ABSTRACT

Title of Dissertation: TOWARDS AUTOMATIC PERFORMANCE TUNING

I-Hsin Chung, Doctor of Philosophy, 2004

Dissertation Directed By: Professor Jeffrey K. Hollingsworth, Department of Computer Science

When the computing environment becomes heterogeneous and applications become modular with reusable components, automatic performance tuning is needed for these applications to run well in different environments. We present the Active Harmony automated runtime tuning system and describe the interface used by programs to make applications tunable. We present the optimization algorithm used to adjust application parameters and the Library Specification Layer which helps program library developers expose multiple variations of the same API using different algorithms. By comparing the experience stored in a database, the tuning server is able to find appropriate configurations more rapidly. Utilizing historical data together with a mechanism that estimates performance speeds up the tuning process. To avoid performance oscillations during the initial phase of the tuning process, we use improved search refinement techniques that use configurations equally spaced throughout the performance search space to make the tuning process smoother. We also introduce a parameter prioritizing tool to focus on those performance critical parameters. We demonstrate how to reduce the time when tuning a large system with

many tunable parameters. The search space can be reduced by checking the relations among parameters to avoid unnecessary search. In addition, for homogeneous processing nodes, we demonstrate how to use one set of the parameters and replicate the values to the remaining processing nodes. For environments where parameters can be divided into independent groups, an individual tuning server is used for each group. An algorithm is given to automatically adjust the structure of cluster-based web systems and it improves the system throughput up to 70%. We successfully apply the Active Harmony system to a cluster-based web service system and scientific programs. By tuning the parameters, Active Harmony helps the system adapt to different workloads and improve the performance up to 16%. The performance improvement cannot easily be achieved by tuning individual components for such a system and there is no single configuration that performs well for all kinds of workloads. All the design and experimental results show that Active Harmony is a feasible and useful tool in performance tuning.

TOWARDS AUTOMATIC PERFORMANCE TUNING

By

I-Hsin Chung

Dissertation 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
Doctor of Philosophy
2004

Advisory Committee:
Professor Jeffrey K. Hollingsworth, Chair
Professor James F. Drake Jr., Dean's Representative
Professor David M. Mount
Professor Alan L. Sussman
Professor Peter J. Keleher

© Copyright by
I-Hsin Chung
2004

Dedication

To my parents – B.T. Chung & C.C. Lu

and

To my wife – H.F. Wen,

For all their love and support.

To my daughter – T.T. Chung,

I hope she has a long and happy life.

Acknowledgements

I would like to thank my advisor, Dr. Jeffrey Hollingsworth, for his help and guidance.

I would also like to thank my fellow students and members of our research group, Bryan Buck, Ray Chen, Jeffrey Odom, Nickolas Rutar, Vahid Tabatabaee, Mustafa Tikir, James Waskiewicz and Chadd Williams, for their help.

Table of Contents

Dedication ... ii
Acknowledgements ... iii
Table of Contents ... iv
List of Tables ... vi
List of Figures ... vii
List of Figures ... vii
Chapter 1: Introduction ... 1
Chapter 2: Related Work ... 6
2.1. Performance Tuning and Steering ... 6
2.2. Performance Characterization, Modeling and Benchmarking ... 11
2.3. Optimization Algorithms ... 14
2.4. Performance Tuning and Management for Large-scale Systems ... 15
2.5. Experiment Design and Parameter Analysis ... 17
Chapter 3: Active Harmony ... 19
3.1. Active Harmony Runtime Tuning System ... 19
3.2. Resource Specification Language ... 20
3.3. The Harmony Parameter API ... 25
3.4. Parameter Tuning Algorithm ... 27
3.5. Summary ... 30
Chapter 4: Library Specification Layer ... 31
4.1. Library Specification File ... 32
4.2. Algorithm Selection ... 37
4.2.1. Matrix Inversion ... 37
4.2.2. Table Abstraction ... 38
4.2.3. Compress Library ... 40
4.3. Discussion ... 45
4.4. Summary ... 46
Chapter 5: Smarter Tuning ... 48
5.1. Historical Data and Request Characterization ... 48
5.1.1. Concept ... 48
5.1.2. Performance Estimation ... 53
5.1.3. Synthetic Data Experiments ... 56
5.2. Improved Search Refinement ... 58
5.2.1. Concept ... 58
5.3. Summary ... 61
Chapter 6: Scalability – High Dimensional Search Space ... 62
6.1. Prioritizing Parameters ... 63
6.1.1. Concept ... 63
6.1.2. Sensitivity Experiment ... 64
6.2. Parameter Duplication ... 67
6.3. Parameter Partitioning ... 69
6.4. Parameter Restriction ... 70
6.5. Summary ... 72

Chapter 7: Cluster-Based Web Service System 74
7.1. Cluster-Based Web Service System 74
7.2. TPC-W Benchmark 76
7.3. Environment 78
7.4. Impact of Varying Workload 79
7.5. Utilizing Historical Data 84
7.6. Improved Search Refinement 86
7.7. Parameter Sensitivity 87
7.8. Putting together 90
7.9. Cluster Tuning 91
7.10. Automatic Cluster Reconfiguration 93
7.11. Summary 98
Chapter 8: Scientific Programs Tuning 99
8.1. PETSc Library 99
8.2. Parallel Ocean Program (POP) 103
8.3. GS2 106
8.4. Summary 112
Chapter 9: Conclusion 113
Appendix A: Performance Modeling with Synthetic Data 116
Bibliography 118

List of Tables

Table 1: TPC-W benchmark workloads 77
Table 2: Experiment environment 79
Table 3: Tuning results for different workloads 82
Table 4: Tuning process with and without prior histories 85
Table 5: Tuning process with and without improved search refinement 86
Table 6: Tuning process using original and improved Active Harmony 90
Table 7: Performance for different methods for cluster tuning 92
Table 8: Variable description 93
Table 9: Parameter changes through iterations 105
Table 10: Parameter tuning 106
Table 11: GS2 tuning result for benchmarking run 109
Table 12: GS2 tuning result for production run 109

List of Figures

Figure 1: Active Harmony automated runtime tuning system.................................. 19
Figure 2: The RSL language: (a) Application description; (b) Resource description. 23
Figure 3: Harmony user interface. 25
Figure 4: Changes required for a typical application.. 26
Figure 5: Possible outcomes for a simplex method step.. 29
Figure 6: Library Specification Language example.. 36
Figure 7: Matrix inversion test case.. 38
Figure 8: 2-D table with time metric... 39
Figure 9: 2-D table with memory space metric 40
Figure 10: LHa: changes of buffer size.. 42
Figure 11: LHa: buffer size and performance after tuning 43
Figure 12: zlib: changes of buffer size.. 44
Figure 13: Buffer size and performance after tuning.. 45
Figure 14: Different step *d*.. 45
Figure 15: Two stages of tuning (a) Training (b) Actual running 49
Figure 16: Data analyzer 52
Figure 17: Function shape and triangulation 53
Figure 18: Triangulation estimation for configuration with two parameters............. 54
Figure 19: Tuning using different experiences 57
Figure 20: Improved search refinement for configuration with two parameters 59
Figure 21: Tuning mechanism evaluation.. 61
Figure 22: Parameters sensitivity of the synthetic data 65
Figure 23: Tuning using only *n* most sensitive parameter(s) using synthetic data..... 66
Figure 24: Parameter duplication.. 68
Figure 25: Search space reduction by parameter restriction 70
Figure 26: Improved Resource Specification Language syntax example.................. 71
Figure 27: Multi-tier architecture.. 75
Figure 28: Applying best configuration after 200 iterations to different workloads .. 80
Figure 29: Tuning responsiveness to the changing workloads 83
Figure 30: Parameter sensitivity in the cluster-based web service system 88
Figure 31: Tuning using only *n* most sensitive parameter(s) of the cluster-based web service system 89
Figure 32: Reconfiguration algorithm for external tuning... 94
Figure 33: Reconfiguration experiment results... 97
Figure 34: Matrix decomposition.. 100
Figure 35: Computation distribution... 102
Figure 36: Block dimension tuning... 104
Figure 37: GS2 layout tuning with different environment....................................... 107
Figure 38: Performance distribution for GS2 configurations 111
Figure 39: Performance distribution 117

Chapter 1: Introduction

Applications are no longer monolithic programs written for a specific purpose. Instead, most software today makes extensive use of libraries and re-usable components. This approach generally results in software that is faster to build and more modular. However, one problem with this approach is that the various libraries used by an application are not tuned to the specific application's need. In addition, the applications are frequently used in very different ways. For example, different users may employ a single commercial simulation application for radically different types of simulations. As a result of this reuse of software, applications may not run well in these varied environments.

Another trend is Grid [30] computing. It suggests that the resources of many computers can be cooperatively managed as a collaboration toward a common objective. The transient, rarely repeatable behavior of Grid computing environment indicates the need to replace standard models of post-mortem performance optimization with a real-time model, one that optimizes application and runtime behavior during program execution. To try to address the needs of this type of computing environment, the Active Harmony system was developed to allow libraries and applications to expose tunable parameters.

Active Harmony is an infrastructure that allows applications to become tunable by applying very minimal changes to the application and library source code. This adaptability provides applications with a way to improve performance based on current configurations with observed performance results. The types of things that can

be tuned at runtime range from parameters such as the size of a read-ahead buffer to what algorithm is being used (e.g., heap sort vs. quick-sort).

A library is a collection of related code that can be used by many programs. Large complex computer programs nowadays are built from modules and libraries. This method helps programs to be developed incrementally from reusable parts. The programmer can develop, debug, and test individual parts separately and then integrate them into the program. The reuse of the program library makes software development more efficient.

Frequently, multiple program libraries with the same or similar functionality coexist to serve requests with different characteristics or under different circumstances. Each individual program library may be specialized in serving requests with specific characteristics. For example, different sorting algorithms are appropriate to different situations due to the differences in the problems characteristics.

Another obvious example is the selection of data structures. The data structure used in a program can affect the performance dramatically. Take a 2-D table implementation as an example. Using a linked list will save memory space but increase search time. On the other hand, using arrays will reduce the search time but waste memory space. Besides, the properties of the data element will also affect data structure selection. It would make the selection more complicated if those characteristics change during the execution time. It would be helpful if selection of data structures can change at runtime, based on observed behavior.

The thesis of this dissertation is that automated performance tuning is useful and even critical in many applications. Furthermore, it is possible for programs to adapt themselves when the execution environment changes rapidly. To achieve this vision, we have refined the Active Harmony system. We have adapted the Nelder-Mead simplex method to handle the practical tuning requirements. A Library Specification Layer has been developed to tune multiple program libraries with the same or similar functionality.

From the experience learned, we saw the need to speed up the tuning process. We do so by making use of the experience we learned in the previous tuning process and by avoiding unnecessary bad performance oscillations during configuration exploration. Performance oscillations are caused by configurations with extreme values that lie on the boundary of the search space. Bad performance due to these oscillations can dominate the whole tuning process and thus make online tuning less practical. We need an intelligent way to utilize the characteristics of the requests and the experience accumulated. In the dissertation, we explain how the Active Harmony tuning server may make use of known information, such as historical data, about the system or application to be tuned.

A more sophisticated approach is needed to deal with large systems with numerous tunable parameters. Scalability becomes a critical issue as the problem complexity increases (i.e., more tunable parameters). The search space increases exponentially when the number of parameters increases. This makes the tuning process time-consuming. We present techniques to improve the process when tuning numerous parameters together. We also show how to decide the relative importance

of the parameters in advance (prior to a specific execution) so that Active Harmony can focus on performance critical parameters. By examining the relations among parameters, we can further reduce the search space. When parameters can be divided into performance independent groups (i.e., there is no interaction among groups), we tune each group separately. If each group "behaves" similarly, we can only tune a representative set of parameters to further reduce burden.

To understand the effectiveness of the Active Harmony tuning system, we first use synthetic data to evaluate the improvements made to the system. Then we apply Active Harmony to several practical applications including a cluster-based web service system and scientific programs to verify all the improvements we made. We show that the techniques we developed are practical and result in a faster, more stable tuning process.

Contributions

The main contributions for my research presented in this thesis are:

Active Harmony development

The tuning kernel within Active Harmony tuning server is improved so the tuning process is faster and the performance is more stable in the initial exploration stage. This is done by not using configurations with extreme values. Besides tuning, the Library Specification Layer [22] is introduced so different programming libraries with the same or similar functionality can be coordinated. This helps the application the select appropriate programming library to achieve better performance.

Smarter tuning

In order to speed up the tuning process, we improved Active Harmony to utilize historical data from log files to "train" the tuning server and to "prepare" the system or application being tuned. Understanding the characteristics of requests (e.g., workload for a web server) helps Active Harmony select the right historical data set as well as the right programming library. We also extend the Resource Specification Language to support functional relations among parameters. This helps to constrain the search space and thus speed up the tuning process.

Scalability

A parameter prioritizing approach is developed so Active Harmony or the user can separate performance critical parameters from those that are not. Techniques to divide parameters into groups are also developed so Active Harmony can either tune a representative set of parameters or have individual tuning servers for each group.

Applied to real applications

To verify the Active Harmony, we applied it to several practical applications including a cluster-based web service system [21] and scientific programs. With parameter tuning, the cluster-based web service system can improve throughput up to 16%. With smarter tuning, the tuning time can be reduced up to 80%. For a climate change modeling code, the simulation time can be reduced up to 17% and for GS2 (a plasma physics code), Active Harmony can make it run up to 3.4 times faster.

Chapter 2: Related Work

2.1. Performance Tuning and Steering

There are several projects that have been seeking to develop techniques to allow applications to be responsive to their available resources or to allow them to be tuned at runtime. Computational Steering provides a way for users to alter the behavior of an application during execution.

CUMULVS [32] (Collaborative User Migration, User Library for Visualization and Steering), developed at the Computer Sciences Group at Oak Ridge National Laboratory, is a software framework that enables programmers to incorporate fault-tolerance, interactive visualization and computational steering into existing parallel programs. The CUMULVS software consists of two libraries, one for the application program, and one for the visualization and steering front-end (called the "viewer"). It handles collecting and transferring distributed data fields to the viewers and oversees adjustments to steering parameters in the application. It also manages the dynamic attachment and detachment of multiple independent viewers to a running parallel application. In addition, CUMULVS provides a user-directed checkpoint/restart mechanism to enable users to integrate fault tolerance into a running parallel application.

Falcon [33] is a set of tools that collectively support on-line program monitoring and steering of parallel and distributed applications. It was developed at the Georgia Institute of Technology. Falcon's monitor specification consists of a low-level sensor specification language and a high-level view specification language. Falcon captures and analyzes on-

line information capture and analysis. It provides program steering with graphical displays of system information.

SCIRun [53] is a scientific programming environment that allows the interactive construction, debugging and steering of large scale scientific programs. The users can design and modify simulations interactively via a dataflow programming model. SCIRun enables scientists to design and modify models and automatically change parameters and boundary conditions as well as the mesh discretization level needed for an accurate numerical solution. The primary goal of SCIRun is to enable the user to interactively control scientific simulations while the computation is in progress. This control allows the user to vary boundary conditions, model geometries, or various computational parameters during simulation. SCIRun is designed to provide high-level control over parameters in an efficient way. This is done through graphical user interfaces and scientific visualization.

Active Harmony's approach is similar in that applications provide hooks to allow their execution to be changed. Many computational steering systems are designed to allow the application semantics to be altered (e.g., adding a particle to a simulation, as part of a problem-solving environment) rather than for performance tuning. Also, most computational steering systems are manual in that a user is expected to make the changes to the program. Active Harmony's goal is to improve the performance rather than alter the execution results.

One exception to this is Autopilot [59, 60], which allows applications to be adapted in an automated way. Autopilot (developed at the University of Illinois, Urbana-Champaign) integrates dynamic performance instrumentation and on-the-fly performance data

reduction with configurable, malleable resource management algorithms. It has a real-time adaptive control mechanism that automatically chooses and configures resource management algorithms based on application request patterns and observed system performance. The goal of the Autopilot project is the creation of an infrastructure for building resilient, distributed, and parallel applications. It uses sensors to extract quantitative and qualitative performance data from executing applications, and provides requisite data for decision-making. Artificial Neural Network (ANN) and Hidden Markov Model (HMM) are used for classification. Autopilot uses fuzzy logic to automate the decision making process. The actuators execute the decision by changing parameter values of applications or resource management policies of the underlying system. Active Harmony differs from Autopilot in that it tries to coordinate the use of resources by multiple libraries and applications. Besides, both the instrumentation using sensors and rule-based decision making require more domain knowledge for the program being tuned. Active Harmony tries to provide a tuning mechanism where little or no domain knowledge is required for tuning.

The ATLAS (Automatically Tuned Linear Algebra Software) [75] project provides automatically tuned software specialized in linear algebra libraries. They have developed a methodology for the automatic generation of highly efficient basic linear algebra routines for each microprocessor. By using a code generator that probes and searches the system for an optimal set of parameters, it can produce highly optimized matrix multiply routines for a wide range of architectures. The difference between ATLAS and Active Harmony is that our work focuses on general applications that use program libraries rather than that of a specific library.

John Mellor-Crummey, et al. [47] investigates using data and computation reordering to improve memory hierarchy utilization for irregular application in which the data access pattern is unknown at compilation time. Besides just moving data closer to where it is used, the paper also applies space-filling curves to tune for multiple levels of cache even when the size of the caches is unknown. A data reordering involves changing the location of the elements of the data, but not the order in which these elements are referenced. A computation reordering involves changing the order in which data elements are referenced, but not the locations in which these data elements are stored. For two particle codes studied, the most effective reordering reduced overall execution time by a factor of two and four, respectively. Preliminary experience with a scatter benchmark derived from a large unstructured mesh application showed that careful data and computation ordering reduced primary cache misses by a factor of two compared to a random ordering.

Another approach is application level scheduling. AppLeS [12] allows applications to be informed of the variations in resources and presented with candidate lists of resources to use. In this system, applications are informed of resource changes and provided with a list of available resource sets. Then, each application allocates the resources based upon a customized scheduling to maximize its own performance. The Network Weather Service [77] is used to forecast the network performance and available CPU percentage to AppLeS agents. Active Harmony differs from AppLes in that we try to optimize resource allocation between multiple libraries and applications, whereas AppLes lets each application or library adapt itself independently. In addition, by providing a structured interface for applications to disclose their specific preferences, Active Harmony will encourage programmers to think about their needs in terms of options and their

characteristics rather than as selecting from specific resource alternatives described by the system.

The Odyssey project [50] developed at the University of California at Berkeley focuses on resource awareness at the application level. In this system, applications are informed of resource changes and provided with a list of available resource sets. Then, each application allocates the resources based upon a customized scheduling to maximize its own performance. Odyssey uses Fidelity as a metric; fidelity refers to changes in quality of the produced output. The metric is data dependent. For examples, with video, Fidelity might measure image clarity or compression rate. At all levels of service Fidelity must be pre-computed and are available at the server. Odyssey only deals with half of the problem. It only handles read operations; it does not concern itself with issues like reintegration, and collaboration with other systems.

Dome [8] is another parallel programming model which supports application-level adaptation using load balancing and checkpointing. While the load balancing for the different CPU and network performance is transparent, the programmers are responsible for writing suitable checkpointing codes using provided interfaces.

Kappa-Pi [19] is an automated performance analysis and tuning project at the Universitat Autónoma de Barcelona. It tries to give parallel programmers some aid when analyzing the performance of their applications. The basic principle of the tool is to analyze the efficiency of an application and provide the programmer some indications about the most important performance problem found in the execution. It helps to detect bottlenecks and provide hints to the developer. The static approach is based on the trace

files and source code. Dynamic Kappa-Pi utilizes an application model and a static call graph to provide "on the fly" analysis of runtime performance data.

The Nimrod/O project [5] tries to reduce the search space for engineering design. Nimrod/O allows a user to run an arbitrary computational model as the core of a non-linear optimization process. Nimrod/O allows a user to specify the domain and type of parameters to the model, and also to specify which output variable is to be minimized or maximized. It applies multiple tuning algorithms including Simplex, P-BFGS, Divide and Conquer, Simulated Annealing. The problem involves computing the shape and angle of attack of the aerofoil that maximizes the lift to drag ratio. The design for the aerofoil is an optimization program that needs to search for the global optima instead the local optima in a large search space. They demonstrate their idea is more flexible and delivers better results than a program that was developed specifically for the problem. They also show that it takes less time to deploy the tool for a new problem and it requires no software development. The Active Harmony project focuses on performance issues. Therefore, global optima are not always required for performance tuning (configuration searching). In other words, finding the configuration with best performance is not a must. Operating points (configurations) on local optima are still acceptable in most of cases if the performance is adequate.

2.2. Performance Characterization, Modeling and Benchmarking

Performance contracts [74] allow the level of performance expected of system modules to be quantified and then measured during execution. Application intrinsic metrics are performance values that are solely dependent on the application code and problem parameters. Examples of such metrics include messages per byte and average

number of source code statements per floating point operation. For *N* metrics, the trajectory through *N*-dimensional metric space is called the application signature. The execution signature reflects both the application demands on the resources and the response of the resource to those demands. Examples of execution metrics are instructions per second and messages per second. Vraalsen, F., et al. [74] project the application signature into a high-dimensional space using a scaling factor for each metric. Application signatures and projections define expected application behavior and runtime measurement capture actual behavior. The early vision of performance contracts includes software that uses a fuzzy rule set to quantify the level of performance expected as a function of available resources. The project plans to integrate fuzzy rule sets with Markov and time-series models to predict resource requirements and identify optimal resource allocation.

Using the application signature together with the convolution method helps to predict the performance more rapidly than simulation while scarifying some accuracy. Snavely, A., et al. [70] present a framework for performance modeling and prediction that is faster than cycle-accurate simulation, more informative than simple benchmarking, and is shown to be useful for performance investigations in several dimensions. The convolution method used is the computational mapping of an application's signature onto a machine profile to arrive at a performance prediction.

Predicting application performance on a given parallel system has been widely studied [6, 9, 14, 20, 28, 29, 37, 44, 62, 64]. Thomas Fahringer [29] introduces a practical approach for predicting some of the most important performance parameters of parallel programs, including work distribution, number of transfers, amount of data transferred,

network contention, transfer time, computation time and number of cache misses. The approach is based on advanced compiler analysis that carefully examines loop iteration spaces, procedure calls, array subscript expressions, communication patterns, data distributions and optimizing code transformations at the program level. It also considers machine specific parameters including cache characteristics, communication network indices, and benchmark data for computational operations at the machine level.

Performance prediction also extends to distributed systems. Kapadia, N.H. et al. [38] evaluate the application of three local learning algorithms (nearest-neighbor, weighted-average, and locally-weighted polynomial regression) for the prediction of the performance for a given set of runtime input parameters. This project focuses on the accuracy of the performance prediction. However, pursuing maximal predictive accuracy may not be appropriate given the variability in a grid computing environment.

The SPEC HPC2002 [3] suite uses benchmarks derived from real HPC applications. The benchmark suite is designed to measure the overall performance of high-end computer systems. It tests the performance for the computer's processors, interconnection system (shared or distributed memory), the compilers, the MPI or OpenMP parallel library implementation, and the input/output system. The suite consists of three benchmarks: SPEC CHEM2002 is based on a quantum chemistry application called GAMESS; SPEC ENV2002 is based on a weather research and forecasting model called WRF; and SPEC SEIS2002 represents an industrial application that performs time and depth migrations used to locate gas and oil deposits.

The NPB (NAS Parallel Benchmarks) [10] is a set of eight programs. This benchmark was designed to help evaluate the performance of parallel supercomputers and is derived

from computational fluid dynamics (CFD) applications. They consist of five kernels and three pseudo-applications.

The Livermore Loops benchmark (officially known as the Livermore Fortran Kernels) [46] was written by Frank McMahon of LLNL (Lawrence Livermore National Laboratory). The benchmark measures floating-point performance for a range of compute-intensive loops using a set of 24 (originally 14) Fortran DO loops extracted from physics simulation codes at LLNL.

2.3. Optimization Algorithms

The kernel of the performance optimization is the function minimization or maximization method. In this section we describe several optimization methods used in mathematical optimization and operations research. We are focusing on methods that do not need derivatives of a function.

For optimization in one dimension, Golden section search is an analogous version of the bisection method. It can be shown that if the new test point is chosen to be a golden section portion along the larger sub-interval, measured from the mid-point, then the width of the full initial interval will reduce at an optimal rate [58].

Parabolic interpolation is another function minimization without using derivatives. It uses three points to form a parabolic function. Based on the function, it uses its minimum point as the next test point.

Brent's rule [17] is a mix of the last two techniques: it uses the golden section when the function is not regular and switches to a parabolic interpolation when the function is sufficiently regular. In particular, it always tries a parabolic step. When the parabolic step is useless then the method uses the golden section search.

Direction set (Powell's) method [17] is another method used to find the minimum point in a *N*-dimensional search space. The basic idea behind Powell's Method is to break the *N* dimensional minimization down into *N* separate 1-dimension minimization problems. Then, for each 1-dimension problem a binary search is implemented to find the local minimum within a given range. Furthermore, on subsequent iterations an estimate is made of the best *directions* to use for the 1D search. This enables it to efficiently navigate along narrow "valleys" when they are not aligned with the axes.

