Using Information from Prior Runs to Improve Automated Tuning Systems

I-Hsin Chung, and Jeffrey K. Hollingsworth
{ihchung, hollings}@cs.umd.edu
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Active Harmony is an automated runtime performance tuning system. In this paper we describe a parameter prioritizing tool to help focus on those parameters that are performance critical. Historical data is also utilized to further speed up the tuning process. We first verify our proposed approaches with synthetic data and finally we verify all the improvements on a real cluster-based web service system. Taken together, these changes allow the Active Harmony system to reduce the time spent tuning from 35% up to 50% and at the same time, reduce the variation in performance while tuning.

1. Introduction

Active Harmony helps programs adapt themselves to the execution environment to achieve better performance. In previous work [17, 21, 29], we showed performance tuning is useful and even critical in many applications including scientific libraries and e-commerce web applications. When tuned, programs can achieve better results such as running faster, better precision, higher resolution or the ability to use a larger data set. When the environment for the systems or the applications changes rapidly, there is frequently no single configuration good for all situations. Manually tuning may not be a feasible since it can be extremely time consuming and the system or the environment may have changed before the manual tuning is completed.

After working with a cluster-based web service system and a scientific simulation program, we saw the need to speed up the tuning process – especially to make use of the experience we learned in the previous program executions and to avoid unnecessary bad performance oscillations during configuration exploration. In this paper, we explain how the Active Harmony tuning server may make use of information such as historical data about the system or application to be tuned.

In addition, 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 potentially longer than the program execution. We present techniques to improve the process when tuning numerous parameters together. Our approach discovers the relative importance of the parameters in advance (prior to a specific execution) so that Active Harmony can focus on performance critical parameters first.

To understand the effectiveness of the Active Harmony tuning system, we use synthetic data to evaluate the improvements made to the system. We show that the parameter prioritizing tool is robust to perturbations (noise) in the performance output. We also demonstrate that focusing on the performance critical parameters helps to reduce the tuning time. Also we show that it takes less time to tune the system when the characteristics of the experience (historical data) are close to the current workload. We then verify all the improvements on a real application, a cluster-based web service system, to show the usefulness of the proposed techniques.

The structure of this paper is organized as follows: Section 2 gives an overview of Active Harmony system. Section 3 describes the concept for prioritizing tunable parameters and its impact on the performance tuning. Section 4 shows the mechanisms we have developed to speed up the tuning process and their evaluations. Section 5 demonstrates the experiments we conduct by applying Active Harmony to synthetic data. We verified all the improvements on a cluster-based web service system in Section 6. Related work is given in Section 7 and Section 8 concludes the paper.

2. Active Harmony

To provide automatic performance tuning, we developed the Active Harmony system [17, 21, 29]. Active Harmony is an infrastructure that allows applications to become tunable by applying minimal changes to the application and library source code. This adaptability provides applications with a way to improve performance during a single execution based on the observed performance. 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).

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 changes them for better performance. The kernel of the adaptation controller is a tuning algorithm. The algorithm is based on the simplex method for a finding a function's minimum value [23]. In the Active Harmony system, we treat each tunable parameter as a variable in an independent dimension. The algorithm

0-7695-2153-3/04 $20.00 (c)2004 IEEE

makes use of a simplex, which is a geometrical figure defined by k+1 connected points in a k-dimensional space. In 2-dimensions, the simplex is a triangle, and for the 3-d space the simplex is a non-degenerated tetrahedron.

The original simplex algorithm assumes a well-defined function and works in a continuous space. However, neither of these assumptions holds in our situation. Thus we have adapted the algorithm by simply using the resulting values from the nearest integer point in the space to approximate the performance at the selected point in the continuous space.

The major challenge when users apply Active Harmony to a large-scale system is the time it takes for tuning. A lengthy tuning process will make the tuning results unusable since the system or the environment may have changed by the time the algorithm finishes. In this paper, we improve Active Harmony by utilizing known information such as previous tuning experience to speed up the tuning process. Also, by investing a small cost testing the tunable parameters, Active Harmony can focus on the performance sensitive parameters and achieve competitive performance results.

3. High-dimensional Search Spaces

When we apply the Active Harmony system to real systems, a practical issue is scalability. As we noted in the previous work [12], tuning can be time-consuming due to the numerous parameters at each node 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 by prioritizing the 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, it 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 initial explorations. This makes the tuning less practical since it takes an extremely long time to converge. In other words, it is not easy to adjust each of the 1,000 parameters to the "appropriate" value after few configuration explorations. 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.

