Exposing Application Alternatives

Peter J. Keleher Jeffrey K. Hollingsworth Dejan Perkovic
(keleher|hollings|dejanp)@cs.umd.edu

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

Abstract

We present the design of an interface to allow applications to export tuning alternatives to a higher-level system. By exposing different parameters that can be changed at runtime, applications can be made to adapt to changes in their execution environment due to other programs, or the addition or deletion of nodes, communication links etc. An integral part of this interface is that an application not only expose its options, but also the resource utilization of each option and the effect that the option will have on the application's performance. We discuss how these options can be evaluated to tune the overall performance of a collection of applications in the system. Finally, we show preliminary results from a database application that is automatically reconfigured by the system from query shipping to data shipping based on the number of active clients.

1. Introduction

Meta-computing, the simultaneous and coordinated use of semi-autonomous computing resources in physically separate locations, is increasingly being used to solve large-scale scientific problems. By using a collection of specialized computational and data resources located at different facilities around the world, work can be done more efficiently than if only local resources were used. However, the infrastructure to support this type of global-scale computation is not yet available.

Both meta-computer environments and the applications that run on them can be characterized by distribution, heterogeneity, and changing resource requirements and capacities. These attributes make static approaches to resource allocation unsuitable. Systems need to dynamically adapt to changing resource capacities and application requirements in order to achieve high performance in such environments.

Active Harmony is a software architecture that manages distributed execution of computational objects in dynamic environments. Most previous approaches to adapting applications to dynamic environments required applications to be solely responsible for reconfiguration to make better use of existing resources. While the actual means that applications use to reconfigure themselves is certainly application-specific, we argue that the decisions about when and how such reconfigurations occur are more properly made in a centralized resource manager.

Moving policy into a central manager serves two purposes. First, it accumulates detailed performance and resource information into a single place. Better information often allows better decisions to be made. This information could conceivably be provided directly to each application. Problems with this approach include duplicated effort and possible contention from the conflicting goals of different applications. More importantly, however, a centralized manager equipped with both comprehensive information on the system's current state, and knobs with which to reconfigure running applications, can adapt any and all applications in order to improve resource utilization.

For example, consider a parallel application whose speedup improves rapidly up to six nodes, but improves only marginally after that. A resource allocator might give this application eight nodes in the absence of any other applications since the last two nodes were not being used for any other purpose. However, when a new job enters the system, it could probably make more efficient use of those two nodes. If decisions about application reconfiguration are made by the applications, no reconfiguration will occur. However, a centralized decision-maker could infer that reconfiguring the first application to only six nodes will improve overall efficiency and throughput, and could make this happen.

This paper describes the application interface to the Harmony resource allocator. Harmony's application interface allows applications to export *tuning options*. Tuning options are sets of mutually exclusive application configuration alternatives. For example, a parallel application might be able to exploit either four or eight nodes. A database system might be able to execute queries at either the client or the server. Options include information about both resource requirements, and resulting performance. Making choices explicitly available to the system allows the resource manager more freedom in matching applications to resources. Applications written to Harmony's interface also allow reconfigurations to occur during application execution, and thus enable changing existing resource allocations in order to accommodate new applications.

The Active Harmony system is targeted at long-lived and persistent applications. Examples of long-lived applications include scientific code and data mining applications. Persistent applications include file servers, information servers, and database management systems. We these applications because they persist long enough for the global environment to change, and hence have higher potential for improvement. Our emphasis on long-lived applications allows us to use on relatively expensive operations such as object migration since these operations can be amortized across the life of the object.

The focus of this paper is on the interface between Harmony and applications. Specifically, we ask the following questions:

1) *Can we build an API that is expressive enough to define real-world alternatives?*
2) *How can we specify the relationships between the requirements?*
3) *Can the Harmony system use this API to improve the behavior of applications during execution?*

2. Harmony structure

Harmony's architecture is shown in Figure 1. The major components are the following:

Adaptation controller: The adaptation controller is the heart of the system. The controller must gather relevant information about both the applications and the environment, project the effects of proposed changes (such as migrating an object) on the system, and weigh competing costs and expected benefits of making various changes.

Active Harmony provides mechanisms for applications to export tuning options, together with information about the resource requirements of each option, to the adaptation controller. The adaptation controller then chooses among the exported options based on more complete information than is available to individual objects. A key advantage of this technique is that the system can tune not just individual objects, but also entire collections of objects. Possible tuning criteria include network latency and bandwidth, memory utilization, and processor time. Since changing implementations or data layout could require significant time, Harmony's interface includes a frictional cost function that can be used by the tuning system to evaluate if a tuning option is worth the effort required.