Linear programming and the simplex method [23] are commonly used in optimizations. Linear programming is a class of mathematical programming models in which the objective function and the constraints can be expressed as linear functions of the decision variables. The simplex method is a general solution method for solving linear programming problems. It was developed in 1947 by George B. Dantzig with some modification for efficiency by D.M. Simmons [65]. It is an iterative algorithm that begins with an initial feasible solution, repeatedly moves to a better solution, and stops when an optimal solution has been found.

The simulated annealing method [40, 41] is another technique used in the optimization. The heart of the method of simulated annealing is an analogy with thermodynamics, specifically with the way that metals cool and anneal. It has been applied to design complex integrated circuits successfully. However, it cannot be applied easily for general purpose performance tuning since domain knowledge is required.

2.4. Performance Tuning and Management for Large-scale Systems

Others have discussed cluster-based web services with different performance metrics. Joel L. Wolf's work [76] proposed a scheme which attempts to optimally balance the

load on the servers of a clustered Web farm. They try to solve the performance problem by achieving minimal average response time for customer requests and thus ultimately achieve maximal customer throughput.

ADAPTLOAD [61] developed by Riska, A., et al. models a clustered web server as a front-end dispatcher and back-end nodes. They use an online algorithm to decide the share of the total workload for each node to achieve load balance. They treat back-end nodes as static while Active Harmony tries to configure the clustered system properly to achieve better performance.

Chen, et al. [7] use a reconfiguration mechanism to improve the throughput of a clustered system. Their focus is to avoid letting a small number of running jobs with unexpectedly large memory allocation block the execution of the majority jobs in the cluster. Active Harmony focuses on a general mechanism to improve overall system performance by several means.

Kalogeraki, et al. [10] migrate objects or jobs from hotspots in the cluster to improve the performance. Their goal is to achieve load balance while Active Harmony focuses on performance improvement.

Gage [13] focuses on load distribution to provide a performance guarantee for cluster-based Internet services. This involves support from network level while the Active Harmony only tries to tune the system to achieve better performance.

Levy, et al. [12] use a queuing model to analyze a cluster-based web service system. Based on the model built, they implement a prototype for a performance management system that is transparent to the system to be tuned.

The major difference between Active Harmony and the other large system tuning projects is that Active Harmony provides a general solution that does not require the user to have domain specific knowledge. The user does not need to analyze the details of the system components or build models.

The K42 project [71] is to develop a new high performance, open source, general-purpose operating system kernel for cache-coherent multiprocessors. K42 employs building-block technology to allow applications to customize and thus optimize the OS services they require. This is particularly important for applications, such as databases and web servers, where given the ability to control physical resources, they can improve performance. K42's design allows implementers on a particular architecture to choose what objects of the system should be customized for that architecture, and as a result allows the implementers to exploit any architecture specific features to improve performance. Active Harmony differs from K42 is that the tuning mechanism does not reside within the OS. In order to minimize the overhead and work with existing systems, Active Harmony provides tuning using a standalone server that communicates with the applications that are being tuned via network.

2.5. Experiment Design and Parameter Analysis

Performance often depends on more than one parameter such as the buffer size and number of threads waiting for requests. Proper analysis is required so the impact of each parameter can be isolated from that of others. Also, it is useful to know the relative importance of parameters to decide in which order to tune things.

Part IV of Jain’s book [36] describes techniques for designing a proper set of experiments for measurement or simulation. Types of experimental design discussed

include full factorial design and fractional factorial design. Fractional factorial design helps to estimate the contribution of each parameter to the performance as well as to isolate the measurement errors. It also discusses how to isolate the measurement errors as well as estimate confidence intervals for model parameters. There are numerous books on design and analysis of experiments including Mason, Gunst, and Hess [45]; Box, Hunter, and Hunter [15]; Dunn and Clark [27]; Hicks [34]; and Montgomery [48].

Plackett and Burman [56] described the construction of economical experimental designs with the number of runs required being a multiple of four (rather than a power of 2). Plackett-Burman designs are very efficient screening designs when only main effects are of interest. Yi, et al. [78] applied this technique on simulation methodology to 1) identify key processor parameters, 2) classify benchmarks based on how they affect the processor, and 3) analyze the effect of processor performance enhancements.

Box and Meyer [16] use a Bayes effect plot to display the probability that each effect is active according to the Bayesian analysis. This analysis is especially useful in saturated or near-saturated fractional factorial designs. It gives the probability that each effect is active when there are not enough degrees of freedom left to estimate error and perform F-tests on the effects.

Lenth [43] proposed a method that is appealing and popular because it utilizes an adaptive estimate of dispersion which should be more robust to the presence of a few large effects.

Chapter 3: Active Harmony

We first introduce the Active Harmony system and its main components: the Resource Specification Language, the Harmony parameter API and a parameter tuning algorithm. The Resource Specification Language is used to communicate between the tunable programs (e.g., application or library) and the Harmony tuning server. The Harmony parameter API was developed prior to this thesis. It is included here in order to aid understanding the rest of the thesis. The API is used to make programs tunable with minimum changes required. The parameter tuning algorithm is the kernel of the Harmony tuning server which will adjust the parameter values based on observed performance.

3.1. Active Harmony Runtime Tuning System

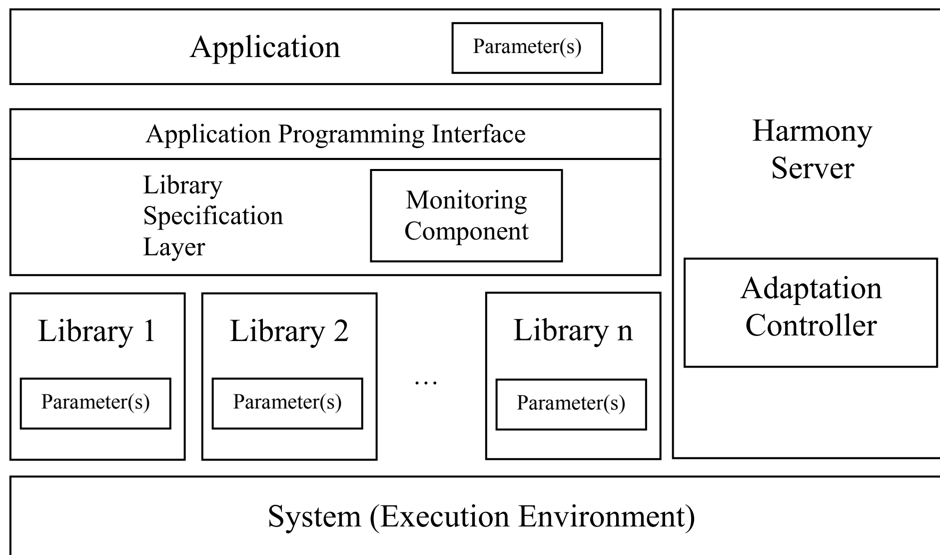


Figure 1: Active Harmony automated runtime tuning system

Figure 1 shows the Active Harmony automated runtime tuning system. There are parameters inside an application that are performance related. In other words, changing the parameter values will only affect the performance but not the correctness of operation. The Library Specification Layer provides a uniform API to library users by integrating those libraries with the same or similar functionality. This layer uses the Harmony Controller to select among different implementations of the library. The Library Specification Layer also monitors the performance of the libraries. The information is used to guide selection among different libraries. The details of the Library Specification Layer will be discussed in Chapter 4.

The Adaptation Controller is the main part of the Harmony server. The Adaptability component manages the values of the different tunable parameters provided by the applications and libraries. It adjusts the values of those parameters during program execution to achieve better performance for the system. For example, tunable parameters exposed by the application or programming libraries may include buffer sizes or number of processes. The Adaptation Controller is written in the Tcl scripting language.

3.2. Resource Specification Language[1]

The current Harmony Resource Specification Language (RSL) is improved from the initial version [35, 39]. It allows the user to describe more types of resource requirements, including resource type and time required (e.g., 20 MB memory for 9 seconds on hostname.cs.umd.edu). The RSL is implemented on top of the Tcl

[1] The resource specification language was originally developed by Cristian Tapus and later improved by the author with better functionality and bug fixing.

scripting language [52]. Tcl was chosen because it provides support for arbitrary expression and function evaluation. Tcl also provides the ability to add specific functions in C or C++ and export them as Tcl commands. Another reason for choosing Tcl was that it permitted the creation of a graphic interface through the use of the Tk toolkit.

The RSL allows applications to describe what resources they need and what options they have in the way they perform their function. Once the Active Harmony system has this information it processes the descriptions by simply calling a Tcl interpreter. Figure 2 shows the general form of an RSL specification for both an application and a resource.

The *harmonyApp* keyword precedes the description of an application. The application description contains tunable parameters, node descriptions and a "goodness" function (described below). A tunable parameter of the application, defined using the *harmonyBundle* tag, is characterized by type and range of values. The definition of applications and their options is one of the major changes that were made to the RSL as part of this thesis. In the initial version, the bundles defined mutually exclusive configurations of the application, with static values of parameters and resources intrinsically defined. In the current version, a *harmonyBundle* represents a variable of the application. A bundle can be used to define the range of allowable values for other bundles as well. For example, consider a program that has two parameters one that describes the maximum number of items to be buffered and a second that describes the desired number of items. The RSL specification for the allowable range of the desired number of items buffered can be expressed as a range

from 1 to the maximum number of items that can be buffered. Thus when the maximum is changed by the tuning system, the upper bound of the desired number of items will also be adjusted.

The resource requirements of the application are defined using the *node* tag. The characteristics of the nodes described by the application are matched against the resource description received from different machines that are part of the system. This way the Adaptation Controller can make decisions on where different applications will be run in the distributed system. The attributes of the *node* block are not restricted to those presented in Figure 2 below. Any attributes can be specified as long as they appear in both application and resource descriptions.

We also allow the user to locally define harmony variables that are not associated with application variables. This allows for cleaner descriptions, permitting reuse of expressions without having to duplicate them in multiple bundle definitions.

The final component of the RSL is a performance function. The performance function represents a metric of the performance of the application. The performance function is required to allow each application to define its own objective function such as throughput or response time.

```
HarmonyApp <Application Name> {
{ harmonyBundle <Parameter Name> {
  enum {<val1> <val2> … <valk>} |
  int {<min> <max> <step>} |
  real {<min> <max> <step>}
  [global]
} }
…
{ node <Name>
  {hostname <Host name> }
  {os <Operating System> }
  {seconds <Time needed> }
  {memory <Minimum memory in Mb> }
   …
  {replicate <value>}
} }
…
{ let <variableName> <funct. of bundleNames> }
…
{link <Node1> <Node2> <Bandwidth>}
{communication <fct of bundles>}
{obsGoodness <min> <max> [<#values>] [global]}
{predGoodness <min> <max>}
}
                                 (a)
HarmonyNode <Name>
  {hostname <Host name> }
  {os <Operating System> }
  {memory <Memory size> }
  {cpu <cpu speed> }
  {processors <# processors> }
  …
}
                                 (b)
```

Figure 2: The RSL language: (a) Application description; (b) Resource description.

The performance function is described using two different components. The *obsGoodness* tag describes an application-defined metric that is used by the tuning algorithm. For example, a scientific simulation might be described by a metric that indicates the time required to process a time step of data. Since a single value of the *obsGoodness* might not be indicative of the overall performance of the application, an optional *numValues* attribute can be defined that indicates the number of values to be collected, aggregated, and reported to the optimization algorithm. The need for collecting and aggregating different values of the performance function arose because some applications may require multiple samples (i.e. time steps) to react to a change

in a harmony parameter. The values are aggregated using an aggregation function written in Tcl.

Another important feature of the RSL is the *global* tag. This tag is used for bundles and for the performance function (*obsGoodness*). The significance of the global tag is as follows: different instances of the same application (i.e. processes of a SPMD program) can define a global bundle, which is used to simultaneously tune the values of the local bundles.

Application programmers can define their own aggregation function if the default one (average) is not appropriate for that application. The functions, written in Tcl, include: *aggregation_local* which combines multiple samples for a single process and *global_aggregation* which combines values from different processes or threads of a parallel program.

The *predGoodness* tag describes the second component of the performance function. This component is also characterized by a range, which specifies the expected range of values for the performance function. The *obsGoodness* tag is used to specify how to **measure** an application's performance, whereas the *predGoodness* is a mathematical expression of the **expected** performance based on an analytical model.

The Active Harmony system also includes a graphic console that plots the performance function and allows users to manually tune their application. Figure 3 shows a screen shot of the user interface. The box in the middle has three sliding controls that allow the user to adjust the values of the three parameters this application is exporting (tileSize, maxReads, and lowW). The graph at the bottom of

the picture shows the recent values for the "goodness" function and permits the user to browse the history of values as well as to change the thresholds that trigger the adaptation mechanism.

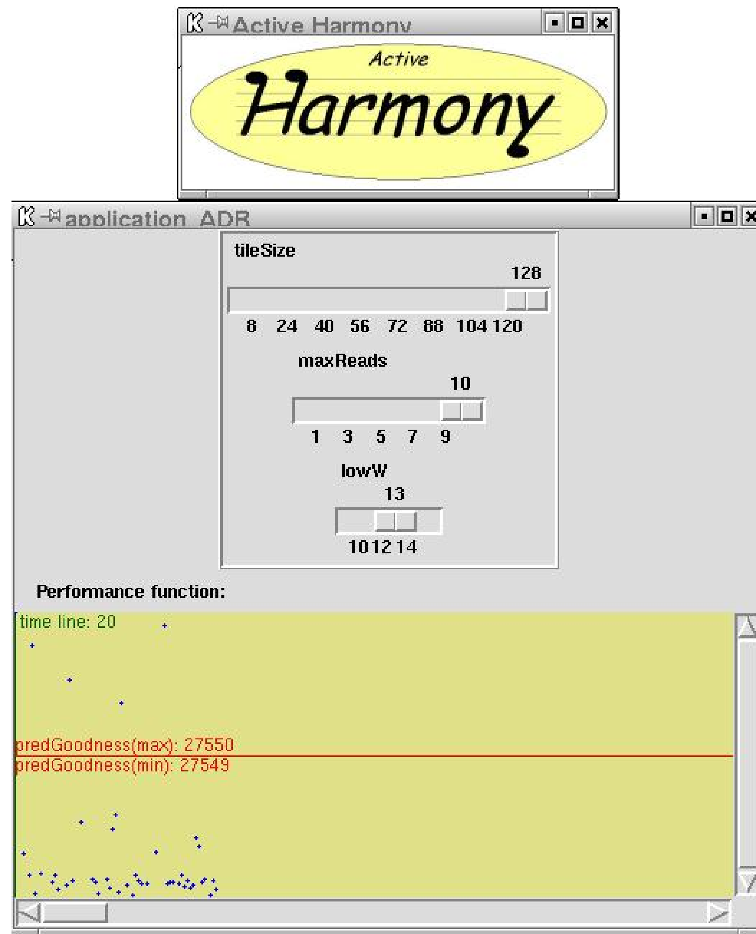


Figure 3: Harmony user interface.

3.3. The Harmony Parameter API

In order to allow the Harmony server to change library or application parameters, we have developed a library of functions that register tunable parameters and provide ways for the code to get the new parameters from the Harmony Adaptation Controller. The changes required to make a program tunable using this interface are relatively

small. For many programs we have "harmonized," the change amounted to less than 50 lines of code.

```
/* initialize */
harmony_startup(0);
harmony_application_setup_file("adr.tcl");

/* register tunable parameters */
low_watermark = (int*) harmony_add_variable("ADR", "lowW",VAR_INT);
max_nreads = (int*) harmony_add_variable("ADR","maxReads",VAR_INT);
tile_size = (int*) harmony_add_variable("ADR","tileSize",VAR_INT);

/* program main loop */
/* update tunable parameters' value */
harmony_request_all();
...
/* report performance result */
harmony_performance_update(performance_result);
/* end of program main loop */

/* finalize */
harmony_end();
```

Figure 4: Changes required for a typical application.

Figure 4 shows the changes made in the main program of a typical harmonized application. First the application has to register with the harmony server using the *harmony_startup* function. Next, it sends to the server the description of the application, which in this case is read from a file. This file contains the RSL specification for the application. This action is performed by the *harmony_application_setup_file* function. Next, the parameters specified by the application in its description have to be bound with variables in the main program. The *harmony_add_variable* function takes care of this. This function binds a harmony variable to an application variable. The application can then use this bound variable, which will be updated periodically by the Harmony system. Finally, the application calls the *harmony_end* function to un-register with the server.

One more change needs to be applied to the main loop of the program. Periodically (typically on a per time step or per query basis) the application sends a

value of the performance function to the harmony server by calling *harmony_performance_update*. The application then requests new values for the bound variables from the Harmony server invoking *harmony_request_all*.

3.4. Parameter Tuning Algorithm

An earlier version of the Active Harmony system [35] had a simple greedy algorithm to handle automatic selection of the appropriate parameters. However, for larger applications a more sophisticated algorithm is needed.

The problem of selecting good parameters requires finding a k-tuple in the value space determined by the values of the tuning parameters specified by the application, such that the application performs best. If we consider that better performance is represented by a smaller value of the performance function, then the goal is to minimize this function.

The problem is more complex due to the nature of the value space and that of the performance function. For example, a simple performance function could be the time spent by an application to complete a certain task. However, the value of this performance function depends not only on declared application parameters, but also on a number of external factors over which we have no real control. These external factors include, but are not restricted to, the current load of the machine, the operating system, application inputs or workload. Because of this, for fixed values of the tuning parameters we might get different values of the performance function even when performing the same task.

Even if we were able to fully isolate performance variation due to external factors, trying to find a minimum point in an arbitrary (and unknown) curve would require an

exhaustive search of the entire space of values by evaluating performance at each point. If the number of different values of each bundle is big this brute force approach is not feasible. Hence, we had to come up with heuristics to solve the problem. While the goal is to get the best performance possible, we are mostly interested in avoiding those k-tuples for which the performance is particularly bad. We have set this goal based on our experience in using the interface with a few test applications (including a database engine and parallel search algorithm). We found that there are frequently many points near the optimal point and that there is also often another region where the application performance is abysmal. Thus, trying to get into the good region even if we don't find the absolute best point achieves most of the benefit of finding the optimal solution.

We had several other goals for our minimization algorithm: 1) it should not require too many evaluations of the performance function and 2) it should avoid using gradients. Some optimization functions use first or second order derivatives to find the minimum or maximum point. This is not feasible for our case since the value space for tunable parameters may be discrete such as integer or Boolean variables.

The algorithm that we developed is based on the simplex method for finding a function's minimization [49]. The algorithm makes use of a simplex, which is a geometrical figure defined by k+1 connected points in a k-dimensions space. For the 2-dimensions space, the simplex is a triangle, and for the 3-d space the simplex is a non-degenerated tetrahedron. The Nelder-Mead simplex method approximates the extreme of a function by considering the worst point of the simplex and forming its symmetrical image through the center of the opposite (hyper) face. At each step a

better point, making the simplex move towards the extreme, replaces the worst point. The concept of the simplex method is shown in Figure 5. This figure shows the search for a minimum point in a three dimensional space. At the beginning of a step, there are four points: three points with low values are around the shaded triangle and the point with high value is at the left bottom corner of the pyramid as shown in Figure 5(a). Based on this performance result, the possible points will be explored by the tuning algorithm will be i) a reflection point, ii) a contraction point, and iii) a multiple contraction point as shown in Figure 5(b).

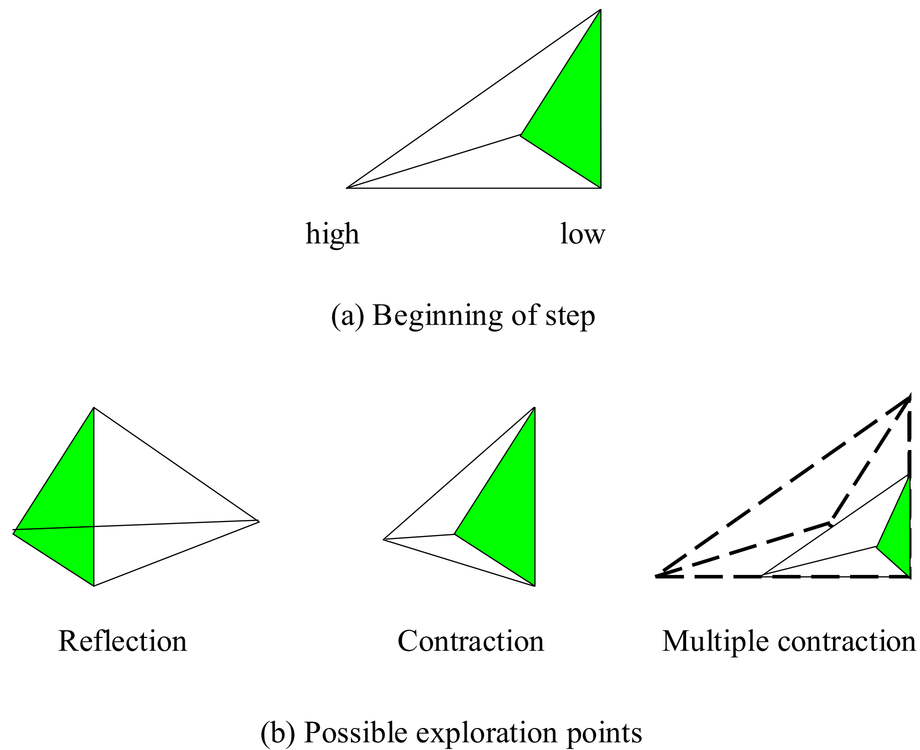


Figure 5: Possible outcomes for a simplex method step

The algorithm described above assumes a well-defined function and works in a continuous space. However, neither of these assumptions holds in our situation. Thus we had to come up with a method to adapt the algorithm to deal with this. Rather than modifying the algorithm to deal with this problem, we simply used the resulting values from the nearest integer point in the space to approximate the performance at the selected point in the continuous space.

3.5. Summary

In this chapter we introduced the Active Harmony system and its main components. Active Harmony provides an API so programs can become tunable with few changes. Applications can then specify resource requirements using the Resource Specification Language to communicate with the Harmony tuning server. We also discussed the algorithm used as the Active Harmony tuning kernel.

Chapter 4: Library Specification Layer

Most software today makes extensive use of libraries and re-usable components. This approach generally results in software that is faster to build and more modular. However, one problem with this approach is that the various libraries used by an application are not tuned to the specific application's need. Different library implementations with the same or similar functionality are used for different situations. As a result of this reuse of software, an application may not run as well as it could since it does not use the library implementation that is best for its requirements.

The Library Specification Layer is a thin, light-weight layer that is transparent to the application. It improves performance by automatically switching among different implementations of the same library with little overhead. When no library switching is necessary, the layer only redirects function calls and monitors the performance. When switching among libraries, the layer performs data structure or state transformation if necessary.

The role of the Library Specification Layer is to help the application use the most appropriate underlying algorithm. In other words, it helps the application to select the "right code" to execute. To achieve such a goal, it first characterizes the request from the application and monitors the performance of underlying program libraries. Based on the collected information, it will redirect the function calls to the selected underlying program library.

The performance metrics commonly used are the utilization of resources by the program library such as CPU time or memory space. Library developers can specify

multiple program library performance metrics in the Library Specification Language. The objective function used for tuning can be a single primitive performance metric such as the CPU time, or it can be a user supplied function of multiple primitive performance metrics such as (CPU time)×(Memory used), depending on the application. The underlying program libraries have to provide function calls in their API to support the measurement as well as the estimation of these performance metrics. Selecting the appropriate underlying program library is the role of the Adaptation Controller. In the current implementation, the Adaptation Controller tries to minimize the value of the first performance metric when searching for an appropriate underlying program library.

The Library Specification Language currently supports libraries written in both C and Fortran. The Library Specification Language generates header files that interpose glue code to allow libraries (or algorithms) with slightly different calling conventions to be integrated into a uniform API for application developers. It also provides the indirection to allow runtime switching among the different implementations. The runtime switching code includes the ability for a library writer to specify mapping functions that can change the underlying data structures (such as going from a dense to sparse matrix representation).

4.1. Library Specification File

The Library Specification Language is used to specify the relations between the API provided to application developers and the function call mapping of the underlying libraries. The syntax describing the API provided is similar to the prototype definition in the C programming language. It also defines the variables used

by the layer and the metrics used for the performance. The next part specifies the underlying program library including the mapping between the API provided and the underlying function calls. The last part describes the rules that are used to setup the decision agent.

An example of a program written in the Library Specification Language is given in the Figure 6. The language allows library developers to specify the mapping of function calls between APIs that will be exposed and the underlying program libraries. This is specified in the *%interface* section. The *%variable* section defines variables used to characterize the requests (e.g., whether the matrix is sparse or not) and other internal layer variables. The *%metric* section defines performance metrics used in evaluating the underlying library performance.

The *%method* section hooks up the Library Specification Layer with the underlying program library API. This section specifies the shared library file from the underlying program library. The *measurement* and *estimation* subsections define the program library performance measured and estimated by subroutines provided in the program library. In other words, the performance for the library in use can be obtained by calling the function specified in the measurement subsection while the estimated performance for libraries not in use can be obtained by calling the function specified in the estimation subsection. The *convertfrom* and *convertfrom_est* are optional sections. The *convertfrom* section specifies the steps that must be performed when switching underlying program libraries. For example, it may be required to transform the underlying data structure from a linked list to an array. If nothing has to be done when switching the underlying program library, this section may be omitted.

