To appear in *Advances in Computers*, edited by Marvin V. Zelkowitz, Academic Press.

Resource-Aware Meta-Computing

Jeffrey K. Hollingsworth
hollings@cs.umd.edu

Peter J. Keleher
keleher@cs.umd.edu

Kyung D. Ryu
kdryu@cs.umd.edu

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

Meta-computing is an increasingly popular and useful method of obtaining resources to solve large computational problems. However, meta-computer environments pose a number of unique challenges, many of which have yet to be addressed effectively. Among these are dynamicism in both applications and environments, and heterogeneity at several different levels.

This chapter discusses current approaches to these problems, and uses the in the Active Harmony system as a running example. Harmony supports an interface that allows applications to export tuning alternatives to the higher-level system. By exposing different parameters that can be changed at runtime, applications can be automatically adapted to changes in their execution environment caused by other programs, the addition or deletion of nodes, or changes in the availability of resources like communication links. Applications expose not only options, but also expected resource utilization with each option and the effect that the option will have on the application's performance. We discuss how this flexibility can be used to tune the overall performance of a collection of applications in a system.

Keywords: Resource-aware, programming models, scheduling.

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 needed to efficiently support this type of global-scale computation is not yet available.

Private workstations connected by a network have long been recognized for use with computation intensive applications. Since a large fraction of the time workstations are unused, idle cycles can be harnessed to run scientific computations or simulation programs as a single process or a parallel job. The usefulness of this approach depends on 1) how much time the machines are available and 2) how well those available resources can be harnessed.

Mutka and Livny [1] found that on the average, 75% of the time machines were idle. Similarly, Krueger and Chawla [2] demonstrated that for 199 diskless Sun Workstations on average the nodes were idle 91% of the time and that only 50% of those idle cycles were made available for background jobs. The other half of the idle cycles could not be harnessed since they belonged either to intervals when the workstation owners were using their machines or during a waiting period to ensure that users were away. More recently, Acharya et al [3] also observed that 60% to 80% of the workstations in a pool are available by analyzing machine usage traces from three academic institutions.

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. The canonical way to run applications in current meta-computing environments is to pass the application's name, parameters, and number of required nodes to the system, and to hope for the best. The execution environment (hardware and system software) is expected to run the program efficiently with little or no information from the application about the application's needs or expectations. This model of application and system interaction is simple, and allows many applications to run well.

However, this model does not accommodate the full generality of application-system interactions required by current applications. In particular, an application may wish to alter its resource requests based on knowledge of available resources. For example, a multi-media application might alter its resolution or frame rate based on available bandwidth. Second, computational resources are not always static and fixed for the life of an application. For example, an application might be running on a system that batch schedules jobs onto idle workstations. To address the needs for these types of applications, the interface between applications and the execution environment needs to change. In this chapter, we present our approach to enhancing the application-system interface.

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 are 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. Alternatively, this information could 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 to permit run-time reconfiguration of 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.

We target 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 target these types of 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 and application reconfiguration since these operations can be amortized across the relatively long life of the application.

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

1) *Can we build an API that is expressive enough to define real-world alternatives?*
2) *Can a system use this information to improve the behavior of applications during execution?*
3) *How can we make accurate predictions of parallel and distributed applications?*

We use the *Active Harmony* system as a running example of a resource manager that can make the tradeoffs discussed above. However, our intent is to foster a more general dialogue about the role of sophisticated resource managers, heterogeneous clusters, and tradeoffs between autonomy and efficiency. Other projects such as AppLeS, Condor, Globus, and Legion provide similar functionality with somewhat different abstractions. Section 5 describes these systems in more detail.

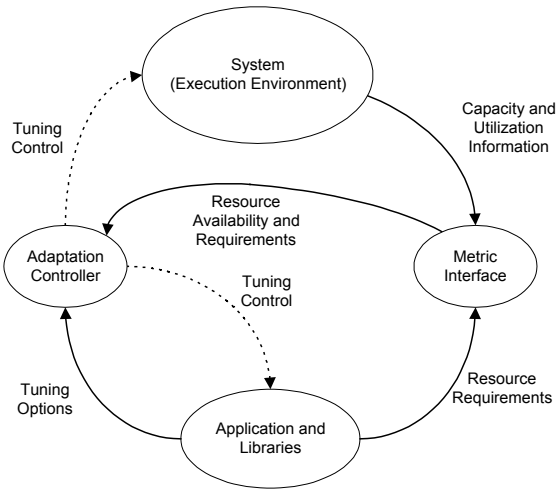


Figure 1: Major Components of Active Harmony

The remainder of this chapter is organized as follows. Section 2 presents an overview of Harmony and describes how it facilitates cooperation between applications and system software. Section 3 describes several sub-systems that extend Harmony by performing sophisticated application adaptation. Section 4 describes how information about the execution environment can be gathered for use at runtime. Section 5 describes additional work related to dynamic resource management. Finally, Section 6 summarizes the issues raised in this chapter.

2. Harmony structure

Harmony applications are linked with a Harmony communication library. Each library communicates with the centralized Harmony scheduler process, which runs on one node in the system. The overall system is shown in Figure 1, and the major components are described below.

The *adaptation controller* is the heart of the scheduler. The controller gathers relevant information about both the applications and the environment, projects 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.

The scheduler uses a *metric interface* to provide 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.

The library stubs communicate with application through a *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.

2.1 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 [4] and the Globus RSI [5] 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.

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 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 [6].

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.

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

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 making 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 the system. 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.

2.1.1 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 [7]. Applications specify requirements by sending TCL scripts to Harmony, which executes them and sends back resource allocation descriptions. An example of one such script is shown in Figure 2.

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's 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.

2.1.2 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 an option bundle are the resource requirements for that option, currently just nodes and links. In addition, nodes contain sub-resources such as memory, CPU, I/O etc. Links currently contain only bandwidth estimates. An example of a 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.
```

2.1.3 Simple parallel application

We next 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 "variable" tag specifies that this node definition should be used to match four distinct nodes, all meeting the same requirements. Second, we use the "communication" 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.

2.1.4 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

```
harmonyBundle Simple - {
 {-          {node "worker"
                    {hostname    "*"}
                    {os          "linux"}
                    {seconds     "300"}
                    {memory      32}}
             {variable  worker  "node"  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

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.

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.

Other ways of modeling costs could also be useful. For example, a somewhat more accurate model of communication costs is CPU occupancy on either end (for protocol processing, copying), plus wire time [8]. If this occupancy is significant, cycles on all worker processes would need to be parameterized based on the amount of communication.

2.1.5 Client-server database

Our third example is that of Tornadito, a hybrid relational database [9]. 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". 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 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 usage does not increase the application's response time, whereas additional network communication does.

2.2 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 re-

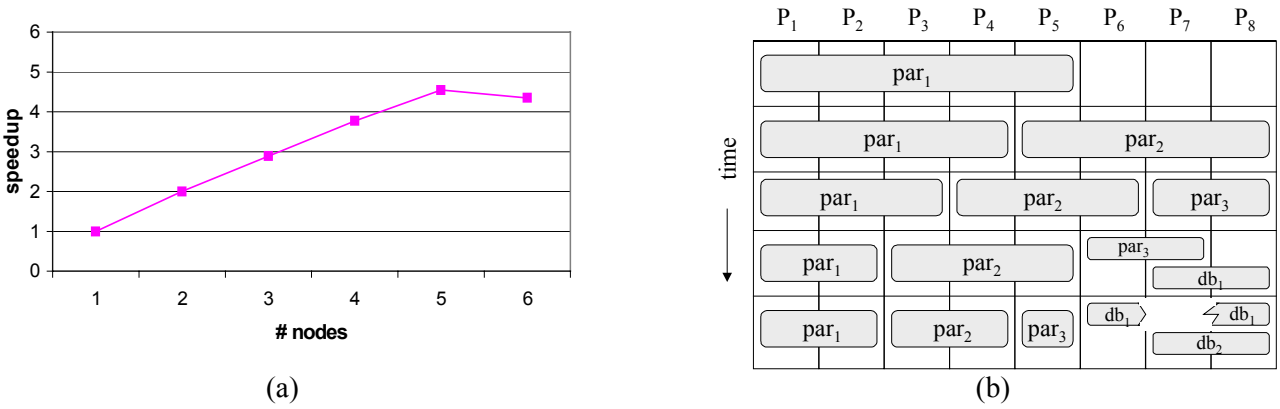


Figure 4: Online reconfiguration – The left side (a) shows the performance of a parallel application and (b) shows the eight-processor configurations chosen by Harmony as new jobs arrive. Note the configuration of five nodes (rather than six) in the first time frame, and the subsequent configurations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

sources. 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.

2.2.1 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.

2.2.2 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 [10]. Other application models could model piece-wise lin-

ear 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.

2.2.3 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. 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).

2.2.4 Search-space heuristics

To be effective, the system needs to find good configurations for each incoming application. Application configuration consists of two parts. First, "Harmonized" applications export mutually exclusive algorithmic alternatives for the system to choose among. Second, a given application and tuning alternative can usually be mapped to multiple configurations of nodes in the system. Hence, the search space of possible configurations is clearly too large for exhaustive search, especially since "looking" at each individual configuration alternative requires evaluating the performance model against that configuration. Thus, heuristics are needed to constrain the search space of configurations for a given application and among applications.

Heuristics are evaluated based on cost and efficiency. A heuristic that performs anything like an exhaustive search clearly has too high of a runtime cost to be useful. On the other hand, a cheap heuristic that leads to poor processor efficiencies and response times is of little use. As a starting point, simple variations on first-fit and best-fit heuristics show promising behavior whether or not currently running applications can be reconfig-

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

ured. Another part of this strategy is to try satisfy a resource request with "local" machines before considering all connected remote machines. While sophisticated approaches such as hill-climbing or simulated annealing have a number of advantages, our expectation is that simple heuristics will perform nearly as well and at a much lower cost than more complicated approaches.

2.2.5 Performance prediction subsystem

To effectively compare candidate configurations, we need accurate, lightweight models for the expected behavior of applications in any possible configuration. Accurate models allow the system to make good decisions about scheduling jobs on constrained resources. Consider a system with eight nodes and three incoming jobs: a two-node job, an eight-node job, followed by another two-node job. A simple FCFS system will delay each job until the others have finished. However, running the two smaller jobs at the same time might be a better approach. The reason that we can not say this with any assurance is that systems have historically not had a priori knowledge of expected running times. If the second two-node job runs much longer than the first, jumping it in front of the larger job will delay the larger job unfairly. *Backfilling* [11] schedulers use knowledge of running times to determine whether or not re-ordering will delay jobs that arrived first. However, such systems depend on accurate knowledge of the running times. Underestimating the times still delays the larger application unfairly. Overestimating times reduces the opportunity for re-ordering.

In addition to simple job duration, we want to be able to handle the more complex resource needs of supporting multiple applications in a system. For example, we need to manage the amount of memory and networking bandwidth used by applications. Furthermore, we want to be able to handle a reasonably large set of performance metrics, in addition to the more "traditional" ones, such as response time and throughput (this will become clearer as we discuss the applications below).

Even if accurate information on running time for a single configuration *is* available, a global system needs to be able to evaluate running times in a variety of configurations, on possibly heterogeneous hosts, and possibly in the presence of contention for a variety of resources. End-users clearly can not provide all of this information.

Thus, analytical models will allow us to develop expressive and accurate predictions of application behavior. We are currently adapting previous work on using simplified analytical models to represent complex systems (e.g., as in [12-15]).

Model Generation

We are using simple analytical models to predict performance at runtime. Efficient resource management depends on good resource usage predictions. Furthermore, better resource management decisions can be made dynamically, at runtime, when more information is available about the currently running applications and their resource demands. Such decisions must be made relatively quickly, and simple analytical models are well suited for these types of problems.