Metric interface: The metric interface provides a unified way to gather data about the performance of applications and their execution environment. Data about system conditions and application resource requirements flow into the metric interface, and on to both the adaptation controller and individual applications.

Tuning interface: The tuning interface provides a method for applications to export tuning options to the system. Each tuning option defines the expected consumption of one or more system resources. The options are intended to be "knobs" that the system can use to adjust applications to changes in the environment. The main concern in designing the tuning interface is to ensure that it is expressive enough to describe the effects of all application tuning options.

Resource management in meta-computing is a complex task. While we do not have any final answers, we feel our approach promises to enable more comprehensive resource policies than previously possible.

3. Application to system API

This section describes the interface between applications and the Harmony adaptation controller (hereafter referred to as "Harmony"). Applications use the API to specify tuning options to Harmony. Harmony differs from previous systems like Matchmaker [19] and the Globus RSI [4] in that it uses simple performance models to guide allocation decisions. These models require more application information than previous systems. While previous systems might accept requests for "a machine and a network," Harmony requires each resource usage to be specifically quantified. This is necessary because Harmony uses performance prediction to optimize an overall objective function, usually system throughput. Estimates of resource usage are employed to build simple performance models that can then predict the interaction of distinct jobs. Performance models give Harmony the ability to make judgements of the relative merits of distinct choices.

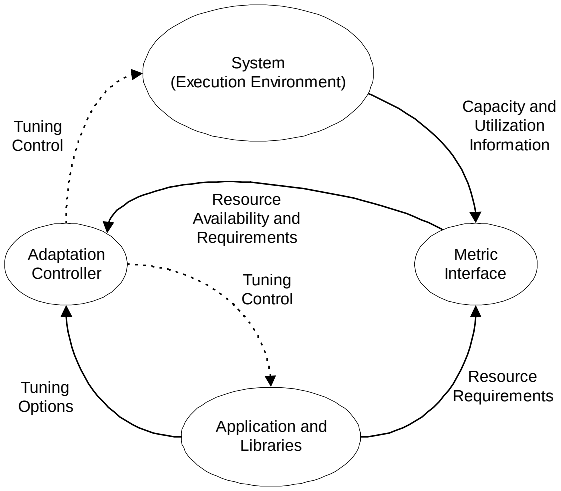


Figure 1: Major Components of Active Harmony

Harmony's decision-making algorithm can also consider allocation decisions that require running applications to be reconfigured. Hence, applications that are written to the Harmony API periodically check to see whether Harmony has reconfigured the resources allocated to them.

We therefore require our tuning option API to have the following capabilities. First, it must be able to express mutually exclusive choices on multiple axises. These options can be thought of as a way of allowing Harmony to locate an individual application in n-dimensional space, such that the choice corresponding to each dimension is orthogonal.

Second, the interface must provide a means to specify the resource requirements of different options. For example, we need to be able to represent that a given option requires *X* cycles and *Y* amount of network bandwidth. However, the "*X* cycles" is problematic to express. Cycle counts are only meaningful with reference to a particular processor, such as "20 minutes of CPU

Tag	Purpose
harmonyBundle	Application bundle.
node	Characteristics of desired node (e.g., CPU speed, memory, OS, etc.)
link	Specifies required bandwidth between two nodes.
communication	Alternate form of bandwidth specification. Gives total communication requirements of application, usually parameterized by the resources allocated by Harmony (i.e., a function of the number of nodes).
performance	Override Harmony’s default prediction function for that application.
granularity	Rate at which the application can change between options.
variable	Allows a particular resource (usually a node specification) to be instantiated by Harmony a variable number of times.
harmonyNode	Resource availability.
speed	Speed of node relative to reference node (400 MHz Pentium II).

Table 1: Primary tags in Harmony RSL

time on a UltraSparc 5.” To circumvent this problem, we specify CPU requirements with reference to an abstract machine, currently a 400 MHz Pentium II. Nodes then express their capacity as a scaling factor compared to the reference machine. Similar relative units of performance have been included in systems such as PVM[7].

Third, the tuning options must be able to express relationships between entities. For example, we need to be able to express “I need two machines for 20 minutes, and a 10Mbps link between them.” Note that the link can be expressed relative to the machines, rather than in absolute terms. The system must therefore be able to understand the topology of the system resources, such as network connections between machines, software services, etc. Possible infrastructures that we can leverage are Remos [14] and Matchmaker [19].

Fourth, the interface must be able to represent the granularity at which the modification can be performed. For example, an iterative data-parallel HPF Fortran application might be able to change the number of processors that it exploits at runtime. However, this adaptation can probably only be performed at the completion of an outer loop iteration.

