A Cost-Benefit Approach to Resource Allocation in Scalable Metacomputers

by
R. Sean Borgstrom

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy

Baltimore, Maryland
September, 2000

Abstract

A *metacomputer* is a set of machines networked together for increased computational performance. To build an efficient metacomputer, one must assign jobs to the various networked machines intelligently. A poor job assignment strategy can result in heavily unbalanced loads and thrashing machines. This cripples the cluster's computational power. A strong job assignment strategy helps a metacomputer complete all of its jobs swiftly.

Resource heterogeneity makes job assignment more complex. Placing a job on one machine might risk depleting its small memory. Another machine might have more free memory but a heavily burdened CPU. *Bin packing* on memory protects the system against thrashing. *Load balancing* protects the system against high CPU loads. Combining the two approaches, however, gives an *ad hoc* heuristic algorithm with no clear theoretical merit.

The **Cost-Benefit Framework**, developed in this work, offers a new approach to job assignment on metacomputers. It smoothly handles heterogeneous resources by converting them into a unitless cost. We assign (and possibly reassign) jobs to greedily minimize this cost.

This approach gives us an online strategy provably competitive with the optimal offline algorithm in the maximum usage of each resource. It has a weak competitive ratio – logarithmic in the number of machines in the cluster – but even this weak ratio is unprecedented in the literature. No other known method offers any competitive guarantee on more than one resource.

We present experimental evidence that this strategy performs extremely well in practice, comparing it to two important benchmarks: the default round robin strategy of the popular PVM metacomputing system, and the powerful adaptive strategy of the Mosix system.

Metacomputing environments are not homogeneous. In some environments, the scheduler has a great deal of information about submitted jobs. In other cases, it has very little. Some systems can migrate jobs without interrupting their execution. Others cannot. We develop variants of the basic "opportunity cost" strategy of the Cost-Benefit Framework for various metacomputing environments, and prove all of them highly efficient.

Finally, we provide two metacomputing systems – a prototype and a complete system – based on these ideas. The Java Market prototype is a metacomputer built atop Java and web technologies, able to integrate any consenting Internet-connected machine. The Frugal System transforms any Jini network into a metacomputer.

This work was carried out under the supervision of Professor Yair Amir.

Acknowledgements

I am indebted to and enormously appreciative of my advisor, Dr. Yair Amir, for his selfless dedication to the development of my career and this work. His assistance and contributions to every stage of the research process made this dissertation possible and taught me a great deal. Thank you.

My gratitude to Dr. Rao Kosaraju, similarly, knows no bounds. His guidance, support, and direction over the years have been invaluable.

I thank Dr. Baruch Awerbuch for bringing the theoretical basis for this work to my attention and Yair's, inspiring the project, as well as for his theoretical work itself. Dr. Amnon Barak's assistance and advice in evaluating our algorithms and ideas in the context of dynamic systems made a significant portion of this work possible. Dr. Yossi Azar and Arie Keren played important roles in developing the theory for use in a practical context.

I thank Drs. John Cherniavsky and Michel Melkanoff for inspiring me to pursue computer science. Their compassion and their dedication to the field revealed the best in both computer science and humanity.

I appreciate the kindness and good nature shown by all of my associates in the CNDS lab at the Johns Hopkins University. These include Jonathan Stanton, Jacob Green, Cristina Nita-Rotaru, Theo Schlossnagle, Claudiu Danilov, John Schultz, and Tripurari Singh. I regret that I did not have more opportunities to work with them on this project, and have enjoyed the projects that we shared.

My research would not have been possible without the Metacomputing grant from DARPA/ITO (F30602-96-1-0293) to the Johns Hopkins University. Special thanks to Dr. Gary Koob, our DARPA program manager, who had the vision to see that the Cost-Benefit Framework could lead to superior performance in practice.

My parents, Karen and Karl, and my sister, Sonja Britt, have been steadfast supporters through my doctoral studies (and for all my life before that). Thank you.

I am also grateful to Dr. David Allen, Cera Kruger, Trevor Placker, Gretchen Shanrock, Brad Solberg, Geoff Grabowski, Bruce Baugh, Phyllis Rostykus, Cameron and Jessica Banks, Stuart Lin, Margaret, Henry, Conrad Wong, Kit Murphy, Christofer Bertani, Alexander Williams, Chrysoula Tzavelas, Bryant Durrell, Rich Dansky, and Deirdre Brooks, among others, on general principles.

Contents

Chapter One: Introduction

1.1 Overview and Highlights

The work described in this thesis develops, tests, and implements several advanced strategies for resource management in a network of machines. Each machine has a number of resources associated with it, e.g. memory and CPU. If any one resource is overused, the system's performance on some jobs plummets. Therefore, the strategies developed here take multiple resources into account in a reasoned manner. These strategies do not depend on any given model for incoming job flow. Nor do they depend on any particulars regarding the strengths of the various machines in the network.

We measure the performance of these strategies in terms of job *slowdown*. If a job can complete in time t_{best} while running alone on the most appropriate machine in the system, and instead completes in time t_{actual}, the job's slowdown is t_{actual} / t_{best}. The key performance measure in our tests was the average slowdown over all jobs. In other words, we evaluated our strategies and a selection of alternate strategies in terms of how much the system load slowed the average job down.

The three new strategies that this thesis develops for managing a metacomputer have a sound theoretical basis. We demonstrate that each of them improves on existing strategies, first in a simulated environment and then in tests on a real system. In other words, using our strategies reduces the average slowdown over all jobs. We explain how to remove certain assumptions in the strategies that may not apply in real systems while preserving most of the performance gain. We also develop two working metacomputers based on these strategies.

The central idea behind our strategies is to convert all of a system's resources into a homogeneous cost based on their percentile utilization. Historically, memory, CPU, and other resources have been incomparable. One cannot even measure them in the same units – megabytes of memory do not easily convert to seconds of CPU time. By converting both of these resources into a unitless cost, however, we gain the ability to compare them. The cost function used by our strategies is not based on heuristic ideas or experiments, which have a bias towards the system on which they are developed or performed. Rather, it derives from a theoretical framework with certain competitive guarantees. These guarantees apply on any imaginable system and any possible stream of jobs.

The key contributions of this Ph.D. research are:

- Developing a basic strategy for resource management on a metacomputer with a solid theoretical basis and proven experimental validity.
- Developing variations on this strategy for metacomputers where reassignments are possible.
- Developing an approximation technique that allows systems to use this strategy even when they cannot provide the scheduler with the information it needs about incoming jobs.

- Demonstrating, with experiments in simulated and real environments, that these algorithms are effective in comparison to optimized heuristic methods.
- Developing a primitive metacomputer based on web technology as a test bed for these strategies.
- Developing a complete and advanced metacomputer based on Sun's Jini system that successfully implements several of these strategies.

1.2 Basic Concepts

The performance of any cluster of workstations improves when the system uses its resources wisely. In particular, when the system can assign incoming jobs to any machine in the cluster, and when it employs an intelligent strategy for doing so, it can keep system loads low and prevent memory overutilization and thrashing. Developing strong strategies for job assignment provides a direct payoff in terms of system efficiency.

System resources are heterogeneous. For example, the CPU resource completes a certain number of computation cycles per second. It divides these cycles between its various jobs. A computational task completes when it has received an appropriate amount of computation. The memory resource, on the other hand, contains a certain number of megabytes. The system does not divide available memory evenly among tasks. Rather, each task determines its own memory requirements. When the memory becomes overutilized, the machine begins to thrash and the system's performance degrades.

One cannot balance CPU cycles against megabytes or vice versa. Nor does either resource convert naturally into communication bandwidth, the unit of measurement for communication resources. In general, no two machine resources are directly comparable. This makes finding the optimal location for a job difficult. Even implementing a greedy strategy, balancing all the system resources across all the machines, is hard. With incomparable resources, this strategy is not well defined.

This work's new *opportunity cost* approach, based on economic principles and competitive analysis, provides a unified algorithmic framework for the allocation of computation, memory, communication, and I/O resources. By converting heterogeneous resources to a homogeneous cost, and then minimizing this cost, the system guarantees near-optimal end-to-end performance on every single instance of job generation and resource availability.

The algorithm derived from this approach is an online algorithm that knows nothing about the future, assumes no correlation between past and future, and is only aware of the current system state. Nevertheless, one can rigorously prove that its performance is always comparable to that of the optimal prescient strategy. Unlike other methods, this algorithm achieves this performance bound for every resource simultaneously.

Some systems have requirements, or abilities, which suggest or require adjustments to the basic algorithm presented herein. Job migration, in which jobs move transparently from one machine to another, offers the system a powerful additional tool for improving performance. Our algorithm adapts easily to a *dynamic* environment that allows job migration. Since the system cannot predict the arrival rate and resource demands of incoming jobs, any online scheduler must inevitably make "mistakes," placing jobs on non-optimal machines. Job migration gives the system an unlimited number of opportunities to fix the mistakes it makes. The Mosix [Mos, BL98] system allows this kind of transparent job migration, assigning and reassigning jobs using the heuristically optimized Mosix strategy. Adapting our algorithm to perform job reassignments as well as job assignments yields a large improvement over the single-assignment approach, because it can fix mistakes. More importantly, however, our adapted algorithm yields a significant improvement over the highly tuned Mosix strategy. Even on a powerful system, able to reassign jobs, a carefully chosen strategy improves system performance. Avoiding a mistake does the system more good than fixing it later.

A system might have less information about jobs than a competitive strategy requires. For example, if the system cannot approximate a job's resource requirements upon its arrival, its worst-case competitive ratio for maximum resource use depends on the disparity between the power of the machines as well as the number of machines. In practice, this work demonstrates that an adapted form of our algorithm can achieve extremely good results with limited information.

1.3 Experimental Methodology

The *opportunity cost* approach offers good performance in practice. We evaluated opportunity cost-based algorithms against a number of other algorithms, choosing two alternatives as meaningful benchmarks. The first benchmark is PVM's default strategy [PVM]. This is the simple round robin approach to job assignment used in PVM, a dominant utility for job assignment and load balancing in computing clusters. The second benchmark is the Mosix strategy. The Mosix strategy is the highly optimized heuristic strategy used by Mosix, a system permitting dynamic job reassignment. Results for the Mosix strategy always assume that the strategy is permitted to reassign jobs, even when we compare it against a strategy that cannot.

Our primary test bed was a Java simulation of a computing cluster and a stream of incoming jobs. Each execution of the simulation ran for at least 10,000 simulated seconds. The simulated cluster had six machines with a realistic distribution of computing power. We based the job flow on a study of the distribution of real jobs. We compared all strategies against the PVM and Mosix strategies in at least 3,000 scenarios.

To validate the simulation, we performed a second series of tests on a real system. This system consisted of a collection of Pentium 133, Pentium Pro 200 and Pentium II machines with different memory capacity, connected by Fast Ethernet, running BSD/OS [4]. The physical cluster and the simulated cluster were slightly different, but the proportional performance of the various strategies was very close to that of the Java simulation. This indicates that the simulation appropriately reflects events on a real system.

Finally, to demonstrate that our strategies were applicable in the real world, we built a complete metacomputer based on these strategies.

1.4 Thesis Structure

The rest of this thesis is organized as follows.

Section 1.5: Related Work describes relevant previous work in metacomputing.

Chapter Two: Summary of the Theory discusses the basic theoretical underpinnings of this work.

Chapter Three: Static Strategies presents our strategy in the context of a static cluster of workstations. Static systems cannot reassign jobs. The key idea in this chapter is our cost function, which measures the cost to the system for using up any portion of any kind of heterogeneous resource. Our strategy assigns jobs to machines in order to minimize the global cost. The experimental results contained in this chapter demonstrate that this strategy performs extremely well in practice. It dramatically outperforms the PVM strategy.

Chapter Four: Dynamic Strategies presents a variant of our strategy in the context of a dynamic cluster of workstations. Dynamic systems *can* reassign jobs. Our algorithm makes both assignments and reassignments to greedily minimize our cost function. Again, experimental results demonstrate that our strategy performs extremely well in practice, outperforming the highly tuned heuristic strategy used by Mosix. This chapter further compares our static strategy, which makes a single intelligent assignment, with the Mosix strategy and its ability to perform reassignments. Our static strategy performs well in this regard, indicating that using our user-level opportunity cost approach is a viable alternative to implementing the Mosix kernel modifications.

Chapter Five: Strategies with Reduced Information presents a variant of our strategy that does not know the resource requirements of jobs upon their arrival. Experimental results show that this strategy variant performs well despite the adverse conditions under which it operates.

Chapter Six: The Java Market discusses our first metacomputer prototype built around these strategies. This work was successful but ultimately impractical due to factors outside our control (see **Section 6.2.1: Features of Web-Based Metacomputers**.) Our desire to build a more practical system led directly to the work in Chapter Seven.

Chapter Seven: The Frugal System discusses the Frugal System package for Jini networks. This package transforms any Jini network into a fully functional metacomputer, using our strategies as its decision algorithm.

Chapter Eight: Conclusions summarizes this research.

1.5 Related Work

1.5.1 Shadow Objects

Metacomputing systems contain a number of objects. These include system resources, worker processes that perform jobs, work objects encapsulating jobs that the system must perform, and data objects representing job input and output. Metacomputers obfuscate but cannot presently erase the distinction between local and remote objects. Many jobs have logical relationships with a fixed machine. They are tied, in one manner or another, to that machine's file systems, input and output channels, and physical computing resources. In addition, implementing communication between the elements of a metacomputing system often depends strongly on the underlying hardware and operating system.

To make metacomputing feasible, many systems implement "shadow objects," which serve as proxies for a remote object. Condor's shadow daemons [Con] and Mosix's deputies [Mos] act as proxy representatives for system resources. When a job uses Condor or Mosix's migration facilities to move to a remote machine, it performs most system calls through a shadow of its originating system. Bond [BHJM99] uses shadows of remote code objects; in Bond, they facilitate communication between components of a distributed computing system.

1.5.2 Matchmaking

System resources are heterogeneous. This thesis assumes that the difference between machines is purely a performance issue. For example, one CPU might accomplish jobs twice as quickly as another. One memory resource might begin thrashing at 32 Megs when another would be scarcely an eighth full. In some contexts, however, it is more important to realize that certain machines simply cannot run certain jobs. Others may be inherently undesirable matches, regardless of system state.

The common solution to this problem is *matchmaking*. System resources advertise their properties. Clients advertise their needs and preferences. Some component of the distributed system then matches resources to clients. In Condor [Con], the centralized scheduler performs matchmaking. In Legion [Leg], object wrappers for system resources push their properties into a "Collection" object, which clients can search for appropriate resources. In Bond [BHJM99], object wrappers for system resources – metaobjects – and properties of the clients serve as a "lock and key" system. This system ensures an appropriate match.

1.5.3 Java, Jini, and Web Technology

The Java [Java] programming language uses the idea of a "virtual machine" to overcome the architecture-specific nature of traditional machine code. Programmers write Java programs to run on the architecture of the Java virtual machine. Java interpreters, written for many real systems, translate this program into instructions for their specific machine. Assuming the correctness of the Java interpreters, a program that runs correctly on any machine will run correctly on every machine. In combination with Java's widespread popularity and acceptance, this makes Java a very useful metacomputing tool. Many of the problems of machine heterogeneity disappear when only a handful of programs (Java interpreters, operating systems, and specialized native code optimized for high performance) depend on the specific machine whereon they run. Projects using Java include ATLAS [BBB96] and JPVM [Fer98].

Java is integrated with the World Wide Web, a technology with a huge population of users. Through an interface these users understand and use regularly, the web browser, Java code can move from machine to machine. This makes web technology another very appealing metacomputing tool. A number of projects [CCINSW97, SPS97, AAB00, Kar98, BKKK97, RN98, Sar98], including our own Java Market, take advantage of web and Java technology to seamlessly distribute code across wide-area networks. The Java "sandbox" gives these research efforts automatic security against malicious code.

Jini [Jin] builds on Java in a different direction. The Java 1.1 API gave Java programs the ability to execute the methods of Java objects from remote machines. Jini builds a lookup service and a suite of tools for distributed computing on top of this technology. Modular services can interact with one another through the lookup service and using these tools. This simplifies the construction of network-aware applications and metacomputing systems.

1.5.4 MPI

MPI [MPI] is a popular architecture-independent parallel programming system. This "Message-Passing Interface" defines a system for building parallel programs whose components interact by exchanging messages. MPI "Communicator" objects provide a scalable context for message exchange and a scope for group communication. MPI programmers can enhance these objects to build virtual network topologies for the processes participating in a given group – conceptually organizing them in a grid, tree, or other graph.

Many software packages have implemented the MPI standard for many different platforms. Several of these platforms are metacomputers as well as parallel programming environments (e.g. Hector [RRFH96] and Globus [IK97].) Its various implementations in Java [Java, MG97] also offer possibilities for wide-area network metacomputing.

1.5.5 PVM: Parallel Virtual Machine

PVM [PVM] is a metacomputing environment designed for use on a cluster of workstations. PVM automatically distributes jobs and their subtasks to appropriate machines according to a policy instantiated in a Resource Manager. This allows static load balancing using algorithms of arbitrary complexity.

PVM is designed to support a variety of distributed computing paradigms. This includes programs that make a fixed division of their workload among a fixed number of processors, adaptive programs that split their workload between multiple machines as needed, programs divided into multiple objects that interact with one another, and programs that divide up their input but not their functionality.

Most of PVM's functionality resides within a set of libraries that programmers can use to parallelize and distribute their jobs. Through these libraries, PVM provides process control, allowing tasks (and subtasks) to discover their unique identifier, spawn subtasks, and stop execution. It provides informational utilities that tasks can use to evaluate the system state. It permits restructuring of the metacomputer, adding hardware to or removing it from the collection of machines the metacomputer uses for tasks. It allows signaling and message passing between tasks, as well as an advanced group-based communication paradigm. PVM also provides a console that users can use to monitor tasks.

PVM daemons running on each host enable this functionality. Each daemon is equipped to start up tasks that the Resource Manager forwards to it, as well as passing messages on behalf of those tasks. Although a PVM daemon must run on each machine for PVM to function, programs can start PVM daemons automatically on authorized hosts.

Since version 3.3, PVM has used a scheduler known as the Resource Manager to make resource allocation decisions. This Resource Manager is just another PVM task, so it can theoretically implement any policy. The default policy for PVM is a straightforward round robin assignment to available machines.

PVM sets a metacomputing standard, with stable code and a large user base. Its simplicity and power appeal to many programmers. Programmers have created variants of PVM designed to work with Java, perl, tcl/tk, Matlab, python, and C++. We often compare our decision-making strategies against the default round robin strategy of PVM. This shows the resource gain from implementing a policy like ours in a standard metacomputing system like PVM.

1.5.6 Mosix

The Mosix [Mos] system offers a flexible approach to metacomputing. It can adapt to changing resource conditions on the cluster by transparently migrating jobs from one machine to another. Where the decisions of PVM's Resource Manager are fixed and cannot be changed, Mosix dynamically responds to load conditions by reassigning jobs to new machines. Combined with its carefully optimized algorithm for making load decisions, Mosix's transparent process migration facility almost invariably outperforms any online static strategy.

Mosix can move jobs from one machine to another without logically interrupting their execution. The Mosix kernel modifications to the Linux operating system allow the system to perform this transparent migration on ordinary Linux jobs. They also offer additional migration-related tools for programmers developing programs with Mosix in mind. Previous incarnations of Mosix have performed the same function for other Unix-like operating systems.

In this respect – the ability to migrate ordinary jobs – Mosix stands out. Most metacomputing systems require programmers to specifically design programs with the metacomputing system in mind. Legacy code must be adapted to the metacomputer. In Mosix, however, there is no concept of legacy code. Programs written for Linux become Mosix programs automatically.

Mosix is designed to preserve a job's state as it moves from machine to machine. To make this possible, Mosix redirects many system calls sent by a job to the machine where the job originated. There, a shadow object called the job's deputy performs the system call and returns the result. Thus, for example, jobs can read and write files only accessible from their home machine, even when migrated to remote hosts. Current research efforts seek to minimize the number of calls that must go through this deputy. For example, the Mosix team is alpha testing a revision of Mosix that allows local I/O. If the machine a job runs on shares a file system with the machine of its origin, this revision allows the job to access files directly. The Mosix team is also developing migratable sockets that allow migrated processes to communicate without going through their deputies.

Mosix uses a very scalable algorithm for job reassignment decisions. Machines do not attempt to discover the overall system state. Rather, at regular intervals, each machine contacts a random subset of the other machines. If it detects a load imbalance, and the higher load is above a certain level, it reassigns jobs to even the load ratio. Further, like our decision-making strategies, Mosix also attempts to minimize the incidence of thrashing. When a machine starts paging due to a shortage of free memory, a "memory ushering" algorithm takes over the reassignment decisions, overriding the load balancing strategy and migrating a job to a node that has sufficient spare memory.

Mosix is complete, functional, and stable. It has seen many years of use as a production system. We have chosen Mosix as our second major benchmark. In a context where our strategy can migrate jobs, Mosix's optimized if heuristic scheduler serves as fair competition against ours. In a context where our strategy *cannot* migrate jobs, Mosix's performance offers a rough upper limit for what an ideal strategy could achieve.

1.5.6.1 Similar Projects

Some projects similar to Mosix include MIST [APMOW97] and Hector [RRFH96]. MIST enhances PVM with transparent job migration capabilities. This uses a more exact but less scalable load balancing algorithm than Mosix. Hector is an MPI system capable of transparent job migration, making decisions through a central scheduler.

1.5.7 SNIPE

The SNIPE [FDG97] project seeks to build an advanced metacomputer atop PVM technology. They hope to increase PVM's scalability and security while adding a global name space, process migration, and checkpointing. They also plan to eliminate PVM's dependence on a single centralized scheduler.

