Active Harmony: Towards Automated Performance Tuning

Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth
crt@cs.caltech.edu, {ihchung, hollings}@cs.umd.edu

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

In this paper, we present the Active Harmony automated runtime tuning system. We describe the interface used by programs to make applications tunable. We present the Library Specification Layer which helps program library developers expose multiple variations of the same API using different algorithms. The Library Specification Language helps to select the most appropriate program library to tune the overall performance. We also present the optimization algorithm used to adjust parameters in the application and the libraries. Finally, we present results that show how the system is able to tune several real applications. The automated tuning system is able to tune the application parameters to within a few percent of the best value after evaluating only 11 out of over 1,700 possible configurations.

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, 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 all configurations.

The transient, rarely repeatable behavior of Grid [4] 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. Automatic program library selection provides a framework to help with this goal; it helps to tune the application during runtime execution by monitoring the underlying library performance and switching underlying program library as needed. This is an important step toward automated performance tuning in the Grid computing.

To meet the needs of this type of computing environment, we have been developing the Active Harmony system that allows runtime switching of algorithms and tuning of library and application parameters. We have also developed a set of runtime tuning algorithms that help to intelligently set these parameters at runtime to tune the overall performance of an application.

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 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 parameter to what algorithm is being used (e.g., heap sort vs. quick-sort).

2. System Design

Figure 1 shows the Active Harmony automated runtime tuning system. The Library Specification Layer provides uniform API to library users by integrating different 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 library to improve the decision for future usage of the program library.

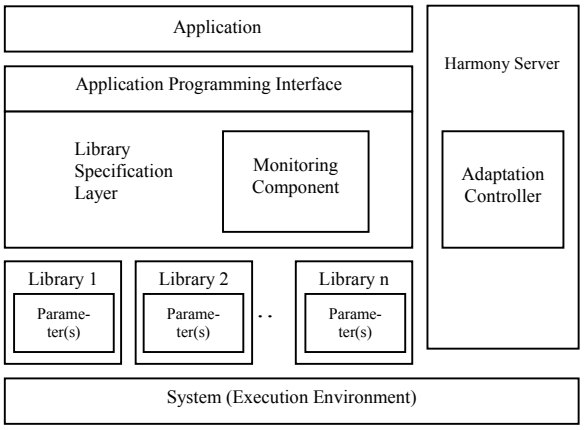


Figure 1: Active Harmony automated tuning system

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 Adaptation Controller is written in Tcl

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

for an easier interpretation of the information provided by applications and resources.

3. Library Specification Layer

The role of the library specification layer is to help the application select the most appropriate underlying algorithm. To achieve this goal, the layer 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.

Selection among the available libraries is done via the Adaptation Controller. During the execution, the library specification layer will send the characteristics of requests to the Adaptation Controller. The Adaptation Controller returns the suggested underlying algorithm to use according to the result of its decision process. In the current implementation, when the Adaptation Controller is selecting an algorithm, it tries to explore all possible algorithms for at least a brief period of time. Based on observed performance, an appropriate algorithm is selected.

The most common performance metrics are CPU time or memory space used. Library developers can specify multiple program library performance metrics in the library specification language. 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 an 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 appropriate underlying program library.

The key idea of algorithm selection is that usage patterns of a library may require different types of requests to underlying algorithms or data structures. For example, in an implementation of a table abstraction, the data structure used and the workload pattern will both affect the performance. In a workload with high search rate and high data element location density, arrays would outperform linked lists. However, if the data element density is sparse and memory space is critical, the linked data structure should be chosen. The characteristics of requests play an important role in selecting appropriate underlying program library. Requests of specific characteristics (that can be defined by Library developers) may favor one implementation of the functionality over another, and can be either variables with primitive data types or expressed by basic Boolean operations on those variables.

The Library Specification Language currently supports libraries written in both C and Fortran. It 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 upper layer users. It also provides the indirection to allow runtime switching among the different implementations. The runtime switching code includes the ability for library writer to specify mapping functions that can change the underlying data structures (such as going from a dense to sparse matrix representation).