The *convertfrom_est* provides an estimated cost for switching between underlying program libraries. This information is used by the adaptation controller to decide if a switch is worth the conversion cost. The Library Specification Layer does not define the conversion process; it must be implemented as part of one of the underlying program libraries. In the example in Figure 6, the library for the linked list method supports the data structure transformation when the library is switched from the array method to the linked list method. Therefore, the layer will call the ll_fromary() function to perform and data structure transformation when switching.

The *%rule* section defines the rules used when selecting the underlying program library. This information is used to setup the Adaptation Controller. The *truthtable* subsection specifies what underlying program library should be used under certain conditions. The decision can be either automatic or manual. When it is set as *manual*, the Library Specification Layer is serving as a "consultant". It provides performance results and suggestions to the application. The application has to call the setmethod()[2] function explicitly to perform the switch. When it is set as *automatic*, the layer does the switch automatically.

The pre-compiler also generates associated utility functions from the Library Specification Language automatically. These functions include initialization and finalization of the layer, queries for underlying program library information, performance metrics, performance measurement and estimation, and conversion and its cost. The upper layer can use those utility functions provided by the Library Specification Layer API to have better control over the layer.

---

[2] The detailed utility functions are described in the library specification layer documentation. The Active Harmony Software is available at http://www.dyninst.org/harmony.

When the programming libraries are designed using object-oriented technology (where multiple libraries with exactly the same interface exist), the mapping for function calls would be redundant. However, the estimation and measurement of each library's performance as well as the conversion information are still needed to let the adaptation controller choose the appropriate underlying programming library.

```
%library Table
        /* Program Header */
        #include "localresource.h"
        ...
%interface c
        /* Function Prototype(s) */
        void init();
        int newtable();
        int insert(int table_id, int x, int y, void *e);
        ...
%variable
        /* Variable Definition(s) */
        int @sparse=1;
        void *data;
        ...
%metric
        /* Performance Metrics */
        int memory;
        float insert_delete_time;
        ...
%method
linklist      { /* Underlying Library */
        filename      libll.so
        function      { /* Mapping for functions */}
                init(): ll_init(&data);
                newtable(): ll_newtable();
                insert(): ll_insert(tid,x,y,e);
        ...}
        estimation {
                /* Mapping for functions used to estimate the performance */
                memory: ll_EstimateMem();
        ...}
        measurement {
                /* Mapping for functions used to measure the performance */
                memory: ll_memUsed();
        ...}
        convertfrom {
                /* Mapping for functions used to convert between method */
                array : void ll_fromary();
                ...}
        convertfrom_est {
                /* Mapping for functions used to estimate the cost for
conversion */
                array : ll_fromary_time();
        ...}
        }
        ...
%rule
predicate     {
                IsSparse: sparse==1
        ...}
truthtable {
                condition (IsSparse): linklist
        ...}
        decision manual
```

Figure 6: Library Specification Language example

4.2. Algorithm Selection

In this section, we evaluate the effectiveness of the Library Speciation Layer by applying Active Harmony to applications that utilize different libraries with the same or similar functionality. All of our tests were run on Redhat Linux with kernel 2.4.0 on a Pentium-III 667MHz with 384 MB main memory.

4.2.1. Matrix Inversion

The first set of program libraries consists of two matrix inversion routines from LAPACK [24]. The major characteristic of the matrix is a Boolean indicating if the matrix is triangular. If the matrix is triangular, using the dedicated triangular matrix inversion library will have better performance. Otherwise, a general matrix inversion library must be used. The result is shown in Figure 7. The library compares the triangular matrix by applying it to both the dedicated triangular inversion matrix library and the general matrix inversion matrix at the beginning. The workload used in the experiment is a mixed set of triangular and general matrices. For each request, the Library Specification Layer detects whether the supplied matrix is triangular and if so, the Library Specification Layer will invokes the matrix inversion library optimized for triangular matrices. Otherwise, the Library Specification Layer will just apply it to the general matrix inversion library.

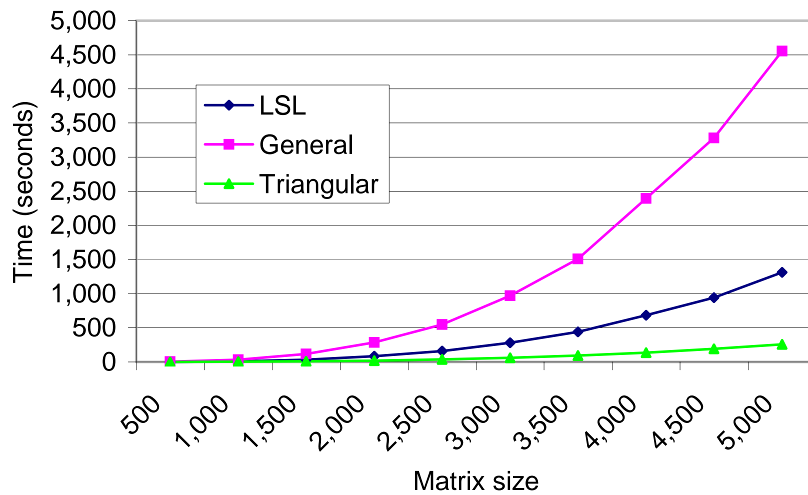


Figure 7: Matrix inversion test case

4.2.2. Table Abstraction

The second set of libraries consists of two libraries. Each of them implements a two dimensional array. The two dimensional array is used to store data elements similar to a table. The focus of this test case is the ability to select different data structures based on API usage patterns. Two program libraries are implemented using linked lists and arrays. Each approach has its advantages: linked lists take less memory space for storage but longer time for insert, delete, and search operations; arrays take more memory but are more efficient in insert, delete, and search operations.

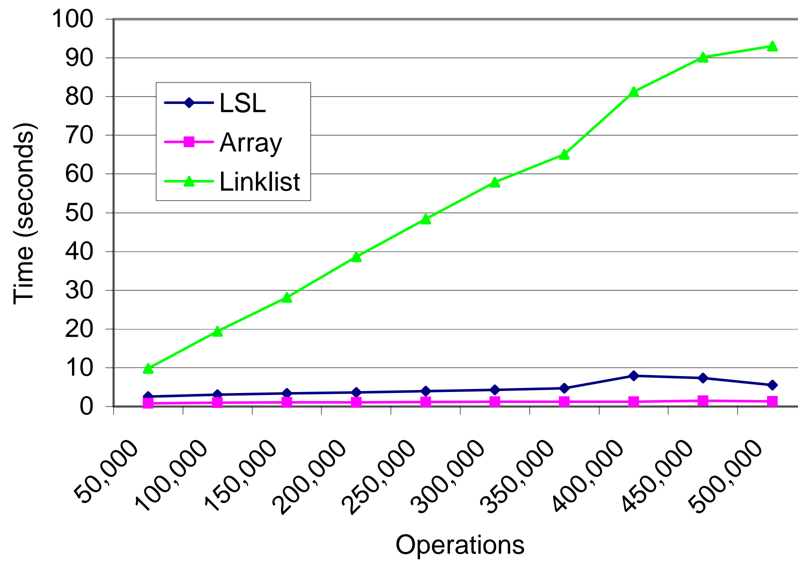


Figure 8: 2-D table with time metric

We ran the program on a set of random operations to store and retrieve data into and from the 2-D table. The first test uses the time to complete each operation as the performance metric. The result is shown in Figure 8. The version using the Library Specification Layer spends some time using the linked list library. Once it found that the performance of the array library is better than the linked list library, it will use the array library for the rest of the program execution. The second test uses memory utilization as the main metric. We repeated the experiment using the same workload. The result is shown in the Figure 9. As expected, the performance of the program with the Library Specification Layer is close to the performance of the program with the linked list version of the library. Typical applications built on top of the table abstraction would be scientific programs involving matrices. When the matrix is sparse and access time is not critical, the linked list method is preferred. On the other

hand, when the matrix is dense and access time is critical, the array method should be used.

Another difference between this test case and other test cases is the switch between underlying program libraries. In this test case, the Library Specification Layer has to perform data structure transformation from linked list to array or vice versa. This data structure transformation has to be supported by both underlying program libraries.

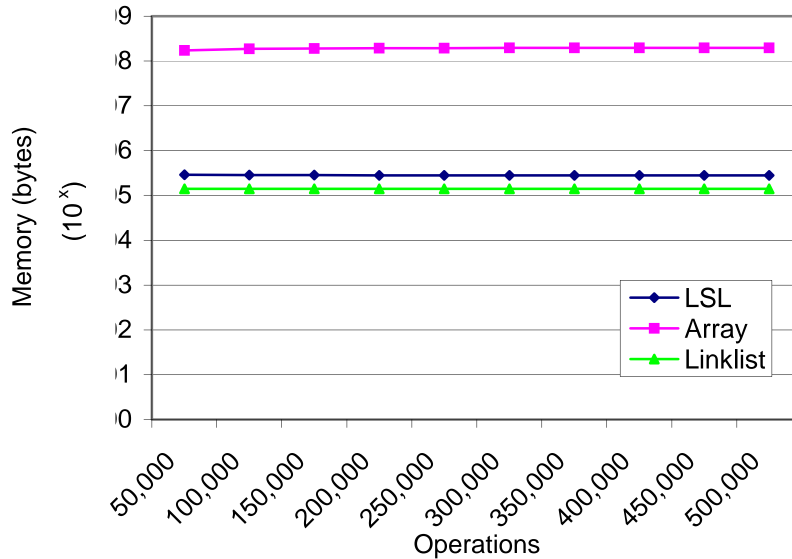


Figure 9: 2-D table with memory space metric

This experiment shows that proper selection can reduce runtime by a factor of 20 or more and space by two orders of magnitude for a set of randomly generated requests to store and lookup items in a table. By harmonizing the table, we can optimize the compression algorithms for either space or time.

4.2.3. Compress Library

In this experiment, we apply the Active Harmony automated runtime tuning system to a real compression library. The compression application uses two possible

underlying compression libraries: zlib[31] and LHa[51]. Both of these libraries are general-purpose, lossless data-compression libraries. The deflation algorithm used is a variation of LZ77 [79]. They both use hash table and binary trees, plus Huffman encoding to compress the data strings.

There are two performance metrics used in the experiment: time and size ratio. The time is the process time used to compress the data file. The size ratio is to compare the file size before and after the compression. Each library has its own tunable variables. The *BUFLEN* in the zlib adjusts the buffer used in reading the data strings. It will only affect the time to compress a file but not the compression ratio. The *MAXMATCH* in the LHa changes the buffer used but also affects the compression ratio. In the original code, those two variables were set to be compile time constants. We use the Harmony API [72] to make those two variables tunable during the application execution.

The big file being compressed (with predefined target size) is composed using files randomly selected from a UNIX file system. In the experiment, we focus on the automatic tuning when using a specific library. We set the library selection to be manual in the Library Specification Layer, and focus on tuning a library's parameters. Instead of optimizing a single performance metric, we selected an objective function that combines both space and time as metrics. The objective function is defined as the distance between a point (x,y) and the line y-0.015x=0 on a 2-dimensional space. Where x represents the buffer size and y represents the compression ratio. This objective function is chosen so the tuning will try to reduce buffer size while maintaining a similar compression ratio. The constant 0.015 is determined based on

the relative ratio of the buffer size and the compression ratio. The results show the tradeoff between the buffer size and the performance metric.

Figure 10 shows the tuning process for LHa compression library. The buffer size (compared to the default value) converges quickly after few iterations. Figure 11 shows the tuning results. The buffer size used is between 3% to 5% of the default one. The file size of the compressed file with tuning is 5% to 8.5% larger than that of the compressed file without tuning.

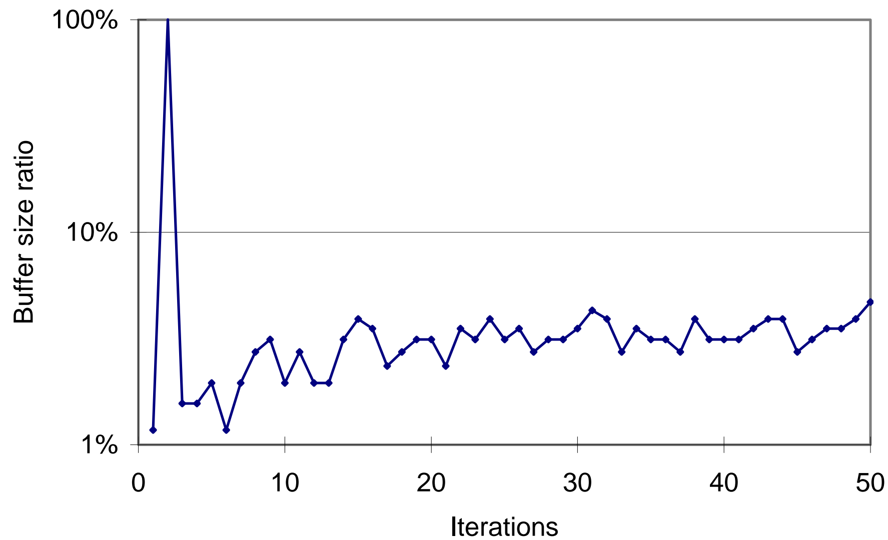


Figure 10: LHa: changes of buffer size

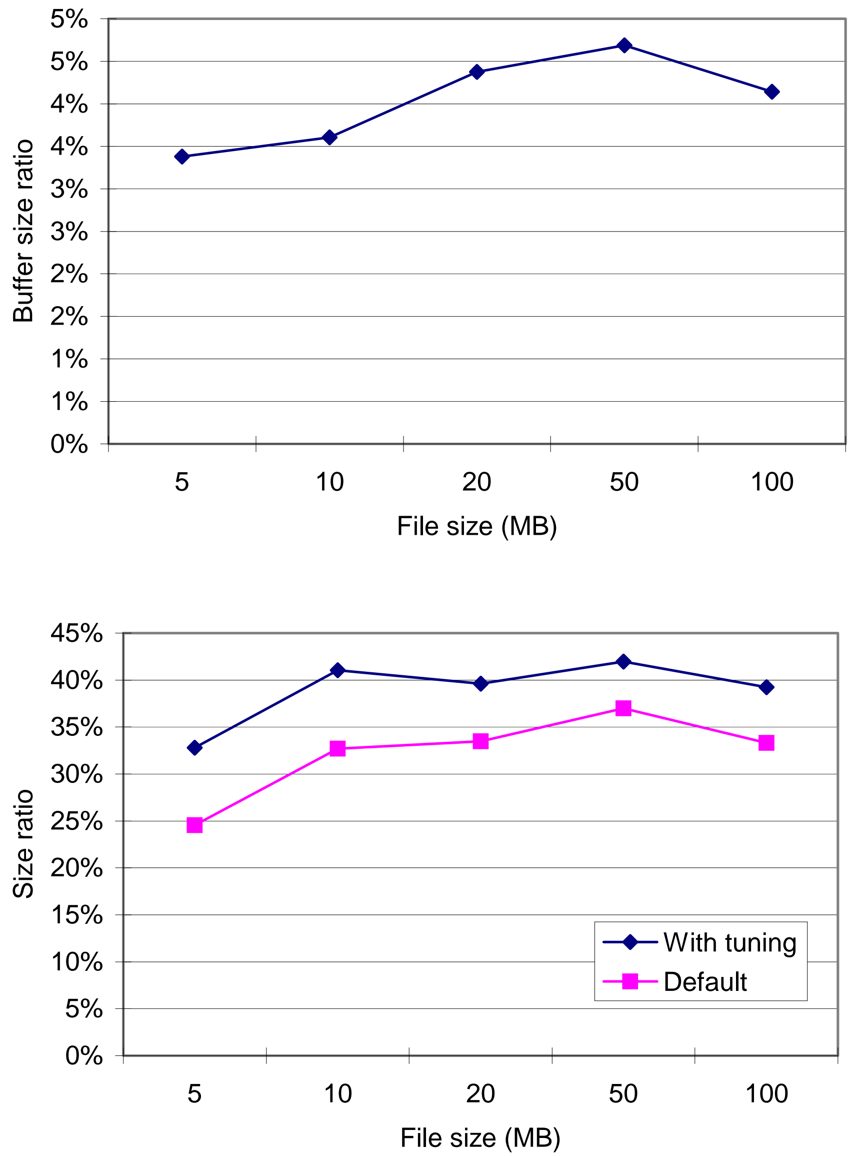


Figure 11: LHa: buffer size and performance after tuning

Figure 12 and 13 show the results when the Library Specification Layer chooses to use the zlib compression library. Figure 12 shows the size of the buffer used by the zlib compression library through the tuning iterations (Each iteration is one compression run). The buffer size converges after 15 iterations. Figure 13 shows the

tuning results. The buffer size is more than 100 times smaller than the original one while the process time increased about 15%.

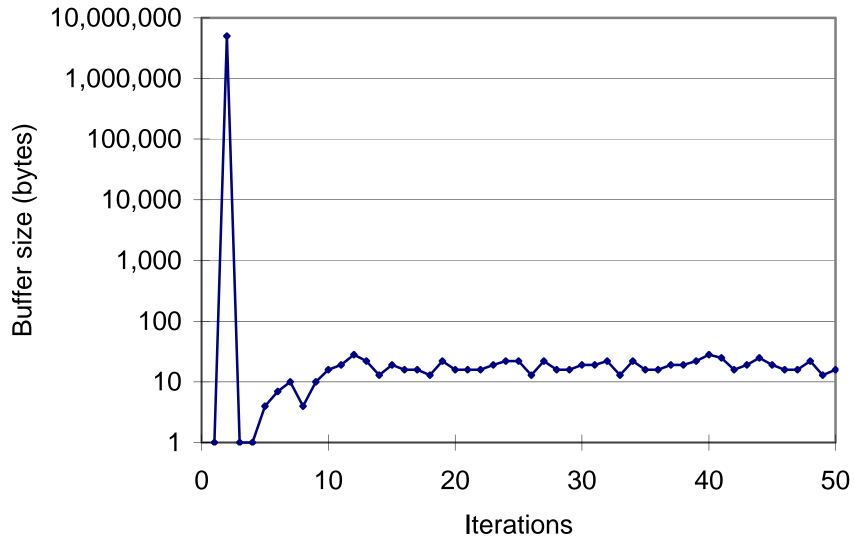
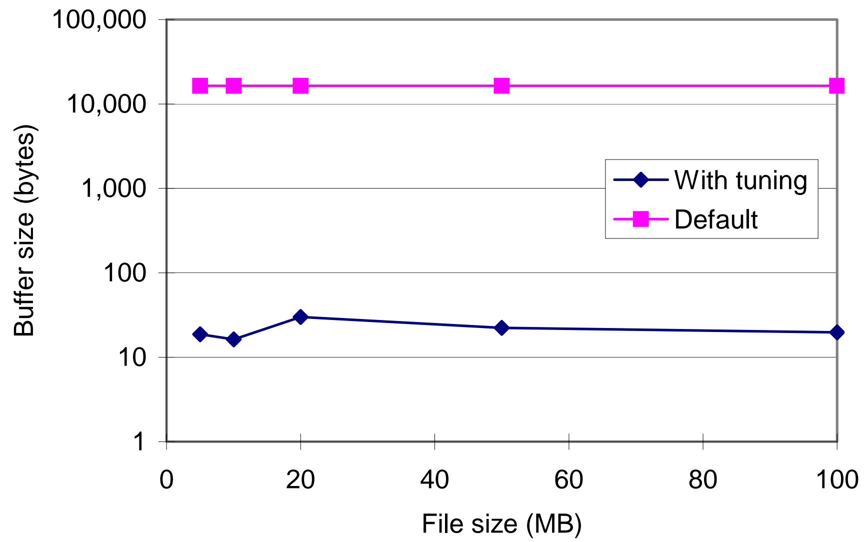


Figure 12: zlib: changes of buffer size



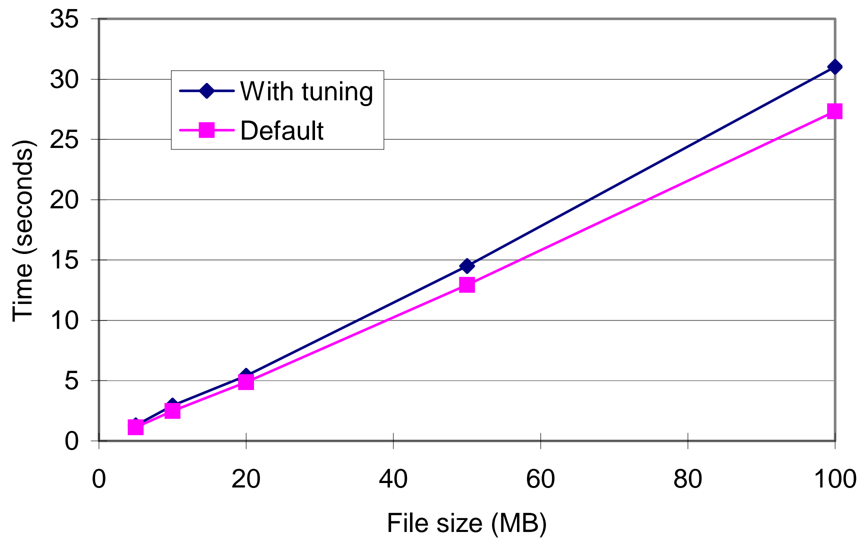


Figure 13: Buffer size and performance after tuning

4.3. Discussion

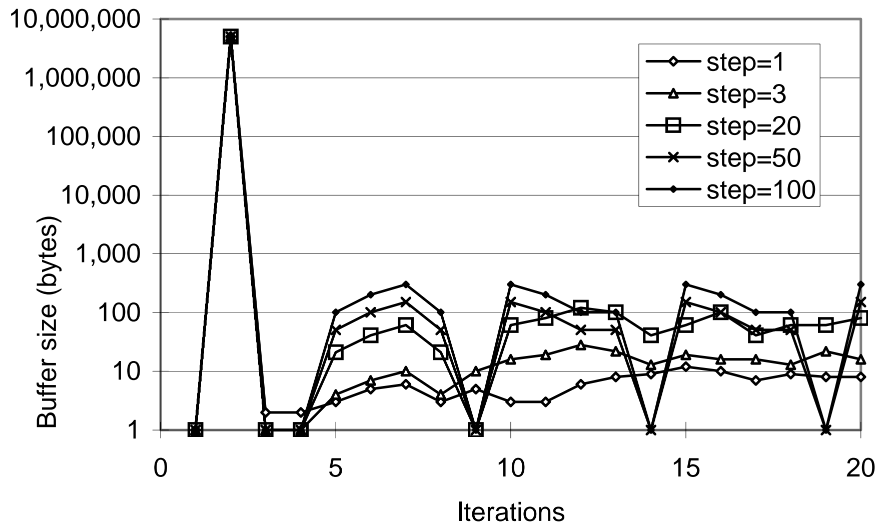


Figure 14: Different step *d*

There are two major factors that influence the tuning ability of the Harmony server. The first one is the selection of the objective function. The objective function

should have its minimum value at the desired operation point (e.g., execution time). A function that is "smooth" and with few "local minima" is preferred to help speed up tuning. The second influence on the search process is the "step" *d* used for the simplex tuning server. This is the minimum distance between the current value and the next value of the tunable variable. Figure 14 shows the tuning process with different *d*. In the figure, the x-axis is the iteration which is the number of compression tries and the y-axis represents the buffer size which is the tunable parameter. Each curve shows a different value for the step size *d*. If *d* is too small, the Harmony server is affected by the "noise" of the performance data since the tuning server is too sensitive to small variations in performance. Therefore in some cases, this could prevent the value of the tunable variable from converging. On the other hand, if *d* is too large, the result of the tuning may not be precise enough and the value of the tunable variable will keep oscillating. In the example shown in Figure 14, $d=1$ or $d=3$ are shown be reasonable choices – the tuning process is smooth and buffer size converges faster (compared to other *d* values).

4.4. Summary

In this chapter we presented a Library Specification Layer which helps program library developers expose multiple variations of the same API using different algorithms. The library has been integrated into the Active Harmony automated runtime tuning system. We presented the optimization algorithm based on the simplex method that we used to adjust parameters in the application and the libraries. We also described how the Library Specification Language helps to select the most appropriate program library to tune the overall performance. Based on a simple

architecture and with minimal changes to the source code of the applications, Active Harmony provides the user the ability to improve the performance of an application using an automatic search through algorithms or parameters at runtime. Another significant advantage provided by the Active Harmony system is the ability to make applications sensitive to the external factors and parameters that characterize the environment in which they are executed.

Finally, we present results that show how the system is able to tune several real applications. The results presented demonstrate that the Active Harmony Library Specification Layer can bring significant improvement to applications and it opens new ways to adapt applications to dynamic environments.