At the heart of the SNIPE project's strategy lies the RCDS – the Resource Cataloging and Distribution System. This is a global fault-tolerant data space that contains replicated resources and metadata regarding those resources. The information in this data space is available via a Uniform Resource Name system. SNIPE stores processes, hosts, and data in the RCDS. If an error in communication during process or data migration results in some process losing track of a collaborator or its data, it rediscovers the necessary information using the relevant URN. As part of SNIPE's general replicated approach to fault tolerance, the Resource Manager itself is replicated to multiple hosts.

SNIPE hosts can execute mobile code inside "playgrounds," which, like Java sandboxes, limit the damage that code can do to the local system. This provides tunable security when running a SNIPE system over the entire Internet.

1.5.8 Condor

Condor [Con, LLM88] is a metacomputing system designed to harvest the otherwise unused CPU cycles on an Intranet. Condor is a mature system. Its first version appeared in the 1980s. It has demonstrated its efficiency on clusters of hundreds of machines. Since Condor uses a single central scheduler, its success demonstrates that such schedulers can scale to very large clusters.

Condor provides a set of libraries that ordinary jobs can link to. By linking to the Condor libraries rather than the standard C libraries, jobs gain the ability to checkpoint themselves and perform remote system calls.

When a job checkpoints itself, it converts its entire state into a "checkpoint file." If the job crashes, it can use this file to restart itself at the last checkpoint. If the system needs to transfer the job to a new machine, it can checkpoint the job and use this file to recreate the job's state on that machine. The job can logically continue execution uninterrupted since no state information has been lost.

The process of job migration is time-independent – jobs can exist in the form of checkpoint files indefinitely. Users of a Condor system define criteria that establish their machine as idle. Condor checkpoints remote jobs when the system they run on becomes unidle. It places jobs on a machine when that machine becomes idle.

Condor is designed for use with distributively owned machines; that is, machines with a variety of different owners, many of which may not share file systems. Since it does not assume a common file system, Condor must use a shadow object to represent the system a given job originated from. Jobs perform most system calls through this shadow, using the resources of their originating machine. The only system resources they access on the remote machine where they actually run are CPU and memory.

Jobs in Condor come with a list of "requirements" and "preferences" in terms of the hardware the system should execute them on. Condor does not migrate jobs to machines that do not meet their requirements. It also attempts to choose machines that meet the job's preferences.

Condor is a system of proven value. Installing Condor on a cluster of relatively homogeneous machines can improve computational throughput by one or two orders of magnitude.

1.5.9 Legion

Traditional operating systems mitigate the complexity of computer hardware and low-level software by providing an abstract representation of these resources that functions similarly or identically regardless of the underlying implementation. The high-level components of distributed systems now possess even more complexity and interface diversity than the low-level elements of individual systems. The Legion [GWLt97, Leg] project argues that distributed systems require a second-order operating system to simplify the management and interaction of diverse high-level components. Legion is one such “wide area operating system.”

Legion takes an object-oriented approach to distributed system management. Each system resource has an object encapsulating it, providing a uniform interface by which Legion programmers can manipulate that resource. CPU resource objects (“host objects”) support various features such as reservations. By extending a CPU resource object, a host can enforce a desired policy for the use of its resources. Legion uses matchmaking to find the correct host object for a given code object.

One of Legion’s interesting properties is the full persistence of its objects. As in Condor [Con], the system can checkpoint objects, storing them in “vault objects” (persistent storage). This permits Legion to deactivate objects and then activate them later, possibly on different hosts. A scheduler running on a Legion system could use a migration-based strategy like E-Mosix (see **Chapter Four: Dynamic Strategies.**)

Legion’s control over code objects relies on “Class Manager” objects, each of which governs a number of Legion objects. Using Class Manager objects, Legion can perform task monitoring and task management. Class Manager objects also implement more specific program controls. Legion can assemble Class Managers into hierarchies, allowing natural scalability.

Legion provides a global namespace for objects. Each object has a unique identifier that encapsulates its public key. An object’s identifier is therefore sufficient information to establish secure communication with that object. Objects can also have Unix-style string pathnames, built atop the underlying identifier structure.

Legion has been fully implemented and is running on a number of sites. Distributed operating systems such as Legion offer an excellent interface to networks of system resources, and accordingly make rich environments for the deployment of intelligent decision-making strategies.

1.5.10 Globus

The Globus project [Glo, IK97] seeks to create a toolkit for building metacomputers. Using the Globus toolkit simplifies the design of metacomputing services and higher-order metacomputing systems. On the whole, Globus focuses on more flexible but lower-level mechanisms than does Legion.

Globus identifies the major issues facing a metacomputing system as:

- The need for *scalability* – matching jobs to resources must remain feasible even in environments with many machines;
- The problem of *heterogeneity* – computing resources and network resources vary widely;
- The problem of *unpredictability* – large-scale networks do not remain static, either in the short term or in the long term; and
- *Distributed ownership* – the components of a metacomputer can have many different owners. These components can have different individual access policies, and can relate to one another in different ways.

Globus includes several components. Globus' communications toolkit provides an abstraction for communications and network behavior. Developers can build higher-level communications models atop this abstraction. For example, metacomputing systems can build a message passing architecture or a remote method invocation structure atop Globus' communications component.

In Globus, a metacomputing directory service (MDS) fills some of the roles of Legion's global object namespace and SNIPE's RCDS. This service lists pertinent information about a service or resource as a series of attribute-value pairs. (These entries also resemble those used in Jini [Jin].)

Globus provides an authentication model that unifies various security models into a single abstraction. It also includes a remote I/O interface for file operations. Together with the communications toolkit and the MDS, these services provide a basic abstraction for a virtual metacomputer composed, in actuality, of many diverse machines.

On top of this basic abstraction, the Globus team also seeks to develop "middleware," such as parallel programming tools and a high-level security architecture. The Globus team has ported MPI [MPI], Compositional C++ [CK93], and other parallel environments to Globus.

Full implementations of Globus are in use on a number of sites, as with Legion. If Globus becomes the standard bottom layer for metacomputing applications, it would be worthwhile to port this work's decision-making strategies to that environment.

1.5.11 HARNESS

The Harness [Har] project, from the creators of PVM, has not yet completed its prototype. Nevertheless, some of the ideas developed for this system are worth mentioning.

Many metacomputing projects simulate a single virtual parallel machine on which users run their jobs. Harness abandons this paradigm, proposing a new kind of structure, wherein the network decomposes into an arbitrary number of virtual machines. These virtual machines can join together to perform large tasks or split apart to adopt an agent model.

Most of Harness' functionality is not native, but rather comes in the form of *plug-ins*. Like browser plug-ins, Harness loads these plug-ins dynamically to provide the extended or adapted functionality that currently running programs need or expect. For example, a metacomputer could implement a particular shared-memory paradigm using a plug-in. It could also turn on intelligent resource allocation by replacing the resource allocation module with a new plug-in component.

Harness' default plug-ins for communications, resource management, and process control will be designed with the uncertain character of distributed computing networks in mind. The system will tolerate machine failures and network partitions. Finally, it will offer group communication semantics to add expressive power to the process management and communications modules.

1.5.12 Javelin

The Javelin [CCINSW97] project, like our Java Market, builds a metacomputer atop web browser and Java technologies. Producer machines, offering computational services to the Javelin system, need only connect to a web page. Javelin can then automatically download work, in the form of Java applets, to their system. This offers a simple interface for producers.

In Javelin, the Java sandbox protects the security of producer machines. It also limits the capabilities of code objects running on those machines. In particular, they cannot write to disk or communicate with machines other than the server that forwarded the code object to that particular producer machine. To overcome these problems, the Javelin project provides libraries that mimic the native Java communication package in interface but (behind the scenes) forward network messages through the server. The results returned by the code, similarly, are stored on the server. While this can produce heavy network loads at the server, it permits applications programmers to create arbitrarily complex parallel applications using Javelin.

Javelin has a native distributed load balancing strategy. The system has three components:

- Clients – consumers of computational resources;
- Hosts – producers of computational resources; and
- Brokers – matchmakers who forward work to hosts.

Client applications use the Javelin libraries to push units of work onto a queue of tasks that the system must accomplish. Brokers manage a pool of hosts and a task queue. If the broker's pool of hosts cannot perform the work in its queue, it randomly forwards a request through the broker network to find a host that can. If its hosts are idle and its task queue empty, it opportunistically steals work from other brokers.

Despite the limitations of Java and web browser technologies, Javelin demonstrates good performance in practice, achieving speedup factors of at least 75% the number of processors used. The possible benefits of adapting their framework to use an opportunity cost strategy are unknown.

1.5.12.1 Similar Projects

Some projects that resemble Javelin include JET [SPS97], ATLAS [BBB96], and NetSolve [CPBD00].

JET focuses on web-based metacomputing. JET's tasks decompose into a number of stateless worker processes that communicate with a master process.

ATLAS uses Java (but not web technology) as the backbone for a work-stealing metacomputer. ATLAS' work-stealing algorithm is particularly interesting in that machines steal work first from other machines in their local cluster, then from sibling clusters, then from siblings of their parents' cluster, and so forth. This gives participants preferential use of the machines they contribute.

NetSolve uses a client, server, and broker model (much like Javelin's) to simplify the world of high-performance scientific applications. This division allows client applications to focus on specifying the problem they wish to solve. Server applications can focus on high performance problem solving. The brokers take full responsibility for resource management. NetSolve can use a Condor system as a "server" to provide checkpointing and migration facilities.

1.5.13 MILAN

MILAN [Mil] is a significant metacomputing initiative. At present, it consists of three major projects: Charlotte, Calypso, and KnittingFactory.

Charlotte [Kar98] is a programming environment for parallel computing built on top of the Java [Java] programming language. Programs using Charlotte run on a virtual parallel machine with infinitely many processors and a common namespace. Charlotte provides the translation between this virtual machine and the heterogeneous, failure-prone, ever-changing collection of web-connected machines that serve as the program's real underlying hardware.

Charlotte programs alternate between serial and parallel steps. On parallel steps, multiple routines can execute simultaneously. Read operations during a parallel step return the object as instantiated when parallel step began. Write operations become visible at the end of the parallel step, and must write the same value. This yields Concurrent Read, Concurrent Write Common (CRCW-Common) semantics.

Charlotte's shared memory exists in user space, and is not dependent on the underlying operating system. Shared data objects maintain their state:

- *not_valid* – the object is not instantiated locally;
- *readable* – the object is instantiated locally; or
- *dirty* – the object has been modified locally during this parallel step.

Charlotte updates data using a *two-phase idempotent execution strategy*. Manager objects buffer the changed data associated with a routine's execution and then update data at the end of a parallel step. This ensures that any number of executions of a code segment is equivalent to executing it exactly once.

Charlotte's *eager scheduling* uses this exactly-once semantics to enable efficient load balancing and fault tolerance. Any worker can download and execute a parallel routine that the program needs to execute and has not yet completed, whether or not some other worker is already performing that computation. In consequence, the system need not distinguish between very slow machines and crashed machines. This provides intrinsic fault tolerance.

Calypso [Caly] provides a C/C++ programming environment for metacomputing on a network of workstations. Its basic purpose and techniques resemble those of Charlotte: it uses eager scheduling and a two-phase idempotent execution strategy to realize a virtual parallel machine independent of the failures and performance abilities of the actual machines in the network.

KnittingFactory [BKKK97] is an infrastructure for collaborative systems over the web. Like Javelin and our Java Market, it relies upon Java and browser technology to provide security and simplicity. Unlike these projects, however, KnittingFactory uses Java Remote Method Invocation to circumvent specific portions of the Java sandbox. Applets that use KnittingFactory can communicate directly with one another, rather than routing their requests through the server through which they came. This yields a scalable solution for web-based metacomputing, directly relevant to future efforts in the Java Market mold.

1.5.14 Popcorn

The Popcorn [RN98] project, developed at the Hebrew University of Israel, sought to create an Internet-wide market for computing services. The goal was to realize the computational potential of the Internet by converting machine cycles into abstract currency ("Popcoins.") Contributors obtained Popcoins by leasing their machine to the system, and then spent Popcoins to buy cycles on remote machines. As with Javelin and the Java Market, Popcorn built on web browser and Java technologies.

Popcorn's computational model required that programmers develop their programs for use with Popcorn, breaking them down into a series of "computelets" that can run independently. This paradigm permits the use of extensive parallelism, as long as the individual parallel chunks perform enough work to overcome the communication overhead.

The core of Popcorn was the decision-making algorithms that the "markets" implemented for matching computelets to producers of computational resources. Popcorn offered three different algorithms that producers and consumers could employ, drawing on economic theory.

The first decision algorithm used by Popcorn is the "repeated Vickrey auction." In a Vickrey auction, each computational resource is auctioned to the highest bidder, but that bidder pays the *second*-highest price. This removes the incentive for buyers to underbid. The dominant strategy is to bid exactly what a resource is worth. The motivation to hack one's Popcorn code to alter its bidding strategy disappears. A *repeated* Vickrey auction sells multiple resources in separate auctions. This undermines the Vickrey auction's guarantee. It potentially allows bidders to strategize based on their knowledge that some combinations of resources are more valuable to other consumers than others. However, practical strategies for abusing Popcorn's repeated Vickrey auction do not currently exist.

Popcorn's second approach was conceptually simpler but more strategically complex. Each bidder gave their minimum price and their maximum price, as well as a rate of change. The seller did the same. Bids continued until a buyer's price met or exceeded a seller's price.

Finally, Popcorn could compute market equilibria from the current demand and supply curves. The market then performed computelet/resource matchmaking based on these equilibria.

As the Popcorn project shows, many ideas from economics can prove valuable for managing decisions and collections of agents in the metacomputing world. The project's tests show that computing market equilibria is particularly useful in providing a high level of service to both buyers and sellers of computational resources.

1.5.14.1 Similar Projects

Nimrod/G [ABG00] also uses an economic paradigm to make scheduling decisions. It uses a strategy superficially similar to Popcorn's market and our Cost-Benefit Framework. Resources assign themselves costs. Clients who wish to perform computation provide a benefit function stating how much they will pay for work completing within a given length of time.

Nimrod/G's computational economy is not fully developed at this time. It lacks a comprehensive strategy for assigning resource costs and does not have sophisticated negotiation algorithms like Popcorn's. However, this is a subject of current research.

Nimrod/G is a narrower but more powerful system than Popcorn, designed for quickly carrying out parameterized simulations on metacomputers. It is built atop the Globus [IK97] toolkit. It does not take advantage of Java or web technology, which limits its potential cluster size but improves its efficiency.

1.5.15 IceT

IceT [GS99] offers a metacomputing paradigm with several unusual features. These include dynamically merging and splitting virtual machines, as in Harness, and the use of architecture-independent native code.

The inspiration for the IceT project is the observation that data, code, and virtual machines, as purely abstract concepts, need not be tied to any given physical machine. IceT provides a message-passing architecture for the transfer of data, an "agent" structure for mobile code, and a system whereby researchers wishing to collaborate can merge their virtual machines. This gives increased computational efficiency and makes shared access to various computations possible.

IceT is designed to execute programs on any machine architecture. Its scheduler can place a given job on various dramatically different machines. Naturally, automatically porting native code across architectures is outside the scope of modern systems; what

IceT *can* do is find the version of a given program compiled for the architecture where it wishes to place the job. Executing a matrix multiplication subprogram on a Linux box, it can automatically locate the Linux source for that program.

IceT, unlike such projects as Mosix and like such projects as Legion, requires that the programmer build their programs within the IceT paradigm.

1.5.16 InfoSpheres

InfoSpheres [Inf] is an exciting project from Caltech. Like Legion, it seeks to build a virtual infrastructure of distributed objects atop the real network. What distinguishes the two projects is not so much the infrastructure model as the paradigm used in its construction. In InfoSpheres, projects and individuals have a loose collection of objects that, in some sense, serves as their network identity. Objects are not anonymous pieces of code owned by various users, but an "InfoSphere" representation of the users themselves. This purely conceptual distinction influences the low-level decisions made by the InfoSpheres team on a number of levels.

The InfoSpheres project seeks to develop a highly composable architecture, where the global behavior of the system emerges from standard interactions between objects. Initiating a collaborative task, such as project development by human users or automated calendar management, requires that the system quickly assemble a virtual network of objects from the relevant InfoSpheres. The project identifies the major problems facing such a system as follows:

- The system must adapt rapidly when the components of the virtual network change;
- The infrastructure must easily extend over any geographical distance, to account for new participants in a collaboration;
- It must scale to billions of active objects, each with multiple interfaces; and
- It must be able to create a logical workflow for collaborative tasks ("virtual organizations.")

To help overcome these difficulties, InfoSpheres objects have a number of important properties. The underlying hardware and software system does not need to support all of the InfoSpheres objects continuously – objects can be "frozen" and moved to persistent storage, then "thawed" when needed. InfoSpheres is a message-passing architecture: objects have a number of "inboxes" and "outboxes." They use these mailboxes to exchange asynchronous peer-to-peer authenticated messages. The InfoSpheres project explicitly studies the algorithms and conceptual structures necessary to build composable objects with minimal programmer effort.

An InfoSpheres network resembles a Jini network in many ways. For example, Jini also uses composable services as a standard programming paradigm and includes support for object mailboxes.

InfoSpheres, as designed, does not need or use resource allocation. It concerns itself only with connecting the objects the system already knows must join together to create a virtual organization. If physical resources were encapsulated in objects, however, as in Legion, then a resource manager could choose which computational resources would participate in a given virtual organization.

1.5.17 AppLeS and the Network Weather Service

AppLeS' [BW97] offers a metacomputing paradigm where resource schedulers are application-specific. An application's scheduler takes into account:

- The resources available in the metacomputer;
- The application's resource needs;
- The formula that connects an application's performance to the performance of those resources; and
- A predictive picture of the network's future usage.

From this information, it generates a plan for resource utilization, makes reservations if possible, and schedules the application.

AppLeS schedulers break down into four components.

- The *Resource Selector* runs on top of a standard metacomputing system such as Globus or Legion. It uses that system's facilities to locate appropriate collections of resources.
- The *Planner* uses these resources to compute a possible schedule.
- The *Performance Estimator* uses predictive techniques (such as those in the Network Weather Service [NetW]) to determine the application's performance under a given plan.
- Finally, the Actuator implements the best schedule.

The Performance Estimator's predictive picture of the network's future use is both more and less sophisticated than the predictive techniques used in this paper. Rather than competitively preparing for all future possibilities, it draws on numerical models, continuous observations of network resource availability, and feedback techniques. With this information, it makes a reasonably precise guess as to what the future will hold.

1.5.18 Bayanihan

The Bayanihan project [Sar98] studies *volunteer computing*, where users on the Internet cooperate in solving large problems. Bayanihan focuses on building a browser-based Java interface for a metacomputer so that volunteers from the Internet or an Intranet can easily donate their computer's time.

The developers of the Bayanihan project identify the major research issues in volunteer computing as follows.

- The system must implement *adaptive parallelism* to handle a dynamically changing collection of volunteer machines;

- It must be *fault-tolerant*, since machines can crash or lose their network connection to the server while still performing work. Moreover, Byzantine errors are possible;
- It must be *secure*, since hackers can attempt to subvert the system;
- It must be *scalable*, since the potential benefits of metacomputing increase with the size of the machine pool. Scalability is particularly difficult in a Java-based web environment, which encourages star network configurations. (But see [BKKK97].)

The Bayanihan prototype does not provide a fixed answer to any of these problems, although [Sar98] lists several possible solutions to each issue above. Instead, it provides a general architecture based on pools of data/work objects that allows programmers to implement their own solutions.

1.5.19 Bond

The Bond project [BHJM99] explores an economic approach to metacomputing based on the stock market [MBHJ98]. This uses the principle that a large collection of agents intent on maximizing personal "profit" can produce intelligent emergent resource management behavior.

Bond creates an object- and agent-oriented world for distributed computing. It wraps system resources (including processors, communication links, sensors, and software resources like libraries and services) in a special class of objects called *metaobjects*. Communicating with these objects allows software objects to find appropriate resources.

In Bond, all objects use KQML [Fin93] message passing to communicate. This allows Bond to support agents as ordinary members of its object hierarchy. Shadow objects serve as local interfaces for remote objects. All communication passes through them. The only system-dependent elements of Bond's design reside in the object-shadow object communication module.

The Bond system draws on economic and biological processes for its inspiration. In [MBHJ98], it suggests that an efficient model for high-level computation on top of Bond middleware – and other systems – derives the "price" of a resource, and thus a process or sub-process' ability to use that resource, from *market consensus*. In essence, the past performance of a resource, plus the system's demand for that resource, helps set its cost. In this model, agents buy and sell both resources and options on those resources, much as human agents buy and sell stocks and options.

1.5.20 Linda, Piranha, Paradise and JavaSpaces

Linda [Lin] is a distributed programming environment built around the idea of a "tuple space" – a shared memory occupied by logically grouped collections of data ("tuples.") Parallel and distributed programs read tuples destructively, read tuples non-destructively, and write tuples into the space. This serves as a generalization of interprocess communication. With tuples serving as the “output” of completed tasks and the “input” of new tasks, it allows interprocess coordination.

Piranha [Pir] extends the Linda framework to create a metacomputer. Like Condor, Piranha monitors workstations to determine whether they are idle. When they become idle, a Piranha task moves to that machine and begins to compute. When the machine becomes active again, the Piranha task retreats. Communication and coordination uses the tuple space.

JavaSpaces [JavSp] creates a similar tuple space for Java and Jini. This tuple space can store code and data objects, allowing very complex system behavior to emerge from simple and easy-to-understand system components. Augmented with instrumentation to detect and respond to system load conditions, a JavaSpace can become a functional metacomputer.

1.5.21 SETI@Home

SETI@Home [Seti] uses the resources of Internet-connected computers to perform a very large-scale data analysis project: the Search for Extra-Terrestrial Intelligence. This project searches for patterns in the electromagnetic noise radiating to Earth from space – noise that includes universal background radiation, transmissions originating from Earth or its satellites, and, possibly, signals originating from alien worlds.