4. The Resource Specification Language (RSL)

In this section we will present the prototype RSL and the shortcomings that came along with it, followed by the new version of the RSL which removes the limitations that we encountered.

The current Harmony RSL is based on the initial version [8, 9] but provides more flexibility. The RSL is implemented on top of the Tcl scripting language [14]. 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 tuning interface through the use of the Tk toolkit.

The new 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. 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.

```
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.

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 was required to evaluate the behavior of applications based on different objective functions.

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, 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.

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

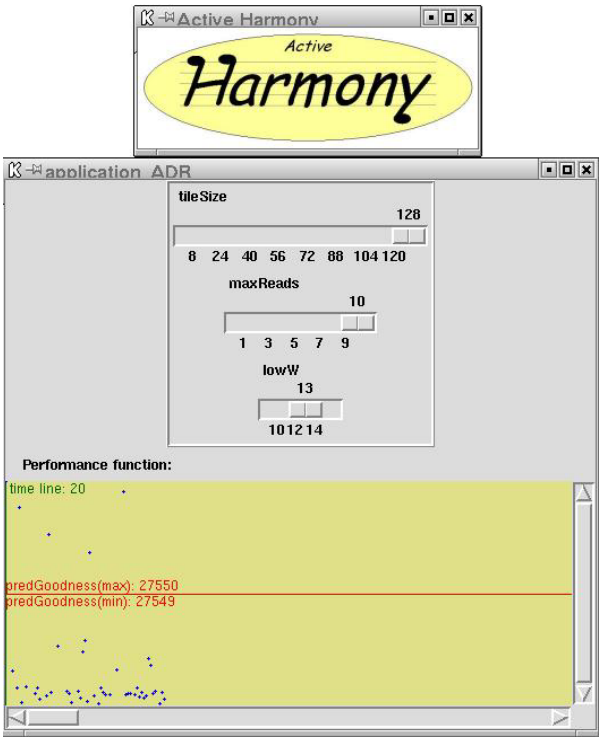


Figure 3: Harmony User Interface.

Figure 4 shows the changes made in the main program of the 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 sever.

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*.

6. Parameter Tuning Algorithm

In an earlier version of the Harmony system [8], we 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.

However, 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 and the operating system. 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 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 were mostly interested in avoiding those k-tuples for which the performance was 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 first and second order derivatives. The algorithm that we developed is based on the simplex method for finding a function's minimization [12].

```
/* 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(performace_result);
/* end of program main loop */

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

Figure 4: Changes required for a typical application.

The algorithm makes use of a simplex, which is a geometrical figure defined by k+1 connected points in a k-dimensions space. In 2-dimensions, the simplex is a triangle, and for the 3-d space the simplex is a non-degenerated tetrahedron. The Nelder-Mead simplex method [12] 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 replaces the worst point, making the simplex move towards the extreme. In our case the algorithm slips down the valley towards the minimum.

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.

7. Experimental Results

We conducted a series of experiments to evaluate the design and its performance. We first present results for the library specification layer. Then we use the Harmony server to tune the selected library through iterations to improve the overall performance.

7.1 Algorithm Tuning Experiments

We evaluated the library specification layer by first applying it to a simple data structure abstraction, then applied it two commonly used math libraries. All of our tests were run on Redhat Linux with kernel 2.4.0 on a Pentium-III 667MHz with 384 MB main memory. The metric used for all four test cases are the time to complete the requests.

***Matrix Inversion*:** The first set of program libraries consists of two matrix inversion routines from LAPACK [3]. The major characteristic of the matrix is a Boolean indicating if the matrix is triangular. If the matrix is triangular, the specialized triangular matrix inversion library will have better performance. Otherwise, a general matrix inversion library must be used. The result is shown in Figure 5; 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. Later for each request, the library specification layer detects whether the request matrix is triangular and if so, the library specification layer will invoke the matrix inversion library optimized for triangular matrices. Otherwise, the library specification layer will just apply it to the general matrix inversion library.