Chapter 5: Smarter Tuning

From our experience in previous work we found several drawbacks in our original tuning process in the Active Harmony system. First, the original Active Harmony has little or no knowledge about the system or the parameters to be tuned. This makes the tuning process lengthy since time can be spent tuning parameters that don't improve performance. Also, the tuning experience is not preserved across executions. In other words, when Active Harmony starts to tune a system, it does not utilize the experience gained from previous tuning of similar requests or workloads. Finally, in the original implementation, some of the initial configuration explorations test extreme values for the parameters. The performance for this initial stage is usually poor and the time spent in this period may dominate the overall tuning process. Unless the program being tuned is expected to run for a very long time, the benefit from the tuning may be limited. In order to overcome these problems, we 1) modified the Active Harmony to utilize historical data from previous tuning experience and 2) changed the search pattern used by the tuning kernel. With these improvements, the tuning time is reduced and the tuning process is smoother.

5.1. Historical Data and Request Characterization

5.1.1. Concept

During the tuning process, Active Harmony will keep a record of all the parameter values together with the associated performance results. When the system restarts, those parameter values and performance results can be fed into the Active Harmony tuning server. This is similar to a "review" or "training" stage. Therefore,

the Active Harmony tuning server may save time by not retrying all those configurations again from scratch. This is important since for many applications or systems, it may take a long time to measure the performance results for a single configuration. In order to "train" the tuning server with historical data, we have a separate stage that is different from the actual tuning stage. The training stage is usually much shorter than the actual tuning stage. In the training stage, it reads data from log files and does some computation where the actual running stage requires program executions. For example, in the cluster-based web service system tuning experiment described in Section 7.5, for each tuning the training stage is usually less than one minute while the actual running stage is about two hours. The details are shown in the Figure 15(a): training the Active Harmony tuning server and Figure 15(b): actually running the system.

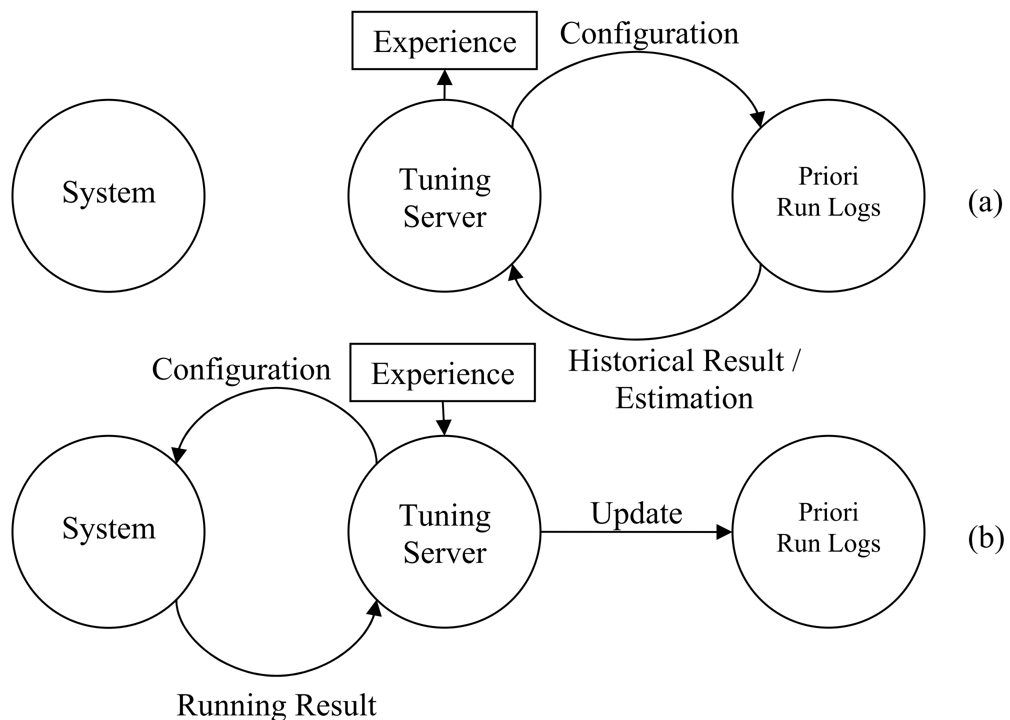


Figure 15: Two stages of tuning (a) Training (b) Actual running

In order to utilize the experience from the historical data, we must take the associated characteristics of the request (e.g., workloads for a web server) into consideration since the characteristics of the request also affect the performance. It may be useful when the system is currently serving requests with characteristic A and the tuning server was trained using historical data that are recorded when serving requests with characteristic A' (that is "closely" related to A). For example, in a cluster-based web service system we use a statistical method to count the frequency for each requested web page. The frequency distribution for the web pages is used to characterize the workload. Likewise, for a scientific system, better data distribution will yield better performance. If the input characteristic is similar to previous runs, the system should use the previous data layout as the starting point for tuning and this may reduce the tuning time.

In the original Active Harmony system implementation, input data are handled by the system and were processed without any "probing" or "observation" by the Active Harmony system. In other words, no characteristics of the input data were measured or recorded. During the runtime, the Active Harmony system tries to change the system configuration to achieve better performance only based on the performance monitored. It has no knowledge about the input and thus treats the system to be tuned as a "black box" every time. This makes the tuning process time consuming since it starts the tuning from the scratch and spends a tremendous amount of time trying different configurations.

We introduced a new component, the data analyzer, into the Active Harmony system so the system will be able to know the characteristics of the input data. The

tuning experience, with associated input request characteristics, will be accumulated in a database for future reference. When the input data is fed into the system, the data analyzer will first examine or observe a small number of sample requests to probe the characteristics of the input data. In order to accomplish such a task, the system to be tuned has to provide the method (function) that the data analyzer can use to characterize the input requests[3]. By using the method provided, the data analyzer can decide the characteristics of the input requests. For example, calling the function with the input matrix as the argument; the function will return the matrix structure (e.g., triangular, sparse … etc.) detected. Based on the known experience from the data characteristics database, the data analyzer makes the Active Harmony tuning server adjust the system more efficiently. For example, in a cluster-based web service system the data analyzer may use a statistical method to count the frequency for each requested web page. Later based on the frequency distribution for the web pages and previous experience, Active Harmony can adjust the parameters more properly.

For those input data with characteristics that have never been seen before, the Active Harmony tuning server may simply use the default tuning mechanism (i.e., no training stage). The tuning results will then be treated as a new experience and update the data characteristics database for future reference.

[3] In the current implementation, the user has to provide an application dependent probing function that returns observed characteristics (a vector of numbers) of input request. This function is used by Active Harmony tuning server for request characterization.

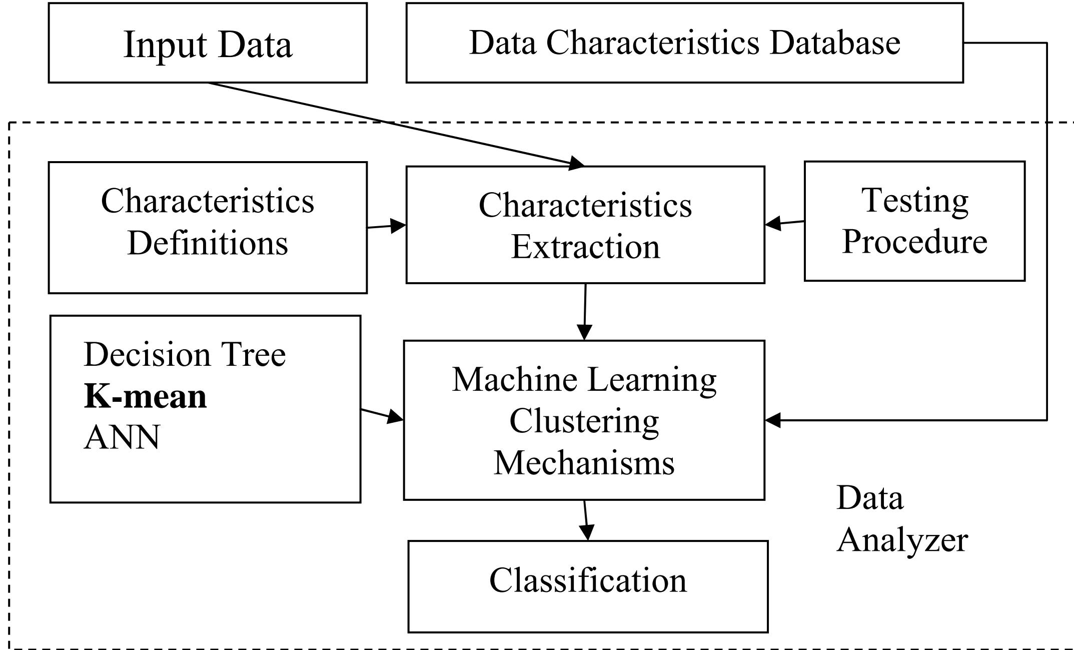


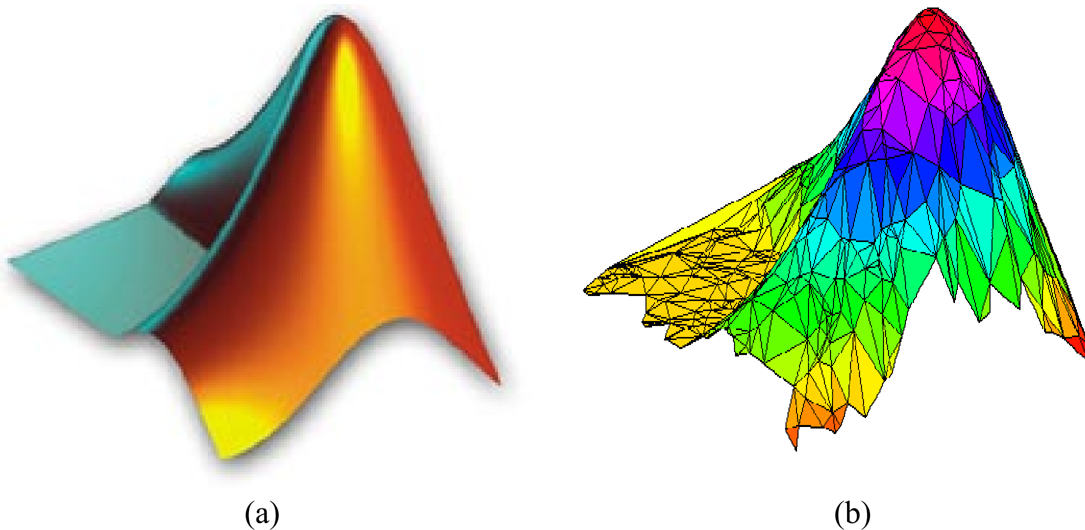
Figure 16: Data analyzer

The details of the data analyzer are shown in Figure 16. The data analyzer will first extract the characteristics using the given characteristics definitions and testing procedures (provided by the user for the system to be tuned) for the input data. After the characteristics of the input data are gathered, the data analyzer will then apply machine learning clustering approaches using a predefined method such as a decision tree together with known classes defined in the data characteristics database. In the current implementation, we use least square error [26] as the classification mechanism. In this approach, a vector $C_i=(c_{i1},c_{i2},\ldots)$ represents the ith workload characteristics stored in the experience database and $C_o=(c_{o1},c_{o2},\ldots)$ the observed workload characteristics. The classification algorithm returns j such that $\sqrt{\sum_k (c_{jk} - c_{ok})^2}$ is the minimum. Other classification mechanisms can easily be substituted depending on the requirements of the application. The classification

output is used as the key to retrieve the configurations from previous experience stored in the database. Then Active Harmony uses those configurations to setup the system being tuned.

5.1.2. Performance Estimation

Another important issue is what to do when the configurations and associated performance results needed for Active Harmony tuning server training are not available. In other words, if the parameter values in the historical data do not match those in the current configuration. In this case, it would be necessary to estimate the performance results at the target configuration that tuning server requires based on those known historical data.



(a) (b)

Figure 17: Function shape and triangulation[4]

---

[4] The function is generated using Matlab; it is used as the Matlab logo. It is the solution $u = u(x, y)$ of the wave equation $\Delta u + \Lambda u = 0$ on a L-shape domain. Δ is the Laplacian operator in two dimensions. The triangulation is generated using SaGA (Spatial and Geometric Analysis toolbox) developed by Kirill Pankratov.

In order to conquer this difficulty, we use triangulation with interpolation or extrapolation to estimate the performance at those “missing” configuration points. As shown in Figure 17, the idea of the triangulation is that: we first select vertices to form a simplex. A vertex in an N dimensional space represents a configuration with N parameters. The projection of the vertex on ith axis is the value for the ith parameter. A simplex in an N dimensional space consists of $N+1$ vertices. For example, a simplex in a two dimensional space is a triangle; a simplex in a three dimensional space is a pyramid. We then put the simplex in an $N+1$ dimensional space where the $N+1$th dimension is the associated performance for each vertex (configuration). We then use those $N+1$ vertices on the simplex to estimate the performance of the target vertex in an $N+1$ dimensional space.

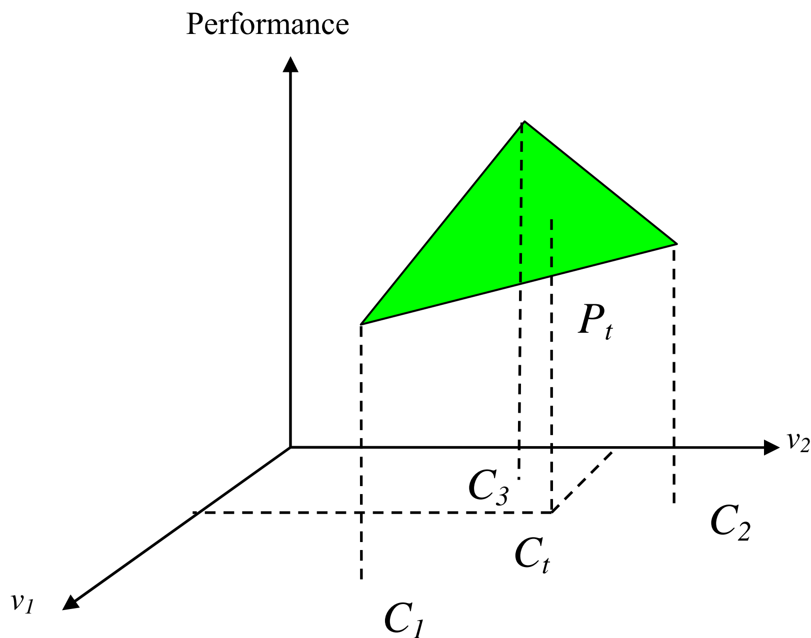


Figure 18: Triangulation estimation for configuration with two parameters

The example in Figure 18 shows how to use triangulation to estimate a configuration with two parameters. First we need to find three configurations C_1, C_2, C_3 and use their associated performance to form a plane in the three dimensional space. Then we use this plane to estimate the performance P_t at the target configuration C_t.

The algorithm is described as follows:

1. For a configuration with *N* parameters, find the "appropriate" *k* configurations (vertices) with associated performance results in the historical data.[5]
2. Let $C_i = \begin{bmatrix} c_{i1} & c_{i2} & \cdots & c_{iN} \end{bmatrix}$ be the *i*th configuration, where c_{ij} represents the *j*th parameter value of the *i*th configuration.
3. Let $A = \begin{bmatrix} C_1 & 1 \\ C_2 & 1 \\ \vdots & \vdots \\ C_k & 1 \end{bmatrix}, b = \begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_k \end{bmatrix}$, where P_i is the performance of the *i*th configuration.
4. Solve $x = A^{-1}b$; for an under- or over-determined system, apply the least square method to decide x
5. Calculate $P_t = \begin{bmatrix} C_t & 1 \end{bmatrix} x$.

[5] Here the appropriate configurations depend on the actual situation: those vertices may be close to the target vertex in the distance in the search space or close to the target vertex in terms of the time recorded in the historical data. This step is challenging since many issues need to be taken into consideration. For example, if the execution environment is static or does not change frequently, vertices close to the target vertex may be used for estimation; when the execution environment is changing frequently, we may need to use the latest vertices to estimate the target vertex. Currently our implementation uses vertices that are close to the target vertex.

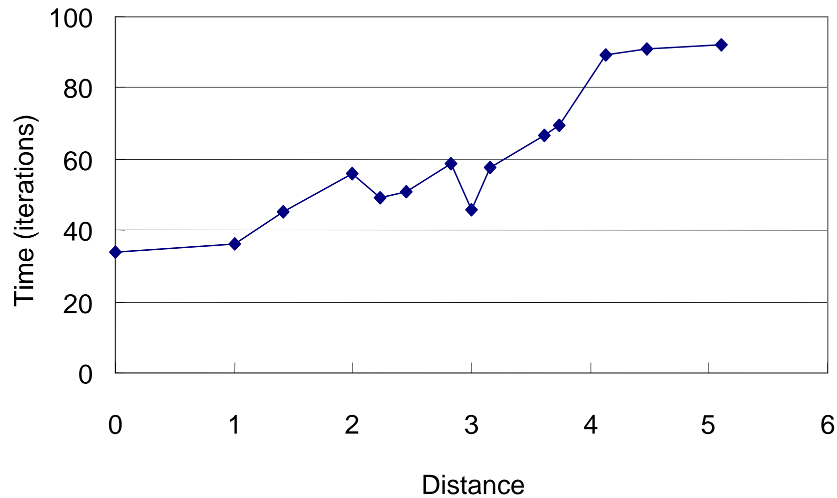
5.1.3. Synthetic Data Experiments

To evaluate the concept of utilizing historical data with request characterization, we first conduct a study using synthetic data for the experiments. The details for the synthetic data generation are given in Appendix A.

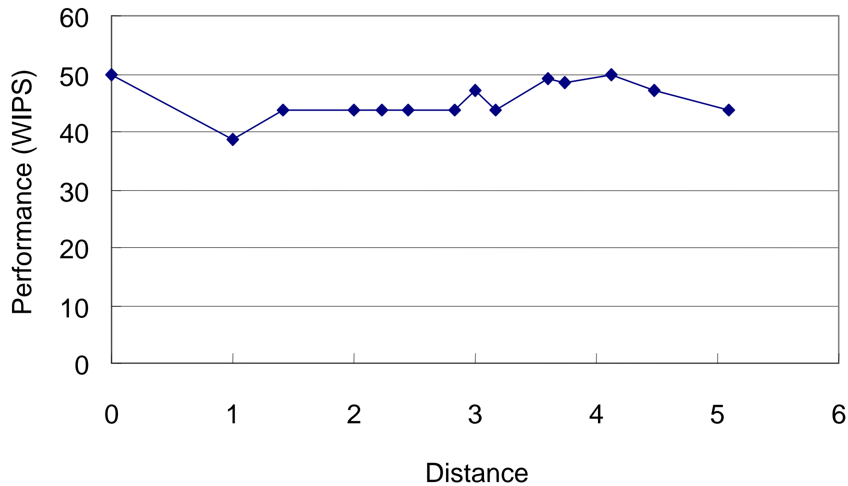
In order to test the effectiveness of performance tuning using historical data, we designed the following experiment: a system is facing a workload A. The data analyzer in the Active Harmony server first spends a few iterations to characterize the incoming workload and decides to use historical data workload A', where A' is the closest experience to A in terms of the characteristics (computed using techniques described in Section 5.1.1).

Figure 19 shows the relation between the experience workload A' and current workload A for synthetic data based on a web server workload. In both figures, the x-axis shows the distance between the current configuration A and the stored workload A'. Each workload is represented by a vector of numbers. The distance between two workload characteristics is calculated using normalized distance in the Euclidean space so that characteristic variables with a wide range of values are not given excessive weight. This data is again taken from synthetic data generated for a system like the cluster-based web service system presented in Appendix A. In Figure 19(a), when the characteristics of the historical data are close to those of the current workload, it takes less time to tune the system (in this example, a distance less than 4 should be close enough). The more they differ from each other, the longer for the Active Harmony to tune the system to achieve similar performance as shown in Figure 19(b). Not surprisingly, this result suggests that when tuning a system with

historical data (experience), one should choose to use historical data that is similar to the current workload. However, even when the distance from the previously observed system is quite large, the system eventually is able to achieve similar performance to cases where the difference are small.



(a) Tuning time



(b) Tuning result

Figure 19: Tuning using different experiences[6]

---

[6] Each experiment takes 200 iterations.

5.2. Improved Search Refinement

5.2.1. Concept

The original Active Harmony tuning server does a decent job in performance tuning. With few explorations, it can help the system or program being tuned find a fairly good configuration for operation. One problem for the original Active Harmony tuning kernel is the configurations to use for initial exploration. In the original implementation of the Active Harmony system, it takes k+1 iterations to explore the values for each of the k parameters. It will start to improve the system to be tuned at the k+2th iteration. The configurations used for those k+1 iterations are predefined. In particular, the original Active Harmony search will explore the extreme values of the k parameters in the k+1 iterations. This is due to the characteristics of the Nelder-Mead simplex method. However, from the experience we had in our previous work, we found that the system usually performs poorly with the parameters at the extreme values. Sometimes the time spent by the system on those configurations with poor performance dominated the whole tuning process. This makes the tuning results less useful compared to the time "wasted". In addition, for a lot of applications to be tuned, the tuning results for the parameter values are far from the extreme values. Consider the maximum number of connections for a web server, allowing only one process will make the system inefficient; allowing too many processes will cause the system thrashing. Only the number of connections that is compatible with the system's capacity will yield the best performance result. Another example is in a climate simulation program. In this application, the computing nodes are divided into groups. Each group of machines is responsible for part of simulation task (e.g., land, ocean…).

Using only one node for one task will often cause load imbalance and thus make the simulation inefficient. Instead, balancing the number of nodes to mach the computational complexity of each task will provide the best performance.

In order to solve this problem, we modified the tuning algorithm to replace predefined parameter configurations at extreme values with values that are closer to the current configuration but which will evenly cover the search space, as shown in Figure 20. The rectangle represents the allowed range for the parameter values. The circle represents a single configuration and the number inside is the order of the configuration to be explored. As shown in the Figure 20(a), the original Active Harmony implementation tries the extreme values for the parameters for the initial exploration. Figure 20(b) shows one possible alternative initial exploration configuration. In the current implementation, we are using configurations that are equally distributed in the whole search space. In other words, for each of *n* parameters, we increase *1/n* of its extreme values every time in the first *n* explorations.

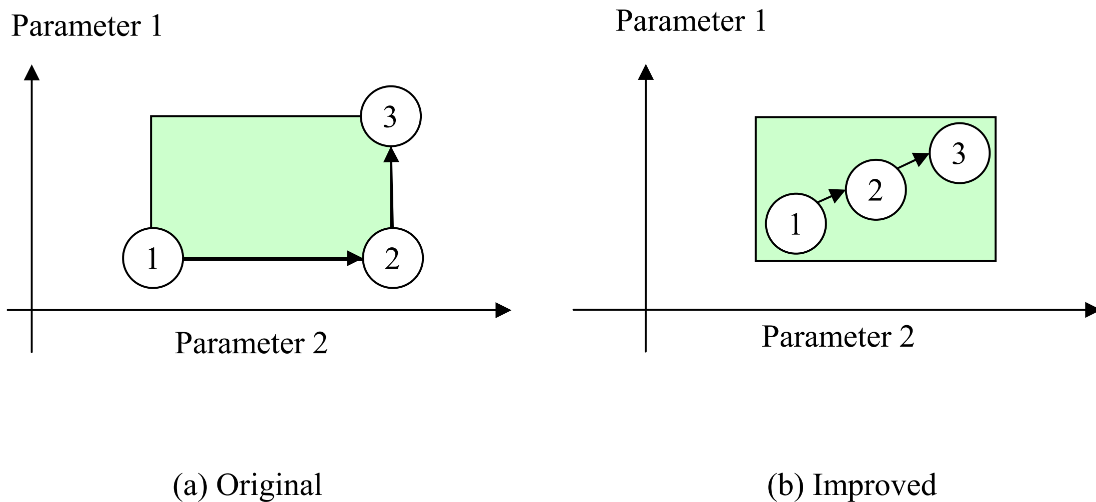


Figure 20: Improved search refinement for configuration with two parameters

Reducing the magnitude of performance oscillations (swings between very good and very poor performance) in the initial tuning process is important because what we care about in the tuning process is not just getting the best configuration, but also the performance of the system while getting there. In other words, the effort or cost when searching for a desirable configuration versus the performance at the resulting configuration should be taken into consideration. Note this is different than most optimization techniques which simply count the number of times the objective function is evaluated. As it is shown the hypothetical performance curve in Figure 21, tuning process A is better if we only look at the tuning result. Tuning process B is more stable if we consider the area below the line curve. Therefore, tuning process B may be more desirable in a practical application. For the performance tuning, we are always looking for a mechanism that not only makes the tuning fast but also makes the tuning process more stable with less performance oscillation.

The Nelder-Mead simplex minimization algorithm uses reflections and contractions when it explores the next configuration. Due to this characteristic, initial configurations used for exploration should be diverse and evenly distributed in the search space. Besides using the pattern shown in the Figure 20(b), we also have tried random configuration points with uniform distribution in the search space. However, this cannot guarantee an even distribution and thus did not improve the tuning process significantly. In future work, we plan to try some other patterns such as points decided by the K-mean algorithm [57].