The SETI@Home phase of the project analyzes data received at the Arecibo telescope in Puerto Rico. The Arecibo telescope has generated 51,166,215 units of data for the project, each unit approximately ¼ of a Megabyte in size. SETI@Home distributes this data to Internet users running the SETI@Home software, which then searches for patterns indicative of alien signal. So far, users have returned 142,250,182 results.

The SETI@Home project has not yet detected extraterrestrial signal. The project will continue another two years, at which point SETI will move on to new approaches to finding extraterrestrial intelligence.

SETI@Home’s users performed work voluntarily, with the SETI@Home software functioning as a screensaver. SETI@Home’s success therefore demonstrates the feasibility of volunteer computing as a method for building efficient metacomputers.

1.5.22 Economics in Computer Science

Spawn [WHHKS92] is an early project integrating economic principles with computer science. Spawn, like Condor, sets out to efficiently use otherwise-idle resources in a computational cluster. It makes resource allocation decisions using a Vickrey auction where processes compete for time slices on various CPUs.

A number of works draw on the idea that intelligent behavior emerges from a market for computational resources [Clear96]. In many cases, as outlined in [WW98], this takes the form of an abstract market where a number of mobile agents bid for raw resources and the services provided by other agents. Such systems can draw on the large body of economic principles to achieve various forms of optimal behavior.

Economic resource allocation in agentless environments is also a significant topic of research. For example, the lottery scheduler for the VINO operating system [SHS99] uses a currency-based negotiation system for resource allocation decisions. This scheduler distributes single-system resource access fairly by handing out numbered "tickets" and holding a drawing to determine who can next access a given resource. Through this probabilistic lottery, each process receives resource access roughly proportional to the number of tickets they possess.

Using the lottery scheduler, individual users can create "currency" backed by these tickets. The total value of their currency equals the total value of their tickets, but the users can distribute currency to their processes as they like. Thus, if every user receives an equal share S of a system resource, a process with currency amounting to 2/3 of a user's tickets receives about 2/3 S of that resource. Processes can exchange tickets and currency through brokers. A heavily CPU-bound process might sell off some I/O tickets in exchange for extra CPU tickets. Market forces produce intelligent behavior, while the stable number of tickets in the system enforces fairness.

Awerbuch *et al.* study resource allocation in networks using economic principles [AAP93]. This work produced routing algorithms competitive in their throughput with the optimal algorithm, and is an ancestor of many of this thesis' ideas.

Chapter Two: Summary of the Theory

This section discusses the theoretical basis for the Cost-Benefit framework and more primitive algorithms for load balancing on a metacomputer. This work measures the effectiveness of online load balancing algorithms using their *competitive ratio*, measured against the performance of an optimal offline algorithm. An online algorithm ALG is c-competitive if for any input sequence I, $\text{ALG}(I) \leq c\ \text{OPT}(I) + \alpha$, where $\text{ALG}(I)$ measures ALG's performance, OPT is the optimal offline algorithm, $\text{OPT}(I)$ measures OPT's performance, and α is a constant.

This chapter expands the theoretical summary found in [AABBK98], a joint work with Dr. Yair Amir, Dr. Baruch Awerbuch, Dr. Amnon Barak, and Dr. Arie Keren. With the exception of the algorithm described in section 2.5.4, the algorithms summarized in this chapter appear in [BFKV92], [AAFPW97], and [AAPW94] and are not part of this thesis' contributions.

2.1 Introduction and Definitions

The problem at hand is minimizing the maximum usage of the various resources on a system. In other words, the various algorithms below seek to balance a system's load. Various practical studies described elsewhere in this work indicate that one algorithm for balancing a system's load also achieves a far more important goal -- producing good performance for the average job. However, the theory itself does not say anything about performance save that the *maximum* resource usage is competitive.

Previous work in this area has identified three relevant machine models and two kinds of jobs. The three machine models are:

1. **Identical Machines.** All of the machines are identical. The speed of a job on a given machine is determined only by the machine's load.
2. **Related Machines.** The machines are identical except that some of them have different speeds. Effectively, we divide the load on each machine m by some constant $r_c(m)$, an indicator of machine speed.
3. **Unrelated Machines.** Many different factors can influence the effective load of the machine and the completion times of jobs running there. Job allocation strategies understand these factors. They know the load a given job adds to a given machine, but that load does not need to have any relationship to the load the job would add to another machine.

The two possible kinds of jobs are:

1. **Permanent Jobs.** Once a job arrives, it executes forever without leaving the system.
2. **Temporary Jobs.** Each job leaves the system when it has received a certain amount of CPU time.

2.2 Identical and Related Machines: the Greedy Algorithm

Assume that reassignments are not possible. The only resource is CPU time, and therefore we seek to minimize the maximum CPU load.

For identical machines and permanent jobs, the greedy algorithm performs well. It assigns the next job to the machine with the minimum current CPU load.

Normalize the loads so that the optimal algorithm places at most one point of load on each machine. The total load imposed by all jobs cannot exceed n, the number of machines. Consider the first time greedy achieves its maximum CPU load, when it places a job with load δ. The minimum CPU load on the system before placing the job is L_{min}. The maximum CPU load on the system after placing the job is $L_{max} = L_{min} + \delta$. Since every machine has a load at least equal to L_{min}, we know that $(n * L_{min}) + \delta < n$. Since $n \geq 1$, we maximize $L_{max} = L_{min} + \delta$ in this equation by maximizing δ. All jobs have load ≤ 1, by the pigeonhole principle. So $L_{max} = L_{min} + \delta < (((n-1)/n) + 1) = 2 - 1/n$.

When the machines are related but not identical, the greedy algorithm can give a competitive ratio of O(log n). To see this, consider a set of machines with speeds as follows:

- The fastest machine, with normalized speed 1;
- [2^0*2^1] Two slower machines with speed 1/2;
- [2^1*2^2] Eight machines with speed 1/4;
- [2^2*2^3] Thirty-two machines with speed 1/8; all the way to
- [2^{2k-1}] machines with speed $1/2^k$.

A stream of jobs comes in: first 2^{2k-1} jobs $1/2^k$ in size. Assuming that faster machines are preferred, greedy uses these jobs to bring every machine faster than $1/2^k$ up to a load of 1. Then come $2^{2(k-1)-1}$ jobs $1/2^{k-1}$ in size. This brings every machine faster than $1/2^{k-1}$ to a load of 2. This continues. Ultimately, the fastest machine achieves a load of k, even though OPT could place these jobs to achieve a maximum load of 1. The best possible competitive ratio is equal to O(log n). The worst-case competitive ratio for greedy on related machines is also O(log n).

2.3 Unrelated Machines: ASSIGN-U

ASSIGN-U is an algorithm for unrelated machines and permanent job assignments. It employs an exponential function to measure the 'cost' of a machine with a given load [AAFPW97]. This algorithm assigns each job to a machine to minimize the total cost of all of the machines in the cluster. More precisely, let:

- a be a constant, $1 < a < 2$,
- $l(m,j)$ be the load of machine m before assigning job j, and
- $p(m,j)$ be the load job j will add to machine m.

The online algorithm assigns *j* to the machine *m* that minimizes the marginal cost, shown in equation 2.1.

$$H_i(j) = a^{l(m,j)+p(m,j)} - a^{l(m,j)}.$$

Equation 2.1: Marginal Cost

This algorithm is O(log n) competitive for unrelated machines and permanent jobs.

The work presented in [AAPW94] extends this algorithm and competitive ratio to temporary jobs, using up to O(log n) *reassignments* per job. A reassignment moves a job from its previously assigned machine to a new machine. In the presence of reassignments, let

- *h(m,j)* be the load of machine *m* just before *j* was last assigned to *m*.

When any job is terminated, the algorithm of [AAPW94] checks a 'stability condition' for each job *j* and each machine *M*. This stability condition, with *m* denoting the machine on which *j* currently resides, is shown in equation 2.2.

If this stability condition is not satisfied by some job *j*, the algorithm reassigns *j* to machine *M* that minimizes $H_M(j)$.

$$a^{h(m,j)+p(m,j)} - a^{h(m,j)} \le 2*(a^{l(M,j)+p(M,j)} - a^{l(M,j)}).$$

Equation 2.2 Stability Condition

For unrelated machines and temporary jobs, without job reassignment, there is no known algorithm with a competitive ratio better than *n*.

2.4 Online routing of virtual circuits

The *ASSIGN-U* algorithm above minimizes the maximum usage of a single resource. A related algorithm for online routing of virtual circuits can extend *ASSIGN-U*'s competitive ratio to environments with multiple resources. In the online routing problem, the scheduler is given:

- A graph G(V,E), with a capacity *cap(e)* on each edge *e*,
- A maximum load *mx*, and
- A sequence of independent requests (s_j , t_j , $p:E \rightarrow [0,mx]$) arriving at arbitrary times. s_j and t_j are the source and destination nodes, and *p(j)* is the required bandwidth. A request that is assigned to some path *P* from a source to a destination increases the previous load *l(e,j)* on each edge $e \in P$ by the amount *p(e,j) = p(j)/cap(e)*.

The scheduler's goal is to minimize the maximum link congestion, which is the ratio between the bandwidth allocated on a link and its capacity.

ASSIGN-U is extended further in [AAFPW97] to address the online routing problem. The algorithm computes the marginal cost for each possible path P from s_j to t_j as per equation 2.3:

$$H_P(j) = \sum_{e \varepsilon P} (a^{l(e,j)+p(e,j)} - a^{l(e,j)}).$$

Equation 2.3: Marginal Cost on a Network

and assigns request j to a path P that yields a minimum marginal cost.

This algorithm is O(log n) competitive [AAFPW97].

Minimizing the maximum usage of CPU and memory, measuring memory usage in terms of the fraction of memory consumed, can be reduced to the online routing problem. This reduction works as follows: create two nodes, $\{s, t\}$ and n non-overlapping two-edge paths from s to t. Machine m is represented by one of these paths, with a *memory* edge with capacity $r_m(m)$ and a *CPU* edge with capacity $r_c(m)$. Each job j is a request with s as the source, t as the sink, and p a function that maps memory edges to the memory requirements of the job and CPU edges to 1. The maximum link congestion is the larger of the maximum CPU load and the maximum memory (over)usage.

Using this reduction, the algorithm above can manage heterogeneous resources and remain O(log n) competitive in its maximum usage of each resource.

2.5 Understanding the Theory: Why ASSIGN-U Works

2.5.1 Definitions

We have n machines, each with R resources. Imagine that each atomic unit of each resource has a price. We set these prices so that the first $(1 / \log 2nR)^{th}$ of each resource has a total price of one dollar, the second $(1 / \log 2nR)^{th}$ of each resource has a total price of two dollars, the third a total price of four dollars, and so forth. Resources have an initial cost of \$1 when unused, so the price for a resource is always less than or equal to the cost incurred for its utilization. The last $(1 / \log 2nR)^{th}$ of a resource costs nR dollars.

We make a simpler version of ASSIGN-U which chooses the machine with the lowest cost for any given job, using this price structure. We also assume for simplicity that each job requires at most $(1 / \log 2nR)^{th}$ of any resource. For a competitive ratio of $\log 2nR$, we assume that the optimal algorithm uses at most one $(1 / \log 2nR)$ slice of each resource all told, while our algorithm uses up to 100% thereof.

The special assignment a_0 puts load 0 and cost \$1 on each resource. Our algorithm then makes assignments $a_1 \ldots a_k$. The optimal algorithm instead makes assignments $a_1' \ldots a_k'$. Each assignment a_i goes to machine $m(a_i)$.

We define a utilization function, $0 \leq u_r(a_i') \leq 1$, such that a_i' uses $u_r(a_i') / \log 2nR$ of $m(a_i')$'s resource r. In other words, $u_r(a_i')$ represents the portion of OPT's $(1 / \log 2nR)$ slice of that resource that OPT uses when assigning the i^{th} job.

We define a cost function, $c_r(a_i)$, as the cost for resource r in assignment a_i assuming that *our* algorithm made all previous assignments. We define $c(a_i) = \Sigma\ c_r(a_i)$. Since our algorithm is greedy, $c(a_i) \le c(a_i')$.

We define a machine cost function, $C(M,i) = \Sigma\ c(a_j) \mid (j \le i\ \&\ m(a_j) = M)$, as the sum of the costs on a machine M after the i^{th} assignment.

2.5.2 Machine Matching

Suppose that we have exactly n jobs, which OPT assigns to one machine each. When we must make assignment a_i, we have the option of making the same assignment OPT does. This adds at most $1 / \log 2nR$ to the load of and at most doubles the cost of any given resource on machine $m(a_i')$, so $c(a_i) \le c(a_i') \le C(m(a_i'),i)$.

After assignment a_0, each machine M has cost $C(M,0) = R$. Suppose that after making assignment a_i, machine $m(a_i)$ has cost $C(m(a_i),i)$. Consider each assignment a_j previously made to machine $m(a_i)$. Using our exponential cost function, the corresponding assignment a_j' had a total cost less than the cost $C(m(a_j'),j)$. So $c(a_j) \le c(a_j') \le C(m(a_j'),j)$. This yields equation 2.4, below.

$$C(m(a_i),i) \le \sum_{j \mid m(a_j)=m(a_i) \& j \le i} C(m(a_j'),j)$$

Equation 2.4: Bound on Machine Cost after Assignment "i"

Consider the series of assignments made to each machine $m(a_j')$ before assignment a_j. Each such assignment a_k has a *unique* corresponding assignment a_k', by our assumption that OPT places at most one job on each machine. This yields equation 2.5.

$$C(m(a_i),i) \le \sum_{j \mid m(a_j)=m(a_i) \& j \le i} C(m(a_j'),j) \le \sum_{k \mid m(a_k)=m(a_j') \& m(a_j)=m(a_i) \& k<j \le i} C(m(a_k'),k)$$

Equation 2.5: Looser Bound on Machine Cost after Assignment "i"

Continuing in this manner, we find equation 2.6:

$$C(m(a_i),i) \le \sum_{\text{Machine } M} C(M,0) = nR$$

Equation 2.6: Final Bound on Machine Cost after Assignment "i"

In other words, the maximum total cost for all the jobs on any machine is nR, which gives us a competitive ratio of $\log nR$.

2.5.3 Multiple Jobs per Machine

Now suppose that we have any number of jobs and that the optimal algorithm can assign more than one job to a machine.

Due to our exponential pricing function, we can bound the cost function as per equation 2.7:

$$c(a_i') \leq \sum_{\text{Resource } r} u_r(a_i') * (\sum_{\text{Assignments } a_j} c_r(a_j) \mid j < i \,\&\, m(a_j) = m(a_i'))$$

Equation 2.7: Cost Function Bound

$$f(a_i) \leq \sum_{\text{Resource } r} u_r(a_i') * (\sum_{\text{Assignments } a_j} c_r(a_j) \mid j < i \,\&\, m(a_j) = m(a_i'))$$

$$g(a_i) \leq \sum_{\text{Resource } r} c_r(a_i) * (\sum_{\text{Assignments } a_j} u_r(a_j') \mid j > i \,\&\, m(a_j') = m(a_i))$$

$$g(a_0) = nR \; (\$1 \text{ per resource})$$

Equation 2.8: Functions Limiting Machine Cost

That is, the cost to use the same resource that the optimal algorithm does is at most $u_r(a_i')$ times the sum of the costs of the jobs already on that resource on machine $m(a_i')$.

We define $f(a_i)$ as this "cap" on the price of assignment a_i, and $g(a_i)$ as the amount assignment a_i contributes to the cap of other assignments. See equation 2.8.

Imagine a flow, where $f(a_i)$ is the input to assignment a_i and $g(a_i)$ is its output. The source of this flow is $g(a_0)$. Since the optimal algorithm uses at most $(1 / \log 2nR)$ of each resource, $c(a_i) \geq g(a_i)$. This means that $f(a_i) \geq g(a_i)$. Since the flow out of an assignment is always greater than the flow into that assignment, it logically follows that the maximum flow into any set of assignments $\{f(a_i)\}$, not counting internal flow, is at most $g(a_0) = nR$.

The set of jobs our strategy assigns to a machine but the optimal algorithm does not has a maximum cost of nR. If we also assign every job to that machine that the optimal algorithm does, we spend $2nR$ on that machine. Our maximum utilization on any resource is exactly 100%, giving a competitive ratio of $\log 2nR = \log nR + 1$.

2.5.4 Jobs of Unknown Size

In **Chapter Five: Strategies with Reduced Information**, this work develops a variant on ASSIGN-U for environments where the system does not know the size of a job until it has been placed. It makes assignment a_i to the machine M with the minimum cost $C(M,i\text{-}1)$.

No algorithm can have an O(*log nR*) competitive bound without knowledge regarding the size of the incoming job. However, suppose that the ratio between a job's usage of a given resource on one machine and its usage on the optimal machine is always less than a resource size constant S. Then $c(a_i) < 2S * c(a_i')$. If we give our algorithm a further $2S$ times as much of each resource, $c(a_i) \leq f(a_i)$ still holds, yielding a competitive ratio of $2S * \log nR$. A continuous pricing structure tightens this to $S * \log nR$.

Chapter Three: Static Strategies

This chapter expands on the work found in [AABBK98], a joint work with Dr. Yair Amir, Dr. Baruch Awerbuch, Dr. Amnon Barak, and Dr. Arie Keren.

3.1 The Model

We must assign each job in an incoming stream of jobs $j_1 \ldots j_k$ to one of n machines.

We define each machine m as a collection of *resources*. These include $r_c(m)$, the CPU resource, measured in cycles per second, and $r_m(m)$, the memory resource, measured in bytes. Note that these resources are incomparable by ordinary standards. Although this work can extend to handle other resources, such as network bandwidth and I/O, we will limit ourselves to CPU and memory. Thus, we define machine m as $\{ r_c(m), r_m(m) \}$.

Each job j_i has the following properties:

- $a(j_i)$, the job's arrival time;
- $t(j_i)$, the number of CPU cycles (the "CPU time") required by the job;
- $m(j_i,m)$, the memory load the job imposes on machine m;
- $l(j_i,m)$, the CPU load the job imposes on machine m; and
- $c(j_i)$, the job's completion time, which varies depending on system load and the system scheduler's decisions.

Of these properties, $a(j_i)$, $m(j_i,m)$, and $l(j_i,m)$ are known when a job arrives, while $t(j_i)$ and $c(j_i)$ are known upon job completion.

3.1.1 System Load

Let $J(t,m)$ be the set of jobs in machine m at time t. Then we define the CPU load $l_c(t,m)$ and the memory load $l_m(t,m)$ of machine m at time t as per equation 3.1.

$$l_c(t,m) = \sum_{j_i \varepsilon J(t,m)} l(j_i,m) \quad = |J(t,m)| \text{ for constant } l(j_i) = 1,$$

and

$$l_m(t,m) = \sum_{j_i \varepsilon J(t,m)} m(j_i,m), \text{ respectively.}$$

Equation 3.1: Machine Load

We assume that when a machine runs out of main memory, disk paging slows the machine down by a multiplicative factor of τ. We therefore define the *effective CPU load* of machine m at time t, $L(t,m)$, as per equation 3.2:

$$L(t,m) = l_c(t,m) \qquad \text{if } l_m(t,m) \le r_m(m),$$

and

$$L(t,m) = l_c(t,m) * \tau, \qquad \text{otherwise.}$$

Equation 3.2: Effective CPU Load

For simplicity, we assume that all machines schedule jobs *fairly*. At time *t*, each job on machine *m* receives 1/*L(t,m)* of the CPU resource. A job's completion time, $c(j_i)$, therefore satisfies equation 3.3.

$$\int_{a(j)}^{c(j)} \frac{r_c(m)}{L(t,m)} = t(j_i), \text{ where } m \text{ is the machine where we assigned job } j_i.$$

Equation 3.3: Completion Time of a Job

3.1.2 Jobs

For purposes of building test beds for our various strategies, we also modeled the characteristics of a stream of jobs. Jobs arrived according to a Poisson distribution and had independently-generated CPU time and memory requirements. See equations 3.4 and 3.5. We based this model of incoming jobs on the observations of real-life processes found in [HD96].

$$a(j_i) = a(j_{i-1}) + \alpha\chi, \text{ where}$$

α is a constant,

$0 \le \beta \le 1$ is a constant, and

$\chi =$ positive integer z with probability $(1-\beta)*(\beta^{z-1})$.

Equation 3.4: Job Arrival Time

$$t(j_i, m) = \xi * \frac{1}{\eta} * \frac{r_c(f)}{r_c(m)},$$

$$m(j_i, m) = \Xi * \frac{1}{\mathrm{I}} * \frac{r_m(b)}{r_m(m)}, \text{ and}$$

$l(j_i) = 1$, where f is the fastest machine, b has the most memory,

ξ, Ξ are constants and

$0 < \eta, \mathrm{I} \le 1$ are uniformly distributed random variables.

Equation 3.5: Job Characteristics

Since this work is specifically concerned with metacomputing clusters, each job also had a 5% chance of being a large parallel batch job. Such batches contained a random number of jobs, uniformly distributed between one and twenty, each with ten times the CPU time requirement the job would otherwise have required. The jobs in a parallel batch arrived simultaneously.

3.1.3 Slowdown

The "slowdown" for a job is conceptually the degree to which system load and the choice of machine assignments slows a job down. If a job requires 10 CPU seconds on the fastest machine, and completes in 50 seconds, it suffers a slowdown of 5. (See equation 3.6.)

$$\frac{(c(j)-a(j))*r_c(f)}{t(j)}, \text{ where } f \text{ is the fastest machine.}$$

Equation 3.6: Job Slowdown

We evaluated the effectiveness of various job assignment strategies in terms of their *average slowdown* over all jobs in a given scenario. The lower the average slowdown, the better the strategy performed.