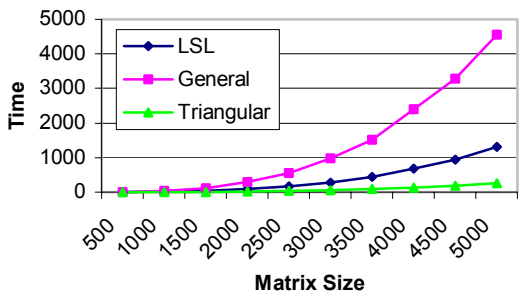


Figure 5: Matrix Inversion Test Case

***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 structure based on API usage patterns. Two program libraries are implemented using linked lists and arrays. Each data structure has its advantages: linked lists take less memory space for storage but longer for insert, delete, and search operations; arrays take more memory for storage but are more efficient for insert, delete, and search operations.

The first test uses the time to complete each operation as the performance metric. The result is shown in the Figure 6. 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 as the underlying supporting library. The second test uses memory utilization as the main metric. The result is shown in Figure 7. As expected, the performance of the program with the library specification layer is close to the performance of the program with linked list library. 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.

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.

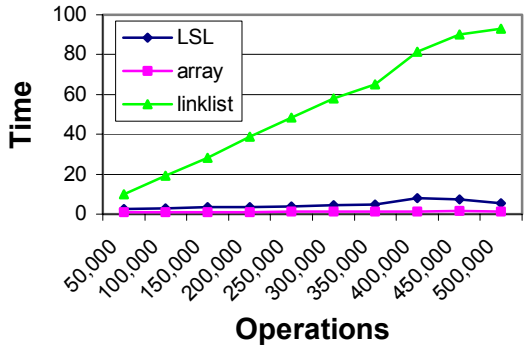


Figure 6: 2-D Table with Time metric

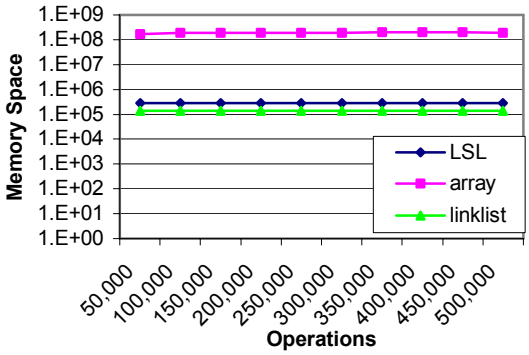


Figure 7: 2-D Table with Memory Space metric

7.2 Parameter Tuning Experiments

Once the library specification layer selects the underlying program library, the Harmony server tunes both the application and the library to improve the performance. Due to the complexity of measuring real applications and possible anomalies due to execution, we initially evaluated our search heuristic using several well-defined functions. We then proceeded to evaluate the search process using a real application.

We tested the effectiveness of our search algorithm on three benchmark functions. The three functions were: a sombrero hat function, a quadratic function, and Rosenbrock's parabolic valley.

Sombrero hat function: We used the following variant of the sombrero hat function: defined for integer values of x and y from –20 to 20, which has a global minimum in (0,0). Figure 8(a) illustrates the path followed by the algorithm to reach the minimum. The minimum was found after only 8 function evaluations. The sombrero function is interesting because it has a series of local minima, which makes it quite difficult to optimize, but our algorithm was able to avoid getting stuck in any of them. The search performed by our algorithm found the global minimum. The solid surface shows the function, and the line shows the evolution of the search algorithm as it tries different points.