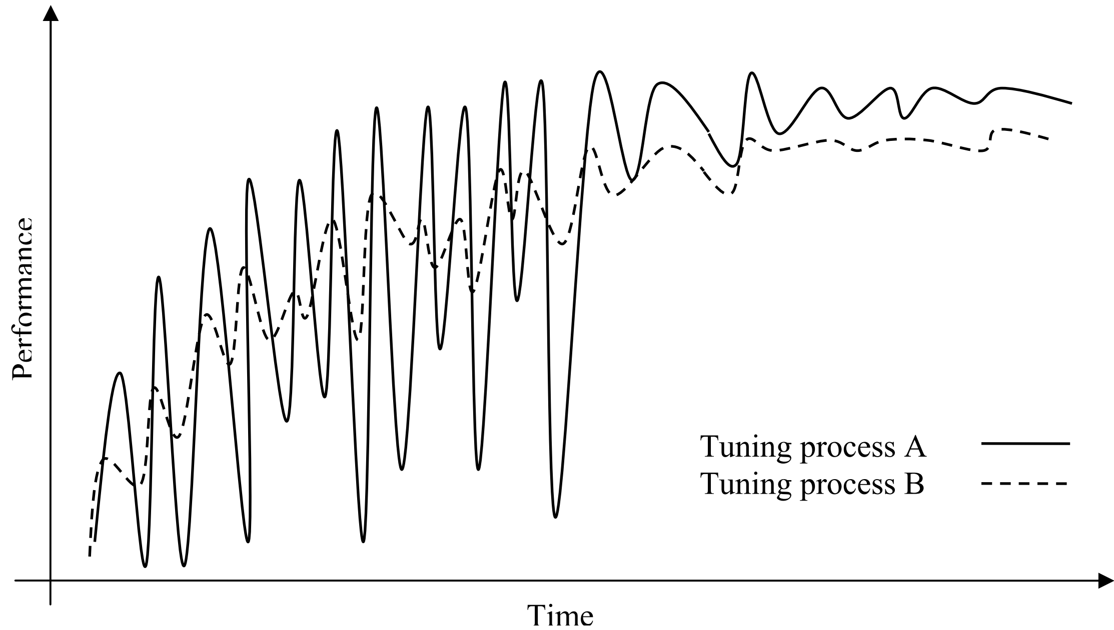


Figure 21: Tuning mechanism evaluation

5.3. Summary

In this chapter, we improved the tuning process by utilizing the historical data and making the tuning process smoother. Request characterization and application behavior help to decide which historical data should be used in the training stage of the tuning process. Improved search refinement helps to reduce the performance oscillations in the initial stage. With these improvements, the tuning time is reduced and the tuning process is smoother.

Chapter 6: Scalability – High Dimensional Search Space

When we apply the Active Harmony system to real systems, a practical issue is scalability. Tuning can be time-consuming due to the numerous parameters at each component in an application. As expected, it takes a long time for the Active Harmony tuning server to adjust numerous parameter values based on one performance result (e.g., throughput). In order to make the Active Harmony system capable of tuning numerous parameters, we improved the tuning with parameter prioritizing, parameter duplication, parameter partitioning, and parameter restriction. Parameter prioritizing helps us to focus on those parameters that are performance related. Parameter duplication tunes the same parameter on different locations concurrently (i.e., in a cluster-based web service system, two application servers may have the same parameter to control the number of connections). Parameter partitioning helps to tune separate parameters in parallel and Parameter restriction reduces the search space by observing the relations among parameters.

One major problem for tuning numerous parameters together is the size of the search space. For a system with 10 parameters where each parameter has 2 possible values, the size of the search space would be 2^{10}. In the previous implementation of the Active Harmony system, tuning using 10 parameters takes 11 initial explorations before it starts to improve the performance. Imagine a system with 1,000 parameters, the size of the search space would be $2^{1,000}$ and it would take 1,001 explorations to improve the performance. This approach would make tuning impractical since tuning would be so time consuming. Even if the values of the parameters will eventually converge, the configuration found may be out of date and thus useless. Also when

tuning some applications, even exploring one configuration could take a significant amount of time. For example, it may take 5 to 10 minutes to explore one configuration for a scientific simulation program, since each exploration requires running one or more time steps of the application.

6.1. Prioritizing Parameters

6.1.1. Concept

When tuning a system or application, it is important to identify those parameters that are affecting the performance from those that are not. For a large system or application with numerous parameters, it would be helpful to focus on the parameters that have greater impact on the performance rather than tuning all parameters at once.

We have developed a standalone software tool that provides the data required for prioritization. It takes possible parameters indicated by the user using the Resource Specification Language as the input. Each parameter will be specified with four values: minimum, maximum, default value and distance between two neighbor values. The distance between two neighbor values decides the number of sample points the software will test. The software tool tests the sensitivity for each of the parameters. For each parameter, the tool runs the applications with the possible values while the rest of the parameters are fixed with the default value. Assume $P_1, P_2, ..., P_n$ are the performance results with those different parameter values $v_1, v_2, ..., v_n$. We defined the sensitivity of a parameter to be $\left|\frac{\Delta P}{\Delta v'}\right|$, where $\Delta P = P_a - P_b, \Delta v' = v'_a - v'_b$, $P_a = \max_{i=1...n} P_i, P_b = \min_{i=1...n} P_i$. Also the parameter value is normalized (e.g., $v'_a = \frac{v_a - v_{\min}}{v_{\max} - v_{\min}}$).

The idea of this sensitivity evaluation is to understand the performance impact when changing one parameter. If the relative sensitivity value (compared to other parameters) for a parameter is large, we expect that changing the value of this parameter will affect the performance directly. Hence it should be considered with higher priority when considering changes to a configuration at runtime. On the other hand, if the relative sensitivity value is small, it has lower priority and may be discarded or used later in the tuning. We choose $\Delta v' = v'_a - v'_b$ rather than $\Delta v' = v'_{\max} - v'_{\min}$ so parameters that affect performance significantly within small portion of their valid range will be considered as highly sensitive. For some parameters, such as buffer size in a web server, the range of valid values may be large, but a significant performance change is only seen in a small range of values.

If we are tuning a large system or program with n parameters and k different possible values for each parameter. The search space for such a system or program will be huge (i.e., n^k). With help from parameter prioritizing, the Active Harmony system can focus on the performance critical parameters and leave the less important ones behind at the cost of $j \times n$, where $j \le k$ (smaller j may cause a less accurate result). This is helpful for a system or program with numerous parameters – a typical cluster-based web service system can have 50 or more parameters.

6.1.2. Sensitivity Experiment

To evaluate parameter prioritization we use the parameter prioritizing tool on a set of synthetic data. This provides a controlled environment to evaluate our approach. When the data was generated, we specified two out of the fifteen parameters to be performance irrelevant so changing the values of those two parameters will not affect

the performance at all. We also perturb the performance output from 0% to 25% randomly. This variance in measured performance is to model the reality that given exactly the same environment and input, the performance output will not always be the same for two different runs. Details of how the synthetic data was generated are in Appendix A. We evaluate our parameter prioritization with regard to this run to run variation in application performance. The result is shown in Figure 22 and Figure 23. In Figure 22, the parameter prioritizing technique helps to identify that parameter H and M are less relevant to the performance by comparing the relative sensitivity values for all parameters. Even with 25% perturbation in the performance output, the parameter prioritizing technique can still correctly identify the parameters that are less relevant to the performance.

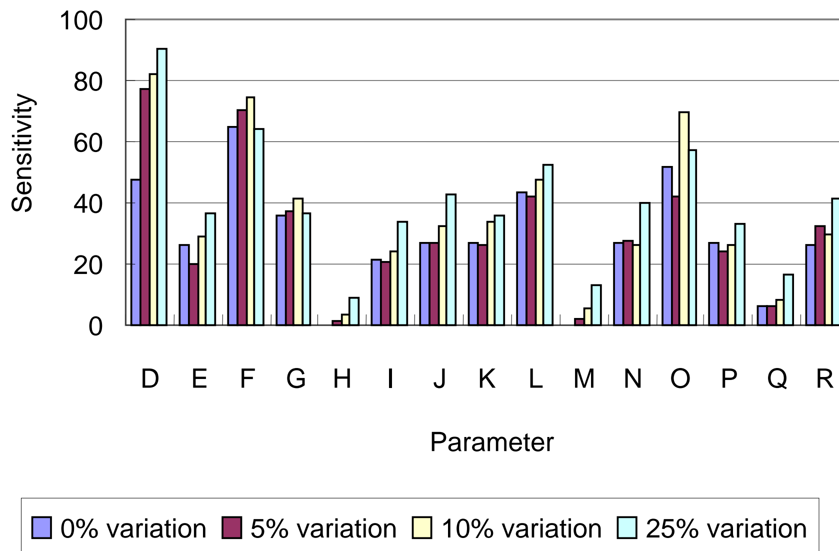
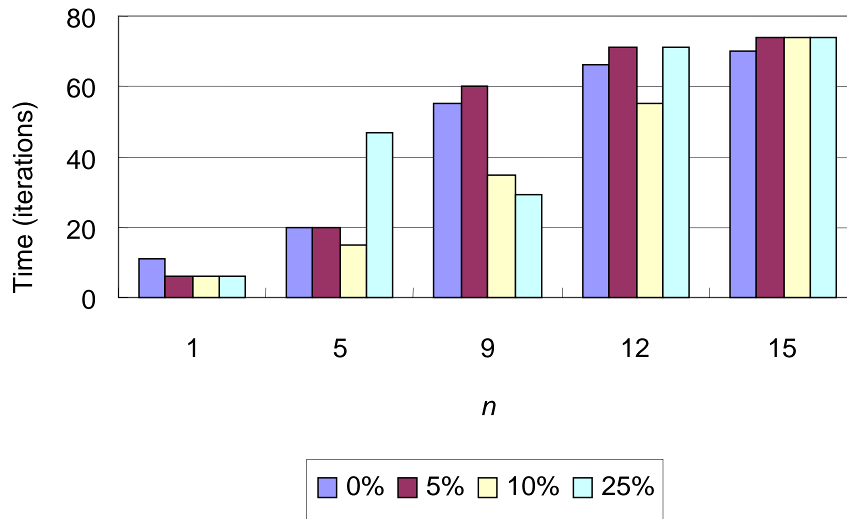
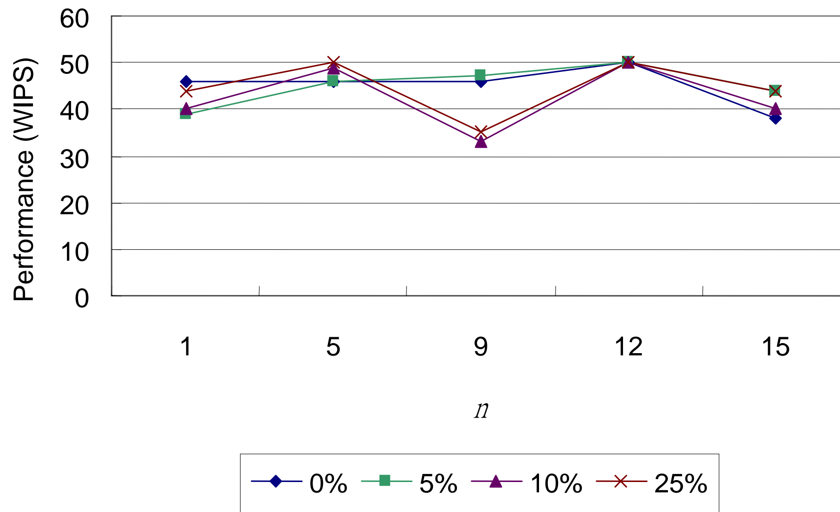


Figure 22: Parameters sensitivity of the synthetic data



(a) Tuning time



(b) Tuning result

Figure 23: Tuning using only *n* most sensitive parameter(s) using synthetic data.

We next consider the impact of variance in the objective function value and its impact on the overall Active Harmony tuning system. In Figure 23, based on the parameter sensitivity obtained, we let the system tune the *n* most sensitive parameters while leaving the rest of the parameters with their default values. The bar in Figure 23(a) shows the time it takes for the tuning and the lines indicate the tuning results. The associated point in Figure 23(b) shows the tuning result. For those cases with less variation, the results show that only tuning a few "performance-critical" parameters will save a dramatic amount of tuning time (up to 85%) while compromising little of the performance (less than 8%). In Section 7.7, we evaluate this technique on a real application.

6.2. Parameter Duplication

When tuning a large-scale system that uses the SPMD (single program multiple data) model, we find that most of the tunable parameters can be categorized into different sets. Most sets are the replica of one of the "basic" sets and the environment is similar for those sets of parameters. Consider a cluster-based web server where there are three tiers and several servers at each tier. Tunable parameters are on all nodes in all tiers. We may first categorize all the parameters into sets based on the tier. Each node in the same tier has same or similar functionality and for the nodes in the same tier, they have same set of tunable parameters. From the results of our work in [21], we also find that the final values of those tunable parameters to be similar. The results also suggest that for similar nodes, we simply tune all the parameters on one of the representative nodes and replicate those values to all other nodes in the same tier.

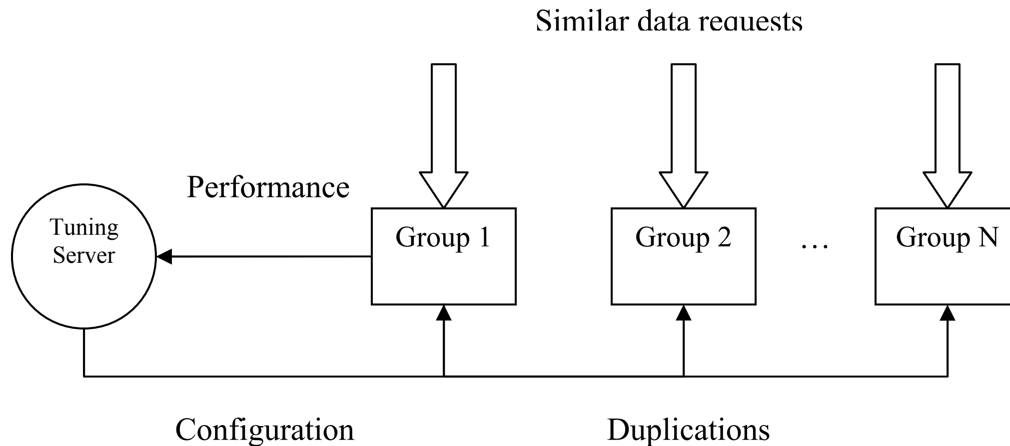


Figure 24: Parameter duplication

The concept of the parameter duplication is illustrated in Figure 24. We extended this concept and integrated it into the Active Harmony system. When sets of parameters are the replica of one basic set of parameters and the environment is the same or similar, we may simply tune one set of parameters and replicate the values to the rest of associated sets of parameters.

When the data analyzer and runtime analyzer find that two or more nodes or processes are similar in their input and application behavior (application signature), the Active Harmony tuning server should only tune one set of parameters from one process and duplicate the values from the tuning result to all the other associated nodes or processes. The similarity we used is the nearest neighbor method from data mining. The similarity in the data request may be the same workload distribution as in the cluster-based web service project. The similarity in terms of the environment may be the same hardware or software environment such as CPU, memory, and OS in the node level. And for the processes level, we can observe their application signature to

decide whether those two processes are having similar environment. We evaluate this technique on the cluster-based web service system in Section 7.9.

6.3. Parameter Partitioning

When the number of tunable parameter increases, another method is to increase the number of tuning servers to share the burden. If the number of tuning servers can increase as the number of the parameters increases, then the scalability issue is solved.

From our experience we find that this is a possible solution for some types of systems or applications. For example, in a cluster-based web server, by observing the data (requests) flow, we may divide the system into work lines. Each work line group consists of at least one server from each tier. A request to the web cluster system is only handled exactly by one work line group. In other words, any server in work line group A will not generate (serve) requests to (from) a server in work line group B. We use a different Active Harmony tuning server to tune the parameters for each of the work lines. The results show that using this method not only speeds up the tuning process (reducing 33% of the tuning time) but also makes the tuning process more stable (reducing the standard deviation from 30 to 9.7). Since each tuning server is responsible for fewer tunable parameters, there are fewer configurations to explore and it is faster to "converge" to the target. Besides, the impact when changing one parameter in a tuning group is limited to that group and will not affect the performance of other groups.

From experience, we find that systems or applications must exhibit certain characteristics for the parameter partitioning mechanism to be applied. First, the parameters should be able to be divided into groups and there should be no interaction

between groups. Also each group of parameters requires a performance measurement to reflect the effect for those parameters. This performance measurement is used as the feedback to its associated tuning server. The latter requirement usually makes it difficult to partition the parameters into groups. For example, if the parameters are partitioned into groups based on the tiers, it is difficult to decide the performance contributed by a particular tier solely.

6.4. Parameter Restriction

In Section 3.2, we described the Resource Specification Language which is used to communicate between the system to be tuned and the Active Harmony tuning server. The system to be tuned specifies the parameters together with their value limit boundaries and the distance between two neighbor values for discrete parameter.

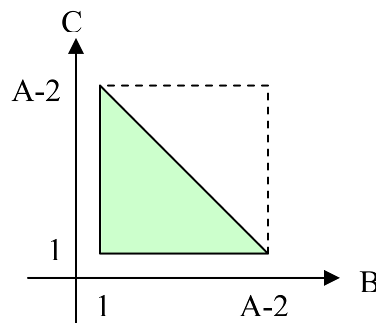


Figure 25: Search space reduction by parameter restriction

We improved the Resource Specification Language to allow the value of one parameter to be a function of another parameter value. This can help to reduce the search space dramatically. For example, assume there is a fixed number A of total processes running on a node. Some number B of those processes are designated to

handle the disk I/O tasks while some other number C of the processes are designated to handle the CPU computational tasks and the remaining D processes are used to handle the network connections. Let's assume B, C, and D are the three tunable parameters. When the relation A=B+C+D is known, we may need to tune two parameters B and C only, since D=A-B-C will be decided automatically after B and C are decided. Furthermore, we may set the value limit boundaries of B to be [1,A-2] and set the value limit boundaries of C to be [1,A-B-1] (assume at least one process is required for each different type of task) as shown in the Figure 25. Whenever the server needs to "figure out" the next configuration, it decides the value for the parameter B first. Then it will decide the value for the parameter C based on the value of B. By doing this, we are able to reduce the high-dimensional search space (the dashed area in the Figure 25).

{ harmonyBundle B { int {1 ***8*** *1} }}*

{ harmonyBundle C { int {1 ***9-$B*** *1} }}*

*{ harmonyBundle D { int {****10-$B-$C 10-$B-$C*** *1} }}*

Figure 26: Improved Resource Specification Language syntax example

Figure 26 shows the syntax using parameter restriction. The first line indicates that parameter B ranges from 1 to 8. The second line indicates that parameter C ranges from 1 to 9-B. The third line defines parameter D as a function of the values for parameters B and C. When the Active Harmony tuning server needs to decide the values for a new configuration, it will first decide a value for parameter B within the range [1, 8]. And then for the parameter C value, the tuning server will make sure it

will be within the range [1, 9-$B]. By doing so, only the "meaningful" configurations will be explored (e.g., configurations that include B=6 and C=6 will be discarded automatically).

One example use of this feature is to constrain the connectors used on a web server. On web servers, there are different types of connectors that handle different kinds of requests (e.g., non-secured, secured … etc.). A connector is a process that handles incoming requests. The number of connectors decides the number of requests that can be handled concurrently. When the total number of connectors is decided, we can use this technique to select the number for each type of connectors. We also apply this technique when tuning a scientific library in Section 8.1. When tuning the library, Active Harmony needs to decide how the matrix with k rows is partitioned into n blocks.

By observing the relations among parameters and eliminating infeasible configurations, this technique helps to reduce the search space and thus speeds up the tuning process.

6.5. Summary

In this chapter, we improved the scalability for the Active Harmony tuning system. Rather than tuning all parameters at once, prioritizing parameters helps to focus on parameters that have a greater impact on the performance. When the parameters can be categorized into sets, parameter duplication tunes the same parameter on different nodes concurrently and parameter partitioning helps to tune separate parameters in parallel. We also improved the Resource Specification Language so the search space is reduced by observing the relations among parameters. All these techniques help to

speed up the tuning process when dealing with numerous tunable parameters. In the next two chapters we will evaluate these techniques using real applications.

Chapter 7: Cluster-Based Web Service System

In this chapter we tune a real system, a cluster-based web service system. Cluster-based web service systems are used as a standard mechanism for online information distribution and exchange. In order to provide such service, e-commerce sites require large online web systems. Such systems must be capable of running continuously and reliably 7 days a week, 24 hours a day. Besides, the systems must be able to accommodate widely varying service demands. They should also be adaptive when the number or nature of requests changes.

Clusters of commodity workstations interconnected by a high-speed network are frequently used to meet these challenges. The infrastructure can tolerate partial failures and allows scaling up by adding more components. The administration mechanism for such a large cluster does not have to be reinvented for each new service.

7.1. Cluster-Based Web Service System

In many web services today, there are (conceptually, at least) three tiers as shown in Figure 27: the presentation, middleware, and database. The presentation tier is the web server that provides the interface to the client. The middleware tier is what sits between the web server and the database. It receives requests for data from the web server, manipulates the data and queries the database. Then it generates results using existing data together with answers from the database. The third tier is the database, which holds the information accessible via the Web.

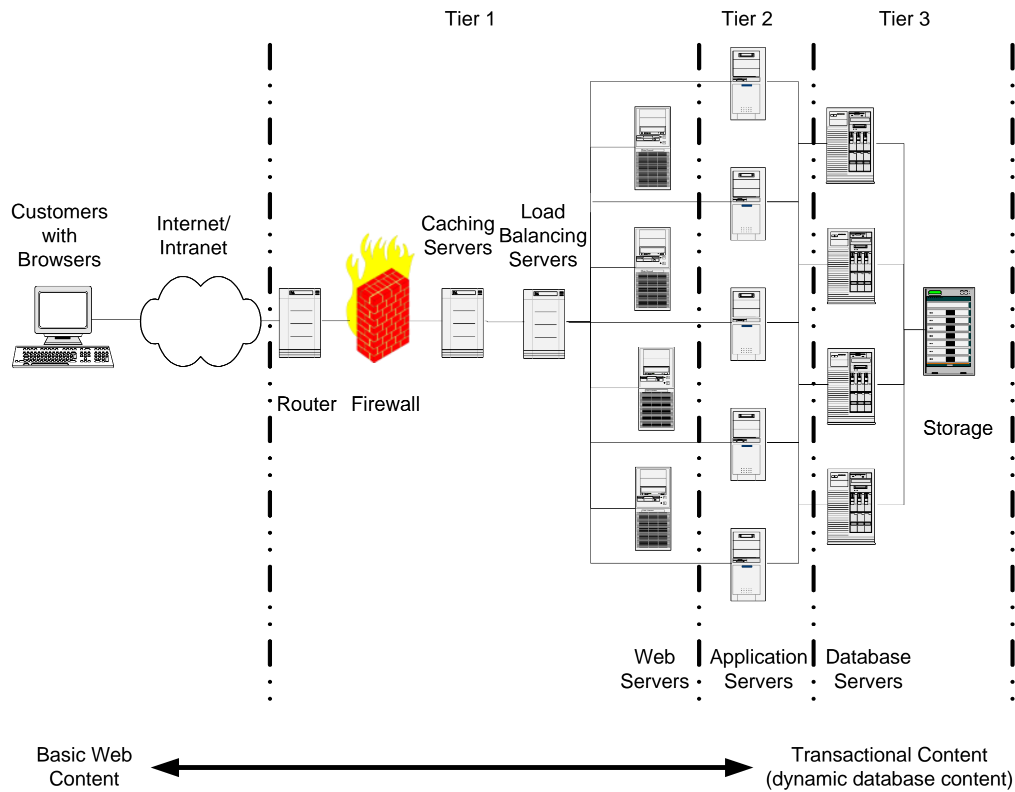


Figure 27: Multi-tier architecture

For such an architecture, all the cacheable and static data are handled by Tier 1. For example, a customer browses the company information or product spec sheets. The server side applications are running in Tier 2. For example, CGI or Java Servlet programs are in this tier. A customer may interact with the Web server to customize his or her merchandise. The request data is received by Tier 1 and then passed to Tier 2. The interaction is then handled by the server side applications and then returned through Tier 1.

While Tier 2 interacts with the customer and processes data, it may need to communicate with Tier 3, the database, for information about pricing, configuration parameters, transaction processing information, etc. After a customer places an order, Tier 2 first queries the price information from Tier 3. Then it process the transaction

based on the query results. Finally the receipt is presented back to the customer through Tier 1.

A scenario for such an architecture can be: the user fill out a form on his(her) web browser; the web server receives the request and passes the information to the middleware. The middleware translates the information into appropriate SQL and queries the database. Tier 2 then takes the data from the database (and does some manipulation or calculation if necessary) and turns the results into HTML pages. These pages are then sent back to the web server, which in turn serves them out to the web browser.

To increase performance, flexibility, and scalability, dedicated machines for different functionality are generally used and multiple machines can be used at each tier to increase throughput. In most systems today, software configuration tuning is done by either experienced system administrators or from the default configurations set by the system developers. The default configurations are set based on a general expectation of the environment in which the system will be executed. Those configurations will make the system work in most of environments but the performance may vary dramatically due to the difference in each customer's environment.