The constructed analytical models must have appropriate knobs that can be turned in order to make a resource allocation decision. The system can then automatically derive models from information provided by the applications. To construct the models automatically, knowledge of resources and the workload to be executed on these resources is required. Clearly, the adaptation control has information about available system resources; thus, what is needed is a proper workload characterization. In order to simplify matters for the application designers, the system needs to perform workload characterization from a variety of information sources. Specifically, the information given in the RSL description can be used as a source of resource demands of each application. More sophisticated models will be constructed based on more detailed descriptions of application, which would require the use of an appropriate specification language. One possible way to represent the models is via an existing modeling language such as UML.

UML (the Unified Modeling Language) [16] provides a standard means of describing software oriented designs and promises to be in wide use in the software industry. UML contains several types of diagrams, which allow different properties of a system design to be expressed. Such diagrams (and the associated information) are useful in the generation of performance models. For instance, Use Case diagrams are useful in characterizing the system workload; implementation diagrams are useful in characterizing the mapping of software components onto hardware resources and in defining contention for these resources.

To facilitate automatic generation of analytical models, we are building a library of model components, for each of the resources available in our system, which can be used in automatic construction of appropriate analytical models at runtime.

Multi-resolution models

More accurate analytical models generally produce better predictions, but also tend to increase the time needed to make resource management decisions. Thus, there is a tradeoff between the "goodness" of the resource management decision and the amount of time and resources required to make it. The optimum accuracy of the model, with respect to its cost and resulting benefits, will to a large extent depend on the application(s) being modeled. Thus, for better results, such decisions should also be made at runtime. To facilitate this, we plan to construct "multi-resolution" models and choose the appropriate resolution, based on predicted benefits and costs, at runtime.

For instance, consider a system that manages a shared pool of resources where more resources are allocated to an application as the need for them arises (i.e., as the workload of that application crosses a predefined set of thresholds). In order to make such a system stable, one should include a hysteresis, i.e., the set of thresholds for adding resources should be distinct from the set of thresholds for removing resources. In this case, an analytical model that accurately represents both the hysteresis behavior and the non-instantaneous resource addition behavior can be quite costly to solve (details can be found in Golubchik and Liu [12]). However, a model that "ignores" this complex behavior is quite simple (actually, a simple birth-death stochastic process). The more accurate model will tend to give better performance predictions, but would also require quite a bit more

time to solve. Thus, depending on the amount of time and resources available to the adaptation controller, it would choose one or the other model in making a decision about allocating more resources to an application.

System state

For many applications, it will be important to model the behavior of multiple invocations, rather than just one. For example, consider a client-server database. There may be little incentive to run a single query on a client because we would need to migrate too much data. However, if multiple queries are considered, we might invest the time to migrate data to a client to allow a workload of queries to run faster.

Therefore the system must maintain state information (e.g., which data is being cached by a database client) and the performance models must incorporate that state can be passed from one invocation to the next (or between applications). We plan to incorporate sate information into the workload characterization process by mapping workflow onto system resources (as described above). Options to represent state in the system include simple finite state automata.

Runtime parameter measurement

The accuracy of the performance prediction models can be further improved by increasing the accuracy of the parameters used by these models, which to a large extent reflect the current state of the system, in terms resource and workload characteristics. To aid in increasing the accuracy of this information, we use runtime performance measurements, such as [10, 17].

2.3 Mechanisms

Most of the mechanisms needed by meta-computer schedulers are identical to those needed by local schedulers. One difference is that meta-computing on non-dedicated resources, such as idle machines, must be prevented from impacting the owners of those machines. This section briefly describes a set of techniques that allow remote jobs to co-exist with local jobs without the performance of the local jobs being affected. We describe how to realize fined-grained cycle stealing (the *linger-longer* approach), and the requirements that this approach imposes on local schedulers. The key feature of fine-grained cycle stealing is to exploit brief periods of idle processor cycles while users are either thinking or waiting for I/O events. We refer to the processes run by the workstation owner as host processes, and those associated with fine-grained cycle stealing as guest processes.

In order to make fine-grained cycle-stealing work, we must limit the resources used by guest processes and ensure that host processes have priority over them. Guest processes must have close to zero impact on host processes in order for the system to be palatable to users. To achieve that goal requires a scheduling policy that gives absolute priority to host processes over guest processes, even to the point of starving guest processes. This also implies the need to manage the virtual memory via a priority scheme. The basic idea is to tag all pages as either guest or host pages, and to give priority on page replacement to the host pages. The complete mechanism is presented in Section 2.3.3.

Previous systems automatically migrate guest processes from non-idle machines in order to ensure that guest processes do not interfere with host processes. The key idea of our fine-grained cycle stealing approach is that migration of a guest process off of a node is *optional*. Guest processes can often co-exist with host processes without significantly impacting the performance of the latter, or starving the former.

A key question in evaluating the overhead of priority-based preemption is the time required to switch from the guest process to the host process. There are three significant sources of delay in saving and restoring the context of a process: 1) the time required to save registers, 2) the time required (via caches misses) to reload the process's cache state, and 3) the time to reload the working set of virtual pages into physical page frames. We defer discussion of the latter overhead until Section 2.3.3. On current microprocessors, the time to restore cache state dominates the register restore time. In a previous paper [18], we showed that if the *effective context-switch time* is 100 microseconds or less, the overhead of this extra context-switch is less than 2%. With host CPU loads of less than 25%, host process slowdown remains under 5% even for effective context switch times of up to 500 micro-seconds.

In addition, our simulations of sequential processes showed that a linger-based policy would improve average process completion time by 47% compared with previous approaches. Based on job throughput, the Linger-Longer policy provides a 50% improvement over previous policies. Likewise our Linger-Forever policy (i.e. disabling optional migrations) permits a 60% improvement in throughput. For all workloads considered in the study, the delay, measured as the average increase in completion time of a CPU request, for host (local) processes was less than 0.5%.

2.3.1 Linux kernel extensions

This section introduces the modifications to the local Linux scheduler necessary to support the Linger-Longer scheduling policy. These extensions are designed to ensure that guest processes can not impede the performance of host processes. We first describe the general nature of our kernel modifications, and then describe how we modified the scheduler and virtual memory system of Linux to meet our needs.

One possible concern with our approach is the need for kernel modifications. In general, it is much harder to gain acceptance for software that requires kernel modifications. However, for the type of system we are building, such modifications are both necessary and reasonable. First, guest processes must be able to stay running, yet impose only an unnoticeable impact on foreground local processes. There is no practical way to achieve this without kernel modifications. Additionally, we feel that kernel modifications are a reasonable burden for two reasons. First, we are using the Linux operating system as an initial implementation platform, and many software packages for Linux already require kernel patches to work. Second, the relatively modest kernel changes required could be implemented on stock kernels using the kernInst technology [19]. KernInst allows fairly complex customizations of a UNIX kernel at runtime via dynamic binary re-writing. All of the changes we have made could be implemented using this technique.

Current UNIX systems support CPU priority via a per-process parameter called the *nice* value. Via nice, different priorities can be assigned to different processes. These priority levels are intended to reflect the relative importance of different tasks, but they do not necessarily implement a strict priority scheme that always schedules the highest priority process. The nice value of a process is just a single component that is used to compute the dynamic priority during execution. As a result, sometimes a lower *static priority* process gets scheduled over higher static priority processes to prevent starvation, and to ensure progress of the lower priority processes. However, we need a stricter concept of priority in CPU scheduling between our two classes of processes. Guest processes should not be scheduled (and can even starve) when any host process is ready no matter what its run time priority is. Meanwhile, the scheduling between the processes in the same class should be maintained as it is under current scheduling implementation.

While many UNIX kernels provide strict priorities in order to support real-time deadlines, these real-time priorities are *higher* than traditional UNIX processes. For Linger-Longer, we require just the opposite, a lower priority than normal.

Current general-purpose UNIX systems provide no support for prioritizing access to other resources such as memory, communication and I/O. Priorities are, to some degree, implied by the corresponding CPU scheduling priorities. For example, physical pages used by a lower-priority process will often be lost to higher-priority processes. Traditional pages replacement policies, such as Least Recently Used (LRU), are more likely to page out the lower-priority process's pages, because it runs less frequently. However, this might not be true with a higher-priority process that is not computationally intensive, and a lower priority process that is. We therefore need an additional mechanism to control the memory allocation between local and guest processes. Like CPU scheduling, this modification should not affect the memory allocation (or page replacement) between processes in the same class.

We chose Linux as our target operating system for several reasons. First, it is one of the most widely used UNIX operating systems. Second, the source code is open and widely available. Since many active Linux users build their own customized kernels, our mechanisms could easily be patched into existing installations by end users. This is important because most PCs are deployed on people's desks, and cycle-stealing approaches are probably more applicable to desktop environments than to server environments.

2.3.2 Starvation-level CPU scheduling

The Linux scheduler chooses a process to run by selecting the ready process with the highest runtime priority, where the runtime priority can be thought of as the number of 10ms time slices held by the process. The runtime priority is initialized from a static priority derived from the *nice* level of the process. Static priorities range from -19 to +19, with +19 being the highest[2]. New processes are given 20+*p* slices, where *p* is the static priority level. The process chosen to run has its store of slices decremented by one. Hence, all runnable processes tend to decrease in priority until no runnable processes have any remaining slices. At this point, all processes are reset to

their initial runtime priorities. Blocked processes receive an additional credit of half of their remaining slices. For example, a blocked process having 10 time slices left will have 20 slices from an initial priority of zero, plus five slices as a credit from the previous round. This feature is designed to ensure that compute-bound processes do not receive undue processor priority compared to I/O bound processes.

This scheduling policy implies that processes with the lowest priority (nice -19) will be assigned a single slice during each round, while normal processes consume 20 slices. When running two CPU-bound processes, where one has normal priority and the other is *niced* to the minimum priority, -19, the latter will still be scheduled 5% of the time. While this degree of processor contention might or might not be visible to a user, running the process could still cause contention for other resources, such as memory.

We implemented a new *guest priority* in order to prevent guest processes from running when runnable host processes are present. The change essentially establishes guest processes as a different class, such that guest processes are not chosen if any runnable host processes exist. This is true even if the host processes have lower runtime priorities than the guest process.

Second, we verified that the scheduler reschedules processes any time a host process unblocks while a guest process is running. This is the default behavior on Linux, but not on many BSD derived operating systems. One potential problem of our strict priority policy is that it could cause priority inversion. Priority inversion occurs when a higher priority process is not able to run due to a lower priority process holding a shared resource. This is not possible in our application domain because guest and host processes do not share locks, or any other non-revocable resources.

2.3.3 Prioritized page replacement

Another way in which guest processes could adversely affect host processes is by tying up physical memory. Having pages resident in memory can be as important to a process's performance as getting time quanta on processors. Our approach to prioritizing access to physical memory tries to ensure that the presence of a guest process on a node will not increase the page fault rate of the host processes.

Unfortunately, memory is more difficult to deal with than the CPU. The cost of reclaiming the processor from a running process in order to run a new process consists only of saving processor state and restoring cache state. The cost of reclaiming page frames from a running process is negligible for clean pages, but quite large for modified pages because they need to be flushed to disk before being reclaimed. The simple solution to this problem is to permanently reserve physical memory for the host processes. The drawback is that many guest processes are quite large. Simulations and graphics rendering applications can often fill all available memory. Hence, not allowing guest processes to use the majority of physical memory would prevent a large class of applications from taking advantage of idle cycles.