Prioritizing Parameters

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 altogether.

We have developed a standalone software tool that provides the data required for prioritization. It takes all the possible parameters indicated by the user as input. Each parameter is 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 of each of the parameters. For each parameter, the software tool will explore possible values $v_1, v_2, \ldots, v_n$ (based on the distance given) while the rest of the parameters are fixed with the default value. Assume $P_1, P_2, \ldots., P_n$ are the performance results with those different parameter values. We defined the sensitivity of a parameter to be $\left|\frac{\Delta P}{\Delta v'}\right|$, where $\Delta P = P_a - P_b, \Delta v' = v'_a - v'_b$, $P_a = \max_{i=1\ldots n} P_i, P_b = \min_{i=1\ldots n} P_i$. Also each parameter value is normalized (e.g., $v'_a = \frac{v_a - v_{\min}}{v_{\max} - v_{\min}}$) so that parameters with a wide range of values are not given excessive weight. The idea of this sensitivity test is to understand the performance impact when changing one parameter. If the sensitivity value 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 sensitivity value is small, we consider it has lower priority and may be discarded or used later in the tuning.

The design for such a parameter prioritizing tool is based on an assumption that the interaction among parameters is relatively small. The goal for using such a parameter prioritizing tool is to help Active Harmony identify performance critical parameters quickly. If this case is not true, the user may need to use full or fractional factorial experiment design [18, 24] to further investigate the relation among parameters when deciding the importance of parameters.

Assume a large system or program with *n* parameters and *k* different possible values for each parameter that needs to be tuned. 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 discard or leave the

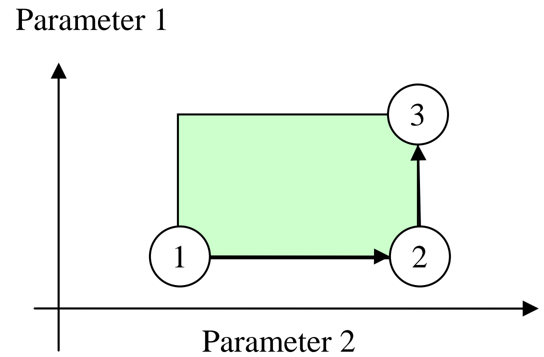
less important ones for later. The parameter prioritizing is done once per new workload and therefore the overhead can be amortized over many runs of the application.

4. Smarter Tuning

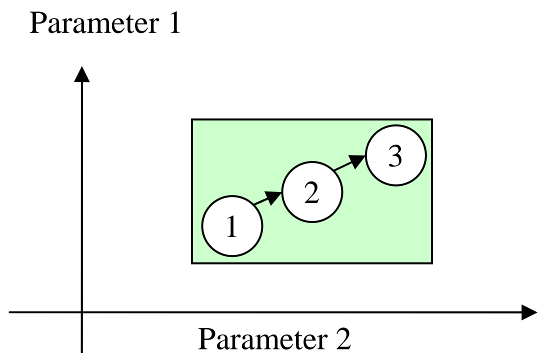
From our previous work we found that there are several drawbacks in our original tuning process in the Active Harmony system. First, in the original implementation, some of the initial configuration explorations test extreme values for the parameters. The performance for this initial stage is usually 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 thus be limited. 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 tuning similar requests or workloads before. Finally, 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 searching starts from scratch every time. In order to overcome these problems, we try to identify some key issues and introduce solutions to improve the tuning.

4.1. Improved Search Refinement

The original Active Harmony tuning server does a decent job in performance tuning. With a 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 what configuration 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. This is due to the characteristics of the Nelder-Mead simplex method. However, from the experience we had in our previous work, we found out that the system usually performs poorly with the parameters at the extreme values. Trying configurations with extreme values often causes the performance of the system to oscillate. Furthermore, for a lot of systems or 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 thrashing. Only the number of connections that is compatible with the system's capacity will yield the best performance. Another example is in a climate simulation program. The computing nodes are divided into groups. Each group of machines is responsible for part of simulation task (e.g., land, ocean, atmosphere). Using a fixed number of nodes for each task will often cause a 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.



(a)



(b)

Figure 1: Improved search refinement for configuration with two parameters (a) Original (b) Improved

In order to solve this problem, we modified the tuning algorithm to replace predefined configurations with parameters at extreme values with values that are closer to the current configuration but which will evenly cover the search space, as shown in Figure 1. 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 1(a), original Active Harmony implementation tries the extreme values for the parameters for the initial exploration. Figure 1(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.

