per-request architecture 4
3.1 Multi-Input Single-Output Controller Architecture 9
3.2 Control Interval Timeline 10
3.3 Various shapes of throughput curves 11
3.4 A concave up and a concave down parabola 13
4.1 TPC-C Client application 18
5.1 Throughput curve of TPCC1-BIGMEM 23
5.2 Client side response time of TPCC1-BIGMEM 23
5.3 CPU and Disk characteristics of TPCC1-BIGMEM 24
5.4 Throughput curve of TPCC1-SMALLMEM measured 25
5.5 CPU and Disk characteristics of TPCC1-SMALLMEM 25
5.6 Throughput curve of TPCC2-BIGMEM 26
5.7 Client side response Time vs. Number of Threads for TPCC2-BIGMEM 26
5.8 CPU and Disk characteristics of TPCC2-BIGMEM 27
5.9 Throughput of the DS2 workload 28
5.10 Client side response Time vs. Number of threads for DS2 28
5.11 CPU and Disk characteristics of DS2 29
6.1 Throughput and Number of threads vs. C 32
7.1 Performance of the Hill-Climbing algorithm with the TPCC1-BIGMEM workload 38
7.2 Performance of the Global Parabola Approximation algorithm with the TPCC1-BIGMEM workload 38

7.3 Performance of the Hill Climbing algorithm with the TPCC1-SMALLMEM workload . . . 39
7.4 Performance of the Global Parabola Approximation algorithm with the TPCC1-SMALLMEM workload . . . 39
7.5 Performance of the Hill Climbing algorithm with the TPCC2-BIGMEM workload . . . 40
7.6 Performance of the Global Parabola Approximation algorithm with the TPCC2-BIGMEM workload . . . 41
7.7 Performance of the Hill Climbing algorithm with the DS2 workload 42
7.8 Performance of the Global Parabola Approximation algorithm with the DS2 workload . . . 42
7.9 Performance of the Hill Climbing algorithm while switching workloads from TPCC1-BIGMEM to TPCC1-SMALMEM . . . 44
7.10 Performance of the Global Parabola Approximation algorithm while switching workloads from TPCC1-BIGMEM to TPCC1-SMALMEM 44
7.11 Performance of the Hill Climbing algorithm while switching workloads from TPCC1-SMALLMEM to TPCC1-BIGMEM . . . 46
7.12 Performance of the Parabola Approximation algorithm while switching workloads from TPCC1-SMALLMEM to TPCC1-BIGMEM . . 46
7.13 Performance of the Hill Climbing algorithm while switching workloads from TPCC1-BIGMEM to TPCC2-BIGMEM . . . 48
7.14 Performance of the Global Parabola Approximation algorithm while switching workloads from TPCC1-BIGMEM to TPCC2-BIGMEM . 48
7.15 Performance of the Hill Climbing algorithm while switching workloads from TPCC2-BIGMEM to TPCC1-BIGMEM . . . 49
7.16 Performance of the Global Parabola Approximation algorithm while switching workloads from TPCC2-BIGMEM to TPCC1-BIGMEM . 50

Chapter 1

Introduction and Motivation

1.1 Thesis Statement

In this thesis we seek to explore the effect of changing the database server worker thread pool size on the performance of the server. We will also attempt to develop an online controller that can adjust the number of worker threads based on current throughput level of the server to achieve and maintain better server performance under varying workload conditions.

1.2 Motivation

More and more database servers are becoming an integral part of many software systems. With the revolution of the Internet, the need for database systems that can perform and sustain load has become a necessity rather than a luxury. Because database systems can be used in different environments, they are expected to perform well under varying workloads. For this reasons, database vendors have provided system administrators with different parameters or knobs that can be used to tweak the performance of the server. These parameters or knobs help system administrators utilize the potential capacity of the underlying hardware.

In this thesis we studied the effect of changing the number of worker threads (i.e., the multiprogramming level of the database server) on the throughput level of the server. By studying this effect, we have a better understand how to automatically adjust this parameter to match the changing workload characteristics or operating environment. We developed a controller that can automatically react to changing workload conditions.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 provides background information on the multiprogramming level parameter and its effects on the database server. We will also describe some of the previous research on this topic. Chapter 3 describes our controller design and the algorithms used by the controller to control the number of worker threads. Chapter 4 describes the workloads that we used and our test methodology and Chapter 5 describes the characteristics of these workloads. Chapter 6 discusses the sensitivity experiments that we performed to set the different algorithm parameters. Chapter 7 evaluates our controller algorithms using the test framework that we developed in Chapter 4. Finally, Chapter 8 summarizes our conclusions.

1.4 Major Thesis Contributions

This thesis studies the effect of changing the number of database server worker threads on the throughput level of the database server. We will use the industry standard TPC-C workload as our main benchmark tool to demonstrate how this parameter can impact the database server throughput levels.

We examine the effect of changing this parameter, and propose a controller that can dynamically alter this parameter. We are hoping to achieve the following objectives:

1. Allow a database server to better utilize the software and hardware resources of the underlying system,
2. Allow a database server to adapt to changing workload conditions, and
3. Allow a database server to achieve 1 and 2 without external intervention using autonomic control of the server multiprogramming level.

Chapter 2

Background and Related Work

2.1 Background

Database servers are used in many software systems, including accounting systems, inventory control systems, content management systems, human resources databases, online stores, and many others. The widespread use of database systems in many business areas has created varying performance expectations. In order to stay highly competitive in the database server market, database server vendors have provided customers with many tweaking parameters or knobs that can be used to adjust many of the server's internal algorithms to operate at a rate that achieves maximum throughput. Having one or two parameters to tweak might be an easy task, but current commercial database servers have over hundred parameters that can alter the server's behaviour [4]. A very skilled Database Administrator (DBA) would not only need to have internal knowledge of the working of the database server, but would also need to be very familiar with the access patterns and characteristics of the application workload. These two requirements make finding skilful DBAs a difficult and expensive task.

In recent years, researchers have started looking at ways to help reduce the total cost of ownership of maintaining and operating a database system. A new area of research has emerged that aims to create what are called *self-tuning database systems* [10, 28, 27]. Database servers with self-tuning algorithms respond to changing workload characteristics or operating system conditions with minimal or virtually no DBA intervention. Two areas in which self-tuning algorithms have been successfully used are automatic cache management [16, 24] and self-tuning histograms [7].

When it comes to servicing database server requests, there are two main architectures that can be identified:

1. **Worker-per-connection architecture**: In this architecture there is a one-to-one mapping between database server connections and database server workers. A worker is created once a connection is established and is responsible for servicing all requests from that connection. When that connection

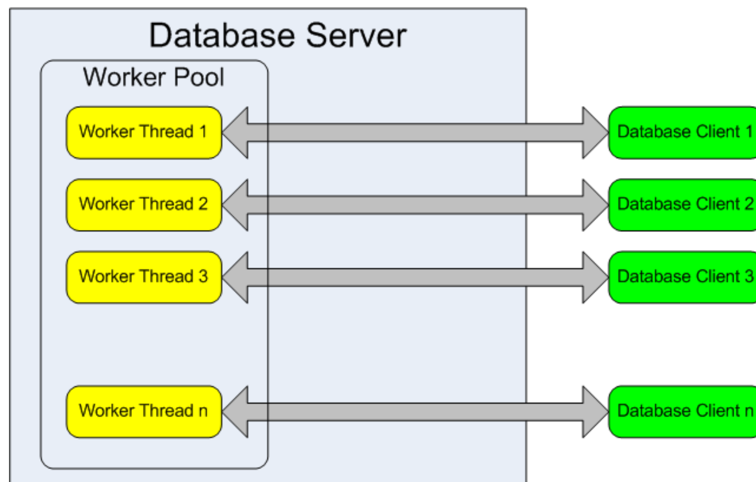


Figure 2.1: Worker-per-connection architecture

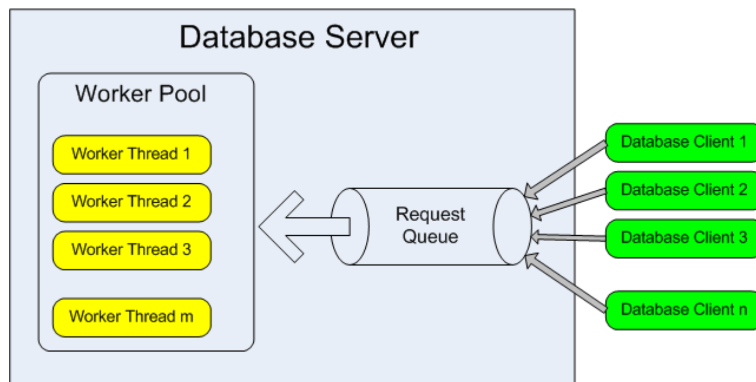


Figure 2.2: Worker-per-request architecture

closes, its associated worker is either terminated or added to an idle workers queue. Figure 2.1 illustrates this architecture.

2. **Worker-per-request architecture**: In this architecture there is one queue that is used to queue all database server requests from all connections. A worker dequeues a request from the request queue, processes the request and then goes back to process the next available request in the queue or block on an idle queue. In this configuration, there are no guarantees that a single connection will be serviced by the same worker. Figure 2.2 illustrates this

architecture.

In our work we will be looking at the latter architecture as it has proved to be more effective and to have less overhead (if configured properly) compared to the first architecture [22]. In addition, many of the widely used commercial database servers (e.g., MS SQLServer [3], Oracle, and SQLAnywhere [15]) employ this architecture. The only difficult issue with this architecture is how to set the size of the worker pool to achieve good throughput levels [13]. This parameter effectively controls the multiprogramming level of the server. DBAs have two choices when setting the value of this parameter:

1. Large number of worker threads: Using a large value for the number of workers can allow the server to process a large number of requests simultaneously. However, the drawback is that there is a substantial risk that a large number of workers can put the server into a thrashing state due to hardware or software resource contention or to excessive context switching between threads [21, 11]. Another issue with a large number of workers is that each thread has its own stack. Having many threads can consume a substantial part of a process's address space. This can negatively affect the size of the database server buffer pool that is available to cache database pages. This issue is more critical for 32-bit database servers.

2. Small number of worker threads: The immediate benefit of this approach is that the server has more memory to be used for caching. However, the drawback is that the multiprogramming level of the server is reduced and the hardware resources maybe under-utilized.

In order to better decide on the best value to use for the number of worker threads, DBAs would run load tests as part of their capacity planning process to find a value that can achieve the minimum required response time needed by the database application. Bowei et al. describe a smart hill-climbing approach to configuring application servers [29]. A similar algorithm can be applied to database servers. Although the experimental approach can be very effective in configuring the database server multiprogramming level parameter, it suffers from some drawbacks. One of the drawbacks is that this parameter setting is only good under the tested conditions and might not handle different or changing workload characteristics. In addition, this discovered value is very specific to the hardware on which the experiments were performed. If a hardware upgrade is performed, the selected value might be too conservative for the new hardware.

In this thesis we are proposing an online controller that can be used to automatically adjust the multiprogramming level of the database server to achieve maximum throughput. We now describe related work in this area.

2.2 Related Work

One of the earliest papers that exposed the problem of adaptive load control in database system was by Heiss and Wagner [14]. In that paper, the authors outlined different sources of contention and suggested a possible feedback control mechanism that limits the concurrency level of the server. Two different algorithms were proposed: an Incremental Steps (IS) approach and a Parabola Approximation (PA) approach. In the IS approach, the controller monitors the throughput level of the server and the current concurrency level. The controller would then increase the concurrency level and monitor its affect on the throughput of the server. If the throughput increases, the controller keeps on increasing the concurrency level; conversely, if the throughput decreases, the controller would reduce the concurrency level up to the point that throughput level would start to suffer. In the PA approach, the performance of the server is approximated using a parabolic function. We have based our algorithms on those two algorithms, but we have extended them. Heiss and Wagner used simulation to validate their findings. In our approach we used a real commercial database server and the industry standard TPC-C workload [25] for validation.

Mönkeberg and Weikum attempted to solve the problem of data contention thrashing in database systems [20]. They proposed an adaptive load control method that is based on one performance metric: conflict ratio. The conflict ratio is the ratio of the number of locks that are held in the system divided by the number of locks held by active (non-blocked) transactions in the system. If the conflict ratio exceeds 1.3 (derived through empirical studies) then data contention (DC) thrashing is detected. Load control is accomplished by two methods: admission control and transaction cancellation. The adaptive load control is performed before the beginning of a transaction (BOT) or after the end of a transaction (EOT). In addition, Mönkeberg and Weikum also suggest using information about transactions to decide what admission policies to use. In Mönkeberg and Weikum's work, the multiprogramming level is adjusted by increasing the number of transactions that enter the system. Their approach assumes that database servers are the only operating process on the server machine. Our approach is more general in that it takes into account contention on hardware resources or changing operating conditions of the server.

Brown et al. attempted to automate performance tuning by adjusting different knobs: the multiprogramming level and the amount of memory used by transactions [9]. The proposed tuning algorithm relies on classifying transactions by expected response time goals. Our approach is different in that we will have no prior knowledge of the cost of transactions or any classification of transactions. In addition, Brown et al. used a simulated environment to evaluate their technique while we used a real commercial database system.

Recent research by Bianca Schroeder and colleagues has considered adjusting the multiprogramming level by adjusting the number of threads in the server worker

pool [22]. This work includes an experimental study on how the multiprogramming level affects throughput and mean response time. She proposed an external scheduling controller that adjusts the multiprogramming level based on periods of observations and reactions. The controller used a simple feedback control loop. Our work builds on this work. We go further into looking at different algorithms for the control loop and we also study how those algorithms react to changes in workload conditions. In addition, our controller design is implemented within the database server.

In similar research, Xue et al. looked at optimizing online response time of the Apache [12] web server [19]. They focused on online tuning the *MaxClients* parameter of the Apache web server which controls the multiprogramming level of the server. They looked at different controller algorithms including hill climbing and a heuristic based one. In their work they found that such online tuning technique help reduce response time by a factor of 10 or more. One of their future work items was to implement these techniques in more complicated server applications such as a database server or an application server. Our work implements a similar tuning algorithm in a database server. The complications in a database server come from the fact that database server workloads usually result in contention for other software resources such as data structures and internal database server algorithms.

In summary, most previous research on adjusting the multiprogramming level was evaluated in simulated environments. Our work attempts to look adjusting the multiprogramming level for improving the performance of the database server. We are using a commercial database system with one of the industry standard online transaction processing (OLTP) workloads.

The next chapter will describe our controller architecture and the algorithms that the controller uses to automatically tune the database server multiprogramming level.

Chapter 3

Design and Implementation

In this chapter we will describe the architecture of our controller and the three different algorithms that we have developed for controlling the multiprogramming level.

3.1 Controller Architecture

Any controller design involves a set of inputs and a set of outputs. We will be using a multi-input single-output (MISO) controller. Figure 3.1 illustrates the architecture of the controller. For inputs, our controller will monitor the throughput level of the server. This is measured by the number of server-side requests (not transactions) completed per minute. A single client transaction can involve one or more server requests. For example, a simple insert transaction could involve the following operations:

- Prepare the SQL statement
- Bind the input parameters and send the data
- Execute the statement
- Retrieve the result

Every one of those operations could translate into a server side request.

Another input parameter that the controller uses is the total number of requests that are outstanding at the server. We will call this number the *workload concurrency level* and it is the sum of the number of requests that are waiting in the request queue and the number of requests that are being serviced by the worker pool. For example, if the worker pool size is 10 and all of the 10 workers are busy servicing requests and there are 5 requests waiting in the request queue, then the workload concurrency level of the server is 15.

Another input to the controller is the control interval. If the interval is too small, then there will be large oscillations and the server will not be stable. On the other hand, if the interval is too big, it will take the controller a long time to adapt to changing workload conditions. In our implementation we will fix the interval to one minute and will not change it throughout our experiments. This length should give the server enough time to stabilize at the next multiprogramming level.

The controller's output is the multiprogramming level that the server will use during the next control interval.

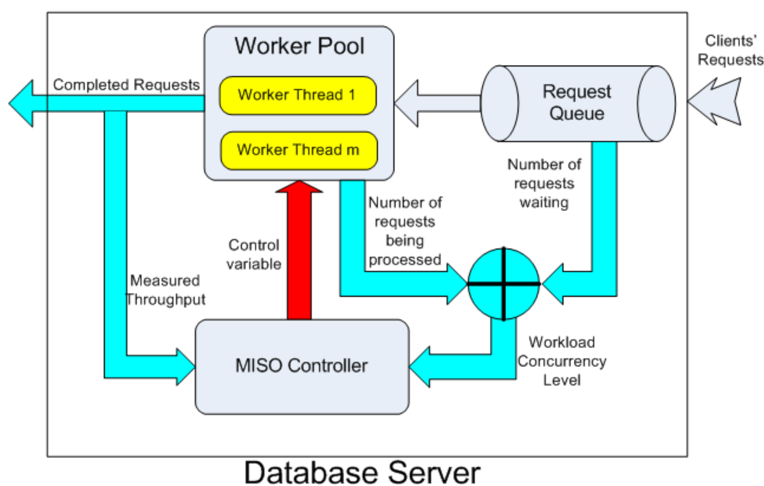


Figure 3.1: Multi-Input Single-Output Controller Architecture

3.2 Auto-tuning Algorithms

In this section we will describe the three algorithms that we will be studying. Table 3.1 introduces some standard notation that we will be using to describe the three algorithms.

To help understand our notation, Figure 3.2 shows a timeline and how the different values relate to each other according to time. The parameter $n^*(t_i)$ is the current number of threads during the control interval $t_{i-1} < t \leq t_i$. The $n^*(t_i)$ value is fixed during a control interval and does not change. $P(t_i)$ is the total number of requests that completed during the control interval $t_{i-1} < t \leq t_i$. The workload concurrency level $n(t_i)$ is measured only once at t_i and is not averaged.

Another concept that we need to introduce is the throughput curve. A throughput curve is a function that describes the relationship between the multiprogramming level and the throughput of the server. The x-axis is the multiprogramming level and is measured by the number of threads. The y-axis is the throughput level.

Notation	Description
t_i	End of the current control interval and start of the next control interval t_{i+1}.
$n^*(t_i)$	The number of worker threads the server has available during the control interval $(t_{i-1}, t_i]$. This is the control variable.
$n(t_i)$	The workload concurrency level at time t_i.
$P(t_i)$	The actual measured throughput level of the server at time t_i. The throughput is the number of requests that completed during the previous measurement interval.

