ABSTRACT

Title of thesis: EXTENSIONS OF AN EMPIRICAL AUTOMATED TUNING FRAMEWORK

Yifan Zhou, Master of Science, 2013

Thesis directed by: Professor Jeffery K. Hollingsworth
Department of Computer Science

Empirical auto-tuning has been successfully applied to scientific computing applications and web-based cluster servers over the last few years. However, few studies are focused on applying this method on optimizing the performance of database systems. In this thesis, we present a strategy that uses Active Harmony, an empirical automated tuning framework to optimize the throughput of PostgreSQL server by tuning its settings such as memory and buffer sizes. We used Nelder-Mead simplex method as the search engine, and we showed how our strategy performs compared to the hand-tuned and default results.

Another part of this thesis focuses on using data from prior runs of auto-tuning. Prior data has been proved to be useful in many cases, such as modeling the search space or finding a good starting point for hill-climbing. We present several methods that were developed to manage the prior data in Active Harmony. Our intention was to provide tuners a complete set of information for their tuning tasks.

EXTENSIONS OF AN EMPIRICAL AUTOMATED TUNING FRAMEWORK

by

Yifan Zhou

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2013

Advisory Committee:
Professor Jeffery K. Hollingsworth, Chair/Advisor
Professor Joseph F. JaJa
Professor Alan L. Sussman

© Copyright by
Yifan Zhou
2013

Dedication

To my parents and friends

Acknowledgments

Firstly, I would like to thank my advisor, Dr. Jeffery K. Hollingsworth. This work would not be done without his patient guidance and never-ending support.

I also would like to thank the members of my defense committee, Dr. Alan L. Sussman and Dr. Joseph JaJa for their discussions and suggestions on my thesis work.

I had a great time with my group mates in the Paradyn research group at UMD. I would like to thank Ray Chen, for his support on the Active Harmony framework. Philip Yang, who expanded my knowledge and inspired me a lot on my work. Tugrul Ince and Mike Lam, who provided a lot of information on how to survive the graduate school. I also enjoyed the time when discussing auto-tuning with Sukhyun Song and Chester Lam.

I would like to acknowledge helps and supports from Tiantao Lu, John Holland and Kelly Emery, who reviewed my thesis paper and provided useful suggestions. Rongjian Lan, who encouraged me when I was depressed.

At last, I owe my deepest thank to my parents, whose helps are invaluable.

Table of Contents

List of Tables vi

List of Figures vii

1 Introduction 1
1.1 Overview . . . 1
1.2 Contributions of the Thesis . . . 2
1.2.1 Prior Data Logging for Active Harmony . . . 3
1.2.2 Online Performance Tuning of Database Systems . . . 3

2 Related Work 5
2.1 Automated Tuning Tools and Usage . . . 5
2.1.1 Empirical Tuning . . . 5
2.1.2 Model-based Tuning . . . 6
2.1.3 Predictive Tuning . . . 7
2.2 Database Self-Tuning Techniques . . . 7

3 Active Harmony 9
3.1 Introduction . . . 9
3.2 Architecture of a Active Harmony . . . 9
3.2.1 Harmony APIs . . . 9
3.2.2 Harmony Server . . . 13
3.2.3 Session Core . . . 13
3.3 Plug-ins of Active Harmony . . . 16
3.4 Tuning Algorithms . . . 17
3.4.1 Random and Brute Force Exhaustive Method . . . 18
3.4.2 Nelder-Mead Simplex Method . . . 18
3.4.3 Parallel Rank Ordering (PRO) . . . 19
3.5 Performance Data Management . . . 21

4 Prior Data Management 24
4.1 Data Logging in Active Harmony 24
4.2 Text Log 24
4.3 XML 25
4.4 TAUdb 28
4.4.1 TAU 28
4.4.2 Schema of TAUdb 29
4.4.3 Visualization of Performance Data 30
4.5 Discussion 31

5 Tuning Database Systems 33
5.1 Motivation 33
5.2 Problem Statement 35
5.3 Platforms 35
5.3.1 PostgreSQL 35
5.3.1.1 Postgres Offline Parameters 35
5.3.1.2 Postgres Online Parameters 36
5.3.2 TPC-H Benchmark 36
5.3.3 Environment 37
5.4 Methodologies 37
5.4.1 Performance Measurement 37
5.4.2 Workload 40
5.4.3 Tuning Parameters 41
5.4.4 Search Space 42
5.4.5 Search Strategy 45
5.5 Online-Offline Co-tuning 45
5.5.1 One-pass Tuning Flow 46
5.5.2 Multi-pass Tuning Flow 47
5.5.3 Scalability 47
5.6 Memory Budget Constraints 49
5.7 Experimental Results 52
5.7.1 Search Space versus Search Ability 52
5.7.2 Impact of Memory Budget Constraints 55
5.7.3 Workload Shifting 57

6 Future Work 60

7 Conclusions 62

Bibliography 64

List of Tables

4.1 Harmony API in Interaction with Logging Plug-ins 25

5.1 Ranking of configuration parameters for a workload of TPC-H queries 40
5.2 PostgreSQL parameters chosen for performance tuning 41

List of Figures

3.1 Active Harmony Automated-tuning System 10
3.2 Harmony Client API Calls 12
3.3 Role of Harmony Server 14
3.4 Architecture of Harmony Session Core 15
3.5 Mechanism of Harmony plug-ins 16
3.6 Illustration of Nelder-Mead Simpex Method 20
3.7 Parallel Rank Ordering Method 22

4.1 An Example of Harmony's XML Output 27
4.2 Data Structures of TAU 29
4.3 Example of ParaProf Tool on an MPI Program 31
4.4 ParaProf's Visualization on One Node in an MPI Program 32

5.1 The Sampling Problem of PostgreSQL 39
5.2 Throughput vs. work_mem on a Workload with 12Q7 42
5.3 Throughput vs. shared_buffers on 2 Different Workloads 43
5.4 Throughput vs. work_mem on Compound Workload 43
5.5 Throughput vs. random_page_cost on Workload with 12Q6 44
5.6 Throughput vs. random_page_cost on Workload with 3Q6, 3Q7, 3Q8, and 3Q13 44
5.7 One-pass Tuning Flow 46
5.8 Multi-pass Tuning Flow 48
5.9 Avoiding infeasible data points by applying penalty factor 52
5.10 Tuning Curve for 4Q6, 4Q9 and 4Q13 53
5.11 Tuning Result for Multiple Workloads 54
5.12 Tuning Curve for 3Q6, 3Q7, 3Q8 and 3Q13 55
5.13 Searching Result by the Nelder-Mead vs. Budget Constraint for On-line Parameters 56
5.14 Workload Shift from One Workload to Another 57
5.15 Shared_buffers Shift Corresponding to Workload Shift 58
5.16 Difference Between Shapes of Shared_buffers 59

Chapter 1: Introduction

1.1 Overview

Automated tuning has already been successfully used in multiple research areas, especially in the field of scientific computing. In today's software applications, programs are no longer written for a specific hardware platform, and libraries or re-usable components are widely shared by a huge number of users. However, applications may not be well optimized on a specific platform due to the differences among hardware architectures, operating systems strategies and other configuration specific details.

Tuning the parameters of applications on a specific platform for optimal performance is a non-trivial problem, and it is hard to obtain a specific performance model for the target platform because: 1. The runtime characteristics of low level systems and the high level applications are difficult to model; 2. It takes a lot of effort to maintain the existing performance models for a specific application. Moreover, due to the large search spaces that are common in scientific computing tasks, finding the global optimal performance point is an NP-hard problem.

Active Harmony is an automated online performance tuning framework that improves the runtime performance of various types of applications. Previous uses

of Active Harmony mainly focused on compiler optimization, such as tuning loop tiling and loop unrolling factors. However, few research topics focused on other areas. One of the goals of this thesis is to show that Active Harmony can also be applied to tuning database servers, and that it is able to improve the throughput of database systems at runtime.

In addition, there is a growing demand for Active Harmony to have a management system that stores performance data from tuning tasks. During the tuning process of an application, empirical performance data were usually considered redundant and thrown away after they are tested. However, with the increasing complexity of target applications, researchers realized that the re-usability of prior data becomes increasingly important, and recent studies [14] [3] show that prior data are useful in capturing the search space of the target problem. Therefore, we present several methods that are developed recently to store and manage prior data in Active Harmony.

Our work can be summarized as follows: Introducing new features into Active Harmony and showing that database servers perform better after tuned by Active Harmony.

1.2 Contributions of the Thesis

The main contributions of this thesis are:

1. Developed new methods used by Active Harmony to manage prior data.
2. Proposed an empirical approach to tune the throughput of database systems

using Active Harmony.

1.2.1 Prior Data Logging for Active Harmony

Learning from prior data has become a popular topic among auto-tuning tools developers in recent years. Prior data have been usually used by researchers to build performance models using machine learning algorithms. The benefits of doing this include 1. Speeding up the tuning process by evaluating the performance based on the model, rather than frequently doing empirical tests, and 2. Finding a better starting point for future empirical tuning tasks. Previous work on empirical auto-tuning often focused on the scalability of the framework and the tuning ability of the search engine. However, with the target application being more complicated, we foresee that the online empirical method would produce much overhead and be slower to converge. Therefore, methods to manage prior data are required to meet the future challenges.

In this thesis, we present three formats supported by Active Harmony to record prior data: text files, XML, and TAUdb.

1.2.2 Online Performance Tuning of Database Systems

Database systems include many parameters that need to be tuned for a specific database workload in order to get the optimal performance. These parameters can be categorized into two classes: 1. Parameters that define the sizes for each type of memory buffers; 2. Parameters that estimate the status of the underlying systems, in

order to affect the strategies used by the query planner. Tuning the performance of a database is a non-trivial problem because: 1. The performance highly depends on the type of workloads. For example, some queries consume more sorting buffers than others, whereas other queries may consume more maintenance buffers. Therefore, it is important to decide how much of each type of memory buffers should be allocated for different workloads, especially when there is a memory resource limitation. 2. All possible configurations form an N-dimensional tuning space, and finding the optimal configuration is an NP-hard problem without prior knowledge.

The rest of this document is organized as follows: Chapter 2 introduces the related work; Chapter 3 introduces Active Harmony, the empirical automated tuning framework that our studies were based on; Chapter 4 introduces the prior data management plug-ins in Active Harmony; and Chapter 5 discusses how we use Active Harmony to tune a database server;

Chapter 2: Related Work

2.1 Automated Tuning Tools and Usage

2.1.1 Empirical Tuning

Empirical-based auto-tuning optimizes the target applications by using an empirical feedback loop. Empirical-based tuning is a technique that provides accurate performance evaluation. However, the tuning speed is lower than other techniques because it measures performance empirically.

Active Harmony [10] is an empirical auto-tuning framework that has been used to tune cluster-based web services, and static/dynamic parameters for multiple applications.

In addition to Active Harmony, Orio [18] is another empirical tuning tool developed by Argonne National Laboratory. Orio uses an empirical feedback loop, a search engine and code transformer to optimize the performance of GPU applications.

Performance tuning of applications that requires variations of the original programs relies on runtime code generation. CHiLL [5] contains a polyhedral loop transformation and code generation framework. CHiLL is very suitable for auto-tuning

frameworks as it mathematically represents iteration spaces and has a flexible script interface for code generation.

Empirical auto-tuning has been widely used to tune applications in various domains. I-hsin et al. [8] used Active Harmony to tune the performance of cluster-based web services. They showed that their empirical approach improved the WIPS(Web Interactions Per Second) of TPC-W benchmark on a simple cluster for 5-16%. It was also used to tune the performance-related parameters of GS2 [7], SMG2000 [28], and others.

2.1.2 Model-based Tuning

Model-based auto-tuning relies on the simulator developed for a specific application on a specific platform. Model-based methods can be useful in well-organized hardware systems such as memory systems or graphic processing units in which there exist a huge number of identical components. Choi et al. [6] proposed a performance model for GPUs and applied this model to tuning the performance of sparse matrix-vector multiplication. However, they only verified their GPU performance model on a few applications.

Model-based tuning achieves good accuracy in performance evaluation. However, such methods are less useful in more complex applications or platforms, such as CISC processors, due to the difficulties in developing and maintaining the models.

2.1.3 Predictive Tuning

Predictive tuning technique uses machine learning algorithms to build a statistical performance model based on empirical performance data. Predictive methods adopt the efficiency of model-driven tuning techniques in performance measurement, but the accuracy of this method depends on how empirical data are sampled.

Ganapathi et al. [14] developed a kernel canonical correlation analysis [13] method to predict the performance of a database query before its execution, so that the prediction can be further used in auto-tuning the performance of data centers. Bergstra et al. [3] proposed a machine learning method to auto-tune GPU programs and to decide which features are passed to the learning model. They also used Boosted Regression Trees to predict how faster a kernel is compared to a reference baseline.

2.2 Database Self-Tuning Techniques

The problem of tuning database parameters has been discussed over decades in the field of database auto-administration.