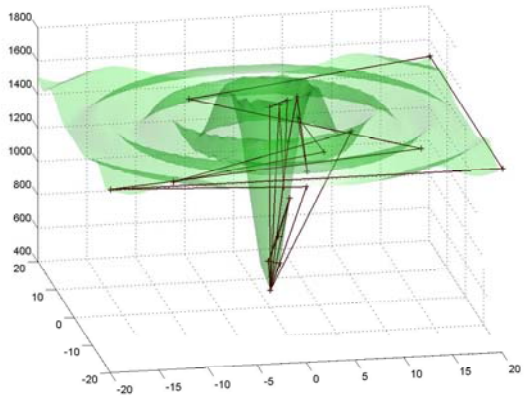
Quadratic Function: Another function that we analyzed was: $Z = (x^2 + y^2) * 10$. The function, defined on the range from –20 to 20, with integer values for x and y, has a minimum in (0,0). The path followed by the algorithm towards the minimum is presented in Figure 8(b). The number of iterations needed to reach the minimum was 15.

Rosenbrock's parabolic valley[1]*:* The only function that caused a problem was the Rosebrock's function defined by: $Z = 100*(y-x^2)+(1-x)^2$, in the integer subspace for x and y between –20 and 20. The function has a minimum of 0 in (x,y)=(1,-1). Our algorithm terminated its search at (x,y)=(4,19), where the function has a value of 909. This happened because of the rounding scheme that we used, and possibly because of the initial simplex as well. However, taking into account that the values of the function in the above range were of the order of 1.7 x 10^6, the value of 909, which was found after only 27 iterations, is a very good approximation, within 0.05% of the minimum. This was considered an acceptable result for the algorithm because, as we already mentioned above, the main goal was to avoid bad performance while aiming for the optimum. The search is illustrated in Figure 8(c).

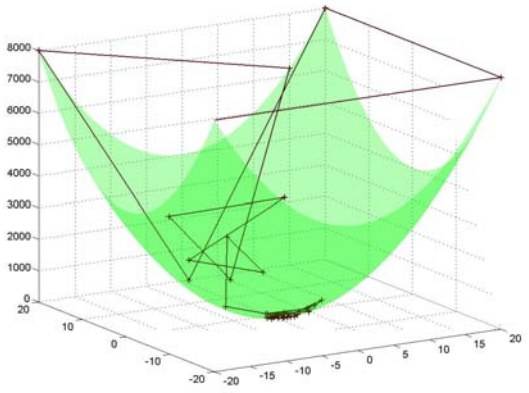
7.3 Real Application Tuning

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[5] and LHa[13]. Both of these libraries are general-purpose, lossless data-compression libraries. The deflation algorithm used is a variation of LZ77[21]. They both use hash table and binary trees, plus Huffman encoding to compress the data strings.

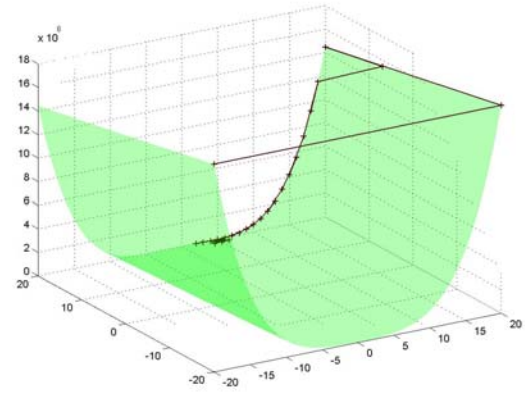
There are two performance metrics used in the experiment: time and size ratio. The time is the process time used to compress the data file. The size ratio is to compare the file size before and after the compression. Each library has its own tunable variables. The *BUFLEN* in the zlib adjusts the buffer used in reading the data strings. It will affect the time to compress a file. 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 made slight modifications so those two variables are tunable during the application execution. The files being compressed are composed of randomly selected UNIX files system with a predefined file size.



(a)



(b)



(c)

Figure 8: The search algorithm applied to three known functions.