Table 3.1: Standard notation used by the different algorithms

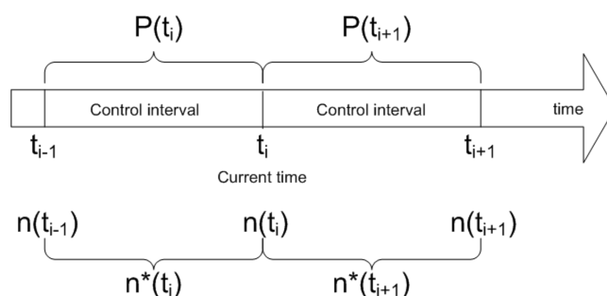


Figure 3.2: Control Interval Timeline

Different workloads have different throughput curves. Figure 3.3 shows some examples of throughput curves. Figure 3.3a shows a hill-shaped curve. Web servers usually follow this throughput curve shape [8]. Figure 3.3b shows a throughput curve that does not depend on multiprogramming level until it is high, at which point it starts to decline. Figure 3.3c shows a throughput curves that starts as a hill-shaped curve but then flattens out (possibly because of some resource becoming fully utilized.) The shape of a throughput curve depend on many factors including the workload, the dependencies between requests, and the contention for hardware or software resources.

Our primary goal is to maximize server throughput. Our secondary goal is to minimize the multiprogramming level. The secondary goal is included because in some workloads it is possible that the throughput is not affected by the multiprogramming level. Having a large number of threads will not only increase contention within the server, but will also make the server susceptible to load spikes. Load spikes can happen if there is a burst of transactions at the server that use up all the available workers. Those load spikes can put the server into a thrashing state that

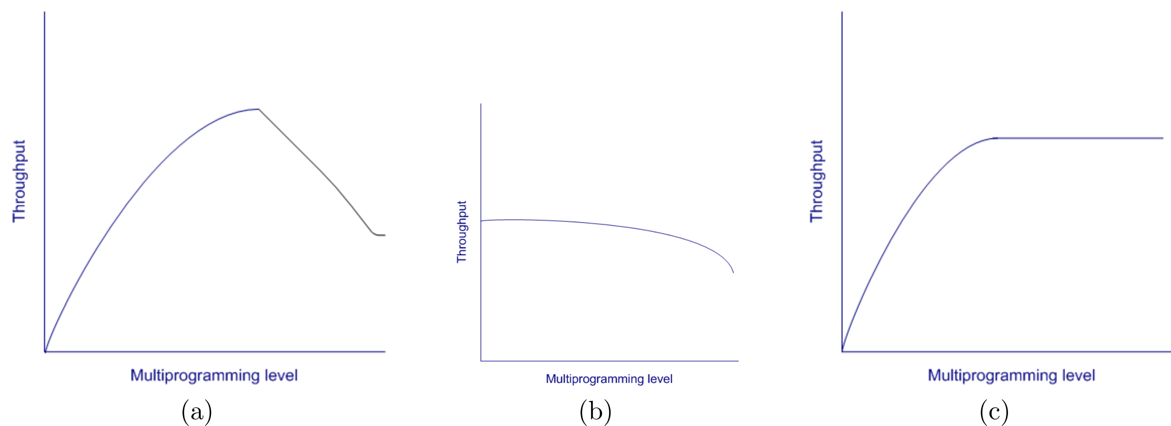


Figure 3.3: Various shapes of throughput curves

the server can not recover from. In addition, having extra threads wastes resources such as memory and CPU. Such resources can be better utilized for other purposes especially in a database server.

In all of the algorithms described in the following section, if the concurrency level $n(t_i)$ is less than the current number of threads $n^*(t_i)$, then the server reduces the number of worker threads to match the measured concurrency level. In other words,

$$n^*(t_{i+1}) = n(t_i) \text{ if } n^*(t_i) > n(t_i)$$

Given this rule, the algorithm discussions below will assume that the concurrency level $n(t_i)$ is greater than or equal the number of server threads $n^*(t_i)$. This is because threads above and beyond the number needed to accommodate the workload concurrency level will be idle and will not provide any benefit.

3.2.1 Hill Climbing

The goal of this algorithm is to keep following the throughput curve as long as the percentage of change in throughput is at least C times the percentage change in the number of threads. On the other hand, if decreasing the number of threads increases performance, then we keep doing so until the performance starts to degrade. This difference between the upward direction and downward direction is due to the fact that this algorithm has bias towards the downward direction. This bias is included to accomplish our second goal of keeping the number of threads as low as possible as well as preventing the controller from going astray in cases where the throughput curve flattens out such as in Figure 3.3c. If the algorithm does not see any gains in throughput in either the forward or backward direction, it will backtrack to the previous multiprogramming setting.

The second parameter that this algorithm depends on is the step size Δ. The step size controls how fast the algorithm converges.

The algorithm can be described by the following formula:

$$n^*(t_{i+1}) := \begin{cases} n^*(t_i) + \Delta & \text{if } n^*(t_i) \geq n^*(t_{i-1}) \text{ and } \frac{P(t_i)-P(t_{i-1})}{P(t_{i-1})} \geq C\frac{n^*(t_i)-n^*(t_{i-1})}{n^*(t_{i-1})} \\ n^*(t_i) - \Delta & \text{if } n^*(t_i) < n^*(t_{i-1}) \text{ and } P(t_i) > P(t_{i-1}) \\ n^*(t_{i-1}) & otherwise \end{cases}$$

where the parameter C takes values from 0 to 1.

This algorithms basically tries to build on the fact that each additional thread added to the worker pool is going to have a smaller benefit (i.e. requests completed) compared to the previous thread. This happens because every additional thread added is going to consume additional memory (for stack and other thread local storage data) and additional CPU overhead (context switching time and CPU cache pressure).

3.2.2 Global Parabola Approximation

This algorithm attempts to model the throughput curve as a parabolic function. We choose a parabola as a model for several reasons. First, a parabola can have a concave downward shape similar to the throughput function found in many servers. Second, the parabola is easy to use and easy to compute. Third, although the real throughput function might have a tail that flattens out, the first part of the throughput function can be modeled well after a parabola and we do not need to model the tail part where the server can have degraded performance levels.

In order to approximate a parabola, we need three data points. Two of the data points will be based on two observations of throughput at some multiprogramming level. For the third point we will use the origin, hence, the global aspect of the parabola approximation. The equation of the throughput curve as a parabolic function is:

$$P(t_i) = a(n^*(t_i))^2 + b(n^*(t_i)) + c$$

The two points that we will be using are $(n^*(t_i),P(t_i))$ and $(n^*(t_{i-1}),P(t_{i-1}))$. Using the following equations, we can solve for a, b, and c:

$$a = \frac{P(t_i)n^*(t_{i-1}) - P(t_{i-1})n^*(t_i)}{n^*(t_i)n^*(t_{i-1})(n^*(t_i) - n^*(t_{i-1}))}$$

$$b = \frac{P(t_{i-1}) - a(n^*(t_{i-1}))^2}{n^*(t_{i-1})}$$

$$c = 0$$

Once we have the values of a, b, and c, the algorithm tries to move to the n^* that maximizes throughput. This would be the point where the slope is 0.

$$n^*(t_{i+1}) = \frac{-b}{2a}$$

Because of possible noise in the measurements, it is possible that a could be positive. A positive a coefficient models a parabola that is concave up and, hence, does not model a realistic throughput curve. Figure 3.4 illustrates how noise at multiprogramming level 180 generated a concave up parabola. In this example, the controller would suggest moving to point 45. This is clearly a bad step from 180.

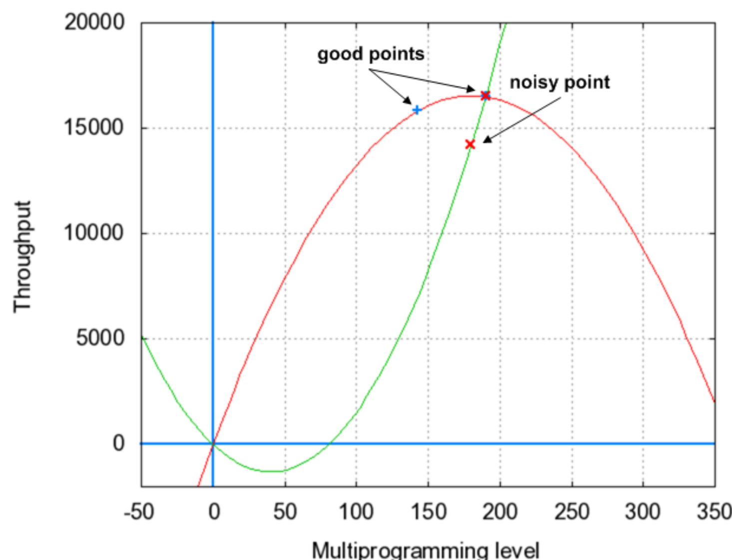


Figure 3.4: A concave up and a concave down parabola

If the computed parabolic function is upward opening then we ignore the collected data and just move forward (or backward) by a random number of steps between 0 and Δ. We keep on alternating between moving forward or backward every time we hit this condition until we start getting more appropriate parabolic functions with positive a coefficient. We alternate forward and backward because the data we collected does not give us a proper direction so we go in either direction to introduce randomness.

On the other hand, noise in the measurement data can also cause the controller to suggest a new value that is too large. In this case, we use a parameter Γ to cap the step size. The Γ value is basically a percentage of the current number of threads. For example, a Γ value of 0.3 would cap the steps size to 30% of the current number of threads. Chapter 6 will go into detail on how we choose the values of this parameter and its affect on the performance of the algorithm.

The algorithm actions can be summarized by the following function:

$$n^*(t_{i+1}) := \begin{cases} min(\frac{-b}{2a}, (\Gamma + 1)n^*(t_i)) & \text{if } a < 0 \text{ and } \frac{-b}{2a} \neq n^*(t_i) \\ n^*(t_i) \pm rand(0, \Delta) & otherwise \end{cases}$$

3.2.3 Local Parabola Approximation

This algorithm is similar to the previous algorithm, the only difference is that it uses the last three observations to solve for the parabolic function. It does not assume that the parabola passes through the origin as in the previous algorithm, hence, the local aspect of the algorithm. By taking the last three data points we are hoping to get a better local approximation of the throughput curve in order to estimate the apex of the curve. The local parabola will give us a local maximum, unlike the global parabola algorithm which gives us a global maximum.

The a, b, and c parameters of the parabola can be computed as follows:

$$a = \frac{(P(t_{i-2}) - P(t_i)) - \frac{(n^*(t_{i-2})-n^*(t_i))(P(t_{i-1})-P(t_i))}{(n^*(t_{i-1})-n^*(t_i))}}{(n^*(t_{i-2}) - n^*(t_i))(n^*(t_{i-2}) - n^*(t_{i-1}))}$$

$$b = \frac{(P(t_{i-1}) - P(t_i)) - a(n^*(t_{i-1})^2 - n^*(t_{i-1})^2)}{(n^*(t_{i-1}) - n^*(t_i))}$$

$$c = P(t_i) - an^*(t_i)^2 - bn^*(t_i)$$

Once we have the equation of the parabola, the algorithm reacts as follows:

$$n^*(t_{i+1}) := \begin{cases} min(\frac{-b}{2a}, (\Gamma+1)n^*(t_i)) & \text{if } a < 0 \text{ and } \frac{-b}{2a} \neq n^*(t_i) \\ n^*(t_i) \pm rand(0, \Delta) & otherwise \end{cases}$$

Because of noise in the data, it is possible that two or more of the observations have identical n^* values but different P values. If this happens, then we alternate moving forward or backward by a random number of steps between 0 to Δ. This algorithm also uses a Γ parameter to limit the change in number of threads to a percentage Γ of the current multiprogramming level.

Summary

In this chapter we have introduced the three auto-tuning algorithms that we have developed. Table 3.2 summarizes the three algorithms and their parameters. Chapter 6 will go into details on what values we decided to use for those parameters.

Algorithm	Parameters	Parameter Description
Hill Climbing	Δ	step size
	C	performance gain ratio threshold
Global Parabola Approximation	Δ	maximum wiggle step size
	Γ	maximum percentage change in n^*
Local Parabola Approximation	Δ	maximum wiggle step size
	Γ	maximum percentage change in n^*

Table 3.2: Summary of parameters for tuning algorithms

Chapter 4

Workloads and Test Methodology

In order to further motivate our problem and see how the auto-tuning algorithms react with different workloads, we will describe in this chapter the benchmarks we used and our test methodology. Section 4.1 describes the database server software. Section 4.2 describes hardware configurations. Section 4.3 describes the workloads that we used, and Section 4.4 discusses our test methodology and the experiments that we performed.

4.1 Database Server Software

For our experiments we used SQL Anywhere [17] as our database server. SQL Anywhere is a full-fledged commercial database server that can serve many concurrent users simultaneously and has row level locking. SQL Anywhere is a multi-platform database server. It can run on a wide range of operating systems and on many of the popular handheld platforms. We have chosen SQL Anywhere because we have access to the source code and because its internal architecture is well known to us. We used a database page size of 4KB in all of our databases.

4.2 Hardware Configuration

The database server machine consists of a 32-bit dual 1.80GHz Intel Xeon (Prestonia) processor machine running Windows 2003 Enterprise Server. The machine has 8KB of L1 cache, 512KB of L2 cache, and 2GB of RAM. The disk system consists of a 48 spindle disk array configured as RAID 0 with a stripe size of 64KB.

For the DS2 workload explained in Section 4.3.2, we used separate client machine to run the driver application. The client machine is a 64-bit 4-way 2.40GHz AMD Opteron processor machine running Windows 2003 Enterprise server. The machine has 64KB of L1 cache, 1024KB of L2 cache, and 16GB of RAM.

4.3 Workloads

For our experiments, we derived four workloads from two different benchmarks. The first three workloads are based on the TPC-C benchmark and are described in Section 4.3.1. The fourth workload is based on the DS2 benchmark and is described in Section 4.3.2

4.3.1 TPC-C

We used the industry standard TPC-C workload [25]. The TPC-C workload simulates the activities of a complex online transaction processing (OLTP) application environment. The TPC-C benchmark simulates a number of warehouses. Each warehouse serves 10 districts. Each of the districts has a terminal in which transactions are entered. There are five types of transactions in the TPC-C workload: New Order, Payment, Order-status, Delivery, and Stock-level. Table 4.1 shows the transaction mix of the TPC-C workload.

Transaction type	Percentage in mix
New Order	45%
Payment	43%
Order-status	4%
Delivery	4%
Stock-level	4%

Table 4.1: Standard Transaction mix of the TPC-C workload

The performance metric of the TPC-C standard is the number of New Order transactions that complete within the 90th percentile of target transaction response times. For our purposes, we will be basing our metric on the server side measured throughput. This will include all TPC-C transaction types and not the New Order transaction only.

We have chosen the TPC-C workload because it is an OLTP workload. Other benchmarks such as TPC-H [26] would not be a good benchmark for us as it measures the speed of the server computation algorithm. The TPC-C workload has the following attractive characteristics:

- Simultaneous execution of multiple transaction types that span a breadth of complexity,
- Multiple on-line terminal sessions: this feature allow us to have many connections open simultaneously to the server,
- Moderate system and application execution time,

- Significant disk I/O,
- Non-uniform distribution of data access through primary and secondary keys
- Databases consisting of many tables with a wide variety of sizes, attributes, and relations
- Contention on data access and update

The client application that we used emulates the TPC-C terminals. Using command line options, the client application can be configured to generate different load characteristics by specifying the number of warehouses. A pool of threads in the client is created to emulate the warehouses' terminals. Figure 4.1 illustrates the TPC-C client application testing environment. The client application runs on the same machine as the database server. Communication with the database server is done through shared memory. We have not observed a need to have the client run on a separate machine since the processing done by the client is very minimal and only takes a few percentages of the server's CPU utilization. The logic for all transactions is implemented in stored procedures on the server side. All that the client application does is call the stored procedures with different parameters.

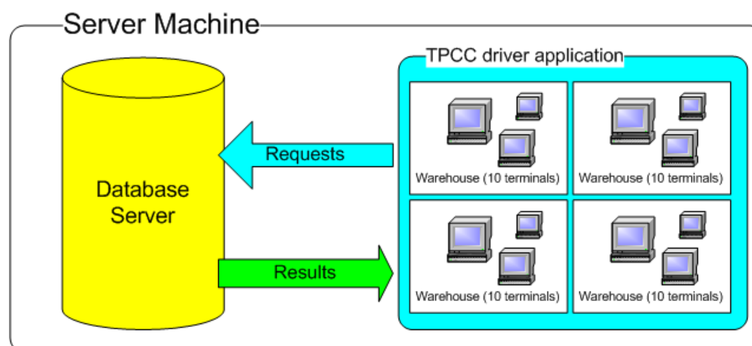


Figure 4.1: TPC-C Client application

For our purposes, we used two transaction mixes of the TPC-C workload. We will call them TPCC1 and TPCC2. TPCC1 uses the same transaction mix as the standard TPC-C workload but differs in that there is no think time or keying time. In other words, transaction are issued as fast as the server can handle them. Setting the think time and keying time to zero allow us to generate a reasonable load on the database server using a smaller database.

Transaction type	Percentage in mix
New Order	25%
Payment	63%
Order-status	4%
Delivery	4%
Stock-level	4%

Table 4.2: TPCC2 workload Transaction mix

TPCC2 uses a slightly modified transaction mix than TPCC1. Table 4.2 shows the transaction mix of the TPCC2 workload.

Like TPCC1, TPCC2 also sends transactions to the server as fast as the server can process them. The TPCC2 workload has fewer I/O requirements (compared to TPCC1) as it processes fewer New Order transactions.

In addition to varying the transaction mix, we varied the database server buffer pool size. We used two buffer pool configurations: SMALLMEM and BIGMEM. In the SMALLMEM configuration, we configured the server buffer pool size to 500MB. In the BIGMEM, it was configured to 1GB.

The TPC-C database that we used in our workloads was a 150 warehouse database. Its size is 13.8GB.