Our goal in this chapter is to develop a method for job assignment that will minimize the average slowdown over all jobs.

3.2 From Theory to Practice

For each machine in a cluster of n machines, with resources $r_1 \ldots r_k$, we define that machine's *cost* as per equation 3.7.

$$\sum_{i=1}^{k} f(n, \text{utilization of } r_i) \qquad \textbf{Generic Cost}$$

or

$$\sum_{i=1}^{k} a^{\text{load on resource } i}, 1 < a < 2 \qquad \textbf{Using ASSIGN-U}$$

Equation 3.7: Machine Cost

The *load* on a given resource equals its usage divided by its capacity. We assume for convenience that our algorithm has resources O(log n) times greater than the optimal algorithm's, so that the two have identical maximum loads. This changes our machine loads, but should not change how we apply the cost function. Accordingly, we rewrite the cost function as shown in equation 3.8.

$$\sum_{i=1}^{k} a^{O(\log n) * \textit{utilized } r_i / \max \textit{usage of } r_i}$$

or, with properly chosen a,

$$\sum_{i=1}^{k} n^{\textit{utilized } r_i / \max \textit{usage of } r_i}$$

Equation 3.8: Machine Cost

The *marginal cost* of assigning a job to a given machine is the amount by which this sum increases when the job is assigned there. Our "opportunity cost" approach to resource allocation assigns jobs to machines in a way that minimizes this marginal cost. ASSIGN-U uses an opportunity cost approach.

In this chapter, we are interested in only two resources, CPU and memory, and we will ignore other considerations. Hence, the above theory implies that given logarithmically more memory than an optimal offline algorithm, ASSIGN-U will achieve a maximum slowdown within O(log n) of the optimal algorithm's maximum slowdown.

This does not guarantee that an algorithm based on ASSIGN-U will be competitive in its *average* slowdown over all processes. It also does not guarantee that such an algorithm will improve over existing techniques. Our next step was to verify that such an algorithm does, in fact, improve over existing techniques in practice.

The memory resource easily translates into ASSIGN-U's resource model. The cost for a certain amount of memory usage on a machine is n^u, where u is the proportional memory utilization (used memory / total memory.) For the CPU resource, we must know the maximum possible load. Drawing on the theory, we will assume that L, the smallest integer power of two greater than the largest load we have seen at any given time, is the maximum possible load. This assumption, while inaccurate, does not change the competitive ratio of ASSIGN-U.

The final cost for a given machine's CPU and memory load, using our method, is shown in equation 3.9.

$$n^{\frac{\textit{used memory}}{\textit{total memory}}} + n^{\frac{\textit{CPU load}}{L}}$$

Equation 3.9: Final Machine Cost

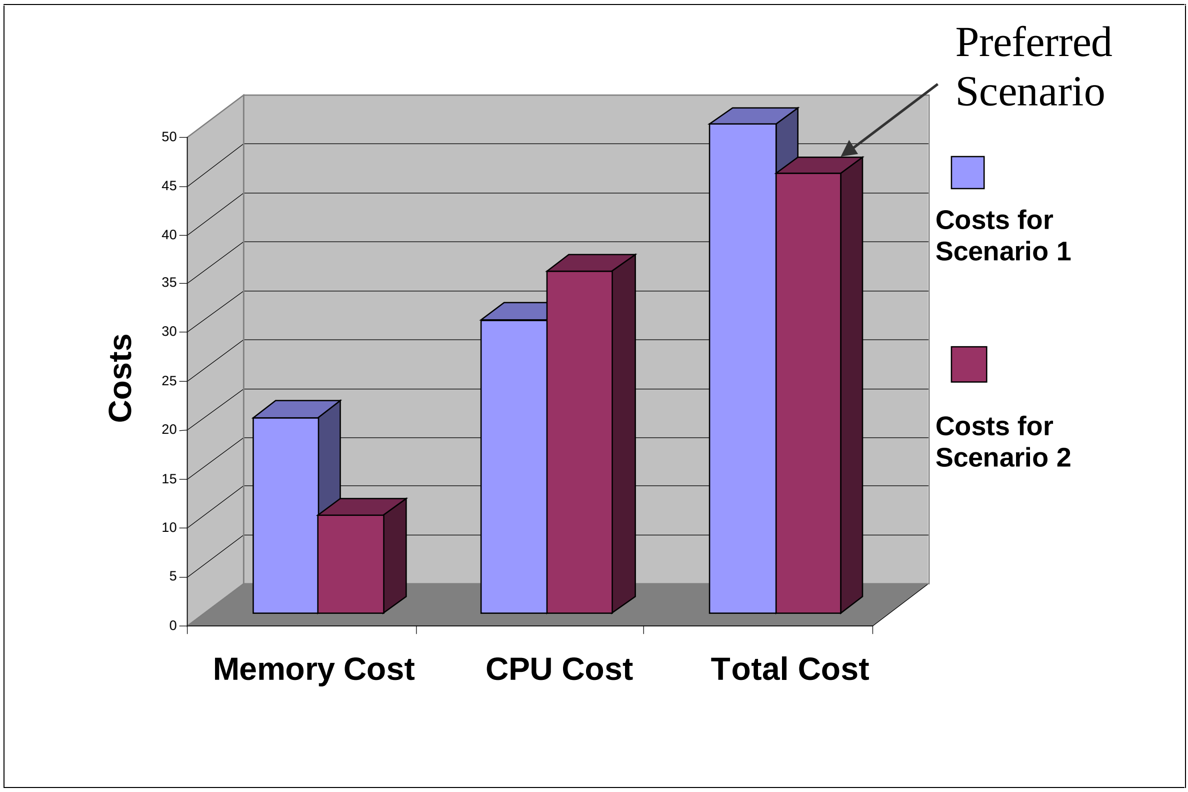

Figure 3.1: Choosing the Right Machine Becomes Easy!

In general, we assign jobs so as to minimize the sum of the costs of all the machines in the cluster. (See Figure 3.1.)

To examine the behavior of this "opportunity cost" approach, we evaluated two different methods for job assignment. PVM is a standard system. E-PVM is a scheduler of our own design, using this algorithm to assign and reassign jobs.

1. **The PVM Strategy** assigns jobs to machines in strict round robin order. PVM (for "Parallel Virtual Machine") is a popular metacomputing environment for systems without preemptive process migration. Unless the user of the system specifically intervenes, PVM assigns jobs to machines using this strategy. It does not reassign jobs once they begin execution. Since round robin is not the strongest job assignment strategy, we will compare our opportunity cost approach against more powerful strategies later in this work.

2. **The Enhanced PVM Strategy** is our modified version of the PVM Strategy. It uses the opportunity cost-based approach to assign each job as it arrives to the machine where the job has the smallest marginal cost at that time. No other factors come into play, so using this strategy is very easy. As with PVM, initial assignments are permanent. We sometimes abbreviate the Enhanced PVM Strategy as E-PVM.

```
We can describe E-PVM using pseudo-code as:
  max_jobs = 1;
  while () {
    cost = MAX_COST;
    when a new job j arrives:
      for (each machine m) {
        marginal_cost = power(n, percentage memory utilization on m if j was added) +
          power(n, (jobs on m + 1) / max_jobs) - power(n, memory use on m) -
          power(n, jobs on m / max_jobs);
        if (marginal_cost < cost) { machine_pick = m; }
      }
      assign job to machine_pick;
      if (jobs on machine_pick > max_jobs) max_jobs = max_jobs * 2;
    }
  }
```

Figure 3.2: E-PVM Pseudo-code

Note the simplicity of the pseudo-code. Implementing the E-PVM decision algorithm on a real system requires as little as 20-30 lines of code.

3.3 The Simulation Test Bed

The first test bed for our opportunity cost approach was a simulated cluster of six machines, matching a set of six machines then available in the Center for Networking and Distributed Systems laboratory. Certain assumptions made in this simulated cluster will be reexamined and altered later in the work, but the simulation has remained an accurate predictor of an opportunity cost scheduler's behavior throughout. These machines are described in Table 3.1.

Jobs arrived at about one per ten seconds for ten thousand simulated seconds, which gave a variety of load conditions to each of our methods.

In each execution of the simulation, both methods were provided with an identical scenario, where the same jobs arrived at the same rate.

We picked our simulation parameters, and miscellaneous factors such as the thrashing constant τ (10), conservatively. Our goal was to provide our algorithm with a harsh testing environment less favorable to it than the real world.

Machine Type	# of these Machines	Processing Speed	Installed Memory
Pentium Pro	3	200 MHz.	64 MB of RAM
Pentium	2	133 MHz.	32 MB of RAM
Laptop w/ Ethernet	1	90 MHz.	24 MB of RAM

Table 3.1: The Simulated Cluster

3.3.1 Simulation Results

We evaluated the results of the simulations in two different ways:

- An important concern is the overall slowdown experienced using each method. The *average slowdown by execution* is an unweighted average of all of the simulation results, regardless of the number of jobs in each execution. The *average slowdown by job* is the average slowdown over all of the jobs in all of the executions of the simulation. The average slowdown by jobs gives more weight to scenarios with many jobs. The difference between these two averages therefore indicates the trend in system performance as scenarios grow harder. These average results, incorporating 3000 executions, are given in Table 3.2.
- The behavior of Enhanced PVM is significantly different in lightly loaded and heavily loaded scenarios. This behavior is illustrated in Figure 3.3, detailing the first 1000 executions of the simulation.

Slowdown	PVM	E-PVM
Jobs	15.404	10.701
Executions	14.334	9.795

Table 3.2: Average slowdown in the Java simulation for the different methods.

Comparison graphs like the figures below will appear with most of the results in this work. In Figure 3.3, the X axis is the average slowdown for the PVM Strategy. The Y axis is the average slowdown for the Enhanced PVM Strategy. The light line is defined by ‘x = y’. Above this line, the unenhanced algorithm does better than the enhanced algorithm. Below this line, the enhanced algorithm does better than the unenhanced algorithm.

Enhanced PVM, as Table 3.2 has already shown, does significantly better than straight PVM in almost every circumstance. More interesting, however, is its behavior as the average slowdown for the PVM Strategy increases. The larger PVM’s average slowdown was on a given execution, the more improvement our enhancement gave. Intuitively, when an execution was hard for all four models, Enhanced PVM did much better than unenhanced PVM. If a given execution was relatively easy, and the system was not heavily loaded, the enhancement had less of a positive effect. The dark line in Figure 3.3 is a linear trend line for the data points, illustrating this effect.

The reason for this phenomenon runs as follows. When a machine becomes heavily loaded or starts thrashing, it does not just affect the completion time for jobs already submitted to the system. If the machine does not become unloaded before the next set of large jobs is submitted to the system, it is effectively unavailable to them, increasing the load on all other machines. If many machines start thrashing or become heavily loaded, this effect will build on itself. Every incoming job will take up system resources for a much longer span of time, increasing the slowdown experienced by jobs that arrive while it computes. Because of this pyramid effect, a ‘wise’ initial assignment of jobs and careful re-balancing can result (in the extreme cases) in a vast improvement over standard PVM, as shown in some of the executions in Figure 3.3.

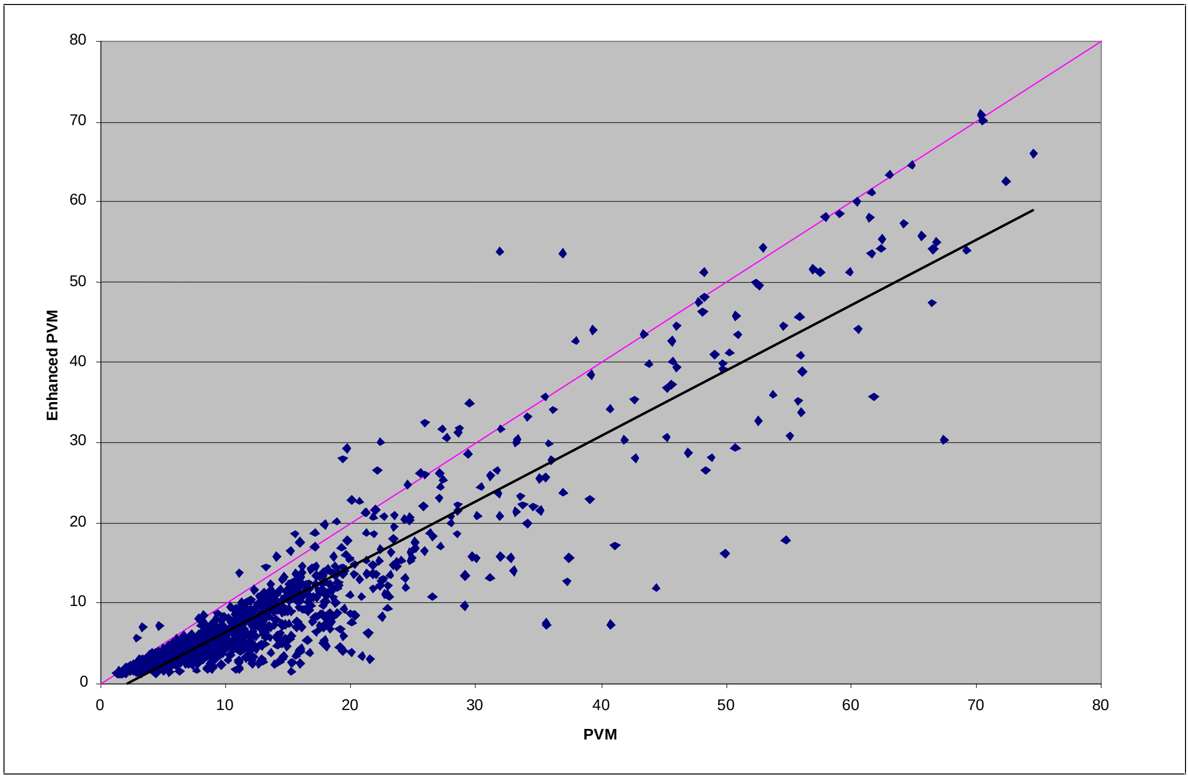

Figure 3.3: PVM vs. Enhanced PVM (Simulation)

3.4 Real System Executions

We also tested these algorithms on a real cluster, using the same model for incoming jobs. We implemented each job with a program that cycled through an array of the appropriate size, performing calculations on the elements therein, for the appropriate length of time. The jobs were assigned using the PVM and Enhanced PVM strategies.

Table 3.3 shows the slowdowns for 50 executions on this real cluster. Figure 3.4 shows the results point-by-point. Again, the light line is defined as 'x = y', and the dark line is a linear trend line. The results of the real system executions are shown below.

Slowdown	PVM	E-PVM
Jobs	19.818	12.272
Executions	18.835	11.698

Table 3.3: Average slowdown in the real cluster for 2 (re)assignment methods.

The test results in Table 3.3 imply that the real-life thrashing constant and various miscellaneous factors increased the average slowdown. This is the expected result of picking conservative simulation parameters. More importantly, these results do not substantially change the relative values. The Enhanced PVM Strategy performed even better on the real cluster, compared to regular PVM. We consider this to be a strong validation of our initial Java simulations and of the merits of this opportunity cost approach.

Table 3.4 shows the total impact of the opportunity cost approach on the static system.

Slowdown on Simulation for …	**PVM vs. E-PVM**
Jobs	1.440
Executions	1.463
Slowdown in Real System for …	**PVM vs. E-PVM**
Jobs	1.615
Executions	1.610

Table 3.4: Average relative slowdowns for 2 job (re)assignment methods.

The results presented in Table 3.2, Table 3.3, and Table 3.4 give strong indications that the opportunity cost approach is among the best methods for adaptive resource allocation in a scalable computing cluster.

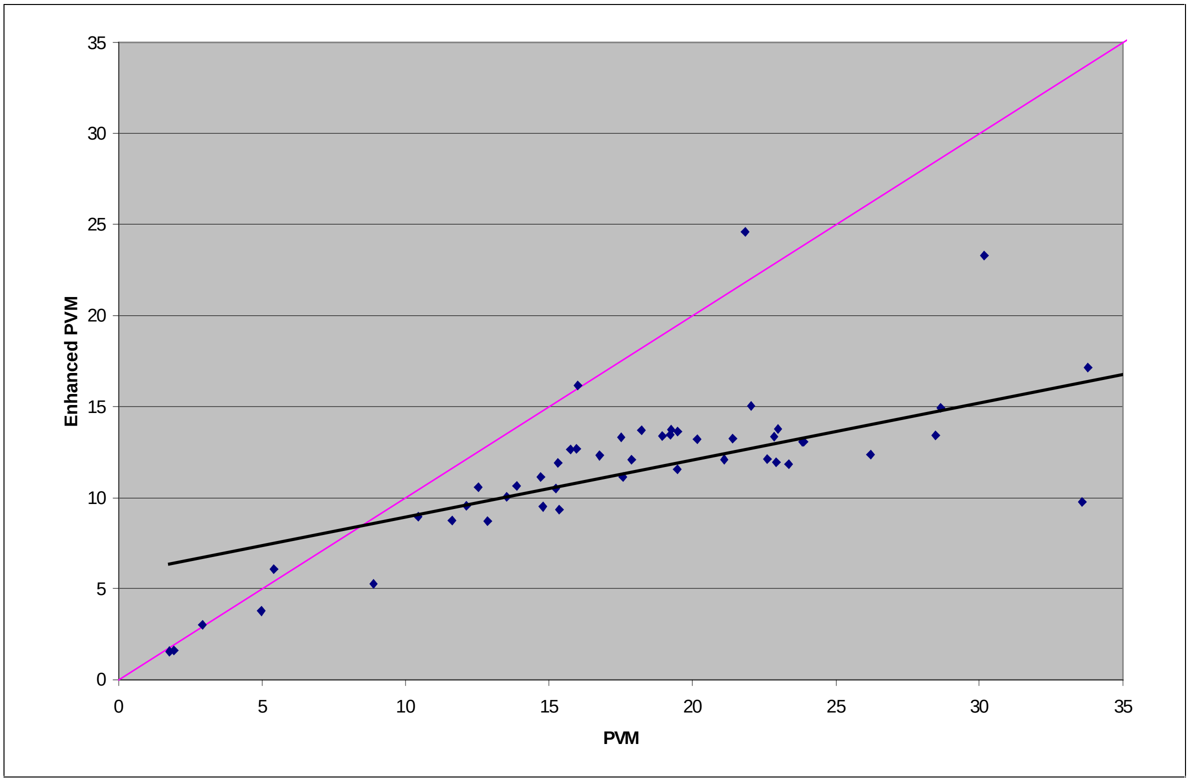

Figure 3.4: PVM vs. Enhanced PVM
(Real Executions)

Chapter Four: Dynamic Strategies

This chapter expands on the work found in [AABBK98], a joint work with Dr. Yair Amir, Dr. Baruch Awerbuch, Dr. Amnon Barak, and Dr. Arie Keren.

4.1 The Model

We must assign each job in an incoming stream of jobs $j_1 \ldots j_k$ to one of n machines. We define these machines and jobs as in section 3.1.

In the dynamic model, the scheduler has additional power. It can *reassign* jobs. More precisely, we define a constant "clock tick" φ and system time t. When t modulo φ is 0, each machine m selects a random fixed-size subset M of the other machines. If it so desires, it can take a job running on m and move it to any machine in M. The job then continues its execution. It does not lose the work performed so far, and therefore the job completion time obeys equation 4.1.

$$\int_{a(j)}^{c(j)} \frac{r_c(m)}{L(t,m)} = t(j_i), \text{ where } m \text{ is the machine where job } j_i \text{ is currently assigned.}$$

Table 4.1: CompletionTime of a Job

4.2 From Theory to Practice

The arrival rate and resource demands of jobs are unpredictable. In light of this unpredictability, any non-prescient strategy sometimes assigns jobs to a non-optimal machine. Job reassignment gives the system the power to correct these mistakes. Applied correctly, this is an extremely powerful tool for reducing the average slowdown.

The model above derives from the behavior of Mosix, an implemented system for job reassignment. Accordingly, results obtained using this model map naturally into the real world.

Migration is not a panacea. Reassigning a job to correct an error does not erase the error – it has already contributed to the slowdown of any jobs that it shared a machine with. This means that those jobs stay on the system longer and in turn contribute a larger amount to the slowdown of other jobs. This compounds like interest or debt, so that a relatively small series of mistakes, quickly corrected, can add a great deal to the system's eventual slowdown.

Since bad decisions have an effect in the dynamic systems world, it makes sense to evaluate the opportunity cost approach in this environment. The reassignment strategy of ASSIGN-U requires the ability to reassign jobs to any machine at any time. Further, it is built on the idea that minimizing reassignments is valuable. Since a real system like Mosix limits the system's opportunities for reassignment, but makes reassignment cheap within that context, we do not employ the ASSIGN-U strategy. Instead, we use the Mosix model for when the system may reassign a job, and use assignments and reassignments to greedily minimize the load. (See Figure 4.1.)