We therefore decided not to impose any hard restrictions on the number of physical pages that can be used by a guest process. Instead, we implemented a policy that establishes low and high thresholds for the number of

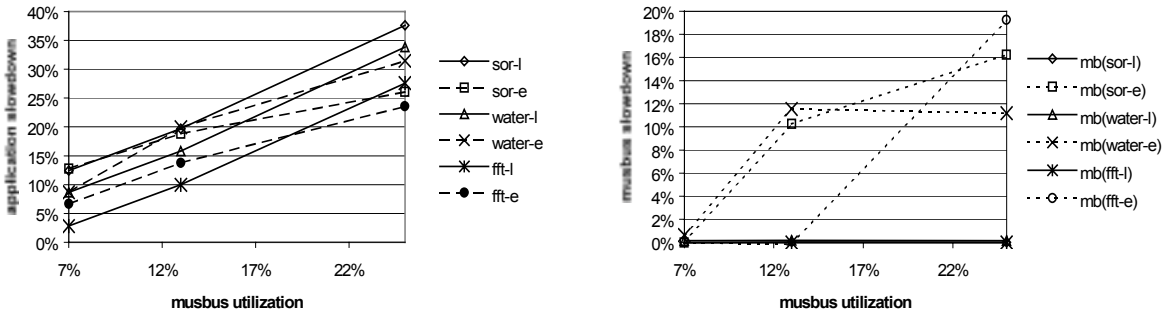
2 Nice priorities inside the kernel have the opposite sign of the nice values seen by user processes.

physical pages used by guest processes. Essentially, the page replacement policy prefers to evict a page from a host process if the total number of physical pages held by the guest process is less than the low threshold. The replacement policy defaults to the standard clock-based pseudo-LRU policy up until the upper threshold. Above the high threshold, the policy prefers to evict a guest page. The effect of this policy is to encourage guest processes to steal pages from host processes until the lower threshold is reached, to encourage host processes to steal from guest processes above the high threshold, and to allow them to compete evenly in the region between the two thresholds. However, the host priority will lead to the number of pages held by the guest processes being closer to the lower threshold, because the host processes will run more frequently.

In more detail, the default Linux replacement policy is an LRU-like policy based on the "clock" algorithm used in BSD UNIX. The Linux algorithm uses a one-bit flag and an *age* counter for each page. Each access to a page sets its flag. Periodically, the virtual memory system scans the list of pages and records which ones have the use bit set, clears the bit, and increments the age by three for the accessed pages. Pages that are not touched during the period of a single sweep have their age decremented by one. Only pages whose age value is less than a system-wide constant are candidates for replacement.

We modified the Linux kernel to support this prioritized page replacement. Two new global kernel variables were added for the memory thresholds, and are configurable at run-time via system calls. The kernel keeps track of resident memory size for guest processes and host processes. Periodically, the virtual memory system triggers the page-out mechanism. When it scans in-memory pages for replacement, it checks the resident memory size of guest processes against the memory thresholds. If they are below the lower thresholds, the host processes' pages are scanned first for page-out. Resident sizes of guest processes larger than the upper threshold cause the guest processes' pages to be scanned first. Between the two thresholds, older pages are paged out first regardless of which process is using them.

Correct selection of the two parameters is critical to meeting the goal of exploiting fine-grained idle intervals without significantly impacting the performance of host processes. Too high of value for the low threshold will cause undue delay for host processes, and too low of value will cause the guest process to constantly thrash. However, if minimum intrusiveness by the guest process is paramount, the low memory threshold can be set to zero to guarantee the use of the entire physical memory by foreground process.



(a) slowdown of parallel application (guest process)

(b) slowdown of host Musbus process

Figure 5: Impact of running one process of four-process CVM applications as a guest process.

In a previous paper [20] we evaluated Linger-Longer's effect on parallel applications on our test cluster. We used the Musbus interactive UNIX benchmark suite [21] to simulate the behavior of actual interactive users. Musbus simulates an interactive user conducting a series of compile-edit cycles. The benchmark creates processes to simulate both interactive editing (including appropriate pauses between keystrokes), UNIX command line utilities, and compiler invocations. We varied the size of the program being edited and compiled by the "user" in order to change the mean CPU utilization of the simulated local user. In all cases, the file being manipulated was at least as large as the original file supplied with the benchmark.

The guest applications are Water and FFT from the Splash-2 benchmark suite [22], and SOR, a simple red-black successive over-relaxation application [23]. Water is a molecular dynamics code, while FFT implements a three-dimensional Fast Fourier transform. All three applications were run on top of CVM [24], Harmony's user-level Distributed Shared-Memory (DSM) layer. DSM's are software systems that provide the abstraction of shared memory to threads of a parallel application running on networks of workstations. These three applications are intended to be representative of three common classes of distributed applications. Water has relatively fine-grained communication and synchronization, FFT is quite communication-intensive, while SOR is mostly compute-bound.

In the first set of experiments, we ran one process of a four-process CVM application as a guest process on each of four nodes. We varied the mean CPU utilization of the host processes from 7% to 25% by changing the size of the program being compiled during the compilation phase of the benchmark. The results of these tests are shown in Figure 5. The left graph shows the slowdown experienced by the parallel applications. The solid lines show the slowdown using our Linger-Longer policy, and the dashed lines show the slowdown when the guest processes are run with the default (i.e., equal priority). As expected, running the guest processes at starvation level priority generally slows them down more than if they were run at equal priority with the host processes. However, when the Musbus utilization is less than 15% the slowdown for all applications is lower with lingering than with the default priority. For comparison, running sor, water, and fft on three nodes instead of four slows them down by 26%, 25%, and 30%, respectively. Thus for the most common levels of CPU utiliza-

tion, running on one non-idle node and three idle would improve the application's performance compared to running on just three idle nodes. Our previous study [18] showed that node utilization of less than 10% occurs over 75% of the time even when users are actively using their workstations.

The right side of Figure 5 shows the slowdown experienced by the host Musbus processes. Again, we show the behavior when the guest processes are run using our Linger-Longer policy and the default equal priority. When the guest processes were run with moderate CPU utilization (i.e., over 10%), all three guest processes started to introduce a measurable delay in the host processes when equal priority was used. For Water and SOR, the delay exceeds 10% when the Musbus utilization reaches 13%. At the highest level of Musbus CPU utilization, the delay using the default priority exceeds 10% for all three applications and 15% for two of the three applications. However, for all three parallel guest applications, the delay seen when running with Linger-Longer was not measurable. This experiment demonstrates that our new CPU priority and memory page replacement policy can limit the local workload slowdown when using fine-grained idle cycles.

2.4 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 7. 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.

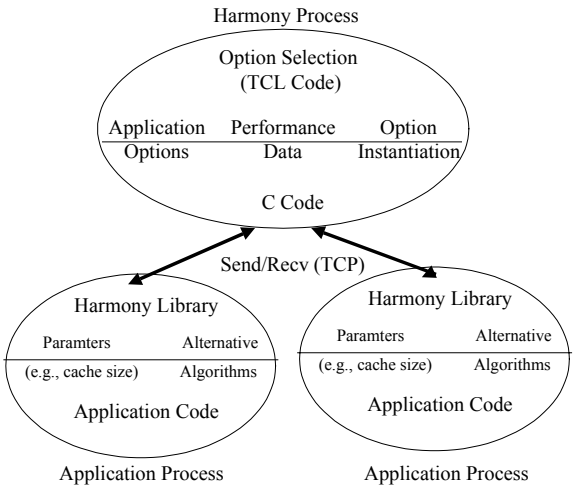


Figure 6: Architecture of Harmony Prototype.

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.

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 re-configuring 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 2.2 for a complete description of the way the evaluation is done). When option bundles are changed, the appropriate variables are updated in each application.

2.5 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, a relational database engine built on top of the SHORE (Scalable Heterogeneous Object REpository) storage manager [25, 26]. 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 [27], 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

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 2.1.1.

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 7: Harmony API Used by Application Programs.

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 8. 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.

3. Transparent system-directed application adaptation

Harmony has several means of adapting applications to their environments. Environments, consisting of local schedulers, can be controlled directly. Application execution can be steered explicitly through the RSL interface.

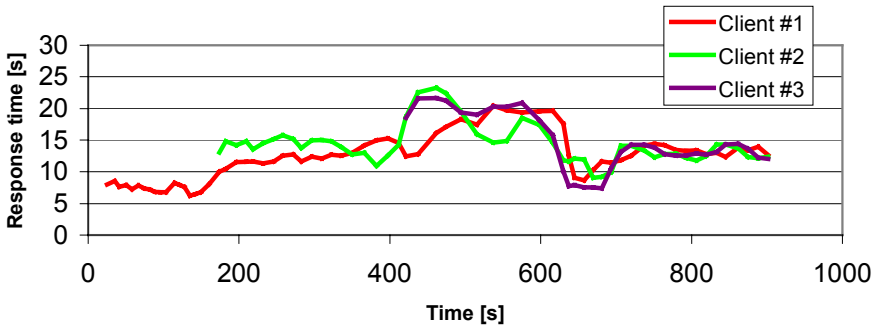


Figure 8: Client-server database application – Harmony chooses query-shipping with one or two clients, but switches all clients to data-shipping when the third client starts.

This section shows a method transparently steering application execution, i.e. a method of dramatically reconfiguring running applications that have not been explicitly "harmonized."

Harmony provides an automatically reconfigurable shared memory abstraction (DSM) to parallel applications running on networks of workstations. DSM applications are multi-threaded, and assumed to have many more threads than the number of nodes used by any one application. Overall performance depends on parallelism, load balance, latency tolerance, and communication minimization. In this section, we focus on communication minimization through *active correlation tracking* [28], a mechanism for tracking data sharing between threads, and its implementation in CVM [29]. Consistency between data located on different processors is maintained by using virtual memory techniques to trap accesses to shared data and a protocol to ensure timely propagation of these updates to other processors. Information on the type and degree of data sharing is useful to such systems because the majority of network communication is caused by the underlying consistency system. When a pair of threads located on distinct machines (nodes) both access data on the same shared page, network communication can only be avoided by moving at least one of the threads so that they are located on the same node.

In order to minimize communication, therefore, the system needs to identify the thread pairs that will cause the most communication if not located on the same node. The information should be complete, in that we need information on all threads in the system, and it must be accurate, in that small errors in the relative ordering of thread pairs might cause large differences in communication.

Ideally, sharing behavior would be measured in terms of access rates. However, a naïve implementation would add overhead to all writes, not just those that occur when the tracking mechanism is turned on. Function cloning could be used to create tracking and non-tracking versions, but every function that might possibly access shared data would have to be cloned. Current systems [30, 31], therefore, merely track the set of pages that each thread accesses. Changes in sharing patterns are usually accommodated through the use of an aging mechanism.

In any case, word-level access densities are not the proper abstraction for a page-based system. We therefore track data sharing between threads by correlating the threads' accesses to shared memory. Two threads that frequently access the same shared pages can be presumed to share data. We define *thread correlation* as the number of pages shared in common between a pair of threads. We define the *cut cost* to be the aggregate total of

thread correlations for thread pairs that must communicate across node boundaries. Cut costs can then be used to compare candidate mappings of threads to nodes in the system. Once the best mapping has been identified, the run-time system can migrate all threads to their new homes in one round of communication.

3.1.1 Thread correlations and cut costs

The cut cost of a given mapping of threads to nodes is the pairwise sum of all thread correlations, i.e. a sum with n^2 terms, where n is the number of threads. This sum represents a count of the pages shared by threads on distinct machines.

We hypothesized that cut costs are good indicators of data traffic for running applications. We tested this hypothesis experimentally by measuring the correlation between cut costs and remote misses of a series of randomly generated thread configurations. A remote miss occurs any time a process accesses an invalid shared page. Pages are invalid either because the page has never been accessed locally, or because another process is modifying the page[3]. In either case, the fault is handled by retrieving a current copy of the page from another node. For purposes of this experiment, we assume that all remote sites are equally expensive to access; thereby ensuring that the number of remote faults accurately represents the cost of data traffic.

In all but one case, we had correlation coefficients are at least 0.72. Aside from a single extraneous miss caused by CVM's garbage collection mechanism, one application's correlation coefficient would be 1.0.

3.1.2 Correlation maps

Thread correlations are used to create *correlation maps*. Correlation maps are grids that summarize correlations between all pairs of threads. We can represent maps graphically as two-dimensional squares where the darkness of each point represents the degree of sharing between the two threads that correspond to the *x,y* coordinates of that point. Correlation maps are useful for visualizing sharing behavior. For example, Figure 9a shows a correlation map for a parallel FFT with 32 threads. Note the prevalence of dark areas near the diagonals, which imply the presence of nearest-neighbor communication patterns. However, the sharing is concentrated in discrete blocks of threads, rather than being continuous.