With different transaction mixes and different buffer pool configurations, we now have three different workload configurations based on the TPC-C benchmark:

- TPCC1-BIGMEM
- TPCC1-SMALLMEM
- TPCC2-BIGMEM

4.3.2 DS2

The DS2 [1] benchmark simulates an online e-commerce DVD store. It supports multiple back-end databases including one for Microsoft SQL Server [2], Oracle [6], and MySQL [5]. For our benchmark purposes, we implemented the SQL Anywhere back-end driver. The benchmark can be configured in one of three different scales: small, medium, and large. The database size of the small scale is 10MB, the size of the medium scale is 1GB, and the size of the large scale is 100GB. In our experiments we used the medium scale configuration. The DS2 benchmark is also an OLTP workload but it has different transaction costs and requirements than the TPC-C.

Every thread performs a predefined sequence of operations:

1. Login with an existing web site user id or create a new one. By default percentage of new web site users is 20%,
2. Perform some browse operations on the database to lookup DVD authors or titles,
3. Place an online purchase order of 1 or more items, and
4. Perform some think time and then go to step 1. The default think time is set to 0 seconds.

The benchmark driver allows us to control many parameters, of which the following are of interest to us:

- Number of threads (i.e. database connections),
- Duration of the warm-up period,
- Duration of the test period,
- Duration of the think time.

4.3.3 Summary

Our four workload configurations are summarized in table 4.3.

Configuration	Transaction Mix	Buffer Pool Size	DB Size
TPCC1-BIGMEM	TPCC1	1.0GB	13.8GB
TPCC1-SMALLMEM	TPCC1	500MB	13.8GB
TPCC2-BIGMEM	TPCC2	1.0GB	13.8GB
DS2	DS2	250MB	1.0GB

Table 4.3: Workloads configurations

4.4 Methodology

All of our experiments were done in a closed system [23]. In this setup, requests arrive at the server at the same rate that the server can complete them. Little's law [18] states that $N = \lambda T$, where N is the number of concurrent requests, λ is the arrival rate, and T is the response time. Since the arrival rate equals the departure rate (throughput), and since N is constant, throughput will be inversely related to response time T.

In order to understand how each workload is affected by the server multiprogramming level, we perform a set of workload characterization experiments for each of the four workloads. In every characterization experiment, we configure and start the database server with a fixed number of threads. We then run the workload and record the overall throughput level as seen from the point of view of the server. This experiment is then repeated for a new fixed number of threads until we have covered the range from zero to the maximum number of client connections. We also collect various OS statistics such as CPU and disk utilization. The purpose of these experiments is to allow us to draw the throughput curve for each of these workloads and also reveal the range of multiprogramming values that achieve optimal server throughput.

Although we are using SQL Anywhere as our database server, the same procedure can be used with any other database server. The shape of the throughput curve that we generate depends on many factors including the database server software, the workload, and the hardware configuration. Different database servers might produce different throughput curves. The characterization experiments will help us understand how the throughput curve look like for the given software and hardware configuration that we have.

Before being able to judge how well the algorithms react to different workloads, we need to select a set of parameters to be used throughout our experiments. To do so, we have to have a good understanding of how each algorithm's parameter affects its behaviour and we have to be able to choose good values for these parameters. We perform a set of parameter sensitivity experiments in which we study how each of the parameters affect the way the algorithms behave. These sensitivity experiments will help us in choosing values for these parameters. Chapter 6 documents the results of these experiments.

After completing the sensitivity experiments, we try our workloads with each of the auto-tuning algorithms that we have discussed in Chapter 3. These experiments will allow us to study how well each of the algorithms was able to reach the range of good multiprogramming values that we have discovered in the workload characterization experiments.

Finally, we study how well the algorithms react to changing workload conditions. In the real-world, a workload condition can change if either the resources on the server machine become limited or if the transaction mix of the workload changes. We will be testing for both of those change conditions by starting an experiment with one of the workload and then half way throughout the run we switch to a different workload. These experiments would simulate close to real-world scenarios of how workloads can change either in resource requirements or in transaction mix. We will study how well each of the algorithms react to these external changes.

Chapter 5

Workload Characterization

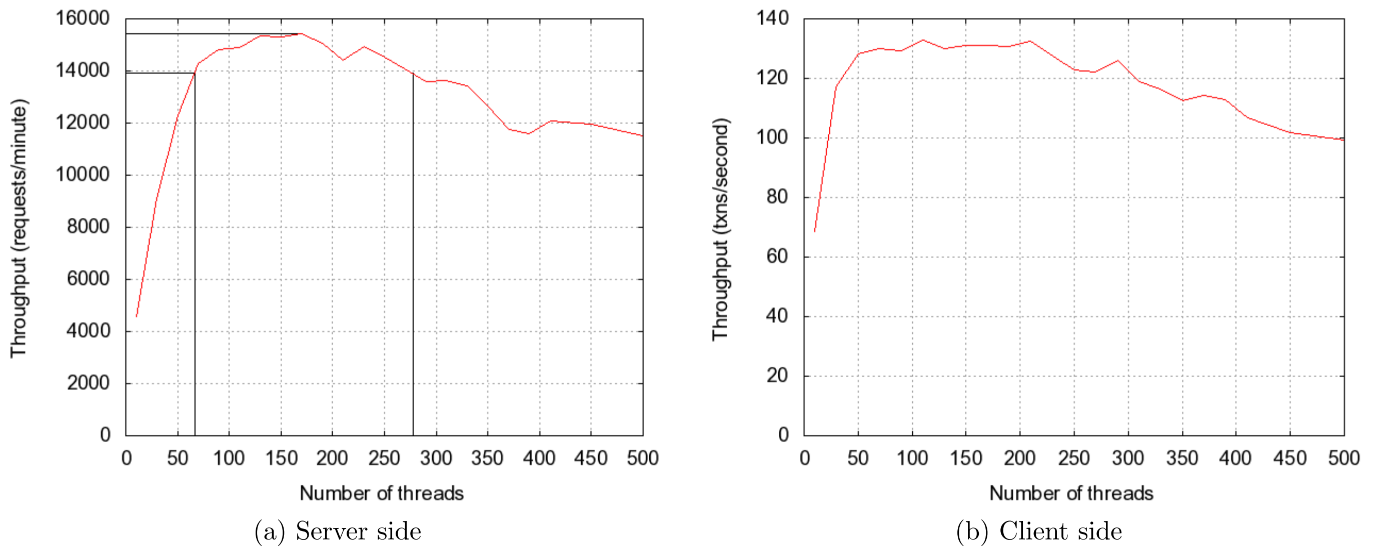
The purpose of this chapter is to illustrate how each of the workloads behave when the number of database server threads is altered. For each of the workloads that that we introduced in Section 4.3, we ran the benchmark with a fixed number of threads then the same experiment was repeated for a different number of threads. We repeated these experiments from 20 to the maximum number of clients of 500. Using the data collected from these experiments we deduce the throughput curve shape for each workload. The throughput curve will help us identify a range of multiprogramming values that can achieve high throughput. In all of the workload characterization experiments, the response time was measured on the client side only.

5.1 TPCC1-BIGMEM

Figure 5.1 shows the throughput curve for the TPCC1-BIGMEM workload measured at the server side and at the client side. We can see from the two curves that the throughput curve of the client closely resembles that of the server even though the throughput on the server is measured in requests per minute while on the client it is measured in transactions per second. A client side transaction can translate into one or more server side requests. Since our tuning algorithms are implemented in the server, we will henceforth focus on server side throughput.

The TPCC1-BIGMEM throughput curve starts as hill-shaped curve but then it flattens out. The throughput climbs quickly from 10 threads to 100 threads. The throughput level with 410 threads matches the throughput level at 50 threads and not that at 10 threads. Between 110 and 230 threads, the throughput level is almost constant. Such a flat curve between 110 and 230, would make it difficult for an auto-tuning algorithm to easily detect that we have reached the peak of the hill. Instead, the algorithm may stray in the flat part of the curve.

Figure 5.2 shows the response time measured on the client side. If we compare this curve to client side throughput curve in Figure 5.1b, we can see that the



(a) Server side (b) Client side

Figure 5.1: Throughput curve of TPCC1-BIGMEM

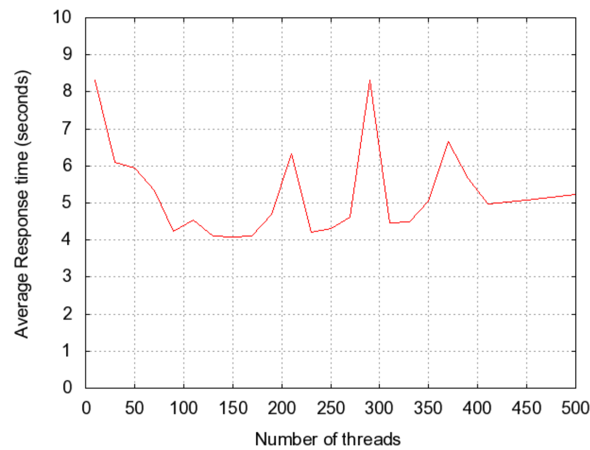


Figure 5.2: Client side response time of TPCC1-BIGMEM

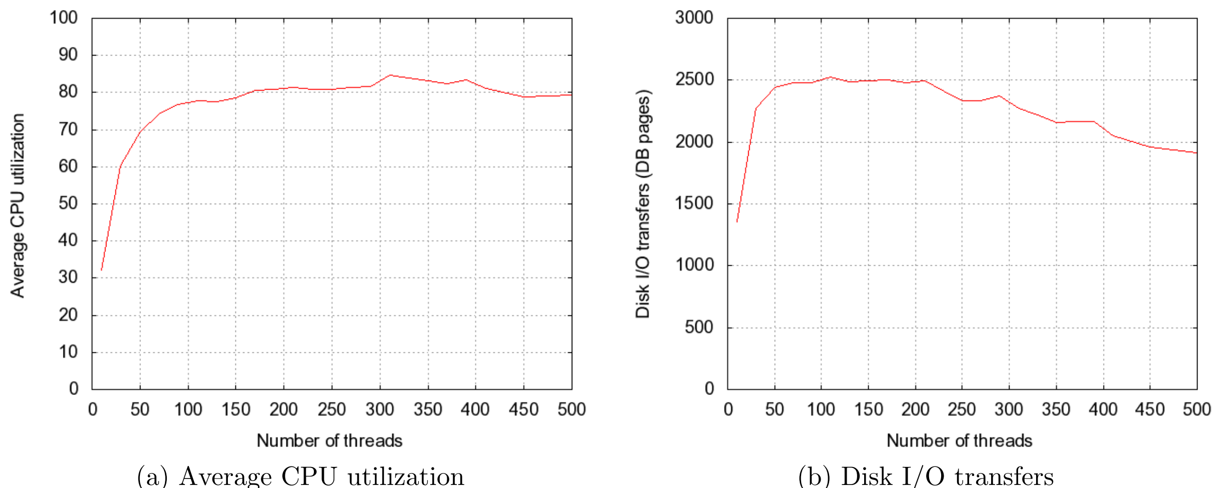
throughput is inversely related to response time. Our experiments are run in a closed system, hence, the N value in Little's law ($N = \lambda T$) is fixed. Since N (number of clients) is fixed, the response time will be inversely related to throughput.

Figure 5.3a shows the CPU characteristics of the TPCC1-BIGMEM workload. The CPU utilization averages 80% once the number of threads exceeds 100 threads. The fact that beyond 100 threads the CPU utilization stays constant indicates that some resource has been fully utilized.

In terms of disk usage, we can see from Figure 5.3b that disk transfers increase until it levels off at around 50 threads. It then starts to drop when the number of threads exceed 200. This drop in disk transfers indicates that either there is high contention for disk access or that the server is not issuing enough I/O. Contention

for server data structures could cause the server to issue fewer I/Os.

Table 5.1 summarizes the characteristics of the TPCC1-BIGMEM workload. For TPCC1-BIGMEM, a good setting for the multiprogramming level is between 67 and 278 threads. This range achieves 90% of the maximum throughput observed with this workload. This range is shown by vertical and horizontal lines in Figure 5.1a.



(a) Average CPU utilization (b) Disk I/O transfers

Figure 5.3: CPU and Disk characteristics of TPCC1-BIGMEM

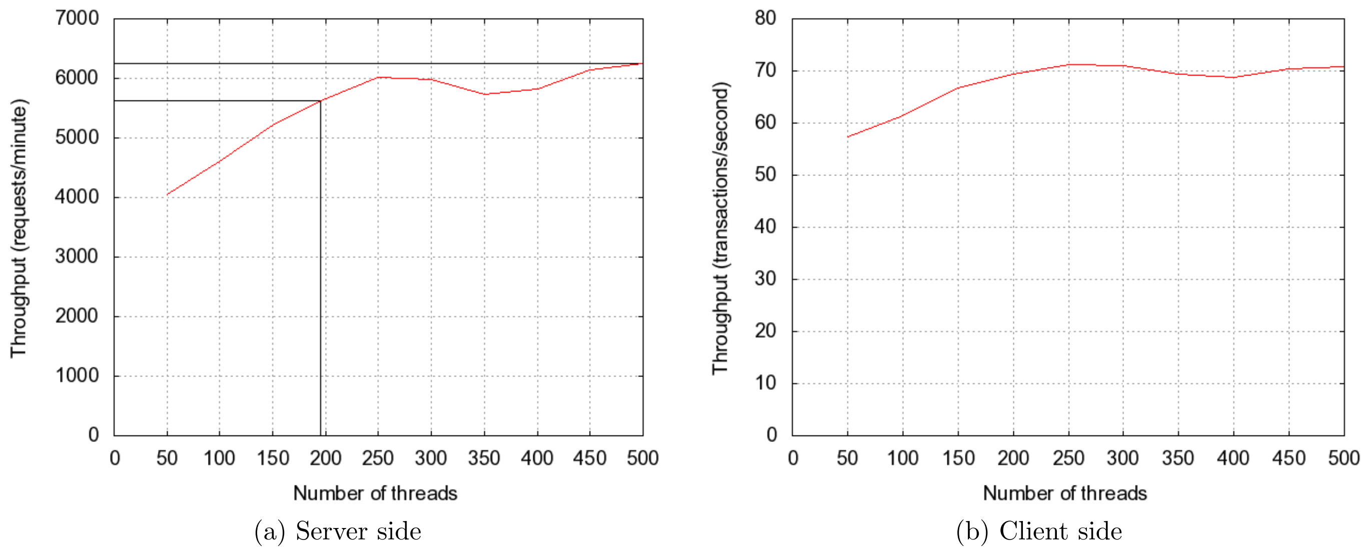
Characteristic	Value
Optimal number of threads	67 - 278
Optimal throughput	13898 - 15347 requests/minute

Table 5.1: Characteristics of the TPCC1-BIGMEM workload

5.2 TPCC1-SMALLMEM

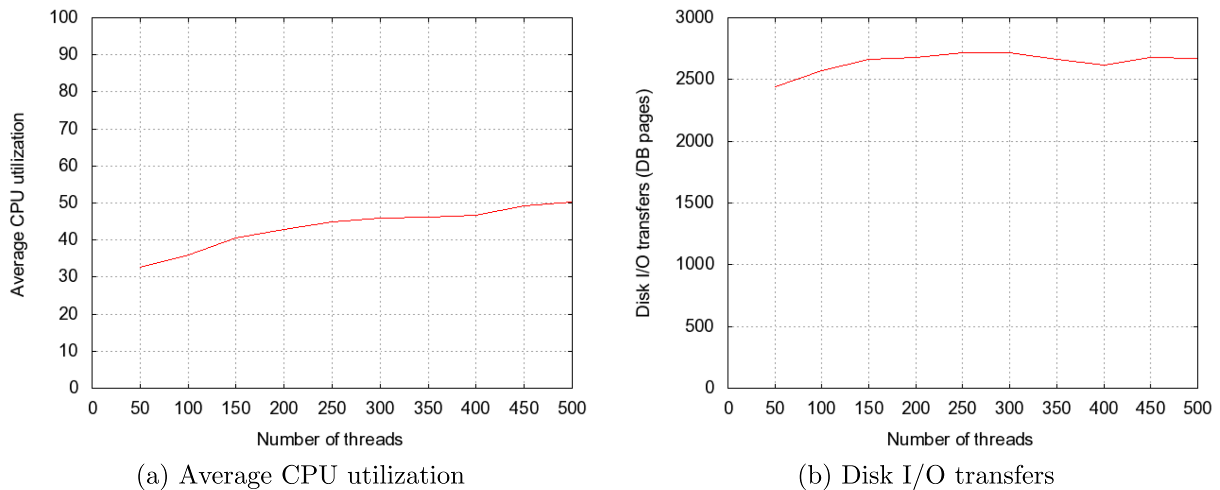
Figure 5.4 shows the throughput measured at the server side and at the client side. The throughput curve for TPCC1-SMALLMEM shows that the more threads that are added, the more the throughput will increase. This is due to the fact that the TPCC1-SMALLMEM workload has half the buffer pool size of TPCC1-BIGMEM. With TPCC1-SMALLMEM, there is more contention for disk resources. If we compare the disk transfers of TPCC1-BIGMEM in Figure 5.3b to the disk transfers of TPCC1-SMALL in Figure 5.5b, we can see that TPCC1-SMALLMEM has higher disk transfers.

The average CPU utilization increases steadily with throughput as shown in Figure 5.5a. The more threads are added, the more there will be overlap between



(a) Server side (b) Client side

Figure 5.4: Throughput curve of TPCC1-SMALLMEM measured



(a) Average CPU utilization (b) Disk I/O transfers

Figure 5.5: CPU and Disk characteristics of TPCC1-SMALLMEM

I/O and CPU and throughput increases. This indicates that more disk throughput can be utilized by increasing the multiprogramming level.