Reducing the magnitude of the performance oscillation 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 efforts or cost when searching for a desirable configuration versus the performance at the desirable configuration should be taken into consideration. Note this is different than most optimization literature. For online 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.

4.2. Using Historical Data

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. We separate this stage from the actual tuning stage and make the normal tuning process two stages.

In order to utilize the experience from the historical data, we must take the associated characteristics of the request into consideration. In other words, the characteristics of the request are also affecting the performance and historical data usage. It will be ideal if the characteristic of the request that the system is currently serving is the same as the historical data that was used for training. However, it is unlikely to have an exact match. Our approach tries to use historical data with characteristics closest to those the system is currently serving. 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. If the input characteristic is similar to previous runs, the system should use previous data layout as the starting point for tuning and this may help to reduce the tuning time.

Besides, one of the important ways to speed up the tuning process is to use knowledge about the characteristics of the input data. The more we understand the characteristics of the input data, the better we can "prepare" the system to be tuned. In the original Active Harmony system implementation, the input data is fed into the system directly. The Active Harmony system tries to change the system configuration to achieve better performance based on the measured performance. It has no knowledge about the input and thus treats the system to be tuned as a "black box". This makes the tuning process time consuming since it 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 the 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. By using the method provided, the data analyzer can decide the characteristics of the input requests. For example, calling a function with the input matrix as the argument; the function might return the matrix structure (e.g., triangular, sparse … etc.). Based on the known experience from the data characteristics database, the data analyzer can make the Active Harmony tuning server adjust the system more efficiently than a blind system. For example, a function is first called to detect matrix structure in the request and later Active Harmony can decide which version of a mathematical library to use. 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 may be treated as a new experience and used to update the data characteristics database for future reference.

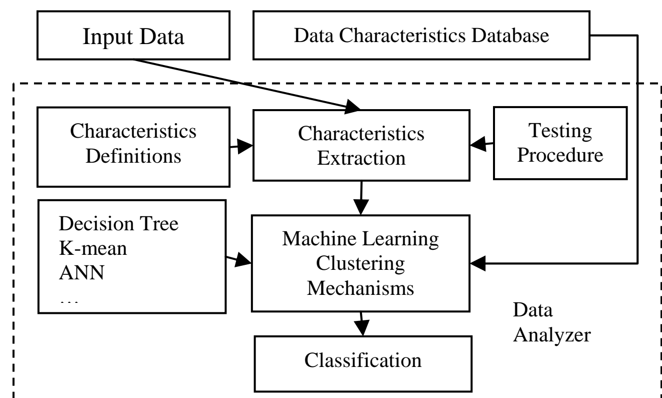


Figure 2: Data Analyzer

The details of the data analyzer are shown in Figure 2. 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 then applies a machine learning clustering approach using predefined methods (such as a decision tree together with known classes defined in the data characteristics database). In the current implementation, we use least square error [14] as the classification mechanism. In this approach, a vector $C_i=(c_{i1},c_{i2},\ldots)$ represents the *i*th 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.

4.3. 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.

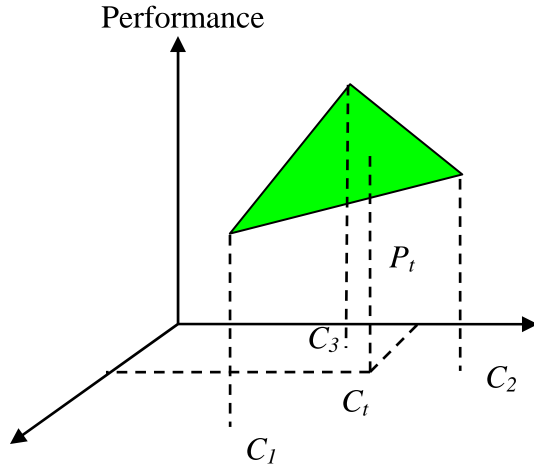


Figure 3: Triangulation estimation for configuration with two parameters

In order to conquer this difficulty, we use triangulation with interpolation or extrapolation to estimate the performance at those "missing" configuration points. 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 *i*th axis is the value for the *i*th 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 a $N+1$ dimensional space.