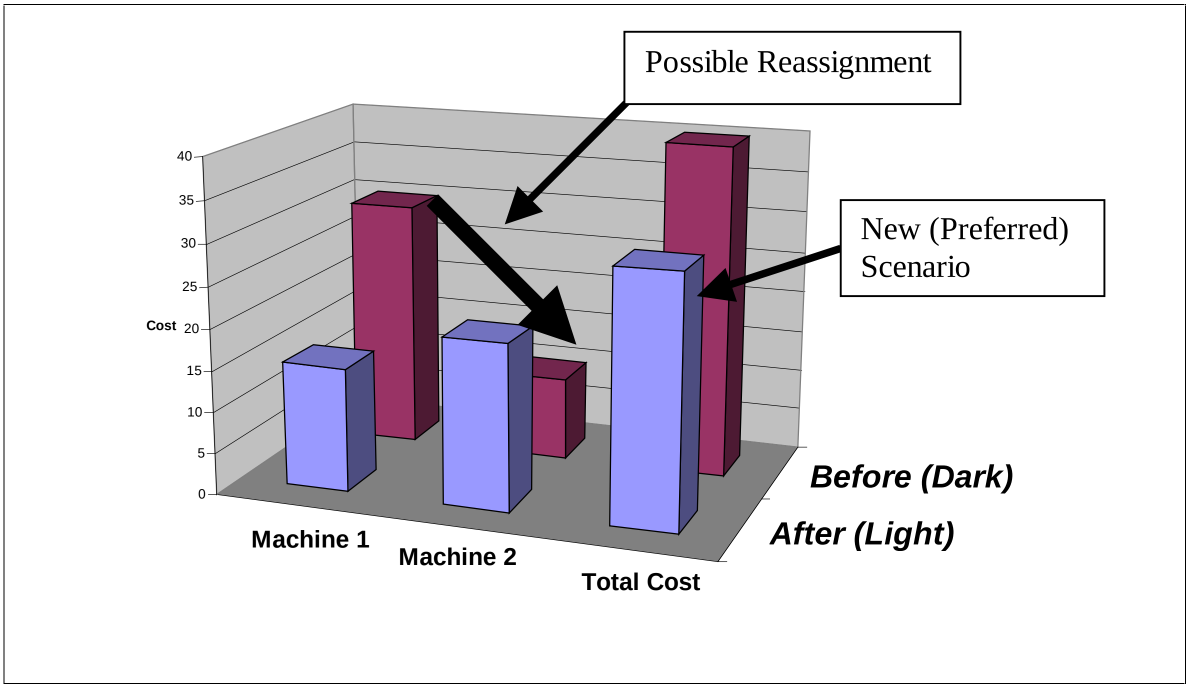

Figure 4.1: Choosing Whether to Reassign Becomes Easy!

To examine the behavior of our approach, we evaluated four different methods for job assignment.

1. **The PVM Strategy**, as previously defined, does not take advantage of job reassignment.
2. **The Enhanced PVM Strategy**, as previously defined, uses our opportunity cost approach but does not take advantage of job reassignment.
3. **The Mosix Strategy** is an adaptive load-balancing strategy that also endeavors to keep some memory free. In Mosix, a process becomes a candidate for migration when the difference between the relative loads of a source machine and a target machine crosses a certain threshold. Older CPU-bound processes receive a higher migration priority. Mosix machines accumulate information about the processes at regular intervals (*e.g.* each time tick φ) and then exchange this information with other machines. If appropriate, they then migrate some set of jobs. Each machine exchanges information only with a small selection of other machines. This limitation on Mosix's knowledge makes it possible to make decisions quickly. The waiting period φ between migrations minimizes the migration overhead. The Mosix kernel enhancements that allow a system to perform job reassignment use this strategy.
4. **The Enhanced Mosix Strategy** is our modified version of the Mosix strategy, using the opportunity cost approach. It greedily assigns or reassigns jobs to minimize the sum of the costs of all the machines. The Enhanced Mosix Strategy

has the same limits on its knowledge and reassignment abilities as the Mosix Strategy. We sometimes abbreviate the Enhanced Mosix Strategy as E-Mosix.

We can describe E-Mosix in pseudo-code as follows:

```
Max_jobs = 1;
while () {
 when a new job j arrives: {
   cost = MAX_COST;
   for (each machine m) {
    marginal_cost = power(n, percentage memory utilization on m if j was added) +
     power(n, (jobs on m + 1) / max_jobs) - power(n, memory use on m) -
     power(n, jobs on m / max_jobs);
    if (marginal_cost < cost) { machine_pick = m; }
   }
   assign job to machine_pick;
   if (jobs on machine_pick > max_jobs) max_jobs = max_jobs * 2;
 }
}
```

Figure 4.2.1: E-Mosix Pseudo-code (Part 1: Job Placement)

Job placement in E-Mosix functions exactly as in E-PVM.

```
every X seconds: {
  for (each machine m) {
  choose a small random set of machines M;
  for (each job j on m) {
   current_cost = power(n, percentage memory utilization on m) +
    power(n, jobs on m / max_jobs) -
    power(n, percentage memory utilization on m if j is migrated away) -
    power(n, ((jobs on m) - 1) / max_jobs);
   for (each machine m2 in M) {
    marginal_cost = power(n, percentage memory utilization on m2 if j was added) +
      power(n, (jobs on m2 + 1) / max_jobs) - power(n, memory use on m2) -
      power(n, jobs on m2 / max_jobs);
    if (marginal_cost < current_cost) {
      transfer j to m2;
      if (jobs on m2 > max_jobs) max_jobs = max_jobs * 2;
      break from loop: for (each machine m2);
    }
   }
  }
}
```

Figure 4.2.2: E-Mosix Pseudo-code (Part 2: Job Migration)

When migrating jobs, E-Mosix looks at each machine individually. Each machine *m* contacts a small random subset *M* of the other machines. E-Mosix then evaluates, job by job, whether it would reduce the total cost of the system to migrate the job to another machine in *M*. If so, it migrates the job. Note that E-Mosix is a greedy algorithm. It reduces overall system cost whenever it sees an opportunity to do so, rather than attempting to calculate the optimum set of job reassignments.

4.3 The Simulation Test Bed

We tested our dynamic opportunity cost approach in the simulated cluster described in Table 3.1. We used the same job arrival scenarios as for our static tests, and can therefore compare the results directly against the results of the static scenarios.

4.3.1 Simulation Results

As before, we evaluated the results of the simulations in two different ways.

- First, we examined the *average slowdown by execution* and the *average slowdown by job*. These results, incorporating 3000 executions, are given in Table 4.1.
- The behavior of Enhanced PVM and Enhanced Mosix are significantly different in lightly loaded and heavily loaded scenarios. This behavior is illustrated in Figures 4.3 through 4.8.

Slowdown	PVM	E-PVM	Mosix	E-Mosix
Jobs	15.404	10.701	9.421	8.203
Executions	14.334	9.795	8.557	7.479

Table 4.1: Average slowdown in the Java simulation for the different methods.

Figure 4.3 is a comparison graph that benchmarks Enhanced Mosix against Mosix. Again, the light line is defined by 'x = y' and the dark line is a linear trend line for the data points. As with E-PVM, the behavior of E-Mosix improves on "hard" scenarios. The trend line illustrates this effect.

Figure 4.4 compares the E-PVM method, which makes no reassignments at all, to the powerful Mosix system. In this case, a single opportunity cost-based job assignment competes against an arbitrary number of reassignments by the carefully optimized Mosix Strategy. Given the power of reassignment, we cannot expect that E-PVM will outperform Mosix. It does, however, perform extremely well. Its performance stays close to Mosix's even in very hard scenarios, as the trend line shows.

Figure 4.5 illustrates the value of migration, and thus the importance of studying the opportunity cost approach in the dynamic context. Enhanced Mosix outperforms Enhanced PVM in almost every case, often considerably. The intelligent decision made by E-PVM is well complemented by the ability to reassign jobs according to the opportunity cost strategy. The linear trend line for this data indicates that the benefits of combining job reassignment with the opportunity cost strategy increase dramatically as scenarios become "hard."

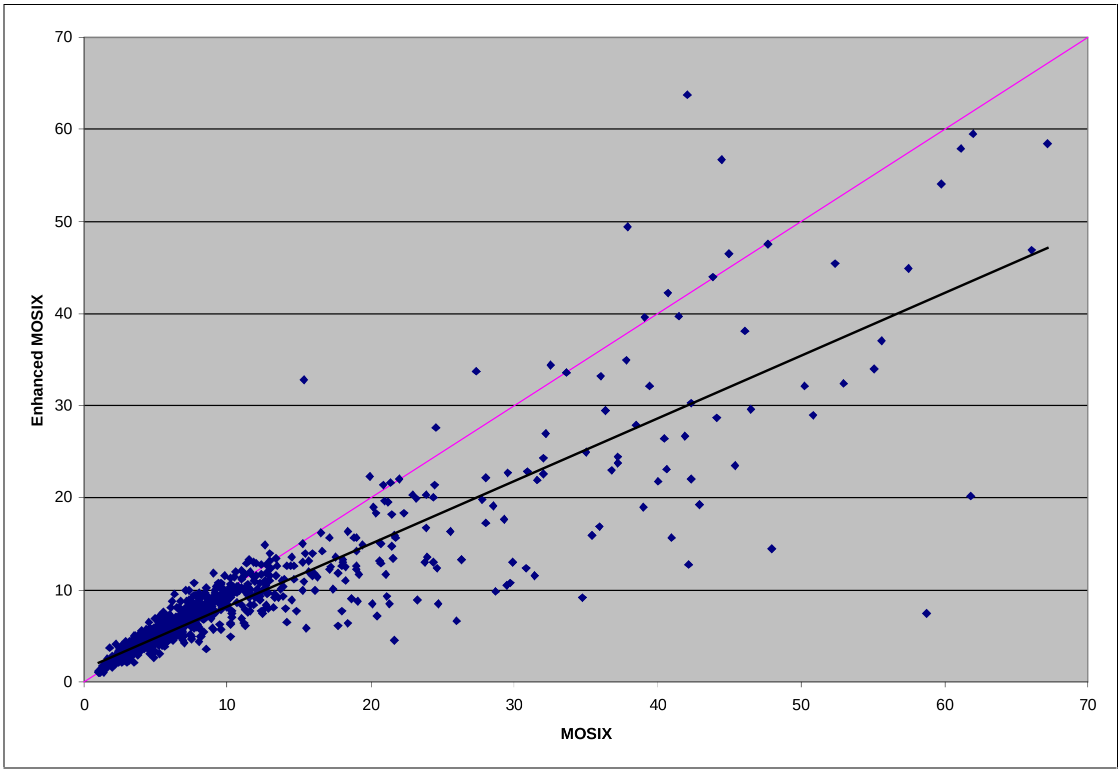

Figure 4.3: Mosix vs. Enhanced Mosix (Simulation)

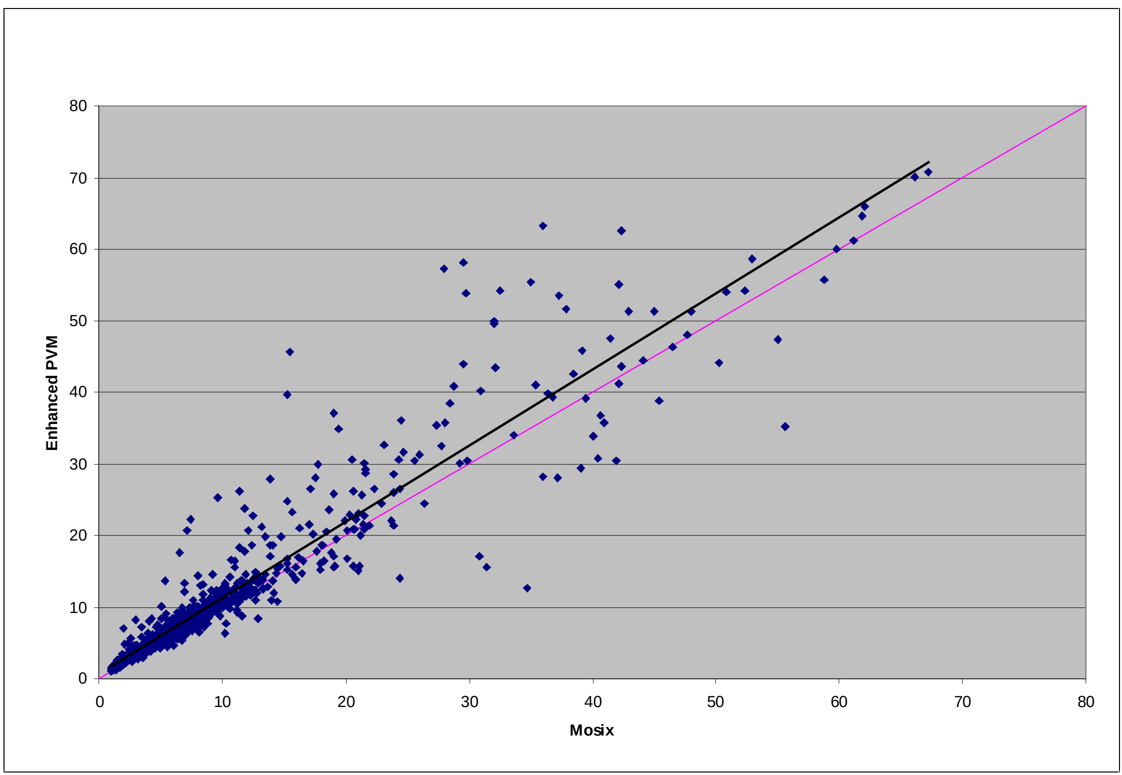

Figure 4.4: Mosix vs. Enhanced PVM (Simulation)

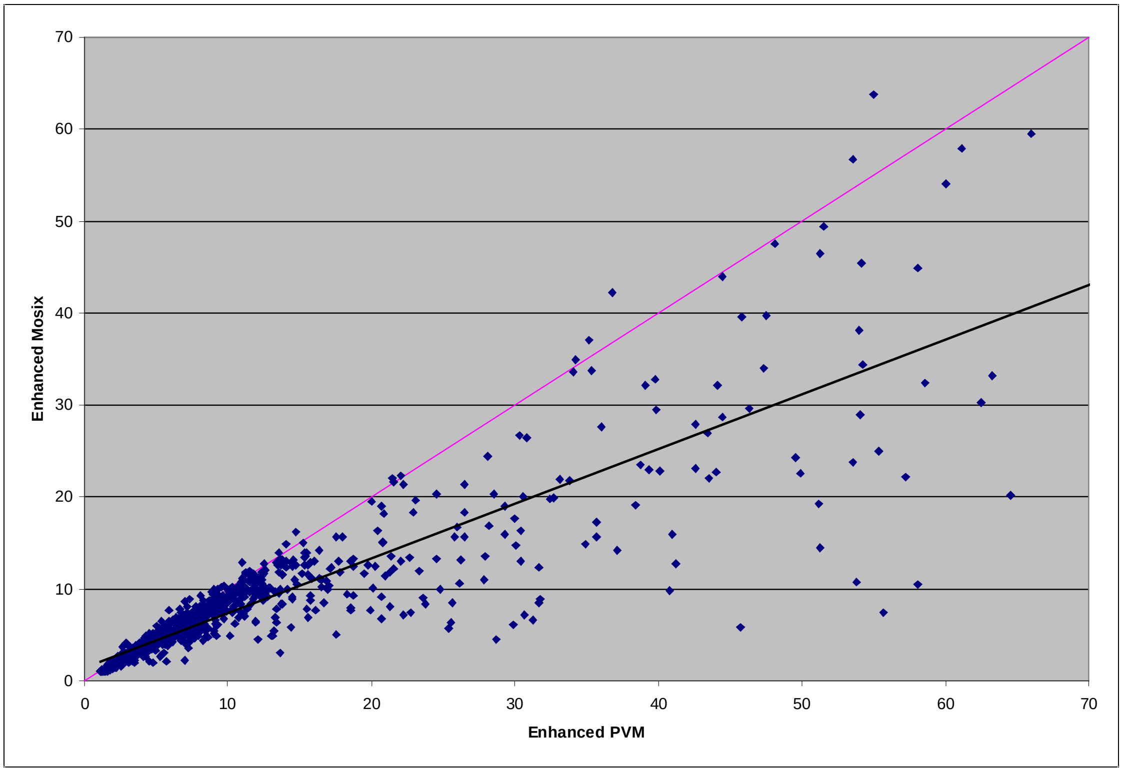

Figure 4.5: Enhanced Mosix vs. Enhanced PVM (Simulation)

4.4 Real System Executions

Testing the E-Mosix strategy on a real cluster requires a modified version of Mosix that uses our opportunity cost algorithms in the kernel. We do not yet have access to such a kernel. However, we were able to test E-PVM against the *unenhanced* Mosix strategy on our real cluster, using the same scenarios and parameters as in section 3.4. Table 4.2 shows the slowdowns for 50 executions on this real cluster. Figure 4.6 shows the results point-by-point.

Slowdown	PVM	E-PVM	Mosix
Jobs	19.818	12.272	8.475
Executions	18.835	11.698	8.683

Table 4.2: Average slowdown in the real cluster for 3 (re)assignment methods.

The test results in Table 4.2 show that the Mosix strategy is stronger in a Mosix-kernel environment than in our simplified simulation. However, E-PVM remains a strong contender. About 2/3 of the benefit for implementing Mosix in the kernel can be realized by instead employing E-PVM at the user level. Since using a Mosix-capable operating system and installing the Mosix kernel enhancements is not always a desirable option, this is a strong argument in favor of E-PVM. It also indicates that E-Mosix is likely to outperform Mosix in practice.

Table 4.3 shows the impact of the opportunity cost approach on dynamic systems.

Slowdown on Simulation for …	**PVM vs. Mosix**	**E-PVM vs. Mosix**	**PVM vs. E-PVM**
Jobs	1.635	1.139	1.440
Executions	1.675	1.145	1.463
Slowdown in Real System for …	**PVM vs. Mosix**	**E-PVM vs. Mosix**	**PVM vs. E-PVM**
Jobs	2.282	1.413	1.615
Executions	2.222	1.380	1.610

Table 4.3: Average relative slowdowns for 3 job (re)assignment methods.

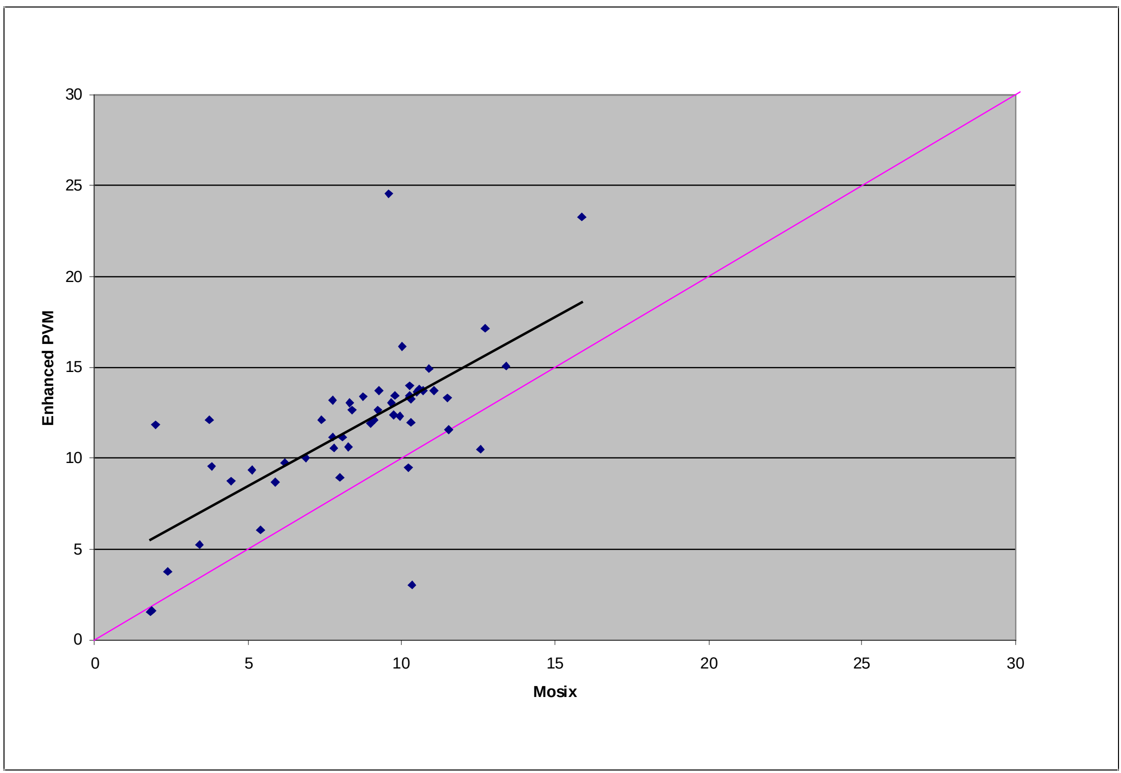

Figure 4.6: Mosix vs. Enhanced PVM (Real Executions)

Chapter Five: Strategies with Reduced Information

5.1 The Model

We define a *reduced information scenario* as follows. We must assign a stream of jobs to one of n machines. Machines are defined as in section 3.1. Also as in section 3.1, each job j_i has the following properties:

- $a(j_i)$, the job's arrival time;
- $t(j_i)$, the number of CPU cycles (the "CPU time") required by the job;
- $m(j_i,m)$, the memory load the job imposes on machine m;
- $l(j_i,m)$, the CPU load the job imposes on machine m; and
- $c(j_i)$, the job's completion time, which varies depending on system load and the system scheduler's decisions.

In a reduced information scenario, $t(j_i)$ and $c(j_i)$ are known upon job completion, as in section 3.1. However, only $a(j_i)$ is known when a job arrives. We do not know $m(j_i,m)$, and $l(j_i,m)$ until immediately after we place a job. In other words, we must make our scheduling decisions without knowing the memory or CPU load a job will add to the system.

In this context, we consider only static strategies. A dynamic system can employ the E-Mosix strategy in a reduced information scenario by tentatively assigning jobs to a heuristically appropriate machine and then reassigning them at the next tick, when their load requirements are known. (Creating opportunity cost strategies with even less information, where a job's resource requirements remain unknown for a period of time after assignment, is a subject of ongoing research.)

This model is intended to reflect real world environments. Real systems can usually obtain roughly accurate load information, but jobs do not necessarily know their resource requirements in advance.

5.2 From Theory to Practice

In a reduced information scenario, we do not know the change in resource usage from placing a job. We do, however, know the slope of the cost function at the current utilization. We use this slope as a very rough approximation of the marginal cost. (We analyzed this approximation in section 2.5.4.)