In the experiment, we focus on automatic tuning using a specific library. We set the library selection to be manual in the library specification layer. Instead of optimizing an individual performance metric, we select an objective function that combines both space and

[1] Although we tried to understand the behavior of the Nelder-Mead algorithm in the context of the three benchmark functions that we presented above, we were unable to locate the real problem with Rosenbrock's function. It appears that by changing the starting simplex the convergence would change too. We could see improvement in the results but the data we presented above is based on the automated simplex generation we implemented in Harmony. It was only after we ran these tests that we found out that the analysis of the Nelder-Mead algorithm for the quadratic function is an open problem and that there is ongoing research on why this algorithm tends to work very well in practice [11].

time as metrics to show the tuning ability for the Harmony system. The results show the tradeoff between the buffer size and the performance metric.

Figure 9 shows the tuning process for LHa compression library. The buffer size (compared to the default value) converges quickly a few iterations. Figure 10 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.

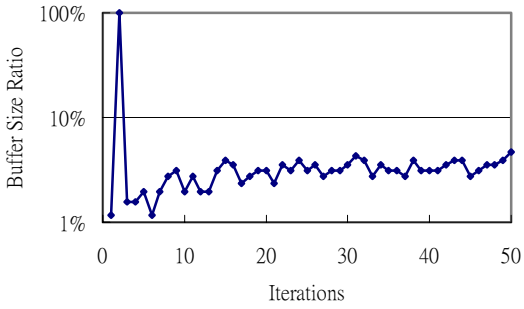


Figure 9: LHa: Changes of Buffer Size

Figure 11 and Figure 12 shows the results when the library specification layer chooses to use the zlib compression library. Figure 11 shows the size of the buffer used by the zlib compression library through the iterations. The buffer size converges after 15 iterations. Figure 12 shows the tuning results. The buffer size is more than 100 times smaller than the original one while the process time increased about 15%.

There are two major factors that influenced the tuning ability of the Harmony server. The first is the selection of objective function. The objective function should have its minimum value at the desired operation point. A function that is "smooth" and with few "local minima" is preferred to help speed up tuning. The second is the "step" *d* used for search algorithm; that is, the minimum distance between current value and the next value of the tunable variable.

Figure 13 shows the tuning process with different *d*. If the *d* is too small, the Harmony server is affected by the "noise" of the performance data since the tuning server is too sensitive to the performance result. Therefore in some cases, the value of the tunable variable may never converge. 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.