The example in the Figure 3 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.[1]
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 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. Synthetic Data Experiment

Due to the complexity of measuring real applications and possible anomalies due to execution, we initially evaluated our search heuristic using synthetic data designed to mimic many attributes of measured data. We then proceeded to evaluate the search process using a real application. We use this synthetically generated data to help us to evaluate aspects of the Active Harmony tuning

[1] 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.

system that are difficult to measure using data from a live system.

5.1. Performance Modeling

In order to understand the tuning process and further test Active Harmony, we used DataGen[2] to generate synthetic data with the desired attributes. The software generates a set of conjunctive normal form rules based on the constraints we specified. 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 $2 \leqq v_k < 8$). 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.

We choose to generate synthetic data that is similar to an existing e-commerce web application. Three extra parameters are used to mimic the characteristics of the input workloads: browsing, shopping and ordering. The performance is decided by both the input characteristics and the tunable parameter values.

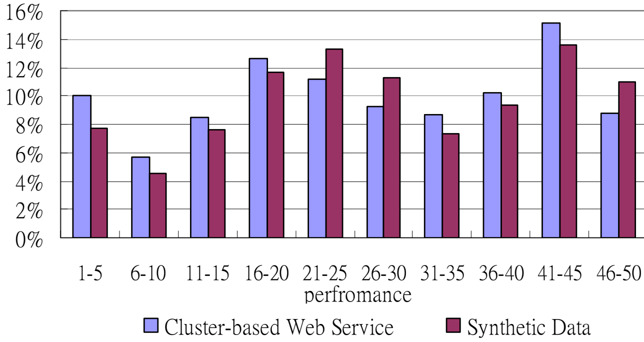


Figure 4: Performance distribution

By comparing the performance obtained through exhaustive search from a clustered-based web service system with a shopping workload (described in the Appendix A) and the synthetic data; we can assess how well our synthetic data emulates a real measured system. Figure 4 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). This shows that the distribution of different performance values. The performance distribution for the synthetic data is approximately the same to its of the real cluster-based web service system.

5.2. Sensitivity Experiment

To evaluate parameter prioritization we ran the parameter prioritizing tool using our 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. In other words, changing the values of those two parameters will not affect the performance. We also perturb the performance output from 0% to ±25% with a uniform random distribution. This is because in real systems, given exactly the same environment and input, the performance output will not always be the same for two different runs. We first consider the sensitivity of our parameter prioritization with regard to this run to run variation in application performance.

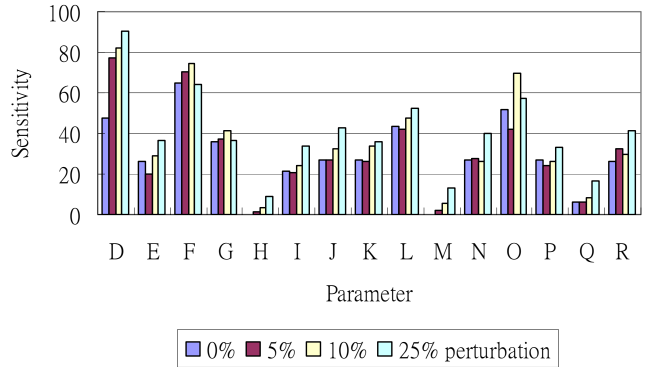


Figure 5: Parameters sensitivity of the synthetic data

The result is shown in Figures 5 and 6. In Figure 5, the parameter prioritizing technique helps the user to identify that parameter H and M are less relevant to the performance. This matches the parameter specification of the synthetic data.

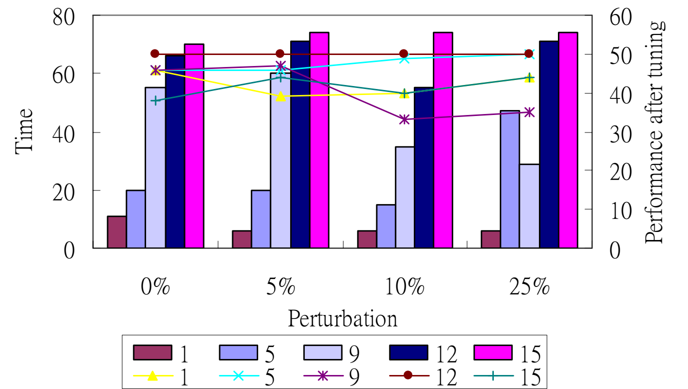


Figure 6: Tuning using only *n* most sensitive parameter(s) of the synthetic data. Bars show tuning time, and points show the resulting application performance.