Fifth, we need to express the frictional cost of switching from one option to another. For example, once the above data-parallel HPF application notices the change request from Harmony, it still needs to reconfigure itself to run with the new option. If two options differ in the number of processors being used, the application will likely need to change the data layout, change the index structures, and move data among nodes to effect the reconfiguration. This frictional cost is certainly not negligible, and must be considered when Harmony makes re-allocation decisions.

Finally, each option must specify some way in which the response time of a given application choice can be calculated by the system. This specification may be either explicit or left to Harmony. In the latter case, Harmony uses a simple model of computation and communication to combine absolute resource requirements into a projected finishing time for an application. An explicit specification might include either an expression or a function that projects response time based on the amount of resources actually allocated to the application.

3.1 The Harmony RSL

The Harmony resource description language (RSL) provides a uniform set of abstractions and syntax that can be used to express both resource availability and resource requirements. The RSL consists of a set of interface routines, a default resource hierarchy, and a set of predefined tags that specifies quantities used by Harmony. The RSL is implemented on top of the TCL scripting language [16]. Applications specify requirements by sending TCL scripts to Harmony, which executes them and sends back resource allocation descriptions.

Several things make TCL ideal for our purposes. First, it is simple to incorporate into existing applications, and easily extended. Second, TCL lists are a natural way to represent Harmony'a resource requirements. Finally, TCL provides support for arbitrary expression and function evaluation. The latter is useful in specifying parametric values, such as defining communication requirements as a function of the number of processors. More to the point, much of the matching and policy description is currently implemented directly in TCL. Performance is acceptable because recent versions of TCL incorporate on-the-fly byte compilation, and updates in Harmony are on the order of seconds not micro-seconds.

The following summarizes the main features of the RSL:

Bundles: Applications specify *bundles* to Harmony. Each bundle consists of mutually exclusive options for tuning the application’s behavior. For example, different options might specify configurations with different numbers of processors, or algorithm options such as table-driven lookup vs. sequential search.

Resource requirements: Option definitions describe requested high-level resources, such as nodes or communication links. High-level resources are qualified by a number of tags, each of which specifies some characteristic or requirement that the resource must be able to meet. For example, tags are used to specify how much memory and how many CPU cycles are required to execute a process on a given node.

Performance prediction: Harmony evaluates different option choices based on an overall objective function. By default, this is system throughput. Response times of individual applications are computed as simple combinations of CPU and network requirements, suitably scaled to reflect resource contention. Applications with more complicated performance characteristics, provide simple performance prediction models in the form of TCL scripts.

Naming: Harmony uses option definitions to build namespaces so that the actual resources allocated to any option can be named both from within the option definition, and from without. A flexible and expressive naming scheme is crucial to allowing applications to specify resource requirements and performance as a function of other resources. More detail on the naming scheme is presented below.

Table 1 lists the primary tags used to describe available resources and application requirements. The "harmonyBundle" function is the interface for specifying requirements. The "harmonyNode" function is used to publish resource availability.

3.2 Naming

Harmony contains a hierarchical namespace to allow both the adaptation controller and the application to share information about the current instantiated application options and about the assigned resources. This namespace allows applications to describe their option bundles to the Harmony system, and also allows Harmony to change options.

The root of the namespace contains application instances of the currently active applications in the system. Application instances are two part names, consisting of an application name and a system chosen instance id. The next level in the namespace consists of the option bundles supported by the application. Below these are hierarchical resource requirements, currently just nodes and links. Nodes contain sub-resources such as memory, CPU, I/O etc. Links currently contain only bandwidth estimates. The fully qualified name would be:

```
    appliction.instance.bundle.option.
        resourcename.tagname
```

For example, if the client in Figure 3 was assigned instance ID 66 by Harmony, the tag describing the memory resources allocated to the client of the data-shipping option would:

```
    DBclient.66.where.DS.client.memory.
```

3.3 Simple parallel application

We next present show the expressiveness of Harmony's interface. Our first example is shown in Figure 2 (a). "Simple" is a generic parallel application that runs on four processors. There are two high-level resource requests. The first specifies the required characteristics of a worker node. Each node requires 300 seconds of computation on the reference machine and 32 Mbytes of memory. The "replicate" tag specifies that this node definition should be used to match four distinct nodes, all meeting the same requirements. Second, we use the "comunication" tag to specify communication requirements for the entire application. Since specific endpoints are not given, the system assumes that communication is general and that all nodes must be fully connected.