To evaluate this approximation, we compared three strategies:

1. **The PVM Strategy**, as previously defined, is unaffected by reduced information scenarios.

2. **The Enhanced PVM Strategy** functions as previously defined. Note that the Enhanced PVM Strategy has full information, as in Section 3.1. In other words, it does not face a reduced information scenario. Rather, it serves as a benchmark against which we compare our approximation. The closer our strategy for reduced information scenarios comes to the performance of E-PVM, the less we suffer from the loss of information.

3. **The Differential PVM Strategy** assigns jobs to the machine whose cost function has the smallest current slope. (Equivalently, it assigns each job greedily to the machine with the smallest current cost.)

We can describe the Differential PVM Strategy using pseudo-code as:

```
max_jobs = 1;
while () {
 cost = MAX_COST;
 when a new job j arrives {
   for (each machine m) {
     machine_cost = power(n, memory use on m) +
       power(n, jobs on m / max_jobs);
     if (machine_cost < cost) { machine_pick = m; }
    }
    assign job to machine_pick;
    if (jobs on machine_pick > max_jobs) max_jobs = max_jobs * 2;
  }
}
```

Figure 5.1: Differential PVM Pseudo-code

5.3 Evaluation

We tested the Differential PVM Strategy against the PVM and E-PVM Strategies in our simulated cluster. We generated scenarios using the same job arrival pattern as for the static and dynamic tests. We could not use the specific scenarios used in earlier tests. We conducted no real cluster tests, as changes in lab configuration rendered this impractical.

5.3.1 Simulation Results

Table 5.1 shows the compiled results of 3000 tests.

Slowdown	PVM	Differential PVM	E-PVM
Executions	16.371	11.038	10.137
Jobs	17.521	12.119	11.098

Table 5.1: Average slowdown in the Java simulation for the different methods.

Averaging by execution, Differential PVM captures 91.8% of the benefit of Enhanced PVM. When averaging by jobs, a slightly harsher measure, it captures 91.6% of the benefit. In other words, users adopting the Differential PVM Strategy – presumably due to constraints on available information – gain 91-92% of the performance increase they would achieve if they could employ E-PVM.

Table 5.2 shows the ratio between the average slowdowns for the PVM and Differential PVM Strategies, as well as the ratio between the Differential PVM and Enhanced PVM Strategies.

Slowdown in Simulation for …	PVM vs. Differential PVM	Differential PVM vs. E-PVM
Jobs	1.483	1.089
Executions	1.446	1.092

Table 5.2: Average relative slowdowns for 3 job (re)assignment methods.

Note the consistency in the third column of Table 5.2. The Differential PVM / E-PVM ratio remains much the same regardless of averaging technique. Since averaging slowdown by jobs gives more weight to hard scenarios with many jobs, this consistency suggests that increasing scenario sizes has little effect on Differential PVM's performance relative to E-PVM's.

Figure 5.2 compares Differential PVM against PVM. Figure 5.3 compares Differential PVM against E-PVM. We expect the E-PVM Strategy to outperform the Differential PVM Strategy, because it uses more information. We expect the Differential PVM Strategy to outperform the PVM Strategy, because it uses load information intelligently while PVM uses no information at all. As the figures show, this intuition is correct in almost every scenario.

More importantly, Figure 5.3 demonstrates that the Differential PVM results are close to the E-PVM results in the vast majority of all scenarios. In "easy" cases, the Differential PVM Strategy outperforms the PVM Strategy, and its performance closely matches E-PVM's. This makes sense, since E-PVM uses its extra information not to optimize performance but to prevent bad performance. In "hard" scenarios, the Differential PVM Strategy's performance still approximates E-PVM's. Both strategies outperform the PVM results in every case – often, significantly.

Figure 5.4 puts these results in the same graph so that one can compare the two trend lines and dispersion patterns directly.

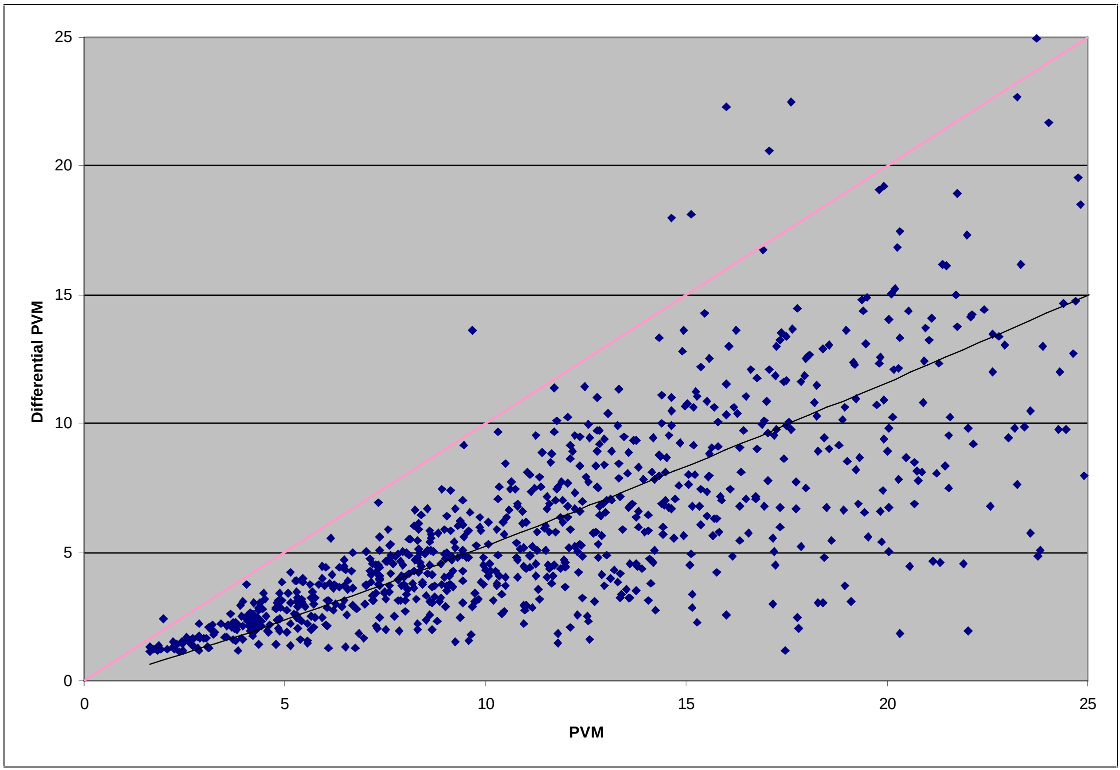

Figure 5.2: Differential PVM vs. PVM (Simulation)

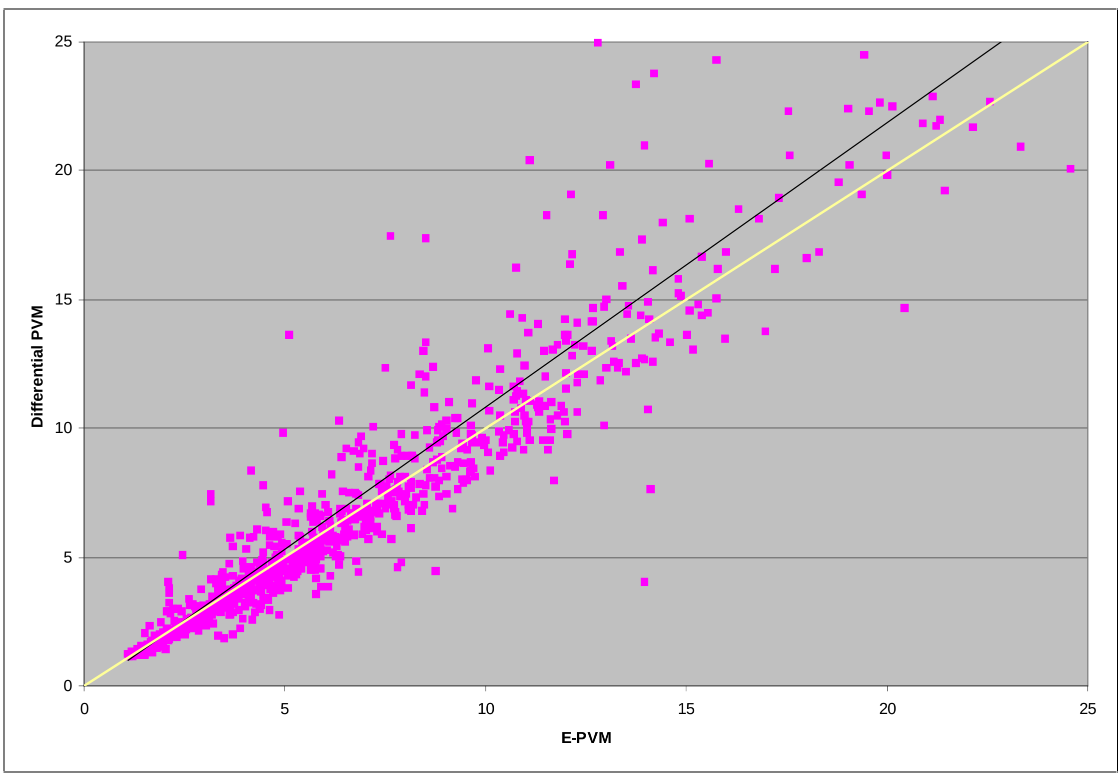

Figure 5.3: Differential PVM vs. E-PVM (Simulation)

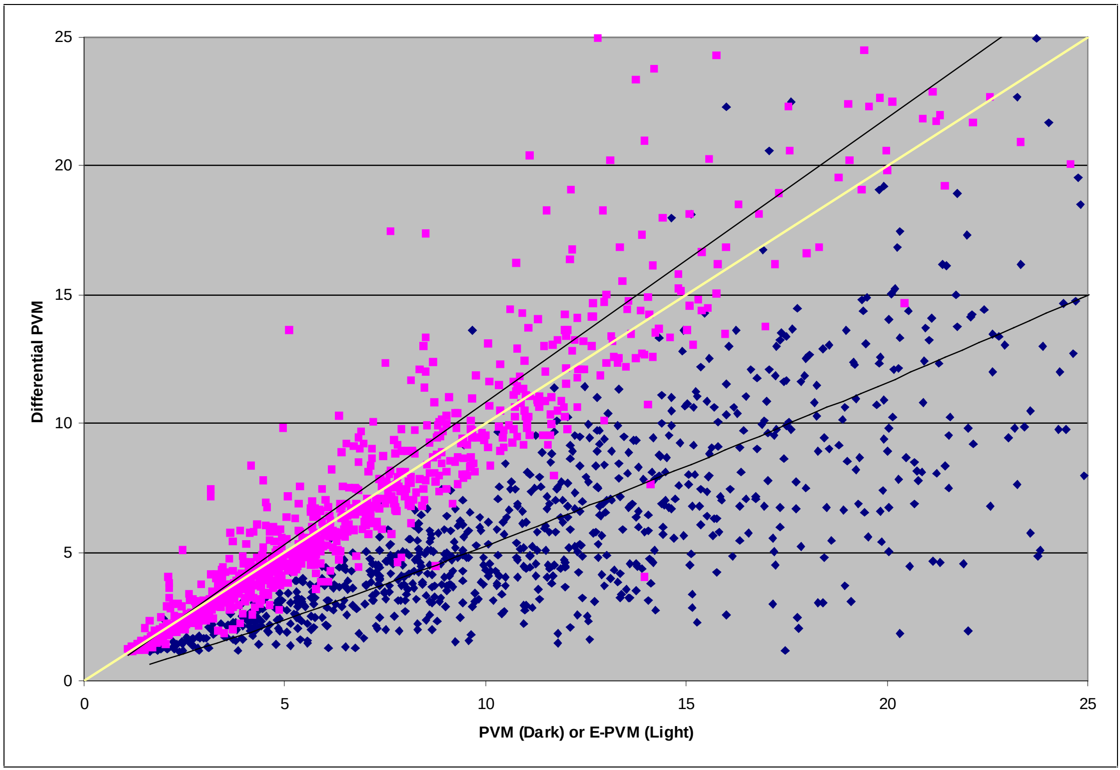

Figure 5.4: Differential PVM vs. Other Strategies (Simulation)

Chapter Six: The Java Market

6.1 Description

We developed the Java Market as the first metacomputer test bed for the Cost-Benefit framework. It was designed as an Internet-wide market for computational services. For technical reasons described in section 6.2.1, we moved on to the Frugal system (described in **Chapter Seven: The Frugal System**) after developing an initial prototype for the Java Market.

The Java Market, although built on the volunteer computing model, does not use the program structures developed in projects such as Bayanihan [Sar98]. Rather than building a programming structure designed to squeeze the most utility out of the Market's environment, the Java Market project uses "legacy" code. That is, it transparently adapts ordinary Java applications to the closest equivalent applets. This increases utility at the expense of efficiency.

This chapter draws on the work found in [AAB00], a joint work with Dr. Yair Amir and Dr. Baruch Awerbuch.

6.1.1 Basic Concepts

Two entities define the Java Market's world: machines and jobs. Both machines and jobs make *contracts* with the Java Market. Machines sell computational services to the Market. Jobs buy such services from the Market. This allows the efficient use of unused cycles available anywhere on the Internet.

The Java Market "pays" producer machines for their services and "charges" consumers for each job they run. For example, a consumer might offer to pay $20 if the Java Market completes a large simulation in 6 hours or less. A machine owner might charge the Java Market $10 for 8 hours of their machine's services. The market tries to maximize its own profit. This is equivalent to maximizing the real wealth generated by the system, in turn equivalent to using the system's resources in the most globally beneficial way.

The prototype did not flesh out this system, but in principle the Market could measure payments and charges in either virtual money (usable to buy Java Market services) or real currency. Virtual money has the advantage of easy acceptance by the public; real currency allows the Market to achieve its full power and utility.

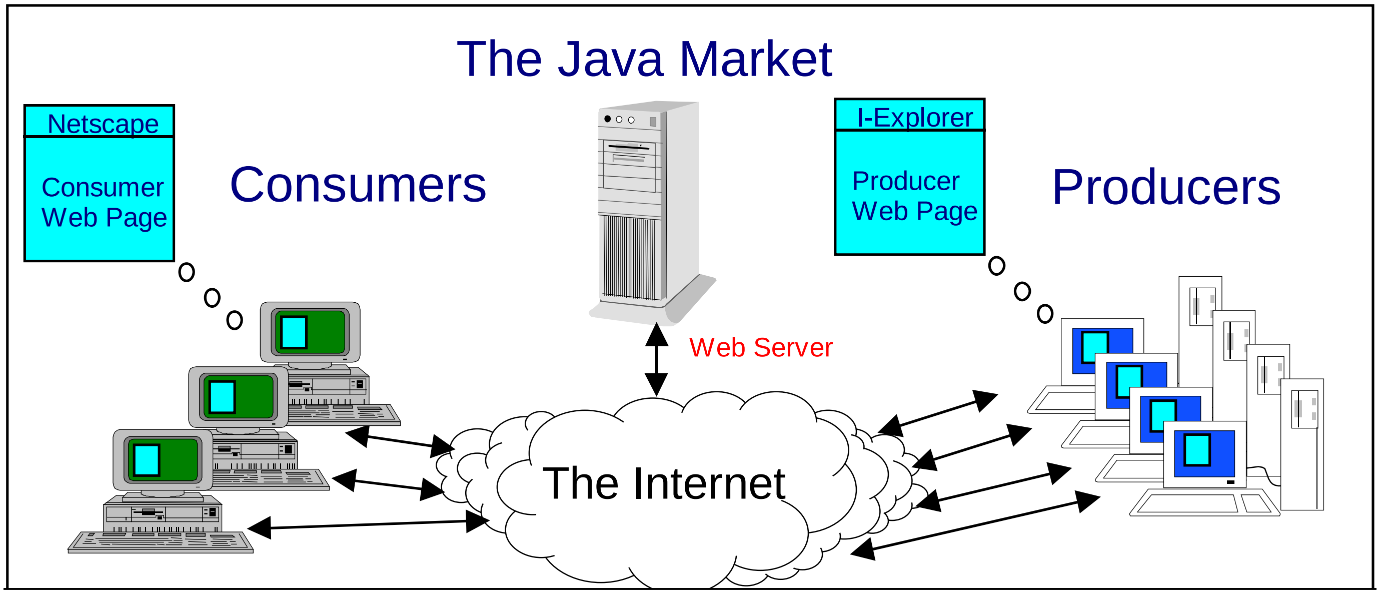

Figure 6.1: An Internet-Wide Metacomputer

6.1.2 The System

The Java Market brokers the distribution of computational resources among machines scattered across the world. As depicted in Figure 6.1, we designed the Java Market to transfer jobs from *any* machine on the Internet *to* any machine on the Internet that wishes to participate. There is no installation or platform-dependent code – the only requirement is that the jobs be written in Java. Further, the Java job does not need to be written especially for the Market – the Market can rewrite Java applications into applets automatically, and provides services that can overcome some of the inherent applet restrictions. The Market can then port these applications automatically to any producer machine. Using the Market is only slightly more difficult than clicking on a browser bookmark.

The Java Market has no dependence on any given architecture. Producers and consumers can use any machine and any operating system that has a Java-capable Web browser. The prototype completely implements the Market's program-transfer technology.

The Java Market uses the Web and the Java language as its primary tools. A producer makes their machine available as a resource by directing its browser to one of the Market web pages. A consumer registers its request for computational resources by posting its program (written in Java) in a Web-accessible location and contacting another of the Market's web pages.

The Java Market is composed of three main entities, as depicted in Figure 6.2:

- **The Resource Manager** keeps track of the available machines – those that have registered themselves as producers.

- **The Task Manager** keeps track of the consumer-submitted tasks.
- **The Market Manager** mediates between the Resource Manager and the Task Manager.

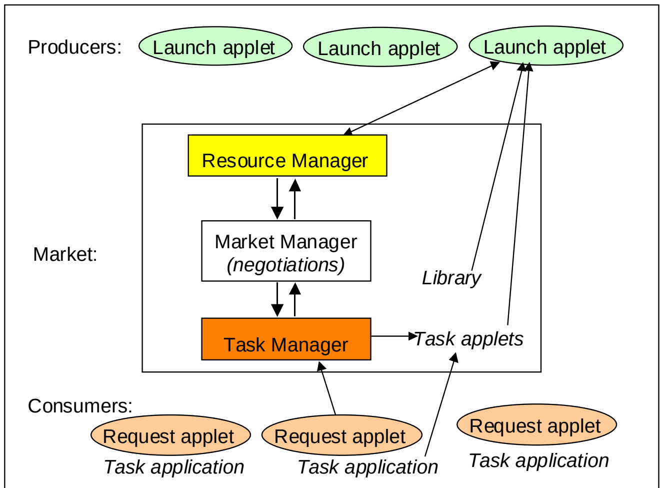

Figure 6.2: Java Market Components

6.1.3 The Resource Manager

The Resource Manager runs on the Market machine. When a producer registers with the Market web pages, they automatically run a special applet, the *Launch Applet*, that tells the Resource Manager about the producer's machine's power. The Resource Manager stores this information, as well as the state of the machine (in this case, 'available,') the IP address, and so forth. This information forms a machine profile.

The Launch Applet is the Resource Manager component executed on the producer's machine. This gives it two responsibilities. It must perform resource discovery: that is, assessing the machine's computational and networking capabilities. It must also direct the producer's browser to the web page containing that machine's assigned task.

6.1.4. The Task Manager

The Task Manager runs on the Market machine. When a consumer registers their task with the Market web pages, they run another special applet, the *Request Applet*, which gathers information about the task they want the Market to perform. Once it gathers this information, the Request Applet sends it to the Task Manager. The Task Manager then gathers the Java files and input files associated with the task from the Web, edits the Java files as necessary, compiles them, and passes the entire task to the Market Manager.

The Request Applet is the Task Manager component executed on the consumer's machine. Its primary responsibility is to wait. First, it waits while the consumer types in the necessary data about their task. It sends this information to the Task Manager proper and then waits again. As the Request Applet waits, the Task Manager downloads, edits, and compiles all of the Java code associated with the task. When it finishes this work, it tells the Request Applet whether the Market has accepted or rejected the task. The Request Applet displays this information and ceases computation.

6.1.5. The Market Manager

The Market Manager oversees Market operations, performing resource allocation and admission control. It must address two key issues:

- A producer machine might not be *available* for the entire time period required by a given task, and
- Maximizing the market profit requires intelligent decision making.

The prototype did not address the issue of predicting resource availability, although the Market *does* respond to resource connections and disconnections. Maximizing the market profit uses the Cost-Benefit Framework, as described in section 6.2.2.

6.1.6. An Example Scenario

The scenario presented in Figure 6.3 illustrates the Java Market metacomputing system.

1. A producer machine registers its availability over the network with the Java Market.
2. A second producer machine registers its availability with the Java Market.
3. A consumer connects to the Java Market and registers a task.
4. The Task Manager downloads the task information from the Web and modifies and compiles the code. It then notifies the consumer of the consumer's success at launching the task.
5. The Market Manager mediates between the Task Manager and the Resource Manager in order to find an appropriate producer to execute the task.
6. The selected producer's browser automatically begins executing the task.
7. The task completes and its results are mailed to the consumer.
8. Later, a producer leaves the Java Market.