Storm et al. [25] proposed a tool called Self-Tuning Memory Manager (STMM), which is able to dynamically allocate memory for a given workload. They assumed that each memory consumer can be regarded as an independent variable to others, and its size versus its impact on performance can be modeled as a simple exponential function. However, they have not considered tuning parameters other than memory buffers. This thesis shows that decision-related parameters may fail to fit in their

proposed model.

Duan et al. [12] approached the problem by modeling the search space using adaptive sampling with Gaussian process. Their model is able to fit into a more complex search space compared to the exponential function model. However, since the number of samples increases with the number of tuning parameters, there is an extra overhead in building and verifying the model. Besides, their optimization is based on standby databases and is less sensitive to a changing workload.

Schnaitter et al. [22] proposed COLT, an online tuning framework that continuously analyzes the data access structure and configures the settings based on previous statistics.

Nguyen et al. [29] developed a new approach in tuning the buffer sizes based on the buffer miss equation. The equation identifies a buffer size limit that is useful for buffer tuning and powering down idle buffers. They use this equation to simulate and predict the I/O costs.

Other researchers tried to optimize database performance from other directions. For example, Chaudhuri et. al [4] and Agrawal et al. [1] both discussed approaching the problem by automated physical database design.

Chapter 3: Active Harmony

3.1 Introduction

We first introduce Active Harmony. Active Harmony is an empirical automated tuning framework developed for tuning the runtime performance of applications [10].

3.2 Architecture of a Active Harmony

Active Harmony uses a client/server architecture. The client is the target application that needs to be tuned, and the server manages the tuning task and executes search heuristics. Active Harmony consists of Harmony client API, Harmony server, Harmony session core, command line tools such as TUNA and other plug-ins, such as code generator and XML Writer. Figure 3.1 shows a system that uses Active Harmony to tune the performance of target applications.

3.2.1 Harmony APIs

The Harmony client API provides functions that the applications (the client code) call to interact with Harmony server. The client code first describes the

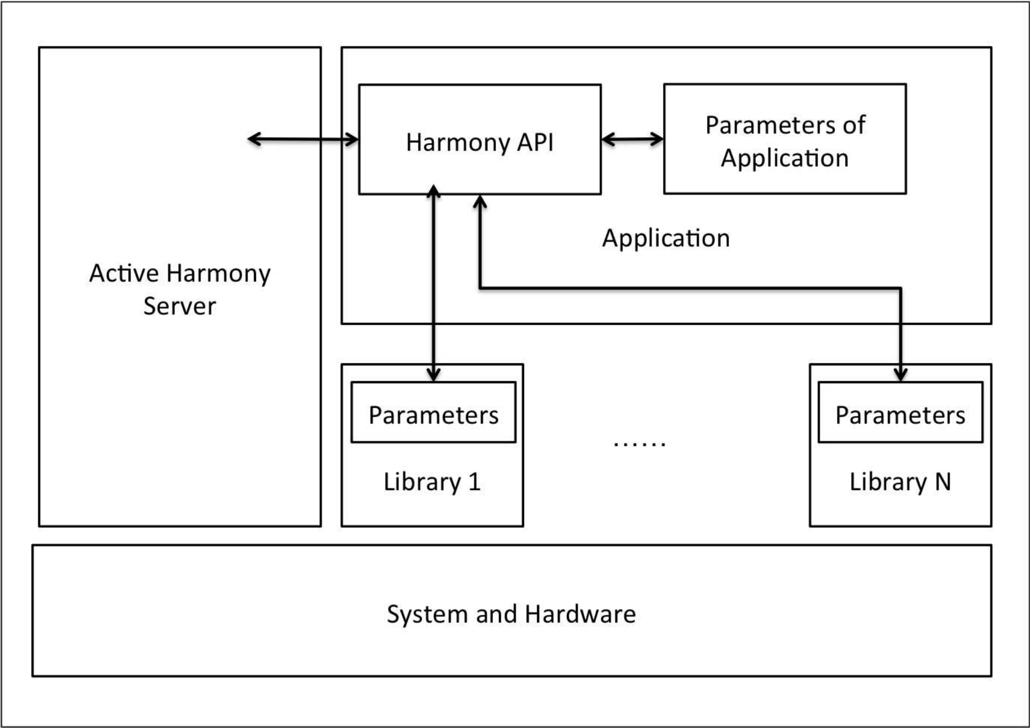


Figure 3.1: Active Harmony Automated-tuning System

problem as a list of constraints and value strides to represent the search space of parameters to the Harmony server, then interacts with it and forms an empirical feedback loop in order to optimize its runtime performance. An example of the usage of Harmony client API is shown in the code snippet in Figure 3.2. At each harmony_fetch() call, the server sends the client a new point in the search space that has not been tested before, and it gets a reply from the client about the performance measurement of that point when the client code calls harmony_report(). Active Harmony has multiple heuristics (search strategies) to generate the next testing points. More details about the search strategies are described in Section 3.4.

Active Harmony has been re-factored to provide a modular architecture. By introducing the concept of "session," a single tuning task is described in the Harmony server as a Harmony session, and the server is able to handle multiple sessions at the same time. This feature not only allows running multiple tuning tasks with one server simultaneously, but it also encourages users to divide unrelated parameters within a single application into several parameter sets in order to speed up the tuning process.

To tune an application using Active Harmony, the original code has to be modified to enable communications with the Harmony server.

The code snippet in Figure 3.2 also points out the difference between a Harmony session and a Harmony client. Harmony session can be regarded as a tuning task handler on the server side. A session core has to be generated by a Harmony client, but it might not only serve one single client. However, a Harmony client can be regarded as a "customer" for a Harmony session. It is the only one served by the

```
/*Initialization of harmony tasks*/
hsession_init();
/* Configure the Harmony session,
 * inducing the search strategy and the intended plug-ins*/
hsession_cfg(&session, CFG_NAME, CFG_VALUE);
/* Register a new tuning parameter to the Harmony session,
 * the below line defines an integer variable */
hsession_int(&session, "name", min, max, step);
/* Launch a new session core on the server side */
hsession_launch(&session);
/* Initialize and allocate a Harmony descriptor on the client side */
harmony_init()
/* Bind the session variables to local variables */
harmony_bind_int(hdesc, "param_name", &param_name);
/* Connect the current client to the session */
harmony_join(hdesc, NULL, 0, name);
/* Inform the server of new configurations */
harmony_inform(hdesc, "CFG_NAME", CFG_VALUE);
/* Fetch a new configuration from the server,
 * and change the local parameters to new values for performance testing */
harmony_fetch(hdesc);
/* Report the performance value of the newly fetched
 * configuration back to the server*/
harmony_report(hdesc, perf_val);
/* Finalization */
harmony_fini(hdesc);
```

Figure 3.2: An example of using Harmony Client API calls

session in serial programs, but in parallel program auto-tuning, each client can be a process or a thread of the program, and each process or thread is co-working with other processes or threads under the same Harmony session.

3.2.2 Harmony Server

In the latest Active Harmony version, the Harmony server simply launches session cores upon client requests and is responsible for passing messages between the clients and the session cores. The client code includes the session ID when fetching a data point or reporting a performance value in the message that it sends to the Harmony server, and the server forwards the message using the specified session ID to the corresponding session core.

3.2.3 Session Core

A session core is an instance of a Harmony session launched by the Harmony server when the client code calls *hsession_init*(). The session core holds the information of the application, such as application name, parameter information, the search strategy and other configuration data customized by the clients. A session core only communicates with the Harmony server. It receives the client requests forwarded by the Harmony server, and it also sends the newly generated points to the Harmony server, which will later be forwarded to the client code.

Active Harmony also provides a plug-in interface for each session core. Users of Active Harmony can develop their own plug-in routines that will be triggered

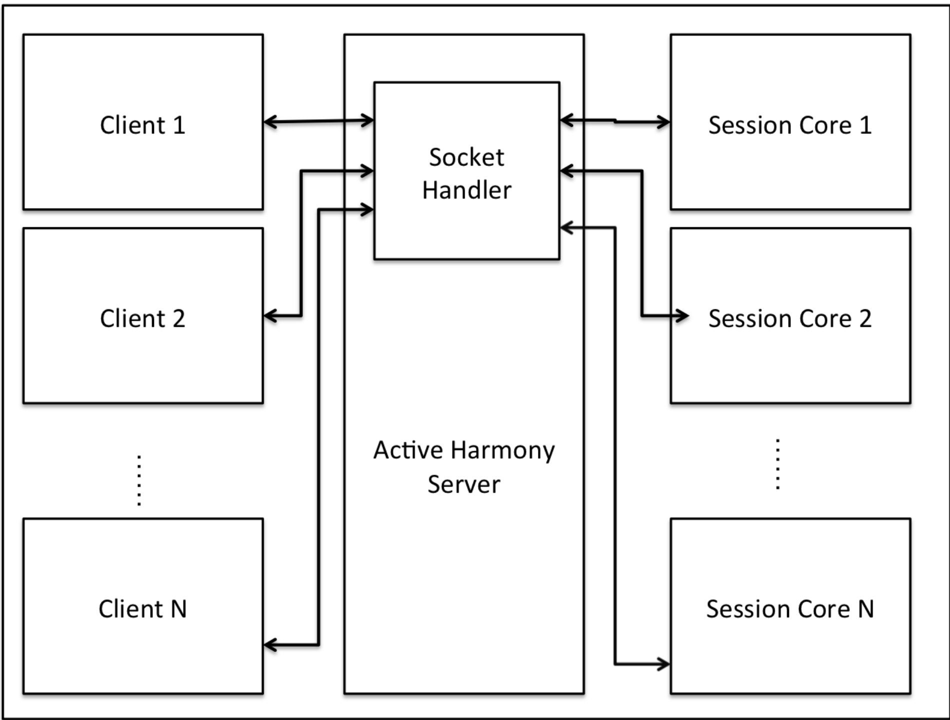


Figure 3.3: Role of Harmony server

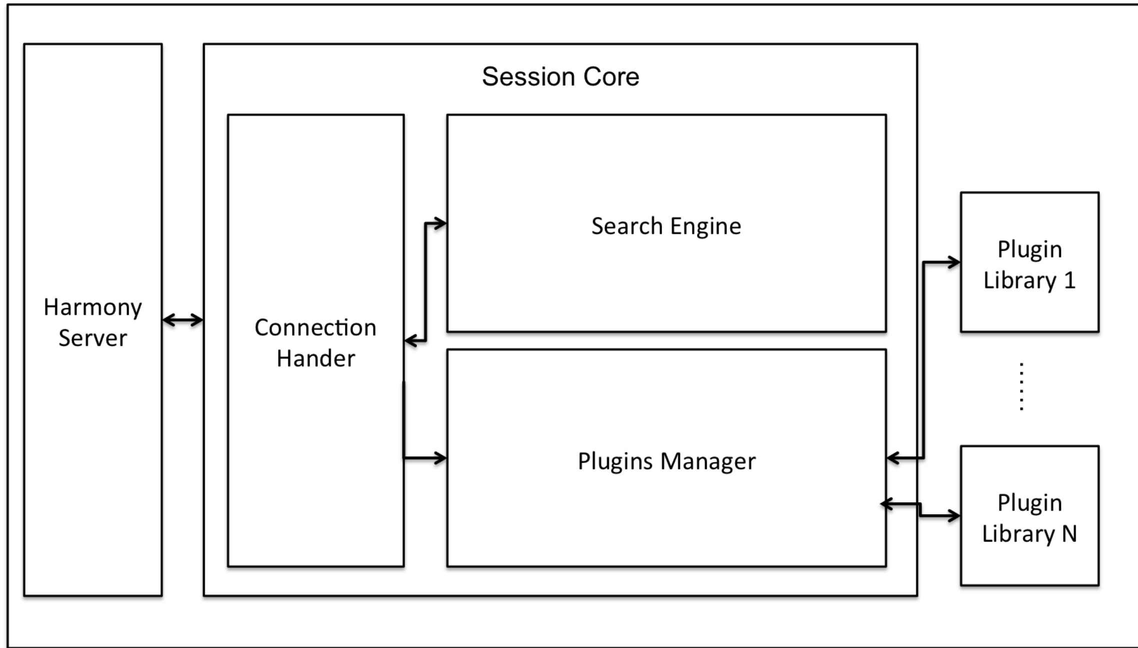


Figure 3.4: Architecture of Harmony Session Core

by the session core if there is a corresponding Harmony API function call from the client code. More details about Harmony plug-ins are described in Section 3.3.

Figure 3.4 illustrates the architecture of a session core. The session core receives the client's request from Harmony server. If the client calls $harmony_fetch()$, it finds an untested configuration using the search heuristics, and if the clients calls $harmony_report()$, it reads the empirical performance measurements of previous configurations and applies them to the search engine to generate a new testing point.