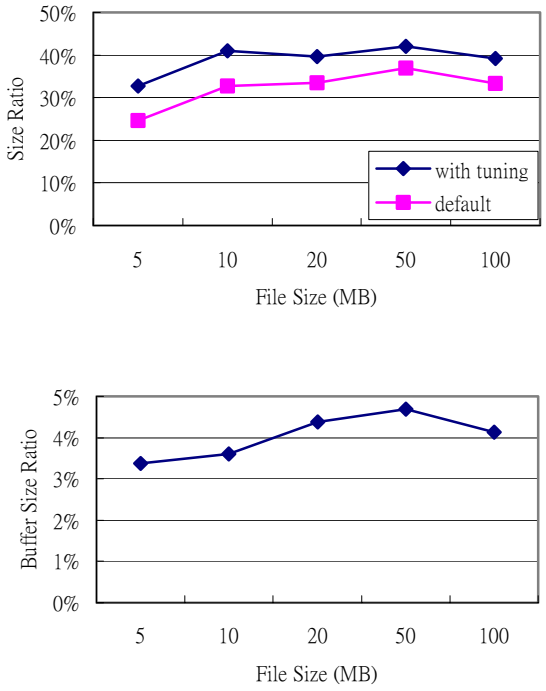


Figure 10: LHa: Buffer size and performance after tuning

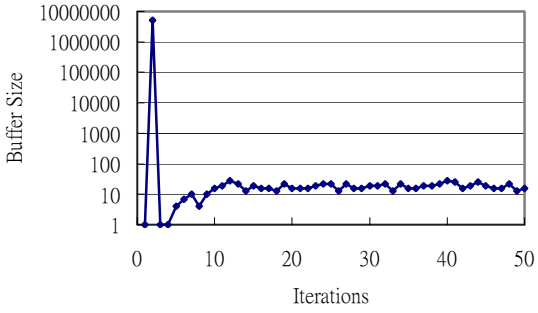


Figure 11: zlib: Changes of Buffer size

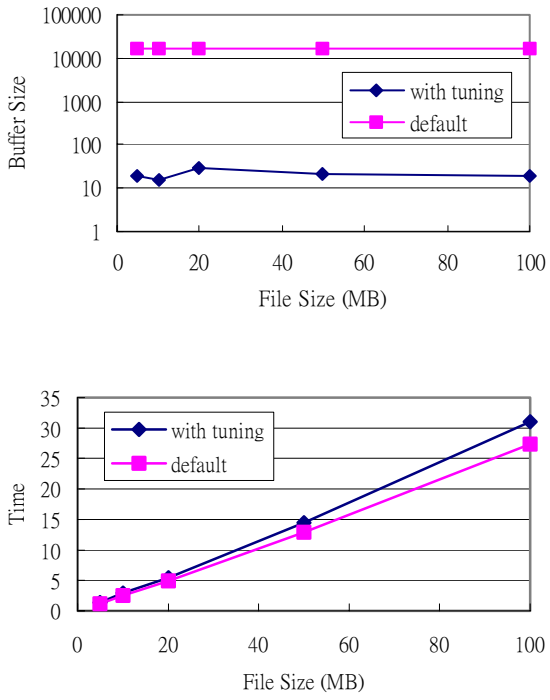


Figure 12: Buffer size and performance after tuning

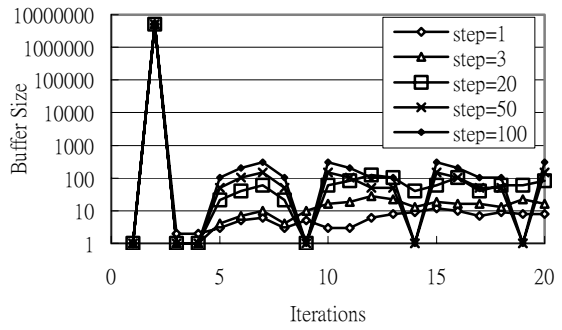


Figure 13: Different step *d*

I/O Intensive Application: To evaluate our optimization system using a real application, we selected a 3-d volume reconstruction application [2] built on top of the Active Data Repository (ADR) middleware [10]. The 3-d volume reconstruction application uses digital images of a space to reconstruct the objects that are visible from the various camera angles. The ADR is an infrastructure that integrates storage, retrieval and processing of large multi-dimensional data sets. ADR provides the user with operations including index generation, data retrieval, scheduling across parallel machines, and memory management. The data is accessed through range queries (i.e., extract all data within a specified region of space). A range query is processed in two steps: query planning followed by query execution. As part of query execution, input and output items are mapped between coordinate systems and the data is aggregated to generate the final result. During the processing phase a temporary dataset, called the accumulator, is created to hold the results of the query being processed.

Because ADR is middleware used to build multiple applications including the Harmony calls in the ADR code makes every application built on top of ADR tunable. The parameters we used were:

tileSize represents the size of the memory tile that is used by the ADR back-end to store information before it is aggregated. It is the size of the tiles which the accumulator will be partitioned if it does not fit into memory. This parameter greatly influences query planning and execution since it is somewhat analogous to the block size in a computational code that has been blocked (tiled) to fit into a cache.

lowWatermark is the upper bound of the number of pending reads and number of ready reads that were issued to the disk in order to resolve a certain query.

maxReads is the maximum number of reads issued in order to resolve the current query if the number of pending read operations and the number of ready read operations are below the lowWatermark.

The original version of the volume reconstruction application used values for the parameters provided by the ADR designers. To harmonize the application, we added calls to expose these parameters to the system. The environment in which we ran the experiments was a Linux cluster of 16 machines, each with two 450 MHz processors connected by 100 Mbps Ethernet.

To see how the Active Harmony infrastructure improves the running time of the Volume Reconstruction application, we created a random set of queries that were submitted to the ADR back-end. First we ran them using the original version of the ADR; then, the harmonized version. Figure 14 below presents the improvement we obtained in the processing time of each of the queries.