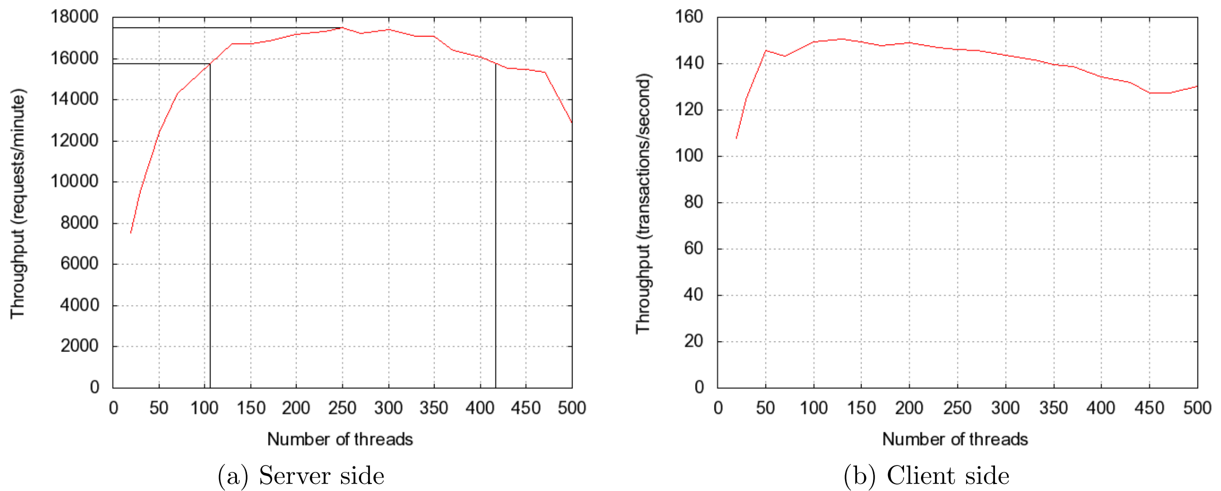
The throughput curve here does not resemble a hill-shaped curve; hence, a tuning algorithm that is looking at throughput only as a metric will likely tend to keep on increasing the number of worker threads as it will observe that throughput is improving.

Table 5.2 summarizes the characteristics of the TPCC1-SMALLMEM workload. In this workload we have observed that a multiprogramming level between 195 and 500 can achieve 90% of maximum throughput. Since this is a large range of values, a good tuning algorithm will attempt to stay on the lower side of this range.

Characteristic	Value
Optimal number of threads	195 - 500
Optimal throughput	5627.89 - 6253.22 requests/minute

Table 5.2: Characteristics of the TPCC1-SMALLMEM workload

5.3 TPCC2-BIGMEM



(a) Server side (b) Client side

Figure 5.6: Throughput curve of TPCC2-BIGMEM

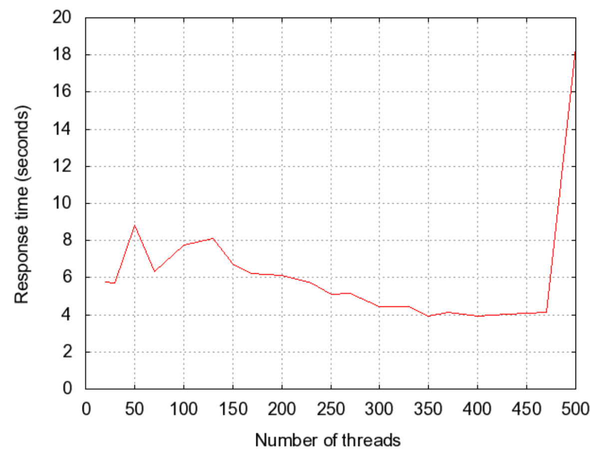
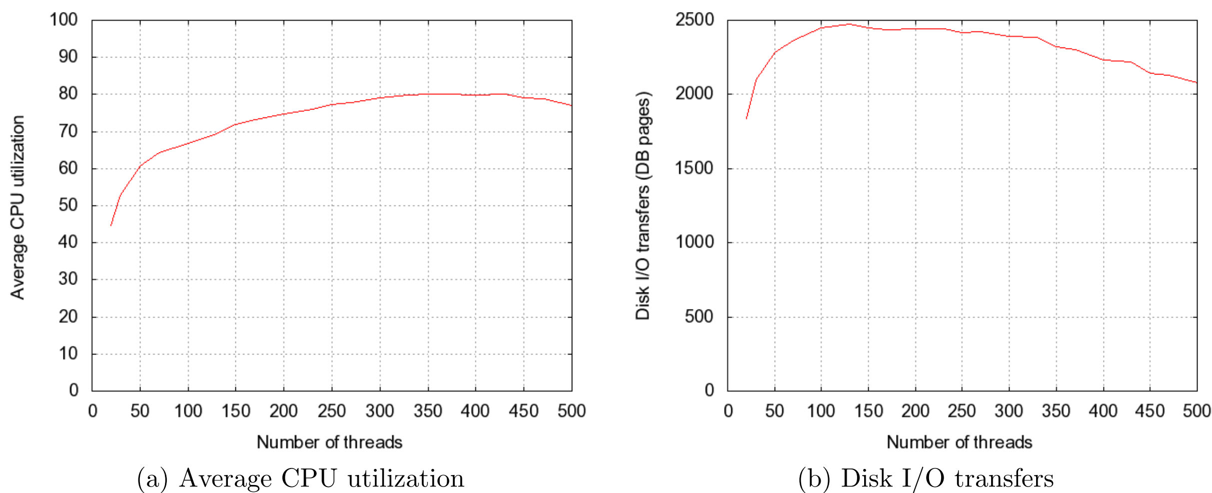


Figure 5.7: Client side response Time vs. Number of Threads for TPCC2-BIGMEM

TPCC2-BIGMEM has a different transaction mix compared to TPCC1-BIGMEM but it uses the same buffer pool size of 1GB. Figure 5.6 shows the throughput at both the server and at the client side. We can see from the server side curve that to



(a) Average CPU utilization (b) Disk I/O transfers

Figure 5.8: CPU and Disk characteristics of TPCC2-BIGMEM

achieve at least 90% of maximum throughput, the multiprogramming level needs to be between 106 and 417. This is a slightly larger range than that of TPCC1-BIGMEM which was between 67 and 278.

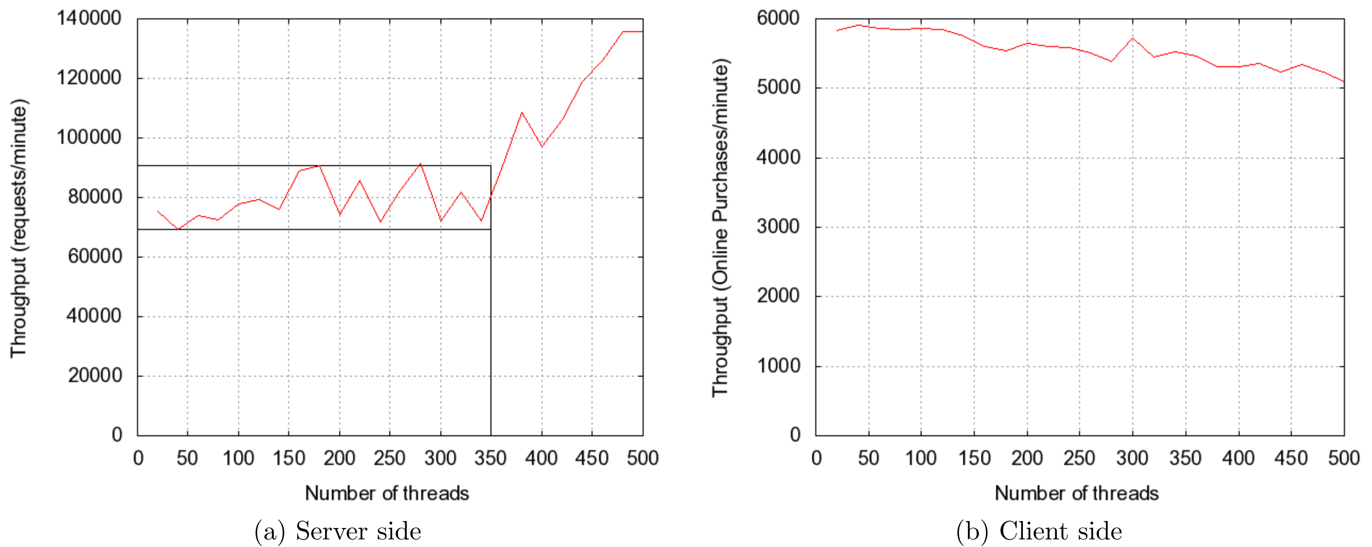
The response time curve (shown in Figure 5.7) also shows how the throughput is inversely related to response time. Figure 5.8 shows the CPU and disk characteristics of the TPCC2-BIGMEM workload. Disk transfers of TPCC2-BIGMEM (shown in Figure 5.8b) are fewer than those of TPCC1-BIGMEM (shown in Figure 5.3b). Table 5.3 summarizes the characteristics of the TPCC2-BIGMEM workload.

Characteristic	Value
Optimal number of threads	106 - 417
Optimal throughput	15753 - 17503 requests/minute

Table 5.3: Characteristics of the TPCC2-BIGMEM workload

5.4 DS2

For the DS2 workload, Figure 5.9 shows the throughput level both at the client and at the server. One thing that is different about this workload is that at high multiprogramming levels, the server throughput looks like it is increasing while the client side throughput level is actually decreasing. Further investigation into the reason for the difference revealed that in the DS2 workload a failed new customer transaction is not counted as a transaction and the client retries until a new customer id is generated. With large numbers of server threads, more new customer transactions fail because of deadlock errors. Because of this, the server has counted



(a) Server side (b) Client side

Figure 5.9: Throughput of the DS2 workload

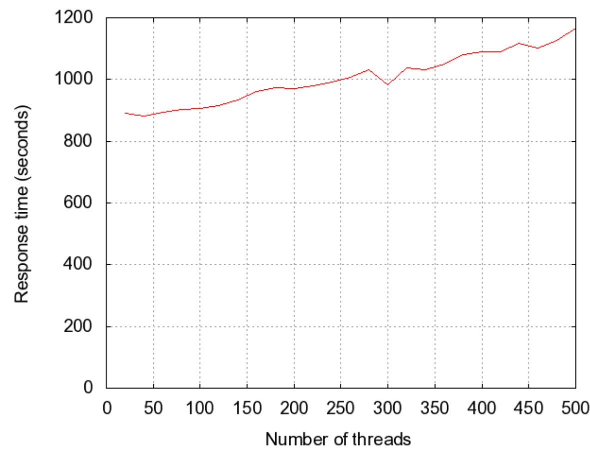


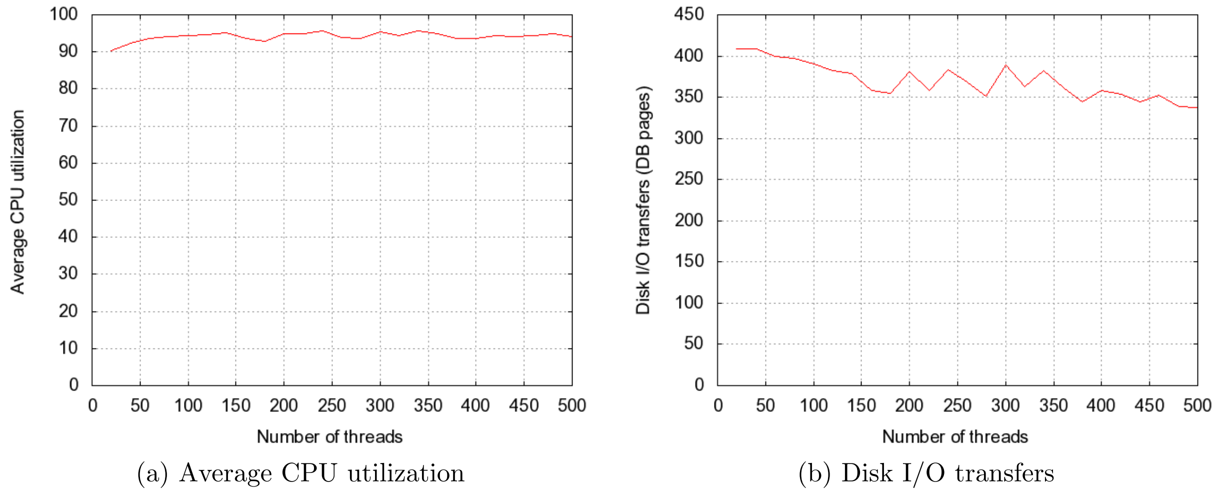
Figure 5.10: Client side response Time vs. Number of threads for DS2

failing transactions that the client has not counted. We will ignore the section of the throughput curve for multiprogramming levels beyond 350 as the rate of deadlock error was high and that would change the transaction mix at the server.

One other characteristic of this workload is that the throughput level stays steady from 20 to 350 threads. Taking 90% of optimal throughput would include a narrower range (150 to 290 threads). The variations at 170 and 270 threads look like noise. Therefore, we will consider any multiprogramming level between 0 and 350 as an optimal setting.

In terms of response time (shown in Figure 5.10), we can see that it is inversely related to throughput. Figure 5.11 shows the disk and CPU characteristics of the DS2 workload. We can see that the DS2 benchmark is CPU bound. This is different

from all of the previous workloads. In terms of disk usage, the disk transfers of DS2 is in the 350 to 400 transfers per second range compared to the previous workloads, which were in the 2,500 disk transfers range. Table 5.4 summarizes the DS2 workload characteristics.



(a) Average CPU utilization

(b) Disk I/O transfers

Figure 5.11: CPU and Disk characteristics of DS2

Characteristic	Value
Optimal number of threads	0 - 350
Optimal throughput	69254.11 - 91277.64 requests/minute

Table 5.4: Characteristics of the DS2 workload

Chapter 6

Algorithm Parameter Sensitivity and Tuning Experiments

This chapter presents the results of the experiments that were conducted to measure the sensitivity of each of the auto-tuning algorithms to their parameters. The TPCC1-BIGMEM workload was chosen as the benchmark for the sensitivity experiments as it is the closest workload to the industry standard TPC-C benchmark. In these sensitivity experiments we fix one of the parameters and vary the other. As was mentioned earlier, the control interval duration is set to 1 minute and will not be varied.

6.1 Hill Climbing

The hill climbing algorithm depends on the following parameters: C and Δ. The C parameter governs how much change in throughput from the last added thread is enough to trigger a change in the number of threads. In other words, if the last thread added to the pool achieves at least C percentage change in throughput compared to the previous thread then we continue adding threads in Δ step size increments.

Section 6.1.1 shows the results of experiments in which we fixed the Δ value and varied C. Section 6.1.2 shows the experiments in which we fixed the C value and varied Δ.

6.1.1 Varying C

Our goal in this section is to characterize the effect of the C parameter on the performance of the Hill Climbing algorithm. In these experiments, we fixed the Δ value and varied C. For each C value we ran the experiments with the TPCC1-BIGMEM workloads. We measured the throughput achieved and the number of

threads used when the algorithm reached steady state. Based on the measured results, we studied how C affects the ability of the Hill Climbing algorithm to maximize performance.

Table 6.1 shows the results of these experiments. The first column shows the C value setting. The second and third columns show the average and standard deviation of the throughput level measured in requests/minute. The fourth and fifth columns show the average and standard deviations respectively of the number of threads. Figure 6.1 shows graphs of the throughput and the average number of threads as functions of C.

	Throughput (requests/minute)		Number of threads		
C	Average	StdDev	Average	StdDev	StdDev/Avg
0.1	14652.61	2796.28	223.30	64.34	0.207454
0.2	14663.29	1815.51	126.10	26.16	0.246894
0.3	13968.71	1960.75	148.12	36.57	0.259636
0.4	14337.62	1813.12	115.97	30.11	0.136209
0.5	13084.16	1541.68	79.29	10.80	0.188181
0.6	12257.41	1704.94	63.29	11.91	0.258363
0.7	11278.64	1729.59	52.91	13.67	0.258363
0.8	8479.94	1449.35	30.10	9.14	0.303654
0.9	8975.50	1306.24	32.83	6.50	0.19799

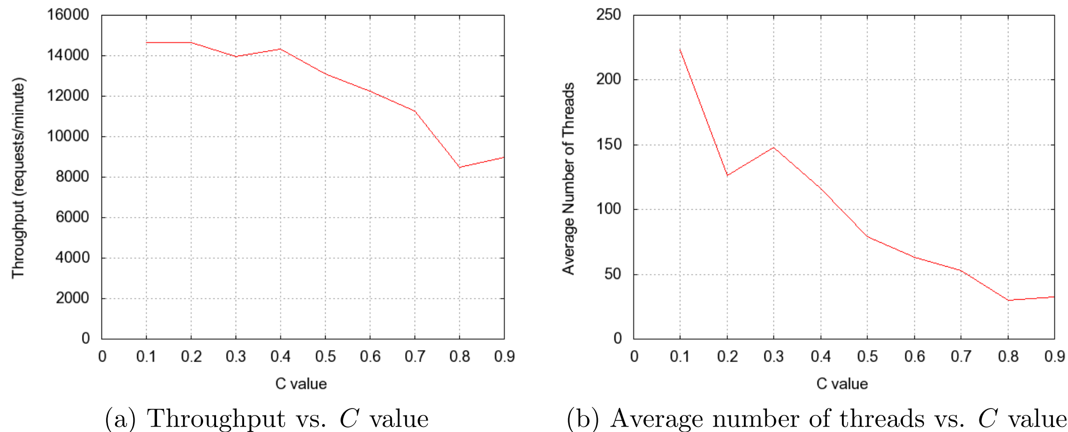
Table 6.1: Various throughput levels with fixed Δ of 10

The results show that our implementation of the hill climbing algorithm behaves as expected. A low C value makes the controller more aggressive in adding threads while high C value makes it more conservative. At 0.1, the average number of threads is 223 while at 0.9 the algorithm is very conservative and keeps the number of threads on average at 32 threads.

Since our primary goal is to increase throughput, a value of C greater than 0.5 is going to work against our goal. For our secondary goal, we aim at lowering the number of threads without hurting throughput. In order to achieve both of those goals, we will need to set C to a value that is less than 0.5 but not less than 0.2. Any C value between 0.3 and 0.5 will suffice. We will set C to 0.4 in our auto-tuning experiments.

6.1.2 Varying Δ

The purpose of these experiments is to fix the value of C and vary Δ. The goal is to study the effect of Δ on the performance of the Hill Climbing algorithm. For each Δ value, we ran the TPCC1-BIGMEM workload and allowed the Hill Climbing algorithm to perform its work. We recorded throughput and average number of



(a) Throughput vs. C value (b) Average number of threads vs. C value

Figure 6.1: Throughput and Number of threads vs. C

threads for every settings of Δ after the algorithm has reached steady state. We then studied how Δ affected the performance of the Hill Climbing algorithm.

We have done two sets of experiments, in the first, we fixed C to 0.3 and varied Δ. In the second, we fixed C to 0.5 and varied Δ. We choose to try 0.3 and 0.5 of C to cover the range of good values of C that we determined from the previous section's experiment.