3.4 Variable parallelism

Our second application, "Bag", is a parallel application that implements an application of the "bag-of-tasks" paradigm. The application is iterative, with computation being divided into a set of possibly differently-sized tasks. Each worker process repeatedly requests and obtains tasks from the server, performs the associated computations, returns the results to the server, and requests additional tasks. This method of work distribution allows the application to exploit varying amounts of parallelism, and to perform relatively crude load-balancing on arbitrarily-shaped tasks.

Bag's interface with Harmony is shown in Figure 2 (b). There are three additional features in this example. First, bag uses the "variable" tag to specify that the application can exploit 1, 2, 4, or eight worker processes. Assuming that the total amount of computation performed by all processors is always the same, the total number of cycles in the system should be constant across different numbers of workers. Hence, we parameterize "seconds" on the "workerNodes" variable defined in the "variable" tag.

```
harmonyBundle Simple - {
 {-          {node "worker"
                   {hostname    "*"}
                   {os          "linux"}
                   {seconds     "300"}
                   {memory      32
                   {replicate   4}}
             {communication "2 + 2 * 4"}
 }}
```

(a) simple parallel application

```
harmonyBundle bag howMany {
     {default        {node       "worker"
       {hostname      "*"}
       {os            "linux"}
       {seconds       "200/workerNodes"}
       {memory        {32}}}
     {variable worker "workerNodes" 1 2 4 8}
     {communication "2 + 2 * workerNodes * workerNodes"}
     {performance {[interp workerNodes {1 1e5} {4 3e4} {8 2e4}]

}}
```

(b) bag-of-tasks application

Figure 2: Harmonized applications

Second, we use the “communication” tag to specify the overall communication requirements as a function of the number of processors assigned. The bandwidth specified by the communication tag defines that bandwidth grows as the square of the number of worker processes. Hence, “Bag” is an example of a broad domain of applications in which communication requirements grow much faster than computation.

Third, we use the “performance” tag to tell Harmony to use an application-specific prediction model rather than its default model. The “performance” tag expects a list of data-points, that specify the expected running time of the application when using a specific number of nodes. Rather than requiring the user to specify all of the points explicitly, Harmony will interpolate using a piecewise linear curve based on the supplied values.

Our model could be improved. For example, a better way of modeling communication costs is by CPU occupancy on either end (for protocol processing, copying), plus wire time [3]. If this occupancy is significant, cycles on all worker processes would need to be parameterized based on the amount of communication, which includes the quadratic expression. This is not difficult or computationally expensive, but less convenient.

3.5 Client-server database

Our third example is that of a hybrid relational database [17]. The database consists of clients and servers, with the distinction being that queries are submitted at clients and the data resides at servers. Queries can execute at either place. In fact, this is the main choice the application bundle exports to Harmony. We assume a single, always available server and one or more clients. The interface to Harmony is handled entirely by the clients. Each client that has queries to execute contacts Harmony with a choice bundle. The bundle consists of two options: *query-shipping*, in which queries are executed at the server, and *data-shipping*, where queries are executed at the client. Each option specifies resource usage on behalf of both the client and the remote server. Although there is no explicit link between clients, Harmony is able to combine server resource usage on behalf of multiple independent clients in order to predict total resource consumption by the server.

Figure 3 shows one possible bundle specification. The DBclient application specifies a bundle named “where,” with two options: QS (query-shipping), and DS (data-shipping). In either case, cycles and memory are consumed at both the client and the server, and bandwidth is consumed on a link between the two. The distinction is that “QS” consumes more resources at the server, and “DS” consumes more at the client. All other things being equal, the query-shipping approach is faster, but consumes more resources at the server. Each option specifies two node resources, and a network link between the two. All numeric arguments are total requirements for the life of the job. Both assume that the server is at “harmony.cs.umd.edu”; presumably the clients and servers can locate each other given a machine name. Additionally, the nodes are qualified by “seconds”, meaning the total expected seconds of computation on our reference machine, and “memory,” which specifies the minimum amount of memory needed.