The Active Harmony system sped up query processing by up to 50% for the set of 70 random queries that we generated. However, the average improvement was about 10%. This is due to the fact that some of the queries that had the greatest speed-ups were very short, compared with others for which the improvement was less than 10%. The performance improvement for the longest query, which took about 10 minutes to be completed was about 18%.

Since we did not know the shape of the performance curve and thus what the best value is, another set of experiments was conducted to compare the behavior

of the Active Harmony adaptation system to the brute force search for the best parameter values. For the purpose of the exhaustive search we submitted to the ADR back-end the same query for each tuple of parameter values. We then recorded the value of the performance function for each of the 1680 tuples. To test the behavior of the Active Harmony we submitted the same query 2000 times to the ADR back-end.

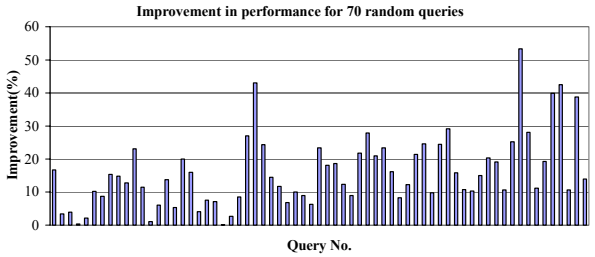


Figure 14: Performance improvement for the volume reconstruction application.

The brute force algorithm recorded values for the performance function of up to 25% slower than the optimum, while the range of values explored by the Active Harmony system was within 5% of the minimum. The minimum was reached by our system by exploring only 11 tuples (out of the almost 1,700 different possible tuples). Figure 15 below presents these results. The axes of the graph are as follows: the vertical one represents the performance function, while the horizontal ones are the tileSize and the lowWatermark. The values obtained for different values of the third parameter: maxReads are stacked one on top of the other in the graph. The lighter points in the graph are from the exhaustive search and are spread on the entire value space. The darker points (lower left corner) trace the path followed by the tuning and they are concentrated near the minimum.

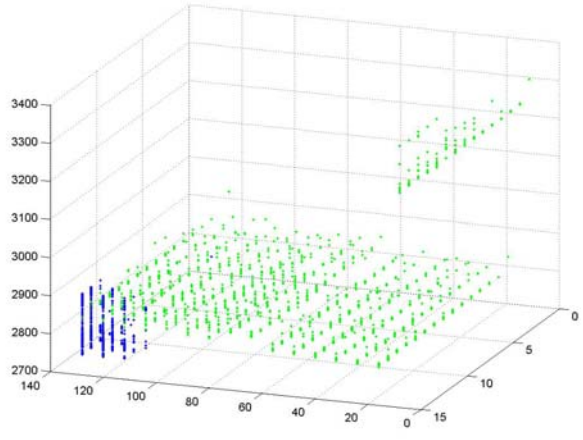


Figure 15: Performance curve (via exhaustive search) for the volume reconstruction application.

8. Related Work

There are several projects that seekin to develop techniques to allow applications to be responsive to their available resources or that allow them to be tuned at runtime. Computational Steering [6, 7, 15-18] provides a way for users to alter the behavior of an application under execution. Harmony's approach is similar since applications provide hooks to allow their execution to be changed. Many computational steering systems are designed to allow the application semantics to be altered; for example, 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.

One exception is Autopilot [17, 18], which allows applications to be adapted in an automated way. Sensors extract quantitative and qualitative performance data from executing applications, and provide requisite data for decision-making. Autopilot uses a fuzzy logic to automate the decision making process. Their actuators execute the decision by changing parameter values of applications or resource management policies of underlying system. Harmony differs from Autopilot in that it tries to coordinate the use of resources by multiple libraries and applications.