7.2. TPC-W Benchmark

The major workload we use when tuning the cluster-based web service is the TPC-W benchmark. The TPC-W is a transactional web benchmark designed to mimic operations of an e-commerce site. The TPC-W workload is made up of a set of web interactions. Different workloads assign different relative weights to each of the web

interactions based on the scenario. The workload explores the breadth of system components together with the execution environment. Like all other TPC benchmarks, the TPC-W benchmark specification is a written document which defines how to setup, execute, and document a TPC-W benchmark run. The details for each workload breakdown are available online [5].

Web Interaction	Browsing (WIPSb)	Shopping (WIPS)	Ordering (WIPSo)
Browse	**95 %**	**80 %**	**50 %**
Home	29.00 %	16.00 %	9.12 %
New Products	11.00 %	5.00 %	0.46 %
Best Sellers	11.00 %	5.00 %	0.46 %
Product Detail	21.00 %	17.00 %	12.35 %
Search Request	12.00 %	20.00 %	14.53 %
Search Results	11.00 %	17.00 %	13.08 %
Order	**5 %**	**20 %**	**50 %**
Shopping Cart	2.00 %	11.60 %	13.53 %
Customer Registration	0.82 %	3.00 %	12.86 %
Buy Request	0.75 %	2.60 %	12.73 %
Buy Confirm	0.69 %	1.20 %	10.18 %
Order Inquiry	0.30 %	0.75 %	0.25 %
Order Display	0.25 %	0.66 %	0.22 %
Admin Request	0.10 %	0.10 %	0.12 %
Admin Confirm	0.09 %	0.09 %	0.11 %

Table 1: TPC-W benchmark workloads

The two primary performance metrics of the TPC-W benchmark are the number of Web Interaction Per Second (WIPS), and a price performance metric defined as Dollars/WIPS. However, some shopping applications attract users primarily interested in browsing, while others attract those planning to purchase. Two secondary metrics are defined to provide insight as to how a particular system will perform under these conditions. WIPSb is used to refer to the average number of Web Interaction Per Second completed during the Browsing Interval. WIPSo is used to

refer to the average number of Web Interaction Per Second completed during the Ordering Interval.

The TPC-W workload is made up of a set of web interactions. Different workloads assign different relative weights to each of the web interactions based on the scenario. In general, these web interactions can be classified as either “Browse” or “Order” depending on whether they involve browsing and searching on the site or whether they play an explicit role in the ordering process. The details for each workload breakdown are shown in Table 1.

7.3. Environment

To evaluate the Active Harmony system using a real e-commerce workload, we configured a cluster using various components. The summary of the environment used for our experiment is shown in Table 2. The 10 machines used include the ones running emulated browsers and the servers for proxy, HTTP, application and database services. Each machine is equipped with dual processors, 1 Gbyte memory and runs Linux as the operating system. For each tier, we select Squid as the proxy server, Tomcat as the HTTP & application server and MySQL as the database server. All computer software components are open-source which allows us to look at source code to understand system performance. The TPC-W benchmark version we chose simulates a store that sells approximately 10,000 items.

The effort it took to harmonize a server ranged from half day to two working days. The major challenge to harmonize a server is to identify the tunable parameters either in the configuration file or inside the source code. Therefore this time can be further shortened if the assistance from the server developer or expert is available. Once the

tunable parameters are identified, Active Harmony API (described in Section 3.3) can be applied easily to harmonize the server.

Hardware	
Processor	Dual AMD Athlon 1.67 GHz
Memory	1Gbyte
Network	100Mbps Ethernet
No. of machines	10
Software	
Operating System	Linux 2.4.18smp
TPC-W benchmark	Modified from the PHARM [13]
Proxy Server	Squid 2.5 [4]
HTTP & Application Server	Tomcat 4.0.4 [1]
Database Server	MySQL 3.23.51 [2]

Table 2: Experiment environment

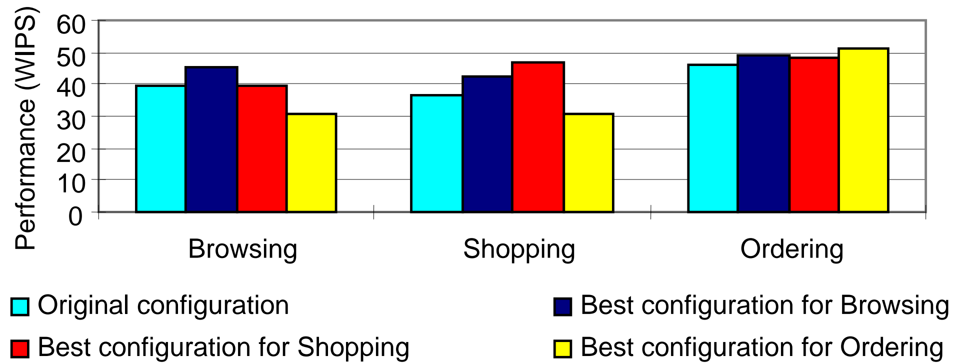
Our goal is to improve the overall system performance using Active Harmony. We first show that there is no single configuration suitable for all the workloads. Active Harmony makes the system perform better by using different configurations when facing different workloads. Then we investigate Active Harmony's scalability as the number of machines grows. One way to solve this problem is to partition the parameters into sets. We show how to use an independent Active Harmony tuning server for each set to speed up the tuning process. Another method is to tune a representative set of parameters and use duplicated values on the rest of nodes. Later we also show how to adjust the number of nodes in each tier dynamically to reduce hot spots.

7.4. Impact of Varying Workload

In this experiment we show that the Active Harmony server can tune the system to adjust each tier's server to provide good performance. We use four machines in

this experiment: one machine for the emulated browsers, one for the proxy server, one for the HTTP & application server, and one for the database server.

In the experiment, we examine the tuning processes for two different workloads: browsing and ordering. Both tuning processes are started using the default configuration. We then let the system warm up for 100 seconds and measure the performance (WIPS) for 1,000 seconds followed by 100 seconds for cooling down. We define such a cycle as one "iteration"[7]. The Active Harmony server will adjust the configuration (parameters values) between two iterations.



	Best configuration after 200 iterations		
	Browsing	Shopping	Ordering
Improvement compared to the default configuration	15%	16%	5%

Figure 28: Applying best configuration after 200 iterations to different workloads

[7] The 1,200 second-iteration is TPC-W benchmark compliant (i.e., specified in the TPC-W documentation). The iteration timescale can be as short as 30 seconds according to our experiment experience.

Figure 28 shows that for different workloads, the system should apply different configurations. Each different colored bar represents the best configuration we found after 200 tuning iterations for a particular workload. We then apply those best configurations to the other two workloads for comparison. The results show that when using a configuration that is tuned for another workload, the system does not perform as well as using a configuration that is tuned for the current workload. The results show that there is no universal configuration that is the best for all kinds of workloads. The table in Figure 28 shows the improvements for those best-tuned configurations compared to the default configuration. The improvements range from 5% to 16%.

Table 3 shows the values of all Harmony tunable parameters before and after tuning for each of the workloads of the TPC-W benchmark. The results show that for the proxy server, it first increases the main memory size for the cache to improve the performance (cache_mem). For the shopping and ordering workloads, the proxy server tries to cache larger objects in the memory (minimum_object_size). For the HTTP server (which is part of the application server), the tuning results show that it spawns more threads to handle the requests during the ordering workload (AJPminProcessors). We believe the main reason for this is that most of the requests in the ordering workload require high latency operations in the database server (i.e., performing update transactions on the database). Thus the average response time is longer compared to other workloads. As long as it is not over the system capacity, the HTTP server should use more threads (minProcessors/maxProcessors) and buffer space (bufferSize) to handle the incoming requests. The waiting queue capacity

should also increase accordingly (acceptCount) as the results show. The same situation happens in the worker part (AJP connector) of the application server. For the database server, the tuning results show it increases the cache and buffer size when the utilization for the database is high (i.e., shopping and ordering workloads). However, it shows that reducing the join buffer size does not impact performance since the table may not be large enough.

Tunable parameters[8]		Default config.	Best config. after 200 iterations		
			Browsing	Shopping	Ordering
Proxy Server	cache_mem	8	13	17	21
	cache_swap_low	90	91	86	91
	cache_swap_high	95	96	96	96
	maximum_object_size	4,096	4,096	4,096	5,888
	minimum_object_size	0	0	50	306
	maximum_object_size_ in_memory	8	6	256	2,560
	store_objects_per_bucket	20	15	25	105
Web Server	minProcessors	5	1	16	102
	maxProcessors	20	11	16	131
	acceptCount	10	6	21	136
	bufferSize	2,048	2,049	3,585	6,657
	AJPminProcessors	5	6	26	136
	AJPmaxProcessors	20	86	296	161
	AJPacceptCount	10	76	306	671
Database Server	Binlog_cache_size	32,768	63,488	153,600	284,672
	Delayed_insert_limit	100	200	400	700
	max_connections	100	201	451	701
	delayed_queue_size	1000	2,600	9,100	7,100
	join_buffer_size	8,388,600	407,552	407,552	407,552
	net_buffer_length	16,384	31,744	38,912	34,816
	table_cache	64	873	905	761
	Thread_con	10	81	91	76
	Thread_stack	65,535	102,400	1,018,880	773,120

Table 3: Tuning results for different workloads

From the results we can see that some parameters significantly affect the overall system performance such as the number of threads or the buffer size. However, there

[8] The parameter names are acquired from configuration files. Parameters with name including "Processors" actually relate to the number of threads or processes.

are some parameters that we expected to be performance related but turned out not to be important. For example, the thresholds (cache_swap_low, cache_swap_high) which control whether the proxy server should swap out objects do not impact the overall system performance. Determining which parameters are important is useful. But it is difficult for system administrators and developers if they do it manually. Since it is automated, the Active Harmony tuning process is also helpful for system administrators and developers to identify those parameters that actually affect system performance.

Figure 29: Tuning responsiveness to the changing workloads

Figure 29 shows the tuning system's responsiveness to changing workloads. The system is started with the default configurations for all of the servers. We change the workload every 100 iterations. As shown in the figure, the response time for the system to adjust itself when the workload changes is fairly short. Only few iterations are needed to adapt to the new workload. The Active Harmony tuning server not only

helps the system react to the changing workload, it also makes the adjustments fairly quickly. As displayed in the figure, it only takes few iterations for the tuning server to react and the performance is improved up to 16%. This is helpful when the system is facing real-world traffic that can change at a rate faster than a person could manually tune the system.

7.5. Utilizing Historical Data

Trying one single configuration on a system may be time-consuming and the tuning process should avoid trying unnecessary configurations. During the tuning process, Active Harmony will keep a record of all the parameter values together with the associated performance results. This is useful for future reference if the system is running with the same or similar workload later. We verify this design using Active Harmony on a cluster-based web service system.

In the cluster-based web service system, the data analyzer will first spend a small amount of time to characterize the requests by observing the frequency of different web interactions. We expect each different workload will have a different web interaction distribution. By observing the frequency distribution for web interactions, the data analyzer can characterize the workload that the system is serving. During the running stage, the configuration used is also stored together with the associated request characteristics for future references. Next time when tuning the application, the Active Harmony system will first analyze the characteristics (frequency and distribution of web interactions in the case of the cluster-based web service application) of the incoming requests as described in Section 5.1. It will then compare

them with the information stored in the data characteristics database, and then use the appropriate historical data to prepare (train) the system to be tuned.

	Shopping workload		
	Convergence time (iteration)	Performance after tuning (WIPS)	Initial performance oscillation average (standard deviation)
Without prior histories	39	56.99	53.34 (9.30)
With prior histories	17	59.30	57.43 (5.72)
	Ordering workload		
	Convergence time (iteration)	Performance after tuning (WIPS)	Initial performance oscillation Average (standard deviation)
Without prior histories	23	76.26	59.66 (17.96)
With prior histories	19	76.26	71.50 (10.96)

Table 4: Tuning process with and without prior histories

In this experiment, we have the system serving a workload A (that the system has never served before) both with and without using historical data. In Table 4, when we use the shopping or ordering workload, the tuning process is smoother and the performance converges faster (56% faster for the shopping workload and 17% faster for the ordering workload) when the tuning server is first trained using historical data recorded from another workload. For the shopping workload with prior histories, there is only one bad performance iteration[9] in the tuning process compared to nine bad performance iterations when without prior histories. And for the ordering workload and prior histories are used, there are three bad performance iterations in

[9] The performance is worse than the performance using default configuration.

the tuning process compared to eleven bad performance iterations when prior histories are not used.

7.6. Improved Search Refinement

In order to make the tuning process more stable, we modify the tuning kernel inside Active Harmony as discussed in Section 5.2. In this section we evaluate these modifications in the tuning algorithm. The modifications tradeoff between the number of configuration evaluated and cumulative performance function. Without using configurations with extreme values, it may take a longer time for tuning. On the other hand, the performance during the tuning process should be more stable since configurations with extreme values often make the system perform poorly. We expect that with these modifications, the tuning process will be more stable and therefore time spent in those iterations with bad performance will not dominate the tuning time.

	Shopping workload		
	Convergence time (iterations)	Performance after tuning (WIPS)	Worst performance[10] WIPS (std. dev.)
Original implementation	90	63	20 (17.6)
Improved search refinement	58	60	27 (6.2)
	Ordering workload		
	Convergence time (iterations)	Performance after tuning (WIPS)	Worst performance WIPS
Original implementation	74	79	29 (11.3)
Improved search refinement	46	80	29 (8.9)

Table 5: Tuning process with and without improved search refinement

[10] The worst performance found in the performance oscillation stage.

We apply the Active Harmony tuning server with this improved kernel to the cluster-based web service system. The summary of the tuning process is shown in Table 5. The convergence time represents the time it takes for tuning. The performance column shows the tuning result and the worst performance column describes how smooth the tuning process is. From the summary shown in the table, the convergence time is much shorter after improving the tuning kernel while maintaining similar performance tuning results. For the improved search refinement, the results show that the proposed improvement helps to speed up the tuning process by reducing the convergence time by about 35%. We believe this is because the desirable configuration points are not at the boundaries of the parameter values. The improved search refinement also helps to reduce the magnitude of the initial performance oscillation for both workloads as indicated by the standard deviation values.

7.7. Parameter Sensitivity

We developed the parameter prioritizing tool (described in Section 6.1) since tuning a large number of parameters can be very slow. A long tuning process makes the tuning result unusable since the workload and the environment may have changed. In this section, we apply the technique to the a real system to verify our design.

We apply our parameter prioritizing tool introduced in Section 6.1 to 10 parameters in the cluster-based web service system. Figure 30 compares the relative sensitivity for all parameters. The results are normalized to show the most sensitive parameter as 100%. When the system faces different workloads, the results show that each parameter has a different degree of importance to the system's performance. For

example, the network buffer size of the MySQL database server is relatively important when the system is serving the ordering workload since most requests are placing orders and the database server is highly utilized. On the other hand, when the system is serving the shopping workload, more browsing activities are coming into the proxy server and this kind of request can be served more quickly with static data stored in the cache memory. Therefore, the size of the cache memory has more impact on the overall system performance. Some parameters like the buffer size for the HTTP web server or maximum number of connections allowed by the database server are relatively less important for the system when facing shopping or ordering workloads.

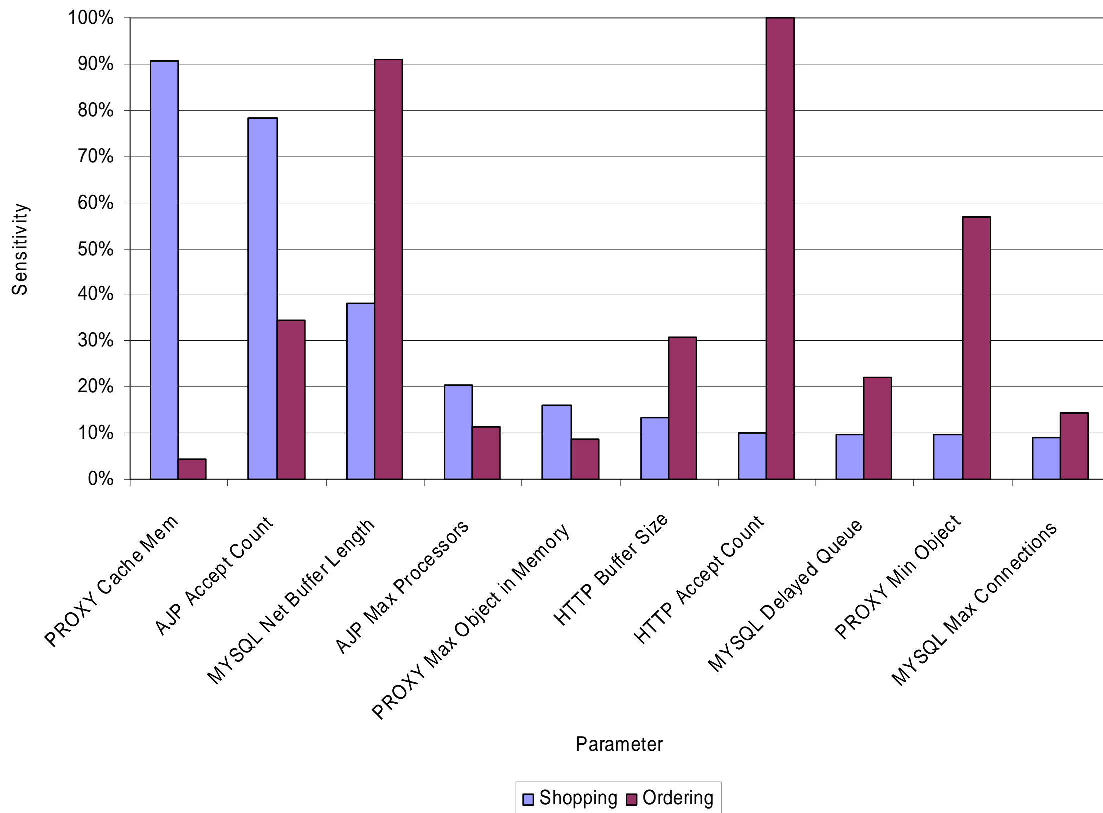


Figure 30: Parameter sensitivity in the cluster-based web service system

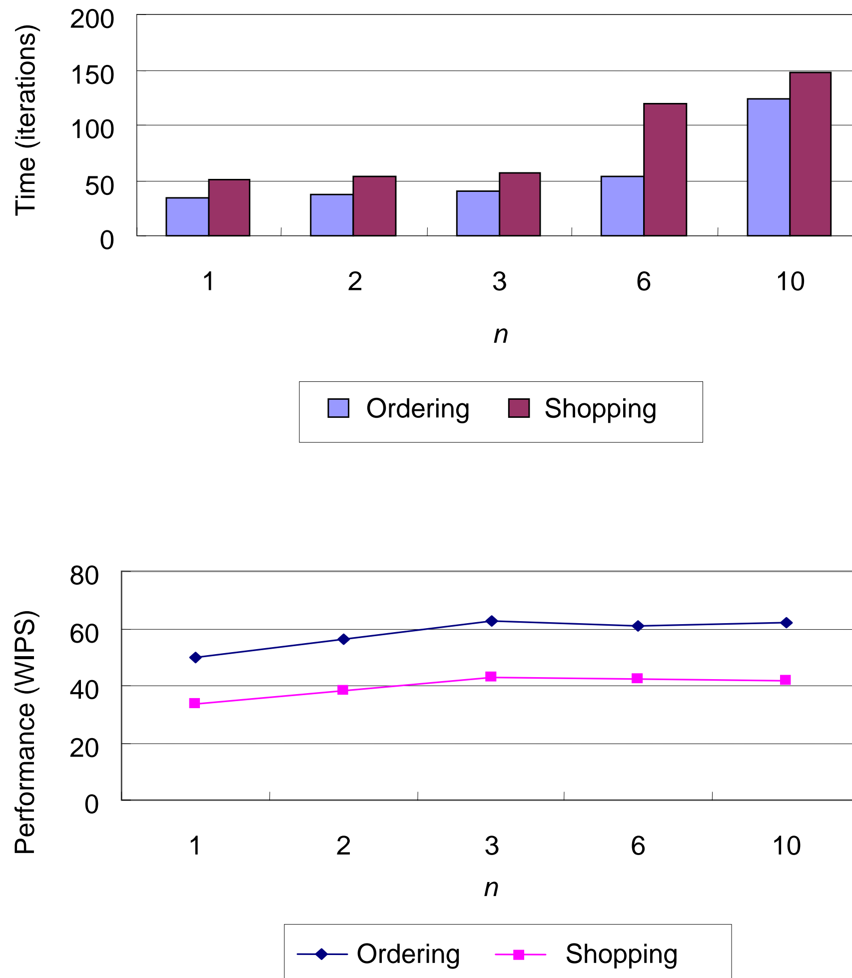


Figure 31: Tuning using only *n* most sensitive parameter(s) of the cluster-based web service system

We now consider the question of how many parameters need to be tuned. We consider tuning using only the top n most important parameters based on our sensitivity analysis. We vary n from 1 to 10. Figure 31 shows that only using a limited number of parameters can reduce tuning time significantly. The bars in the first figure show the time it takes for the tuning process and the lines in the second indicate the tuning results. The lines show that about the same overall performance was obtained, but the bars indicate that by tuning only the important parameters, the

time required for tuning can be reduced. When tuning a system with numerous parameters, it is helpful to first spend some effort separating performance related parameters from those that are less relevant to performance. Tuning only those performance related parameters reduces the tuning time (up to 71.8%), while compromising a little of the performance in the tuned system (less than 2.5%).

7.8. Putting together

In this section, we tune the cluster-based web service system with the improved Active Harmony. The experiment starts with 20 parameters. The improved Active Harmony that utilizes historical data and improved search refinement selects 10 out of the 20 parameters for tuning. As shown in Table 6, even though the improved Active Harmony spent iterations to test each parameter, the tuning process is still faster compared to the original Active Harmony. One can expect this speedup in the tuning process time will be bigger when there are more parameters. Also with improved Active Harmony, the tuning process is more stable (with smaller standard deviation).

Active Harmony	Shopping workload		
	Tuning time (iterations)[11]	Performance after tuning (WIPS)	Worst performance[12] WIPS (std. dev.)
Original	205	63	21 (19.6)
Improved	108	61	25 (12.2)
Active Harmony	Ordering workload		
	Convergence time (iterations)	Performance after tuning (WIPS)	Worst performance WIPS
Original	174	80	29 (13.1)
Improved	96	79	30 (9.3)

Table 6: Tuning process using original and improved Active Harmony

[11] For improved Active Harmony, tuning time includes iterations spent for parameter prioritizing.
[12] The worst performance found in the performance oscillation stage.

7.9. Cluster Tuning

When the number of servers increases, the number of tunable parameters also increases. This makes the tuning process lengthy and the tuning results may not be useful since the environment could change during the tuning process.

In the original Active Harmony system, to tune n parameters at once requires exploring $n+1$ configurations before improvements to the system will take effect. If there are numerous servers in the cluster and each server contains tens of parameters, the tuning process will be fairly long. In order to reduce the initial exploration period, we partition the components inside the cluster into groups and use separate Active Harmony tuning servers for each group.

There are several ways to group servers. When all the machines in the same tier are homogeneous, we try to partition all the servers into tuning groups using two methods. The first one is parameter duplication: we only tune one server for each tier, and the values for those parameters are duplicated to other servers in the same tier. This tuning mechanism is based on the assumptions that (a) servers in the same tier will have the same or similar behavior for the same configuration and (b) the workload is evenly distributed among all the servers in the same tier.

The second way to group nodes, parameter partitioning, is based on a static work line. A work line group consists of at least one server from each tier. A request to the web cluster system is handled by exactly one work line. In other words, any server in work line group A will not generate (serve) requests to (from) a server in work line B. We use a different Active Harmony tuning server to tune the parameters for each

work line. The assumption for this tuning mechanism is that (a) all the work lines are running in parallel and (b) there is no interaction between any of the work lines.

Tuning method	WIPS			Iterations
	After tuning[13] (improvement)	Average during tuning	Standard Deviation[14]	
None (No Tuning)	110.4	110.4	2.1	-
Default method	130.6 (18.3%)	112.1	30.0	159
Parameter duplication	133.7 (21.2%)	116.6	29.5	33
Parameter partitioning	131.3 (19.0%)	121.8	9.7	107

Table 7: Performance for different methods for cluster tuning

Both of these approaches to grouping nodes require some domain knowledge about the role of each node. However, grouping of nodes could easily be exported to Active Harmony as part of the tuning API.

To compare these two approaches, we tuned the system using three different tuning methods: default, parameter duplication and parameter partitioning. Table 7 shows the tuning results. The results for all three methods are very similar. The default method takes the longest time since there are many parameters and only one performance result per iteration. The parameter duplication method provides both a larger performance improvement and faster convergence to the tuned configuration. It speeds up the tuning process since the tunable parameters are distributed to multiple tuning servers and there are fewer parameters for each tuning server to tune. The time (iterations) spent for the grouping by parameter partitioning method is about 2/3 of the default method.