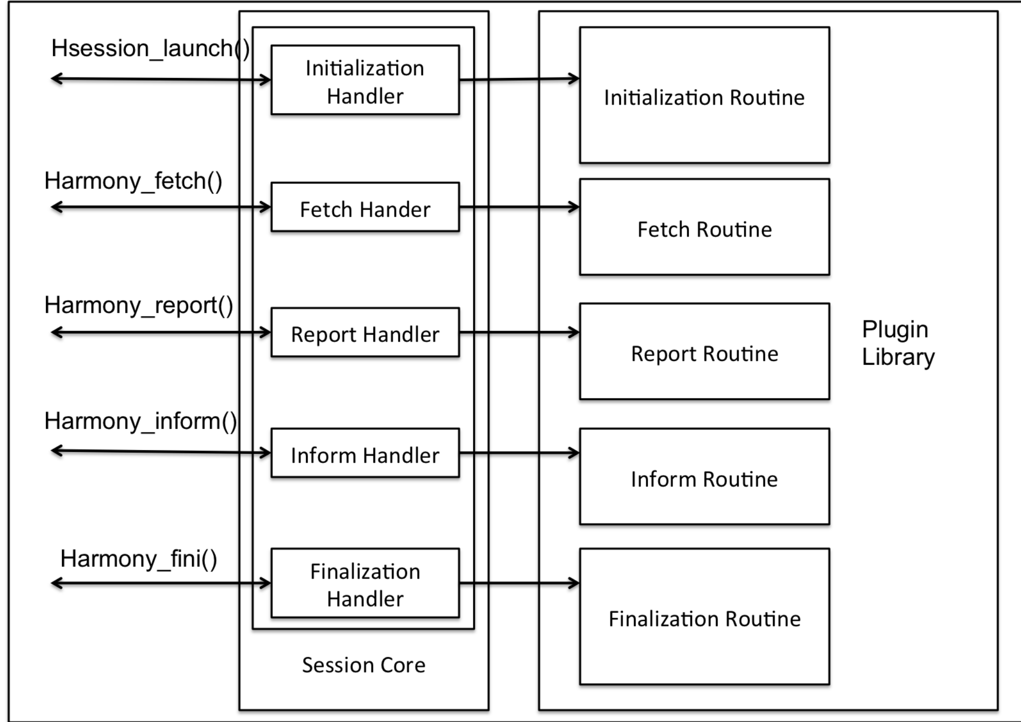


Figure 3.5: Mechanism of Harmony plug-ins

3.3 Plug-ins of Active Harmony

As mentioned in Section 3.2.3, the latest Active Harmony allows users to develop their own plug-ins to extend the functionality of this auto-tuning framework. Since the plug-ins are launched by the session core, they are commonly used to manipulate the tuning information in the session core. A typical Harmony plug-in is a code generator used to re-generate the code snippet based on new parameters (e.g. loop tiling or loop unrolling factors). A plug-in can also be a user-defined search strategy if any domain-specific tuning strategies are appropriate.

Figure 3.5 presents how each plug-in routine is triggered by the corresponding

Harmony client API function call from the client code. Users can develop plug-ins that optimize the generated points, store the performance measurements in their own formats or build models based on performance data that have already been tested.

3.4 Tuning Algorithms

An efficient tuning algorithm is the key to reaching an optimal point in the search space, especially for empirical-based auto-tuning. The reason is that measuring the performance of one point has more overhead than other tuning techniques.

Finding a good set of configurations is equivalent to searching for a k-tuple in the entire search space defined by the value space of the parameters in the target application. Optimization in an N-dimensional space is an application dependent problem and has been studied for decades. Many heuristics have been developed to prevent the search results from falling into a local minimum area. However, for runtime auto-tuning tools, since the tuning speed is also a very important metric to evaluate the performance of a search strategy, local optimization algorithms are usually chosen for this purpose.

Active Harmony is an experiment-based tool which treats the objective function as a black-box, and it only relies on the function output to carry out its tuning process. A good search strategy for auto-tuning should consider both tuning speed and ability of convergence.

Four search strategies are provided with the Active Harmony release, including

random search, brute force exhaustive method, Nelder-Mead simplex method, and parallel rank ordering method.

3.4.1 Random and Brute Force Exhaustive Method

Active Harmony provides a brute force and a random method as simple tuning strategies. Neither of them is commonly used in tuning the performance of real applications, but they are regarded as auxiliary methods for users to better understand the search space.

The brute force method is commonly used to rebuild a small search space with grid sampling and to judge whether a smarter tuning strategy could be applied to such a search space. The random method is often used to make comparison with more complex strategies in terms of optimized performance value and the number of steps they take to converge.

3.4.2 Nelder-Mead Simplex Method

The Nelder-Mead simplex method [20] is a commonly used nonlinear optimization technique. It is suitable for either continuous spaces or discrete spaces. Therefore, it can be applied to a wider range of optimization problems compared to gradient methods.

The Nelder-Mead method is based on the concept of a simplex, which is a polytope of $N+1$ vertices in a N-dimensional search space. Each vertex in the simplex is assigned a value which is generated by the objective function. During the tuning

process, the simplex is repeatedly transformed by different operations. At each operation, it generates a new vertex that replaces an old one, in the hope of finding a better configuration. The operations include reflection, expansion, contraction and reduction. Figure 3.6 illustrates these operations.

The method first orders the values of each vertex in the initial simplex to find the point with the worst value. An N-dimensional centroid point is calculated based on the remaining vertices and the worst vertex is reflected with respect to the centroid. If the new point achieves the best performance among all points in the new simplex, it is further expanded with respect to the centroid. If the reflected point is the worst, the simplex is contracted and the area it covers is smaller. If the contracted vertex is the worst, all vertices except the best one are shrunk with respect to the best point.

The Nelder-Mead simplex algorithm is designed for local optimization, and the search results highly depend on the initial simplex. Therefore, methods that can form good initial simplexes are required in the development of Active Harmony.

3.4.3 Parallel Rank Ordering (PRO)

Parallel rank ordering is derived from the Nelder-Mead simplex method and is used to optimize an objective function in parallel. Similar to the Nelder-Mead method, the PRO algorithm also uses the concept of a simplex and its operations consist of reflection, expansion, and shrinking. The main difference between the Nelder-Mead and the PRO method is that the Nelder-Mead is moving only one point

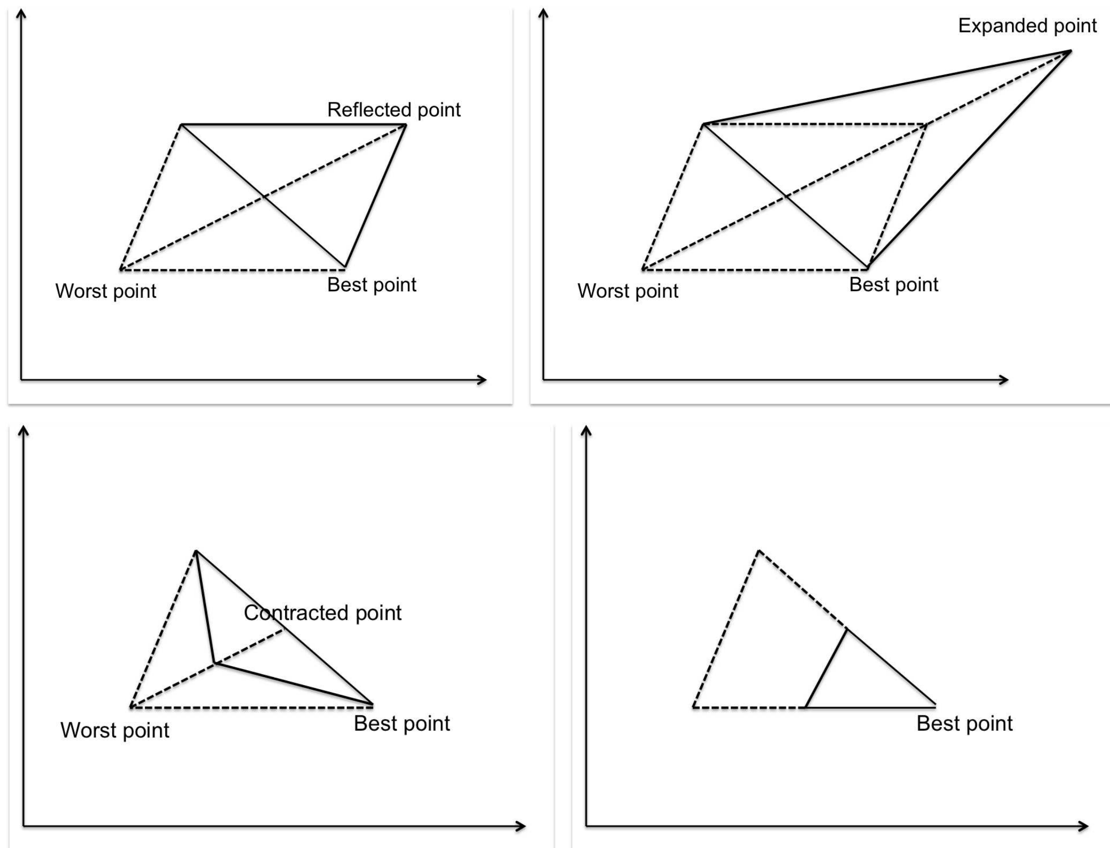


Figure 3.6: Nelder-Mead simpex method. The upper-left figure shows how the worst point is reflected to form a new simplex. The upper right shows how the reflected simplex is expanded if the new point is found to be the best. The lower left figure shows how the worst point contract if the reflection step fails. And the lower right figure shows how simplex reforms if the contraction step fails.

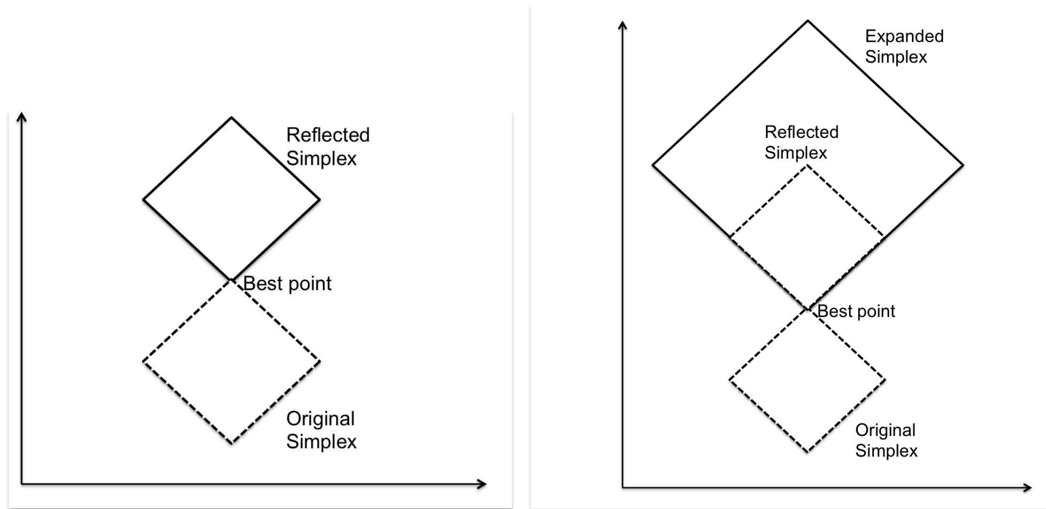
at each step, and the PRO is moving all points except the best point simultaneously. Therefore, the PRO method has the ability of utilizing multiple processes or threads when tuning parallel programs to exploit the search space more efficiently.

Another difference between the Nelder-Mead and the PRO is that the simplex in the PRO method can consist of either $N + 1$ or $k * N$ points ($k > 1$), whereas the simplex in the Nelder-Mead method only consists of $N + 1$ points. Although the simplex of the Nelder-Mead method can be made of more points, it only tests one point at a time, so having more vertices in the simplex doesn't help much in speeding up the tuning process. However, since the PRO method is a technique designed for parallel optimization problems, all points in a simplex can be tested simultaneously. Therefore, if there are enough process or thread resources, PRO's search ability can be improved by adding more points to the simplex and it only takes little extra overhead to maintain a bigger simplex. Figure 3.7 illustrates the transformation operations of the PRO method.

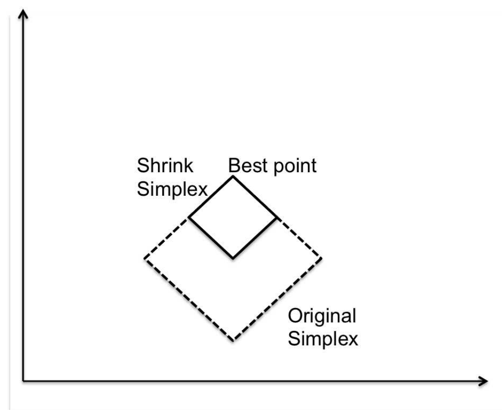
For both the Nelder-Mead and the PRO methods, it is important to keep the simplex an N-polytope to maintain their search ability. When the simplexes of them can be represented using a lower-dimensional space, they lose the ability to exploit in certain directions.