In Figure 6, 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 bars in the figure show the time it takes for the tuning and the lines indicate the tuning results. The figure shows that a larger performance output perturbation (10%, 25%) does affect the tuning process. For those cases with less perturbation, 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%).

Another interesting point is tuning more sensitive parameters does not increase the tuning time linearly (e.g. comparing tuning time for n=12 and n=15). This may be simply because those added parameters are less sensitive to the performance and thus converge faster compared to more sensitive parameters.

5.3. Historical Data Experiment

In order to test the effectiveness for performance tuning using historical data, we carefully design the experiment as follows: the 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 4.2).

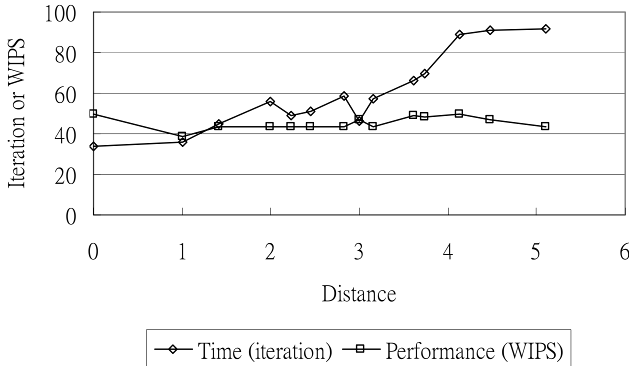


Figure 7 : Tuning using different experiences

Figure 7 shows the relation between the experience workload A' and current workload A. In this figure, the x-axis shows the distance between the current configuration A and the stored workload A'. The distance between two workload characteristics is calculated using distance in Euclidean space. This data is again taken from synthetic data generated for a system like the cluster-based web service system presented in Appendix A. When the characteristics of the historical data are close to those of the current workload, it takes less time to tune the system. The more they differ from each other, the longer it takes for Active Harmony to tune the system to achieve similar performance (tuning result). 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.

6. Cluster-Based Web Service Application

We verify our proposed tuning approaches described in previous sections on a large-scale real application with many parameters. In this study, we apply the improved Active Harmony system to a cluster-based web service system from our previous work [12].

6.1. Cluster-Based Web Service System

A cluster-based web service system consists of a collection of machines. The machines are separated into sets (or tiers). Each tier of machines is focused on serving different parts of a request. The incoming requests are handled in a pipeline fashion by different tiers.

In many web services today, there are (conceptually, at least) three tiers: 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 database. Those results are presented to the client through the presentation tier. The third tier is the database, which holds the information accessible via the network. It is the backend that provides reliable data storage and transaction semantics.

In the project, we try to improve the overall system performance by automatic tuning across all tiers using the Active Harmony system. A more detailed description of the environment on which we conduct the experiment is given in Appendix A. The application we are focusing on tuning is an implementation of the TPC-W benchmark [5]. It is a transactional web benchmark designed to emulate operations of an e-commerce site. The performance metric used in this benchmark is Web Interactions per Second (WIPS), where a higher WIPS value is better. A brief introduction to the benchmark is given in Appendix A.

6.2. Parameter Sensitivity

We apply our parameter prioritizing tool to 10 parameters in the cluster-based web service system. Figure 8 shows when the system faces different workloads; the value for each parameter will have different importance to the system 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 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.