This correlation map represents a version of FFT with 32 threads distributed equally across four nodes. The points inside the dark squares represent those thread pairs that are located on the same nodes, and hence do not figure into cut costs or require network communication. There are four squares, since there are four nodes, or regions where sharing is free. Since all of the dark regions are inside the "free zones" that represent nodes, we can infer that communication requirements will be relatively minimal.

[3] This is a gross simplification, but captures the essence.

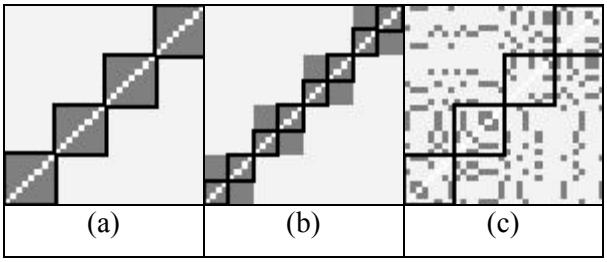


Figure 9: 32-thread FFT, 2^6 x 2^6 x 2^6 - (a) on four nodes, squares indicate thread sharing that does not cause network communication, (b) on eight nodes, as above, (c) randomized thread assignments for four nodes

Now consider instead Figure 9 (b). This picture represents a configuration of four threads running on each of eight nodes. The correlation map is the same, but the smaller "free zones" encompass only half of the dark areas. Hence, we can infer that this configuration has more communication than the four-node version. Together with information on the ratio of communication to computation in the application, a runtime system could potentially make a rough guess at whether the eight-node configuration would have any performance advantage over the four-node version.

Finally, consider Figure 9 (c). This is the same application, with unchanged sharing patterns. However, we have randomly permuted the assignment of threads to nodes. Doing so results in a configuration with a much higher cut cost, which is not addressed effectively by either the four-node or eight-node configurations. Similar situations would arise with applications in which sharing patterns change slowly over time.

3.1.3 Correlation-tracking mechanisms

Previous systems obtained page-level access information by tracking existing remote faults. Remote faults occur when local threads attempt to access invalid shared pages. Remote faults are satisfied by fetching the latest version of the shared page from the last node that modified it. The underlying DSM can overload this process to inexpensively track the causes of remote faults, slowly building up a pattern of the pages accessed by each thread.

The problem is that this approach only captures information about the first local thread that accesses a page, and captures no information about sharing between local threads. Even after multiple (ten or more) rounds of thread migrations, passive tracking only comes close to obtaining complete information for one of the applications that we tested, and this application is by far the least complex of our applications.

The solution used by Harmony is to use *active correlation tracking*. Multiple rounds of threads migrations can be avoided by obtaining additional information about correlations between local threads before any thread migration takes place. We obtain this information through an *active correlation-tracking* phase, which iteratively obtains access information for each local thread.

The reader is referred to [28] for details. In summary, however, the active approach consists of a discrete phase where page faults are forced to occur at the first access to each page by each local thread. This information is collected at the next global synchronization operation, giving the synchronization owner complete infor-

Applications		Time (secs)	Remote Misses	Total Mbytes	Cut Cost
Barnes	m-c	43.0	120730	218.1	125518
	ran	46.5	124030	254.2	129729
FFT7	m-c	37.3	22002	172.2	8960
	ran	68.9	86850	685.9	14912
LU1k	m-c	7.3	11689	121.3	31696
	ran	97.1	231117	1136.2	58576
Ocean	m-c	21.2	123950	446.3	26662
	ran	28.9	171886	605.5	29037
Spatial	m-c	240.1	125929	551.8	273920
	ran	273.7	249389	870.8	289280
SOR	m-c	3.6	881	5.4	28
	ran	5.9	8103	47.7	252
Water	m-c	19.3	20956	49.0	21451
	ran	21.1	33188	72.0	23635

Table 2: 8-node performance by heuristic

mation about page accesses by all threads in the system. This information is complete, and collected without multiple rounds of migrations. Assuming static sharing behavior, the cost of the extra local page faults can be amortized across the entire execution of the application.

3.1.4 Using correlation maps to direct migration

Thread correlations are primarily useful as a means of evaluating cut costs (and, indirectly, communication requirements) of candidate mappings of threads to nodes. Such comparisons are only meaningful if applications can be configured to match arbitrary thread mappings. Hence, reconfigurations require thread migrations. We assume a DSM system that supports per-node multithreading [32] (multiple threads per node) and thread migration. Per-node multithreading is only problematic when DSMs only allow dynamically allocated data to be shared, like CVM. The problem is that it exposes an asymmetry in the threads' view of data. Threads on a single node share the same copy of statically allocated global data, but each node has distinct copies. This problem is usually handled by restricting threads from accessing any of these variables. Instead, threads can access only stack and globally shared data.

Given the above, thread migration can be accomplished through little more than copying thread stacks from one machine to another. Care must be taken to preserve the stack's address before and after a copy so that pointer values do not become orphaned. Additionally, thread migration in systems that support relaxed consistency models must ensure that the thread's view of shared data at the destination is not missing any updates that were visible at the source.

3.1.5 Identifying good thread assignments

The combination of finding the optimal mapping of threads to nodes is a form of the *multi-way cut* problem, the general form of which is NP-hard (meaning that it is at least as hard as any problem in NP, and possibly harder).

While good approximation schemes have been found for the general form of the communication minimization problem [33], our problem is complicated by the fact that we must also address load balancing and parallelism.

For the purposes of this chapter, we restrict the problem to merely identifying the best mapping of threads to nodes, given a constant and equal number of threads on each node. We investigated several ways of identifying good mappings. We used integer programming software to identify optimal mappings. We developed several heuristics based on cluster analysis [34], and showed that two heuristics identified thread mappings with cut costs that were within 1% of optimal for all of our applications. We collectively refer to these heuristics as *min-cost* ("m-c" in the captions).

Table 2 shows communication requirements, counts of remote misses, and overall performance for each application with both *min-cost* ("m-c") and a random assignment ("ran") of threads to nodes.

4. Resource information and metrics

In order to make intelligent decisions about resource allocation and adaptation, data about the application and its behavior are required. Previously two major types of information were available. First, static performance prediction has been used to try to predict the behavior on an application when it executes on a given collection of hardware. Second, runtime-profiling tools have been used to record information about application execution to allow programmers to revise their code between executions to improve its performance. In this section, we review the first two types of information. We then present a new type of data we *call predictive metrics* that are a combination of performance prediction and runtime observation that allow adaptive systems to use data about their current execution to forecast the impact of possible runtime configuration changes.

4.1 Prediction models

An important component of the Harmony approach to adaptive systems is to include performance prediction to allow the system to evaluate tuning alternatives. Although there has been considerable work in the area of performance prediction, much of the work has concentrated on abstract system performance prediction rather than predicting the performance of a specific application. However, there has been some recent work to allow accurate prediction of application performance.

The POEMS project [35] is developing an integrated end-to-end performance prediction environment for parallel computation. They are combining analytical modeling with discrete simulation to allow different levels of fidelity in their predictions based on user need. The *Performance Recommender* allows an application programmer to select parameters for a specific problem instance by drawing on a database of information derived from previous program executions and inferred via modeling. The POEMS project differs from Harmony in that POEMS considers static information such as number of nodes, whereas the Harmony system also includes runtime information such as system load.

Schopf and Berman [36] have combined stochastic modeling of applications with runtime observations to create time varying predictions of the execution time of applications. Their approach represents the predicted

application performance as a *structural model* that represents application behavior as component models and component interactions. Values for component models (such as the bandwidth used or functional unit operation counts) are modulated by runtime observations of available resources to provide accurate predictions of the completion time of a component operation for a system with multiple users. Component interactions capture the attributes of the parallel application such as synchronization points and data decomposition.

4.2 Runtime performance metrics

To effectively adapt an application, raw performance data needs to be distilled down into useful information. Historically, performance metrics have been used to provide programmers with data to allow them to improve the performance of their application. In a system that automatically adapts, the role of performance metrics is to provide similar insights not to the programmer, but to the tuning infrastructure. In this section, we review different performance metrics, and in the next section we describe how one of them has been adapted to support automatic application tuning.

4.2.1 Parallel Performance Metrics

Simply extending sequential metrics to parallel programs is not sufficient because, in a parallel program, improving the procedure that consumes the largest amount of time may not improve the program's execution time. Inter-process dependencies in a parallel program influence which procedures are important to a program's execution time. Different parallel metrics measure and report these interactions differently. A common way to represent the execution of a parallel program is in terms of a graph of the application's execution history that incorporates both these inter-process dependencies as well was the sequential (intra-process) time. We refer to this graph as a Program Activity Graph (or PAG). Nodes in the graph represent significant events in the program's execution (e.g., message sends and receives, procedure calls and returns). Arcs represent the ordering of events within a process or the synchronization dependencies between processes. Each arc is labeled with the amount of process and elapsed time between events.

One of the first metrics specifically designed for parallel programs was Critical Path Analysis [37, 38]. The goal of this metric is to identify the procedures in a parallel program that are responsible for its execution time. The Critical Path of a parallel program is the longest CPU time weighted path through the PAG. Non-productive CPU time, such as spinning on a lock, is assigned a weight of zero. The Critical Path Profile is a list of the procedures or other program components along the Critical Path and the time each procedure contributed to the length of the path. The time assigned to these procedures determines the execution time of the program. Unless one of these procedures is improved, the execution time of application will not improve.

Although Critical Path provides more accurate information than CPU time profiles such as gprof, it does not consider the effect of secondary and tertiary paths in limiting the improvement possible by fixing a component on the Critical Path. An extension to Critical Path called Logical Zeroing [39] addresses this problem. This metric calculates the new Critical Path length when all of the instances of a target procedure are set to zero. The

difference between the original and new Critical Paths is a prediction of the potential improvement achieved by tuning the selected procedure.

Critical Path provides detailed information about how to improve a parallel program, but building the PAG and calculating the metric requires significant space and time. On-line Critical Path Analysis [40] permits computing the critical path profile value or logical zeroing value of a selected procedure during program execution. This is done by "piggy-backing" instrumentation data onto the application messages or locks.

4.2.2 Automating Performance Diagnosis

Different performance metrics provide useful information for different types of bottlenecks. However, since different metrics are required for different types of bottlenecks the user is left to select the one to use. To provide better guidance to the user, rather than providing an abundance of statistics, several tools have been developed that treat the problem of finding a performance bottleneck as a search problem.

AtExpert [41] from Cray Research uses a set of rules to help users improve FORTRAN programs written with the Cray auto-tasking library. The auto-tasking library provides automatic parallelism for FORTRAN programs; however, there are a number of directives that can that greatly affect performance. AtExpert measures a program that has been auto-tasked and attempts to suggest directives that would improve the performance of the program. Since, the tool works on a very specific programming model, FORTRAN programs on small scale shared-memory multi-processors, it is able to provide precise prescriptive advise to the user.

Cray also produced the MPP Apprentice for their T3D platforms [42, 43]. This tool differs from ATExpert in that it handles a larger variety of parallel programming semantics (not limited to helping with auto-tasking). As a result, it is more generally applicable, but provides a less definitive suggestions on how to fix your program. MPP Apprentice uses the compiler to automatically insert instrumentation into an application. This instrumentation is in the form of counters and timers, so is (relatively) compact and finite size. The compiler produces a Compiler Information File (CIF), as a guide to map the counter/timer information back to the source code of the application program. After the program completes, a Run-time Information file (RIF) is produced, containing the values of the counters and timers. MPP Apprentice includes a rich set of performance visualizations that correlate this information with the application source code. These tools allow the programmer to navigate quickly through the performance data.

Predicate profiling [44] permits comparing different algorithms for the same problem as well as the scalability of a particular algorithm. It defines a common currency, time, and then calibrate all losses in terms of how many cycles it consumed. Losses due to load imbalance, starvation, synchronization, and the memory hierarchy are reported. Results are displayed in a bar chart showing how the available cycles were spent (both to useful work and various sources of loss). Information is presented for the entire application, which provides descriptive information about the type of bottleneck. However, they do not include suggestions about how to fix the problem or information about which procedure contain the bottleneck.