Table 6.2 show the results of the experiments were we fixed C to 0.3 and varied Δ. The first column is the Δ value. The second and third columns show the average and standard deviation of the throughput level. The fourth and fifth columns show the average and standard deviation of the number of threads used.

Δ	Throughput		Number of threads	
value	Average	StdDev	Average	StdDev
10	13968.71	1960.75	148.12	36.57
20	14088.97	2083.33	155.65	35.48
40	14285.74	1988.28	112.08	35.49

Table 6.2: Various throughput levels with fixed C value of 0.300

The results show that there is not a major effect of the step size on the throughput levels. Therefore, we will choose a small value for Δ. A smaller value for Δ is desirable as it lowers abrupt fluctuations in throughput. In addition, it allows the algorithm to take finer steps in order to reach a number of threads setting that is closer to the optimal values. Conversely, smaller step size Δ will make the algorithm react slowly, which might not be desirable.

In the second set of experiments, the C value was set at 0.5 and Δ was varied between 10 and 60. Table 6.3 shows the results. With the higher C value, the con-

troller is now more conservative in taking steps to increase the number of threads. This is evident from the fact that the average number of threads is lower compared to the settings were C was set to 0.3. With respect to the Δ, a lower value for Δ also minimizes the variations in throughput and helps achieve higher throughput. This is true because with a smaller Δ and a more conservative C value, the controller takes more cautious finer grained steps.

Δ	Throughput		Number of threads	
value	Average	StdDev	Average	StdDev
10	13084.16	1541.68	79.29	10.80
20	14167.03	1198.10	58.86	12.88
40	13106.03	2214.60	76.50	23.55
50	10570.95	3297.31	54.80	31.28
60	11150.19	3457.77	60.80	37.28

Table 6.3: Various throughput levels with fixed C value of 0.5

Based on the results of the above two sets of experiments, we will fix the value Δ to 10.

6.2 Global Parabola Approximation

The Global Parabola Algorithm has two parameters: Γ, and Δ. The Γ parameter limits the move from the current number of threads to the next number of threads. The Δ parameter is only used when the parabola approximation generates a curve that is concave up. Such a curve reflects noise in the data and hence the data points are not useful. The Δ parameter is used in this case to just wiggle the control variable in either the positive or negative direction. Because the Δ parameter should be used only when noisy data is detected, our sensitivity experiments will concentrate on studying the effect of the Γ parameter. Noise in the data can arise from many reasons including smaller control intervals, a transient change of transaction mix, or a change in resource utilization on the hardware machine due to other applications sharing disk and memory resources of the server.

For the sensitivity experiments, we fixed the Δ parameter to 5 and varied the Γ parameter from 0.1 to 0.9.

Table 6.4 shows the effect of changing the Γ value while keeping the Δ fixed. The first column shows the Γ value used. The second and third columns show the measured throughput level. The fourth and fifth columns show the number of threads.

In order to better understand this data, we have also collected information about the number of times each rule of the Global Parabola Approximation algorithm was applied. The data is shown in Table 6.5.

Γ	Throughput (requests/minute)		Number of threads	
	Average	StdDev	Average	StdDev
0.1	13442.12	2828.79	56.07	10.03
0.2	15939.70	1972.75	91.41	16.41
0.3	16054.75	1888.48	121.17	32.27
0.4	16156.59	869.51	111.54	36.46
0.5	15959.93	1715.21	156.82	61.28
0.6	15992.43	1709.76	137.28	50.00
0.7	15807.90	1695.17	219.43	54.94
0.8	15759.24	1704.44	220.20	77.27
0.9	16104.90	1800.46	125.67	43.42

Table 6.4: Effect of Γ value on the throughput and number of threads for the Global Parabola Approximation

Γ	Num control intervals	$a < 0$	Γ used	$a > 0$	Duplicate n^* values
0.1	100	87	62	7	6
0.2	99	91	29	3	5
0.3	99	91	15	6	2
0.4	98	86	5	7	5
0.5	97	93	8	1	3
0.6	100	92	3	7	1
0.7	100	93	4	5	2
0.8	101	97	4	3	1
0.9	99	89	1	5	3

Table 6.5: Number of times branches of the Global Parabola are taken for various Γ values

The first column of Table 6.5 is the Γ value. The second column is the number of control intervals. As was mentioned previously, each interval is 1 minute in length. The third column is the number of times the parabola approximation gave a valid a coefficient that is less than 0. The fourth column is the number of times in which there was a valid a coefficient but the control value was so large that the Γ limit was applied. The fifth column is the number of times the a coefficient was positive. In this case, the parabola is concave up and hence can not be used for choosing the next control variable value. When $a > 0$, the Δ value will be used to wiggle the control variable so that noise can be avoided. The last and final column is the number of times the new control value and previous value were equal. Since duplicate values would not help define the parabolic function, the Δ rule is used to wiggle the control variable.

We can see from Table 6.5 that as the Γ value increases, the Γ rule is less

frequently used to limit the step size of the control variable. In addition, for almost all values of Γ, the percentage of times the throughput curve is approximated correctly by the parabola (i.e. coefficient is positive) was high compared to the total number of samples. This indicates that the controller relied on the parabolic function to generate the next control value rather than randomly wiggling the control variable by Δ steps.

Based on the sensitivity experiments above, we can see that if Γ is higher than 0.5 it has little effect. The Γ value helps control the aggressiveness of the algorithm. There are cases when the algorithm will try large jumps in the control variable and the Γ value helps dampen those jumps. We will use a value of 0.333 in our experiments.

6.3 Local Parabola Approximation

The Local Parabola algorithm has the same parameters as the Global Parabola algorithm. In the same manner, we have conducted experiments were we varied the Γ value from 0.1 to 0.9 while keeping Δ fixed at 5. Tables 6.6 and 6.7 summarize the results.

Γ	Throughput		Number of threads	
	Average	StdDev	Average	StdDev
0.1	12512.24	1658.83	40.54	8.64
0.2	12246.56	2441.62	48.62	16.27
0.3	12907.50	3076.28	45.28	14.56
0.4	13862.48	2313.27	49.27	12.38
0.5	13638.60	2133.30	50.82	11.52
0.6	14346.95	1950.68	57.53	15.66
0.7	14876.37	1911.11	70.93	11.73
0.8	14885.00	3275.64	79.90	26.48
0.9	12100.96	2156.23	42.45	15.75

Table 6.6: Effect of Γ value on the throughput and number of threads for the Local Parabola Approximation

The results shown in Table 6.7 show that in more than 50% of the time wiggling the control variable was used to make control decisions. Wiggling the control variable using Δ is done when the a coefficient is positive or if the algorithm produces duplicate points that do not help define a parabolic function. The fact that the Δ rule was used indicates that the algorithm is affected by normal measurement variations (noise). This means that in most control intervals the algorithm is making small random moves. Because of noise sensitivity, we will not by studying this algorithm any more.

Γ	Num control intervals	$a < 0$	Γ used	$a > 0$	Duplicate n^* values
0.1	106	40	17	36	30
0.2	102	39	6	36	27
0.3	107	39	1	29	39
0.4	103	38	0	35	27
0.5	103	35	0	31	34
0.6	102	43	1	25	34
0.7	100	38	0	32	30
0.8	100	38	2	33	29
0.9	106	42	1	41	23

Table 6.7: Number of times branches of the Local Parabola are taken for various Γ values

Chapter 7

Evaluation and Results

This chapter presents experimental results along with an analysis of these results. Section 7.1 will show how each of the auto-tuning algorithms performed with stable workloads. Section 7.2 will show how the algorithms reacted to changing workloads.

7.1 Auto-tuning Experiments

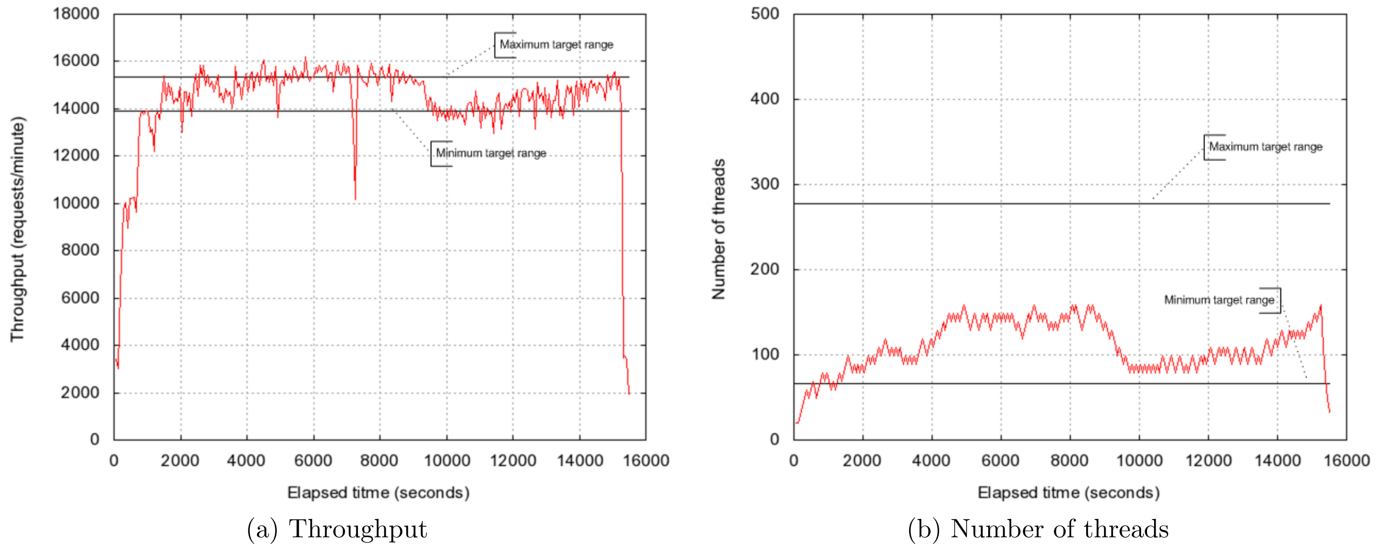
In this section we will show the results of how each of the auto-tuning algorithms introduced in Section 3.2 behaved with our workloads. We will compare these results to the throughput curves that we have deduced in Chapter 5.

7.1.1 TPCC1-BIGMEM

Figure 7.1 shows the different multiprogramming levels that the controller has chosen using the Hill Climbing algorithm as well as the throughput levels of the server. The left graph shows the throughput levels of the server and the right graph shows the current threads count. We can see from the graph that the Hill Climbing algorithm was able to stay within the region of optimal number of threads for the TPCC1-BIGMEM workload. On thing to note is that the algorithm was on the lower side of the range. This is a good behaviour as it helps achieve our second goal of minimizing the number of threads. Throughput was also within its expected region.

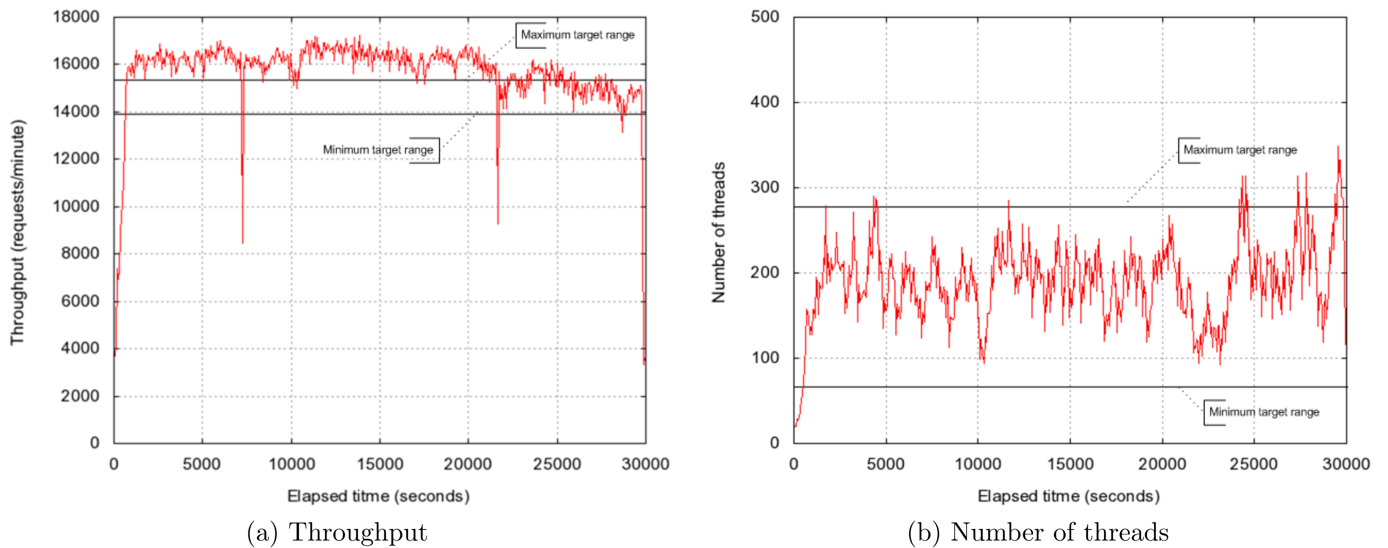
Figure 7.2 shows that Global Parabola Approximation algorithm was also able to reach the region of optimal values for the number of threads setting. The values chosen by this algorithm were on the mid to high end of the range. In terms of throughput, this algorithm was able to match and exceed the values of the optimal region that we have found previously.

To compare the two algorithms' performance, we can see that although the two algorithms were able to find the region of optimal values, the Hill Climbing



(a) Throughput (b) Number of threads

Figure 7.1: Performance of the Hill-Climbing algorithm with the TPCC1-BIGMEM workload

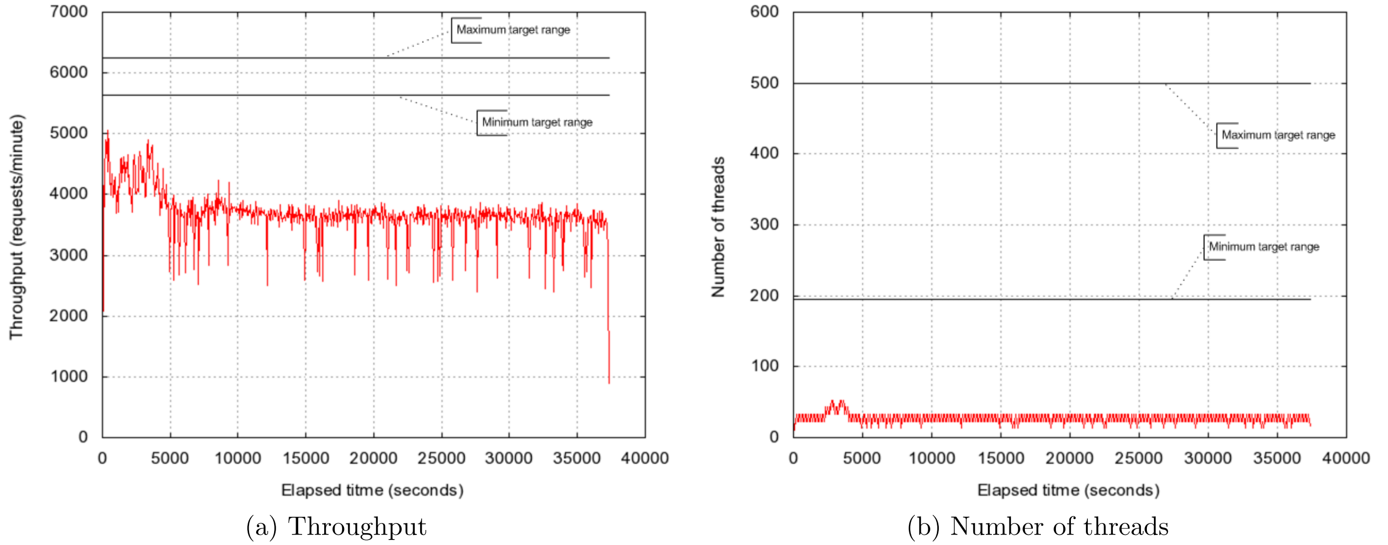


(a) Throughput (b) Number of threads

Figure 7.2: Performance of the Global Parabola Approximation algorithm with the TPCC1-BIGMEM workload

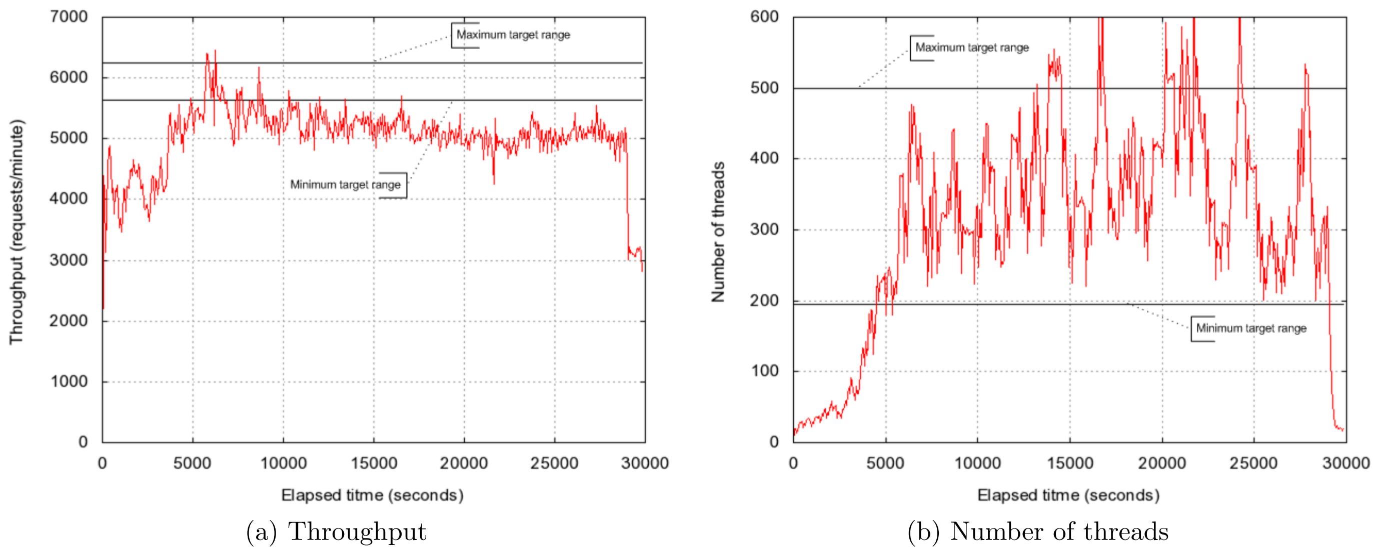
algorithm was taking smaller n^* values. Another observation is that the Global Parabola Approximation algorithm shows more variability in its choice of the n^* values.