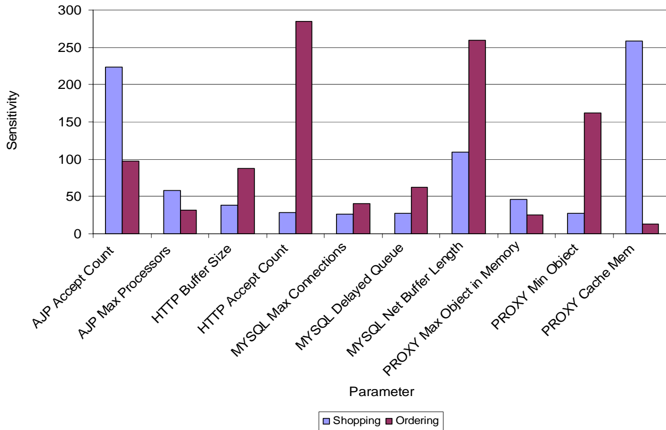


Figure 8: Parameter sensitivity in the cluster-based web service system

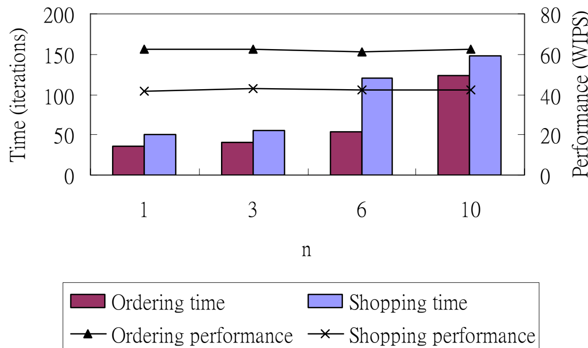


Figure 9: Tuning using only *n* most sensitive parameter(s) of the cluster-based web service system

We now consider the question of how many parameters need to be tuned. We consider the top n most important parameters based on our sensitivity analysis. We vary n from 1 to 10. Figure 9 shows that only using a limited number of parameters can reduce tuning time significantly. The bars show the time it takes for the tuning process and the lines indicates the tuning results. The results show that when tuning a system with numerous parameters, it is helpful to first spend some efforts identifying performance related parameters from those that are less performance relevant. Only tuning those performance related parameters will save a significant amount of tuning time (up to 71.8%) while compromising a little of the performance in the tuning result (less than 2.5%).

6.3. Improved Search Refinement

In order to verify our proposed approach to make the tuning process more stable, we also evaluate the modifications discussed in Section 4 to the Active Harmony tuning algorithm kernel. We hope with these modifications, the tuning process will be more stable and therefore the time spent in those iterations with poor performance will not dominate the tuning time.

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 the Table 1. The performance column shows the tuning result. The convergence time represents the tuning time 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 the tuning kernel improvement 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. Also, when the cluster-based web service system is facing a shopping workload, the proposed configuration exploration even helps to reduce the magnitude of the initial bad performance oscillation. However, this may vary from case to case since it depends on the shape of the performance function. For example, the performance function for the ordering workload has more "bad performance" configurations that do not lie on the boundaries of parameter values. Therefore avoiding parameters with extreme values does not improve significantly in reducing the magnitude of the initial bad performance oscillation.

	Shopping workload		
	Performance WIPS	Convergence time (iterations)	Worst performance[2] WIPS
Original implementation	63	90	20
After improvement	60	58	27
	Ordering workload		
	Performance WIPS	Convergence time (iterations)	Worst performance WIPS
Original implementation	79	74	29
After improvement	80	46	29

Table 1: Tuning process summary

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

6.4. Tuning with Experience

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 for different kind of web interactions. We expect different workloads will have different relative weights on each of the web interactions. By observing the frequency distribution for web interactions, the data analyzer can characterize the workload that the system is serving. During the actual 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 distribution of web interactions) of the incoming request and 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.

In this experiment, we have the system serving a workload A with and without first training using historical data (which is never seen by Active Harmony server). In the Table 2, when the system serves shopping and ordering workloads, the tuning process is smoother and the performance converges fast (56% for the shopping workload and 17% 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 in the tuning process compared to nine bad performance iterations when without prior histories. And for the ordering workload with prior histories, there are three bad performance iterations in the tuning process compared to eleven bad performance iterations when without prior histories.

	Shopping workload		
	Convergence time (iteration)	Performance (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 (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 2: Tuning process with and without prior histories

7. Related Work

Performance contracts [30] allows the level of performance expected of system modules to be quantified and then measured during execution. The application includes intrinsic metrics 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 operations. For N metrics, the trajectory through N-dimensional metric space is called application signature. The execution signature reflects both the application demands on the resources and the response of the resource to those demands.

Using application signatures together with a convolution method helps to model the performance more rapidly [28]. Snavely, A., et al. present a framework for performance modeling and prediction that is faster than cycle-accurate simulation, more informative than simple benchmarking, and is shown 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, 7, 9, 11, 15, 16, 19, 22, 25, 26]. Thomas Fahringer [16] introduced 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.

A performance prediction study by Kapadia et al. also extends this work to distributed systems [20]. The paper evaluates 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 runtime input parameters. This project focuses on the accuracy of the performance prediction. However, perusing maximal predictive accuracy may not be appropriate given the variability in a grid computing environment.

These performance prediction and estimation projects are focused on characterizing the application behavior on systems. Active Harmony uses performance estimation that is based on past experience to reduce the tuning time and make the tuning process more stable.