Another approach is to provide a search system that is independent of the programming model and machine architecture. Paradyn's Performance Consultant [45] uses a hierarchical three axis search model (the "why", "where", and "when" of a performance bottleneck). The "why" axis represents hypotheses about potential bottlenecks in a parallel program (i.e., message passing, computation, I/O). The "where" axis defines a collection of resource hierarchies (CPU, interconnect, disk) that could cause bottleneck. The "when" axis isolates the bottleneck to a specific phase of the program's execution. A unique feature of the Performance Consultant is that it searches for bottlenecks while the program is executing. This requires an adaptive style of instrumentation, but it can greatly reduce the volume of performance data that needs to be collected. Only the performance data required to test the current hypothesis for the currently selected resources need be collected.

4.3 Adaptation metrics

In order to make informed choices about adapting an application, Harmony needs metrics to predict the performance implications of any changes. To meet this need, we have developed a metric called Load Balancing Factor (LBF) to predict the impact of changing where computation is performed. This metric can be used by the system to evaluate potential application reconfigurations before committing to potentially poor choices.

We have developed two variants of LBF, one for process level migration, and one for fine-grained procedure level migration. Process Load Balancing Factor (LBF) predicts the impact of changing the assignment of processes to processors in a distributed execution environment. Our goal is to compute the potential improvement in execution time if we change the placement. Our technique can also be used to predict the performance gain possible if new nodes are added. Also, we are able to predict how the application would behave if the performance characteristics of the communication system were to change.

To assess the potential improvement, we predict the execution time of a program with a virtual placement, during an execution on a different one. Our approach is to instrument application processes to forward data about inter-process events to a central monitoring station that simulates the execution of these events under the target configuration.

The details of the algorithm for process level-LBF are described in [46]. Early experience with process-LBF is encouraging. Figure 10 shows a summary of the measured and predicted performance for a TSP application, and four of the NAS benchmark programs [47]. For each application, we show the measured running time for one or two configurations and the predicted running time when the number of nodes is used. For all cases, we are able to predict the running time to within 6% of the measured time.

While process LBF is designed for course-grained migration, procedure-level LBF is designed to measure the impact of fine-grained moved of work. The goal of this metric is to compute the potential improvement in execution time if we move a selected procedure, F, from the client to the server or visa-versa.

Application Target	Meas. Time	Pred.	Error		Pred.	Error	
TSP			4/1			4/1	
4/4	85.6	85.5	0.1	(0.1%)	85.9	-0.3	(-0.4%)
4/1	199.2	197.1	2.1	(1.1%)	198.9	0.3	(0.2%)
EP - class A			16/16			16/8	
16/16	258.2	255.6	2.6	(1.0%)	260.7	-2.5	(-1.0%)
FT- class A			16/16			16/8	
16/16	140.9	139.2	1.7	(1.2%)	140.0	0.9	(0.6%)
IS- class A			16/16			16/8	
16/16	271.2	253.3	17.9	(6.6%)	254.7	16.5	(6.0%)
MG- class A			16/16			16/8	
16/16	172.8	166.0	6.8	(4.0%)	168.5	4.3	(2.5%)

Figure 10: Measured and predicted time for LBF. For each application, we show 1-2 target configurations. The second column shows the measured time running on this target configuration. The rest of the table shows the execution times predicted by LBF when run under two different actual configurations.

The algorithm used to compute procedure is based on the Critical Path (CP) of a parallel computation (The longest process time weighted path through the graph formed by the inter-process communication in the program). The idea of procedure LBF is to compute the new CP of the program if the selected procedure was moved from one process to another[4].

In each process, we keep track of the original CP and the new CP due to moving the selected procedure. We compute procedure LBF at each message exchange. At a send event, we subtract the accumulated time of the selected procedure from the CP of the sending process, and send the accumulated procedure time along with the application message. At a receive event, we add the passed procedure time to the CP value of the receiving process **before** the receive event. The value of the procedure LBF metric is the total effective CP value at the end of the program's execution. Procedure LBF only approximates the execution time with migration since we ignore many subtle issues such as global data references by the "moved" procedure. Figure 11 shows the computation of procedure LBF for a single message send. Our intent with this metric is to supply initial feedback to the programmer about the potential of a tuning alternative. A more refined prediction that incorporates shared data analysis could be run after our metric but before proceeding to a full implementation.

We created a Synthetic Parallel Application (SPA) that demonstrates a workload where a single server becomes the bottleneck responding to requests from three clients. In the server, two classes of requests are processed: *servBusy1* and *servBusy2*. *ServBusy1* is the service requested by the first client and *servBusy2* is the service requested by the other two clients.

The results of computing procedure LBF for the synthetic parallel application are shown in Figure 13. To validate these results, we created two modified versions of the synthetic parallel application (one with each of *servBusy1* and *servBusy2* moved from the server the clients) and measured the resulting execution time[5]. The

[4] Our metric does not evaluate *how* to move the procedure. However, this movement is possible if the application uses Harmony's shared data programming model.
[5] Since Harmony's shared data programming model is not yet fully implemented, we made these changes by hand.

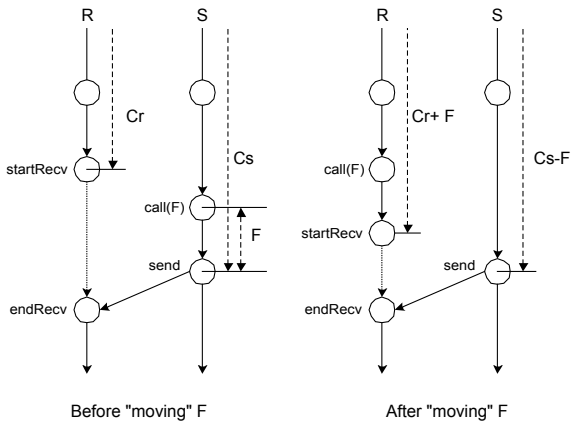


Figure 11: Computing procedure LBF - The PAG before and after moving the procedure F. The time for the procedure F is moved from the sending process (which is on the application's critical path) to the receiving one (which is not).

results of the modified programs are shown in the third column of Figure 13. In both cases, the error is small indicating that our metric has provided good guidance to the application programmer.

For comparison to an alternative tuning option, we also show the value for the Critical Path Zeroing metric [40]. CP Zeroing is a metric that predicts the improvement possible due to optimally tuning the selected procedure (i.e., reducing its execution time to zero) by computing the length of the critical path resulting from setting the time of the selected procedure to zero. We compare LBF with Critical Path Zeroing because it is natural to consider improving the performance of a procedure itself as well as changing its execution place (processor) as tuning strategies.

The length of the new CP due to the movement of *servBusy1* is 25.4 and the length due to *servBusy2* is 16.1 while the length of the original CP is 30.7. With the Critical Path Zeroing metric, we achieve almost the same benefit as tuning the procedure *ServBusy1* by simply moving it from the server to the client. Likewise, we achieve over one-half the benefit of tuning the *ServBusy2* procedure by moving it to the client side.

4.4 Other sources of information

In addition to gathering data directly from applications, it is important to incorporate performance data into the system from other sources such as the operating system or the network.

Procedure	Procedure LBF	Measured Time	Difference
ServBusy1	25.3	25.4	0.1 (0.4%)
ServBusy2	23.0	23.1	0.1 (0.6%)

Figure 13: Procedure LBF accuracy.

Procedure	LBF	Improve-ment	CP Zeroing	Improve-ment
ServBusy1	25.3	17.8%	25.4	17.4%
ServBusy2	23.1	25.1%	16.1	47.5%

Figure 12: Procedure LBF vs. CP Zeroing.

4.4.1 Operating System Instrumentation Techniques

Operating systems are another important source of data for adaptable systems. While it is possible to instrument an operating system, generally it is better if the basic instrumentation is already built into it. For example, most operating systems keep track of statistics about the virtual memory system, file system and file cache. However, since this data is intended for use by the operating system or for system administration tools these counters are difficult to gather. To support application oriented instrumentation systems, it is important that OS level counters be exposed via APIs that permit efficient access from user processes. An example of such a facility is the reading performance counters from the UNIX kernel memory via the /dev/kmem psuedo-device. Harmony (and other resource-aware meta-computing environments) run atop commodity operating and thus making this type of data available in current operating systems is critical.

4.4.2 Active network monitoring

There are two major ways to measure a network: passive monitoring and active monitoring. Passive network monitoring inserts measurement systems or instruments network components to observe and record the information about the traffic that passes through the network. Active monitoring involves injecting new traffic into the network, or changing the transmission of existing traffic. Active monitoring can gather additional information not available via passive monitoring; however, because it alters the traffic on the network it is more intrusive.

A basic type of passive monitoring for local area networks takes advantage of the fact many local area networks use broadcasts at the physical media level. For example, the *packet filter* [48] puts an Ethernet adapter into promiscuous mode and then observes all traffic on that segment. Since gathering all traffic on a network can result in an enormous amount of data, the packet filter provides a simple predicate language to filter traffic based on packet content. With switched Ethernet, it is no longer possible to implement a packet filter at an arbitrary compute node. However, many switches provide a monitoring port that can be configured to receive all, or a filtered subset, of the traffic passing through the switch.

In addition, the RMON protocol [49] provides a way for hubs and switches to record and report statistics about the traffic passing through them. The RMON protocol defines a set of SNMP variables that can be extracted and displayed by any SNMP compliant monitoring station. RMON permits gathering statistics about packet counts, a matrix of traffic by sending and receiving host, and statistics about selected TCP and UDP ports.

A basic type of active monitoring is to use Internet Control Message Protocol (ICMP) echo packets, often called *ping* packets, to measure the network performance between two hosts. By sending an echo packet and noting the time it leaves the host and the time when it returns, it is possible to compute the average round-trip delay between hosts. By attaching a sequence number to each echo packet, estimates of network losses may be made by counting the number of lost packets. In addition, since the size of ICMP echo packets can be varied it is possible to use them to estimate the available bandwidth between hosts.

A second type of active monitoring is to exploit the time-to-live field in IP packets to discover the route taken between two hosts. This technique is used by the ``traceroute" utility. All IP packets have a time-to-live field which limits the number of hops that a packet can travel. This field was originally intended to prevent packets from looping through the network when a routing error caused a cycle in the path between two hosts. At each hop in the network, the time-to-live field is decremented by one. When the count reaches zero, the packet is dropped and an ICMP message containing the identity of the host where the packet was dropped is sent back to the originating host. By setting the time-to-live field to one, a host can discover the first hop in routing a packet to a destination. By repeatedly incrementing the time-to-live field, a sender can discover the entire route to the desired destination. Since IP networks route traffic on a per packet basis, it is possible that the routes taken by these probe packets may be different. However, over short periods of time, the routes taken by packets bound for the same destination tend to travel the same path. Like echo packets, trace-route packets can be used to gather information about the time packets take to reach their target. By comparing the return times of adjacent nodes it is possible to identify the bottleneck link between the source and destination.

Another way to estimate the delays in the network is based on sending “packet pairs”, two back-to-back packets [50, 51]. The key idea of this approach is that by measuring the difference in the arrival times of the two packets it is possible to estimate the queuing delay of the bottleneck switch between the sender and receiver.

Network Weather Service [52, 53] provides dynamic resource forecasts in meta-computing environments. The service gathers the time varying load data from a distributed set of sensors. The load data includes CPU availability and network performance (bandwidth between two nodes). Numerical models are then used to predict and forecast the load conditions for a given time frame. Their prediction models embody various stochastic techniques such as mean-based methods, median-based methods and auto-regressive models. Since different estimation techniques yield the best forecasts at different times, the Network Weather Service dynamically chooses one prediction model based on the error between all predictors and sampled data.

4.4.3 Hardware counters

Some performance information only can be provided efficiently with hardware support. This information includes high resolution timers, memory system performance, and internal details of processor use (such as floating point unit utilization or FLOP counts). Fortunately, many modern processors provide high-resolution performance data via special registers or memory-mapped locations.