7.1.2 TPCC1-SMALLMEM



(a) Throughput (b) Number of threads

Figure 7.3: Performance of the Hill Climbing algorithm with the TPCC1-SMALLMEM workload



(a) Throughput (b) Number of threads

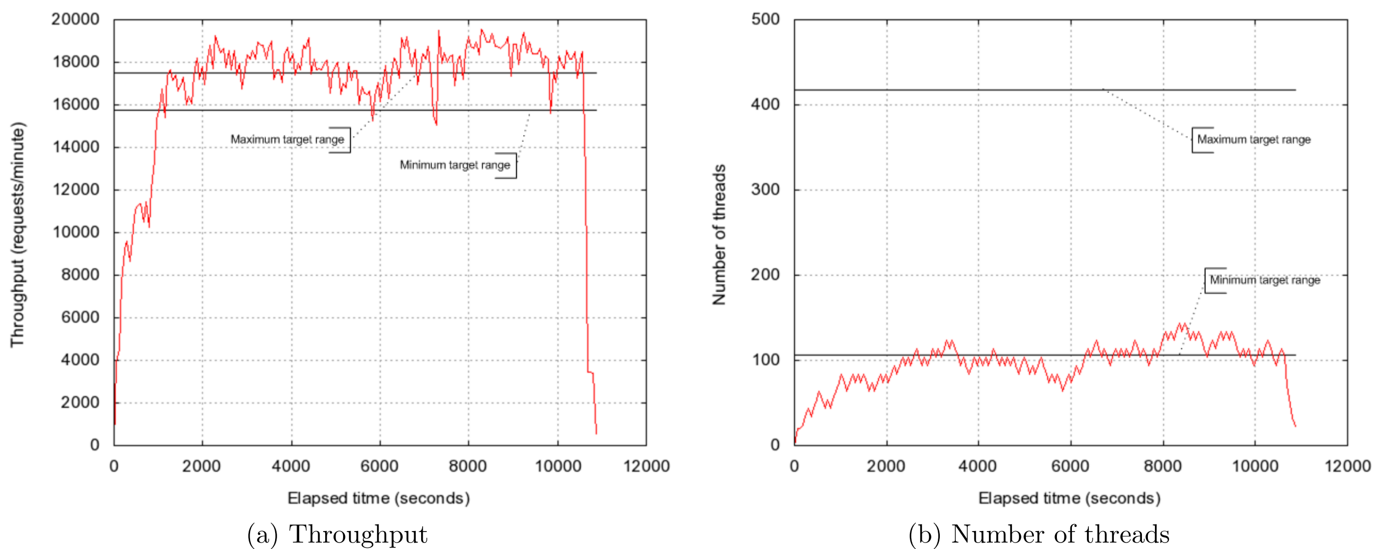
Figure 7.4: Performance of the Global Parabola Approximation algorithm with the TPCC1-SMALLMEM workload

Figure 7.3 shows how the Hill Climbing algorithm performed with the TPCC1-SMALMEM workload. We can see that algorithm kept the number of threads at around 28 threads on average. Clearly the algorithm sees that the amount of gain in throughput is not enough to increase the number of threads. The standard deviation for the number of threads is 6.71 which means that the algorithm did not attempt to make wild changes. Compared to the characterization experiments of the TPCC1-SMALLMEM, the Hill Climbing decisions were very conservative. As a result, the average throughput with auto-tuning was around 3,676 requests/minutes which is almost half of that at the peak of 6017 requests/minute that we found previously. These results indicate that the Hill Climbing algorithm failed in reaching the region of optimal values and was very conservative.

On the other hand, Figure 7.4 shows how the Global Parabola approximation algorithm performed with TPCC1-SMALLMEM. The Global Parabola approximation algorithm had an average throughput of 5013 requests/minute which is slightly below the optimal region. However, the average number of threads of 307 is within the optimal region. We can see that although the algorithm was able to maintain an average number of threads within the optimal region, it was doing large swings in the control variable n^*. The fact that the throughput level did not achieve that in the optimal region might be due to the large variability of n^*.

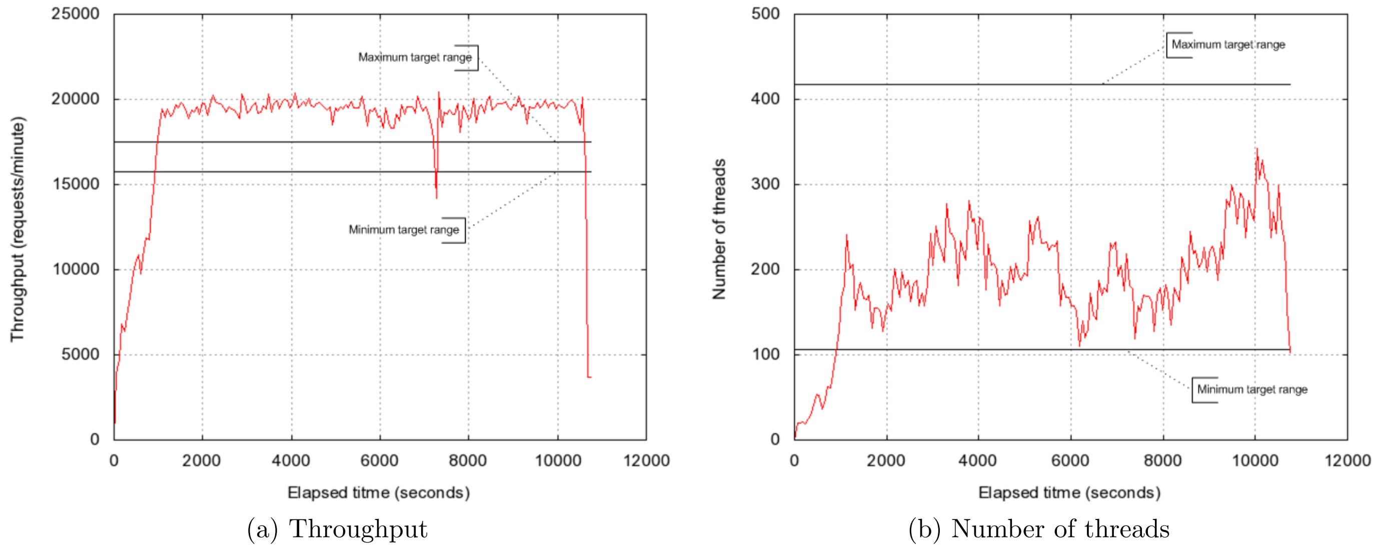
Comparing the performance of the two algorithms, we can see that the Hill Climbing algorithm failed in reaching our region of optimal value but the Global Parabola Approximation did not.

7.1.3 TPCC2-BIGMEM



(a) Throughput (b) Number of threads

Figure 7.5: Performance of the Hill Climbing algorithm with the TPCC2-BIGMEM workload



(a) Throughput

(b) Number of threads

Figure 7.6: Performance of the Global Parabola Approximation algorithm with the TPCC2-BIGMEM workload

Figure 7.5 shows the performance of the Hill Climbing algorithm. The Hill Climbing algorithm was able to reach the region of optimal values with an average number of threads of 102. Again, this value is on the lower end of the range of optimal values. The optimal throughput region was achieved. These results show how the algorithm succeeded in reaching good control values for n^*.

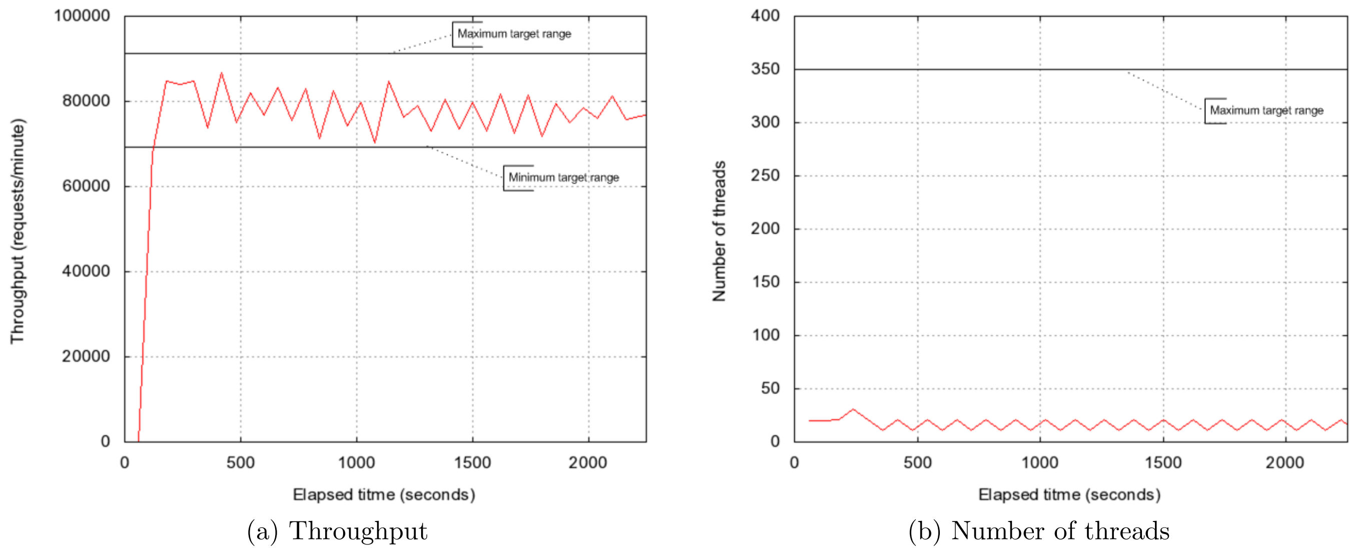
Figure 7.6 shows the performance of the Global Parabola Approximation algorithm. As usual, the Global Parabola Approximation algorithm is more aggressive in changing the number of threads. The average number of threads of 202 was well within the optimal region values. In terms of throughput, the algorithm was able to achieve better throughput than our optimal settings. This might have been a consequence of the large variability in the number of threads.

One noticeable difference between the two algorithms is that the Global Parabola Approximation algorithm was able to reach the optimal region in 961 seconds (16 minutes) while the Hill Climbing algorithm took 2,343 seconds (39 minutes). Another difference is that there is more variability with Global Parabola Approximation algorithm.

7.1.4 DS2

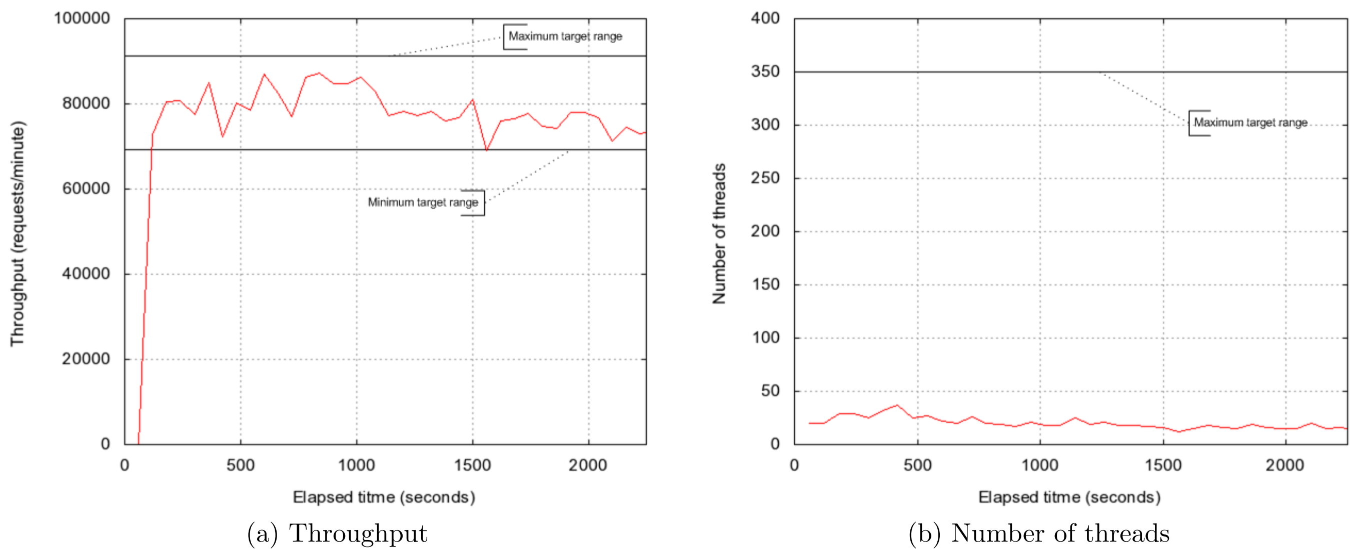
Figure 7.7 shows the results for the Hill Climbing algorithm. The graph shows that the algorithm was able to reach the optimal region setting and was able to achieve the corresponding throughput. The algorithm was varying the number of threads between 11 and 21. Optimal throughput was also achieved.

The Global Parabola Approximation algorithm (shown in Figure 7.8) was also



(a) Throughput (b) Number of threads

Figure 7.7: Performance of the Hill Climbing algorithm with the DS2 workload



(a) Throughput (b) Number of threads

Figure 7.8: Performance of the Global Parabola Approximation algorithm with the DS2 workload

able to achieve the optimal number of threads settings. This algorithm was changing the number of threads between 12 and 25. It was very interesting how this algorithm was making very fine adjustments at this low number of threads values. In terms of throughput, the algorithm was also able to achieve the required throughput levels.

In this experiment one interesting finding is that both algorithms attempted a number of threads setting that is lower than 20 and was able to achieve the target throughput levels. In our characterization experiments we only attempted numbers

from 20 to 500. These algorithms were able to find an optimal value that we did not attempt.

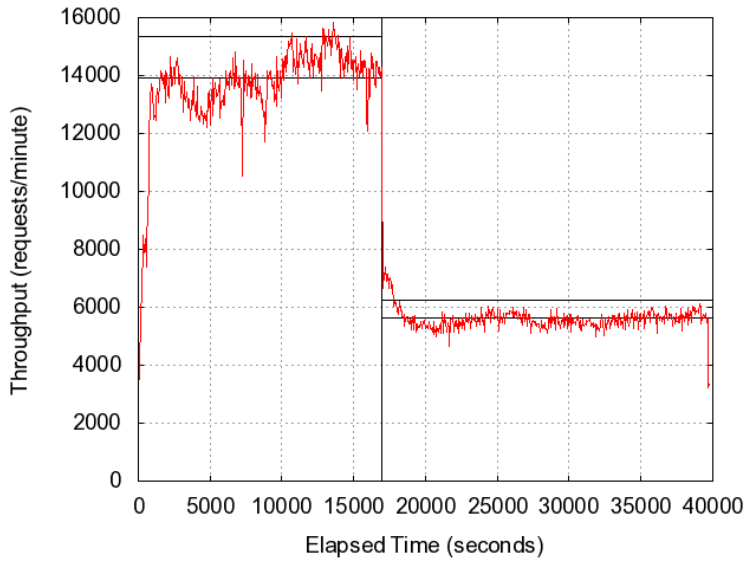
7.2 Changing Workload Experiments

In this section we show the results of the Changing Workload experiments. In these experiments we started the server with one workload and configured it to use one of the auto-tuning algorithms. Half way through the experiment we switched to a different workload without restarting the server. We repeated the same experiment for the second algorithm. The purpose of these experiments is to see how well the auto-tuning algorithms respond to changing workload conditions. Workload switching was done by either configuring the client to change the transaction mix, or by configuring the server to change its buffer pool size, depending on which workloads we are switching between. We will compare the results of these experiments to those reported in the characterization experiments in Chapter 5 to validate how well the algorithms were able to stabilize after the workload switch happened.

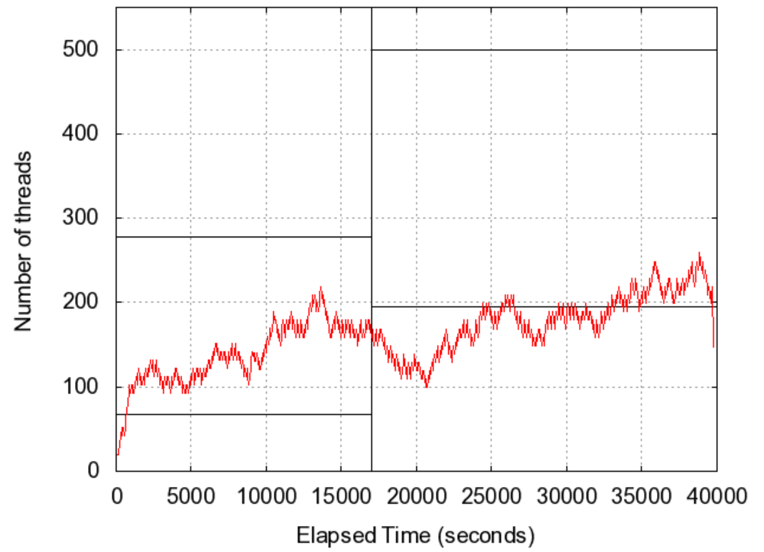
7.2.1 TPCC1-BIGMEM to TPCC1-SMALLMEM

In this experiment we switched the workload from TPCC1-BIGMEM to TPCC1-SMALLMEM. The two workloads have the same transaction mix. The TPCC1-SMALLMEM workload, has a smaller buffer pool size. In the workload characterization experiments we found that the throughput levels of the TPCC1-BIGMEM is higher than those for TPCC1-SMALLMEM. When we change from the first workload to the second, we expect that there will be a period of fluctuation in n^* followed by a stabilization period. When the algorithms stabilize on a new multiprogramming level, we want to see how well this value compares to the values we found in the workload characterization experiments.