Direction set (Powell's) method [10] is a 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. This method is similar to the Active Harmony parameter prioritizing tool which explores one parameter at a time. However, this method does not explore the relation among parameter while the Nelder-Mead simplex method (Active Harmony tuning kernel) does.

Linear programming and the simplex method [13] 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 Simmons [27]. 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 major challenge when applying linear programming and the simplex method in performance tuning is the domain knowledge required for the system to be tuned. Active Harmony provides a general solution for performance tuning and little or none domain knowledge is required.

8. Conclusion

In order to speed up a black-box tuning process, it is worthwhile to utilize information from prior runs, and to dedicate some time running the system with test configurations to understand the search space. In this paper, we introduced and evaluated a technique, parameter prioritizing, when tuning large systems with numerous parameters. When tuning numerous parameters in a large system, it is critical to prioritize the parameters by their relative impact to the performance. This helps the tuning focus on the performance-related parameters. The experimental results show that tuning only few most sensitive parameters will save a dramatic amount of tuning time while compromising little to none of the application's performance.

We improve the Harmony tuning algorithm by changing the configurations used for initial explorations. This helps the system to be tuned reduce the time spent exploring extreme values with poor performance. We also improved the Active Harmony system by making use of the experience learned in previous runs. During the runtime, the tuning process will benefit from knowing the characteristics of the requests. The tuning system may make use of the stored information to help find the appropriate configurations more rapidly. The experience can help to speed up the tuning process since the tuning server may start with a better configuration rather than start from scratch. When tuning the cluster-based web service system, all these techniques help to reduce the time spent in the initial unstable performance stage from 35% up to 50% and make the tuning process more stable and smoother (i.e., fewer configurations with bad performance). In the future, we plan to apply Active Harmony to a variety of applications to assure the quality of the improvements.

With all these improvements, we demonstrated that the Active Harmony system is a feasible and useful tool in performance tuning for large parameter spaces. The techniques are helpful especially for long-running programs and systems. It is worthwhile to spend a small amount of time in characterizing the inputs as well as prioritizing the parameters (e.g., 20-30 iterations for the cluster-based web service system). This initial investment of time yields a dividend of a major reduction in the time spent tuning.

Acknowledgement

This work was supported in part by NSF award EIA-0080206, DOE Grants DE-FG02-01ER25510 and DE-CFC02-01ER254489.

References

1. *The Apache Jakarta Project http://jakarta.apache.org/.*
2. *DataGen 3.0 http://datasetgenerator.com.*
3. *MySQL Database Server*, MySQL AB *http://www.mysql.com*.
4. *Squid Web Proxy Cache http://www.squid-cache.org/.*
5. *TPC Benchmark W http://www.tpc.org/tpcw*.
6. Anglano, C. *Predicting parallel applications performance on non-dedicated cluster platforms*. in *International conference on Supercomputing*. 1998. Melbourne, Australia.
7. 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.
8. Bezenek, T., et al., *Java TPC-W Implementation Distribution http://www.ece.wisc.edu/~pharm/tpcw.shtml*.
9. 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.
10. Brent, R.P., *Algorithms for Minimization Without Derivatives*. 1973, Englewood Cliffs, NJ: Prentice-Hall.
11. 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.
12. Chung, I.-H. and J.K. Hollingsworth. *Automated Cluster-Based Web Service Performance Tuning*. in *HPDC-13*. 2004. Honolulu, Hawaii, USA.
13. Dantzig, G.B., *Linear programming and extensions*. 1963, Princeton, N.J.: Princeton University Press.

14. Duda, R.O. and P.E. Hart, *Pattern classification and scene analysis*. 1973, New York: John Wiley & Sons.
15. Faerman, M., et al. *Adaptive performance prediction for distributed data-intensive applications*. in *ACM/IEEE conference on Supercomputing*. 1999. Portland, Oregon, United States.
16. Fahringer, T., *Automatic Performance Prediction of Parallel Programs*. 1996: Kluwer Academic Publishers, Boston. 296.
17. 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.
18. Jain, R., *The Art of Computer Systems Performance Analysis*. 1991: John Wiley & Sons, Inc.
19. 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.
20. 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.
21. Keleher, P.J., J.K. Hollingsworth, and D. Perkovic. *Exposing Application Alternatives*. in *ICDCS*. 1999. Austin, TX.
22. 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.
23. Nelder, J.A. and R. Mead, *A Simplex Methd for Function Minimization.* Comput. J., 1965. **7**(4): p. 308--313.
24. Plackett, R.L. and J.P. Burman, *The Design of Optimum Multifactorial Experiments.* Biometrika, 1946. **33**(4): p. 305-325.
25. 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.
26. 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.
27. Simmons, D.M., *Linear programming for Operations Research.* 1972, San Franscisco: Holden-Day.
28. Snavely, A., et al. *A Framework for Application Performance Modeling and Prediction*. in *Supercomputing 2002*. 2002. Baltimore, MD.
29. Tapus, C., I.-H. Chung, and J.K. Hollingsworth. *Active Harmony: Towards Automated Performance Tuning*. in *SC'02*. 2002. Baltimore, Maryland.
30. Vraalsen, F., et al. *Performance Contracts: Predicting and Monitoring Grid Application Behavior*. in *Grid Computing - GRID 2001*. 2001. Denver, CO.