3.5 Performance Data Management

Active Harmony uses a data logging method that appends the performance data of the current run to a text file that contains configuration vectors and per-



(a) Simplex reflect (b) Simplex expand



(c) Simplex shrink

Figure 3.7: PRO transformations: (a) Reflect all the point except the best one with respect to the best point. The reflection succeeds when the point reflected by the worst one is performing better than the worst one. If it performance better than the best point, it triggers the expansion operation; (b) Expand the simplex; (c) Shrink the simplex when the reflection fails

formance measurements. However, the data in such text files are hard to analyze because they are lack of organized structures and useful metadata, such as machine and operating system configuration information.

To address the logging problem, we provide two new methods to store and manage prior data, including storing the data in XML format, and managing performance data using TAUdb [17]. We believe that these methods are able to provide users a complete set of information about the tuning tasks. More details about prior data management are provided in Chapter 4.

Chapter 4: Prior Data Management

Empirical-based tuning is a time-consuming process and referring to prior data is an effective way to speedup the time spent on performance measurement. In this chapter, we present several methods used by Active Harmony to log and manage prior data.

4.1 Data Logging in Active Harmony

In Chapter 3, we introduced the plug-in work flow of Active Harmony and showed that a plug-in routine can be triggered by a specific Harmony API function call from the client code. In order to record a complete set of information about the applications tuned by Active Harmony, we developed plug-ins that made use of such triggers to record metadata and performance data. Table 4.1 shows how each Harmony API function call triggers a specific Active Harmony plug-in routine.

4.2 Text Log

Text Log has already been developed in the early versions of Active Harmony. Using a text file is a basic approach of saving configuration values and the corresponding performance measurements. The Harmony server saves all the per-

Table 4.1: Harmony API in interaction with logging plug-ins

Harmony API Function	Function
hsession_launch	Triggers the initialization of the target format, such as creation of logging file
harmony_inform	This call can be used to inform the server about the metadata on the client side
harmony_report	Triggers the logging system to record the configuration-performance pair
harmony_fetch	Triggers the logging file to detect the source of the fetch, and build a client mapping table for parallel program tuning

formance data it observes during a single tuning task in a local text file. However, text files doesn't include any metadata information, which makes it hard for the users to analyze the tuning process.

Text files can be easily parsed by recursive regular expressions, because it only consists of configuration values and performance values which forms a k-tuple separated by certain delimiters, where k is equal to the number of parameters + 1.

4.3 XML

One major drawback of using text files is its lack of metadata information, including hardware configurations and operating systems information. Metadata is important because when referring to prior data which may or may not be produced on the same machine, the performance measurements generated on similar hardware configurations are more reliable. For example, when tuning an application on a x86 machine, we would always refer to the prior data generated by an x86 machine rather

than an ARM machine. Although We can easily put all such metadata into the text files, much extra effort will be spent on parsing such information. To address this problem, we also provide an XML format to store history data.

The XML format is a more formalized and a more common method of sharing data among different users. The intuition of using this format rather than text files is to provide the users a complete set of information of their tuning tasks, not limited to performance measurements. Other advantages of using the XML format include that it is easy to parse, to read and to modify.

There are several major advantages of using the XML format: 1. XML is basically a text file and doesn't rely on underlying systems, therefore it is portable; 2. XML has an organized and easy-to-read schema. Since there are well developed libraries for parsing and modifying XML files, it can also be easily used by others.

Figure 4.1 shows an example of the XML file generated by Active Harmony. The corresponding tree structure is composed of two parts: metadata and raw data. Metadata collects the general application information, the search space, as well as the operating system and hardware information of each client. Raw data collects each configuration generated by Active Harmony, the corresponding performance value reported by the client, as well as the source node that the measurement is done. The reason that we record the source node is to handle such cases that a parallel program is executing on a heterogeneous cluster where each node has different hardware configurations. In such cases, measurements from different nodes should not be mixed together.

```
▼<Harmony>
  ▼<HarmonyData>
    ▼<Metadata>
      ▼<Nodeinfo>
        ▼<Node>
           <HostName>bug00.umiacs.umd.edu</HostName>
           <sysName>Linux</sysName>
           <Release>2.6.18-128.el5.bug</Release>
           <Machine>i686</Machine>
           <ProcessorNum>4</ProcessorNum>
           <CPUVendor>GenuineIntel</CPUVendor>
           <CPUModel>Intel(R) Xeon(TM) CPU 2.66GHz</CPUModel>
           <CPUFreq>2658.009</CPUFreq>
           <CacheSize>512 KB</CacheSize>
           <ClientID>6</ClientID>
         </Node>
        ▶<Node>...</Node>
       </Nodeinfo>
       <AppName/>
      ▼<ParamList>
        ▼<Param>
           <paramName>TI</paramName>
           <paramMin>2</paramMin>
           <paramMax>500</paramMax>
           <paramStep>2</paramStep>
         </Param>
        ▶<Param>...</Param>
        ▶<Param>...</Param>
        ▶<Param>...</Param>
        ▶<Param>...</Param>
       </ParamList>
       <StartTime>124139</StartTime>
     </Metadata>
    ▼<RawData>
      ▼<Data>
        ▼<Config>
           <TI>251</TI>
           <TJ>251</TJ>
           <TK>251</TK>
           <UI>4</UI>
           <UJ>4</UJ>
         </Config>
         <Perf>488.000000</Perf>
         <Time>12:41:54</Time>
         <Client>6</Client>
       </Data>
      ▶<Data>...</Data>
     </RawData>
   </HarmonyData>
 </Harmony>
```


Figure 4.1: An example of Harmony's XML output

4.4 TAUdb

In order to better analyze prior data, saving them locally is inefficient because it reduces the accessibility of data as well as the portability of tuning tasks. Besides, the lack of metadata in text files also impedes the analysis of such information. For example, one may wish to investigate how CPU configuration impacts the performance of the application under the same configuration.

There are also several drawbacks of using the XML format regardless of its portability and ability of organizing data. One major problem is performance issue since indexing a specific data inside an XML file is currently based on linear search. This approach creates a performance bottleneck when storing massive data. Besides, since each single tuning task has its own XML data file, the management of such files is dependent on the file system and it is difficult to combine the files' names with their contents. Users are left to manually organize the naming of each XML file and have a knowledge of what is included in a specific file to do data analysis.

Given the disadvantages of using the XML format, storing data in databases becomes an attractive alternative. Compared to the XML format, storing data in a database has the advantage of ease of management, fast indexing and better inserting and updating performance.

4.4.1 TAU

The TAU Performance System [23] is a toolkit for analyzing the performance of parallel programs and is capable of gathering performance information through

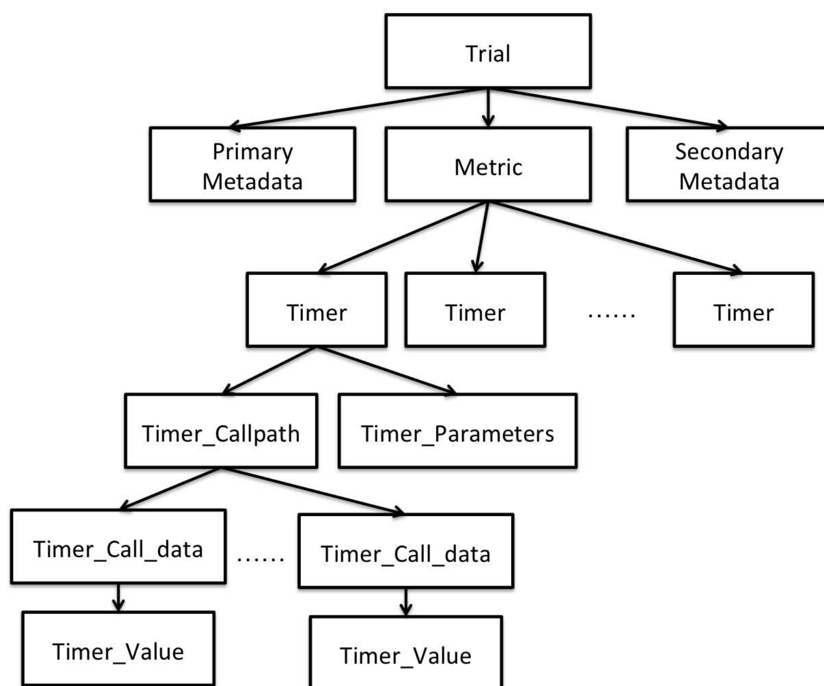


Figure 4.2: Data structures of TAU

instrumentation of functions, methods, basic blocks and statements. We integrated Active Harmony with TAU in order to save Harmony data in the database provided by TAU: TAUdb.

TAUdb is a derived work from PerfDMF [16] project, but it is more concise in the schema compared to PerfDMF. Besides, TAUdb provides C API functions that allowed us to directly call them from Active Harmony plugins.

4.4.2 Schema of TAUdb

Figure 4.2 represents the data structures used by TAU to record performance data. The database schema is also corresponding to this structure.

In Active Harmony, we use Trial to represent a single run of the target tuning task. Primary metadata corresponds to the general application information, such as the search space of tuning parameters, name of the application, etc. Secondary metadata is used to record the hardware or operating system information related to a specific thread or process. Metric refers to the cost function that needs to be optimized. In most performance tuning cases, metric is the runtime of the target applications. Each timer records all performance information for a specific routine. Timer call path are subroutines of a timer. Timer call data is used to record duplicate values when the same routine is called multiple times, and the performance values are stored in the timer value. Timer parameters holds the parameters of the routine, which are tuned by Active Harmony. Each new set of parameters generated by Active Harmony can be considered as a new timer since the target routine has been re-configured.

4.4.3 Visualization of Performance Data

TAU provides several tools to visualize the performance data, including PerfExplorer and ParaProf. With the help of such tools, Harmony users can have a better understanding of how their applications are tuned compared to the default settings.

Figure 4.3 shows the overview of an MPI program running on 4 nodes which is tuned by Active Harmony. The TAUdb plugin is able to distinguish performance data from different clients and load it into the database.

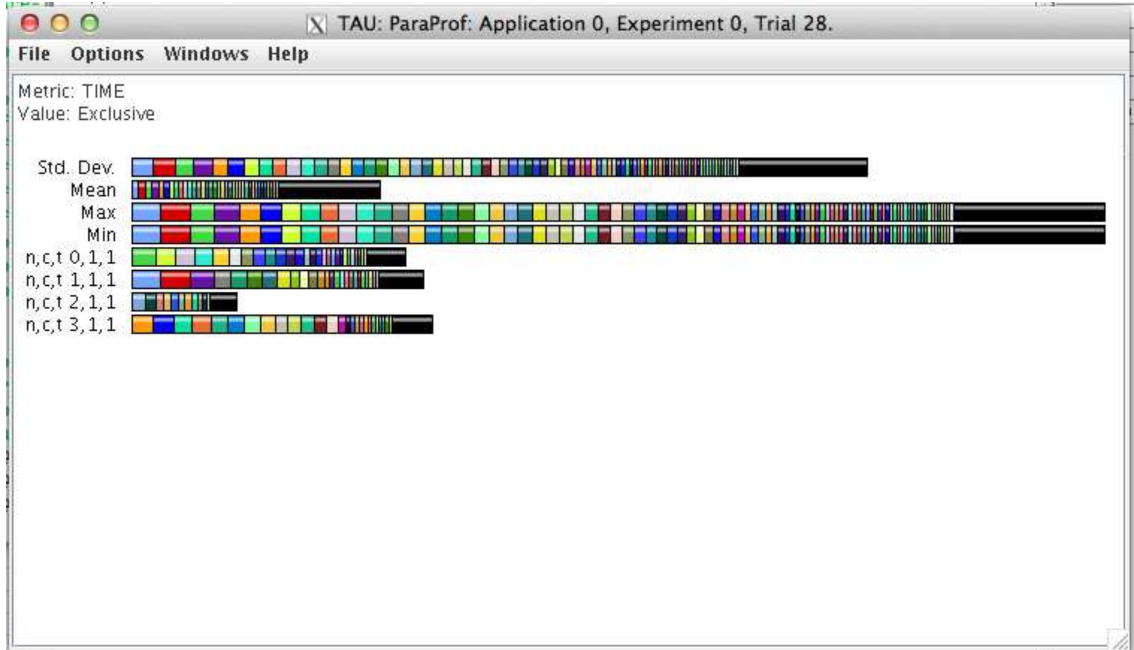


Figure 4.3: Example of "ParaProf" tool on an MPI program

Figure 4.4 shows how one of the 4 nodes is being tuned.

4.5 Discussion

In this chapter we introduced the architecture and functions of Active Harmony. We also presented some new methods of managing performance data generated by Active Harmony. We showed that these data are well organized and they can be applied to machine learning algorithms for search space modeling.