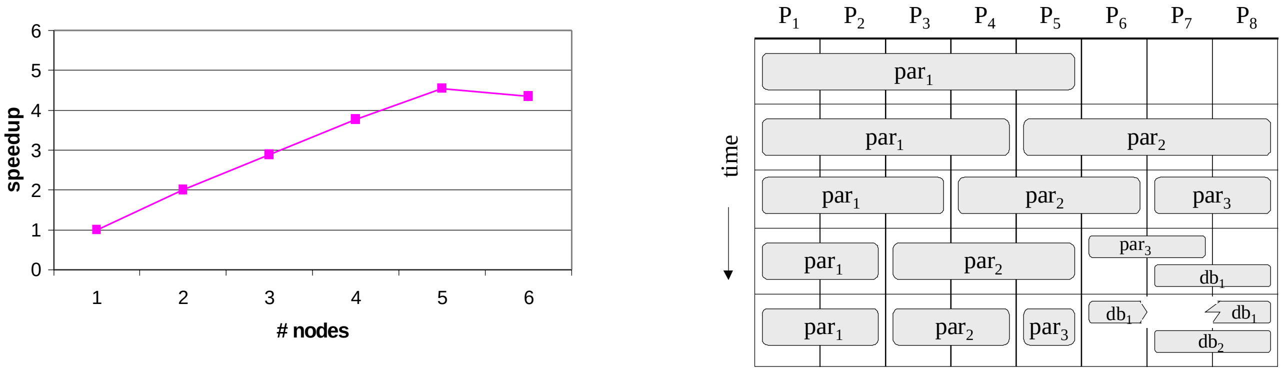
```
harmonyBundle Dbclient:1 where {
  {QS     {node server
                    {hostname harmony.cs.umd.edu}
                    {seconds                9}
                    {memory                 20}}
          {node client
                    {hostname               *}
                    {os                     linux}
                    {seconds                1}
                    {memory                 42"}}
          {link client server 2}}
  {DS     {node server
                    {hostname harmony.cs.umd.edu }
                    {seconds                1}
                    {memory                 20}}
          {node client
                    {hostname               *}
                    {os                     linux}
                    {memory                 >=17}
                    {seconds                9}}
          {link client server
                    {44 + (client.memory > 24 ? 24 :
                    client.memory) - 17}}
  }}
```

Figure 3: Client-Server Database

Both options specify the nodes equivalently. The names “server” and “client” are used within the option namespace to identify which node is being referred to. For example, the “link” option specifies the total communication requirements between “server” and “client”, without needing to know at application startup exactly which nodes are being instantiated to these names.

In addition to basic functionality, the example illustrates two relatively sophisticated aspects of Harmony’s resource management. First, resource usage is higher at the server with query-shipping than data-shipping. This allows the system to infer that server load grows more quickly with the number of clients with query-shipping than with data-shipping. At some number of clients, the server machine will become overloaded, resulting in data-shipping providing better overall performance. The specification does not require the same option to be chosen for all clients, so the system could use data-shipping for some clients and query-shipping for others.

Second, the memory tag of “>= 32” tells Harmony that 32 MB is the minimal amount of memory that the application requires, but that additional memory can be used profitably used as well. The specification for bandwidth in the data-shipping case is then parameterized as a function of “client.memory.” This allows the application to tell Harmony that the amount of required bandwidth is dependent on the amount of memory allocated on the client machine. Harmony can then decide to allocate additional memory resources at the client in order to reduce bandwidth requirements. This tradeoff is a good one if memory is available, because additional memory



(a) (b)

Figure 4: Online reconfiguration – The left side (a) shows the performance of a parallel application and (b) shows the eight-processor configurations chosen by Harmony as new jobs arrive. Note the configuration of five nodes (rather than six) in the first time frame, and the subsequent configurations that optimize for average efficiency by choosing equal partitions for multiple instances of the parallel application, rather some large and some small.

usage does not increase the application's response time, whereas additional network communication does.

4. Policies

A key piece of Harmony is the policies used by the automatic adaptation system to assign resources to applications. This section describes how Harmony matches application resource requirements to the available resources. We then describe how we compose the performance information from individual nodes into a global picture of resource utilization. Finally, we describe the overall objective function that Harmony optimizes. The current polices, although simple, allow us to gain experience with the system.

4.1 Matching Resource Needs

Resources are assigned to new applications under Harmony based on the requirements described in the corresponding RSL. When Harmony starts execution, we get an initial estimate of the capabilities of each node and links in the system. For nodes, this estimate includes information about the available memory, and the normalized computing capacity of the node. For links, we note the bandwidth and latency attributes. As nodes and links are matched, we decrease the available resources based on the application's RSL entries.

We start by finding nodes that meet the minimum resource requirements required by the application. When considering nodes, we also verify that the network links between nodes of the application meet the requirements specified in the RSL. Our current approach uses a simple first-fit allocation strategy. In the future, we plan to extend the matching to use more sophisticated policies that try to avoid fragmentation. However, for now our goal is to demonstrate the ability of our system to optimize application performance based on the options, so any initial match of resource requirements is acceptable.