Figure 7.9 shows how the Hill Climbing algorithm reacted when we switched from TPCC1-BIGMEM to TPCC1-SMALMEM 16,935 seconds into the run. When we switched the workloads, the number of threads value started to drop from the time the switch happened until 20,000 seconds into the run. This indicates that the algorithm has noticed there is a drop in throughput and it reacted by reducing the number of threads until it reached 100 threads. When the Hill Climbing algorithm starts to go in the downward direction, it continues to do so until it notices a drop in performance. After 20,000 seconds into the run, the algorithm started to go in the positive direction and increased the multiprogramming level steadily from 20,000 seconds to 35,000 seconds. Because the Hill Climbing algorithm is slow in making changes, it took some time to reach the new optimal multiprogramming level of the second workload. These results show that the Hill Climbing algorithm was effective in handling the workload switch and was able to react appropriately.

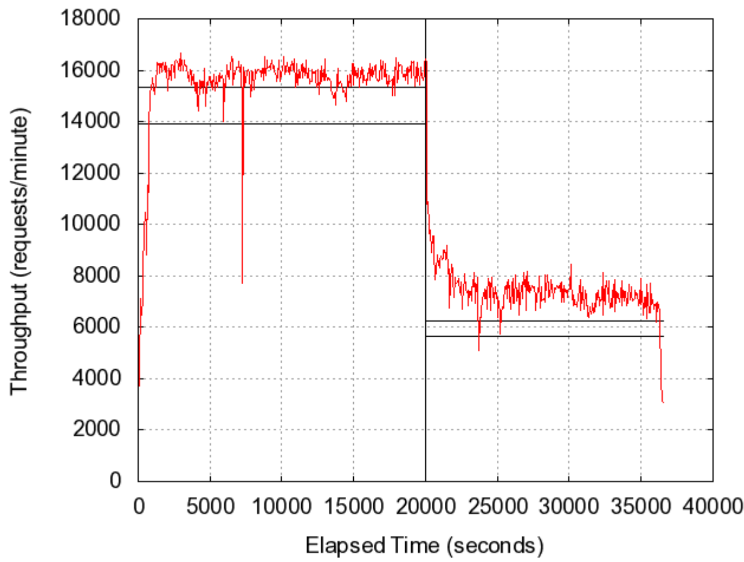


(a) Throughput

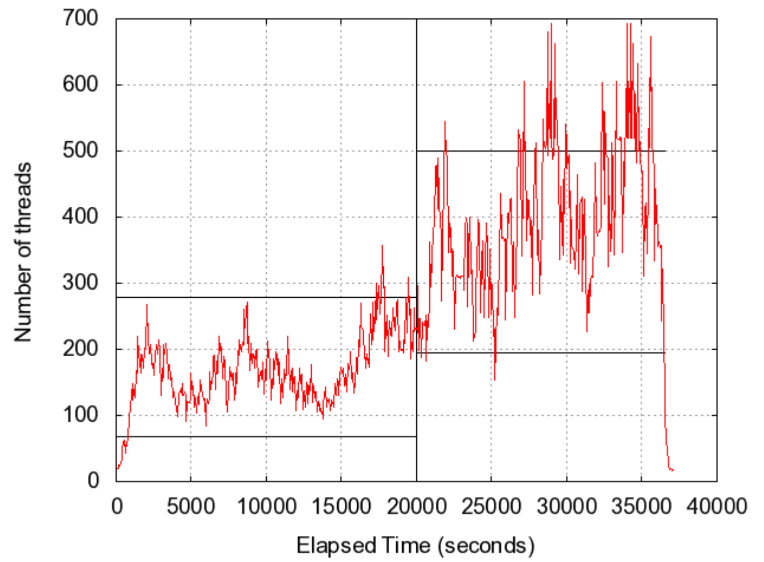


(b) Number of threads

Figure 7.9: Performance of the Hill Climbing algorithm while switching workloads from TPCC1-BIGMEM to TPCC1-SMALMEM



(a) Throughput



(b) Number of threads

Figure 7.10: Performance of the Global Parabola Approximation algorithm while switching workloads from TPCC1-BIGMEM to TPCC1-SMALMEM

If we observe how the Hill Climbing algorithm did in the second phase (TPCC1-SMALLMEM), we can see that it was able to reach the optimal number of threads settings. When we ran the auto-tuning algorithm for the TPCC1-SMALLMEM workload alone with Hill Climbing in Section 7.1.2, the algorithm was not able to reach the optimal number of threads region. The difference between the two experiments is the initial value of the multiprogramming level. In the workload switch experiment, the algorithm started at a higher multiprogramming level than in the auto-tuning experiment. This shows that the starting value can influence the performance of the Hill Climbing algorithm.

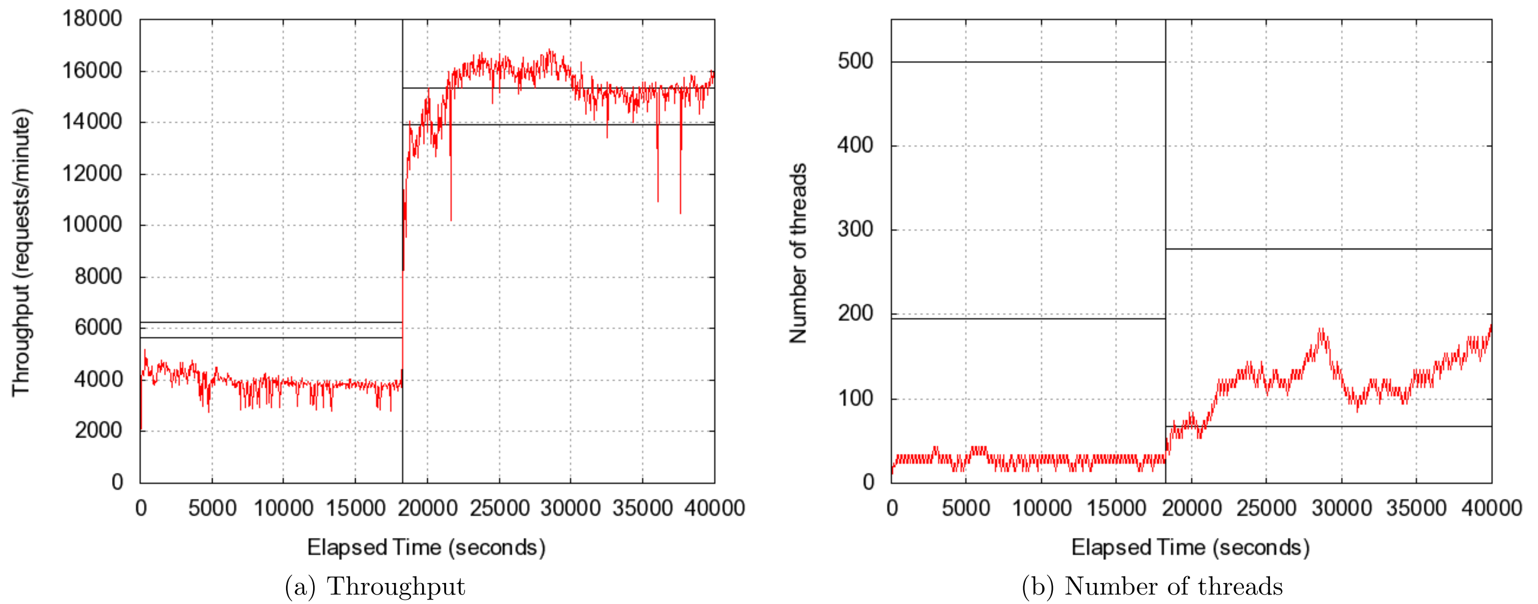
Figure 7.10 shows how the Parabola Approximation algorithm reacted when we switched from TPCC1-BIGTMEM to TPCC1-SMALLMEM 20,000 seconds into the run. When the workload switch happened, the algorithm reacted by changing the number of threads appropriately into the new optimal multiprogramming region of the second workload. Because the Global Parabola approximation algorithm has a higher variability in the number of threads, we can see that it was able to recover more quickly from the workload switch. Optimal throughput was also achieved for both workloads. These results show that the algorithm was able to handle the workload switch successfully.

In comparing how both algorithms performed, the Global Parabola Approximation algorithm is more aggressive in making changes to the multiprogramming level. Because its step size is governed by the parabolic function, it was able to recover more quickly than the Hill Climbing algorithm and was not affected by the starting value of the control variable n^*. In terms of throughput, the Global Parabola Approximation algorithm achieved higher throughput levels in each of the workload switch phases than Hill climbing did.

7.2.2 TPCC1-SMALLMEM to TPCC1-BIGMEM

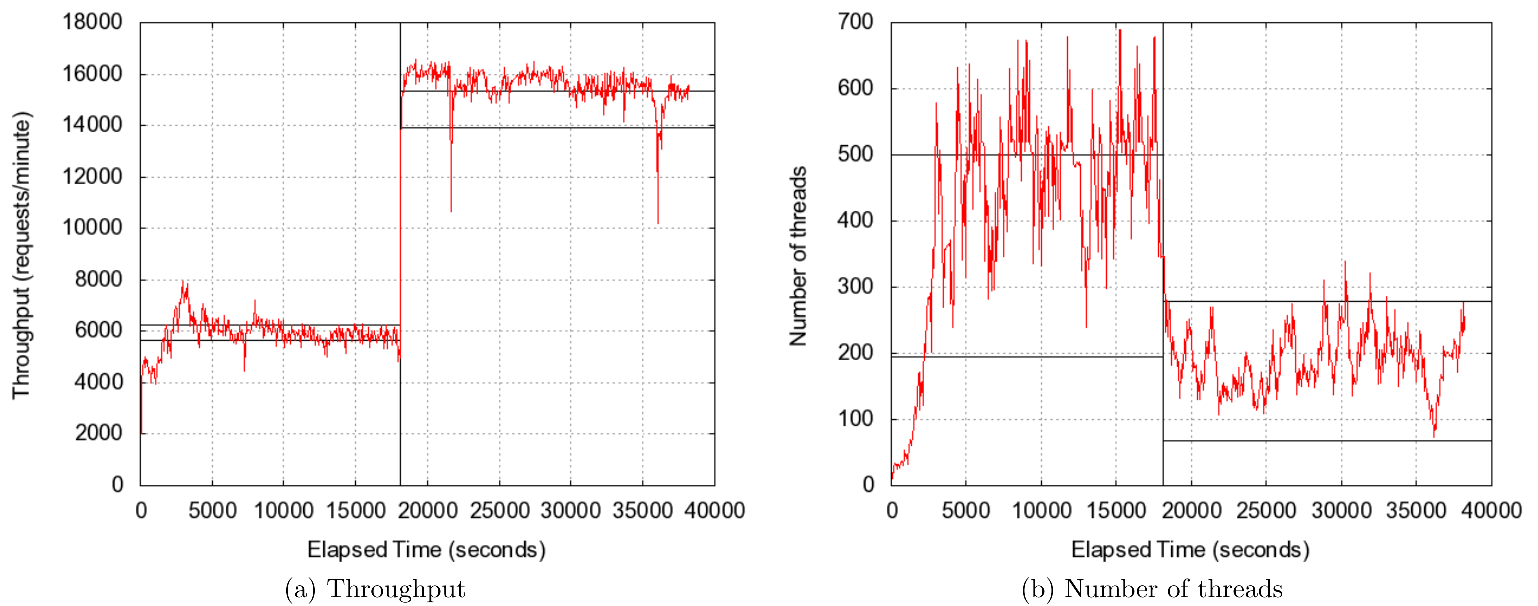
In this section, we performed a workload switch that is the reverse of the switch from the previous section. Here, we switched from a workload that prefers a high number of threads (TPCC1-SMALLMEM) to a workload that prefers a lower number of threads. We expect that initially the algorithm will stabilize on high number of threads suitable for the TPCC1-SMALLMEM workload and then move to lower number of threads suitable to the TPCC1-BIGMEM workload.

Figure 7.11 show how the Hill Climbing algorithm handled the workload switch. In the first phase of the experiment, the Hill Climbing algorithm stabilized on a lower multiprogramming level than the optimal region for this workload. This is of no surprise to us as this is consistent with how it performed in the auto-tuning experiment with a stable workload in Section 7.1.2. When the workload switch happened at 18,254 seconds into the run, the algorithm noticed the increase in throughput. It started to adjust the multiprogramming level appropriately until it reached the optimal region of the second workload. In terms of throughput, the algorithm was able to achieve the required throughput for the second phase.



(a) Throughput (b) Number of threads

Figure 7.11: Performance of the Hill Climbing algorithm while switching workloads from TPCC1-SMALLMEM to TPCC1-BIGMEM



(a) Throughput (b) Number of threads

Figure 7.12: Performance of the Parabola Approximation algorithm while switching workloads from TPCC1-SMALLMEM to TPCC1-BIGMEM

These results show that the algorithm was able to handle the workload switch appropriately; however, it did not meet our expectation for the first phase of the run.

Figure 7.12 shows how the Global Parabola Approximation algorithm handled the workload switch. In the first workload phase, the algorithm stabilized on a high number of threads that is in the high region of the optimal values. After the workload switch was performed at 18,075 seconds into the run, the algorithm adjusted the number of threads appropriately to match the new workload conditions. The algorithm was able to stabilize on the optimal region for TPCC1-BIGMEM. The corresponding throughput level for both workloads was also achieved. These results show that the Global Parabola Approximation algorithm was able to handle the switch in the reverse direction effectively.

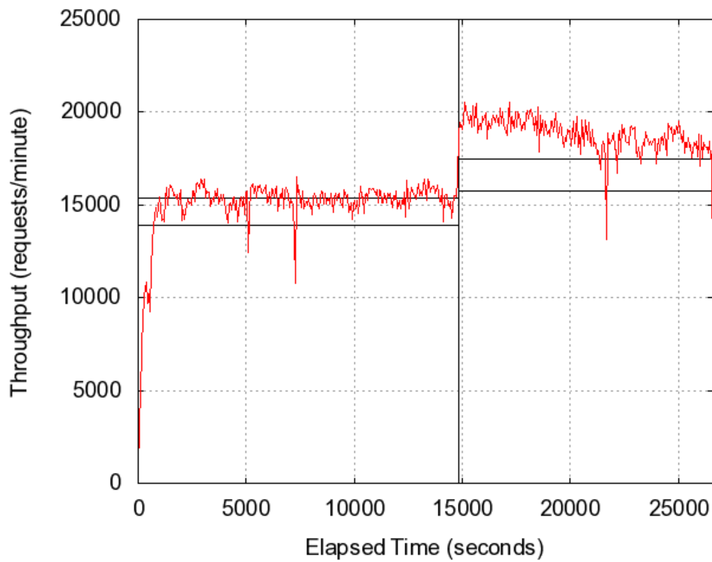
To compare the performance of the algorithms, the Global Parabola Approximation algorithm was more aggressive in changing the multiprogramming level. It had more variability than the Hill Climbing algorithm. Nonetheless, it was able to perform as expected in both phases of the run. The Hill Climbing algorithm failed to reach the optimal multiprogramming region for the first workload but was able to handle the workload switch appropriately and stabilize on a new multiprogramming for the second workload.

7.2.3 TPCC1-BIGMEM to TPCC2-BIGMEM

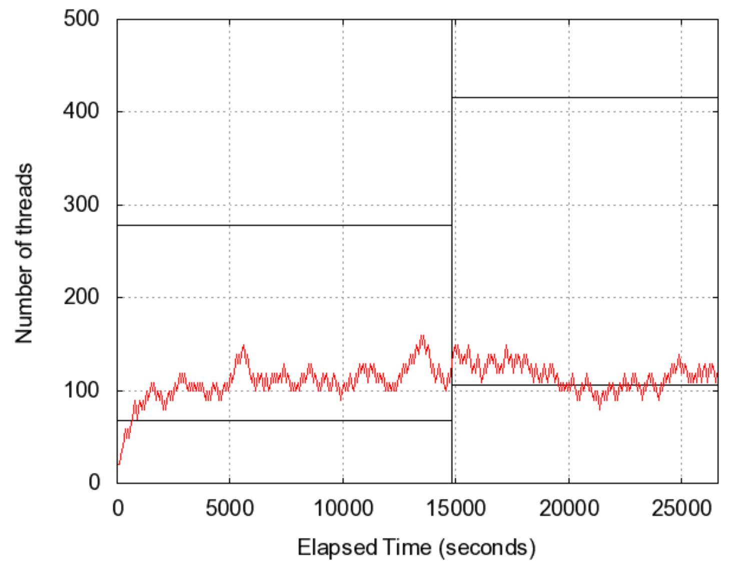
In this section we switched from TPCC1-BIGMEM to TPCC2-BIGMEM. The two workloads are similar in terms of the size of the server buffer pool size but are different in the transaction mix. The region of optimal throughput levels for TPCC1-BIGMEM is slightly lower than that of TPCC2-BIGMEM; however, the optimal number of threads levels are almost the same. We expect that when we switch from the first workload to the second, that the controller will keep the number of threads almost at the same level as the optimal number of threads regions are very similar. The challenge is that the throughput levels for the two workloads is different, so we are trying to see if the two algorithms will continued to keep the number of threads almost at the same level even though the throughput level has changed.

Figure 7.13 shows how the Hill Climbing algorithm handled the workload switch. We can see that the algorithm was able to reach the optimal region for the first workload. When the workload switch happened at 14,833 seconds into the run, the algorithm continue to keep the number of threads at the same level of around 116 threads. Although the throughput level increased from 15,000 requests/minute to 18,700 requests/minute, the algorithm kept the number of threads setting at the same level for the new workload.