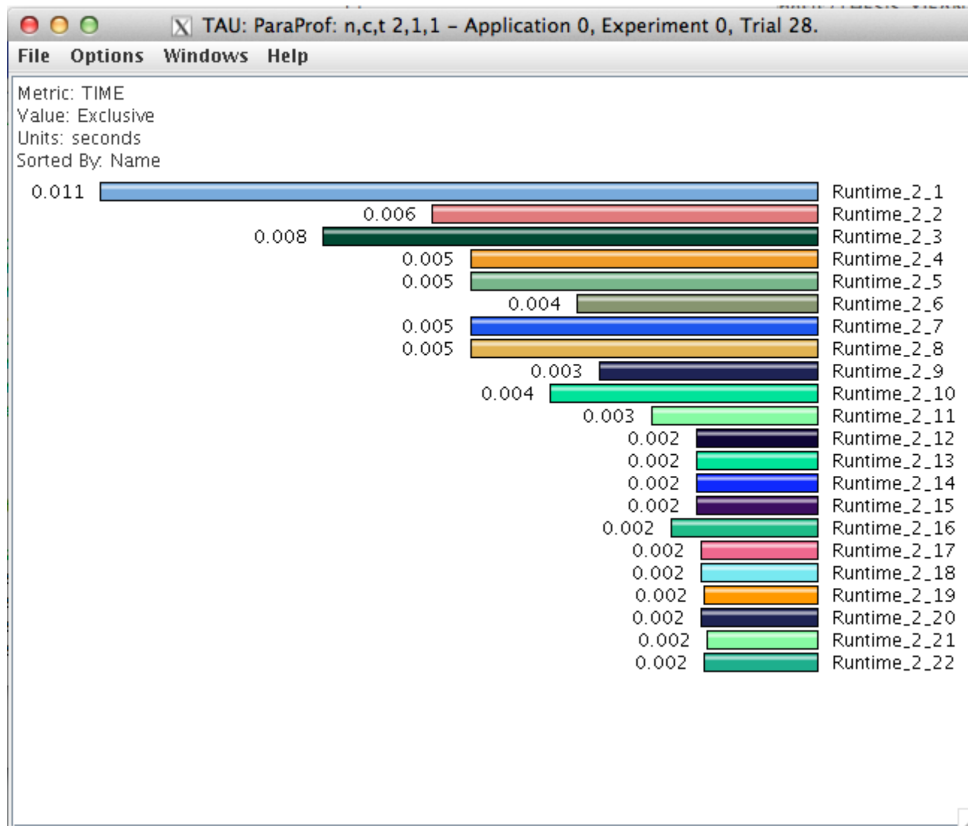


Figure 4.4: ParaProf's visualization on one node in an MPI program

Chapter 5: Tuning Database Systems

Performance of database systems has become a critical issue due to the increasing use of DBMS's in e-commerce applications. Database optimization methods usually include physical design tuning, query optimization, and tuning memory buffers or other parameters. In this chapter, we focus on applying Active Harmony to tuning database parameters such as memory buffers sizes, and parameters that affect the decisions made by the query planner.

5.1 Motivation

One traditional method for a database administrator to improve the performance of the database is by looking into the current transaction profiles and the details of incoming queries, then configuring the database parameters accordingly. However, this can be a tedious and repeating task, and it also requires trial-and-error for the administrator to get the optimal performance for a specific workload. In the worst case scenario it will take a DBA days or weeks to go through the process of performance tuning. Our work tries to address this problem by using Active Harmony to automate the entire tuning process.

Previous work on database optimization includes iTuned [12], STMM [25]

and other approaches [4] [1] [29]. iTuned used an adaptive sampling method with Gaussian process to predict the best point and to adapt the model into the real search space with empirical tests. STMM proposed an exponential function that tries to model the relationship between the size of each memory consumer and the total throughput of the database system.

This work was also motivated by the fact that previous methods can be improved by Active Harmony. iTuned applied Gaussian process on empirical data to capture the search space. However, this method requires a relatively large number of samples to fit the real hyper space to make the estimation accurate. In addition, this method is an offline strategy that runs on standby machines. Furthermore, iTuned is not responsive to changing workloads.

According to our observations, the model proposed by STMM is accurate for some workloads. However, this model may fail to model the search space for parameters that are not related to memory buffers allocation. Besides, STMM requires the client connections being stable. To address these problems, our approach applies a Nelder-Mead simplex method to optimize the parameters in a hyper space. Our results not only show that Active Harmony is able to tune various parameters efficiently with a semi-online tuning strategy, but also show that it is adaptive to changing workloads.

The final goal of this work is to automate the performance tuning process of database servers and improve the throughput compared to the default settings under certain workloads. We also compared the tuned performance with parameter values recommended by both PostgreSQL official site and its wiki [24].

5.2 Problem Statement

The database parameter optimization problem is similar to other optimization problems and can be stated as follows:

Having a series of parameters $P_1, P_2, ..., P_n$, an unknown environmental input E and a black-box function $F(P_1, P_2, ..., P_n, E)$, find the set of $(P_1, P_2, ..., P_n)$ *subject to* C that produces the most optimal output. In the case of database tuning, $P_1, P_2, ..., Pn$ are database settings, the environmental input is the workload that sends queries to the database and the objective function value of $F(P_1, P_2, ..., P_n, E)$ is the throughput (number of transactions per minute) of the database during a given time period.

5.3 Platforms

5.3.1 PostgreSQL

PostgreSQL [15](also Postgres) is chosen in our experiments as the target database to tune. PostgreSQL has a large number of settings that need to be tuned for optimal performance, and we mainly focus on memory buffer sizing parameters such as shared_buffer and work_mem, as well as the parameters that affect the strategy in which the server takes to process the queries.

5.3.1.1 Postgres Offline Parameters

There are some parameters in PostgreSQL that require restarting the database before they become effective. These parameters include shared_buffers, maximum

number of connections, etc. All of them are considered global because clients are not allowed to change them.

5.3.1.2 Postgres Online Parameters

Most of the parameters in PostgreSQL can be reconfigured online, which means that these parameters can be reconfigured and reloaded without restarting the server. Such configurations include effective_cache_size, work_mem, random_page_cost, etc. Each of them is a major factor for a specific class of SQL queries.

Some of the online parameters are globally initialized but can be customized by each database client. However, in our study, all memory buffer sizes and parameters are treated as global variables, which means each client consumes the same amount of memory. The intention was to find a global configuration that optimizes the throughput for a workload, instead of each individual client.

5.3.2 TPC-H Benchmark

We use the TPC-H benchmark [9] to generate the workload of Postgres. TPC-H is a decision support benchmark that consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The query models and data set have broad industry-wide relevance and can be used to simulate industrial database workloads. The benchmark simulates a decision supporting system that examines large volumes of data and executes queries with a high degree of complexity. The

performance of TPC-H queries is measured by Composite Query-per-Hour, which can reflect the database throughput when queries are submitted concurrently. More details can be found on TPC website [9].

The benchmark can generate different queries under the same template. We used this feature to randomly generate queries under the same TPC-H query type. For example, a workload of 6Q7 includes 6 concurrent TPC-H query number 7s with different contents.

We built the workloads by selecting different types of queries from the benchmark and combined them to build multiple search spaces. The details are discussed in Section 5.4.2.

5.3.3 Environment

The machine that runs the database server is equipped with four Intel (R) Xeon(R) processors, each with a clock frequency of 2.33GHz. The total available RAM is 4 GB. The operating system we used is 64-bit Linux 2.6.18 smp, . We have also populated a database using a data set of 10 GB from the TPC-H benchmark.

5.4 Methodologies

5.4.1 Performance Measurement

Previous work on self-tuning of database systems usually aimed at optimizing the runtime for each query or optimizing the number of queries committed in a given time interval, in other words, improving the throughput of a database.

There have been multiple research projects that tuned the amount of running time for a single query or a certain combination of different queries. However, in an industrial database workload, queries are usually an interleaved mixture of different segments, so it is hard to find a specific configuration to satisfy the performance requirements for each client query. The reason is that the client workload might be a large number of combinations of different basic query blocks, and the search space varies for each combination.

An alternative is to find a global configuration shared by all clients that optimizes the number of transactions processed by the database within a certain period of time. In our experiments, the database performance is based on the number of transactions processed per minute.

When a new configuration is applied to the database system, we also assign an amount of time to warm up the cache and memory, so that the memory access pattern can be stabilized before starting to sample the performance value.

One major problem of sampling the throughput value of PostgreSQL is that when running the same configuration under same workload for multiple times, the performance may vary. There are two major reasons that cause the problem: 1. The database performance is vulnerable to external noise; 2. The throughput metric used by PostgreSQL is not accurate. The integer number of committed queries is recorded by the variable xact_commit in a table called pg_stat_database. However, a query is not considered committed even if it is mostly done. This leads to the fact that although a query might almost be finished within a sampling period, it might contribute the throughput number to the next sampling period.

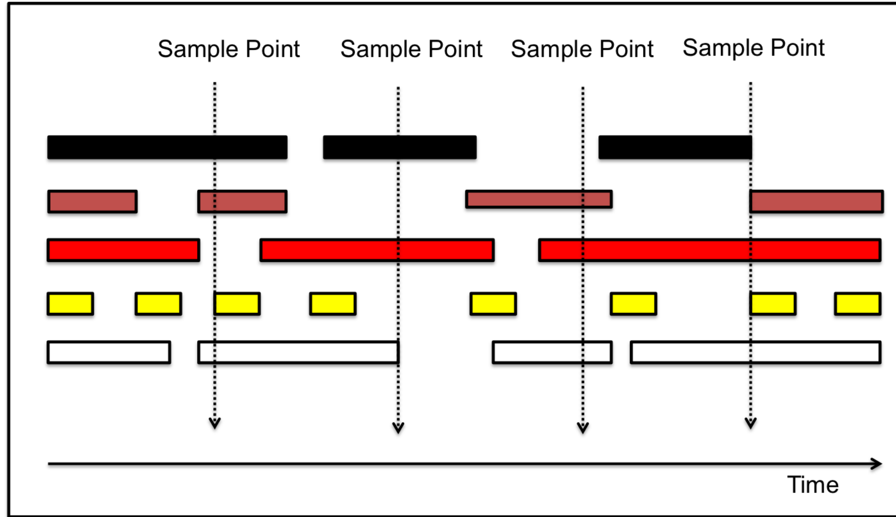


Figure 5.1: The Sampling Problem of PostgreSQL: the colored lines are the timespans for the server to process a single query. The sampling point only considers committed queries and ignores running queries, which may cause the sampled performance to be unstable for the same configuration

Figure 5.1 illustrates an example for the sampling problem stated above. It shows that although an interval may contribute most for some queries, it can only be recorded as a valid transaction after it finishes.

To address this problem, multiple runs have been done on the same configuration and the best performance value among them is reported to the Active Harmony server. An empirical test has been done to decide the number of runs that is needed to avoid such fluctuation, and in our case, we ran 3 samples for each set of configurations.

5.4.2 Workload

The optimal value for each configuration varies among different TPC-H queries. For each class of the queries, there are some specific database settings that dominate the performance of that query. The SARD [11] project provides a statistical research on how PostgreSQL configurations have an impact on different queries. We took into consideration their statistics to help us decide the tuning parameters.

Table 5.1: Ranking of configuration parameters for a workload of TPC-H queries [11]

Parameter	Q1	Q8	Q9	Q13	Q16
cpu_tuple_cost	15	6	15	15	15
effective_cache_size	15	15	11	1	15
geqo	15	6	7	15	15
maintenance_work_mem	15	2	3	15	15
deadlock_timeout	15	6	15	15	15
max_connections	15	6	5	15	15
random_page_cost	15	12	1	15	15
shared_buffers	15	12	3	2	2
temp_buffers	15	12	7	15	15
work_mem	1	1	13	3	1

Table 5.1 shows that for a specific query, the importance of different database parameters varies. We are interested in setting up multiple workloads, and each of them consists of a combination of queries that are dominated by more than one database parameter. We also studied the characteristics of the top ranked parameters. The detailed analysis and results are shown in Section 5.4.4

5.4.3 Tuning Parameters

Table 5.1 [11] provides an overview of how performance of each type of TPC-H queries is affected by database parameters. To maximize the benefit of tuning, we focus on tuning the parameters having the greatest performance impact. Besides, in order to study the search ability of search heuristic given a fixed memory budget (the buffer limit), we included all the parameters related to memory buffer allocation. Table 5.2 shows the list of PostgreSQL parameters that are chosen to be tuned by Active Harmony in this work.