4.2 Explicit (response time) models

Harmony's decisions are guided by an overarching objective function. Our objective function currently minimizes the average completion time of the jobs currently in the system. Hence, the system must be able to predict the lifetime of applications. Harmony has a very simple default performance model that combines resource usage with a simple contention model.

However, this simplistic model is inadequate to describe the performance of many parallel applications because of complex interactions between constituent processes. For example, we might use the critical path notion to take inter-process dependencies into account [11]. Other application models could model piece-wise linear curves. Figure 4 shows an example of Harmony's configuration choices in the presence of our client-server database and an application with variable parallelism. The parallel application's speedup curve is described in an application-specific performance model[1].

In the future we plan to investigate other objective functions. The requirement for an objective function is that it be a single variable that represents the overall behavior of the system we are trying to optimize (across multiple applications). It really is a measure of goodness for each application scaled into a common currency.

4.3 Setting Application Options

The ability to select among possible application options is an integral part of the Harmony system. In order to make this possible, we need to evaluate the likely performance of different options and select the one that maximizes our objective function. However, the space of possible option combinations in any moderately large system will be so large that we will not be able to evaluate all combinations. Instead, we will need a set of heuristics that select an application option to change and then evaluate the overall system objective function.

Currently, we optimize one bundle at a time when adding new applications to the system. Bundles are evaluated in the same lexical order as they were defined.

[1] This performance model matches a simple bag of tasks parallel application and a client-server database we have modified to support Harmony.

`harmony_startup(<unique id>, <use interrrupts>)`

A program registers with the Harmony server using this call.

`harmony_bundle_setup("<bundle definition>")`

An application informs Harmony of one of its bundles this way. The bundle definition looks like the examples given in Section 3.1.

`void *harmony_add_variable("variable name", <default value>, <variable type>)`

An application declares a variable that to communicate information between Harmony and the application. Harmony variables include bundle values, and resource information (such as the nodes that the application has been assigned to use). The return value is the pointer to the variable.

`harmony_wait_for_update()`

The application process blocks until the Harmony system updates its options and variables.

`harmony_end()`

The application is about to terminate and Harmony should re-evaluate the application's resources.

Figure 5: Harmony API Used by Application Programs.

This is a simple form of greedy optimization that will not necessarily produce a globally optimal value, but it is simple and easy to implement. After defining the initial options for a new application, we re-evaluate the options for existing applications. To minimize the search space, we simply iterate through the list of active applications and within each application through the list of options. For each option, we evaluate the objective function for the different values of the option. During application execution, we continue this process on a periodic basis to adapt the system due to changes out of Harmony's control (such as network traffic due to other applications).

5. Prototype

We have developed a prototype of the Harmony interface to show that applications can export options and respond to reconfiguration decisions made by system. The architecture of the prototype is shown in Figure 6. There are two major parts, a Harmony process and a client library linked into applications.

The Harmony process is a server that listens on a well-known port and waits for connections from application processes. Inside Harmony is the resource management and adaptation part of the system. When a Harmony-aware application starts, it connects to the Harmony server and supplies the bundles that it supports.

A Harmony-aware application must share information with the Harmony process. The interface is summarized in Figure 5. First, the application calls functions to initialize the Harmony runtime library, and define its option bundles. Second, the application uses special Harmony variables to make run-time decisions about how the computation should be performed. For example, if an application exports an option to change its buffer size, it needs to periodically read the Harmony variable that indicates the current buffer size (as determined by Harmony controller), and then update its own state to this size. Applications access the "Harmony" variables by using the pointer to a Harmony variable returned by the `harmony_add_variable()` function.

New values for Harmony variables are buffered in the until a `flushPendingVars()`call is made. This call sends all pending changes to the application processes. Inside the application, a I/O event handler function is called when the Harmony process sends variable updates. The updates are then applied to the Harmony variables. The application process must periodically check the values of these variables and take appropriate action.

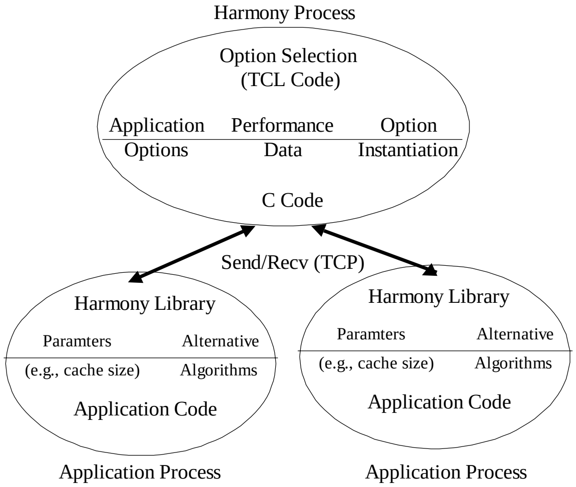


Figure 6: Architecture of Harmony Prototype.