A crucial hardware feature for instrumentation is an accurate clock. To be useful to measure fine-grained events, a clock should provide sufficient resolution so that it does not roll over during an application’s execution. For example, at current processor clock rates, a 32 bit counter will roll in less than 10 seconds. 64 bit clocks should be considered a basic requirement. Most recent micro-processors include high resolution clocks[54-57] include high resolution clocks.

However, high resolution alone is not sufficient, for a clock to be useful it must be accessible with low latency to permit measuring fine-grained events. Clock operations need to be supported by user-level instructions

that execute with similar performance as register-to-register instructions. The Intel Pentium family provides a clock that meets this requirement, however SPARC v9 does not.

As more and more features are integrated onto a single chip, it is increasingly important that instrumentation be incorporated into the chips since many useful types of information are no longer visible to an external instrumentation system. Modern processors, such as the Sun UltraSPARC, Intel Pentium Pro, and IBM Power2 provide a rich collection of performance data. For example, the Pentium Pro provides access to its performance data through the Model Specific Registers (MSR's). These registers include counts of memory read & writes, L1 cache misses, branches, pipeline flushes, instructions executed, pipeline stalls, misaligned memory accesses, bus locked cycles, and interrupts. The UltraSPARC, Power2, and MIPS R10000 provide a similar set of counters.

In addition to processors it is increasing important that other system components provide instrumentation at the hardware level. For example, network interfaces, and I/O systems are increasing using high speeds and higher levels of integration. One system that provided detailed system instrumentation was the IBM RP3 [58] which included a passive hardware monitor, with instrumentation built into almost every subsystem of the computer. Each device recognized its own events and passed the information through I/O pins to a Performance Monitor Chip (PMC) which counts the events. The PMC also sampled memory references to provide statistics about memory usage. The capabilities were limited, however, due to constraints on space and cost imposed on the designs.

5. Related work

Although we have tried to present much of the work in resource ware computing throughout this chapter, in this section we summarize some of the key enabling technology for resource-ware computing, and present a few projects that have looked at various aspects of the problem.

5.1 Meta-computing and adaptive applications

Globus [59] and Legion [60] are major projects to build an infrastructure for meta-computing. They are trying to address various requirements to seamlessly aggregate heterogeneous computing resources. The required services include global naming, resource location and allocation, authorization, and communications. Prospero Resource Manager (PRM) [61] also provides a uniform resource access to the nodes in different administration domains so that users don't have to manage them manually. By contrast, our work is concentrating on the specific problem of developing interfaces and policies to allow applications to react to their computing environment

From the application's view, it is advantageous for the application to adjust itself to changing resource status since the application knows more than the underlying resource manager how to obtain good performance from different resources. Based upon this concept, the AppLeS [62] project developed application-centric scheduling. AppLes 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 schedul-

ing to maximize its own performance. This is different from most other systems, which strive to enhance system-wide throughput or resource usage. The Network Weather Service [52] is used to forecast the network performance and available CPU percentage to AppLeS agents so that the applications can adapt by appropriate scheduling. 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.

Dome [63] is another parallel programming model which supports application-level adaptation using load balancing and checkpointing. While the load balancing for the different CPU and network performance is transparent, the programmers are responsible for writing suitable checkpointing codes using provided interfaces.

The Odyssey [64] project also focuses 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. When the application is informed that the resource availability goes outside the requested bounds, it changes the fidelity and tries to register a revised window of tolerance. For example, when the available network bandwidth decreases, the video application can decrease the fidelity level by skipping frames, and thus displaying fewer frames per second.

EMOP[65] provides mechanisms and services (including object migration facilities) that allow applications to define their own load-balancing and communication services. Its programming model is based on CORBA and uses an Object Request Broker (ORB) for communications between application components. EMOP supports multiple, possibly user defined, communication protocols. Its automatic protocol selection adaptively chooses the most suitable protocol at run-time. However, the decision is made based only on a predefined order. The first available protocol will be selected from an ordered list of preferred protocols. Therefore, EMOP cannot consider changing available bandwidth in choosing communication protocol. The load-balancing mechanisms are based on Proxy Server Duality. The server object acts as a proxy when the load increases, and forwards the requests to other server objects. When the load decreases, it switches back to server mode, and processes the requests.

5.2 Computational steering

Computational Steering [66-69] 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 [66], 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 applications.

5.3 Idle-cycle harvesting and process migration

Many research prototypes and practical systems have been developed to harvest those idle cycles. Condor [70] is built on the principle of distributing batch jobs around a cluster of computers. It identifies idle workstations and schedules background jobs on them. The primary rule Condor attempts to follow is that workstation owners should be able to access their machine immediately when they want it. To do this, as soon as the machine's owner returns, the background job is suspended and eventually migrated to another idle machine. This low perturbation led to successful deployment of the Condor system. For fault tolerance, Condor checkpoints jobs periodically for restoration and resumption. It also provides machine owners with the mechanisms to individually describe in what condition their machine can be considered idle and used. IBM modified Condor to produce a commercial version, Load Leveler [71]. It supports not only private desktop machines but also IBM's highly parallel machines.

Sprite [72] is another system providing process migration to use only idle machines and respecting the ownership of workstations. A migrated process is evicted when the owner reclaims their machine. The major difference from Condor is that job migration is implemented at the operating system level. The advantage is the migration overhead is much lower: a few hundred milliseconds while user level evictions typically occur in a few seconds. However, the fully custom OS kernel hinders the wide deployment and extension to the heterogeneous environment.

DQS [73] is an early non-commercial cluster computing system. Like Condor, it supports most of the existing operating systems. Different job queues are provided based on architecture and group. DQS ensures the local autonomy of a private machine by suspending currently running background jobs when keyboard or mouse activities are detected. However, their emphasis is placed on distributing the jobs to different shared machine clusters in a balanced manner. Jobs can be suspended and resumed, but migration is not supported. DQS 3.0 [74], the latest version, is widely used by companies such as Boeing and Apple Computer.

Load Sharing Facility (LSF) [75] was developed to automatically queue and distribute jobs across a heterogeneous network of Unix computers. This system was intended for much larger groups of workstation clusters consisting of thousands of machines. The basic assumption on each participating machine is that a host is by default sharable, which is opposite to the policy of Condor-like systems. Local autonomy is not respected, and hence few mechanisms are provided to protect machine owners. LSF focuses on two major goals. The first is to place jobs on nodes that meet the application's requirement. The second is to balance the load across machine clusters to achieve better turn-around time of jobs and system utilization in the entirety. Since job migration is

not supported, the process should finish on the node it started. It also lacks the checkpointing mechanism, so remote jobs are vulnerable to node failure.

The Butler system [76] also gives users access to idle workstations. This system requires the Andrew System [77] for shared file access. The basic concept of this system is to provide transparent remote execution on idle nodes. Lack of support for job migration in Butler can lead to loss of work by remote jobs when the machine owner returns. The system will just warn the remote user, then kill the process so as not to disturb the machine owner.

There is one approach, which supports local autonomy of individual machines in a different way. The Stealth system [2] runs remote processes with lower priority to preserve the performance of local work. Thus, when the owner reclaims their machine, the remote job does not leave the node, rather keeps running with low priority. They prioritized several major resources including CPU, memory and the file buffer cache on the MACH operating system so as not to interfere with local processes. This is similar to our fine-grain cycle stealing approach. However, lack of support for job migration can lead to extreme delay or even starvation of background jobs. Also, this system is not intended for parallel jobs.

There have been studies of specific issues involved in using idle cycles. Theimer and Lantz [78] investigated into how to find idle machines more efficiently. They found that a centralized architecture can be scaled better and can be more easily monitored, while a decentralized architecture is easier to implement. Bhatt et al [79] investigated finding an optimal work size to run remotely assuming that the partial result will be lost when idle machines are reclaimed. In their cycle stealing model, too small chunks of remote work will suffer from high network overhead and too large chunks can waste cycles by losing larger partial work. While checkpointing can avoid this problem, a study showed that it should be carefully designed. Basney and Livny [80] used data from the Condor system to show that their initial matchmaking and checkpointing design could cause a bursty utilization of network bandwidth, and thus interfere with interactive processes and even unrelated remote job migrations. They suggested a better design that gives priority to applications with low network requirements. Also, this approach limits the bandwidth consumed by job placement and checkpointing for a given time period.

Immediate migration does not always satisfy machine owners since it takes a noticable time to recover the previous state of machine such as CPU cache, I/O cache buffers and memory pages. Arpaci et al [81] limited this perturbation by restricting the number of times when returning users notice disruptions on their machine to a fixed limit per day. Petrou et al [82] present a more complex solution to increase the available time of idle machines. Their system predicts when a user will return based on the past history and actively restores the memory-resident state in anticipation of a user returning.

General load balancing and process migration mechanisms have been studied extensively. MOSIX [83] provides load-balancing and preemptive migration for traditional UNIX processes. DEMO/MP [84], Accent [85], Locus [86], and V [87] all provide manual or semi-automated migration of processes.

5.4 Parallel job scheduling

Non-interactive computation-intensive applications are often found in the form of parallel programs. Since a collection of idle machines connected by a high speed network can be viewed as a virtual parallel machine, parallel computing in this environment is a natural match. However, this is far more complicated than running multiple independent sequential jobs on idle machines. Furthermore, multiple parallel jobs should be able to be served at the same time for two reasons. First, a very large pool of workstations can be wasted if only single parallel job can run at once. Second, the response time of each parallel job should be acceptable as well as the system throughput. In general, scheduling of multiple parallel programs is classified to two styles: time sharing and space sharing [88].

In time shared scheduling, different parallel jobs share the nodes and take turns for execution. However, global coordination across the processors are essential since independent process switching on each node will lead to large inefficiencies. In Gang scheduling, context switches between different parallel jobs occurs simultaneously at all the processors in a synchronized manner. Thus, constituent processes(or threads) can interact at a fine granularity.

There have been extensive studies to efficiently implement the Gang scheduling. Ousterhout [89] suggested and evaluated a few algorithms for coscheduling. He verified that avoiding fragmentation of slots for processes on processors are critical for system throughput. A number of variants of coscheduling were also explored. Sbalvarro et al [90] presented demand-based coscheduling. This dynamic scheduling algorithm coschedules the processes exchanging messages. The implicit coscheduling work by Dessau et al [91] shows that coscheduling can be achieved implicitly by independent decisions of local schedules based on the communication pattern. This study focuses more on how to immediately schedule the corresponding process and keep the communicating processes on the processors without blocking.

Space sharing partitions the nodes and executes a number of applications side by side. This has the advantage of reducing the operating system overhead on context switching [92]. However, an important constraint is that activities on one partition should not interfere with the others. Therefore, processor partitioning in a classic parallel machine needs to be aware of the interconnection architecture between processing units. The simplest way of space slicing is fixed partitioning. Fixed partitions are set by the system administrators. Certain partitions can be dedicated to a certain group of users or different job classes [93, 94]. Many commercial systems split the system into two partitions: one for interactive jobs and the other for batch jobs. This is because the responsiveness of interactive jobs shouldn't be compromised by a heavy load of batch jobs. In spite of its simplicity, this system will suffer from internal fragmentation. Variable partitioning can change the partition size upon job's requests. Nonetheless, external fragmentation remains an issue because free processors left might not be enough for any queued jobs. Another issue is the scheduling decision: which job in the queue should be scheduled first. Obviously, first-come-first-service will introduce more external fragmentation. The "shortest job first" can give better average response time, but starve long jobs. In addition, it is not easy to know the lifetime of submitted jobs in advance. Other policies such as "smaller job first" and the opposite, "longer job first" were

explored and turned out to be not much better than a simple FCFS scheduling. Backfilling [11] is another scheduling algorithm to reduce external fragmentation while still offering fairness to queued jobs with respect to their arrival time.