Table 5.2: PostgreSQL parameters chosen for performance tuning

Parameter	Function
work_mem	Specifies the amount of memory to be used by internal sort operations and hash tables before switching to temporary disk files
shared_buffers	Specifies the amount of memory that PostgreSQL uses for shared memory buffers
maintenance_work_mem	Specifies the maximum amount of memory to be used in maintenance operations, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY
effective_cache_size	Tells the database the size of the operating system data cache, so that the database can draw different execution plans based on that data
random_page_cost	Sets the ratio of doing a random disk access from the database over a sequential access, which influences the planner's choice of index vs. table scan, this value highly depends on the hardware performance
temp_buffers	Sets local buffers for accessing temporary tables

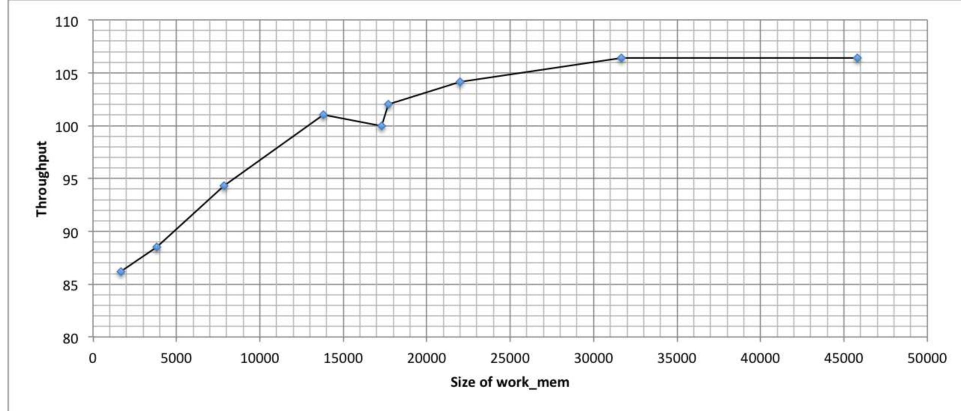


Figure 5.2: Throughput vs. work_mem on a workload with 12Q7. It shows that the buffer size vs. throughput can be modeled as an exponential function under a simple workload

5.4.4 Search Space

The multi-dimensional search space varies among different workloads and its projection on a lower dimensional space shows its nonlinear characteristics [12]. In order to understand the key features of the space, we projected the space onto a 1-dimensional space. We first verified the accuracy of the exponential function model proposed by STMM, and then present cases that the exponential model does not work well.

Figure 5.2 and 5.3 show the impact of work_mem and shared_buffers vs. throughput under simple workloads(which are composed of same type of TPC-H queries), when all other parameters are set with their default values. It is an example that verifies the statement by STMM project [25] that the memory characteristics can be modeled as a simple exponential function.

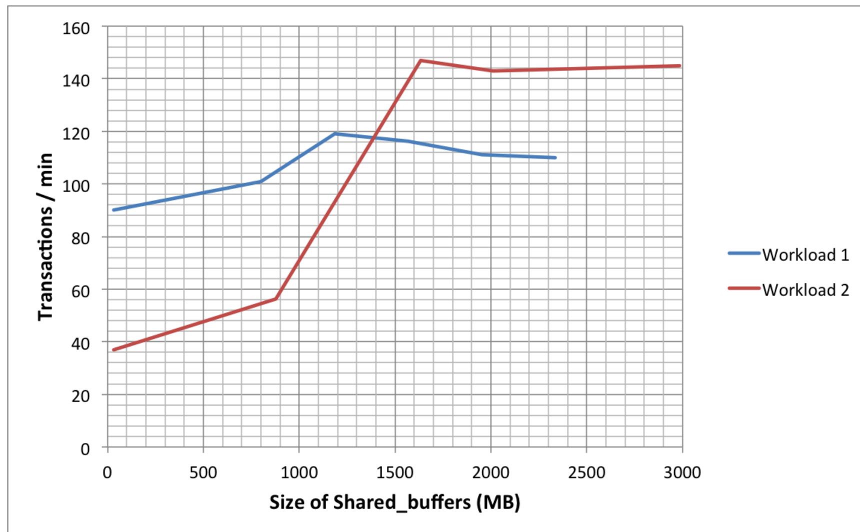


Figure 5.3: Throughput vs. shared_buffers on 2 Different Workloads

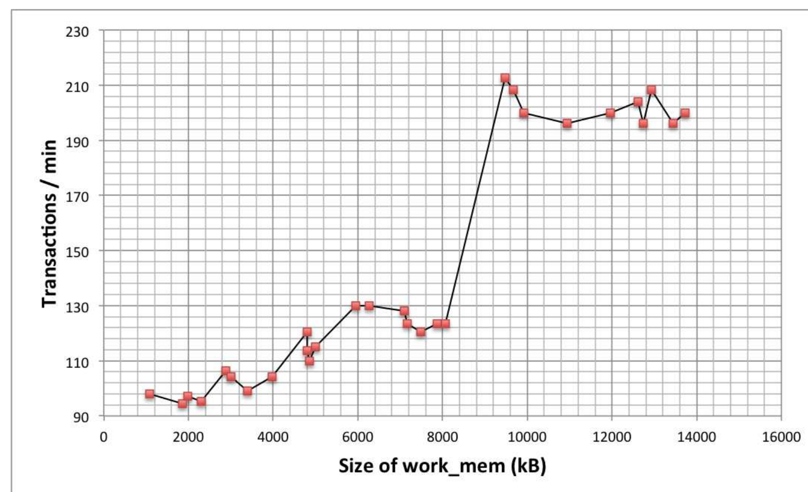


Figure 5.4: Throughput vs. work_mem on workload with 3Q6, 3Q7, 3Q8, 3Q13. The model is much worse in this case and It produces higher error rate when fitting into these samplings

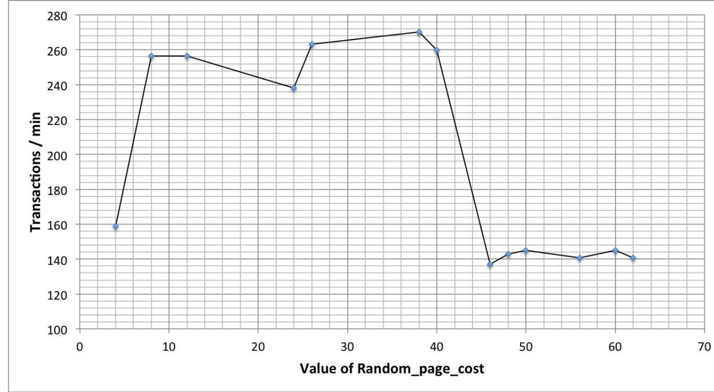


Figure 5.5: Throughput vs. random_page_cost on workload with 12Q6

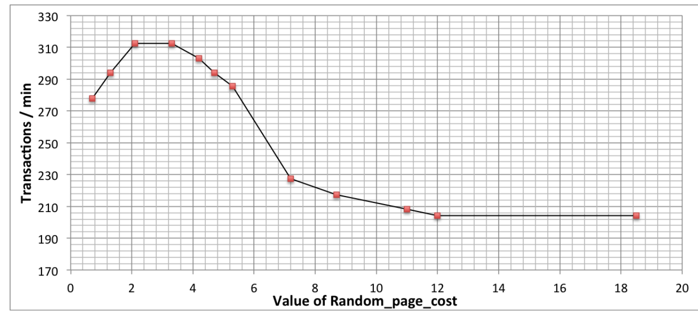


Figure 5.6: Throughput vs. random_page_cost on Workload with 3Q6, 3Q7, 3Q8, and 3Q13

The exponential model becomes less accurate for more complex workloads because of the cumulative effects of different types of TPC-H queries. Figure 5.4 illustrates a more complicated 1-D projection for work_mem vs. throughput for a more complex workload with 3Q6, 3Q7, 3Q8 and 3Q13.

The exponential function model fails for the parameters that are not related to buffer allocation. Figure 5.5 shows the projection of random_page_cost vs. the throughput on a workload with 12Q6. This search space has a "cliff" that is rel-

atively hard for methods such as gradient descent to effectively optimize. Yet, Figure 5.6 presents the shape of a workload with mixed queries, which appears to be smoother than the previous curve.

5.4.5 Search Strategy

In this work, we focused on Active Harmony's ability to improve the throughput of a single database. We used the standard Nelder-Mead simplex method to optimize the database performance. The details of this algorithm are presented in Chapter 3.

5.5 Online-Offline Co-tuning

Given that the parameters are divided into online and offline parameters, the entire tuning task can now be divided into the online session and the offline session.

During the offline tuning session, the Nelder-Mead simplex consists of 2 vertices to optimize a single parameter: shared_buffers. It forms a 5 parameter simplex during the online tuning session. The major reason that prevented us from doing linear search on each parameter is that, under memory budget, the parameter order can hardly be determined given an unknown workload.

The offline session focuses on tuning shared_buffer. During the offline session, the database is restarted at every new configuration and disallows any incoming connections during the restart. Only a few iterations are needed to tune the offline session because only one parameter is involved. The online session focuses on tun-

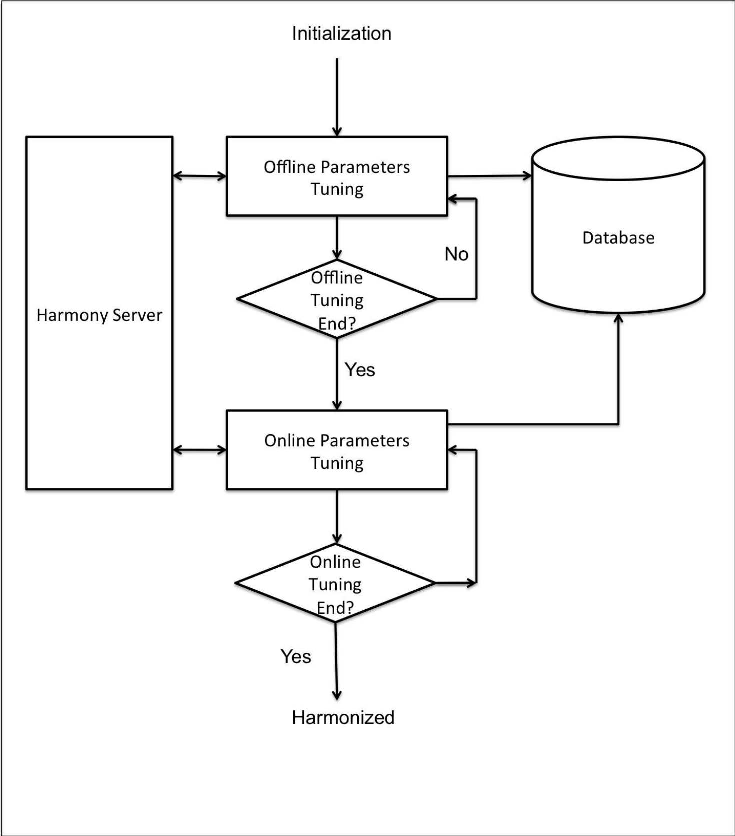


Figure 5.7: One-pass tuning flow

ing work_mem, maintenance_work_mem, effective_cache_size, random_page_cost and temp_buffers. Online session requires more iterations to converge.The combination of the two sessions forms a semi-online tuning flow.

5.5.1 One-pass Tuning Flow

Figure 5.7 illustrates a one-pass tuning flow. A one-pass tuning represents a work flow that tunes the offline session first, followed by the online session. This is a simple work flow for tuning a non-changing workload.

5.5.2 Multi-pass Tuning Flow

The major disadvantage of the one-pass tuning work flow is that, it is not adaptive to changing workloads. Figure 5.8 shows a multi-pass tuning strategy which aims at addressing this problem by repeatedly running offline and online parameter tuning. It provides several advantages: 1. When the performance value changes dramatically after the simplex of the online tuning session converges, we assume that there is a change in the workload, and Active Harmony can immediately restart both tuning session to adapt to the new workload; 2. If there is a workload change but produces the same performance as the converged value, the Harmony server can still adapt to it after the online tuning loop is over. Therefore, the tuning process can be made sensitive to changes in workload over time; 3. Active Harmony can abort the online session and restart from the offline session if the simplex of the online tuning session falls into a local minimum. Active Harmony detects a local minimum for the online session if the performance value of all vertices in the simplex are worse than the best performance in the previous offline session by a given threshold. If there exists a route to escape from the local minimum, the multi-pass tuning flow provides the ability for Active Harmony to escape from local minimum areas.