Appendix A

TPC-W Benchmark

The major performance metric we use when tuning the cluster-based web service system is the TPC-W benchmark. The TPC-W is a transactional web benchmark designed to mimic operations of an e-commerce site. The workload explores a 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 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.

Environment

Hardware		Software	
Processor	Dual AMD Athlon 1.67 GHz	Operating System	Linux 2.4.18smp
Memory	1Gbyte	TPC-W benchmark	Modified from the PHARM [8]
Network	100Mbps Ethernet	Proxy Server	Squid 2.5 [4]
No. of machines	10	Application Server	Tomcat 4.0.4 [1]
		Database Server	MySQL 3.23.51 [3]

Table 3: Experiment Environment

The summary of the environment used for our experiment is shown in the Table 3. The 10 machines used include the ones running emulated browsers and the servers for proxy, application and database services. Each machine is equipped with dual processors, 1 Gbyte memory and runs Linux as the operating system. We select Squid as the proxy server, Tomcat as the 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 scale factor is 10,000 items. In other words, the number of the items that the store sells in the experiment is approximately 10,000.

Appendix B

Parameter Restriction

In our previous work, we developed the resource specification language which is used to communicate between the system to be tuned and 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.

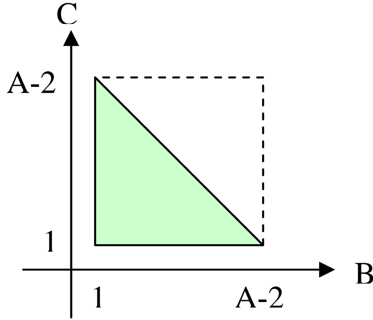


Figure 10: Search space reduction by parameter restriction

We improved the resource specification language so it can support basic functional relations among parameters. In other words, it allows that the value of one parameter is a function of another parameter value. This will 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 the 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 rest 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 tasks) as shown in the Figure 10 . Whenever the server needs to "figure out" the next configuration, it decides the value for the parameter B first. And then it will decide the value for the parameter C based on it is of B. By doing this, we are able to reduce the high-dimensional search space (the dashed area in the Figure 10).

If A=10 in this example, part of the resource specification language before parameter restriction can be:

{ harmonyBundle B { int {1 10 1}}}
{ harmonyBundle C { int {1 10 1}}}
{ harmonyBundle D { int {1 10 1}}}

After we apply the parameter restriction technique to reduce the search space, this part of the resource specification language is simplified as:

{ harmonyBundle B { int {1 ***8*** *1} }}*
{ harmonyBundle C { int {1 ***9-$B*** *1} }}*
*{ harmonyBundle D { int {****10-$B-$C 10-$B-$C*** *1} }}*

The last line for parameter D specification can be further removed since the value for parameter D is decided after the values for parameter B and C are known. 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).

We implement and apply this technique when tuning the number of each type of connectors on the HTTP & application server in the cluster-based web service system. On the HTTP & application server, 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. When tuning the library, Active Harmony needs to decide how the matrix with *k* rows is partitioned into *n* blocks. Without knowing the relations among row partitions, the size for each row partition ranges from 1 to *k*. However, the search space can be significantly reduced if the size for row partition *i* is decided after previous *i*-1 row partition sizes are decided. The program in the resource specification language would look like the follows:

{ harmonyBundle P_1 { int {1 k-n+1 1} }}
{ harmonyBundle P_2 { int {1 k-n+2-$\$P_1$ 1} }}
…
{ harmonyBundle P_{n-1} { int {1 k-1-($\$P_1$+$\$P_2$+…+$\$P_{n-2}$) 1} }}

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.