Our system uses a polling interface to detect changes in variables at the application. Many long-running applications have a natural phase where it is both easier and more efficient to change their behavior rather than requiring them to react immediately to Harmony requests. For example, database applications usually need to complete the current query before reconfiguring the system from a query shipping to a data-shipping configuration. Likewise, scientific applications generally have a time-step or other major loop that represents a natural point to re-configure the application.

The Harmony process is an event driven system that waits for application and performance events. When an event happens, it triggers the automatic application adaptation system, and each of the option bundles for each application gets re-evaluated to see it should be changed (see Section 4 for a complete description of the way the

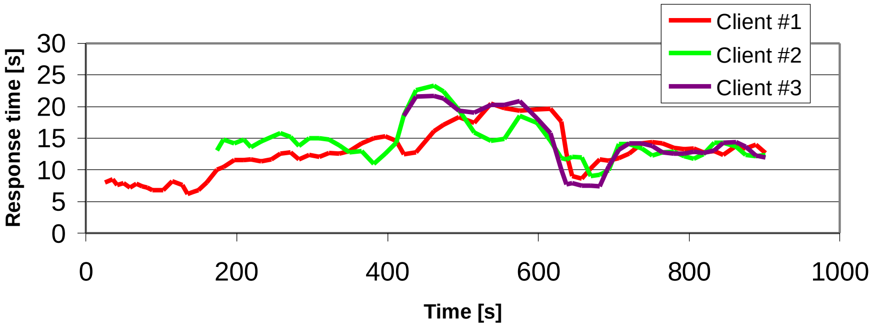


Figure 7: Client-server database application – Harmony chooses query-shipping with one or two clients, but switches all clients to data-shipping when the third client starts.

evaluation is done). When option bundles are changed, the appropriate variables are updated in each application.

6. An Example Application

To explore the ability of the Harmony server to adapt an application, we modified a hybrid client-server database to allow Harmony to reconfigure where queries are processed: on client nodes or on server nodes. The database system used was Tornadito [17], a relational database engine built on top of the SHORE (Scalable Heterogeneous Object REpository) storage manager [2, 22]. All experiments were run on nodes of an IBM SP-2, and used the 320Mbps high performance switch to communicate between clients and the server. Each client ran the same workload, a set of similar, but randomly perturbed join queries over two instances of the Wisconsin benchmark relations [9], each of which contains 100,000 208-byte tuples. In each query, tuples from both relations are selected on an indexed attribute (10% selectivity) and then joined on a unique attribute. While this is a fairly simple model of database activity, such query sets often arise in large databases that have multiple end users (bank branches, ATMs), and in query refinement.

The Harmony interface exported by this program is the set of option bundles shown in Figure 3. For our initial experiments, the controller was configured with a simple rule for changing configurations based on the number of active clients. We then ran the system and added clients about every three minutes. The results of this experiment on shown in Figure 7. In this graph, the x-axis shows time, and the y-axis shows the mean response time of the benchmark query. Each curve represents the response time of one of the three clients. During the first 200 seconds, there is only one client active and the system is processing the queries on the server. During the next 200 seconds, two clients are active, and the response time for both clients is approximately double the initial response time with one active client.

At 400 seconds, the third client starts, and the response time of all clients increases to approximately 20 seconds. During this interval one of the clients has a response time that is noticeably better than the other two (client #1 for the first 100 seconds, and then client #2). This is likely due to cooperative caching effects on the server since all clients are accessing the same relations.

The addition of the third client also eventually triggers the Harmony system to send a re-configuration event to the clients to have them start processing the queries locally rather than on the server. This results in the response time of all three clients being reduced, and in fact the performance is approximately the same as when two clients were executing their queries on the server. This demonstration shows that by adapting an application to its environment, we can improve its performance.

7. Related work

GLOBUS [6] and Legion [12] are both large projects that are trying to address many of the different requirements to build a meta-computing environment. By contrast, our work is concentrating on the specific problem of developing interfaces and policies to allow applications to react to their computing environment

The Odyssey [15] and EMOP [5] projects are also focused on online adaptation. Odyssey gives resources, such as network bandwidth, to applications on a best-effort basis. Applications can register system callbacks to notify them when resource allocations stray outside of minimum and maximum thresholds. EMOP provides mechanisms and services (including object migration facilities) that allow applications to define their own load-balancing and communication services.