5.5.3 Scalability

Since Active Harmony is a scalable framework that can be connected by multiple clients, it can be utilized to tune multiple nodes in a database cluster in parallel

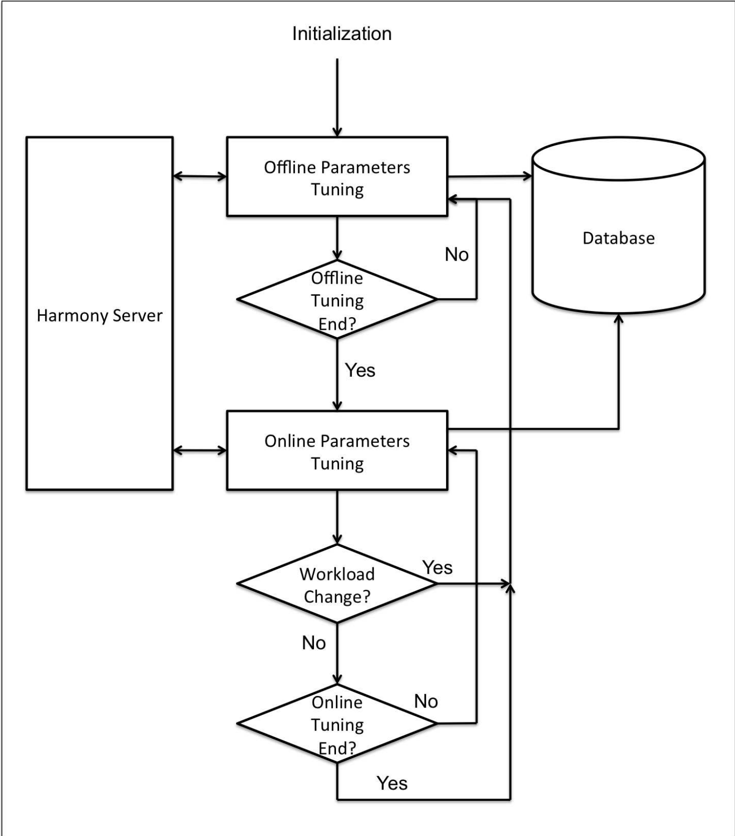


Figure 5.8: Multi-pass tuning flow

using PRO algorithm (see Section 3.4.3), if the nodes are load-balanced and applied with similar types of workloads. The PRO method proved to be much faster than the sequential Nelder-Mead algorithm [28] in tuning parallel programs because it measures newly generated points in parallel at each iteration.

5.6 Memory Budget Constraints

A DBMS always has a problem when the desired memory space exceeds the total available physical memory. For example, setting the work_mem to be 100MB may satisfy the performance requirements of queries including sorting operations. However, doing so is dangerous because work_mem is allocated upon each connection and sorting operation. To address this issue, we define a global memory budget constraint on all memory consumers.

All memory buffers should follow the inequality 5.1 to prevent memory overflow.

$$W * n + \ S \ + \ M \ + \ T \ + \ O \ < \mathit{Total\ Available\ RAM} \tag{5.1}$$

where W is work_mem, n is the total number of connections and sort operations, M is maintenance_work_mem, S is shared_buffer, T is temp_buffers and O refers to operating system reserved memory.

However, directly applying this rule to the tuning system is problematic because this rule includes all memory buffers, but the offline tuning session is separate from online parameters. To address this problem, we set a budget for shared_buffers

and allow it to consume up to 50% of the RAM size, then, we define a new memory budget constraint for the online session according to the remaining RAM space.

After the offline search, we defined an inequality 5.2, a memory budget constraint rule for online parameters and applied it to the search space.

$$W * m + M + T < \ Total\ RAM - Shared_buffers - O \tag{5.2}$$

where m is the number of connections. This inequality is an estimation of the memory constraint and has an error rate of $W * (n - m)$.

At runtime, in order for the online parameters to follow the constraint inequality 5.2, we used a tool called Omega Test [21] to handle the constraint problem. Omega Test is a system to manipulate sets of affine constraints over integer variables and it was originally designed as a decision test for the existence for integer solutions to affine constraints. The main goal for us to use Omega Test was firstly, to detect if the generated configuration meets the memory budget constraint; Secondly, to determine the upper and lower bounds for individual parameters given a memory constraint. It helps Active Harmony to reduce the search space by creating a bounding box for the feasible area. When initializing the simplex, a pure random sampling is applied on the bounded space to generate the initial simplex. The initialization does not end until all initial vertices are feasible, which takes a few seconds.

Although we guarantee that all vertices in the initial simplex meet the constraint rule, they may move to the infeasible area during simplex transformation

operations (e.g. reflection). There are two commonly used strategies to handle infeasible parameters. Tiwari et al. [2] used an approximate nearest neighbor (ANN) projection server [26] to project an infeasible point to its nearest feasible neighbor. However, such strategy requires pre-loading feasible samples of the entire search space. Since the search space highly depends on the workload, ANN server has to reload feasible samples every time the workload changes, and the overhead produced by loading such samples at runtime is not acceptable in our case.

We used a second strategy by reporting a penalized performance measurement for an infeasible point. In this method, Harmony server forwards every point generated by the search engine to the client and notifies the client if the point is feasible or not. The client reports the penalty value immediately back to the server whenever it finds the point infeasible, without re-configuring the database server. Since the penalized measurement is pre-assigned to a large value, the Nelder-Mead method always discards this point and later contracts (or reduces) the simplex to eventually reach a feasible point. Using this method only produces little overhead.

Figure 5.9 illustrates the process in which one of the vertices goes out of the feasible area. Since the new point is assigned to a penalized value and is guaranteed to be the worst in the reflected simplex, Active Harmony then considers the reflection to be a failure and contract the original simplex that finds a new point within the feasible area.

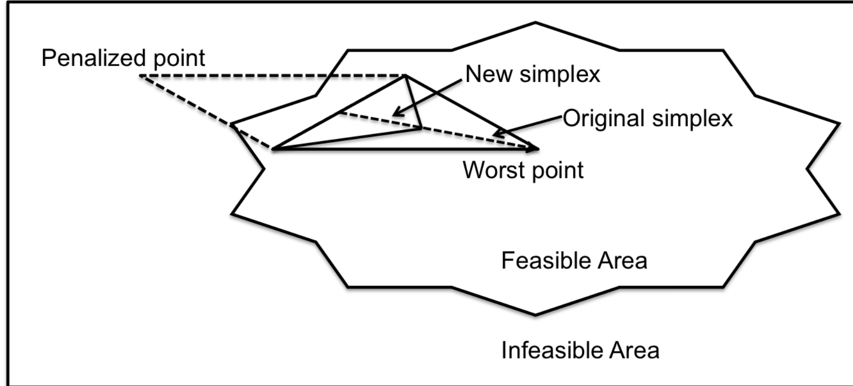


Figure 5.9: Avoiding infeasible data points by applying penalty factor. In this case the reflected point is discarded and it causes the simplex to constract.

5.7 Experimental Results

5.7.1 Search Space versus Search Ability

In our experiments, we set the upper and the lower bounds for each parameter using commonly used PostgreSQL parameter ranges. We assigned was set upper and lower bounds for work_mem with a range of (1MB, 200MB), maintenance_work_mem with a range of (1MB, 1GB), temp_buffers with a range of (1MB, 1GB), and shared_buffers with a range of (32MB, 2GB). During the tuning process, the search space is reduced by Omega based on the given memory budget. Since random_page_cost is set to 4 by default based on a 90% cache hit rate, we set the random_page_size from 0.1 to 30 assuming that the workload does not generate a cache miss rate which is approximately higher than 75%. And effective_cache_size is assigned a range of (128MB, 2GB). We also applied a memory budget of 1GB on

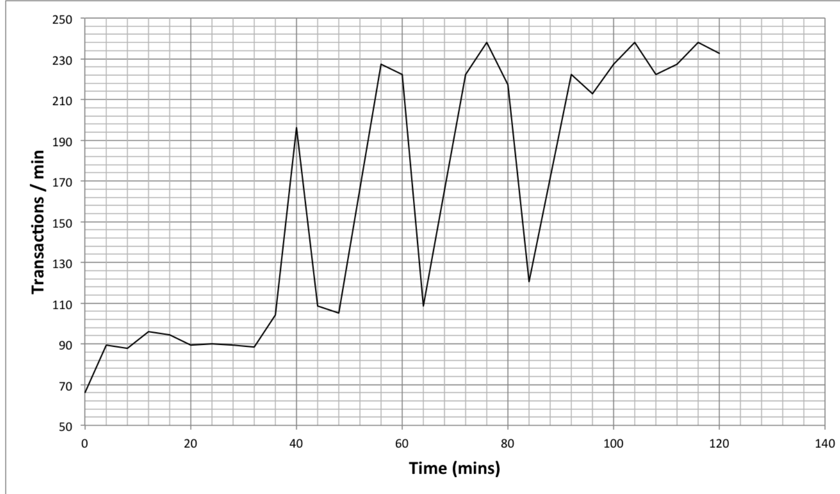


Figure 5.10: Tuning curve for 4Q6, 4Q9, and 4Q13

all online memory buffers parameters.

Figure 5.10 shows the tuning process for a workload with 4Q6, 4Q9 and 4Q13. Figure 5.11 shows the results of the harmonized throughput for 6 different workloads. The results show that Active Harmony is capable of tuning the throughput of the database empirically under reasonable constraints and greatly improves the performance of the database compared to the default settings by an average of 300%. We also compared our results to the performance generated by the recommended setting. The recommended setting is: 25% of total RAM space (1GB) for shared_buffers, 50MB for work_mem, 128MB for maintenance_work_mem, 4 for random_page_cost, 2GB for effective_cache_size and 256MB for temp_buffers. The results show that our method can achieve an average of 100% improvement over the recommended settings for 4 out f 6 workloads.

Active Harmony is sensitive to the user-defined search space for specific prob-

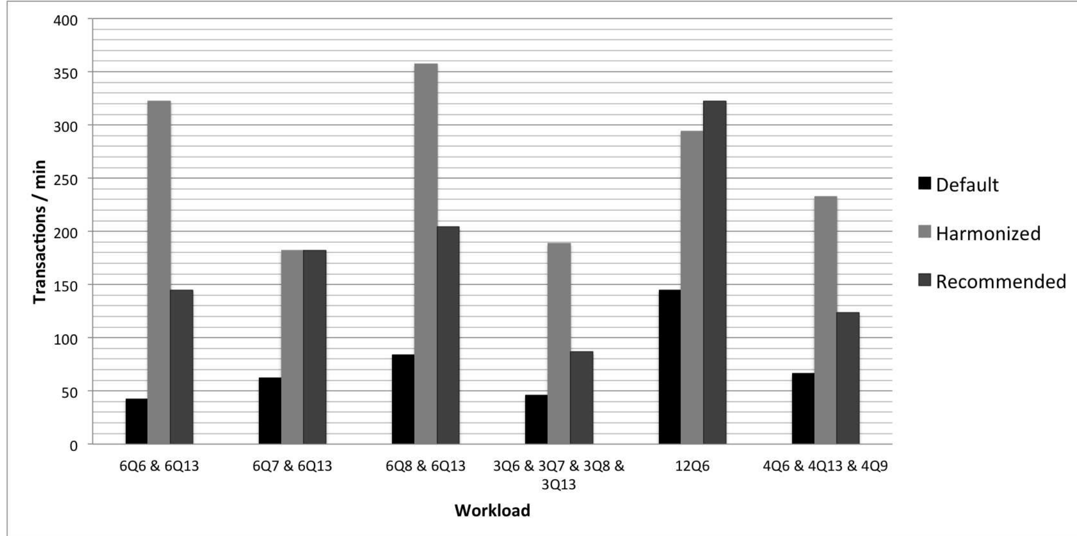


Figure 5.11: Tuning result for multiple workloads

lems. Our experiments also show that enlarging the search space degrades the final converged performance in database tuning. To show how the search range affects the simplex algorithm in database tuning, we assumed that the administrator is ignorant about database administration and adopts redundant search space by assigning a large search range for parameters without any budget constraint.

Figure 5.12 shows the tuning process of a workload with 3Q6, 3Q7, 3Q8 and 3Q13. Although Active Harmony greatly improves the throughput of this workload compared to the default settings, it fails to perform as good as the result shown in Figure 5.11 for the same workload. This is mainly because random_page_cost is converged at a value greater than 100, which indicates that the cache miss rate is very high for this workload, and this causes the query planner to use index scan rather than random disk access. Therefore, enlarging the range of parameters may limit the search ability for the Nelder-Mead method because it is harder to find a

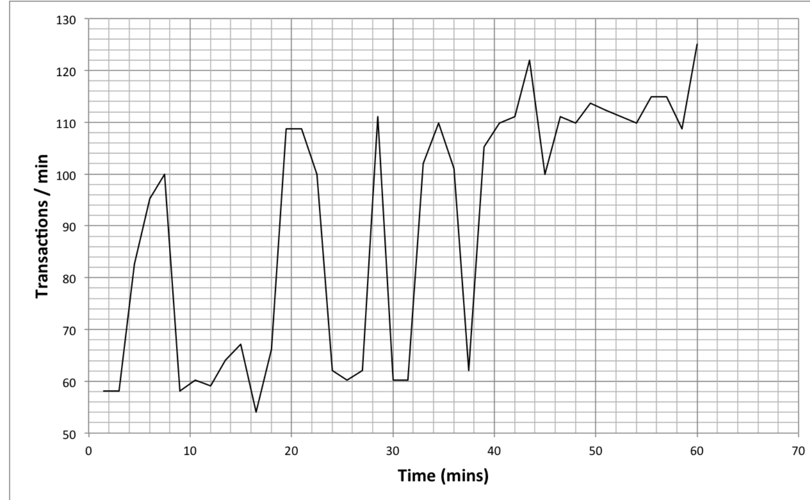


Figure 5.12: Tuning curve for 3Q6, 3Q7, 3Q8, and 3Q13

good initial simplex.