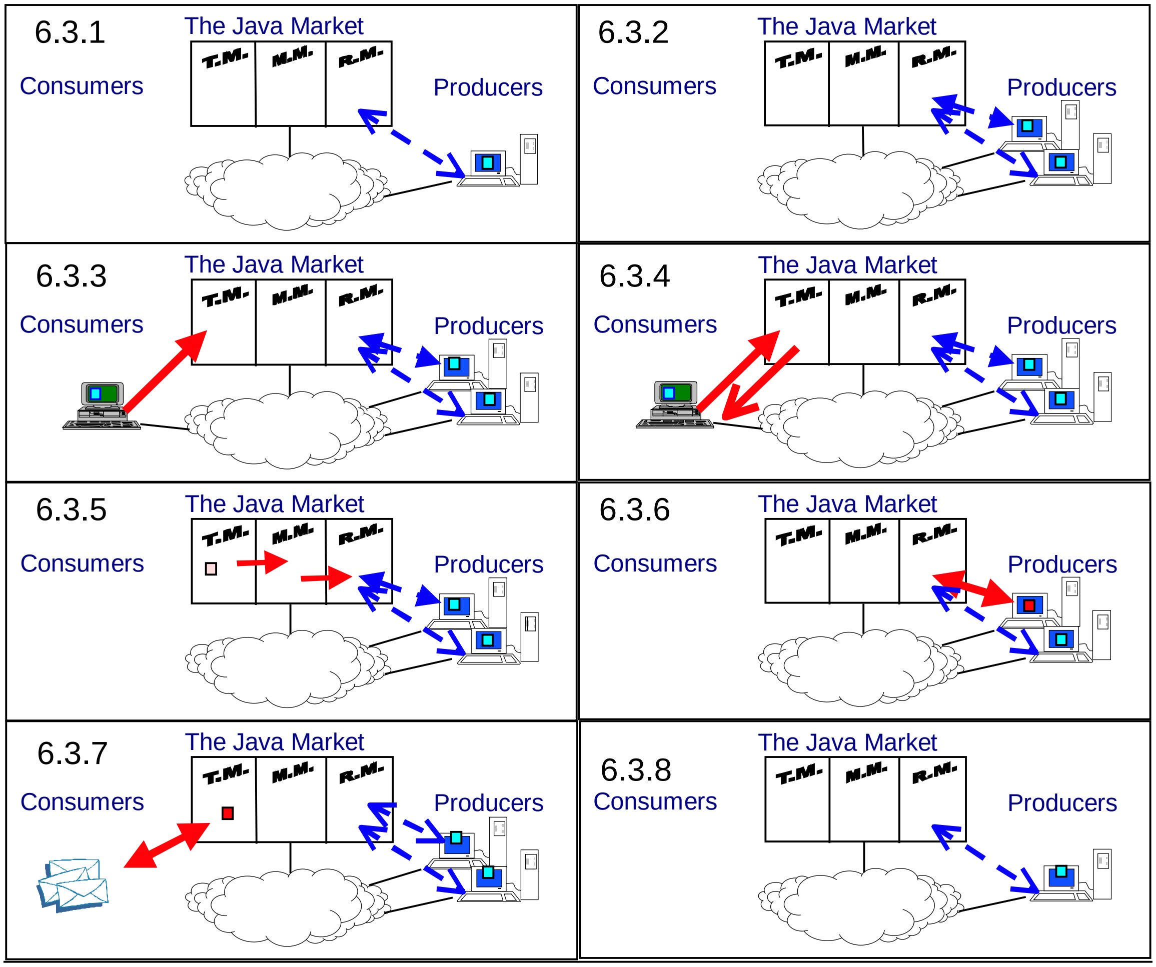

Figure 6.3: A Sample Java Market Scenario

6.2 Java Market Design

6.2.1 Features of Web-Based Metacomputers

The Java Market builds on Java and web technologies. This gives the project certain inherent advantages and disadvantages.

Java bytecodes are machine-independent. Any sort of machine can execute them. For this reason, machine and operating system heterogeneity do not significantly complicate the Java Market's task. It can accept jobs from and execute jobs on any kind of machine. The Market itself, written in Java, can run on any machine, requiring only parameters describing the local environment. Except for some GUI details, Market code and submitted jobs run in identical fashion on any machine and any browser. This ability to handle heterogeneous machines – accentuating the Cost-Benefit Framework's ability to manage heterogeneous resources – played a large part in choosing Java and web technology as the first platform for the Cost-Benefit Framework.

The Java Market offers its machines strong security by way of the Java sandbox. All submitted jobs run inside a browser window. Running within a properly implemented browser, the jobs cannot harm the producer machine – they cannot corrupt its permanent storage, steal sensitive information about the producer machine, or otherwise injure the producer.

Using Java and web technology also brings several hidden costs to the table. First, as a rule, Java runs slowly compared to longer-established languages. Slowed by a web browser and the applet sandbox, it runs at a comparative crawl. The Java Market prototype, like other projects, demonstrates the general feasibility of web-based metacomputing – but the speed of its computations shows the inadequacy of current technology. This is, of course, a temporary problem; many projects in research and industry seek to improve the performance of Java-enabled browser technology.

Second, the Java sandbox imposes strong restrictions on the communication and file I/O submitted jobs can perform. (This is particularly true given that the Java Market was developed for Java 1.0.) Jobs can only communicate with the server that launched them. File I/O on the producer machine is impossible. Therefore, all I/O – and any other form of communication – must go through the server machine, the Java Market itself. This intensive communication load limits the Java Market's scalability. It can only scale to large numbers of machines if the processes are CPU-bound. KnittingFactory [BKKK97] uses a variant approach to web-based metacomputing that could potentially overcome this limitation.

Third, connecting to a web page and uploading job information does not fundamentally resemble compiling and running a job on the local system. An ideal metacomputer is completely transparent – appearing to the user as a local parallel machine – which, by definition, the Java Market is not. One can increase the transparency in various ways; local "compiling" might post jobs in a web-accessible location which the Java Market monitors, for example. The prototype did not implement this functionality.

Mitigating this lack of transparency, web-based metacomputers are *easy to use*. Posting a job to the Java Market, or making one's machine available for work, takes a few minutes at worst. While users must be actively aware of the Java Market's existence, using it is not troublesome.

When we began the Java Market research, we knew that these three obstacles existed, but not whether they would render the product unfeasible. After the prototype, we elected to seek a scalable metacomputing solution offering greater speed and transparency. This system is the Frugal System, described in Chapter Seven.

6.2.2 Features of the Java Market

In addition to machine and operating system heterogeneity, which the Java Market can tolerate by virtue of its basic design, the Market must cope with two other forms of heterogeneity appearing in a distributively owned metacomputing system. The *quality* of various machine resources is heterogeneous – some machines have powerful CPUs, while others have fast network connections. Also, the various machine resources *themselves* are heterogeneous.

Two machine resources have particular relevance to the Java Market: the speed of the machines' CPU and the speed of its connection to the Java Market. The Launch Applet measures these speeds when a machine connects to the Market. This allows the Java Market to rationally consider machines with different power and connectivity.

To manage the heterogeneity between these two resources – the dissimilarity between CPU speed and connectivity – the Java Market converts the impact of assigning a job to any given machine into a unitless cost. Since we could assign only one job to a machine at any given time, we developed a new variant of the opportunity cost approach for this environment. (The prototype did not implement this functionality.)

In this variant of the basic ASSIGN-U algorithm, the scheduler groups producer machines into a single virtual machine. This machine has two resources: messages per second and computations per second (at peak capacity). We measure the cost of assigning a job in terms of the depletion of these pools. Assigning a job to a machine that can perform *cps* computations per second reduces the size of our computation pool by *cps*.

This reformulation of the problem gives us a normalized unit cost for each possible assignment. If necessary, observed trends in what people want to sell their machines for and buy resources for can be used to convert this to real money. We then use the opportunity cost strategy for resource assignment.

The Java Market does not provide Quality of Service guarantees. During its development, we developed the basic framework for a complex decision making strategy. Producers can register as "opportunistic" machines, making themselves available for an unknown period of time, receiving compensation for each second of work, or as "deterministic" providers, selling the Market a fixed block of time for a fixed reward. Consumers could provide a benefit function describing the value of completing the task in any given length of time. Even matching benefit functions to producer guarantees, however, we did not consider the Java Market a viable setting for Quality of Service guarantees. Dedicated producer machines can still crash or suffer a network partition separating them from the Java Market. The Java language, as a deliberate design feature, has limited support for checkpointing. This makes it difficult to protect work against such crashes. The web interface reduces the speed of program completion. These problems are not insurmountable, but attempting to provide hard guarantees would interfere with the Market's primary decision-making function: the search for "profit."

An important concern in metacomputing is *awareness* of the changing state of the system resources. The Java Market performs well in this regard, maintaining continuous contact with all resources. When a resource disconnects or crashes, the Market detects the loss of the resource and removes it from the list of resources. As noted above, it cannot recover lost work. Circumventing the limits of Java and adding a basic checkpointing facility was a subject of ongoing research when the project ended.

As noted above, communication- and I/O-intensive jobs create a high communications overhead on the Market machine. We know from Condor [Con], however, that a single scheduler can scale to manage hundred of machines. In principle, for CPU-bound jobs, the Java Market can do the same.

6.3 Lessons Learned

The first major test of the Java Market was performed in the Johns Hopkins Center for Networking and Distributed Systems (CNDS). Since we had implemented the CPU-intensive simulation described in chapters 3 and 4 entirely in Java, it was a natural choice as our test job. We ran one hundred simulations in the following two ways:

- Running the simulation alone on a Pentium II machine using the Java Developer's Kit, and
- Submitting one hundred copies of the simulation to the Java Market, and, through it, to six producer machines with combined power roughly equal to 4.7x that of the standalone machine.

The completion time for one hundred executions of the simulation on the standalone machine, without compilation or remote I/O, was approximately 127 minutes. Using the Java Market, which had to download and recompile the simulation code each time (attaching it to the Java Market libraries) and perform I/O remotely, we completed all one hundred executions in 35 minutes. This showed a speedup of approximately 3.6, or 76% of the best possible speedup. This is comparable to other results (see **1.5.12: Javelin**) but unsatisfying.

Overall, the Java Market research proved the feasibility of the basic design. With web and Java technology, one can transfer jobs from any machine on the Internet to any other machine on the Internet. Users can take advantage of this technology to achieve significant speedups on CPU-intensive parallelizable jobs. Automatically converting Java code from application to applet format is both possible and quick, so the users need not rewrite their code for the Java Market or similar applications.

In many contexts, bundles of resources – e.g., machines – can only accept one job, regardless of their strength. Studying the Cost-Benefit Framework in the Java Market, we evolved a technique for handling such situations: grouping the machines into a larger conceptual unit, with depletable pools of each of the resources. This technique can prove valuable in adapting the opportunity cost approach to many other specialized environments.

Expecting users to specify job benefit functions, or producers to precisely outlay the benefit they expect for each second of use, is unrealistic in a system designed for simplicity. However, the three-legged hierarchy developed here – with consumer benefit functions, producer cost functions, and the Framework's beneficial theoretical properties mediating between them – has significant potential in future research efforts. With some level of bidding and selling automation, or a software agents structure, this approach could form the backbone of a high-utility metacomputer.

This research demonstrated the value of the Java language and web technology as a basis for a Cost-Benefit Framework metacomputer. These tools offered utility, security, and ease of use. An ideal environment for these algorithms would preserve these beneficial features while adding efficiency and elegance. The search for such an environment led to a study of the new Jini technology from Sun, and thence to the development of the Frugal System: a fully-functional metacomputer for Jini networks capturing most of the best features of the Java Market, yielding high performance, requiring minimal programming effort, and allowing the use of sophisticated programming techniques.

Chapter Seven: The Frugal System

7.1 Description

Remote Method Invocation allows Java objects to call the methods of objects running on remote machines. This technology allows programs to hand off worker objects from one machine to another – the program can transfer the worker object, and all its functionality, by passing the worker as an argument to a remote method. With Remote Method Invocation, objects can run on any machine that the system considers appropriate.

Sun Microsystems' Jini builds on RMI technology to create a new, object-oriented, network paradigm. In Jini, programs running on the network are *services*, *clients*, or both. Clients find the services they need through a central *lookup service*. Because Java RMI defines a world of objects rather than traditional programs, services often subclass standard objects or implement standard interfaces. Clients can find the service they desire by searching, not for a specific class, but for a standard object or interface.

This model has incredible power. It allows two programs to interact seamlessly even when neither program's designers knew that the other program existed. A spreadsheet can find the local statistical analysis package and perform analysis, whether or not the spreadsheet's developers explicitly considered interactions with that statistical package. A system management GUI can automatically add a new graphical component for new system components – even if the relevant peripheral performs a function unknown at the time of the GUI's creation.

In a network built around Jini, the old models of metacomputing do not apply. Java processes do not behave like processes but rather like virtual machines – the flow of work moves from one Java object to another, regardless of physical machines, based on the activation of the services' methods. In this context, one naturally wishes to perform resource allocation within the network of these virtual machines. The physical resources of the system exist only as qualities of the appropriate Java programs.

The Frugal System, developed in this work, builds on Jini technology to create a fully functional metacomputer with strong resource allocation. It includes two major components: Frugal Resources and Frugal Managers.

Frugal Resources, running on different physical machines, perform computational work for clients. A Frugal Resource converts a physical machine into a Jini-enabled Java object with computational power.

A Frugal Manager oversees a collection of Frugal Resources. Frugal Managers use the Differential PVM Strategy to rationally distribute computational resources – Frugal Resources – to clients that need computational work performed. In other words, clients can protect the system against memory overuse and ensure low loads by using a Frugal Manager to help place their computational tasks.

The Frugal System also contains miscellaneous components such as Java Beans that display information about the system state and several utilities for the system's users.

7.1.1 Basic Concepts

Each Frugal Resource *fr* encapsulates a physical machine. It transforms the physical machine into five virtual properties:

- M_{fr}, the total memory allocated to the Frugal Resource;
- m_{fr}, the free memory available to the Frugal Resource;
- S_{fr}, the maximum computation speed seen on this machine;
- s_{fr}, the machine's most recently calculated computation speed; and
- L_{fr}, the maximum load seen on this machine.

In addition, Frugal Resources can perform work and can join *groups* on the various lookup services. Group membership helps define the proper use for the Resource. For example, a Frugal Resource might join the group corresponding to a specific research project if code associated with that project should use that machine.

Each Frugal Manager *fm* acts as a scheduler, implementing the Differential PVM algorithm in the Jini network. It joins a set of groups assigned by an administrator and performs resource allocation within that group.

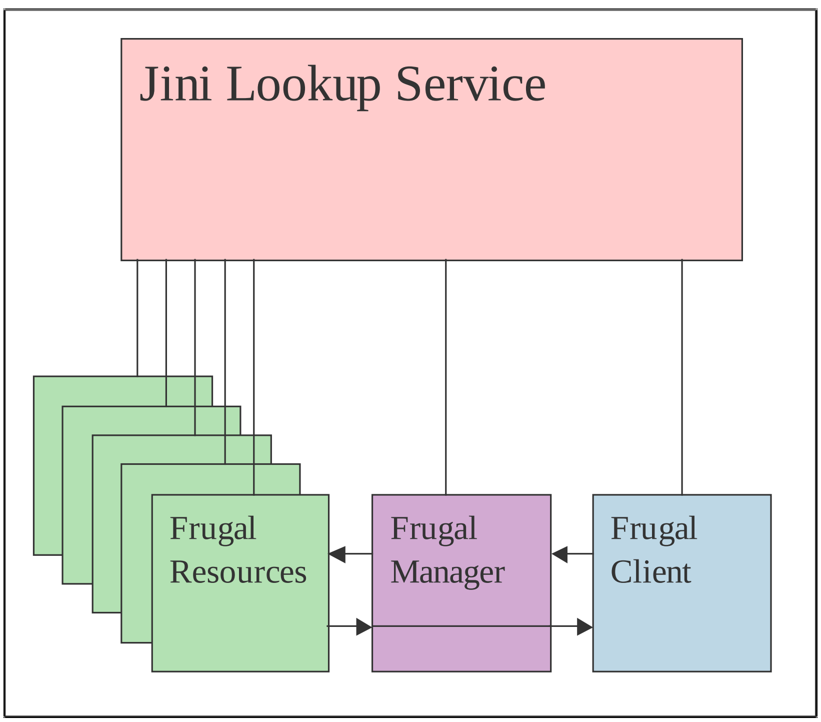

Figure 7.1: Frugal System Layout

Figure 7.1 visually displays the structure of the Frugal System. First, a client discovers that it needs a computational task performed. It contacts the Frugal Manager and asks where to perform this task. The Manager returns a reference to the most appropriate Frugal Resource. The client can then communicate with the resource directly and ask it to perform work.

A full implementation of the Frugal System exists at this time.

7.1.2 Internal Structure

The Frugal System provides a standard interface for Frugal services, and implements two such services: Frugal Resources and Frugal Managers. Frugal implements each service with an administrable *server* and a service *implementation*. The server performs standard management functions for a Jini service: group membership, registration with the lookup service, and so forth. The implementation performs service-specific behavior.

Frugal also contains several miscellaneous components, such as the *Projectable* interface that worker objects must implement.

7.1.3 Frugal Resources

The implementation of Frugal Resources includes the following components:

- The administrable server that maintains the Frugal Resource;
- A Frugal Resource object that can perform work for clients;
- Lookup properties that display information about the resource; and
- A load testing program that keeps track of the system's effective speed.

The server and Resource object are both straightforward in function. Note that Frugal Resources do not need to implement the Differential PVM Strategy, as the Frugal Managers handle this. They do not need to implement security directly, as a standard Java policy file provides customizable security.

Lookup properties associated with the Resource describe its permanent ID as well as the five Resource properties detailed earlier: total memory, free memory, current speed, maximum speed, and the maximum load. In addition, the Resource maintains information on the global maximum load. It does not need this information to function, but can use this information to advertise its cost.

The Java environment does not give Frugal Resources direct access to the total memory and free memory on the system. Instead, each virtual machine receives a heap of memory with an initial size and maximum size determined by the user. Querying the system's memory from within Java returns the size of this heap and the amount of free memory in this heap. To avoid dependence on system-dependent native code, Frugal Resources treat the allocated memory as the system memory. Thus, the Differential PVM Strategy does not protect Frugal Resources directly against thrashing, but rather prevents memory exhaustion. Choosing the virtual machine heap size carefully, users can use this property to keep their machines from thrashing.

The Frugal Resource's load tester does not directly measure system load, as accessing maintained load information is also a highly system-dependent activity. Since Frugal Resources abstract away the physical machine, we consider it important to make them executable on any machine. Thus, Frugal Resources measure the system's "load" by regularly evaluating its speed of computation. The maximum computation speed yet seen divided by the current computation speed is the system's load. Thus, if we have seen a computation speed of 3 million computations per second, but the system only performed half a million computations per second when last tested, we estimate the load as 6.0.

7.1.4 Frugal Managers

The implementation of Frugal Managers includes the following components:

- The administrable server that maintains the Frugal Manager;
- A Frugal Manager object that implements the Differential PVM Strategy; and
- A Frugal Manager Ears object that monitors Frugal Resources on the system.

As with Frugal Resource servers, Frugal Manager servers function in a straightforward manner. A standard administrative interface allows system administrators to choose which groups on which lookup services a given Frugal Manager monitors. The server then maintains a registration for a Frugal Manager object in those groups on those lookup services.

The Frugal Manager object performs scheduling for its clients. When a client consults a Frugal Manager, the Manager applies the Differential PVM Strategy to the group of Frugal Resources it administers. It treats the number of Resources *currently* administered as the number of machines "n". It scans them for the smallest and largest calculation speeds seen so far on any of the machines. This determines the maximum load seen by the system, as well as the ratio of the speeds of the different machines. From this information, it can calculate the Differential PVM cost function.

The "Frugal Manager Ears" object maintains an up-to-date list of all the Frugal Resources that a given Frugal Manager can administrate. It responds appropriately when new Frugal Resources enter the system or old Frugal Resources leave the system. For these purposes, a Frugal Resource "leaves" the system when it fails to regularly renew its registration on the lookup service.

7.1.5 Miscellaneous Components

The Frugal System also contains three miscellaneous components:

- A standard interface for work objects submitted to the Frugal System;
- A standard administrative object for managing the interface between the Frugal System and the Jini lookup service; and
- A Frugal Ears object for clients that monitors Frugal Managers on the system.

Worker objects that clients wish to allocate using the Frugal System must implement the *Projectable* interface. This interface specifies the existence of a “main” method that takes an array of objects as arguments and returns an array of objects as its result.

The administrative object performs standard functions. It allows users to customize the groups and lookup services that a Frugal service connects to, the “Entry” objects that identify them to other services, and so forth.

The “Frugal Ears” object maintains an up-to-date list of all the Frugal Managers that a given client can find. It responds appropriately when a new Frugal Manager enters the system or an established Frugal Manager leaves the system.

7.1.6 Class Structure

Figure 7.2 depicts the general Frugal Service class hierarchy. Grey boxes are Frugal System code. Boxes with a fancy border belong to the Frugal Manager component of the system. Shadowed boxes belong to the Frugal Resource component of the system. White boxes are part of the Jini connection technology or a client’s code.

Figure 7.3 shows the class structure for Frugal Resources and Frugal Managers. Note that clients use only the *interfaces* for Frugal Resources and Frugal Managers. This means that the underlying Frugal System code can add new functionality without changing the behavior of established Frugal System clients.

Figure 7.4 shows the class structure for Frugal Resource lookup properties. Finally, 7.5 shows miscellaneous Frugal classes and the non-Frugal classes they depend upon.