Figure 7.14 shows how the Global Parabola Approximation algorithm handle the workload switch. In the first phase of the experiment, the algorithm settled on an average number of threads of 175. When the workload switch happened at 14,832 seconds into the run, the algorithm settled on a new average of 226 threads. This

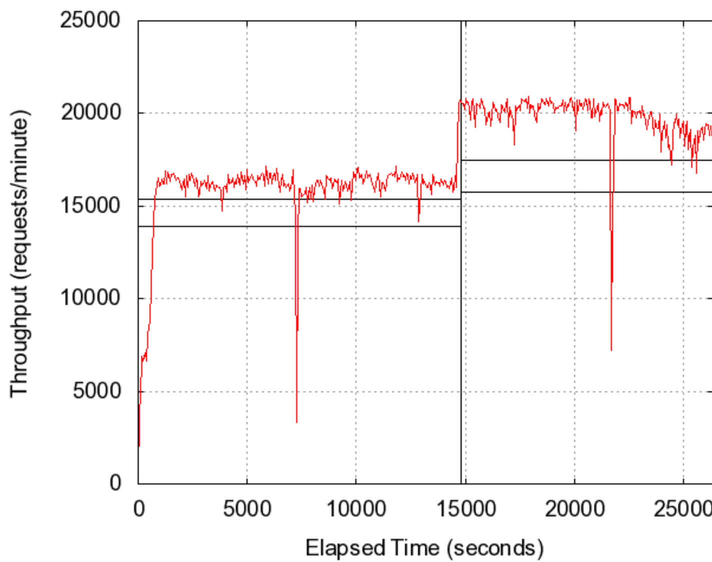


(a) Throughput

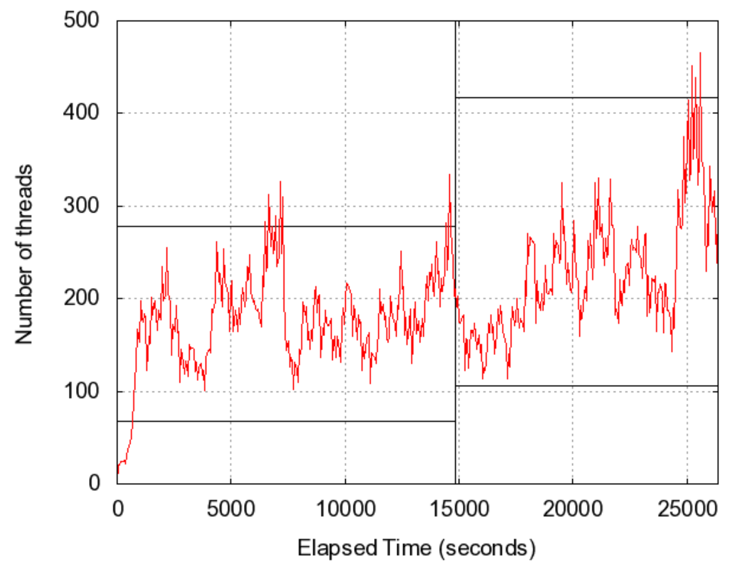


(b) Number of threads

Figure 7.13: Performance of the Hill Climbing algorithm while switching workloads from TPCC1-BIGMEM to TPCC2-BIGMEM



(a) Throughput



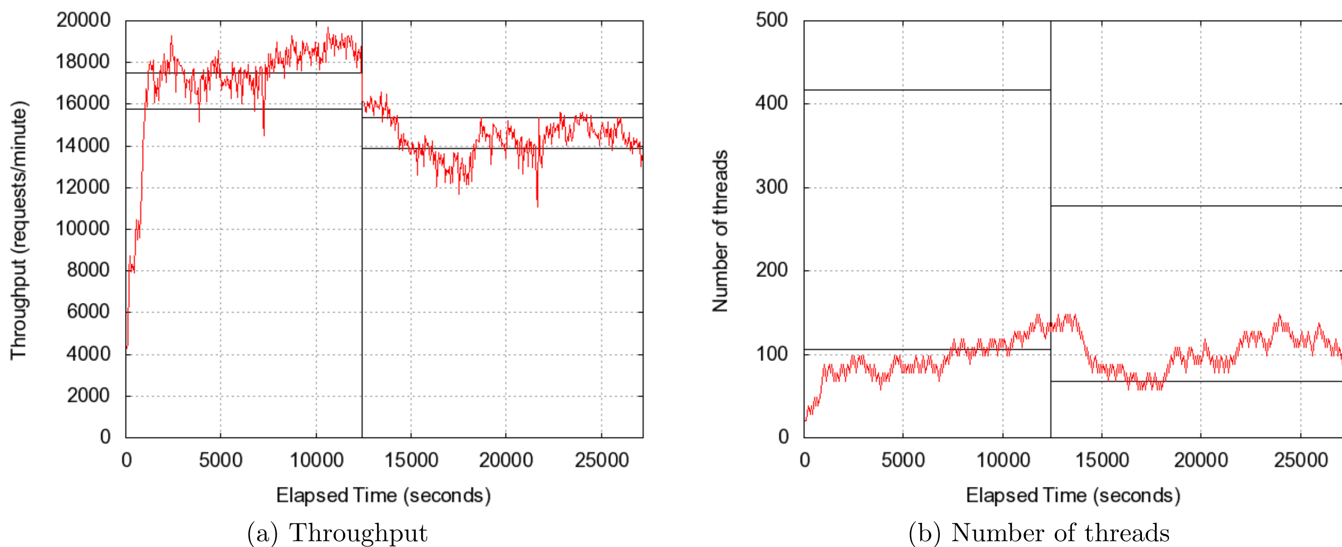
(b) Number of threads

Figure 7.14: Performance of the Global Parabola Approximation algorithm while switching workloads from TPCC1-BIGMEM to TPCC2-BIGMEM

number is slightly higher but still within the allowable values for the second workload. The throughput in the first phase of the run averaged 16,186 requests/minute. In the second phase, the throughput averaged 19,822 requests/minute. We can see that the throughput has changed and that the algorithm was able to adjust the number of threads appropriately.

The results above show that both algorithms continued to use the same number of threads even when the throughput levels has changed between the two workloads. The Global Parabola algorithm increased the number of threads slightly were the Hill Climbing kept the number of threads almost constant. The Global Parabola Approximation algorithm continues to show variability in its moves compared to the Hill Climbing algorithm.

7.2.4 TPCC2-BIGMEM to TPCC1-BIGMEM

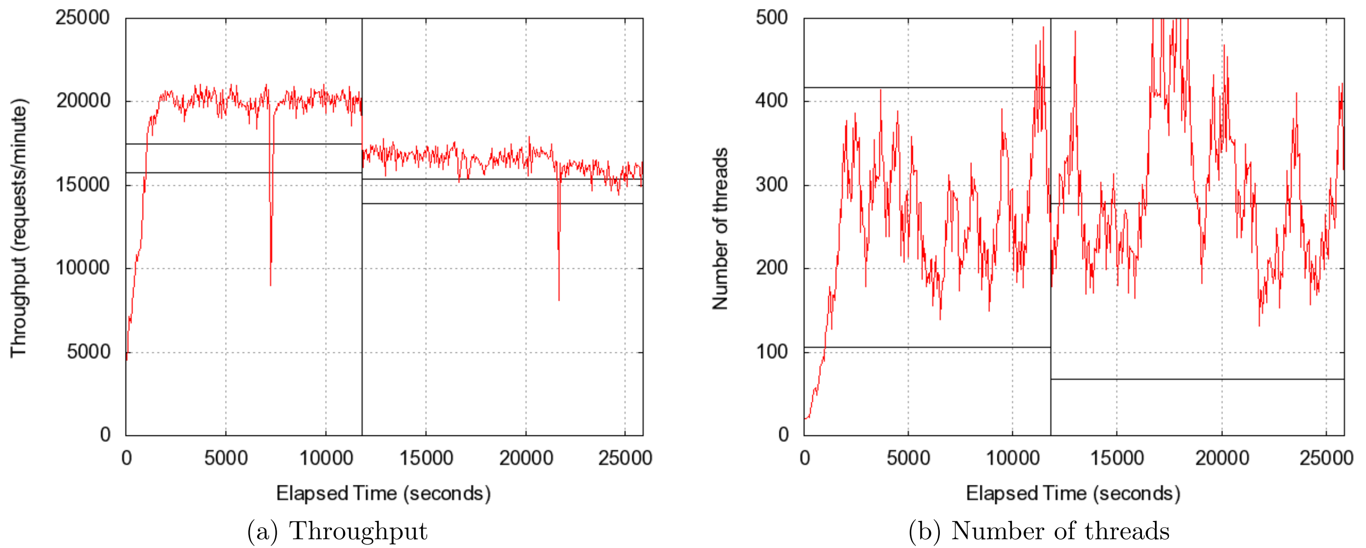


(a) Throughput (b) Number of threads

Figure 7.15: Performance of the Hill Climbing algorithm while switching workloads from TPCC2-BIGMEM to TPCC1-BIGMEM

In this experiment we repeated the same experiment as in the previous section but now going from the TPCC2-BIGMEM workload to the TPCC1-BIGMEM workload. In this scenario, we are going from a workload with an expected higher throughput level to a workload with an expected lower throughput level. The regions of optimal number of threads between the two workloads overlap. Since the region of optimal values are very close, we expect the controller to stay within the same number of threads range. The challenge is that when the workload switch happens, the throughput will drop and the algorithm have to handle the drop properly but kept the number of threads at about the same level.

Figure 7.15 shows how the Hill Climbing handled the workload switch. In the first phase of the experiment, the algorithm took 7,387 seconds to reach the region



(a) Throughput

(b) Number of threads

Figure 7.16: Performance of the Global Parabola Approximation algorithm while switching workloads from TPCC2-BIGMEM to TPCC1-BIGMEM

of optimal settings for the first workload. When the workload switch happened at 12,371 seconds into the run, we can see that the algorithm started to reduce the number of threads until about 17,500 seconds in which it started to recover back from the workload switch. It then started to increase the number of threads and settled on 117 threads. The affect of the drop in number of threads can also been seen from the throughput curve.

Figure 7.16 shows how the Global Parabola Approximation algorithm handled the workload switch. In the first phase of the run, the algorithm was able to reach the optimal number of threads region for the TPCC2-BIGMEM workload. When the workload switch happened 11,889 seconds into the run, the algorithm continued to keep the number of threads almost at the same level. This number of threads is considered on the high region of the TPCC1-BIGMEM workload.

The results above show the two algorithms were able to handle the workload switch properly but the Global Parabola Approximation algorithm had high variability in its choice for the number of threads. The Hill Climbing had smoother transitions between the two workloads.

7.3 Summary

In this chapter we have showed the results of two main sets of experiments. In the first set of experiments, we studied how each of the algorithm performed with stable workloads

In the second set of experiments in Section 7.2, we studied how well each of the

algorithms handled a change in workload conditions. We tried to move from one workload to another and also in the reverse direction to see if there will be any problems in the algorithms adjusting to the change.

Based on all the experiments above, we have found that both of the algorithms have certain pluses and certain minuses. We now summarize our findings about the algorithms separately.

7.3.1 Hill Climbing

We have identified the following advantages for the Hill Climbing algorithm:

1. This algorithm consistently shows that it has less variations in its step taking action. We do not see large swings in the control variable. As a result, both the throughput and number of threads had small fluctuations. This feature might be appealing to system administrator as the algorithm can handle changes more gradually.

2. The degree of aggressiveness can be controlled by adjusting the algorithm variables.

We have identified the following drawbacks for the Hill Climbing algorithm:

1. The algorithm can take a long time to reach the workload optimal values. This is mainly dependent on the step size. If the step size is set small then the algorithm can make finer grained adjustments at the expense of longer convergence times. In contrast, large step sizes allow the algorithm to converge faster.

2. In workloads that have a throughput curve that moderately climbs, the algorithm might choose a value that is too conservative. For example, with TPCC1-SMALLMEM, the algorithm settled at 27 threads with throughput of 3,676.19 requests/minute where the target value was at 250 threads with a throughput of 6,017.90 requests/minute.

3. The starting value for the control variable greatly affects the algorithm's ability to make better control decisions.

7.3.2 Global Parabola Approximation

We have identified the following pluses for the Global Parabola Approximation algorithm:

1. Faster convergence times.

2. The initial starting point of the control variable does not seem to affect the decisions of algorithm.

3. The algorithm can handle workload shifts and does not have problems with changing workload conditions.

We have identified the following drawbacks for the Global Parabola Approximation algorithm:

1. This algorithm was more aggressive in taking large steps in both the positive and negative directions. This caused very large swings in the control variable. In some cases the swings were as large as 143 as in TPCC1-SMALLMEM.

2. This algorithm tends to settle on a larger number of threads compared to the hill climbing algorithm. Using larger number of threads might make the server susceptible to load spikes. This behaviour of the algorithm might make the server less attractive to system administrators who prefer to see a more gradual load transitions.

Chapter 8

Conclusion and Future Work

In this thesis we studied how the multiprogramming level of the database server can affect its throughput. In particular, we looked at four different workload configurations and we studied the shape of the throughput curve for these workloads. We have found in our study that not all workloads exhibit a hill-shaped curve. There are many factors that can influence the shape of the throughput curve. Some of those factors are related to hardware resources, others are related to software resources. Hardware resources include CPU, disk I/O, or network capacity. Software resources include contention on data structures, memory accesses, or the database server software itself. Every workload has an optimal multiprogramming setting that can achieve maximum throughput.

The optimal multiprogramming level setting for each workload is difficult to predict. In this thesis, we developed an online controller that attempts to automatically adjust the multiprogramming level of the database server. Our primary goal is to improve the throughput level of the server and our secondary goal is to choose the minimum number of threads that can maximize throughput.

As part of our controller design, we have developed and studied three different algorithms that the controller can use. These algorithms were: Hill Climbing, Global Parabola Approximation, and Local Parabola Approximation. Some of these algorithms were already suggested by previous research.

For the first algorithm, we have built into it a mechanism to make it biased towards minimizing the number of threads while trying to maximize throughput. This mechanism allow the Hill Climbing algorithm to handle throughput curves that are flat or that do not resemble a hill-shaped curve.

The second algorithm tries to model the throughput curve using a parabolic function that passes through the origin. Because the parabolic function is used to model only the first half of a hill-shaped throughput curve, it will automatically be biased towards the first part of the hill and will try to minimize the number of threads.

The third algorithm also uses a parabolic function but does not assume that

it passes through the origin. In a similar manner, the parabolic function approximation will try to minimize the number of threads in order to handle throughput curves that do not resemble a hill-shaped one. We have found that this algorithm is affected greatly by measurement noise and hence we decided to stop from studying it any further.

Both the Hill Climbing and the Global Parabola approximation algorithms did well at adjusting the multiprogramming level. The Hill Climbing algorithm excelled in that it was taking more gradual steps in adjusting the multiprogramming level. There were less fluctuations and variations in both the throughput and the number of threads. The shortcoming of this algorithm is that it is more conservative in terms of steps. This made the algorithm take more time to reach the target values. One issue with this algorithm is that it seems to get affected by the starting value of the control variable.

The Global Parabola algorithm was more aggressive in changing the number of threads. There were more fluctuations in both throughput and number of threads. This algorithm on most cases settled on a larger multiprogramming level compared to the Hill Climbing algorithm.

Although each algorithm has its advantages and disadvantages, we found the Hill Climbing to be more desirable in a production system as it exhibits more stability and less fluctuations. The only issue that needs to be addressed is the convergence time of this algorithm and the starting value issue.

The results presented in this thesis show that it is feasible to design an automatic tuning algorithm to improve server throughput especially in situations were the workload can change. In our implementation, we relied on throughput alone as a guide to the auto-tuning controller. We have observed that such metric was very effective in that the auto-tuning algorithms were able to reach our target values.

In the future, we plan to investigate ways to improve on the convergence time of the Hill Climbing algorithm. There is always room to develop other controller algorithms that depend on other metrics or combination of metrics.

Lastly, we plan to investigate other workloads to see how well these and other algorithms can perform. There is a chance that other workloads might show other shortcomings or opportunities of these algorithms.

Bibliography

[1] Dell DVD Store 2 application. Available online at http://linux.dell.com/dvdstore.

[2] Microsoft SQL Server. Available online at http://www.microsoft.com/sql.

[3] Microsoft SQL Server: How to determine proper SQL Server configuration settings. Available online at http://support.microsoft.com/kb/319942.

[4] Microsoft SQL Server: Tips for Performance Tuning SQL Server's Configuration Settings. Available online at http://www.sql-server-performance.com/sql_server_configuration_settings.asp.

[5] MySQL. Available online at http://www.MySQL.com.

[6] Oracle. Available online at http://www.oracle.com.

[7] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 181–192, New York, NY, USA, 1999. ACM Press.

[8] T. Brecht, D. Pariage, and L. Gammo. accept()able strategies for improving web server performance. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[9] Kurt P. Brown, Manish Mehta, Michael J. Carey, and Miron Livny. Towards automated performance tuning for complex workloads. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 72–84, Santiago, Chile, 1994.

[10] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[11] Shiping Chen and Ian Gorton. A predictive performance model to evaluate the contention cost in application servers. In *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, page 435, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Apache Software Foundation. Available online at http://www.apache.org.

[13] Stavros Harizopoulos. *Staged Database Systems*. PhD thesis, Carnegie Mellon University, 2005.

[14] Hans-Ulrich Heiss and Roger Wagner. Adaptive load control in transaction processing systems. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 47–54, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[15] iAnywhere Solutions. *SQL Anywhere Server Database Administration: Setting the database server's multiprogramming level*.

[16] iAnywhere Solutions. *SQL Anywhere Server User's Guide: Using the cache to improve performance - Dynamic cache sizing*.

[17] iAnywhere Solutions: SQL Anywhere Server. Available online at http://www.ianywhere.com/sqlanywhere.

[18] J.D.C. Little. A proof of the queuing formula $l = \lambda w$. In *Operations Research*, pages 383–387, 1961.

[19] Xue Liu, Lui Sha, Yixin Diao, Steve Froehlich, Joseph L. Hellerstein, and Sujay S. Parekh. Online Response Time Optimization of Apache Web Server. In *Proceedings of the 11th International Workshop on Quality of Service*, pages 461–478, 2003.

[20] Axel Mönkeberg and Gerhard Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 432–443, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[21] J. Ousterhout. Why threads are a bad idea (for most purposes). *Presentation given at the 1996 Usenix Annual Technical Conference, January*, 1996.

[22] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, Erich Nahum, and Adam Wierman. How to determine a good multi-programming level for external scheduling. In *Proceedings of the 22nd International Conference on Data Engineering*, page 60, Washington, DC, USA, 2006. IEEE Computer Society.

[23] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: a cautionary tale. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[24] J. Teng. Goal-oriented dynamic buffer pool management for data base systems. In *ICECCS '95: Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*, page 191, Washington, DC, USA, 1995. IEEE Computer Society.

[25] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification. Available online at http://www.tpc.org/tpcc.

[26] Transaction Processing Performance Council. TPC Benchmark H, Standard Specification. Available online at http://www.tpc.org/tpch.

[27] Gerhard Weikum, Christof Hasse, Axel Mönkeberg, and Peter Zabback. The COMFORT automatic tuning project. *Information Systems*, 19(5):381–432, 1994.

[28] Gerhard Weikum, Axel Mönkeberg, Christof Hasse, and Peter Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 20–31, 2002.

[29] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 287–296, New York, NY, USA, 2004. ACM Press.