[13] Performance for the best configuration after 200 iterations
[14] For the second 100 iterations

Based on the time for the tuning process, parameter duplication tuning seems to be the best. It takes a much shorter time for tuning. However, if stable performance during the tuning process is critical, parameter partitioning by work lines is a reasonable choice.

7.10. Automatic Cluster Reconfiguration

One of the advantages for a cluster-based web service is the ability to reconfigure hardware easily. By dynamically changing the roles of servers for different workloads, it is possible to make the best of available resources.

The parameter tuning part of the Active Harmony system helps to tune the cluster-based web service at a fine time granularity. However, when the load is not balanced among tiers in the web service system, changing the parameters for all the servers will not provide much help to solve the problem. Instead, it is necessary to adjust the infrastructure by changing the number of servers in each tier dynamically to reduce the load imbalance.

Variable	Description
R_{ij}	Utilization of resource j on node i
LT_{ij}	Low threshold for resource j on node i
HT_{ij}	High threshold for resource j on node i
M_{pq}	Cost to move a job for node p to node q
A_i	Average process time on node i
F	Configuration cost in terms of time
L	List of nodes
N_i	Number of jobs on node i
$Head(L)$	First node in the List L
$Tier(i)$	The tier that node i belongs to
$M(t)$	Number of nodes in tier t

Table 8: Variable description

1. For all node i, resource j do

   If $R_{ij} > HT_{ij}$ then add i to the list L_1

   //find out what nodes are highly or over loaded

2. For all node i do

   If $R_{ij} < LT_{ij}$ for all j then add i to the list L_2

   //find out what nodes are lightly loaded

3. Sort L_1 based on the "degree of urgency[15]"

   //decide the priority for the nodes to be relieved

4. Let $i = Head(L_1)$, find the node k in L_2 such that satisfies (a)(b)(c)

   //find out the appropriate node to be reconfigured

   (a) $Tier(i) \neq Tier(k)$

   (b) $M(Tier(k)) > 1$

   (c) $F + N_k \times M_{km} - N_k \times A_k$ is minimal, where $k \neq m$ and $Tier(k) = Tier(m)$

5. Reconfigure k such that $Tier(i) = Tier(k)$

   //reconfiguration

Figure 32: Reconfiguration algorithm for external tuning

The Active Harmony system applies a simple mechanism to achieve load balance among tiers. While the tuning is in progress, the Active Harmony system monitors the resource utilization for all nodes of all tiers. The resources that are monitored include CPU load, memory usage, network bandwidth used and disk I/O activity (currently the system information is obtained using Linux SAR utility tool). Periodically, Active Harmony detects whether (1) there is a resource on node A that is over utilized[16], (2) all the resources on node B are under utilized and node B is suitable for

[15] The degree of urgency for each node depends on the characteristics of the application. It may vary from case to case. For example, overloading the CPU may cause bigger problem than utilizing all the network bandwidth for some applications. Therefore, nodes with overloaded CPUs will have higher priority than nodes whose network bandwidth is highly utilized.

[16] Static thresholds (e.g., CPU idle time is less or equal than 5%) are used in the current implementation.

reconfiguration. If both situation (1) and (2) exist, Active Harmony tries to reconfigure node B to run the same server process as node A.

Unlike parameter tuning, which is done for each iteration, the reconfiguration algorithm is run at a lower frequency (e.g., every 50 iterations) since it is designed to react to longer term trends, and incurs a greater overhead to make changes. Table 8 shows the definition for variables in the algorithm and Figure 32 shows the concept of the reconfiguration algorithm.

Step 1 determines which nodes are overloaded. It checks the resource utilization against the predefined high threshold. Step 2 tries to find nodes that are lightly loaded. If all the resources on the node are idling most of the time (i.e., utilization is smaller than the lower threshold), the node is considered under utilized. Step 3 finds out which node is the most "urgent" node that should be relieved first. Step 4 is to ensure correct operation (that there is at least one node left in each tier) and decides if the reconfiguration should be done immediately (by moving existing requests to the neighbor nodes in the same tier) or if it should wait until all existing requests finish. Finally Step 5 does the reconfiguration.

$$F + N_k \times M_{km} - N_k \times A_k \qquad (1)$$

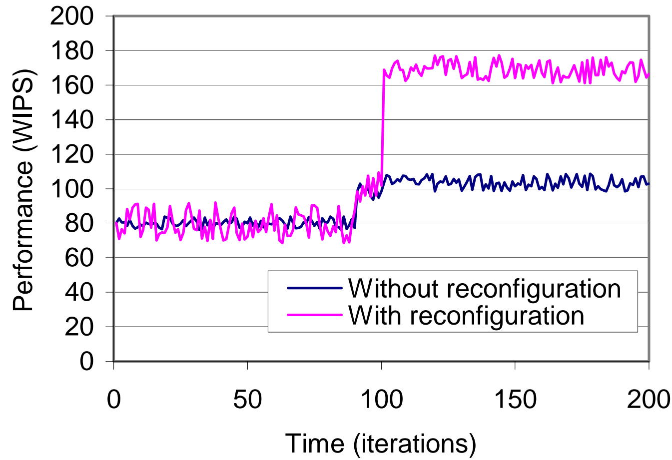
When the result of equation (1) for the selected node *k* in Step 4(c) is non-negative, the Active Harmony system will not reconfigure node *k* immediately until all jobs on it are finished. This is because it will be more cost-effective to wait than to reconfigure node *k* immediately. On the other hand, when the result of the equation is

negative, the Active Harmony system will reconfigure node *k* immediately. This is because the cost for immediate reconfiguration will be less than waiting for the system to be idle to reconfigure.

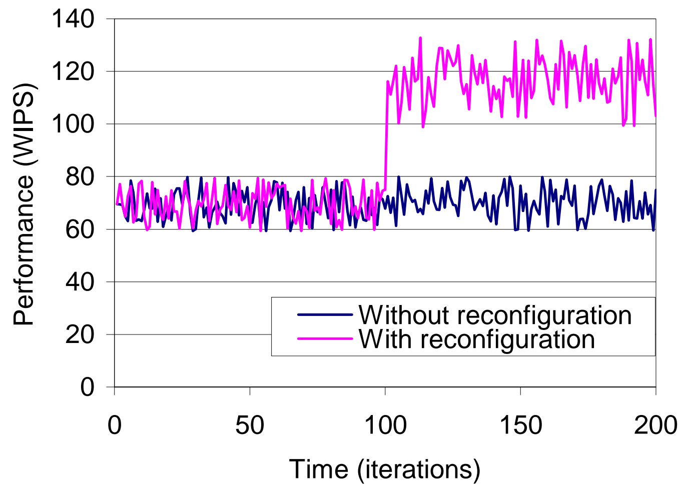
Active Harmony can automatically perform node reconfiguration without taking the system down. While one node is being reconfigured from one tier to another, all the remaining nodes in the system are still serving requests normally.

Figure 33 shows the experimental results when applying the reconfiguration algorithm. The initial configuration for Figure 33(a) has four nodes serving the proxy tier and another two nodes for the application tier; all six nodes are homogeneous. The experiment starts with a browsing workload and changes to an ordering workload after the 90th iteration (the performance gains between 90th and 100th iterations are due to different workloads). We forced the Active Harmony system to do the dynamic adjustment checking exactly once, immediately after the 100th iteration of the tuning process. Figure 33(a) shows the performance improvement when Active Harmony decides to move a node from the proxy server tier to the application server tier based on the algorithm. This is expected since when the system has a workload dominated by ordering, it requires more application servers to handle the dynamic data from the database. On the other hand, most browsing workloads require static data that can be served from the proxy servers. Before the adjustment, the application servers are highly loaded (CPU utilization is always close to 100%) and some proxy servers are idling most of the time (CPU utilization is close to 0% and there are very few network or disk I/O requests). After the adjustment, the average utilization of the application

servers is lowered while the average loading for the proxy servers increases a little. The major bottleneck is relieved and the system performance is improved about 62%.



(a) One node moved from the proxy server tier to the application server tier
(Workload changes from browsing to ordering)



(b) One node moved from the application server tier to the proxy server tier (Browsing workload)

Figure 33: Reconfiguration experiment results

Figure 33(b) shows the performance improvement with a different starting configuration. There are six nodes, two of them serving as the proxy servers and four serving as application nodes. However, the proxy servers are highly utilized under the

browsing workload. Using dynamic adjustment after the 100th iteration, it moved a node from the application server tier to the proxy server tier for the adjustment automatically. The CPU and disk I/O are highly loaded on the proxy servers before the adjustment and some application servers are idling most of the time. After the adjustment, the average load on all proxy servers is lowered, the average utilization on the remaining application servers is increased and the system performance is improved for about 70%.

7.11. Summary

In this chapter, we applied Active Harmony to a cluster-based web service system application. We started with tuning all parameters at once, then demonstrated ideas to improve the tuning process: utilizing historical data so the tuning won't start from scratch every time; improved search refinement helps to search the possible configurations first; parameter sensitivity helps to focus on performance-critical parameters. Finally, due to the characteristics of the cluster-based system, dynamic reconfiguration makes the best of available resources. In the experiments, we were able to improve the cluster-based web service system throughput up to 16% using parameter tuning and up to 70% with dynamic reconfiguration. With parameter duplication, the tuning time can be reduced up to 80%.

Chapter 8: Scientific Programs Tuning

In this chapter, we show that by changing the data and computation distribution, we can improve the performance of scientific libraries and applications significantly. Scientific library and application tuning is an important problem today. Due to the fact that frequently these applications have large computational demands and thus need to be run on large-scale parallel computers. Even a small percentage improvement in the execution time will reduce the cost dramatically. Alternatively, with improved execution time, the program can also achieve better results such as higher resolution, better precision or using a larger data set. Therefore, we try to apply the Active Harmony system to some widely used scientific libraries and applications.

8.1. PETSc Library

PETSc [66] (Portable, Extensible Toolkit for Scientific Computation) is a suite of data structures and routines for the scalable (parallel) solution of scientific application modeled by partial differential equations. PETSc is intended for use in large-scale application projects. Software packages that use or interface to PETSc include TAO[11], SCIRun[53], Magpar[63], libMesh[54], and Snark[7]. It is widely used in optimization, biology, computational fluid dynamics, and wave propagation.

PETSc uses the MPI standard for all message communication. It integrates architecture dependent optimized libraries such as BLAS and LAPACK. It includes parallel linear and nonlinear equation solvers that can be easily integrated into C, C++, and Fortran programs. PETSc also provides interfaces to Matlab and Mathematica.

From the performance point of view, it allows users to have detailed control over the solution process. For example, the user may specify the details of matrix decomposition for data storage or array distribution for computation. This makes performance tuning for this library interesting and challenging since those details are environment and problem dependent.

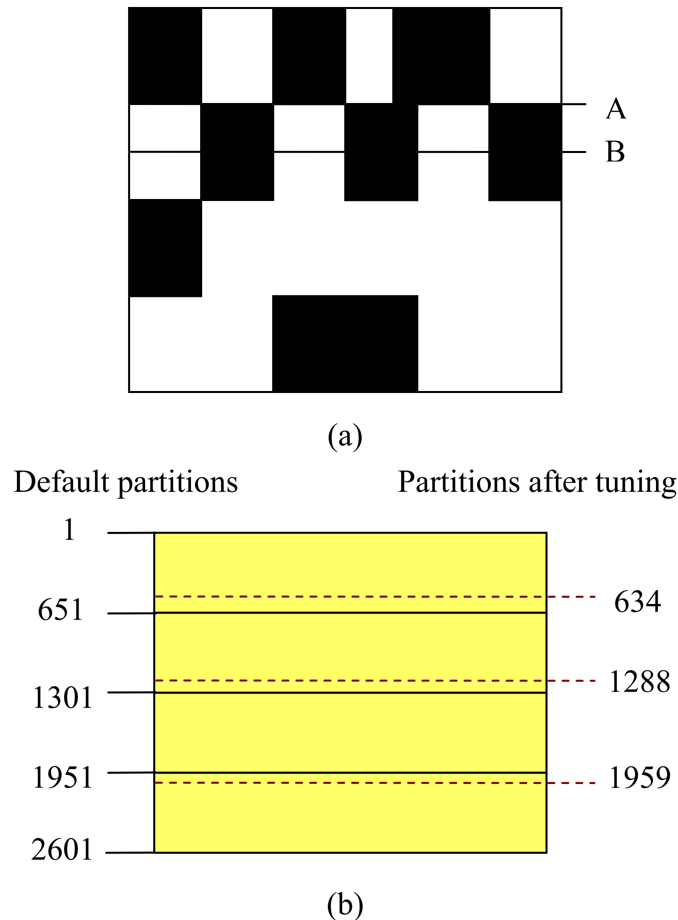


Figure 34: **Matrix decomposition**

We applied the Active Harmony to two PETSc examples to show its tuning ability. The first example solves a linear system in parallel with SLES (linear equation solver). The key point we are interested in is the matrix decomposition. In other words, how the matrix is broken into pieces and stored across processing nodes will change the data locality. This will affect the amount of communication

throughout the computation and thus has a dramatic impact on the performance. The concept is shown in Figure 34(a), the black blocks represent non-zero elements of the matrix. There will be better data locality if the matrix decomposition boundary is using the line A rather than using the line B.

We made slight modifications to the source code to allow Active Harmony to change the boundary for matrix decomposition. Each partition has at least one row and the number of rows for a single partition can be as small as one. Figure 34(b) shows the results for a small sample program running on four processing nodes. The default configuration (solid lines) decomposes the matrix into four even size partitions. After tuning, the result (dashed lines) shows that by changing the decomposition boundaries, the performance is improved. Later we run the example with a matrix size of 21,025×21,025 using 32 processing nodes. This results in an 18% improvement in execution time after tuning.

In order to test the improved Active Harmony, we use a matrix of size 90,601×90,601 as the input to the program. To achieve similar execution time, the program tuned by the improved Active Harmony takes 120 iterations compared to 133 iterations if the it is tuned using the original Active Harmony. This is about a 10% improvement in the tuning time.

The second example is a nonlinear driven cavity with multiple grids in two dimensions. The 2D driven cavity problem is solved in a velocity-vorticity formulation. It uses the SNES (non-linear equation solver) object in the PETSc library. The Harmony tuning involves computation distribution. The problem consists of numerous grid points. The tunable parameter is how the grid points are distributed

among processing nodes. The default configuration divides the grid points into distributed arrays with equal size. This works well in general when the processing nodes are homogeneous (i.e., all the processing nodes have the same processor type and speed). When using heterogeneous processing nodes (where there are nodes with different characteristics), the performance will be influenced dramatically by the layout of the computing grid points.

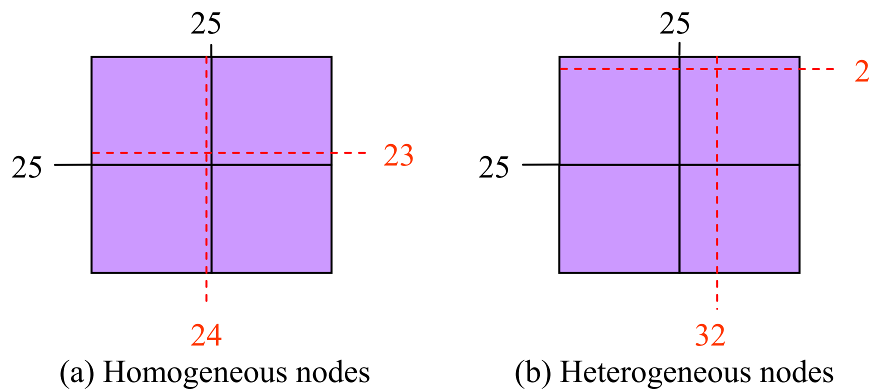


Figure 35: Computation distribution

Figure 35 shows the configurations for a small example problem before and after tuning when using homogeneous and heterogeneous processing nodes. This problem consists of 2,500 grid points with 4 processing nodes. The solid lines are the default configurations and the dashed lines are the results after tuning. The distribution is different for homogeneous and heterogeneous nodes. By comparing Figure 35 (a) and (b), it can be seen that when the processing nodes are homogeneous, the grid points should be divided into distributed arrays with equal size and in the heterogeneous environment, the system should try to utilize the processing nodes (the bottom two nodes) that have more computational powerful.

We applied Active Harmony to the computation distribution example with 40,000 grid points using 32 processors. As a result of tuning, up to an 11.5% improvement in the execution time (compared to default partitioning without tuning) was observed. With the improved Active Harmony, the tuning time is reduced by 14.3%.

8.2. Parallel Ocean Program (POP)

The Parallel Ocean Program (POP) [67, 68] was developed at Los Alamos National Laboratory. POP is a descendent of the Bryan-Cox-Semtner class of ocean models first developed at the NOAA (National Oceanic and Atmospheric Administration) Geophysical Fluid Dynamics Laboratory in Princeton, NJ in the late 1960s[18]. POP is currently used by the Community Climate System Model (CCSM) as the ocean component. POP solves three-dimensional primitive equations for fluid motions on a sphere using hydrostatic and Boussinesq approximations. Spatial derivatives are computed using finite-difference discretizations which are formulated to handle any generalized orthogonal grid on a sphere, including dipole and tripole grids.

We improved the performance (execution time) by enabling Active Harmony to adjust the block (a group of grid points) dimensions and parameter values. The problem size is 3,600×2,400 grid points. The program divides the problem into 480 blocks (processors) and runs on a large SP-3 at NERSC (National Energy Research Scientific Computing Center). The default configuration came with 180×100 as the dimension for each block.

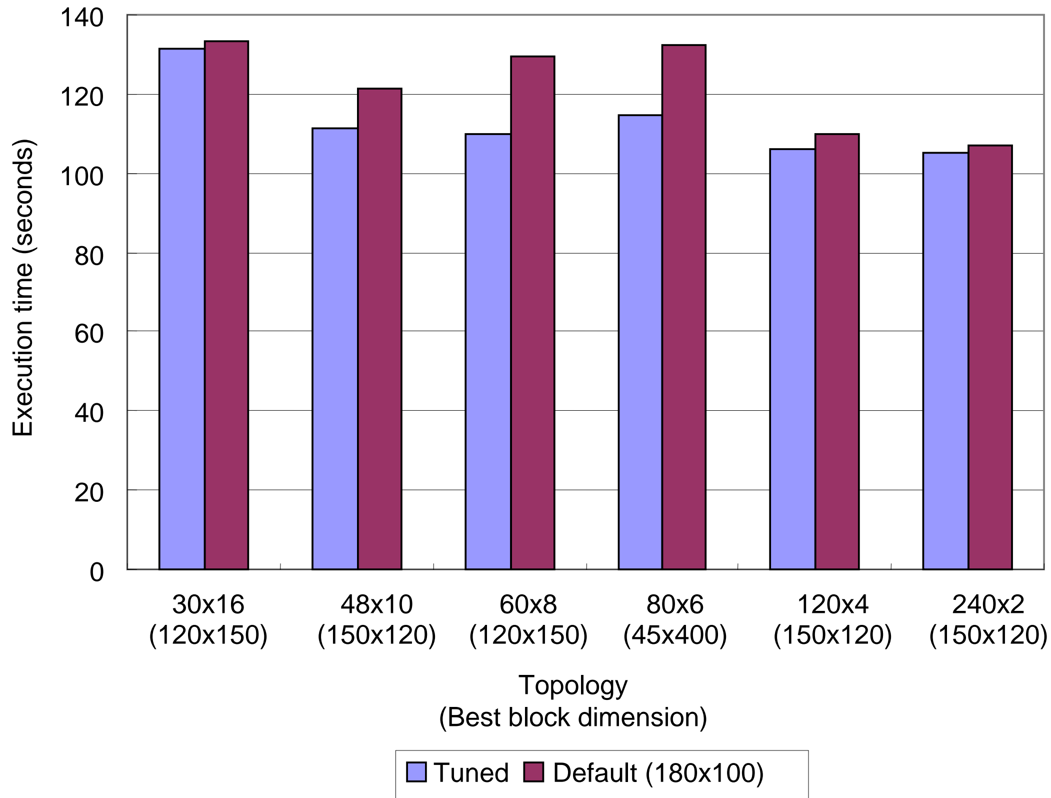


Figure 36: Block dimension tuning

Figure 36 shows the tuning result. There are two dimensions in the x-axis. The first dimension represents the topology for processing nodes and the second dimension in the parenthesis represents the best block dimension found. The two different bars represent the performance for layouts before and after tuning. The first bar is the performance for the layout found (block dimension within the parenthesis on the x-axis) after tuning. The second bar is the performance using default layout (180×100). Consider the second set of bars here. In this application, 48×10 indicates that the topology used in the experiment is 48 nodes with 10 processors on each node and the best block dimension found by tuning is 150×120. The figure shows that there is no single block dimension good for all topologies and the block dimension should

be adjusted accordingly (120×150 is best for topologies 30×60, 60×8; 150×120 is best for topologies 48×10, 120×4, and 240×2; 45×400 is best for topology 80×6). With tuning for block dimension, the execution time can be reduced up to 15%. This shows that as the processors topology changes, the layout needs to be changed for better performance. Similarly, the same scientific program often runs on different platforms with a different number of processors per node, the results in this experiment show that the program should be tuned based on the machine configuration to achieve better performance.

Iteration	Parameter	Change from	To
0	(use default configuration)		
1	num_iotasks	1	32
2	hmix_momentum_choice	anis	del2
3	hmix_tracer_choice	gent	del2
4	kappa_choice	constant	variable
5	slope_control_choice	notanh	clip
6	hmix_alignment_choice	east	grid
7	state_choice	jmcd	linear
8	state_range_opt	ignore	enforce
9	ws_interp_type	nearest	4point
10	shf_interp_type	nearest	4point
11	sfwf_interp_type	nearest	4point
12	ap_interp_type	nearest	4point

Table 9: Parameter changes through iterations[17]

Besides changing the block dimension, we also apply the Active Harmony system to adjust the parameter values using 32 processors (8 nodes, 4 processors/node) on Hockney at NERSC. The POP program has numerous parameters and there are about 20 parameters that are performance related. There are 2 to 4 possible values for each of the parameters. This makes the search space fairly large. However, the tuning results show that the Active Harmony system can achieve a 12.1% improvement in

[17] Each row shows only the parameter that changes; all the rest parameters remain the same compared to the previous iteration.

execution time after trying just 12 configurations. In addition, the best improvement is 16.7% in execution time after 27 iterations. Table 9 shows how the parameter values have changed for initial 12 iterations. Table 10 shows the values for parameters that are changed before and after tuning. In this application a harmony iteration is one simulation run. Some of these parameters affect scientific results. Ultimately tuning scientific programs requires assistance from experts with domain knowledge to make the tuning results useful and practical. We include these parameters in this experiment since our primary goal was to study the scalability of the harmony system, and not the output of the program being tuned.

Parameter	Default	After tuning
num_iotasks	1	4
hmix_momentum_choice	anis	del2
hmix_tracer_choice	gent	del2
kappa_choice	constant	variable
slope_control_choice	notanh	clip
hmix_alignment_choice	east	grid
state_choice	jmcd	linear
state_range_opt	ignore	enforce
ws_interp_type	nearest	4point
shf_interp_type	nearest	4point
sfwf_interp_type	nearest	4point
ap_interp_type	nearest	4point

Table 10: Parameter tuning

8.3. GS2

GS2 [25, 42] is a physics application, developed to study low-frequency turbulence in magnetized plasma. Its development is funded primarily by the United States Department of Energy, as part of the Scientific Discovery through Advanced Computing (SciDAC) program. It is typically used to assess the microstability of

plasmas produced in the laboratory and to calculate key properties of the turbulence which results from instabilities. It is also used to simulate turbulence in plasmas which occur in nature, such as in astrophysical and magnetospheric systems. Each of these modes uses the same simulation code on radically different time and space scales. The simulation involves billions of mesh points. We tune the program with two different collision modes controlled by the *collision_model* variable (that controls which collision operator may be used in the run).

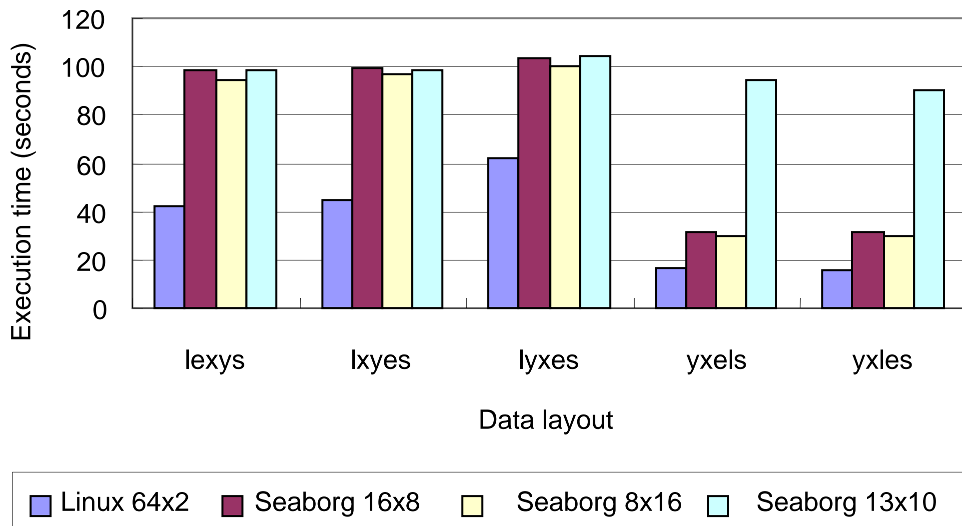


Figure 37: GS2 layout tuning with different environment[18]

The initial analysis was done using 128 processors on NERSC Seaborg (8 nodes, 16 processors/node) system. In order to reduce the execution time, we applied Active Harmony to tune the program. By changing the data layout, the program execution time was reduced from 55.06s to 16.25s (3.4× faster) without collision mode and from 71.08s to 31.55s (2.3× faster) with collision mode. This is for a typical

[18] The dimension A×B following the machine name represents A nodes and B processors per node.

benchmarking run of 10 time steps. Production runs tend to have 1,000 or more time steps.