The ATLAS [19] project has developed automatically tuned linear algebra libraries. They develop a methodology for the automatic generation of high efficient basic linear algebra routine for a given 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 for a wide range of architectures. The difference between our work and ATLAS is that our work focuses on general applications that use program libraries rather than that of a specific library.

Another approach is application level scheduling. AppLeS [1] 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 [20] is used to forecast the network performance and available CPU percentage to AppLeS agents. 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, 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.

Prior work in the active Harmony project [8, 9] concentrated on the API to make applications tunable, and in defining an interface to express the different options via a Resource Specification Language. This paper extends that work be providing an improved search algorithm (rather than a simple greedy approach). In addition, we describe the new Algorithm Adaptation layer that provides the glue code to allow existing (slightly) different APIs to be "harmonized."

9. Conclusion

This paper presented an infrastructure for tuning distributed applications for better performance and an optimization algorithm based on the simplex method for function minimization.

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. The results demonstrate that Active Harmony can bring significant improvement to distributed applications and permit new ways to adapt applications to dynamic environments.

References

1. Berman, F. and R. Wolski. *Scheduling from the perspective of the application*. in *Proceedings of 5th IEEE International Symposium on High Performance Distributed Computing*. 1996. Syracuse, NY, USA 6-9 Aug. 1996.
2. Borovikov, E., A. Sussman, and L. Davis. *An Efficient System for Multi-Perspective Imaging and Volumetric Shape Analysis*. in *Workshop on Parallel and Distributed Computing in Image Processing, Video Processing Multimedia (PDIVM'20001)*. 2001: IEEE Computer Society Press.
3. Dongarra, J. and e. al., *LAPACK - Linear Algebra PACKage*.
4. Foster, I. and C. Kesselman, eds. *The Grid: Blueprint for a New Computing Infrastructure*. 1998, Morgan-Kaufmann: San Francisco.
5. Gailly, J.-l. and M. Adler, *zlib - A Massively Spiffy Yet Delicately Unobtrusive Compression Library*.
6. Geist, A.G., J.A. Kohl, and P.M. Papadopoulos, *CUMULVS: Providing Fault tolerance, Visualization, and Seering of Parallel Applications.* International Journal of Supercomputer Applications and High Performance Computing, 1997. **11**(3): p. 224-35.
7. Gu, W., et al. *Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs*. in *Frontiers '95*. 1995. McLean, VA: IEEE Press.
8. 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.
9. Keleher, P.J., J.K. Hollingsworth, and D. Perkovic. *Exposing Application Alternatives*. in *ICDCS*. 1999. Austin, TX.
10. Kurc, T., et al. *Querying Very Large Multi-dimensional Datasets in ADR*. in *Proceedings of SC99*. 1999. Orlando, FL: ACM Press.
11. Lagarias, J.C., et al., *Convergence properties of the Nelder-Mead simplex method in low dimensions.* SIAM Journal for Optimizations. **9**(1): p. 112-147.
12. Nelder, J.A. and R. Mead, *A Simplex Methd for Function Minimization.* Comput. J., 1965. **7**(4): p. 308--313.
13. Okamoto, T., *LHa for UNIX 1.1.4i*. 2000.
14. Osterhout, J.K. *Tcl: An Embeddable Command Language*. in *USENIX Winter Conf.* 1990.
15. Parker, S.G. and C.R. Johnson. *SCIRun: a scientific programming environment for computational steering*. in *Supercomputing'95*. 1995. San Diego.
16. Reed, D.A., et al. *The next frontier: interactive and closed loop performance steering*. in *ICPP Workshop on Challenges for Parallel Process*. 1996. Bloomingdale, Ill.
17. 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.
18. Ribler, R.L., et al. *Autopilot: Adaptive Control of Distributed Applications*. in *High Performance Distributed Computing*. 1998. Chicago, IL.
19. Whaley, R.C. and J.J. Dongarra. *Automatically tuned linear algebra software (ATLAS)*. in *Supercomputing*. 1998. Orlando, FL.
20. 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.
21. Ziv, J. and A. Lempel, *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory. **23**(3): p. 337-343.