The flexibility of applications can give more opportunity for the scheduler for global optimization. Many parallel programs can be written so that they can run on different number of processors. However, this does not necessarily mean that it can adapt the partition size at run time. For these parallel jobs, the scheduler can decide the number of processors to be allocated. Fairness plays a more important role here since the decision is made mostly by the scheduler with little application involvement. Various on-line algorithms were suggested. Equipartition [95] allocates the same number of processors to all queued jobs. However, since "non-malleable" jobs cannot change the level of parallelism at run time, all running jobs will not have the same number of processors as jobs come and go. Equipartition works as it is intended when treating "moldable" jobs that can adapt the partition size at run time. However, too frequent reconfiguration should be avoided to limit overhead. For dynamic partitioning with moldable parallel jobs, two different schemes have been proposed: the two-level scheduler [96] designed at University of Washington and the "process control" scheduler [92] designed in Stanford University. An interesting study on running run-time reconfigurable parallel jobs was done by Zahorjan et al [97]. They measured the job efficiency of different processor allocations and found the best configurations yielding maximum speedup at run time. This self-tuning is based on the fact that the speedup stops increasing after a certain size of the partition due to a heavy communication overhead. Studies showed that the memory constraint should be considered since it can put a lower bound on the partition size [98] and below this lower bound, using virtual memory unacceptably degrades the parallel job performance due to heavy paging [99].

The scheduling policies surveyed above were originally developed for dedicated parallel systems. Thus, they cannot be directly applied to the network of workstation environment where interactive jobs require a quick response time. In most such systems, local interactive processes are not controllable and should be protected from aggressive background jobs. Parallel jobs are much more difficult to run on idle machines than sequential jobs because the suspension of one constituent process can block the whole parallel job resulting in poor system usage.

There have been many studies on running parallel jobs on non-dedicated workstation pools. Pruyne and Livny [100] interfaced the Condor system and PVM [6] through CARMI (Condor Application Resource Management Interface) to support parallel programming. Their Work Distributor helped the parallel job adapt as the resources came and went. The MIST [101] project also extended PVM to use only idle machines. The distinction between the two systems is that CARMI requires a master-workers style programming and the inclusion and exclusion of machines is handled by creation and deletion of new worker processes, whereas MIST migrates running PVM processes. Piranha [102] works similarly, but it is restricted to programs using Linda [103] tuple-space based communication, whereas CARMI and MIST can serve in a general message passing environment. Cilk-NOW [104] also supports this adaptive parallelism for parallel programs written in Cilk. When a given

workstation is not being used by its owner, the workstation automatically joins in and helps with the execution of a Cilk program. When the owner returns to work, the machine automatically retreats from the Cilk program.

The NOW project [105] investigated various aspects of running parallel jobs on a network of workstations. They developed the River system [106] that supports I/O intensive parallel applications, such as external sort, running on dynamically changing resources. Their load balancing scheme, using distributed queue and data replication, removes the drastic performance degradation of a I/O intensive parallel application due to the reduced I/O bandwidth on some nodes. Another study [81] in the NOW project investigated, through simulation, running parallel jobs and interactive sequential jobs of local users. This showed that a non-dedicated NOW cluster of 60 machines can sustain a 32-node parallel workload. Acharya et al [3] also studied running adaptive parallel jobs on a non-dedicated workstation. Their experiments confirmed NOW's 2:1 rule in running parallel jobs. They also showed that the parallel job throughput depends on the flexibility of adaptive parallel jobs. Restricting the possible number of constituent processes to a certain number, like power of two, would yield a poor performance since not all of the available nodes are used.

5.5 Local scheduling support by operating systems

The concept of "time-sharing" a machine has been widely adopted and served for a long time to provide a good response time to each process and better system utilization. For the same class of jobs, the system strives to ensure the "fairness" in allocating the CPU time to existing processes. Most current UNIX systems [107, 108] are using dynamic priority adjustment to achieve fair scheduling. For example, if a process releases the CPU before its time-quanta expires, it is rewarded by a temporary priority boost for the future. On the other hand, it is sometimes necessary to treat some types of processes differently or unfairly. A typical example is the case of running CPU-intensive jobs in background with interactive jobs. Unix systems provide users with different user-level priorities so that processes that are more important can execute more frequently by using a higher priority. However, the definition of "priority" is quite different depending on implementation. In addition, the priority is enforced only to CPU scheduling, thus, can be compromised by competition for other resources such as memory and disk I/O.

Some systems provide special local resource scheduling for different classes of jobs. Host processes that belong to the machine owner should run as if there were no other activities. Guest processes, which are either initially intended as background jobs or moved from other machines to balance the load, can use only the time and resources which are not used by host processes. The Stealth Distributed Scheduler [2] supports this by a local scheduler that protects the performance of owner's processes. They prioritized not only CPU scheduling but also memory management and file accesses. Stealth was implemented on a network of Sun 3/50 and IBM RT workstations using a customized version of Mach 2.5.

Verghese et al. [109] investigated dynamic sharing of multi-user multiprocessor systems. Their primary scheme is to isolate the performance of the processes belonging to the same logical entity, such as a user. Logical smaller machines named Software Performance Units (SPU) were suggested to achieve two performance

goals: 1) Isolation: If the allocated resources meet the requirement of an SPU, its performance should not be degraded by the load placed to the systems by others, and 2) Sharing: If the allocated resources are less than the requirement, the SPU should be able to improve its performance utilizing idle resources. Like Stealth, only idle time of the CPU and unused memory can be loaned to non-host SPUs and will be revoked immediately when the host SPU needs them. Their system was implemented in the Silicon Graphics IRIX operating system. While their approach requires changes to similar parts of the operating system, their primary goal was to increase fairness to all applications, while our goal is to create an inherently unfair priority level for guest processes. Having logical smaller machines in this study is similar to the classic virtual machine concept developed by IBM machines [110]. However, virtual machines on a physical machine was developed mainly to provide an illusion of having multiple machines and run different operating systems on a single machine. The system efficiency can be limited due to internal fragmentation resulted from lack of support for dynamic resource sharing between virtual machines.

In the current version of IRIX operating system [111], the Miser feature provides deterministic scheduling of batch jobs. Miser manages a set of resources, including logical CPUs and physical memory, that Miser batch jobs can reserve and use in preference to interactive jobs. This strategy is almost opposite of our approach, which promotes interactive jobs.

6. Conclusions

Tying together multiple administration domains and systems is the most cost-effective method of meeting today's high-end computational demands. However, this approach poses a number of difficult challenges, most notably that of dealing with dynamic and heterogeneous environments. We feel the best way to address the issue of dynamic environments is to use a rich interface between applications and the system and to allow the system fine-grained control over application resource use. Much of our work has focused on building an API that is expressive enough to define real-world alternatives, and the scheduling infrastructure that can use this information to improve application performance. The remaining missing piece is that of deriving simple, accurate performance models. A great deal of work has been done in performance models in other domains. It remains to be seen whether this work can be directly applied to meta-computing.

7. Acknowledgements

We tank the other members of the Harmony Project (Bryan Buck, Hyeonsang Eom, Dejan Perkovic, Kritchalach Thitikamol) for their contributions to the project. We also thank Dr. Leana Golubchik for her insights about performance modeling and prediction. We gratefully acknowledge the sponsors of the project: NSF (awards ASC-9703212, CCR-9624803 and ACI-9711364), and DOE (Grant DE-FG02-93ER25176).

References

1. M. W. Mutka and M. Livny, "The available capacity of a privately owned workstation environment," *Performance Evaluation*, **12**, 1991, pp. 269-284.

2. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *International Conference on Distributed Computing Systems (ICDCS)*. May 1991, Arlington, TX, pp. 336-343.

3. A. Acharya, G. Edjlali, and J. Saltz, "The Utility of Exploiting Idle Workstations for Parallel Computation," *SIGMETRICS'97*. May 1997, Seattle, WA, pp. 225-236.

4. R. Raman, M. Livny, and M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing," *The 7th International Symposium on High-Performance Distributed Computing*.

5. 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.

6. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*. 1994, Cambridge, Mass: The MIT Press.

7. J. K. Osterhout, "Tcl: An Embeddable Command Language," *USENIX Winter Conf.* Jan 1990, pp. 133-146.

8. 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.

9. N. Padua-Perez, *Performance Analysis of Relational Operator Execution in N-Client 1-Server DBMS Architectures*,Masters, Computer Science University of Maryland,1997.

10. J. K. Hollingsworth, "Critical Path Profiling of Message Passing and Shared-memory Programs," *IEEE Transactions on Parallel and Distributed Computing*, **9**(10), 1998, pp. 1029-1040.

11. D. G. Feitelson and A. M. a. Weil, "Utilization and Predictability in Scheduling the IBM SP2 with Backfilling," *2th Intl. Parallel Processing Symposium*. April 1998, Orlando, Florida, pp. 542-546.

12. L. Golubchik and J. C. S. Lui, "Bounding of Performance Measures for a Threshold-based Queueing System with Hysteresis," *Proceedings of 1997 ACM SIGMETRICS Conf.*, Seattle, WA.

13. J. C. S. Lui and L. Golubchik, "Stochastic Complement Analysis of Multi-Server Threshold Queues with Hysteresis," *To appear in the Performance Evaluation Journal*, .

14. M. Y.-Y. Leung, J. C.-S. Lui, and L. Golubchik, "Buffer and I/O Resource Pre-allocation for Implementing Batching and Buffering Techniques for Video-on-Demand Systems.," *Proceedings of the Intl. Conference on Data Engineering (ICDE '97)*, Birmingham, UK.

15. L. Golubchik, J. C. S. Lui, E. d. S. e. Silva, and H. R. Gail, *Evaluation of Tradeoffs in Resource Management Techniques for Multimedia Storage Servers Technical Report CS-TR#3904*, University of Maryland, .

16. R. Pooley and P. Stevens, *Component Based Software Engineering with UML*. November 1998: Addison-Wesley.

17. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer*, **28**(11), 1995, pp. 37-46.

18. K. D. Ryu and J. K. Hollingsworth, "Linger Longer: Fine-Grain Cycle Stealing for Networks of Workstations," *SC'98*. Nov. 1998, Orlando, ACM Press.

19. A. Tamches and B. P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels," *Third Symposium on Operating Systems Design and Implementation (OSDI)*. (to appear) February 1999, New Orleans.

20. K. D. Ryu, J. K. Hollingsworth, and P. J. Keleher, "Mechanisms and Policies for Supporting Fine-Grained Cycle Stealing," *ICS*. June 1999, Rhodes, Greece, pp. 93-100.

21. K. J. McDonell, "Taking Performance Evaluation Out of the 'Stone Age'," *Summer USENIX Conference*. June 1987, Phoenix, AZ, pp. 8-12.

22. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 1995, pp. 24-37.

23. C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, February 1996.

24. P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," *ICDCS*. May 1996, Hong Kong, pp. 91-98.

25. M. Carey, *et al.*, "Shoring Up Persistent Applications," *ACM SIGMOD*. May 24 - 27, 1994, Minneapolis, MN.

26. 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.

27. J. Gray, *The Benchmark Handbook for Database and Transaction Processing Systems*. Second ed. 1993, San Mateo, CA: Morgan Kaufmann.

28. K. Thitikamol and P. J. Keleher, "Active Correlation Tracking," *The 19th International Conference on Distributed Computing Systems*.

29. P. Keleher, "The Relative Importance of Concurrent Writers and Weak Consistency Models," *Proceedings of the 16th International Conference on Distributed Computing Systems*. May.

30. A. Itzkovitz, A. Schuster, and L. Wolfovich, *Thread Migration and its Applications in Distributed Shared Memory Systems*, LPCR #9603, Technion IIT, July.

31. Y. Sudo, S. Suzuki, and S. Shibayama, "Distributed-Thread Scheduling Methods for Reducing Page-Thrashing," *Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing*.

32. K. Thitikamol and P. Keleher, "Multi-Threading and Remote Latency in Software DSMs," *The 17th International Conference on Distributed Computing Systems*.

33. E. Dahlhaus, D. S. Johnson, C. H. Papdimitriou, P. D. Seymour, and M. Yannakakis, "The Complexity of Multiterminal Cuts," *SIAM Journal on Computing*, **23**, 1994, pp. 864-894.

34. R. A. Jarvis and E. A. patrick, "Clustering using a similarity based on shared near neighbors," *IEEE Transactions on Computers*, **C-22**(11), November 1973.