5.7.2 Impact of Memory Budget Constraints

Knowing how the memory budget constraint affects the search ability of the Nelder-Mead method is an interesting topic to study, and it helps us to understand the limitations of this search strategy. In order to conduct such an experiment, we intended to test on a workload whose performance is affected by more than one memory buffer. Then, we tightened the constraint so that the memory buffers compete with each other. We built a workload which consists of multiple TPC-H query types, including 3Q6, 3Q7, 3Q8, 3Q13 and 3Q15.

In this experiment, we only focused on online parameters (shared_buffers is set to 2GB for all tests). We compared the performance obtained by the Nelder-Mead search with different memory budgets, ranging from 2GB down to 64MB. We also

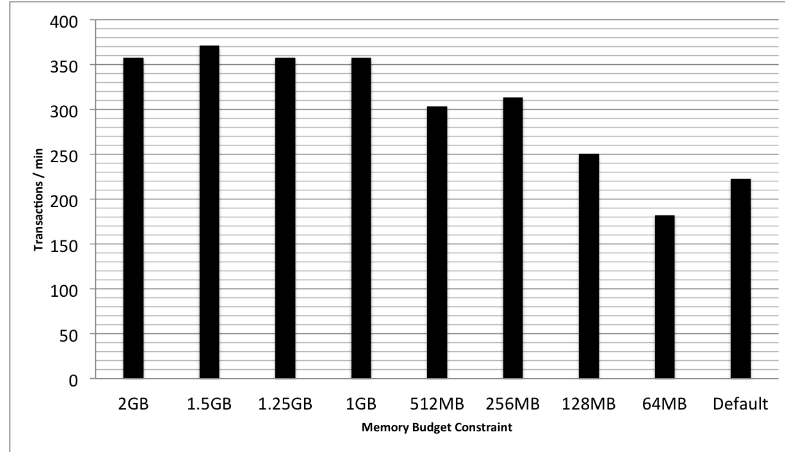


Figure 5.13: Search result by the Nelder-Mead vs. memory budget constraint for online parameters

tested the performance values for default online parameter setting and compared it to our tuning results.

Figure 5.13 shows the relationship between the values of memory budget constraints versus the final throughput optimized by Active Harmony. We ran 3 tests using single-pass tuning work flow on each of the constraints and picked the worst result to reduce the possibility that a good performance was produced by randomness. The results show that the performance begins to degrade from a budget of 512MB in the first pass. And it performs worse than the default setting with a budget of 64MB. However, by applying a multi-pass tuning flow, when the online session falls into a local minimum, Active Harmony can detect it and restart from the offline session, and it is possible to escape from the local minimum area.

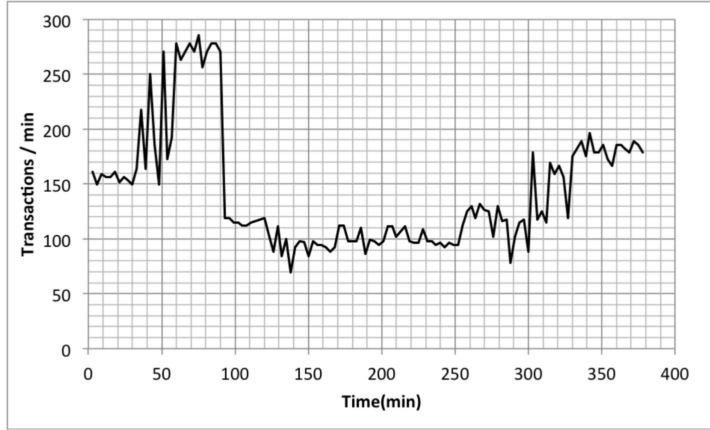


Figure 5.14: Workload shift from one workload to another after 90 minutes.

5.7.3 Workload Shifting

Active Harmony is sensitive to dramatic shifts of database workloads after the Nelder-Mead simplex converges during the online tuning session. The convergence of the simplex is defined as: all performance values in the simplex are subject to a given variance, or all vertices are converged to a single point. At the moment when the simplex converges, Active Harmony records the best and the worst value among all vertices in the converged simplex and creates a bound. Whenever there is a fixed number (current value is 3) of later data points that are out of the bound, Active Harmony regards such condition as a workload shift.

To show how Active Harmony detects workload shifts, we carried out an experiment to run a workload with 12Q6 for 90 minutes, and then shifted to a workload with 6Q7 and 6Q13. A memory budget of 512MB was also applied to the search space for online parameters.

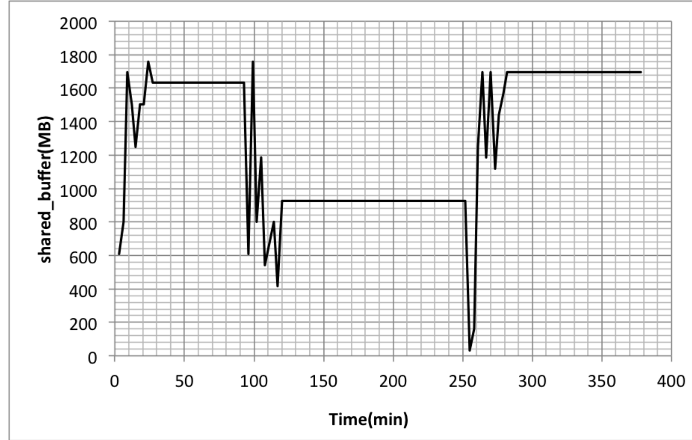


Figure 5.15: Shared_buffers shift corresponding to workload shift

Figure 5.14 shows that the workload changes dramatically at 90 min, which was detected by Active Harmony. Then, Active Harmony restarts from the offline tuning session to adapt itself to the new workload.

Figure 5.15 shows the corresponding shift for shared_buffers. After Active Harmony detects the workload shift, it begins to re-configure this parameter to adapt to the new workload.

This is also an example to show the reason why a multi-pass tuning work flow is necessary for tuning long running servers. After shifting the workload, Active Harmony is able to detect when the simplex converges to a local minimum and to restart the offline tuning session to approach the global minimum if there exists a route. We observed that Active Harmony finds a much better value during the second pass. Figure 5.16 shows that the curves for shared_buffers vs. throughput change under different online parameter values. It indicates that offline and online sessions are dependent to each other to some extent, and this is one of the reasons

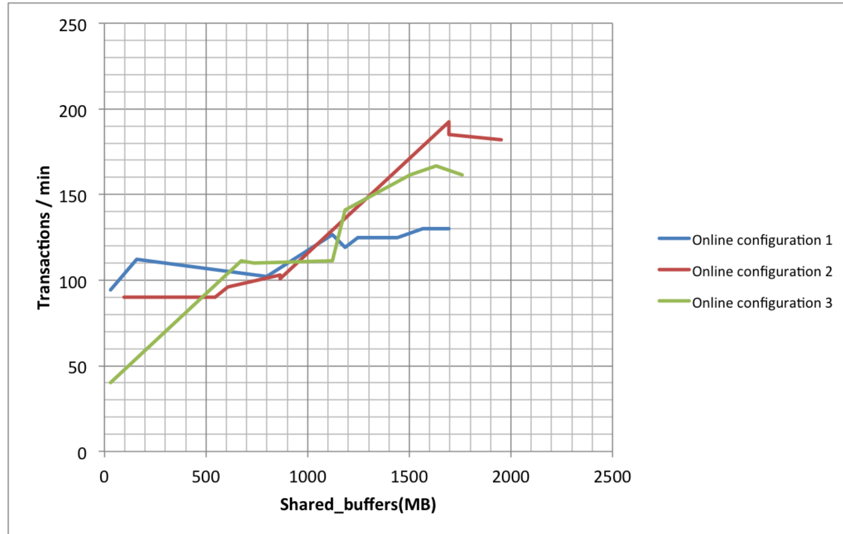


Figure 5.16: Shapes of shared_buffers differ under 3 different online parameters for the same workload. It indicates that the search space for offline parameter is dependent on the status of online parameters

that we use a multi-pass tuning flow in order to drive offline and online parameters out of the local minimum area.

Chapter 6: Future Work

In the short term, one of the future directions is to find a good initial simplex for Nelder-Mead method. The current approach in Active Harmony uses pure random sampling or boundary values to initialize the simplex. The former method is not always robust, and when a strict constraint is applied to the search space, both methods become not robust. One strategy to address the problem is to apply Latin Hypercube Sampling [19] technique to create a more evenly distributed sampling on the space. Another method is to refer to the prior data stored in TAUdb or XML files to form the initial simplex using best configurations from prior runs.

Another short-term goal is applying our method on a load-balanced database cluster, where all nodes have the same physical design and similar types of query loads. Multiple servers can cooperate with each other to tune the performance in parallel, and we believe that the parallel tuning strategy can outperform the sequential Nelder-Mead method due to its ability to exploit the search space in parallel.

In future auto-tuning tasks, runtime performance is not the only metric to evaluate a system. Researchers have also focused on tuning and balancing power versus performance in high performance systems [27]. Also, in database auto-

administration, balancing the performance and the memory consumption is also an important topic to study.

Chapter 7: Conclusions

In this thesis, we reviewed Active Harmony, an empirical automated tuning framework. We discussed several previous use cases that used Active Harmony to tune different type of applications. We also pointed out the limitations of previous versions of Active Harmony in managing prior data, and we presented the development of two methods to store prior data in order to provide the users a complete set of tuning information.

We also applied Active Harmony to optimize the performance of database servers for different workloads. Compared with other works, our approach achieves both effectiveness and efficiency with a semi-online strategy. It can be applied to not only memory buffer sizing, but also the parameters that are not related to memory buffers. Results show that the throughput can be improved by an average of 300% for the 6 workloads that we tested compared to the default settings. Besides, most tuned servers also outperformed the performance of servers under recommended settings from PostgreSQL website. It further proves that with proper underlying tuning algorithms, Active Harmony is a useful tool in tuning many kinds of application, not limited to compiler optimization problems. We also discussed the impact of constraints by showing that the search result using the Nelder-Mead method is

robust with a loose constraint but starts to degrade upon a threshold, and showed that a multi-pass tuning work flow can improve the search ability of Nelder-Mead method under a certain constraint.

Bibliography

[1] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. Automatic physical design tuning: workload as a sequence. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 683–694, New York, NY, USA, 2006. ACM.

[2] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, SODA '94, pages 573–582, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[3] J. Bergstra, N. Pinto, and D. Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing (InPar), 2012*, pages 1 –9, may 2012.

[4] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 3–14. VLDB Endowment, 2007.

[5] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, 2008.

[6] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[7] I-Hsin Chung. *Towards automatic performance tuning.* PhD thesis, College Park, MD, USA, 2004. AAI3153156.

[8] I-Hsin Chung and Jeffrey K. Hollingsworth. Automated cluster-based web service performance tuning. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04, pages 36–44, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Transaction Processing Performance Council. http://www.tpc.org.

[10] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[11] Biplob K. Debnath, David J. Lilja, and Mohamed F. Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering Workshop*, ICDEW '08, pages 11–18, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, 2(1):1246–1257, August 2009.

[13] Kenji Fukumizu, Francis R. Bach, and Arthur Gretton. Statistical consistency of kernel canonical correlation analysis. *J. Mach. Learn. Res.*, 8:361–383, December 2007.

[14] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 592–603, Washington, DC, USA, 2009. IEEE Computer Society.

[15] The PostgreSQL Global Development Group. http://www.postgresql.org.

[16] Kevin A. Huck, Allen D. Malony, Robert Bell, and Alan Morris. Design and implementation of a parallel performance data management framework. In *IN: PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON PARALLEL COMPUTING*, pages 473–482, 2005.

[17] Keven Hucks, Suzanne Millstein, Allen D. Malony, and Sameer Shende. Taudb: Perfdmf refactored, 2012.

[18] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. Autotuning stencil-based computations on GPUs. In *Proceedings of IEEE International Conference on Cluster Computing (Cluster'12), Beijing, China, September 24-28, 2012*. IEEE Xplore, May 2012.

[19] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, February 2000.

[20] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[21] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278, July 1994.

[22] Karl Schnaitter, Serge Abiteboul, Tova Milo, and Neoklis Polyzotis. Colt: continuous on-line tuning. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 793–795, New York, NY, USA, 2006. ACM.

[23] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[24] Greg Smith. http://wiki.postgresql.org/wiki/tuning_your_postgresql_server.

[25] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 1081–1092. VLDB Endowment, 2006.

[26] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Ananta Tiwari, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. Auto-tuning for energy usage in scientific applications. In *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*, Euro-Par'11, pages 178–187, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. Tuning parallel applications in parallel. *Parallel Comput.*, 35(8-9):475–492, August 2009.

[29] Dinh Nguyen Tran, Phung Chinh Huynh, Y. C. Tay, and Anthony K. H. Tung. A new approach to dynamic self-tuning of database buffers. *Trans. Storage*, 4(1):3:1–3:25, May 2008.