In Figure 37, we compare the tuning results with different topologies on Seaborg and a result from a Linux cluster. The Linux cluster has 64 nodes; each node is equipped with dual Intel® Xeon™ 2.66GHz processors (with Hyper Threading enabled), 2GBytes main memory and a Myrinet network. In this experiment we consider different data layouts. The data layout is specified with five variables x,y,l,e, and s. The variables x and y are the spatial coordinates; l and e are velocity coordinates and s is the particle specie. The notation indicates the order of the dimensions of the primary 5-dimension array in the simulation. The default data layout used by GS2 is “lxyes”. In the figure, it shows that when the data can be aligned properly with the topology (Linux 64×2, Seaborg 16×8, Seaborg 8×16), using the right data layout (yxles, yxels) will improve the performance significantly.

Benchmarking run with “lxyes” layout		
Tuning method (*negrid*, *ntheta*, nodes)	Tuning time (iterations)	Tuning result – seconds (improvement %)
Default - no tuning (16,26,32)	-	43.7
Original (8,16,14)	7	28.0 (36.0%)
Improved (8,22,8)	8	18.4 (57.9%)
Benchmarking run with “yxles” layout		
Tuning method (*negrid*, *ntheta*, nodes)	Tuning time (iterations)	Tuning result – seconds (improvement %)
Default - no tuning (16,26,32)	-	16.4
Original (8,16,20)	7	16.9 (-3.0%)
Improved (8,22,8)	9	14.8 (9.8%)

Table 11: GS2 tuning result for benchmarking run

Production run with “lxyes” layout		
Tuning method (*negrid*, *ntheta*, nodes)	Tuning time (iterations)	Tuning result – seconds (improvement %)
Default - no tuning (16,26,32)	-	1480.3
Original (12,18,26)	8	266.7 (82.0%)
Improved (10,20,28)	9	244.2 (83.5%)
Production run with “yxles” layout		
Tuning method (*negrid*, *ntheta*, nodes)	Tuning time (iterations)	Tuning result – seconds (improvement %)
Default - no tuning (16,26,32)	-	384.9
Original (14,18,32)	18	239.3 (37.8%)
Improved (10,16,18)	11	240.8 (37.4%)

Table 12: GS2 tuning result for production run

We then proceed to further improve the performance (execution time[19]) on the Linux cluster. Based on the layout tuning result, we used Active Harmony to tune the

[19] Longest execution time for all nodes.

program first using benchmarking runs (10 time steps) and then with input for production runs (1,000 time steps). There are three tunable parameters: *ntheta* (number of grid points per 2 pi segment of field line), *negrid* (energy grid), and the nodes (number of nodes). These parameters were identified by the application developer who is the expert with domain knowledge. The tuning result for the benchmarking runs is summarized in Table 11 and for production runs in Table 12. In the experiments, we compare the tuning time (iterations) and results for the original Active Harmony and the improved version. The three values in the first column are the parameter values after tuning. There is larger improvement when the data layout is “lxyes” compared to a better layout “yxles”. However, starting with the better data layout, Active Harmony achieves a better overall performance result.

The experimental results also suggest that proper data alignment with the number of processors is the major factor deciding the performance. The dimension of the mesh points should be adjusted so the data can be aligned with the number of nodes and thus reduce the communication overhead. Therefore even with “poor” data layout such as “lxyes”, by adjusting the resolution and number or processors, it can still achieve comparable performance results.

While changing *negrid* and *ntheta* may affect the simulation resolution, the dramatic performance gains possible warrant considering using such parameters. Practical scientific program tuning ultimately involves experts with domain knowledge who can make informed choices about these tradeoffs. If these tradeoffs can be quantified, other metrics such as fidelity and scheduling policy can also be

specified and integrated into the objective function so the system can automate this tradeoff.

In order to compare the tuning result with the search space and to understand how well Active Harmony does the tuning, we also explore the whole search space using sampling (i.e., using configurations that are evenly distributed in the whole search space) for the production runs. The performance distribution is shown in Figure 38.

The best configuration found in this "exhaustive sampling" is (*negrid*, *ntheta*, nodes)=(8,16,32) and its performance is 125.8s. However, these are rare points and there are only few configurations (less than 2%) in the whole search space with an execution time less than 200 seconds.

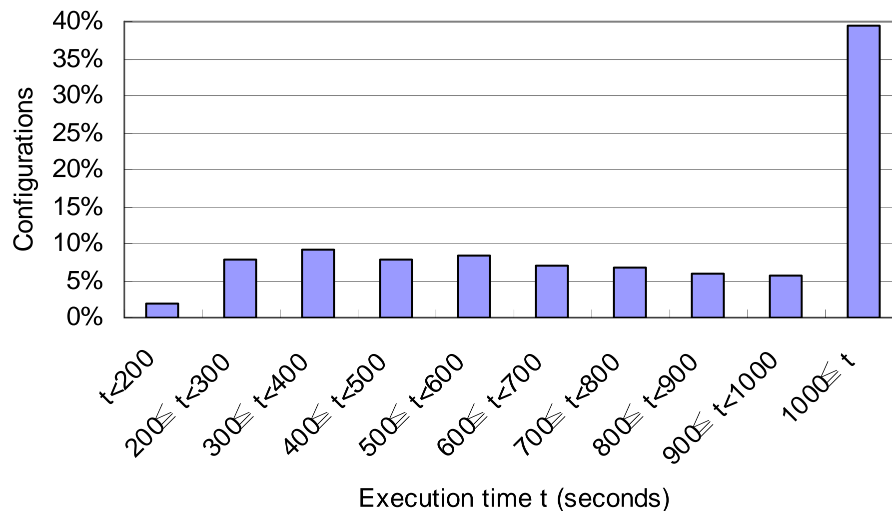


Figure 38: Performance distribution for GS2 configurations

Compared to the performance gathered from exhaustive sampling, the configuration found by Active Harmony is within the top 5% of the configurations. Using the improved Active Harmony tuning kernel, the tuning time is further reduced

from 18 down to 11 iterations (39%). By investing a small amount of effort, Active Harmony helps to tune the GS2 program to produce dramatically better performance.

8.4. Summary

In this chapter, we demonstrated that in order for scientific programs to achieve better performance, it is necessary to adjust the parameter values such as data layout or number of tasks based on the runtime environment such as system capacity or topology. With runtime performance tuning, Active Harmony helps scientific programs to adapt themselves to the environment. The programs can achieve better results, such as reduced execution time, as demonstrated in the experiments.

Chapter 9: Conclusion

In this dissertation we have shown how to make automated performance tuning practical. In the context of the Active Harmony system, we showed how to speed up the tuning process as well as handle the scalability issues when tuning applications with numerous parameters.

To evaluate these ideas, we applied the Active Harmony system to a variety of applications. We first tested it with some simple applications to evaluate our approach. We then applied it to more complicated applications such as a cluster-based web service or scientific programs such as PETSc and GS2. Working with real applications helps to improve the system to make it more practical and robust. This thesis demonstrated the following ideas:

The need for online tuning

We showed that performance tuning is useful and even critical in many applications. When tuned, programs can achieve better results such as higher resolution, better precision or the ability to use a larger data set. Another important reason we need performance tuning is adaptation. The execution environment may change rapidly and there is no single configuration good for all kinds of environments. Manually tuning may not be a feasible solution since it can be extremely time consuming and the system may have changed again before the manual tuning is completed. Therefore, we need the Active Harmony system to do automatic tuning quickly.

The Library Specification Layer

With the Library Specification Layer, we have demonstrated the ability to compose multiple programming libraries with the same or similar functionality to improve the

performance of the programs that use those libraries. The experimental results show the Library Specification Layer helps to select the most appropriate program library and the overall performance is better or close to the best performance of the underlying individual program library. By using the appropriate library, the Library Specification Layer reduces the inversion time for matrices (size 4,500) up to 70%. For 2-D table implementation, using libraries with appropriate data structure can dramatically improve either the access time or the memory space.

Using experience and request characterization

During runtime, the tuning process will benefit from knowing the characteristics of the requests as well as the execution behavior of the application. The tuning system may make use of stored information to help find the appropriate configurations more rapidly.

The Active Harmony system makes use of the experience learned in previous runs. This experience can help to speed up the tuning process since the tuning server may start with a better configuration rather than start from scratch. In the cluster-based web service system tuning, this technique helps to make the tuning process more stable and reduce the tuning time up to 50%.

Prioritizing tool

When tuning numerous parameters in a large system, it is critical to prioritize the parameters by their relative impact on the performance. The tuning should focus on those parameters that most impact performance.

In the cluster-based web service system tuning, the results show that tuning only a few important parameters will improve the performance significantly and reduce the

tuning time up to 72%. This technique is useful when tuning a system with numerous parameters.

Parameter duplication and partitioning

When tuning a large system consisting of many homogeneous nodes, the tuning time may be reduced dramatically (79% for the cluster-based web service project) if we only tune a representative set of parameters. Another approach would be to divide the system being tuned into independent groups and use one tuning server for each group. This method uses about 2/3 of the original tuning time.

The ultimate test of the ideas in this thesis is can we make applications run faster? For the large applications studied in this thesis we were able to improve the cluster-based web service system throughput up to 16% (up to 70% with reconfiguration). With parameter duplication, the tuning time can be reduced up to 80%. For the POP program, the simulation time can be reduced up to 17% and for the GS2 program, Active Harmony makes it run up to 3.4 times faster. We have also shown that Active Harmony scales to tune applications running on hundreds of processors.

Appendix A: Performance Modeling with Synthetic Data

In order to understand the tuning process and further test Active Harmony, we used DataGen [55, 69, 73] to generate synthetic data that mimics characteristics of real applications. The advantage of using synthetic data is to allow us to conduct experiments that may not be conducted easily with real systems. For example, with the same input, configuration, and execution environment, we can perturb the performance output by adding "noise" and observe the impact on the tuning server. The time it takes for the experiment can also be reduced significantly with this method since it may be time consuming to evaluate each configuration with real systems.

DateGen is a rule-based synthetic data generator. The software generates a set of conjunctive normal form rules randomly based on the constraints we specified (e.g., number of conjunctions, number of rules … etc.). Each rule is in the form of $P_i \leftarrow C_a(v_j)$ & $C_b(v_k)$ & $C_c(v_l)$…, where P_i represents the performance result; v_j, v_k, v_l,… are the input variables that represent a set of tunable parameters (i.e., one configuration) and workload characteristics. C_a, C_b, C_c… are Boolean functions that test its input variable (e.g., if $v_j = 3$ or if $2 \leqq v_k < 8$).

To estimate the performance using synthetic data, the program will decide whether a rule is satisfied for the given configuration. A rule is satisfied and performance P_i is returned when all its Boolean function results in the rule are true. The set of rules are carefully generated so that no more than one rule will be satisfied for all possible combinations of input variables (i.e., no conflicts). When no rule is satisfied, it will return the performance result from the closest rule.

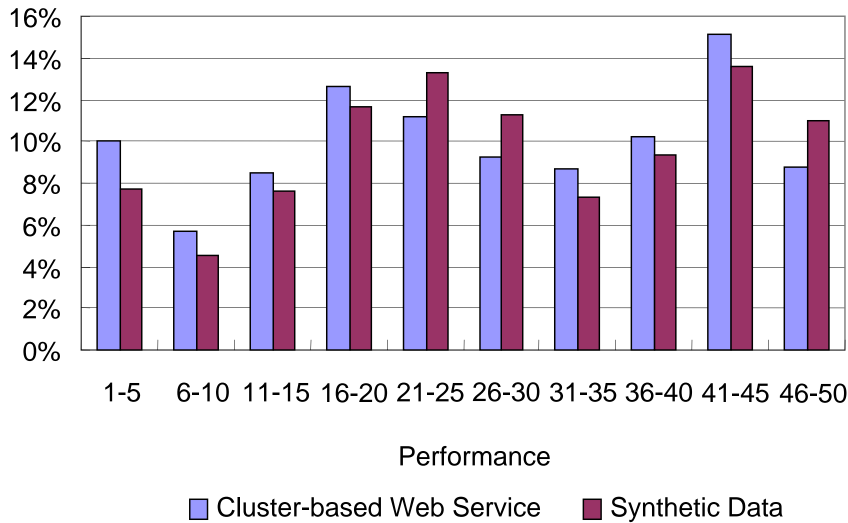


Figure 39: Performance distribution

By comparing the performance obtained through exhaustive search from a clustered-based web service system with a shopping workload (described in Chapter 7) and the synthetic data; we can assess how well our synthetic data emulates a real measured system. Figure 39 shows the closeness of the two performance distributions. The normalized performance (1 to 50, 1 is the worst and 50 is the best) is divided into 10 buckets in the x-axis. The bars show the percentage of points in the search space (y-axis). We use this synthetically generated data to help us to evaluate aspects of the Active Harmony tuning system that are difficult to measure using data from a live system.

Bibliography

1. *The Apache Jakarta Project http://jakarta.apache.org/*.
2. *MySQL Database Server*, MySQL AB *http://www.mysql.com*.
3. *SPEC - Standard Performance Evaluation Corporation http://www.specbench.org*.
4. *Squid Web Proxy Cache http://www.squid-cache.org/*.
5. Abramson, D., et al. *An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics*. in *SC*. 2001. Denver.
6. Anglano, C. *Predicting parallel applications performance on non-dedicated cluster platforms*. in *International conference on Supercomputing*. 1998. Melbourne, Australia.
7. Appelbe, B., et al. *Toward Rapid GeoScience Model Development - The Snark Project*. in *ACES*. 2002.
8. Arabe, J., et al., *Dome: Parallel programming in a heterogeneous multi-user environment*. 1995, Carnegie Mellon University.
9. Bagrodia, R., et al. *Performance prediction of large parallel applications using parallel simulations*. in *ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1999. Atlanta, Georgia, United States.
10. Bailey, D.H., et al., *The NAS Parallel Benchmarks.* International Journal of Supercomputer Applications, 1991. **5**(3): p. 63-73.
11. Benson, S., et al., *Toolkit for Advanced Optimization (TAO) Users Manual*. 2003, Argonne National Laboratory.
12. Berman, F. and R. Wolski. *Scheduling from the Perspective of the Application,*. in *Proceedings of the 5th International Symposium on High Performance Distributed Computing (HPDC-5)*. 1996.
13. Bezenek, T., et al., *Java TPC-W Implementation Distribution http://www.ece.wisc.edu/~pharm/tpcw.shtml*.
14. Block, R.J., S. Sarukkai, and P. Mehra. *Automated performance prediction of message-passing parallel programs*. in *ACM/IEEE conference on Supercomputing*. 1995. San Diego, California, United States.
15. Box, G.E.P., W.G. Hunter, and J.S. Hunter, *Statistics for Experimenters*. 1978, New York: Wiley.
16. Box, G.E.P. and R.D. Meyer, *An Analysis of Unreplicated Fractional Factorials.* Technometrics, 1986. **28**(1): p. 11-18.
17. Brent, R.P., *Algorithms for Minimization Without Derivatives*. 1973, Englewood Cliffs, NJ: Prentice-Hall.
18. Brown, K., *A numerical method for the study of the circulation of the world ocean.* J. Comput. Phys, 1969. **4**(347).
19. César, E., et al., *Dynamic performance tuning supported by program specification.* Scientific Computing, 2002. **10**(1): p. 35-44.
20. Chen, P.M. and D.A. Patterson, *A new approach to I/O performance evaluation: self-scaling I/O benchmarks, predicted I/O performance.* ACM Transactions on Computer Systems (TOCS), 1994. **12**(4): p. 308-339.
21. Chung, I.-H. and J.K. Hollingsworth. *Automated Cluster-Based Web Service Performance Tuning*. in *HPDC-13*. 2004. Honolulu, Hawaii, USA.

22. Chung, I.-H. and J.K. Hollingsworth, *Runtime Selection Among Different API Implementations.* Parallel Processing Letters, 2003. **13**(2).
23. Dantzig, G.B., *Linear programming and extensions*. 1963, Princeton, N.J.: Princeton University Press.
24. Dongarra, J. and e. al., *LAPACK - Linear Algebra PACKage*.
25. Dorland, W., et al., *Electron Temperature Gradient Turbulence.* Phys. Rev. Lett., 2000. **85**(5579).
26. Duda, R.O. and P.E. Hart, *Pattern classification and scene analysis*. 1973, New York: John Wiley & Sons.
27. Dunn, O.J. and V.A. Clark, *Applied Statistics, Analysis of Variance and Regression*. 1974, New York: Wiley.
28. Faerman, M., et al. *Adaptive performance prediction for distributed data-intensive applications*. in *ACM/IEEE conference on Supercomputing*. 1999. Portland, Oregon, United States.
29. Fahringer, T., *Automatic Performance Prediction of Parallel Programs*. 1996: Kluwer Academic Publishers, Boston. 296.
30. Foster, I. and C. Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure*. 1998, Morgan-Kaufmann: San Francisco.
31. Gailly, J.-l. and M. Adler, *zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library*.
32. Geist, A.G., J.A. Kohl, and P.M. Papadopoulos, *CUMULVS: Providing Fault tolerance, Visualization, and Steering of Parallel Applications.* International Journal of Supercomputer Applications and High Performance Computing, 1997. **11**(3): p. 224-35.
33. Gu, W., et al. *Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs*. in *Frontiers '95*. 1995. McLean, VA: IEEE Press.
34. Hicks, C.R., *Fundamental Concepts in the Design of Experiments*. 1973, New York: Holt, Rinehart & Winston.
35. Hollingsworth, J.K. and P.J. Keleher. *Prediction and Adaptation in Active Harmony*. in *The 7th International Symposium on High Performance Distributed Computing*. 1998. Chicago.
36. Jain, R., *The Art of Computer Systems Performance Analysis*. 1991: John Wiley & Sons, Inc.
37. Jin, R. and G. Agrawal. *Performance prediction for random write reductions: a case study in modeling shared memory programs*. in *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2002. Marina Del Rey, California.
38. Kapadia, N.H., J.A.B. Fortes, and C.E. Brodley. *Predictive application-performance modeling in a computational grid environment*. in *The Eighth IEEE Symposium on High Performance Distributed Computing*. 1999. Redondo Beach, CA , USA.
39. Keleher, P.J., J.K. Hollingsworth, and D. Perkovic. *Exposing Application Alternatives*. in *ICDCS*. 1999. Austin, TX.
40. Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi, *Optimization by Simulated Annealing.* Science, 1983. **220**: p. 671-680.

41. Kirkpatrick, S., C.D.G. Jr., and M.P. Vecchi, *Optimization by Simulated Annealing: Quantitative Study.* Journal of Statistical Physics, 1984. **34**: p. 975-986.
42. Kotschenreuther, M., G. Rewoldt, and W.M. Tang, *Comparison of Initial Value and Eigenvalue Codes for Kinetic Toroidal Plasma Instabilities.* Comp. Phys. Comm., 1995. **88**(128).
43. Lenth, R.V., *Quick and Easy Analysis of Unreplicated Factorials.* Technometrics, 1989. **31**(4): p. 469-473.
44. Lim, C.-C., et al. *Performance prediction tools for parallel discrete-event simulation*. in *workshop on Parallel and distributed simulation*. 1999. Atlanta, Georgia, United States.
45. Mason, R.L., R.F. Gunst, and J.L. Hess, *Statistical Design and Analysis of Experiment*. 1989, New York: Wiley.
46. McMahon, F.H., *Livermore FORTRAN Kernerls: A Computer Test of the Numerical Performance.* 1986, Lawrence Livermore National Laboratories: California, USA.
47. Mellor-Crummey, J., D. Whalley, and K. Kennedy, *Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings.* International Journal of Parallel Programming, 2001. **29**(3).
48. Montgomery, D.C., *Design and Analysis of Experiments*. 1984, New York: Wiley.
49. Nelder, J.A. and R. Mead, *A Simplex Methd for Function Minimization.* Comput. J., 1965. **7**(4): p. 308--313.
50. Noble, B.D., et al. *Agile Application-Aware Adaptation for Mobility*. in *16th ACM Symposium on Operating Systems Principals*. 1997.
51. Okamoto, T., *LHa for UNIX 1.1.4i*. 2000.
52. Osterhout, J.K. *Tcl: An Embeddable Command Language*. in *USENIX Winter Conf.* 1990.
53. Parker, S.G. and C.R. Johnson. *SCIRun: a scientific programming environment for computational steering*. in *Supercomputing'95*. 1995. San Diego.
54. Peterson, J.W., *A Numerical Investigation of B´enard Convection in Small Aspect Ratio Containers*. 2004, University of Texas at Austin.
55. Piatetsky-Shapiro, G. and W.J. Frawley, *Knowledge Discovery in Databases*. 1991, Menlo Park, California: AAAI Press.
56. Plackett, R.L. and J.P. Burman, *The Design of Optimum Multifactorial Experiments.* Biometrika, 1946. **33**(4): p. 305-325.
57. Pollard, D., *Strong consistency of k-means clustering.* The Annals of Statics, 1981. **9**: p. 135-140.
58. Press, W.H., et al., *Numerical Recipes in C: The Art of Scientific Computing*. 1993: Cambridge Univ Press.
59. Ribler, R.L., H. Simitci, and D.A. Reed, *The Autopilot Performance-Directed Adaptive Control System.* Future Generation Computer Systems, special issue (Performance Data Mining), 2001. **18**(1): p. 175-187.

60. Ribler, R.L., et al. *Autopilot: Adaptive Control of Distributed Applications*. in *High Performance Distributed Computing*. 1998. Chicago, IL.
61. Riska, A., et al. *ADAPTLOAD: Effective Balancing in Custered Web Servers Under Transient Load Conditions*. in *22 nd International Conference on Distributed Computing Systems (ICDCS'02)*. 2002.
62. Saavedra, R.H. and A.J. Smith, *Analysis of benchmark characteristics and benchmark performance prediction.* ACM Transactions on Computer Systems (TOCS), 1996. **14**(4): p. 344-384.
63. Scholz, W., *Magpar: Parallel Micromagnetics Code*.
64. Schumann, M. *Automatic Performance Prediction to Support Cross Development of Parallel Programs*. in *SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*. 1996. Philadelphia, PA.
65. Simmons, D.M., *Linear programming for Operations Research*. 1972, San Franscisco: Holden-Day.
66. Smith, B., et al., *PETSc - Portable, Extensible Toolkit for Scientific Computation*. 2003.
67. Smith, R.D., J.K. Dukowicz, and R.C. Malone, *Parallel Ocean General Circulation Modeling*. Physica D, 1992. **60**: p. 38-61.
68. Smith, R.D., S. Kortas, and B. Meltz, *Curvilinear coordinates for global ocean models*. 1995, Los Alamos National Laboratory.
69. Smyth, P. and R.M. Goodman, *Rule Induction Using Information Theory.* Piatetsky-Shapiro and Frawley, 1991: p. 159-176.
70. Snavely, A., et al. *A Framework for Application Performance Modeling and Prediction*. in *Supercomputing 2002*. 2002. Baltimore, MD.
71. Soules, C.A.N., et al. *System Support for Online Reconfiguration*. in *Usenix*. 2003.
72. Tapus, C., I.-H. Chung, and J.K. Hollingsworth. *Active Harmony: Towards Automated Performance Tuning*. in *SC'02*. 2002. Baltimore, Maryland.
73. Uthurusamy, R., U.M. Fayyad, and S. Spangler, *Learning Useful Rules from Inconclusive Data.* Piatetsky-Shapiro and Frawley, 1991: p. 141-158.
74. Vraalsen, F., et al. *Performance Contracts: Predicting and Monitoring Grid Application Behavior*. in *Grid Computing - GRID 2001*. 2001. Denver, CO.
75. Whaley, R.C. and J.J. Dongarra. *Automatically tuned linear algebra software (ATLAS)*. in *Supercomputing*. 1998. Orlando, FL.
76. Wolf, J. and P.S. Yu, *On Balancing the Load in a Clustered Web Farm.* ACM Transactions on Internet Technology, 2001. **1**(2): p. 231-261.
77. Wolski, R. *Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service*. in *High Performance Distributed Computing (HPDC)*. 1997. Portland, Oregon: IEEE Press.
78. Yi, J.J., D.J. Lilja, and D.M. Hawkins. *A statistically rigorous approach for improving simulation methodology*. in *The Ninth International Symposium on High-Performance Computer Architecture*. 2003.
79. Ziv, J. and A. Lempel, *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory. **23**(3): p. 337-343.