The Condor system[13] tries to allocate computing resources to nodes that are otherwise idle. Condor uses Classified Ads[20] to match resource suppliers and consumers. Their approach provides a flexible way to describe the attributes of a node that is required such as memory and processing speed. However, it is designed to handle the placement of sequential applications. The only runtime feedback an application is to leave a node when the workstation owner returns.

One system that does try to adapt applications to available resources is AppLes [1]. AppLes allows applications to be informed of the variations in resources and presented with candidate lists of resources to use. Each application is then able to develop its own customized resource allocation policy. Harmony differs from AppLes in that we try to optimize resource allocation between applications, whereas AppLes lets each application 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.

Computational Steering [8, 10, 18] provides a way for users to alter the behavior of an application under execution. Harmony's approach is similar in that applications provide hooks to allow their execution to be changed. Many computational steering systems are designed to allow the application semantics to be altered, 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 to this is Autopilot [21] which allows applications to be adapted in an automated way. Harmony differs from Autopilot in that it tries to coordinate the use of resources by multiple applications.

8. Conclusions

We have presented an overview of the application adaptation interface that we are building as part of the Harmony project. We described the way application can export different candidate options to the system. We also reported on the development of an early prototype of the system and a client-server database application that have been extended for use with Harmony. Finally, we showed that our system was able to change the configurations of the applications during execution to improve not only the performance of a single client but also the overall throughput of the system.

9. References

1. F. Berman and R. Wolski, "Scheduling from the Perspective of the Application," *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*.
2. M. Carey, *et al.*, "Shoring Up Persistent Applications," *ACM SIGMOD*. May 24 - 27, 1994, Minneapolis, MN.
3. D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. v. Eicken, "LogP: Towards a Realistic Model of Parallel Computation," *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 262-273.
4. K. Czajkowski, I. Foster, C. Kesselman, N. Karonis, S. Martin, W. Smith, and S. Tuecke, "A Resource Management Architecture for Metacomputing Systems.," *IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*. 1998.
5. S. Diwan and D. Gannon, "Adaptive Utilization of Communication and Computational Resources in High Performance Distribution Systems: The EMOP Approach," *The 7th International Symposium on High Performance Distributed Computing*.
6. I. Foster, N. Karonis, C. Kesselman, G. Koenig, and S. Tuecke, "A Secure Communications Infrastructure for High-Performance Distributed Computing," *Proc. 6th IEEE Symp. on High-Performance Distributed Computing*.
7. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*. 1994, Cambridge, Mass: The MIT Press.
8. A. G. Geist, 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*, **11**(3), 1997, pp. 224-35.
9. J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*. Second ed. 1993, San Mateo, CA: Morgan Kaufmann.
10. W. Gu, et. al, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," *Frontiers '95*. Feb 6-9, 1995, McLean, VA, pp. 422-429.
11. J. K. Hollingsworth, "Critical Path Profiling of Message Passing and Shared-memory Programs," *IEEE Transactions on Parallel and Distributed Computing*, **9**(10), 1998.
12. M. J. Lewis and A. Grimshaw, "The Core Legion Object Model," *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*.
13. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.
14. B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok, "A Resource Query Interface for Network-Aware Applications," *The 7th International Symposium on High-Performance Distributed Computing*.
15. B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker, "Agile Application-Aware Adaptation for Mobility," *Proceedings of the 16th ACM Symposium on Operating Systems Principles*.
16. J. K. Osterhout, "Tcl: An Embeddable Command Language," *USENIX Winter Conf*. Jan 1990, pp. 133-146.
17. N. Padua-Perez, *Performance Analysis of Relational Operator Execution in N-Client 1-Server DBMS Architectures*,Masters, Computer Science University of Maryland,1997.
18. S. G. Parker and C. R. Johnson, "SCIRun: a scientific programming environment for computational steering," *Supercomputing*. Nov. 1995, San Diego, vol.II, pp. 1419-39.
19. R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," *The 7th International Symposium on High-Performance Distributed Computing*.
20. R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," *Seventh IEEE International Symposium on High Performance Distributed Computing*. July 1998, Chicago.
21. R. L. Ribler, J. S. Vetter, H. Simitci, and D. A. Reed, "Autopilot: Adaptive Control of Distributed Applications," *High Performance Distributed Computing*, Chicago, IL, pp. 172-9.
22. M. Zaharioudakis and M. Carey, "Highly Concurrent Cache Consistency for Indices in Client-Server Database Systems," *ACM SIGMOD*. May 13 - 15, 1997, Tucson, AZ, pp. 50 - 61.