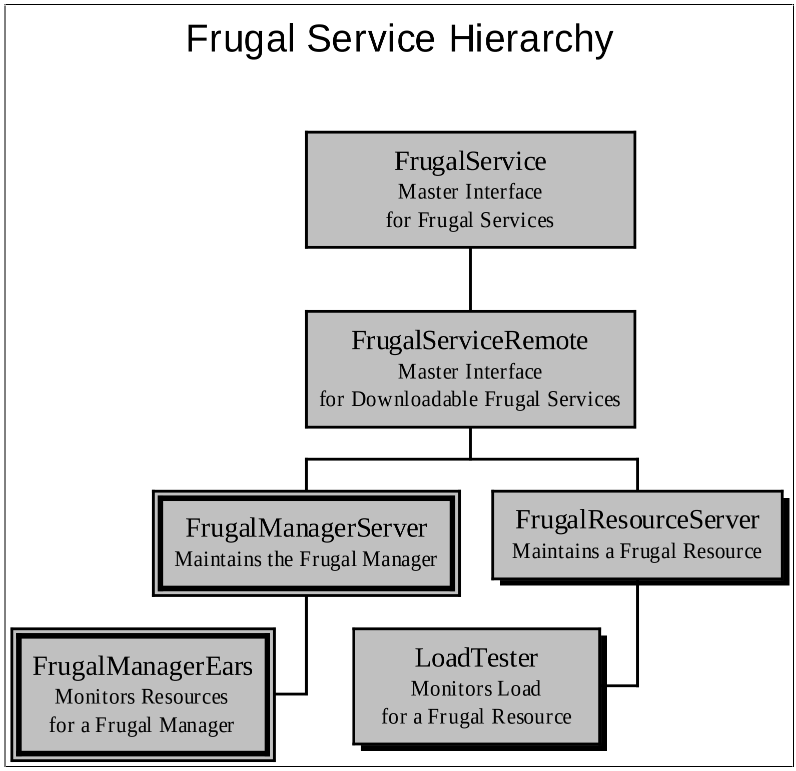

Figure 7.2: Frugal Object Hierarchy (Part 1)

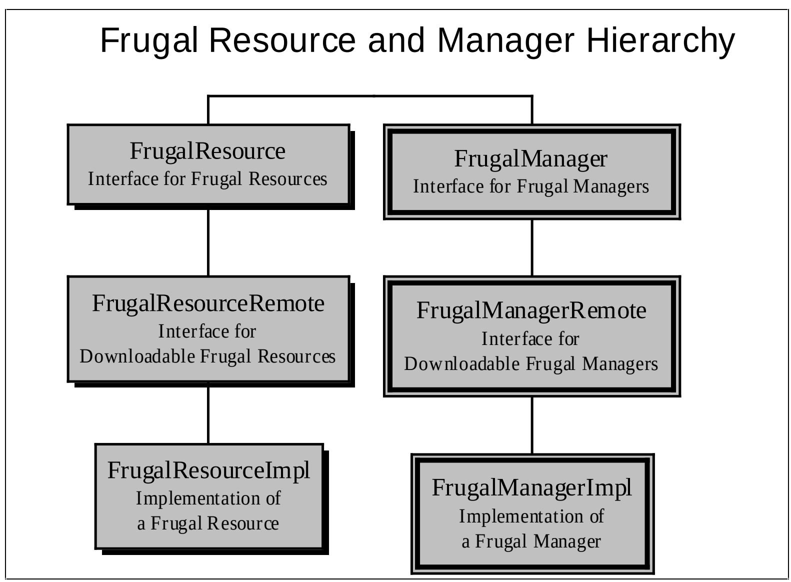

Figure 7.3: Frugal Object Hierarchy (Part 2)

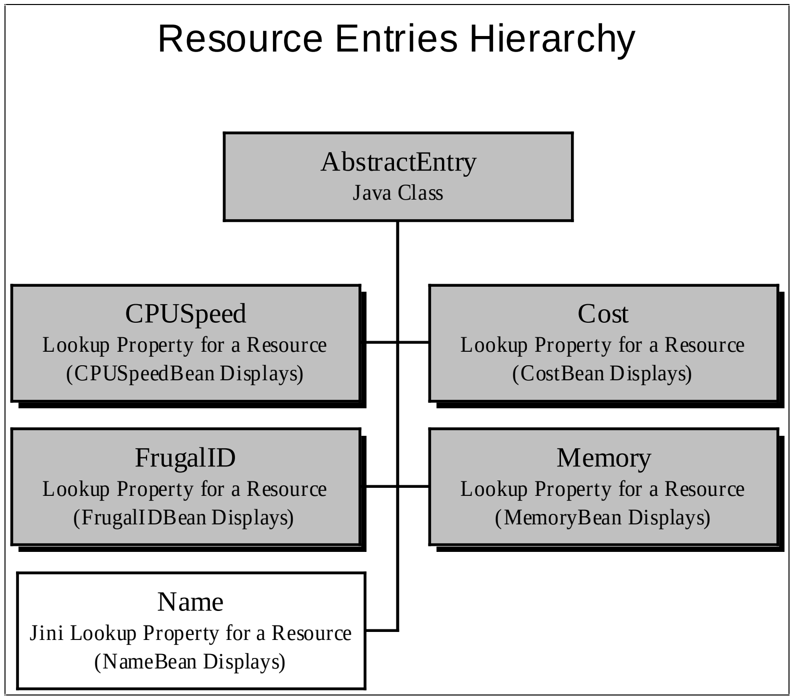

Figure 7.4: Frugal Object Hierarchy (Part 3)

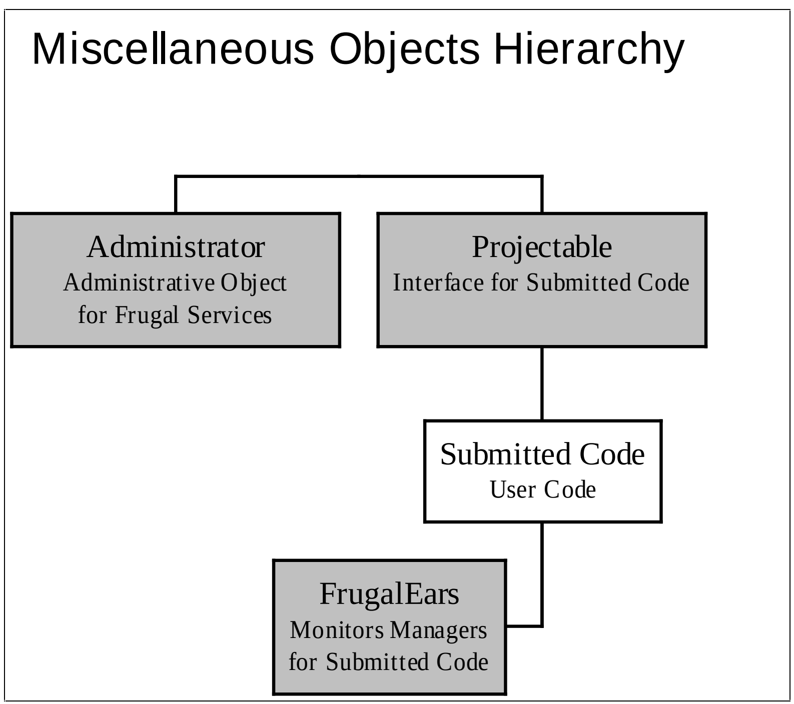

Figure 7.5: Frugal Object Hierarchy (Part 4)

7.2 Frugal System Design

7.2.1 Features of Jini-Based Metacomputers

The Frugal System builds on Jini connection technology. This also gives the project certain inherent advantages and disadvantages.

The Frugal System, like the Java Market, uses pure Java code. The Frugal System does not depend on any machine architecture or operating system. Any machine with a Jini environment and the ability to interpret Java bytecodes can take advantage of the Frugal System.

The security model used by modern Java environments, outside of web browsers, possesses considerable sophistication and power. Administrators can tailor the security restrictions on the Frugal System and on individual clients thereof to suit their needs. Clients cannot use the Frugal System to affect, modify, or damage the machines on which the Frugal Resources run unless the system policy file gives them specific authorization.

Jini connection technology runs with reasonable efficiency. The performance issues that plague browser-based programs manifest to a much reduced extent. It could use approximately 91% of the real resources of the test cluster, as opposed to the Java Market's 75%, as described below.

Except when mandated by the security policy file, Jini imposes no restrictions on the communication and file I/O performed by submitted jobs. The Frugal Manager does not need to act as an intermediary for file and network I/O operations. It acts only as a scheduler. Therefore, we expect it to scale to manage hundreds of Frugal Resources simultaneously. For very large clusters, installing multiple Frugal Managers to balance different sets of Resources is very simple.

Jini technology allows a great deal of transparency. Jini programs often forward objects to other machines, and in many cases, those objects perform work on the remote machine. Thus, clients use Frugal services just as they use any other service. To use an ordinary Java program with the Frugal System requires small modifications to the code. These changes fit the standard Jini programming model and are not conceptually unique to the Frugal System.

One obstacle does appear when translating the Cost-Benefit Framework into the Jini context. As a deliberate design decision, Java virtual machines do not allow direct access to the resources of the underlying machine. Without using native code, and sacrificing the Frugal System's machine independence, we could not access load or memory information about the underlying system directly. The Frugal System protects the memory assigned to its *virtual* machines from exhaustion, rather than directly protecting the physical machines' memory from paging. (Naturally, the two tasks interrelate.) It determines system load and speed by regularly testing the system's computational speed; between tests, load information is out of date.

7.2.2 Features of the Frugal System

The Frugal System reacts dynamically to changing cluster configurations. When the set of running and accessible Frugal Resources changes, the Frugal Manager adjusts its list of resources, machine count, and global maximum load automatically. The cost calculations used in the Differential PVM Strategy draw on information about the currently active cluster.

Combining the Frugal System with various Jini-related technologies (such as JavaSpaces), one can implement advanced metacomputing techniques such as Charlotte's eager scheduling. A properly designed program can easily implement checkpointing and limited migration.

The Frugal System makes efficient use of its resources.

7.3 Lessons Learned

We performed the first major test of the Frugal System at the Johns Hopkins Center for Networking and Distributed Systems (CNDS). We used the simulation described in chapters 3 through 6 as our test job. We ran three thousand executions of this simulation in the following two ways:

- Running the simulation alone on a Pentium II machine using the Java Developer's Kit, and
- Distributing 3000 copies of the simulation to the Frugal System, and, through it, to five machines with combined power roughly equal to 3.56x that of the standalone machine. We executed 60 copies of the simulation simultaneously at any one time. Each machine performed two simulations and then returned their results. The first 60 assignments arrived over the course of a minute. (We chose these parameters to make good use of the Frugal System, but did not seek an optimal configuration.)

The standalone machine completed 3000 simulations in 22,533,157ms, or approximately six hours and sixteen minutes. A purely random assignment of simulation instances to machines in the cluster resulted in a completion time of 15,191,534 ms, for a speedup of 1.48x – only 42% of the best possible speedup. The Frugal System, on the other hand, completed 3000 simulations in 6,938,848ms, slightly under two hours. This shows a speedup of 3.24x, or 91% of the best possible speedup. The missing 9% results from the following factors:

- An imbalance in the memory of the machines. Due to differences in physical memory and running system processes, the virtual machines received memory assignments of 32, 32, 35, 64, and 85 Megabytes of memory.
- The update time for CPU load. A machine's awareness of its CPU load updates every 30 seconds. This reduces the ability of the system to load balance properly. With a task completing every 2-3 seconds, the update time gave the system a significant handicap.

- The non-negligible overhead for resource discovery. It took between 73ms and 27 seconds to perform resource discovery and selection, with a median time of 632ms and an average time of 1625ms. This was the time taken to perform resource selection for two executions of the simulation, which lasted an average of 3001ms longer. While significant, this is not a 50% overhead – with 60 executions running simultaneously, the periods of resource discovery overlapped. The Frugal Manager handled approximately 20 requests at any given time, so this represents a resource selection overhead of 1.8%.

Overall, the Frugal System demonstrates the feasibility of implementing the ideas of the Cost-Benefit Framework in a real metacomputing system. It contains a complete and functional implementation of the Cost-Benefit Framework's Differential PVM Strategy, and can make its decisions quickly.

In adapting the Cost-Benefit Framework to Jini networks, we advanced the Cost-Benefit Framework's technology in several key respects. First, we observed a weakness in Enhanced PVM's model. Enhanced PVM assumes that jobs advertise their CPU and memory requirements, which is not always the case. This observation led directly to the creation of the Differential PVM Strategy, as described in Chapter Five.

Second, in creating the Frugal System, we confronted the issue of variable system membership. Machines join a Frugal Manager's resource pool when an administrator starts a Frugal Resource. Machines leave the resource pool when an administrator kills the Frugal Resource, removes it from the relevant groups on a lookup service, or the machine crashes. The set of machines that the Frugal Manager manages may change completely from one day to the next.

Accordingly, the Frugal Manager "outsources" the computation of the system's calculated maximum CPU load. That maximum becomes the smallest integral power of 2 greater than the highest load any of the individual Resources currently accessible has seen. The Manager itself does not maintain maximum load information.

Chapter Eight: Conclusions

One can reasonably think of a modern computing cluster as a single conceptual machine – a metacomputer – that possesses the computational power of all its component machines. Such metacomputers are often more powerful than strong supercomputers. At the same time, individual serial or parallel tasks submitted to the metacomputer must run on specific hardware. To use the cluster to its fullest advantage, the system's scheduler must appropriately divide its workload among the cluster's various physical machines.

Standard approaches to resource management divide into theoretically sound approaches, which consider CPU load or memory load, and heuristic approaches, which consider multiple resources. This work develops a theoretically sound approach to resource allocation in metacomputers that considers multiple resources and displays good performance in practice.

We presented the Enhanced PVM Strategy for resource allocation. This strategy makes permanent job assignments. It is theoretically competitive with the optimal prescient strategy in terms of maximum resource utilization, assuming permanent jobs. Its competitive ratio is O(log n). Experiments demonstrate that this strategy can complete jobs in approximately 62% of the time that a naïve strategy requires. This captures 66-70% of the benefit of employing the highly tuned Mosix strategy on a kernel enhanced to allow temporary job assignments.

We presented the Enhanced Mosix Strategy for resource allocation. This strategy makes temporary job assignments. It both assigns and reassigns jobs. It does not preserve Enhanced PVM's competitive bound. It does, however, share many of the performance properties of the Enhanced PVM Strategy. Simulations indicate that it can complete jobs in approximately 87.5% of the time they would require using the Mosix Strategy for assignment and reassignment.

We presented the Differential PVM Strategy. This strategy adapts the Enhanced PVM Strategy for environments where the system does not have sufficient information about jobs entering the system. It captures approximately 91% of the benefit of Enhanced PVM without knowing anything about jobs until after it places them permanently on a machine.

We implemented these strategies in two metacomputing test beds. The first test bed, called the Java Market, conceptually converted the entire Internet into a metacomputer. Users anywhere on the Internet could submit jobs to the Java Market or make their machine available for use by the Market, simply by connecting to a web page. We built a functional prototype of the Java Market, able to accept and place jobs on contributing machines.

The second test bed, called the Frugal System, enhances Jini networks with metacomputing capability. We have completed a full version of this system, using the Differential PVM Strategy to place jobs. This system allows Java objects to perform work on any machine in the computing cluster, and moreover allows programs to choose the "best" machine to perform their work on. Using the Frugal System helps to ensure low CPU loads and, by conserving the memory assigned to the system's virtual machines, prevent thrashing or memory exhaustion.

In sum, this work introduces a new approach to resource allocation in metacomputers with beneficial theoretical properties. It shows that using this approach produces excellent results in practice. It implements this work in a new, functional, real-world metacomputing platform to demonstrate the feasibility of implementing it in practice.

References

[AAB00] **A Cost-Benefit Framework for Online Management of a Metacomputing System,** Y. Amir, B. Awerbuch, R.S. Borgstrom, The International Journal for Decision Support Systems, Elsevier Science, 28 (2000): 155-164

[AAFPW97] **On-Line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling,** J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, O. Waarts, Journal of the ACM, 44(3): 486-504, May 1997

[AAP93] **Throughput-competitive on-line routing,** B. Awerbuch, Y. Azar, S. Plotkin, 34th IEEE Symposium on Foundations of Computer Science, 1993

[AAPW94] **Competitive Routing of Virtual Circuits with Unknown Duration,** B. Awerbuch, Y. Azar, S. Plotkin, O. Waarts, ACM-SIAM Symposium on Discrete Algorithms (SODA), 1994

[ABG00] **Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid,** D. Abramson, R. Buyya, J. Giddy, HPC Asia 2000 (to appear)

[APMOW97] **Dynamic Load Distribution in MIST**, K. Al-Saqabi, R. Prouty, D. McNamee, S. Otto, J. Walpole, the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications, June/July 1997

[AAPW94] **Competitive Routing of Virtual Circuits with Unknown Duration,** ACM-SIAM Symposium on Discrete Algorithms (SODA), 1994.

[BBB96] **Atlas: An Infrastructure for Global Computing**, J.E. Baldeschwieler, R.D. Blumofe, E.A. Brewer, the 7th ACM SIGOPS European Workshop: Systems Support for Worldwide Applications, September 1996.

[BFKV92] **New Algorithms for an Ancient Scheduling Problem,** Y. Bartal, A. Fiat, H. Karloff, R. Vohra, the ACM Symposium on Theory of Algorithms, 1992

[BHJM99] **Structural Biology Metaphors Applied to the Design of a Distributed Object System**, L. Bölöni, R. Hao, K. Jun, and D. Marinescu, the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, April 1999.

[BKKK97] **KnittingFactory: An Infrastructure for Distributed Web Applications,** A. Baratloo, M. Karaul, H. Karl, Z.M. Kedem, TR 1997-748, Department of Computer Science, New York University

[BL98] **The MOSIX Multicomputer Operating System for High Performance Cluster Computing,** A. Barak, O. La'adan, Journal of Future Generation Computer Systems, 13(4-5): 361-372, March 1998.

[BW97] **The AppLeS Project: A Status Report,** F. Berman, R. Wolski, the 8th NEC Research Symposium, May 1997.

[Caly] http://www.cs.nyu.edu/milan/calypso/index.html

[CCINSW97] **Javelin: Internet-Based Parallel Computing using Java**, P. Cappello, B. Christiansen, M. Ionescu, M.O. Neary, K.E. Schauser, D. Wu, 1997 ACM Workshop on Java for Science and Engineering Computation, June 1997.

[Clear96] **Market-Based Control: A Paradigm for Distributed Resource Allocation,** edited by S.H. Clearwater, World Scientific.

[CPBD00] **Deploying Fault-tolerance and Task Migration with NetSolve**, H. Casanova, J. Plank, M. Beck, J. Dongarra, The International Journal on Future Generation Computer Systems, to appear.

[Con] http://www.cs.wisc.edu/condor/

[Fer98] **JPVM: Network Parallel Computing in Java,** A.J. Ferrari, ACM 1998 Workshop on Java for High-Performance Network Computing

[FDG97] **Scalable Networked Information Processing Environment (SNIPE),** G.E. Fagg, J.J. Dongarra, A. Geist, SC97: High Performance Networking and Computing, November 1997

[Fin93] **Specification of the KQML Agent Communication Language,** T. Finin *et al.*, DARPA Knowledge Sharing Initiative Draft, June 1993.

[Glo] http://www.globus.org/

[GS99] **Metacomputing with the IceT System,** P. A. Gray, V.S. Sunderam, the International Journal of High Performance Computing Applications, 13(3): 241-252, 1999.

[GWLt97] **The Legion Vision of a Worldwide Virtual Computer,** A.S. Grimshaw, W.A. Wulf, the Legion Team, Communications of the ACM 40(1), January 1997.

[Har] http://www.epm.ornl.gov/harness/

[HD96] **Exploiting Process Lifetime Distributions for Dynamic Load Balancing,** M. Harchol-Balter, A. Downey, ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, 1996.

[Inf] http://www.infospheres.caltech.edu/

[IK97] **Globus: A Metacomputing Infrastructure Toolkit,** I. Foster, C. Kesselman, International Journal of Supercomputer Applications, 11(2): 115-128, 1997.

[Java] http://java.sun.com/

[JavSp] http://java.sun.com/products/javaspaces/index.html

[Jin] http://www.sun.com/jini/

[Kar98] **Metacomputing and Resource Allocation on the World Wide Web,** Mehmet Karaul, doctoral thesis, Department of Computer Science, New York University, May 1998.

[KC93] **CC++: A declarative concurrent object oriented programming notation,** K.M. Chandy, C. Kesselman, Research Directions in Object Oriented Programming, The MIT Press, 1993, 281-313

[Leg] http://www.cs.virginia.edu/~legion/

[Lin] http://www.sca.com/linda.html

[LLM88] **Condor – A Hunter of Idle Workstations,** M. Litzkow, M. Livny, M.W. Mutka, the 8th International Conference of Distributed Computing Systems, June 1988.

[MBHJ98] **An Alternative Model for Scheduling on a Computational Grid**, D. Marinescu, L. Bölöni, R. Hao, and K. Jun, the 13th International Symposium on Computer and Information Sciences, IOP Press, 1998.

[MG97] **Towards Portable Message Passing in Java: Binding MPI**, S. Mintchev, V. Getov, EuroPVM-MPI, 1997

[Mos] http://www.mosix.org

[Pir] http://www.sca.com/piranha.html

[PVM] http://www.epm.ornl.gov/pvm/pvm_home.html

[MPI] http://www.mpi-forum.org/

[NetW] http://nws.npaci.edu/NWS/

[RRFH96] **A Task Migration Implementation for MPI**, J. Robinson, S. Russ, B. Flachs, B. Heckel, the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC-5), August 1996.

[Sar98] **Bayanihan: Web-Based Volunteer Computing using Java,** Luis F.G. Sarmenta, the 2nd International Conference on World-Wide Computing and its Applications (WWCA '98).

[SPS97] **The Design of JET: A Java Library for Embarrassingly Parallel Applications,** L.M. Silva, H. Pedroso, J.G. Silva, WOTUG'20 – Parallel Programming and Java Conference, April 1997.

[WHHKS92] **Spawn: A distributed computational economy**, C.A. Waldspurger, T. Hogg, B.A. Huberman, J.O. Kephart, W.S. Stornetta, IEEE Transactions on Software Engineering, 18(2): 103-117, February 1992

[WW98] **Market-aware agents for a multiagent world,** M.P. Wellman, P.R. Wurman, Robotics and Autonomous Systems, 1998.