35. E. Deelman, *et al.*, "Poems: end-to-end performance design of large parallel adaptive computational systems," *WOSP: International Workshop on Software and Performance*. Oct. 1998, Santa Fe, NM, pp. 18-30.

36. J. M. Schopf and F. Berman, "Performance Prediction in Production Environments," *IPPS/SPDP*. April 1998, Orlando, FL,, IEEE Comput. Soc, pp. 647-53.

37. C.-Q. Yang and B. P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs," *8th Int'l Conf. on Distributed Computing Systems*. June 1988, San Jose, Calif., pp. 366-375.

38. C.-Q. Yang and B. P. Miller, "Performance measurement of parallel and distributed programs: A structured and automatic approach," *IEEE Trans. Software Eng.*, **12**, 1989, pp. 1615-1629.

39. J. K. Hollingsworth, R. B. Irvin, and B. P. Miller, "The Integration of Application and System Based Metrics in A Parallel Program Performance Tool," *1991 ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*. April 21-24 1991, Williamsburg, VA, pp. 189-200.

40. J. K. Hollingsworth, "An Online Computation of Critical Path Profiling," *SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*. May 22-23, 1996, Philadelphia, PA, pp. 11-20.

41. J. Kohn and W. Williams, "ATExpert," *Journal of Parallel and Distributed Computing*, **18**(2), 1993, pp. 205-222.

42. W. Williams, T. Hoel, and D. Pase, *The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D*, in *Programming Environments for Massively Parallel Distributed Systems*. 1994, North-Holland.

43. D. Pase and W. Williams, *A Performance Tool for the Cray T3D*, in *Debugging and Performance Tuning of Parallel Computing Systems*, M.L. Simmons, *et al.*, Editors. 1996, IEEE Computer Society Press. p. 207-230.

44. M. E. Crovella and T. J. LeBlanc, "Performance Debugging Using Parallel Performance Predicates," *1993 ACM/ONR Workshop on Parallel and Distributed Debugging*. May 17-18, 1993, San Diego, CA, pp. 140-150.

45. J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," *7th ACM International Conf. on Supercomputing*. July 1993, Tokyo, pp. 185-194.

46. H. Eom and J. K. Hollingsworth, "LBF: A Performance Metric for Program Reorganization," *The International Conference on Distributed Computing Systems*.

47. D. H. Bailey, E. Barszcz, J. T. Barton, and D. S. Browning, "The NAS Parallel Benchmarks," *International Journal of Supercomputer Applications*, **5**(3), 1991, pp. 63-73.

48. J. C. Mogul, R. F. Rashid, and M. J. Accetta, "The Packet Filter: An Efficient Mechanism for User-level Network Code," *11th SOSP*. November 1987, Austin, Texas, pp. 39-51.

49. S. Waldbusser, "Remote Network Monitoring Management Information Base," *RFC 1757*. February 1995, IETF.

50. S. Keshav, "A Control-Theoretic Approach ot Flow Control," *Proceedings of SIGCOMM'91*. September 1991, Zurich, Switzerland, pp. 3-15.

51. R. L. Carter and M. E. Crovella, "Measuring Bottleneck Link Speed in Packet-Switched Networks," *Proceedings of Performance '96*. Oct. 1996, Lausanne, Switzerland,.vol.27-28, pp. 297-318.

52. R. Wolski, "Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service," *High Performance Distributed Computing (HPDC)*. August 1997, Portland, Oregon, IEEE Press, pp. 316-325.

53. R. Wolski, *Dynamic Forecasting Netwoprk Performance Using the Network Weather Service*, TR-CS96-494, UCSD, May 1997.

54. T. Mathisen, "Pentium Secrets," *Byte*, **19**(7), 1994, pp. 191-192.

55. *DECchip 21064 and DECchip21064A Alpha AXP Microprocessors - Hardware Reference Manual*, EC-Q9ZUA-TE, DEC, June 1994.

56. M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Supercomputing*. Nov. 1996, Pittsburg, PA.

57. *The Sparc Architecture Manual, Version 9*. 1994: SPARC International, Inc.

58. W. C. Brantley, K. P. McAuliffe, and T. A. Ngo, *RP3 Performance Monitoring Hardware*, in *Instrumentation for Future Parallel Computer Systems*, M. Simmons, R. Koskela, and I. Bucker, Editors. 1989, Addison-Wesley. p. 35-47.

59. I. Foster and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *Intl J. Supercomputer Applications*, **11**(2), 1997, pp. 115-128.

60. M. J. Lewis and A. Grimshaw, "The Core Legion Object Model," *Proceedings of the 5th International Symposium on High Performance Distributed Computing*.

61. B. C. Neuman and S. Rao, "Resource Management for Distributed Parallel Systems," *the 2nd Symposium on High Performance Distributed Computing*. July, pp. 316-323.

62. F. Berman and R. Wolski, "Scheduling from the Perspective of the Application," *the 15th Symposium on High Performance Distributed Computing*.

63. J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, *Dome: Parallel programming in a heterogeneous multi-user environment*, CMU-CS-95-137, Carnegie Mellon University, March 1995.

64. 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*.

65. 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*.

66. 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.

67. W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavurupu, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," *Frontiers '95*. Feb 6-9, 1995, McLean, VA, IEEE Press, pp. 422-429.

68. 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.

69. 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.

70. M. Litzkow, M. Livny, and M. Mutka, "Condor - A Hunter of Idle Workstations," *International Conference on Distributed Computing Systems*. June 1988, pp. 104-111.

71. IBM, *IBM LoadLeveler: General Information.*, Kingston, NY, September.

72. F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software-Practice and Experience*, **21**(8), 1991, pp. 757-85.

73. T. Green and J. Snyder, *DQS, A Distributed Queueing System*, Florida State University, March.

74. D. Duke, T. Green, and J. Pasko, *Research Toward a Heterogeneous Networked Computing Cluster: The Distributed Queueing System Version 3.0*, Florida State University, May.

75. S. Zhou, X. Zheng, J. Wang, and P. Delisle, "Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems," *SPE*, **23**(12), 1993, pp. 1305-1336.

76. R. B. Dannenberg and P. G. Hibbard, "A Butler Process for Resource Sharing on Spice Machines," *ACM Transactions on Office Information Systems*, **3**(3), , pp. 234-52.

77. J. H. Morris, M. Satyanarayanan, M. H. Cnner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications on ACM*, **29**(3), 1986, pp. 184-201.

78. M. M. Theimer and K. A. Lantz, "Finding idle machines in a workstation-based distributed system," *8th International Conference on Distributed Computing Systems*. June 1988, San Jose, CA, pp. 13-17.

79. S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg, "On Optimal Strategies for Cycle-Stealing in Networks of Workstations," *IEEE Transactions on Computers*, **46**(5), 1997, pp. 545-557.

80. J. Basney and M. Livny, *Improving Goodput by Co-scheduling CPU and Network Capacity*, http://www.cs.wisc.edu/condor/doc/goodput.ps, Dept of Computer Science, University of Wisconsin, .

81. R. H. Arpaci, A. C. Dusseau, A. M. Vahdat, L. T. Liu, T. E. Anderson, and D. A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," *SIGMETRICS*. May 1995, Ottawa, pp. 267-278.

82. D. Petrou, D. P. Ghormley, and T. E. Anderson, *Predictive State Restoration in Desktop Workstation Clusters*, CSD-96-921, University of California, November 5.

83. A. Barak, O. Laden, and Y. Yarom, "The NOW Mosix and its Preemptive Process Migration Scheme," *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments*, **7**(2), 1995, pp. 5-11.

84. M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," *SOSP*. 1983, pp. 110-119.

85. E. R. Zayas, "Attacking the Process Migration Bottleneck," *SOSP*. 1987, pp. 13-24.

86. G. Thiel, "Locus Operating System, A Transparent System," *Computer Communications*, **14**(6), 1991, pp. 336-346.

87. M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Premptable Remote Execution Facilities for the V-System," *SOSP*. Dec. 1985, pp. 2-12.

88. D. G. Feitelson and L. Rudolph, "Parallel Job Scheduling: Issues and Approaches," *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag LNCS,.vol.949.

89. J. K. Ousterhaut, "Scheduling Techniques for Concurrent Systems," *the 3rd International Conf. on Distributed Computing Systems*. October, pp. 22-30.

90. P. Sobalvarro and W. Weihl, "Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Multiprocessors," *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag LNCS,.vol.949.

91. A. C. Dusseau, R. H. Arpaci, and D. E. Culler, "Effective distributed scheduling of parallel workloads," *SIGMETIRCS*. May 1996, Philadelphia, PA, pp. 25-36.

92. A. Tucker and A. Gupta, "Process control and scheduling issues for multiprogrammed shared memory multiprocessors," *12th Symp. Operating Systems Principles*. December 1989, pp. 159-166.

93. P. Messina, "The Concurrent Supercomputing Consortium: year 1," *IEEE Parallel and Distributed Technology*, **1**(1), 1993, pp. 9-16.

94. V. K. Naik, S. K. Setia, and M. S. Squillante, "Schedulint of large scientific applications on the distributed shared memory multiprocessor systems," *the 6th SIAM conf. Parallel Processing for Scientific Computing*. August,.vol.2, pp. 174-178.

95. S. Setia and S. Tripathi, *An Analysis of Several Processor Partitioning Policies for Parallel Computers*, CS-TR-2684, University of Maryland Dept. of Computer Science, May.

96. J. Zahorjan and C. McCann, "Processor scheduling in shared memory multiprocessors," *ACM SIGMETRICS*. May, pp. 214-255.

97. T. D. Nguyen, R. Vaswani, and J. Zahorjan, "Maximizing Speedup through Self-Tuning of Processor Allocation," *Proceedings of IPPS'96*.

98. C. McCann and J. Zahorjan, "Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers," *ACM SIGMETRICS*, Ottawa, Ontario, Canada, pp. 208-219.

99. S. Setia, "The interaction between memory allocation and adaptive partitioning in message-passing multicomputers," *Job Scheduling Strategies for Parallel Processing*, Springer-Verlag LNCS,.vol.949.

100. J. Pruyne and M. Livny, "Interfacing Condor and PVM to harness the cycles of workstation clusters," *Future Generation Computer Systems*, **12**(1), 1996, pp. 67-85.

101. J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, and J. Walpole, "MIST: PVM with Transparent Migration and Checkpointing," *Annual PVM Users' Group Meeting*. May 7-9, 1995, Pittsburgh, PA.

102. N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, "Adaptive parallelism and Piranha," *IEEE Computer*, **28**(1), 1995, pp. 40-59.

103. D. Gelernter, N. Carriero, S. Chandran, and S. Chang, "Parallel programming in Linda," *International Conference on Parallel Processing*. March, pp. 255-263.

104. R. D. Blumofe and P. A. Lisiecki, "Adaptive and reliable parallel computing on networks of workstations," *USENIX Annual Technical Conference*. 6-10 Jan. 1997, Anaheim, CA, USA, USENIX Assoc; Berkeley, CA, USA, pp. 133-47.

105. T. E. Anderson, D. E. Culler, and D. A. Patterson, "A case for NOW (Networks of Workstations)," *IEEE Micro*, **15**(1), 1995, pp. 54-64.

106. R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick., "Cluster I/O with River: Making the Fast Case Common.," *IOPADS '99*. May, Atlanta, Georgia.

107. S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *4.3 BSD UNIX Operating System*. 1988: Addison Wesley.

108. B. Goodheart and J. Cox, *The Magic Garden Explained: The internals of UNIX System V Release 4*. 1993: Prentice-Hall.

109. B. Verghese, A. Gupta, and M. Rosenblum, "Performance Isolation: Sharing and Isolation in Shared-Memory Multiprocessors," *ASPLOS*. Oct. 1998, San Jose, CA, pp. 181-192.

110. R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System," *IBM Systems Journal*, **9**(3), 1970, pp. 199-218.

111. SiliconGraphics, *IRIX 6.4 Technical Brief*, http://www.sgi.com/software/irix6.5/techbrief.pdf, , .